A Practical Guide to Parallelization in Economics*

Jesús Fernández-Villaverde† David Zarruk Valencia‡

October 9, 2018

Abstract

This guide provides a practical introduction to parallel computing in economics. After a brief introduction to the basic ideas of parallelization, we show how to parallelize a prototypical application in economics using, on `CPUs`, `Julia`, `Matlab`, `R`, `Python`, `C++-OpenMP`, `Rcpp-OpenMP`, and `C++-MPI`, and, on `GPUs`, `CUDA` and `OpenACC`. We provide code that the user can download and fork, present comparative results, and explain the strengths and weaknesses of each approach. We conclude with some additional remarks about alternative approaches.

Key words: Computational Methods, Parallel Computing, Programming Languages.
JEL classifications: C63, C68, E37.

*The pdf file of the paper includes several embedded links of interest. The Github repository of the code used in the paper is: https://github.com/davidzarruk/Parallel_Computing. We thank many cohorts of students at the University of Pennsylvania who have endured earlier drafts of the slides behind this guide and pointed out ways to improve them. We have also received useful feedback from David Hsieh, Galo Nuño, Jim Nason, and Carlo Pizzinelli.

†University of Pennsylvania. Email: jesusfv@econ.upenn.edu
‡University of Pennsylvania. Email: davidza@sas.upenn.edu

Contents

1 Introduction

Economists, more than ever, need high-performance computing. In macroeconomics, we want to solve models with complex constraints and heterogeneous agents to simultaneously match micro and aggregate observations. In industrial organization, we aim to characterize the equilibrium dynamics of industries with several players and multiple state variables. In asset pricing, we seek to price complex assets in rich environments with numerous state variables. In corporate finance, we like to track the behavior of firms with rich balance sheets and intertemporal choices of funding. In international economics, we are interested in analyzing multisectoral models of trade. In econometrics, we have to evaluate and simulate from moment conditions and likelihood functions that often fail to have closed-form solutions. And machine learning and big data are becoming ubiquitous in the field.

One of the most important strategies to achieve the required performance to solve and estimate the models cited above is to take advantage of parallel computing. Nowadays, even basic laptops come from the factory with multiple cores (physical or virtual), either in one central processing unit (CPU) or several CPUs. And nearly all of them come with a graphics processing unit (GPU) that can be employed for numerical computations. Many departments of economics and most universities have large servers that researchers can operate (a few even have supercomputer centers that welcome economists). Furthermore, cloud computing services, such as `Amazon Elastic Compute Cloud` or `Google Cloud Compute Engine`, offer, for economical prices, access to large servers with dozens of CPUs and GPUs, effectively making massively parallel programming available to all academic economists.

Unfortunately, there are barriers to engaging in parallel computing in economics. Most books that deal with parallelization are aimed at natural scientists and engineers. The examples presented and the recommendations outlined, valuable as they are in those areas, are sometimes hard to translate into applications in economics. And graduate students in economics have taken rarely many courses in programming and computer science.

Over the years, we have taught parallel programming to many cohorts of young economists. Thus, it has become clear to us that an introductory guide to parallelization in economics would be useful both as a basis for our lectures and as a resource for students and junior researchers at other institutions.

In this guide, we discuss, first, why much of modern scientific computing is done in parallel (Section 2). Then, we move on to explain what parallel programming is with two simple examples (Section 3). And while parallel programming is a fantastic way to improve the performance of solving many problems, it is not suitable for all applications. We explain this

point in Section 4 with two fundamental problems in economics: a value function iteration and a Markov chain Monte Carlo. While the former is perfectly gathered for parallelization, the latter is much more challenging to parallelize. That is why, in Section 5, we introduce a life-cycle model as our testbed for the rest of the paper. The life-cycle model is sufficiently rich as to be an approximation to problems that a researcher would like to solve in "real life." At the same time, parallelization of the model is not trivial. While we can parallelize its solution over the assets and productivity variables, we cannot parallelize over the age of the individual. Section 6, then, briefly introduces the different types of computers that a typical economist might encounter in her parallel computations. We spend some time explaining what a cloud service is and how to take advantage of it.

Sections 7 and 8 are the core of the paper. In Section 7, we outline the main approaches to parallelization in CPUs, with `Julia`, `Matlab`, `R`, `Python`, `C++-OpenMP`, `Rcpp-OpenMP`, and `C++-MPI`. In Section 8, we present `CUDA` and `OpenACC`. In these sections, we discuss the strengths and weaknesses of each approach while we compute the same life-cycle model that we presented in Section 5. We do not envision that a reader will study through each part of these two sections in full detail. A reader can, for instance, pick those approaches that she finds more attractive and study them thoroughly while only browsing the other ones (although we strongly recommend reading Subsection 7.1, as it presents several key ideas that we will repeat for all programming languages). Regardless of the situation, the reader should follow our explanations while checking the code we have posted at the Github repository of this paper and forking it with her improvements. Learning computation without trying it oneself is nearly impossible, and in the case of parallelization, even more so. We cannot stress this point enough.

We conclude the guide with a mention in Section 9 of existing alternatives for parallel computation beyond CPUs and GPUs (such as manycore processors, field-programmable gate arrays, and tensor processor units) and with some brief remarks summarizing our computational results in Section 10.

Before we get into the main body of the guide, we must highlight three points. First, this guide's goal is pedagogical. Our codes were written putting clarity ahead of performance as a basic criterion. We did not optimize them to each language or employe advanced coding styles. Instead, we compute the same model in the same way across languages and compare the performance of parallel computing, using as the benchmark the serial computation for each language. Most students in economics do not come to the field from computer science or engineering, and they prefer clearer codes to sophisticated ones. This means that, for example, we will employ too few anonymous functions, too many loops, and too much old-

style imperative programming (also vectorization and related techniques work less well in the problems we deal with in economics than in other fields; see Aruoba and Fernández-Villaverde, 2015). While there is much to be said about improving coding style among economists, that is a battle for another day. You teach one lesson at a time.

Second, we do not cover all possible combinations of languages, parallelization techniques, and computers. Doing so would be next to impossible and, in any case, not particularly useful. We are, instead, highly selective. There is no point, for instance, in covering parallelization in `Java` or `Swift`. Neither of these languages is designed for scientific computation, nor are they widely used by economists. Among programming languages, perhaps the only absence that some readers might miss is `Fortran`. In previous related exercises (Aruoba and Fernández-Villaverde, 2015), we kept `Fortran` more as a nod to old times than out of conviction. But, by now, we feel justified in dropping it. Not only does `Fortran` not deliver any speed advantage with respect to `C++`, but it also perpetuates approaches to numerical computation that are increasingly obsolete in a world of massive parallelization. To program well in parallel, for instance, one needs to understand `MapReduce` and employ rich data structures. `Fortran` fail to meet those requirements. While keeping legacy code and libraries may require knowledge of `Fortran` for some researchers, there is no point in young students learning what is by now an outdated language despite all the efforts to modernize it in the most recent standards (such as the soon-to-be-released `Fortran 2018`). Among programming techniques, we skip `POSIX` and `Boost threads` and `threading building blocks` (too low level for most users) and `OpenCL` (as we believe that `OpenACC` is a better way to go).

Perhaps the only topic of importance that we miss is the use of `Spark` (possibly with `Scala`), a general-purpose cluster computing system particularly well-adapted for the manipulation of large data sets.[1] Future editions of this guide might correct that shortcoming, but the current version is sufficiently long, and we are afraid of losing focus by introducing a whole new set of issues related to data parallelization.

Third, we will not explain how to write algorithms in parallel since this would require a book in itself (although we will cite several relevant books as we move along as references for the interested reader). This lack of explanation implies that we will not enter into a taxonomy of computer architectures and parallel programming models (such as the celebrated Flynn's classification) beyond passing references to the differences between parallelism in tasks and parallelism in data or the comparison between the shared memory of `OpenMP` and the message passing structure of `MPI`.

With these three considerations in mind, we can enter into the main body of the guide.

[1] See https://spark.apache.org/.

2 Why Parallel?

In 1965, Gordon Moore, the co-founder of Intel, enunciated the most famous law of the computer world: the number of components (i.e., transistors) per integrated circuit (i.e., per "chip") will double each year (Moore, 1965). Although this number was later downgraded by Moore himself to a *mere* doubling every two years (Moore, 1975), this path of exponential growth in the quantity of transistor chips incorporated is nothing short of astonishing. Moore's law predicts that the computational capability of human-made machines will advance as much during the next 24 months as it has done from the dawn of mechanical devices until today.

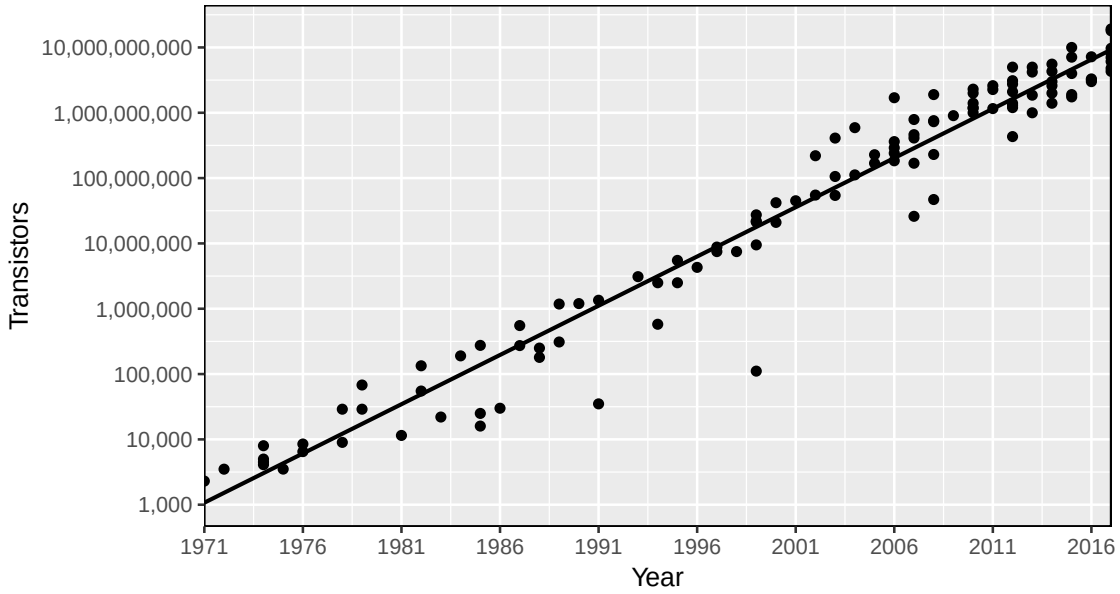


Figure 1: Number of transistors

But Moore's law has been nearly as breathtaking in terms of its forecasting success. Figure 1 plots how the number of transistors has evolved from 1971 to 2017.[2] Moore's law predicted that the representative chip would have 8,388,608 times more transistors by 2017 than in 1971. Indeed, the 2017 `32-core AMD Epyc` processor has 8,347,800 times more transistors than the 1971 `Intel 4004` processor. If you look, however, at the end of the sample, there are some indications that the law may be slowing down to a doubling of transistor count every 2.5 years: the `32-core AMD Epyc` processor has many more transistors than other processors of its generation and it achieves that density because it suffers from higher cache latency (see, for more quantitative evidence of the slowdown of Moore's law, Flamm, 2018).

[2]Data from Figure 1 come from https://en.wikipedia.org/wiki/Transistor_count.

The recent hints of a slowdown in the rate of transistor growth might be telltale signs of future developments. Although the demise of Moore's law has been announced in the past and technology has kept on surprising skeptics, there are reasons to believe that we are approaching the maximum number of transistors in a chip, perhaps in a decade or two. Not only does the electricity consumption of a chip go up by x^4 when the transistor size falls by a factor x, but we must endure considerable increases in the heat generated by the chip and in its manufacturing costs.

At a more fundamental level, there are inherent limits on serial chips imposed by the speed of light (30 cm/ns) and the transmission limit of copper wire (9 cm/ns). This means that there are hard constraints on the speed at which a processor can compute an operation and that it is virtually impossible to build a serial Teraflop machine without resorting to parallelization.[3] Even more relevant is the observation that the real bottleneck for scientific computing is often memory access. Random Access Memory (RAM) latency has been only improving around 10 percent a year.

Figure 2: Cray-1, 1975

That is why the industry has moved from building machines such as the serial `Cray-1`, in 1975 (Figure 2), to the massively parallel `Sunway TaihuLight` (Figure **??**), the fastest

[3] A Teraflop is 10^{12} floating point operations per second (i.e., the number of calculations involving real numbers). As a point of comparison, the `Microsoft Xbox One X` has a top theoretical performance of 6 Teraflops thanks to its aggressive reliance on GPUs.

supercomputer in the world as of April 2018.[4] The `Cray-1`, built around the idea of vector processing, aimed to deliver high performance through the simultaneous operations on one-dimensional arrays (vectors, in more traditional mathematical language).[5] Although the `Cray-1` represented a leap over existing machines, it required a specific arranging of the memory and other components (the circular benches around the main machine that can be seen in Figure 2 were essential for the power supply to be delivered at the right place) as well as quite specialized software. These specificities meant that the cost curve could not be bent sufficiently quickly and that the limitations of single processor machines could not be overcome.

Figure 3: IBM Summit, 2018

The solution to these limits is employing more processors. This is the route followed by the `IBM Summit`. The computer is built around 9,216 manycore processors, each with 22 cores, and 27,648 Nvidia Tesla V100 GPUs. Instead of a single piece of machinery like the `Cray-1`, Figure 3 shows rows of cabinets with racks of processors, each handling a part of some computation. Thanks to this parallelization, the 160 Megaflops (10^6) `Cray-1` has been replaced by the 200 Petaflop (10^{15}) `summit`, an increase in speed of around $1.25 * 10^9$. Without these speed gains, everyday computational jobs on which we have grown to depend –such as high-definition video rending, machine learning, search engines, or the tasks behind `Spotify`, `Facebook`, and blockchain– would not be feasible.

[4] Link to Wikipedia image of Figure 2 and link to Oak Ridge National Laboratory image of Figure 3.

[5] This computer was named after Seymour Cray (1925-1996), whose pioneering vision created modern high-performance computing (see Murray, 1997). The company he founded to manufacture this machine, `Cray Research`, led the industry for decades. Modern-day `Cray Inc.` descendes from the original Cray Research after several turbulent changes in ownership. An early example of a parallel computer was the pathbreaking `CDC 6600`, released in 1964.

3 What Is Parallel Programming?

The main idea behind parallel programming is deceptively simple: if we have access to several processors (either in the same computer or in networked computers), we can divide a complex problem into easier pieces and send each of these components to a different processor. Within the field of high-performance computing, this division can be done either for numerical analysis –for example, to multiply two matrices– or for the handling of large amounts of data –for instance, the computation of a sample average through `MapReduce`.

Let us develop these two examples to understand the basic idea. In our first example, we have two matrices:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

and

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

and we want to find:

$$C = A \times B = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix}.$$

Imagine, as well, that we have access to a machine with five processors. We can select one of these processors as a "master" and the other four as "workers" (sometimes, and less delicately, also called "slaves"). The master processor will send $(a_{11}, a_{12}, b_{11}, b_{21})$ to the first worker and ask it to compute $a_{11}b_{11} + a_{12}b_{21}$. Similarly, it will send $(a_{11}, a_{12}, b_{12}, b_{22})$ to the second worker and ask it to compute $a_{11}b_{12} + a_{12}b_{22}$ and so on with the third and fourth workers. Then, the results of the four computations will be returned to the master, which will put all of them together to generate C. The advantage of this approach is that while the first worker is computing $a_{11}b_{11} + a_{12}b_{21}$, the second worker can, at the same time, find $a_{11}b_{12} + a_{12}b_{22}$. In comparison, a serial code in one processor will need to finish the computation of $a_{11}b_{11} + a_{12}b_{21}$ before moving into computing $a_{11}b_{12} + a_{12}b_{22}$.

Doing a parallelization to multiply two matrices 2×2 is not efficient: we will lose more time transferring information between master and workers than we will gain from parallelizing the computation of each of the four entries of C. But a simple extension of this algorithm will parallelize the multiplication of matrices with hundreds of rows and columns, a costly task that appears in all sorts of scientific computations (for instance, when we compute a standard OLS estimator with many regressors and large data sets), and generate significant

time gains.

In our second example, we want to find the average of a sample of observations:

$$y = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

and, again, we have access to a machine with five processors divided between a master and four workers. The master can send $\{1, 2\}$ to the first worker to compute the average of the first two observations, $\{3, 4\}$ to the second worker to compute the average of the next two observations, and so on with the third and fourth workers. Then, the four averages are returned to the master, which computes the average of the four sub-averages $\{1.5, 3.5, 5.5, 7.5\}$. Thanks to the linearity of the average operator, the average of sub-averages, 4.5, is indeed the average of y. As was the case for our matrix multiplication example, computing the average of 8 observations by parallelization is inefficient because of the overheads in transmitting information between master and workers.

But this approach is a transparent example of the `MapReduce` programming model, a successful paradigm for the handling of millions of observations (among other tasks, later we will show how it can be applied as well to the solution of a life-cycle model). `MapReduce` is composed of two steps: a `Map` operator that takes an input and generates a set of intermediate values (in this case the input is y and the intermediate inputs are the sub-sample averages) and a `Reduce` operator that transforms the intermediate values into a useful final value, which, in our example, computes the average of the four sub-sample averages. In addition, `MapReduce` can be applied recursively, and therefore, the final value can be used as on intermediate input in another computation.[6]

The previous two examples highlight two separate issues: first, the need to develop algorithms that divide problems of interest into components that can be parallelized; second, the need to have coding techniques that can "tell" the computer how to distribute the work between a master and workers.

In the next pages, we can offer only a very parsimonious introduction for economists to these issues, and the interested reader should consult more complete references. A few books we have found particularly useful to extend our knowledge include:

1. Introduction to High Performance Computing for Scientists and Engineers by Hager

[6] `MapReduce`'s name derives from two combinators in `Lisp`, a deeply influential functional programming language created in 1958, `map` and `reduce`. `Google` developed the modern-day `MapReduce` approach to handle the immense amounts of data its search engine requires. `Hadoop` is a popular implementation of `MapReduce`. Currently, a more efficient alternative, `Spark`, based on the resilient distributed data set (RDD), is quickly overtaking `Hadoop` as a weapon of choice in handling large data problems with massive parallelization.

and Wellein (2011).

2. An Introduction to Parallel Programming by Pacheco (2011).

3. Principles of Parallel Programming by Lin and Snyder (2008).

4. Parallel Programming for Multicore and Cluster Systems by Rünger and Rauber (2013).

5. Structured Parallel Programming: Patterns for Efficient Computation by McCool, Robison, and Reinders (2012).

Below, we will cite some additional books as we deal with the concrete aspects of parallelization.

4 When Do We Parallelize?

The two examples above show the promise of parallelization for high-performance computing, but also its potential drawbacks regarding algorithmic design and communication among processors.

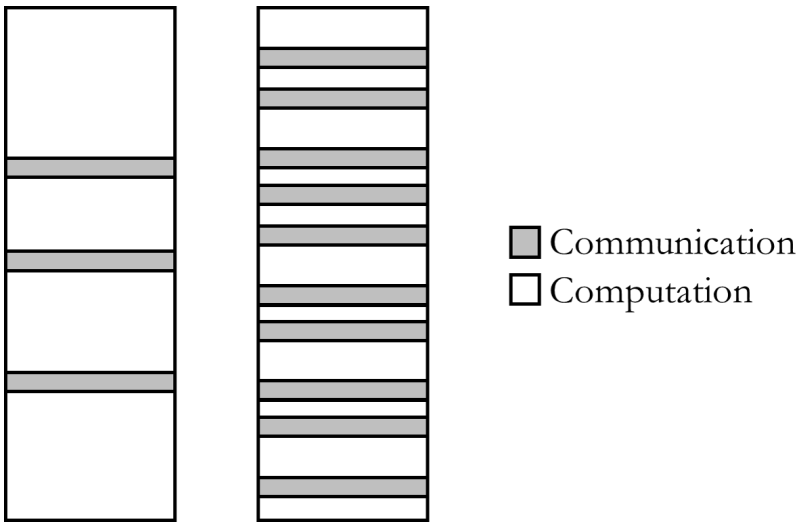


Figure 4: Granularity

Problems in high-performance computing can be classified according to their scalability (i.e., how effectively we can divide the original problem of interest into smaller tasks) and granularity (i.e., the measure of the amount of work involved in each computational task

in which the original problem of interest is subdivided). Depending on their scalability, problems are either strongly scalable –i.e., inherently easy to parallelize– and weakly scalable –i.e., inherently difficult to parallelize. Depending on their granularity, problems can be coarse –i.e., requiring more computation than communication between a master and workers –or fine –i.e., requiring more communication than computation. In Figure 4, we plot two columns representing the tasks involved in the computation of two problems. The left column is a computation requiring mostly computation and little communication among processors. The right column, in comparison, is a problem with much more communication. While the former problem is a good candidate for parallelization, the latter is not.

However, whether the problem is easy to parallelize may depend on the way you set it up. You need to take advantage of the architecture of your computer and exploit it to the maximum. This will help you to control for overheads (for instance, in the example of sample average computation, the final sum of sub-sample averages) and achieve optimal load balancing (we need to assign to each different processor chunks of the problem that require roughly the same amount of time).

These considerations dictate relying on productivity tools such as a state-of-the-art IDE (integrated development environment), a debugger, and a profiler that can handle parallel code. Without these tools, coding, debugging, and profiling your program for accuracy and performance can become most difficult, as there is much more potential for mistakes and inefficiencies in a parallel code than in a serial one. Also, these productivity tools may help you achieve a better tradeoff between speed-ups delivered by parallelization and coding time. There is no point in parallelizing a code if doing so takes more time than the savings in running time, even after considering the numerous times a code might need to be run during a whole research project.

To illustrate some of these ideas, we introduce two common applications in economics: one where parallelization works wonderfully well and one where it struggles to achieve speed improvements.

4.1 An application where parallelization works: Value function iteration

Our first application is a computational problem where parallelization works with ease: the canonical value function iteration problem.[7]

[7] A complete treatment of dynamic programming with asynchronous algorithms can be found in Bertsekas (2012, pp. 138-156).

Let us assume that we have a social planner that is maximizing the discounted utility of a representative household in a standard deterministic neoclassical growth model. The recursive representation of the problem of this social planner can be written in terms of a value function $V(\cdot)$ as:

$$V(k) = \max_{k'} \{u(c) + \beta V(k')\}$$
$$\text{s.t. } c = k^{\alpha} + (1-\delta)k - k'$$

where k is aggregate capital in the current period, k' the aggregate capital in the next period, c the consumption of the representative household, $u(\cdot)$ the period utility function, β the discount factor, α the elasticity of output to capital, and δ the depreciation rate.

The solution of the social planner's problem can be found by applying value function iteration. Imagine that we have access to j processors in a parallel pool. Then, we can implement the following algorithm:

Data:
Grid of capital with j points, $k \in [k_1, k_2, ..., k_j]$.
Initial guess of value function over the capital grid $V^0(k)$.
Tolerance ε.
Result:
Converged value function $V^{\infty}(k)$
Optimal decision rule $k' = g^{\infty}(k)$
while $\sup |V^n(k) - V^{n-1}(k)| < \varepsilon$ **do**
  Send the problem:

$$\max_{k'} \{u(c) + \beta V^n(k')\}$$
$$\text{s.t. } c = k_1^{\alpha} + (1-\delta)k_1 - k'$$

  to processor 1 to get $V^{n+1}(k_1)$. Store optimal decision rule $k' = g^{n+1}(k_1)$.
  Send the problem associated with k_i to worker i.
  When all processors are done, gather $V^{n+1}(k_i)$ for $k \in [k_1, k_2, ..., k_j]$ back and construct $V^n(k)$.
end

Algorithm 1: Parallelized value function iteration

The intuition of the algorithm is straightforward. Given a current guess of the value function $V^n(k)$, processor $m \leq j$ can solve for the optimal choice of capital next period

given capital k_m without worrying about what other processors are doing for other points in the grid of capital. Once every processor has solved for the optimal choice of capital, we can gather all $V^{n+1}(k_m)$ and move to the next iteration. This scheme is often called the "fork-join" paradigm.

Figure 5 illustrates the execution of the fork-join parallel code that would compute the model presented above. At every iteration n, the master node splits the computation across the parallel workers (*fork*). When all processors are done with the computations, the information is passed back to the master node (*join*) and the process is repeated.

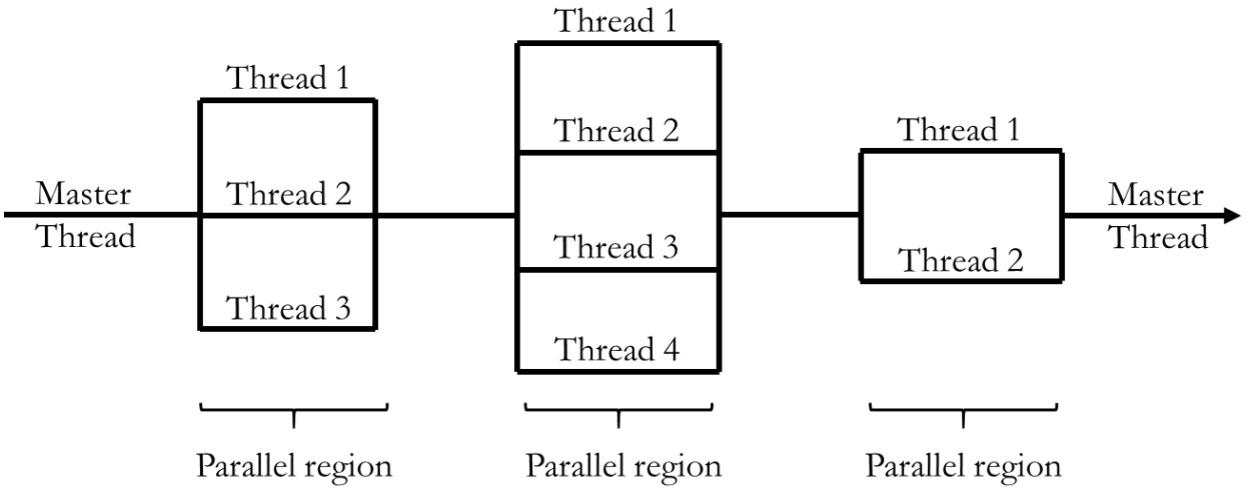


Figure 5: Fork-join parallel programming

The algorithm is easily generalized to the case where we have more grid points j than workers w by sending to each processor $\lfloor \frac{j}{w} \rfloor$ grid points (and the remaining points $j - w * \lfloor \frac{j}{w} \rfloor$ to any worker). Furthermore, this parallelization algorithm is efficient, as it allows an increase in speed that can be nearly linear in the number of processors, and it is easy to code and debug.[8]

[8]The speed-up will depend on the quantity of information moved and the quality of the hardware performing the communication among the processors. Also, the gain in efficiency is diminished by the difficulty in applying properties of the solution such as the monotonicity of the decision rule of the social planner (i.e., $g^{\infty}(k_j) < g^{\infty}(k_{j+1})$) that simplify the computation involved in the max operator in the Bellman equation. However, when the number of grid points is sufficiently larger than the number of workers, this difficulty is only a minor inconvenience. Furthermore, there are several "tricks" to incorporate additional information in the iteration to capture most of the gains from monotonicity.

4.2 An application where parallelization struggles: A random walk Metropolis-Hastings

Our second application is a computational task where parallelization faces a difficult time: a random walk Metropolis-Hastings, the archetypical Markov chain Monte Carlo method.

Many problems in econometrics require drawing from a distribution. More concretely, we want to draw a sequence of $\theta_1, \theta_2, \theta_3, ...$, such that:

$$\theta_i \sim P(\cdot).$$

This draw may allow us to simulate some stochastic process where the θ's are the innovations or to evaluate functions of interest of the distribution –such as its moments– appealing to a law of large numbers (i.e., the theoretical mean of the distribution will be well approximated by the mean of the drawn θ's).

In principle, computers face a difficult time drawing a random number: they are, after all, deterministic machines that follow a predetermined set of instructions. Computer scientists and applied mathematicians have developed, nevertheless, deterministic algorithms that generate a sequence of pseudo-random numbers and whose properties are, up to some cycle, very close to those of a truly random sequence.[9]

These pseudo-random number generators typically produce a sequence of draws that mimics a draw from a uniform distribution in $[0, 1]$. Through different change-of-distribution transformations, such a sequence from a uniform distribution can yield a sequence from a wide family of known parametric distributions (Devroye, 1986, provides an encyclopedic treatment of such transformations).

The challenge appears when the distribution $P(\cdot)$ does not belong to any known family. In fact, it is often the case that we cannot even write down $P(\cdot)$, although we can numerically *evaluate* it at an arbitrary point θ_i. This happens, for example, with the likelihood function of a dynamic equilibrium model: a filter (either linear, such as the Kalman filter, or non-linear, such as the particle filter in Fernández-Villaverde and Rubio-Ramírez, 2007) provides the researcher with a procedure to evaluate the likelihood of the model given some observations for arbitrary parameter values, but not how to write a closed-form expression for it. We want to draw from this likelihood function to implement a Bayesian approach (after the likelihood has been multiplied by a prior) or to maximize it, from a frequentist perspective, through

[9] When true randomness is absolutely required, such as in cryptography, one can rely on different physical phenomena built into hardware with true random number generators. This is, however, rarely necessary in economics.

some stochastic optimization search algorithm.

A random walk Metropolis-Hastings is an algorithm that allows us to draw from an arbitrary distribution a Markov chain whose ergodic properties replicate those of the distribution of interest. The arrival of the random walk Metropolis-Hastings and other Markov chain Monte Carlo methods in the 1990s revolutionized econometrics and led to an explosion of research in Bayesian methods (Robert, 2007) and estimation by simulation (Gourieroux and Monfort, 1997).

The algorithm is as follows:

Data:

Distribution of interest $P(\cdot)$.

Initial value of the chain θ_1.

Scaling of the innovation λ.

Result:

Sequence $\{\theta_1, \theta_2, ..., \theta_m\}$.

while $n \leq m$ **do**

Given a state of the chain θ_{n-1}, generate a proposal:

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

Compute:

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

Set:

$$\begin{aligned} \theta_n &= \theta^* \text{ with probability } \alpha \\ \theta_n &= \theta_{n-1} \text{ with probability } 1-\alpha \end{aligned}$$

end

Algorithm 2: Random walk Metropolis-Hastings

The key of the algorithm is to generate a proposal θ^* for the new state of the chain through a random walk specification

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

(the proposal is the new state plus an innovation) that is accepted with some probability α

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

construed to induce the desired ergodic properties of the drawn sequence. By appropriately tuning λ, we can get an optimal acceptance rate and achieve faster convergence (Roberts, Gelman, and Gilks, 1997).

From the perspective of parallelization, the problem of the algorithm is that to generate θ^*, we need θ_{n-1}, so we cannot instruct processor i to compute θ_{n-1} and, at the same time, instruct processor j to compute θ_n. The Markov property of the chain makes it an inherently serial algorithm. Thus, we cannot easily subdivide the problem. The "obvious" solution of running parallel chains fails because it violates the asymptotic properties of the chain: we need a long sequence of size m, not many short sequences that add up to m, as the latter might not have traveled to the ergodic regions of the distribution.

Researchers have attempted to come up with proposals to parallelize the random walk Metropolis-Hastings algorithm (see, among others, Ingvar, 2010, Calderhead, 2014, and Martino, Elvira, Luengo, Corander, and Louzada, 2016). Also, in economics, other parts of the problem (such as the evaluation of the distribution $P(\cdot)$) can be parallelized. For example, if $P(\cdot)$ comes from a particle filter, such a filter is easy to parallelize. But despite these partial remedies, at the end of the day, the random walk Metropolis-Hastings algorithm is not a good candidate for parallelization. Unfortunately, many other problems in economics (such as those involving long simulations) suffer from the same drawback. Parallelization is a great tool for many computational challenges in economics, but not for all.

4.3 The road ahead

We will now move on to show how to parallelize in a realistic application in economics: a life-cycle model of consumption-saving. Slightly more complex versions of this model are at the core of much research in economics regarding consumption and savings across ages (Fernández-Villaverde and Krueger, 2011), portfolio choice (Cocco, 2005), household formation (Chang, 2017), human capital accumulation (Ben-Porath, 1967), housing decisions (Zarruk-Valencia, 2017), taxation policies (Nishiyama and Smetters, 2014), social security reform (Conesa and Krueger, 1999), etc. Not only is the application simple enough to fulfill its pedagogical goals, but it also presents sufficient sophistication as to illustrate the most relevant features of parallelization.

5 A Life-Cycle Model

We consider the following model for our experiments. We have an economy populated by a continuum of individuals of measure one who live for T periods. Individuals supply inelastically one unit of labor each period and receive labor income according to a market-determined wage, w, and an idiosyncratic and uninsurable productivity shock e. This shock follows an age-independent Markov chain, with the probability of moving from shock e_j to e_k given by $\mathbb{P}(e_k|e_j)$. Individuals have access to one-period risk-free bonds, x, with return given by a market-determined net interest rate, r.

Given the age, t, an exogenous productivity shock, e, and savings from last period, x, the individual chooses the optimal amount of consumption, c, and savings to carry for next period, x'. The problem of the household during periods $t \in \{1, \ldots, T-1\}$ is described by the following Bellman equation:

$$\begin{aligned} V(t, x, e) = \max_{\{c \geq 0, x'\}} \quad & u(c) + \beta \mathbb{E} V(t+1, x', e') \\ \text{s.t. } & c + x' = (1+r)x + ew \\ & e' \sim \mathbb{P}(e'|e), \end{aligned}$$

for some initial savings and productivity shock.[10] The problem of the household in period T is just to consume all resources, yielding a terminal condition for the value function:

$$V(T, x, e) = u((1+r)x + ew).$$

A terminal condition, such as a value of bequests or altruism for future generations, could be easily added.

As referred to earlier, even this simple life-cycle model has important features related to parallelization. First, it is a problem with three state variables. This means that if the number of grid points on each variable is sufficiently large, the model might take some time to compute. If another state variable is included, the model might become infeasible to estimate. For instance, any model that studies health, housing, durable goods, or the consequences of having multiple financial assets implies keeping track of additional state variables and estimation requires the repeated solution of the model for many different parameter values. Second, although there are three state variables, the problem is not parallelizable along the

[10] For the purpose of this paper, we do not need to worry too much about how we specify those initial conditions, and we would compute the solution of the model for all possible initial conditions within a grid. In real research applications, the initial conditions are usually given by the desire to match observed data.

three dimensions, as will become clear later.

The following pseudo-code summarizes how to compute the model in serial using a standard backward-induction scheme:

Data:

Grid for assets $X = \{x_1, \ldots, x_{n_x}\}$.

Grid for shocks $E = \{e_1, \ldots, e_{n_e}\}$.

Transition matrix $\mathbb{P}(e'|e)$.

Result:

Value function $V(t, x_i, e_j)$ for $t = 1, ..., T$, $i = 1, ..., n_x$, and $j = 1, ..., n_e$.

for $\forall\ x_i \in X$ *and* $e_j \in E$ **do**

$$V(T, x_i, e_j) = u((1 + r)x_i + e_j w)$$

end

for $t = T - 1, \ldots, 1$, **do**

Use $V(t + 1, x_i, e_j)$ to solve:

$$V(t, x_i, e_j) = \max_{\{c, x' \in X\}} \quad u(c) + \beta \sum_{k=1}^{n_e} \mathbb{P}(e_k|e_j) V(t + 1, x', e_k)$$

$$\text{s.t. } c + x' = (1 + r)x_i + e_j w$$

end

Algorithm 3: Life-cycle value function

Note that the expectation in the right-hand side of the Bellman operator has been substituted by a sum over future value functions by taking advantage of the Markov chain structure of the productivity shock.

The `Julia` code in Box 1 computes the value function in serial:

In the code, the function `Value` computes the maximum value attainable given state variables `age, ix, ie`. We loop over all possible combinations of the state variables to get our value function. The function `Value` is described in Box 2. The function searches, among all the possible values of the grid for assets, for the optimal level of savings for the next period. For the last period of life, the function picks, trivially, zero savings (if we had a terminal condition such as a valuation for bequests, such a choice could be different). We could have more sophisticated search methods but, for our argument, a simple grid search is most transparent.

This life-cycle problem can be parallelized along the saving and productivity shock di-

```
for(age = T:-1:1)
  for(ix = 1:nx)
    for(ie = 1:ne)

      V[age, ix, ie] = Value(age, ix, ie);

    end
  end
end
```

Box 1: `Julia` code to compute the value function.

```
function Value(age, ix, ie)
   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

   return(VV);
end
```

Box 2: `Julia` code to compute the value function, given state variables `age, ix, ie`.

mensions (x and e), but not along the age dimension. Given the age of the household, t, the computation of $V(t, x_i, e_j)$ is independent of the computation of $V(t, x_{i'}, e_{j'})$, for $i \neq i'$, $j \neq j'$. While one of the processors computes $V(t, x_i, e_j)$, another processor can compute $V(t, x_{i'}, e_{j'})$ without them having to communicate. For a given t, a parallel algorithm could

split the grid points along the x and e dimensions, and assign different computations to each of the processors, so that total computation speed can be significantly reduced. In other words, the problem can be parallelized along the x and e dimensions in the same way that we parallelized the basic value function iteration in Subsection 4.1. This is not the case, however, for the age dimension. For the computation of any $V(t, x_i, e_j)$, the computer has to first have computed $V(t+1, x_i, e_j)$. The computation is, by its very nature, recursive along t.

The following pseudo-code computes the model in parallel:

begin

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

end

Algorithm 4: Parallelizing the value function

Figures 6 and 7 illustrate the performance of each of the processors of an 8-thread CPU when the process is executed in serial (Figure 6) and in parallel with the maximum number of threads (Figure 7). While in the first figure only one processor works (CPU2, represented by the line that jumps in the middle), in the second figure all 8 CPUs are occupied.

6 Different Computers

Before we show how to implement Algorithm 4 in practice, we must spend some time discussing how the choice of a particular computer matters for parallelization and how we implement it.

Amdahl's law (name after the computer architect Gene Amdahl) states that the speed-up of a program using multiple processors in parallel is limited by the time needed for the sequential fraction of the program. That is, if 25 percent of the computation is inherently serial, even a machine with thousands of processors cannot lower computation time below 25 percent of its duration when running serially. Thus, the speed of each processor is still of paramount relevance.

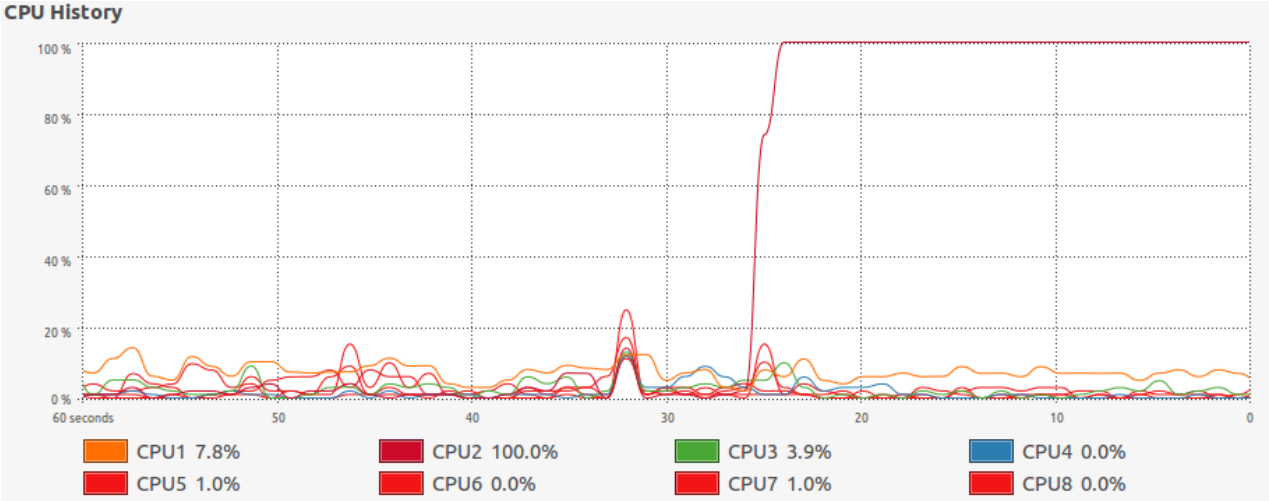


Figure 6: 1 Core used for computation

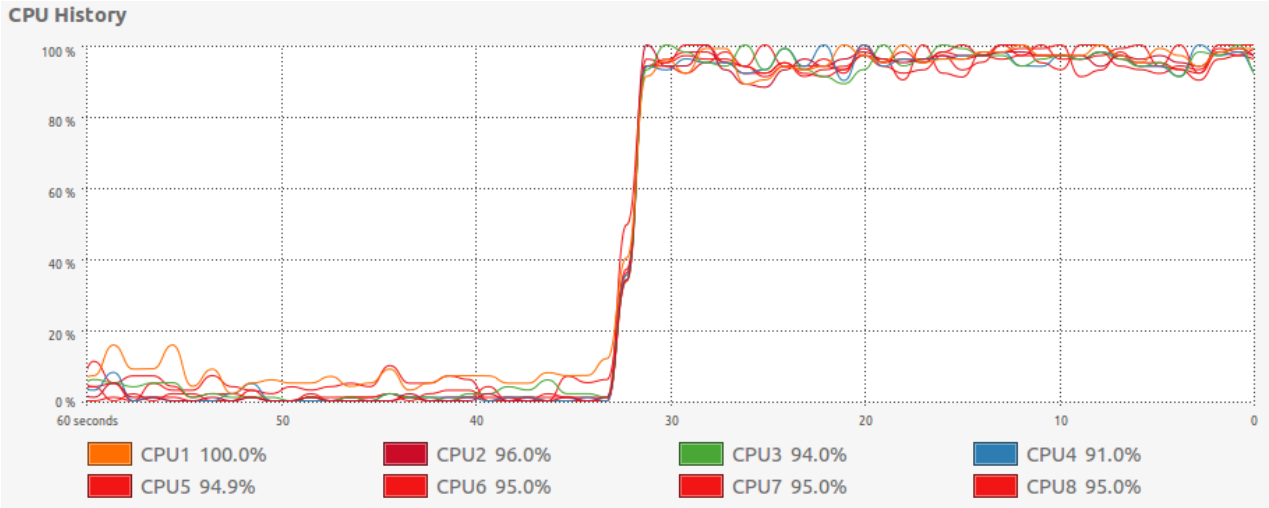


Figure 7: 8 Cores used for computation

Similarly (and as we will describe below in more detail for each programming language), we face the fixed cost of starting a thread or a process/worker when we want to distribute computations among workers, transfer shared data to them, and synchronize results. These tasks are time intensive and, often, they depend more on the hardware allowing the communication among processors than on the quality of the processors themselves. Indeed, much of the cost of state-of-the-art supercomputers is in top-of-the-line equipment that maximizes network bandwidth and minimizes latency, not in the processors. Furthermore, the primary barrier facing extremely large machines is load imbalance among processors. In actual applications, it is difficult to use more than 10 percent of the theoretically available computing

power.

These issues are somewhat less relevant for users of workstations and desktops with just a few processors, but cannot be completely ignored. For instance, an `Intel Core™ i9-7980XE Extreme Edition` has 18 cores and can run up to 36 threads. A performance workstation equipped with such a processor is priced at less than $5,000 and, thus, affordable to many researchers. This unprecedented level of sophistication for a personal computer requires an investment of time and effort in getting to know the limits a particular computer imposes on parallelization. A thorough discussion of such issues is well beyond the scope of this chapter (and would be, in any case, contingent on rapidly evolving hardware).

We would be remiss, however, if we did not mention other hardware possibilities. For example, many institutions provide affiliated researchers with large servers, some of them with dozens of cores, and even with mighty supercomputers. Similarly, several companies offer computing services on demand over the internet. The main advantage of these services is the replacement of a large initial capital cost (purchasing a workstation) for a variable cost (use-as-needed), the supply of constant updates in the hardware, and the avoidance of maintenance considerations (at times acute in a department of economics where IT support can be spotty).

A popular example of those services is `Amazon Elastic Compute Cloud (EC2)`, which allows requesting nearly as much computational power as one would desire in economics.[11] As of April 2018, you can rent on-demand 8 processors with 32 GiB of memory, running on `Linux` and gathered for general purposes for $0.3712 per hour and 96 processors with 384 GiB of memory for $4.608 per hour. These prices mean that one can use 96 processors for an entire month, non-stop, for $3,318.[12] Prices can be substantially cheaper if the user is willing to pay for spot pricing at lower-demand times.

The access to `EC2` is straightforward. After setting up an account, one can go to Amazon EC2, click on "Launch a virtual machine" and follow the webpage links (for example, one can select `Ubuntu Server 16.04 LTS`, an instance that runs on the well-known `Ubuntu` distribution of `Linux`).

The only difficulty lies in the need to use a public key. The required steps are i) create a new key pair; ii) download the key; iii) store it in a secure place (usually `~./ssh/`); and iv) run an instance.

[11] https://aws.amazon.com/ec2/, where in addition, the reader can find complete documentation of how to create and operate an instance online.

[12] https://aws.amazon.com/ec2/pricing/on-demand/. If one considers the expected life of a server before it becomes obsolete plus the burden of maintenance in terms of time and effort, the user cost of the server per month might be quite higher than $3,318.

On an Ubuntu terminal this requires, first, transfering the folder from local to instance with `scp`:

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

Second, to make sure the key is not publicly available:

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

And, third, to connect to instance with `ssh`:

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

Once the instance is running, the terminal where it operates works like any other `Linux` terminal and the user can execute its parallel codes.

7 Parallelization Schemes on CPUs

We are now ready to review several of the most popular parallelization schemes that take advantage of having access to multiple cores (physical or virtual), either in one or several CPUs. This is the most traditional form of parallelism. In Section 8, we will review parallelization schemes for GPUs and mixed approaches that can use both CPUs and GPUs. This second form of parallelism has gained much attention during the last decade and it deserves its own section.

Except for the case of `MPI` in Subsection 7.8 with its rich set of operations, the schemes in this section will be, in some form or another, simple variations of two different ideas: the `for` loop and `Map` and `Reduce`. In the `for` loop approach, we add a statement before the `for` loop that we want to parallelize. For example, in the code displayed in Box 2, if we want to parallelize the computation of the value function along the productivity (`x`) grid, we add a statement on the code specifying that the `for(ix = 1:nx)` loop should be computed in parallel. This approach is simple to implement, transparent in its working, and efficient for many computations. However, it faces scalability constraints.

In comparison, a `Map` function receives as inputs the function `Value` described in Box 1 and a data structure containing every possible combination of the state variables, and returns the value associated which each of them. This computation is done in parallel, so there is no need to loop over all state variables as in Box 2. Although our particular case does not

require it, a `Reduce` function takes the values computed by the `Map` and reduces them in the desired way, such as taking the average or the sum of them.

In most programming languages, there is at least one package/extension to perform parallel computing, and most follow one of the two procedures described above. Some of them require the programmer to add a statement before the loop that should be computed in parallel, while some others require the use of a `Map` function. In the remaining sections of this paper, we show how to implement these schemes in some of the most popular programming languages in economics. More concretely, we will deal with `Julia`, `Matlab`, `R`, `Python`, `C++` with `OpenMP`, and `Rcpp` with `OpenMP`. We will close with explanations of how to apply `MPI`, which can interact with different programming languages.[13]

7.1 Julia 1.0 - `Parallel for`

`Julia` is a modern, expressive, open-source programming language with excellent abstraction, functional, and metaprogramming capabilities. Designed from the start for high-performance numerical computing, `Julia` is particularly well-suited for economics.

A design principle of `Julia` was to provide tools for easy and efficient parallelization of numerical code. One of those tools is the `Distributed` module, and its `@distributed for` prefix, designed for executing `for` loops in parallel. This way of parallelizing is most convenient for computing "small tasks," or situations where there are few computations on each parallel iteration. This might be the case of models with few control variables, few grid points on control variables, or when the computation requires relatively few steps.[14]

To implement `@distributed for`, we start by importing the `Distributed` package and setting up the number of workers according to the architecture of our machine with the instruction `addprocs()`:

```
using Distributed
addprocs(6) # Here we have 6; check the number of processors you have
   available
```

[13] As we indicated in the introduction, we do not try to be exhaustive in our coverage of programming languages, as such an exercise would be too cumbersome. Furthermore, the set of languages we select spans other alternatives with minimal changes. For example, once one understands how to implement parallelization in `C++` with `OpenMP`, doing the same in `Scala` or `Fortran` is straightforward. For a comparison of programming languages in economics that motivates our choice of languages for this exercise, see Aruoba and Fernández-Villaverde (2015).

[14] See Julia Parallel Computing. These paragraphs follow the syntax of `Julia v1.0.1`, the current release as of August 2018.

In `Julia`, as in most languages described below, initializing the pool of workers can take some time. However, the programmer has to consider that this is a one-time fixed cost that does not depend on the size of the problem. If the parallel pool is initialized to perform a time-consuming task or to repeat the same task multiple times, the fixed cost vanishes as a proportion of total computation time. However, if the user only uses the parallel pool to perform a one-time small task, initializing the pool of workers might make parallel computing more time expensive than serial computing.

Once a pool of workers has been initialized, there are two useful functions to monitor and modify the state of the parallel pool. The function `workers()` returns the number of parallel workers in the pool, as well as their identification number. The function `rmprocs()` allows the user to remove existing workers from the pool:

```
workers()       # Number of parallel workers
rmprocs(2,3,5)  # Removes workers with id 2, 3 and 4
```

When variables are declared in `Julia`, they are not part of the shared memory by default. That is, variables are not generally observable/accessible by workers. To grant the workers access to the variables they need for their computations, we must explicitly declare those variables to be accessible/modifiable by workers. If we only want a variable to be observed by workers, but not modified on parallel iterations, we should declare it as global with `@everywhere`:

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

However, there are some variables that we want the parallel workers to modify simultaneously in each iteration. In our example, we want the workers to modify the entries of the value function matrix `V` that are assigned to them. To accomplish this, we have to load the `SharedArrays` package and declare these variables as `SharedArray`:

```
using SharedArrays
V = SharedArray{Float64}(T*nx*ne);
```

Once the required variables have been declared to be observable and modifiable by the workers, we should explicitly state the `for` loop we want to be executed in parallel. For this,

it suffices to add the directive `@distributed` before the `for` statement. For example, if we want to compute in parallel the values assigned to each point in the savings grid, we add `@distributed` before the `for` loop associated with the x grid:

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

As explained above, our life-cycle problem can be parallelized across the savings and productivity dimensions, x and e, but not across age, t. This means that, for every t, we should assign grid points in the $E \times X$ space to each worker to compute the value assigned to those points, gather back the computations, stack them on the matrix $V(t, \cdot, \cdot)$, iterate to $t-1$, and repeat. Before iterating to $t-1$, we need all the workers to finish their job so that we can collect all the information in $V(t, \cdot, \cdot)$. To make sure that every worker finishes its job before iterating to $t-1$, we must add the directive `@sync` before the `@distributed for` statement:

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

These three elements (i. declaring observable and modifiable variables, ii. a directive `@distributed for` loop, and iii. a directive `@sync`) are all one needs to run the code in parallel in `Julia`. The execution of the code in the REPL terminal or the batch model is the same as for any other `Julia` file.

There is, however, an important further point to discuss. In our example, we stated that the problem could be computed in parallel across two dimensions: x and e. Naturally, it turns out that it is not possible to embed a `@distributed for` loop inside another `@distributed for` loop. The programmer must choose whether to perform parallel computing on one grid but not on the other, or to convert the two nested loops into a single loop that iterates over all the points in both grids.

Let us explore the alternative of parallelizing along one dimension, but not the other. The code below presents the two possible choices of the loop 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
```

Although the choice of computing in parallel the x or the e grid might seem innocuous, the implied performance differences are large. Computing in parallel the loop corresponding to the x grid is much faster. The reason is mainly the time spent in communication vs. actual computation. There are many more grid points in the x grid (`nx=350`), than in the e grid (`ne=9`)[15]. In the left column above, for every point in the e grid, the master node has to split the x grid among the workers. That is, on every iteration on the e grid, the master node has to communicate and exchange information with the workers. In the above example, with 9 points on the e grid, the master node has to move information 9 times. On each of these 9 iterations, the master node divides a grid of 350 points among 8 workers. If the tasks are equally split, each worker will have around 43 computations on every iteration. In the right column, when we choose to parallelize across the e grid, the opposite happens. Since `nx=350`, the master node has to move information 350 times, one after every iteration. Moreover, the computing time is small, as the 9 grid points in the e grid are divided among 8 workers; that is, at most, two computations per worker! In this case, the communication time is very large when compared to computation time, so parallel computing is not only slower than in the first case, but it can be even slower than any parallelization at all.

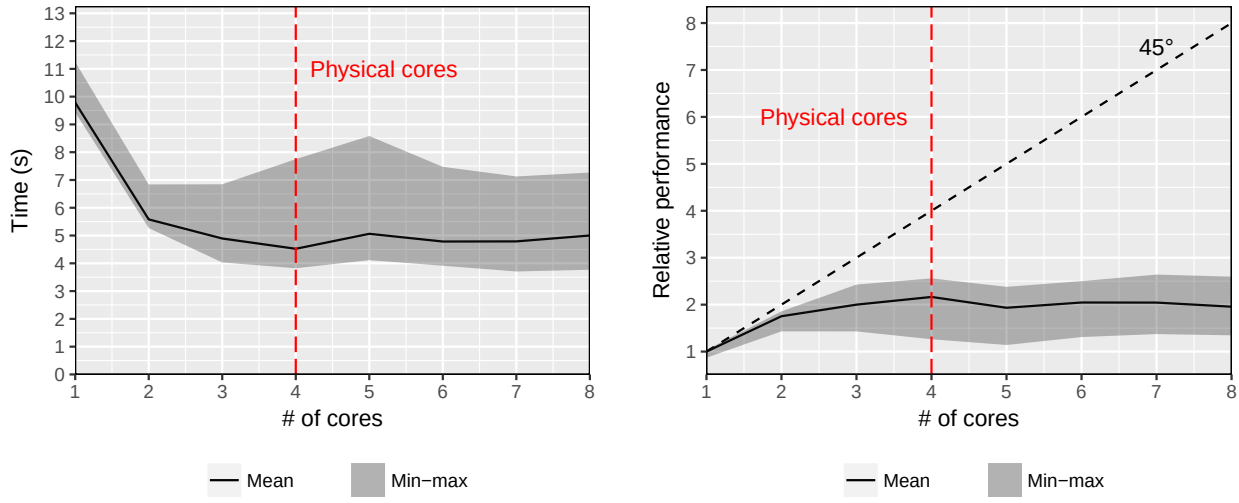
The second alternative collapses the two nested loops into a single loop that iterates over all the grid points. In this case, the parallelization is performed over all the grid points, which minimizes the communication time and maximizes the computations per worker:

```
  @sync @distributed for(ind = 1:(ne*nx))
      ix = convert(Int, ceil(ind/ne));
      ie = convert(Int, floor(mod(ind-0.05, ne))+1);
      # ...
  end
```

In the Github repository of this paper, the reader can find an example in which we

[15] This is typical in life-cycle models; in other models, the largest grid can be another dimension.

compute the baseline model described with parallel loops across both x and e dimensions. We repeat the computation of the model with a different number of cores for parallelization, up to 8 cores. The experiments are performed on a laptop computer with 4 physical cores and 8 virtual cores.[16] In each case, we execute the code 20 times, to account for small differences in performance between runs.



(a) Computing time (s)

(b) Performance gains.

Figure 8: Results in `Julia` with different number of processors. Number of experiments: 20.

Figure 8a illustrates the average total computing time, with the number of cores employed on the x-axis and the time in seconds on the y-axis. The performance gains are substantial up to the third core, going from around 10 seconds to 5, but there are no large gains after that. Since total time spent in communication is large, adding workers does not necessarily increase speed. This point is stressed in Figure 8b, which plots the number of cores against the performance gains in terms of the normalized performance of one processor. If the parallelization is perfect and no time is spent in communication, the performance gains would be linear. That is, doubling the number of processors should double the speed of computation, as represented by the 45° line (the inherently serial part of the code affected by Amdahl's law is small). In reality, the performance gains are significantly below the optimal gains. To make this argument clear, we plot in both figures the mean of the runs and the band between the min and max performance. In the `Julia` case with `@distributed for`, there are gains below optimal up to the third core, and the performance is almost constant after that. Of course, in larger problems (for instance, with more points in the state variable

[16] More concretely, we use a Lenovo Y50-70 with `Intel Core™ i7-4710HQ` CPU @2.50GHzx8, 15.6 GiB, and `NVIDIA GeForce GTX 860M/PCIe/SSE2` GPU. The operating system is `Ubuntu 14.04`, 64-bit.

grids), performance might continue increasing with additional processors. Similarly, better machines (for example, with more physical processors) might deliver improved performance for a longer range of processors used.

When implementing parallel computing in `Julia`, the programmer must take into account that the total computation speed is particularly sensitive to the use of global (`@everywhere`) variables and large `SharedArray` objects. Many elements in shared memory can significantly hamper the potential time gains that parallel computing brings. In applications where shared memory is sizable, it might even be preferable to perform serial computing.

7.2 Julia 1.0 - `MapReduce`

Recall that the `@distributed for` is designed for applications where the problems assigned to each worker are small, such as our life-cycle model. In cases where there are many more control variables or where we work with fine-mesh grids, or in models with many discrete choices, `@distributed for` might slow down the computation. In these situations, a `MapReduce`-type of function built into `Julia` is a superior approach.

To do so, we must look at our life-cycle problem from a different perspective. To solve the problem serially, we compute the maximum attainable utility –given the restrictions– for every point –t, x, and e– in the grids (see Algorithm 4). A `MapReduce` function, instead, takes as inputs a function and vector of values where we want to evaluate that function (recall our discussion in Section 3). In our case, the inputs would be a function `Value(states)` that computes the maximum utility for some given state variables `states`, and a data structure containing all possible combinations of the points in the grids for the state variables (or, perhaps, some relevant subset of those combinations).

We show now how to implement this idea in practice. As before, the first step is to load the `Distributed` module and initialize the number of parallel workers:

```
using Distributed
addprocs(6)
```

Next, we define a data structure `modelState` composed of all parameters and variables needed for the computation in `Value`, and declare this structure as global with `@everywhere`:

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

Note that the idea of creating a rich data structure to store all the state variables can be applied to a wide class of dynamic models that goes well beyond our simple life-cycle economy. More generally, using rich data structures allows for cleaner and less bug-prone code, since we can define operators on the whole structure or overload already existing ones, instead of operating on each element of the structure one at a time.[17]

Given that the function `Value` will now depend on the data structure `modelState`, instead of on each of the parameters, the function must be modified accordingly to take the structure as an input:

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

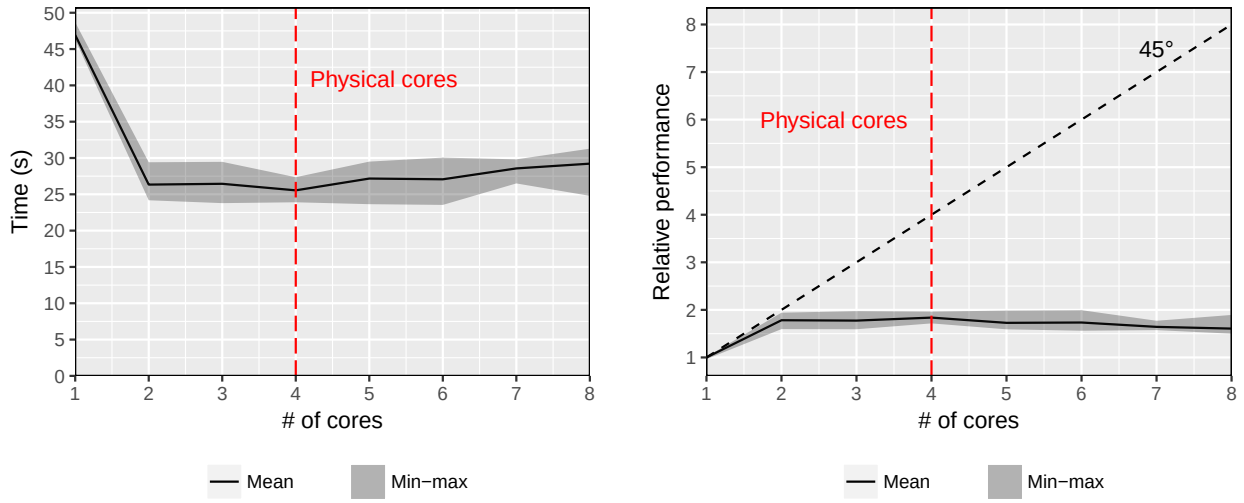
Since the workers in the parallel pool must be able to observe the function `Value`, this must also be declared as global with the directive `@everywhere`.

The last step is to use the `Map`-type of function to compute `Value` in parallel. In `Julia`, this is done with the `pmap` function (`pmap` generalizes `map` to the parallel case), which receives as inputs the function `Value` and a vector with all possible combinations of state variables as stored in our data structure `modelState`:

[17] Overloading an operator is to define a different effect of the operation when the inputs are different, i.e, the operator "+" works differently when applied to two integers than when we apply it to two matrices of complex numbers. Also, we can be flexible in the definition of structures. For example, we can incorporate on them just the states of the model, but not the parameters as well, as we do in the main text.

```
for(age = T:-1:1)
    pars = [modelState(ix, age, ..., w, r) for ind in 1:ne*nx];
    s = pmap(Value,pars);
    for(ind = 1:ne*nx)
      ix     = convert(Int, floor((ind-0.05)/ne))+1;
      ie     = convert(Int, floor(mod(ind-0.05, ne))+1);
      V[age, ix, ie] = s[ix];
    end
end
```

Figures 9a and 9b illustrate the total computing time and the time gains, respectively, with a different number of processors when using the `pmap` function. Absolute computing times using this procedure are very large when compared to the results of the last section, and the performance gains from adding more workers to the parallel pool are very poor. One possible explanation is that the `pmap` function is designed to be applied to tasks where each worker is assigned a large task. In our example, this is not the case. The computation speed, in this case, could also be reduced by applying `Julia`-specific tricks, such as wrapping the code inside a function. However, since the purpose of this paper is to make the comparison as transparent as possible, we avoid applying language-specific procedures different from the parallelization itself.



(a) Computing time (s)

(b) Performance gains.

Figure 9: Results with `pmap` function in `Julia` with different number of processors. Number of experiments: 20.

7.3 Matlab

`Matlab` performs parallel computing through the `parallel toolbox`.[18] While this tool is probably the simplest of all the alternatives covered in this paper to implement, it also suffers from limited performance and it is expensive to purchase.

As opposed to `Julia`, in which variables are neither observable nor modifiable by default by all the workers in the parallel pool, in `Matlab` they are. Every time you declare a variable, it can be accessed by any worker in the parallel pool. In this way, the programmer only has to take care of initializing the parallel pool of workers and explicitly stating which `for` loop she wants to be parallelized.

To initialize the parallel pool of workers, we use the function `parpool()`:

```
parpool(6)
```

In `Matlab`, as in `Julia`, initializing the workers in the parallel pool is also time consuming. It can take some seconds, even a considerable fraction of a minute. Again, if the parallel pool is initialized to perform a time-consuming task or to repeat the same task multiple times, the fixed cost of initializing the parallel pool vanishes as a proportion of total computation time, and parallel computing might increase performance. If the code only performs a small task, the initialization cost of parallel computing might offset its benefits.

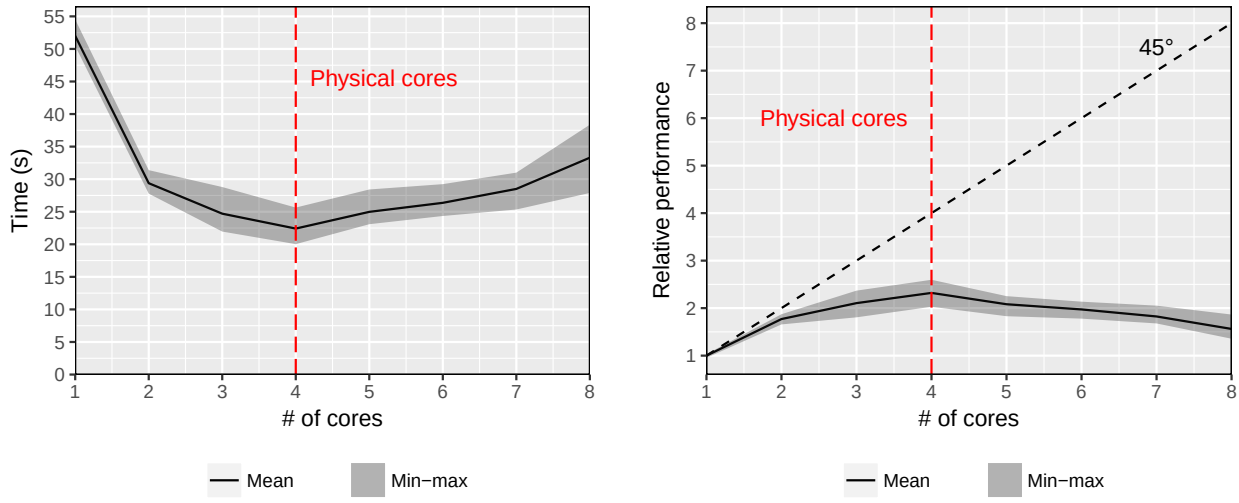
To state which loop is to be computed in parallel, instead of declaring the `for` loop, the programmer must declare it as `parfor`:

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

Beyond these two steps (i.e., initializing the parallel pool and employing `parfor`), the developer does not need to exert further effort and `Matlab` takes care of all the rest.

Total computing time in `Matlab` is considerably higher than in `Julia` with the `@parallel for` procedure, as illustrated by Figure 10a. While `Julia` takes around 16 seconds to compute the model with one core, `Matlab` takes over 50 seconds. Similar to `Julia`, the performance gains of using parallel computing in `Matlab` are not very large, as illustrated in Figure

[18] We employ `Matlab 2017b`, with its corresponding `parallel toolbox`.



(a) Computing time (s)

(b) Performance gains.

Figure 10: Results in `Matlab` with different number of processors. Number of experiments: 20.

10b. The speed-ups are well below the perfect linear gains represented by the 45° line. On average, using 4 workers in the parallel pool decreases computing time by less than 2.5 times. Computing in `Matlab` is very simple to implement, but does not bring substantial benefits when compared to other languages. Indeed, after the computer reaches the limit of physical cores, computing time deteriorates with additional virtual cores.

7.4 R

Parallelization in `R` is important because the language's speed in serial computations is quite often disappointing in comparison with other alternatives (see Aruoba and Fernández-Villaverde, 2015), and yet a researcher might want to code on it to take advantage of its superior statistical and graphics capabilities.

`R`, as an open source language, has multiple parallel computing packages created by different developers. Here we will cover only the widely used `parallel` package, although there are other alternatives, such as the `foreach` and `doParallel` (we use `R 3.4.1`, which includes `parallel` by default). In later sections, we will describe how to perform parallel computing with `C++` and how to embed `C++` code in `R` through the `Rcpp` package, allowing the user to expand the use of parallel computing in `R`.

The main functions in the `parallel` package for parallel computing are the `parLapply` and the `parSapply` function, which are `Map`-type functions that perform the same task as the

`lapply` and `sapply` functions, respectively. `lapply` applies a function to a list of variables. `sapply` performs the same task in a more user-friendly manner. `R` has strong functional programming capabilities to manipulate lists that are often overlooked (for a crip explanation of them, see Wickham, 2014).

To access the `parallel` package in `R`, the user must first install it and open the library:

```
install.packages("parallel")
library("parallel")
```

This package has the `parLapply` function. This is a parallel `Map` kind of function. As we did with `Julia`, when the parallel procedure employs a `MapReduce` scheme, the user must define the function `Value`:

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

In `R` there is no need to define a data structure, as the vector of parameter and state variables can be set as a `list`. Also, as opposed to `Julia`, in `R` all variables are observable and modifiable by the workers of the parallel pool. Therefore, it suffices to generate a list that contains all parameters and every possible combination of state variables. In our example, we perform this using the `lapply` function, which is a `Map` that returns a list with parameters and all possible combinations of state variables:

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

Here we could have used the `parLapply` function to create the `list` of inputs to solve our main problem. Since we want to emphasize the use of parallel computing to compute the value function, we omit this step.

The function `makeCluster` initializes the desired number of workers:

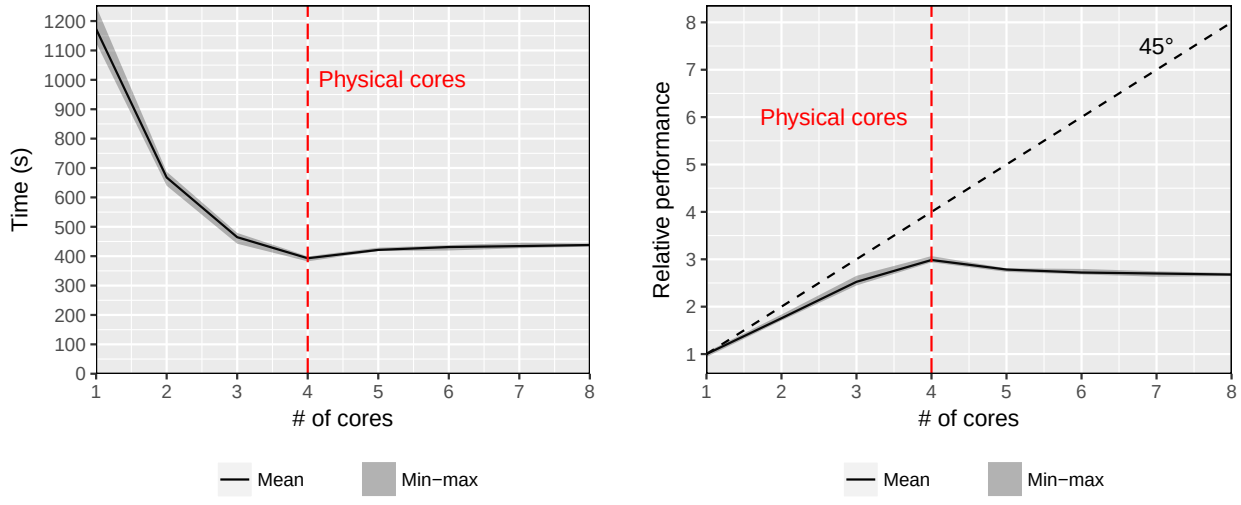
```
cl <- makeCluster(8)
```

Finally, the function `parLapply(cl, states, Value)` computes the function `Value` at every element of the list `states`, which contains all combinations of state variables. The package uses the cluster `cl` initialized above:

```
for(age in T:1){
   states = lapply(1:(nx*ne), function(x) list(age=age,ix=x, ...))
   s = parLapply(cl, states, Value)
}
```

The output of the `parLapply` function is a list with the elements of the evaluation of `Value` at each of the elements in `states`. To store them in a three-dimensional matrix `V`, the user must finally loop over all the elements of `s` and store them at `V`.

The results of our computation are reported in Figures 11a and 11b, where we see that `R` is extremely slow at computing the model. It takes over 1100 seconds with one core, and over 400 seconds with 8 cores (that is why we only run 10 experiments; with long run times, small differences in performance between one run and the next are negligible). However, the performance gains are good enough. Using 4 workers for the parallel computation reduces total computing time to almost a third. These results imply that, although `R` is a slow alternative to compute this class of models, the performance gains are relatively large.



(a) Computing time (s)

(b) Performance gains.

Figure 11: Results in `R` with different number of processors. Number of experiments: 10.

7.5 Python

`Python` is an open-source, interpreted, general purpose language that has become popular among many scientists due to its elegance, readability, and flexibility. It is also a very easy language to learn for those without previous experience in coding in general domain languages.

As was the case with `R`, there are multiple modules to perform parallel computing in `Python`. In this section, we will review the `Parallel` function from the `joblib` module, which is a `Map`-like function.[19] Thus, the reader will notice the similarities to previous sections that used `Map`-type functions.

First, we import the `joblib` and `multiprocessing` modules:

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

Next, we define a data structure that contains the parameters and state variables, which is going to be one of the inputs of the `Map`-type function:

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

In `Python`, the variables declared are globally observable and modifiable by the workers in the parallel pool, so there is no need to specify their scope when defining them.

Next, we define the function `Value` that computes the maximum given some values of the states:

```
def Value(states):
   nx = states.nx
   age = states.age
   # ...
   VV = math.pow(-10, 3)
   for ixp in range(0,nx):
```

[19] We use `Python 2.7.6`. `joblib` provides lightweight pipelining in `Python` and, thus, is a good choice for our exercise. For a more general description of parallelization in `Python`, see Gorelick and Ozsvald (2014).

```
        # ...
    return[VV];
```

The function `Parallel`, from the `joblib` package, receives as inputs the number of workers of the parallel pool, the function `Value`, and a list of elements of the type `modelState`, one for each combination of values of the state variables. This function returns the function `Value` evaluated at each of the state variables. This parallelization is done at every age $t \in \{1, \ldots, T\}$:

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

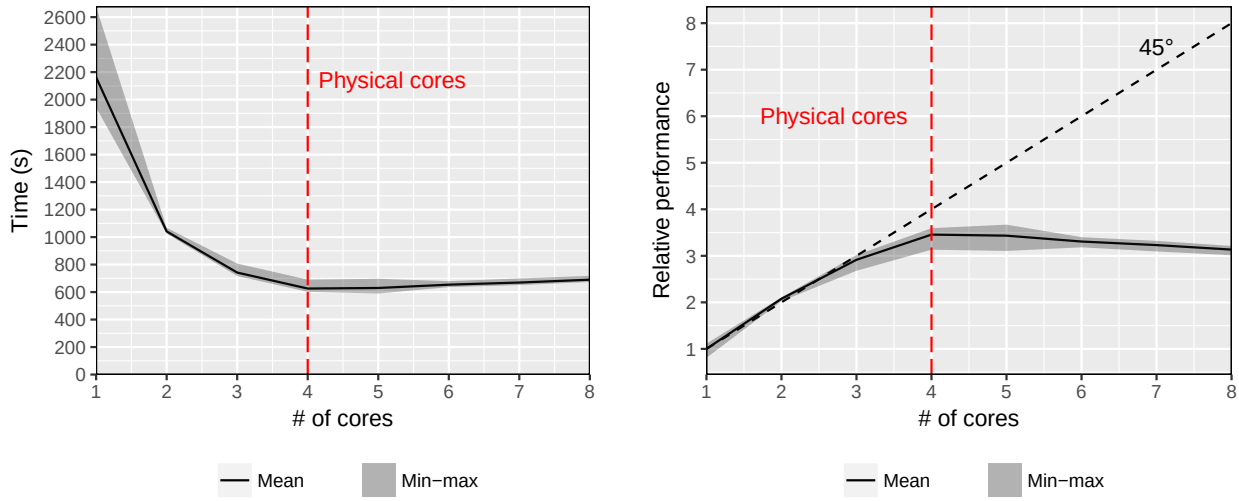
Finally, we construct the value function by storing the values of `results` in a matrix `V`:

```
for age in reversed(range(0,T)):
    results = Parallel(n_jobs=cores)(delayed(value_func)
        (modelState(ind,ne, ...)) for ind in range(0,nx*ne))
    for ind in range(0,nx*ne):
        ix = int(math.floor(ind/ne));
        ie = int(math.floor(ind%ne));
        V[age, ix, ie] = results[ind][0];
```

The results of the computations with `Python` are displayed in Figures 12a and 12b. As in `R`, total computing time with one core is extremely slow. It takes almost 2,000 seconds, on average, to compute the life-cycle model. However, parallel gains turn out to be very good, as increasing the number of cores up to 4 decreases computing time to almost a third. Increasing the number of cores beyond 4 does not generate any time improvements.

When compared to `Julia`, `Matlab`, and `R`, the performance gains of using 4 computing cores with the `Parallel` function in `Python` are the largest. However, given its absolute performance, it is still not a desirable language to compute models such as the one presented in this paper.

`Python` can be accelerated with tools such as `Numba`. Dealing with them here, however, requires mixing both parallelization and acceleration in `Python`, complicating our expository goals. See, for more information, Gorelick and Ozsvald (2014).



(a) Computing time (s)

(b) Performance gains.

Figure 12: Results in `Python` with different number of processors. Number of experiments: 20.

7.6 C++ and OpenMP

So far, we have dealt with scripting languages such as `Julia`, `Matlab`, `R`, and `Python` that allow for fast parallelization using either built-in directives or additional packages. This approach had the advantage of simplicity: minor changes to the serial code deliver programs that can be run in parallel. Thus, these scripting languages are great ways to start working on parallel or to solve mid-size problems. On the other hand, our results also show the limitations of the approach: scaling up the problem in terms of processors is difficult. And even with parallel processing, a researcher might still need extra speed from each processor.

A more powerful, but also more cumbersome, approach is to switch to a compiled language and take advantage of the tools available for parallel programming in that language. `C++` is a natural choice of a compiled language because of its speed, robustness, flexibility, industry-strength capabilities, the existence of outstanding open-source compilers such as `GCC` (indeed, for all the experiments with `C++` in this paper, we will use `GCC 4.8.5`), excellent teaching material, and broad user base.[20]

One of the easiest tools to implement parallel computing in `C++` is `OpenMP` (Open Multi-

[20]Future standards of `C++`, such as `C++20`, plan to include executors as a building block for asynchronous, concurrent, and parallel work. Parallel programming in `C++`, then, might look closer to the code of the scripting languages. In comparison, explicitly parallel languages such as `Unified Parallel C`, μ`C++`, `Charm++`, `Chapel`, and `X10` have never left the stage of small experimental languages, and we will not cover them here. `High Performance Fortran` and `Coarray Fortran` have suffered from the demise of `Fortran` outside a small niche of specialized users, dealing more often than not with legacy code from previous decades.

Processing). `OpenMP` is an open-source application programming interface (API) specification for parallel programming in `C`, `C++`, and `Fortran` in multiprocessor/core, shared-memory machines (wrappers for `OpenMP` exist for other languages, including `Python` and `Mex` files in `Matlab`). `OpenMP` was released in 1997 and its current version, as of April 2018, is 4.5. `OpenMP` is managed by a review board composed of major IT firms and it is implemented by the common compilers such as `GCC` and supported by standard integrated development environments (IDEs) such as `Eclipse`.

`OpenMP` can be learned quickly as it is easy to code, and yet rather powerful. For example, most implementations of `OpenMP` (although not the standard!) allow for nested parallelization and dynamic thread changes and advanced developers can take advantage of sophisticated heap and stack allocations. At the same time, most problems in economics only require a reduced set of instructions from the API. However, `OpenMP` is limited to shared-memory machines, such as personal computers or certain computer clusters, and the coder needs to worry about contention and cache coherence.[21]

`OpenMP` follows a multithreading "fork-join" paradigm: it executes the code serially until a directive for parallel computing is found in the code. At this point, the part of the code that can be computed in parallel is executed using multiple threads, and communication happens between the master thread and the rest of the workers in the parallel pool. After the parallel region is executed, the rest of the code is computed serially. Figure 13 illustrates the execution of an `OpenMP` code. This suggests that a good rule of thumb for `OpenMP` is to employ one thread per core.

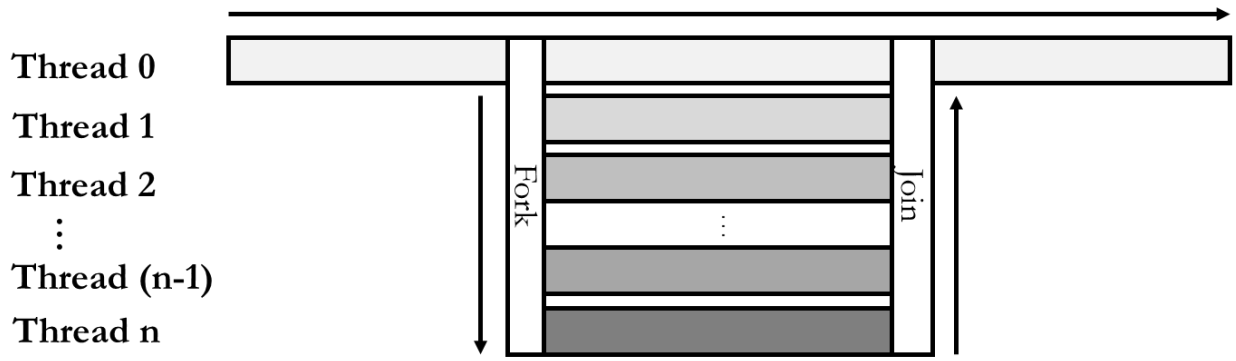


Figure 13: `OpenMP` computing

The "fork-join" scheme, however, means that it is the job of the user to remove possible

[21] See `http://www.openmp.org/` for the complete documentation and tutorials. Particularly instructive is the tutorial at `https://computing.llnl.gov/tutorials/openMP/`. A more exhaustive treatment is Chapman, Jost, and Pas (2007).

dependencies across threads (for instance, variable values in thread n required in thread $n - j$) and synchronize data. To avoid race conditions (some variables being computed too early or too late), the developer can impose fence conditions and/or make some data private to the thread, but those choices may cause performance to deteriorate. Synchronization is expensive and loops suffer from time overheads.

We implement `OpenMP` as follows. First, before compilation, we must set the environmental variable `OMP_NUM_THREADS` equal to the number of cores used in the parallel computation of the code. In a `Linux/Unix` computer, this can be done with a `Bash` shell by opening the terminal window and typing:

```
export OMP_NUM_THREADS=32
```

Second, at compilation time, the user must add the specific flag, depending on the compiler. These flags tell the compiler to generate explicitly threaded code when it encounters `OpenMP` directives. With the `GCC` compiler, the compilation flag is `-fopenmp`. With the `PGI` compiler, the required flag is `-ta=multicore`. Our particular example compiles as:

```
g++ Cpp_main_OpenMP.cpp -fopenmp -o -O3 Cpp_main
```

In the absence of this flag, the code will be compiled as regular `C++` code, always a good first step to debug and optimize the code before running it in parallel (perhaps with a less challenging computing task, such as smaller grids or more lenient convergence tolerances). Note that we also use the `-O3` flag to generate optimized code. This flag can be omitted in a first exercise while we debug and test the code, but, in "production" code, it accelerates the performance of the code considerably.

Inside the code, we include the header `omp.h` to load the relevant library:

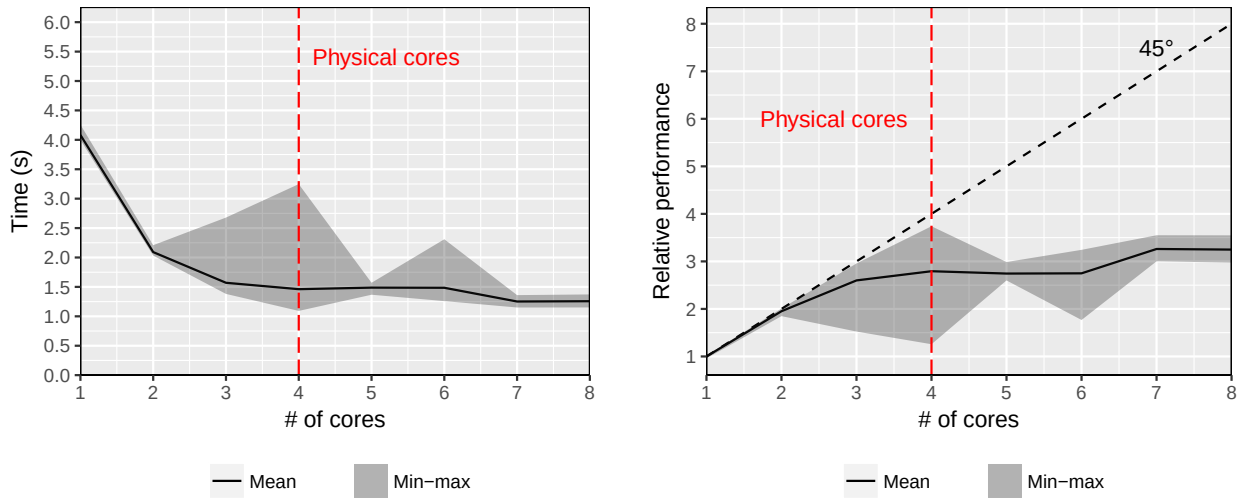
```
#include <omp.h>
```

Then, we add the directive `#pragma omp parallel for` right before each `for` loop we want to parallelize. This directive has multiple options, and has to be accompanied by the variables that are private to each worker and those that are shared. This is done using the `shared(...)` and `private(...)` directives:

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

The programmer can specify additional options for the parallelization, such as explicitly setting barriers that synchronize every thread, setting access to shared-memory objects by workers, dividing tasks between the master thread and the workers, setting private threads for specific workers, and so on.

Given the simplicity of implementing parallel computing in `OpenMP`, it is worth using it even for small tasks. First, as opposed to `Julia` and `Matlab`, the time taken to initialize the parallel pool is negligible, and the code is executed almost immediately. Second, the coding time needed to compute the code in parallel is relatively low, as only a couple of lines of code are needed. Third, as mentioned above, you can always compile your code without the `-fopenmp` flag to run the code serially to compare the results obtained with and without parallelization and debug them.



(a) Computing time (s)

(b) Performance gains.

Figure 14: Results in `C++` with `OpenMP` with different number of processors. Number of experiments: 20.

Total computing times in `OpenMP` with a different number of cores are presented in Figure 14a. As can be observed, the total time with only one core is around 4 seconds, considerably better than `Julia` in Section 7.1 and orders of magnitude better than `Matlab`, `R`, and `Python`. This is not a surprise given the advantages of compiled languages over scripting ones. See, for similar results, Aruoba and Fernández-Villaverde (2015). It is worth recalling that the `C++` code for these computations was compiled with the `-O3` optimization flag. In its absence, `C++`'s performance deteriorates to nearly 20 seconds (see the Appendix for details about the

consequences of different optimization options).[22]

Figure 14b illustrates the performance gains with a different number of cores. The performance grows almost linearly up to the third core; using 3 cores improves computation speed by a factor of more than 2.5. Although there seems to be no performance gain in adding the fourth core on average, the best case of the experiments made shows that it is possible to achieve an almost-linear gain. In the best experiment, using 4 cores improves computation speed by a factor of more than 3.5.[23] Also, in contrast to the scripting languages, there are gains from adding virtual cores beyond the number of physical cores. When computing the problem with 8 cores, the average computing time falls by more than a factor of 3.

7.7 Rcpp and OpenMP

Having described the relatively poor performance of parallel computing in `R` and the great gains in `C++` at a relatively low cost, it is worth asking whether it is possible to perform the computations in `R` by creating a `C++` function that uses `OpenMP`. Fortunately, the `Rcpp` package in `R` allows the user to create functions in `C++`. These functions can use `OpenMP` to perform the computations in parallel.[24]

All the documentation of the `Rcpp` package is available on the Rcpp for Seamless R and C++ Integration website. To use `Rcpp`, the user must first install the package and import the library from within `R`:

```
install.packages("Rcpp")
library("Rcpp")
```

The user must write the `C++` function with the appropriate `C++` directives, as described in Section 7.6:

[22] Compilers for `C++` such as `GCC` and `Intel` have options for automatic parallelization (see https://gcc.gnu.org/wiki/AutoParInGCC and https://software.intel.com/en-us/node/522532) that try to include `OpenMP`-like directives without any direct intervention by the programmer. Our experiences with automatic parallelization over the years and in different applications have been disappointing.

[23] The relatively high variance of the experiments is due to the short run time of the solution of the model in `C++`. A fix to avoid this variance would be to increase the number of grid points. But then, the computation of `R` and `Python` would take too long. Since the primary goal of this paper is to illustrate how each parallelization method works –not to measure their time performance with utmost accuracy –we keep the experiments the way they are.

[24] Although other packages allow parallel computing with `Rcpp`, such as the `RcppParallel`, their implementation costs are higher. As described in the last section, with `OpenMP`, parallel computing can be achieved by adding only a couple of lines of code. Also, a similar mixed-programming approach can be implemented with `Mex` files in `Matlab`. Our experience is that `Rcpp` is much easier and efficient than `Mex` files. See, for instance, the code and results in Aruoba and Fernández-Villaverde (2015).

```
#include <omp.h>
// ...
   #pragma omp parallel for shared(...) private(...)
   for(int ix = 0; ix<nx; ix++){
      //...
   }
```

Before importing the function with `Rcpp`, we must explicitly state which functions in the `C++` code are to be imported to `R`, by adding the following code on the line before the function `C++` code:

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

There are three options to set the number of workers on the parallel pool. The first one is to set the environmental variable `OMP_NUM_THREADS` to the desired number of workers directly from the terminal. In a `Linux/Unix` machine, this can be done in a `.bash_profile` file or from the terminal:

```
export OMP_NUM_THREADS=32
```

The second option is to include the following code right before the first parallel region of the `C++` code:

```
omp_set_num_threads(n)
```

The third option is to include the `num_threads(n)` option on the #`pragma` statement, where n is the number of desired workers in the parallel pool:

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

Given that the `C++` is compiled from within `R`, we must be sure that the `OpenMP` flag is added at compilation. For this, we should add the $-$`fopenmp` flag using the `Sys.setenv()` function from within `R`:

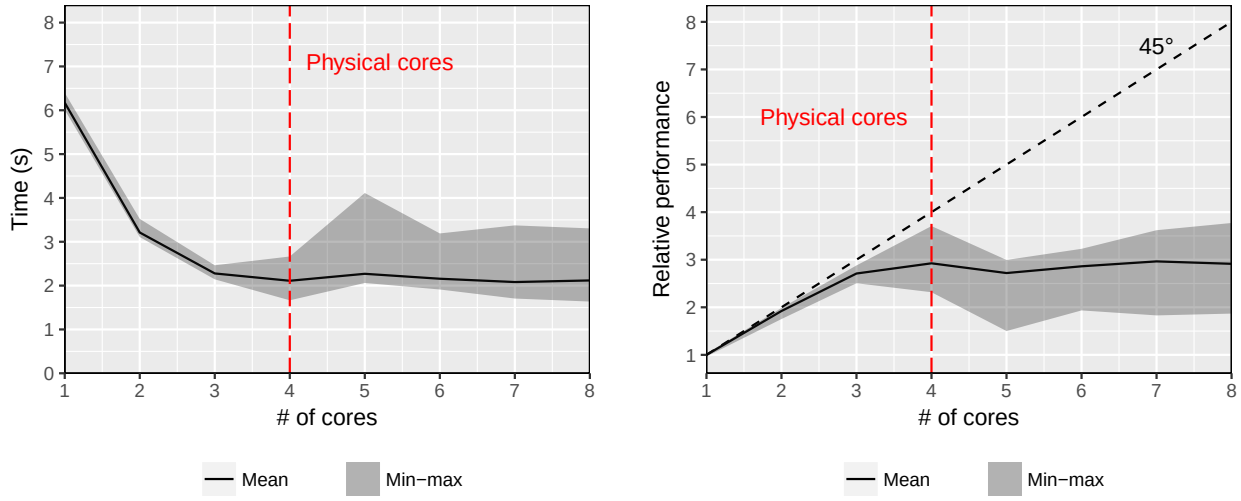
```
Sys.setenv("PKG_CXXFLAGS"=" -fopenmp")
```

Finally, the C++ code can be compiled and imported from R, so the function that computes the value function can be called from R. The function sourceCpp is used to compile the C++ code and create such a function:

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

The sample codes in R and C++ can be found on the Github repository of this paper. The user can compare how the C++ code has to be modified in order to to import it with Rcpp, and how it can be executed from R.

The results, presented in Figures 15a and 15b, are outstanding and very similar to those from C++ with OpenMP in Section 7.6. This is not surprising, as Rcpp is nothing different than compiling and executing the C++ from within R. Again, the performance improvements are almost linear up to the third core. The best-case scenario when adding the fourth core is almost linear. Adding workers beyond the fourth core generates small improvements in average performance. One difference with the C++ code is that the Rcpp code runs much faster. This is because the C++ code in Section 7.6 was compiled without any optimization flag. In contrast, Rcpp compiles the code with the -O3 optimization flag by default, which significantly improves absolute computation speeds. Section 10 presents the results when the C++ code is compiled with different optimization flags.



(a) Computing time (s)

(b) Performance gains.

Figure 15: Results in Rcpp with OpenMP with different number of processors. Number of experiments: 20.

7.8 MPI

Despite its simplicity, `OpenMP` requires, as we have repeated several times, the use of shared memory. Such a requirement limits the range of machines that we can use for parallelization. Most large computers use distributed memory as shared memory becomes unfeasible after some size limit. Also, the use of shared memory by `OpenMP` might complicate the portability of the code among machines with different memory structures. Beyond the requirement of the reproducibility of research, portability matters in parallel computing more than in serial applications because a researcher may want to develop, debug, and test the code on a computer different from the one on which the final code will be run. For example, the former tasks can be done on a regular workstation (cheap and convenient), while the latter is done on a large server (expensive and more cumbersome).

The natural alternative is to use `MPI` in `C++` (or other appropriate language; `MPI` has bindings for other programming languages, such as `Fortran`, `Python`, and `R`). `MPI` stands for message passing interface, and is a library and a set of function calls that allow communication between parallel processes executed on distributed systems, such as computer clusters and multicore computers. `MPI` ensures that the code will be portable from one platform to another with minimal changes and it allows for outstanding optimization of performance.

More concretely, we will use a popular open source implementation of `MPI`, called `Open MPI`, prepared by an extensive group of contributors and supported by many members and partners (including `AMD`, `Amazon`, `Intel`, and `NVIDIA`). This implementation conforms with the `MPI-3.1` standards and is fully documented on Open MPI's website.[25]

In `MPI`, the whole code is executed simultaneously by every thread, and communication directions are executed, which means that the programmer must explicitly give instructions for the communication that should take place between workers in the parallel pool. Figure 16 illustrates the behavior of the threads in an `MPI` program; all workers perform tasks simultaneously, and communication happens between them at different moments. For instance, thread 1 sends message m_α to thread 2, while it receives message m_γ from thread 3. You can compare Figure 16 with the "fork-join" paradigm of `OpenMP` shown in Figure 13, where some parts of the code are executed in parallel and some others serially.

[25] A basic tutorial of `MPI` can be found at Linux Magazine. More advanced tutorials are at LLNL tutorials and An MPI Tutorial. Gropp, Lusk, and Skjellum (2014) is an excellent textbook on `MPI`.

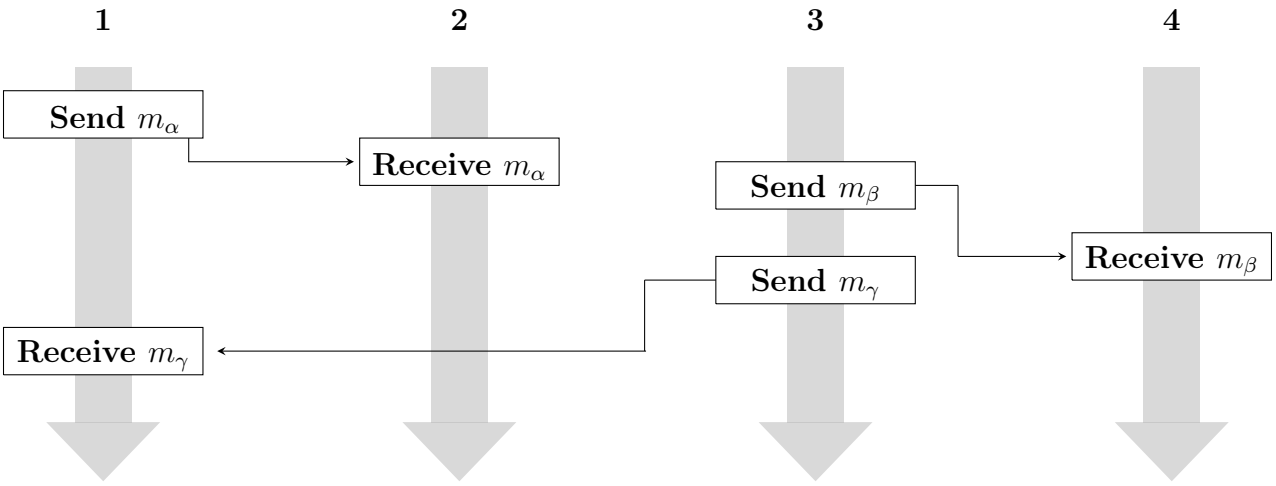


Figure 16: MPI computing

In particular, `MPI` can broadcast (transfers an array from the master to every thread), scatter (transfers different parts of an array from the master node to every thread; note the difference with broadcasting), and gather information (transfers an array from each of the threads to a single array in the master node). Figure 17 illustrates the ideas of broadcasting, scattering, and gathering of information among workers.

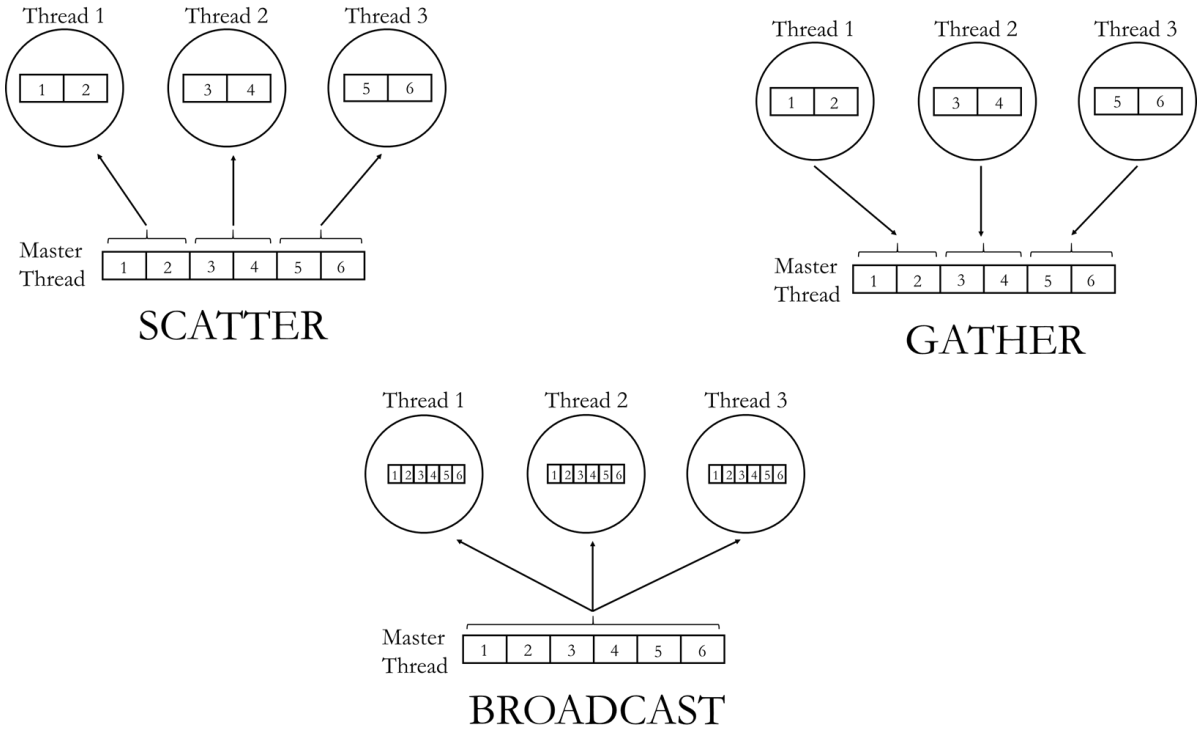


Figure 17: `MPI` Functions

This richness in capabilities means that, in comparison with the simplicity of `OpenMP`, `MPI` is much more complex. For example, `MPI` 3.1 has more than 440 functions (subroutines). This richness is required to provide the multiple directives for communication between threads to ensure that the library can handle even the most challenging numerical problems. While a description of all the `MPI` functions and their inputs would require a whole book, some of the main ones are:

- `MPI_Init`: initializes the threads.
- `MPI_Comm_size`: returns the total number of MPI processes in the specified communicator.
- `MPI_Comm_rank`: returns processor rank within a communicator.
- `MPI_Bcast`: broadcasts an array.
- `MPI_Scatter`: scatters an array.
- `MPI_Gather`: gathers an array.
- `MPI_Barrier`: stops the execution until every thread reaches that point.
- `MPI_Send`: sends messages between threads.
- `MPI_Recv`: receives messages from another thread.
- `MPI_Finalize`: finalizes the threads.

Other functions, such as `MPI_Barrier`, are used to synchronize the work of each thread in the parallel pool, as illustrated in Figure 18.

The `MPI` programmer must explicitly divide the state space between the workers in the parallel pool and assign specific tasks to each worker. This strategy is, therefore, very different from the parallel computing routines in previous subsections, in which it sufficed to state the loop that had to be parallelized or to use a `Map`-type function.

The first step to start the parallel pool is to initialize the threads with the `MPI_Init()` function:

```
MPI_Init(NULL, NULL)
```

Then, we store the total number of threads and thread id to assign jobs:

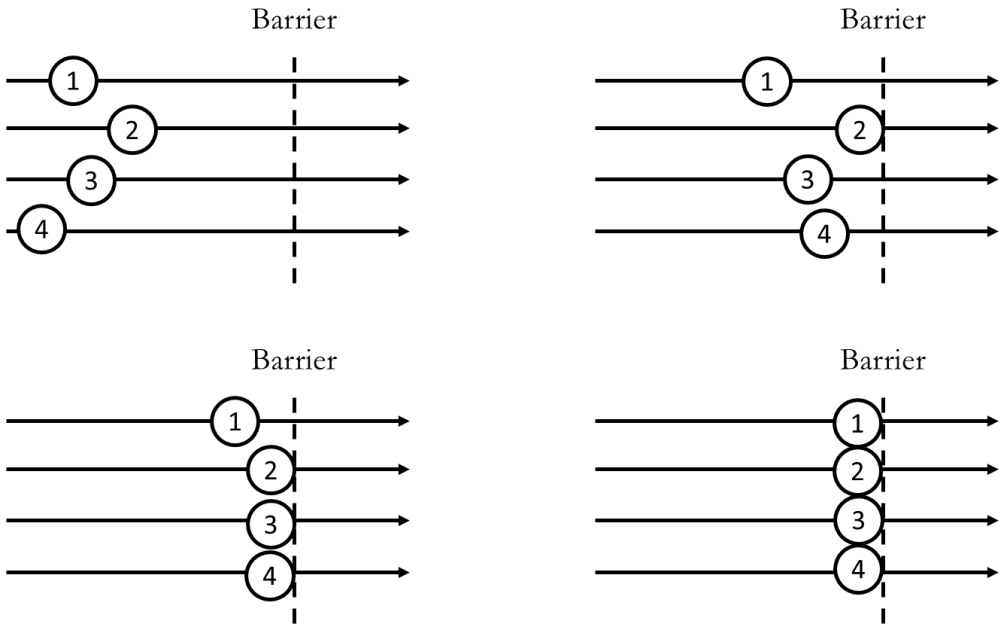


Figure 18: MPI synchronization with barrier function

```
int tid, nthreads;
MPI_Comm_rank(MPI_COMM_WORLD, &tid)
MPI_Comm_size(MPI_COMM_WORLD, &nthreads)
```

Next, we define the portion of state space that each thread must execute, according to the thread id, `tid`:

```
int loop_min = (int)(tid * ceil((float) nx*ne/nthreads))
int loop_max = (int)((tid+1) * ceil((float) nx*ne/nthreads))
```

The `MPI_Bcast` function instructs the master thread to send copies of the same information to every worker in the parallel pool. In our model, at every iteration t, we need the master thread to send a copy of the value function at $t+1$. For this reason, we must employ the `MPI_Bcast` function at the beginning of the loop:

```
MPI_Bcast(Value, T*ne*nx, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Given that every thread will execute the same task, each one for a different portion of the state space, we must make this explicit on the `for`:

```
for(int ind = loop_min; ind < loop_max; ind++){
  ix = floor(ind/ne);
  ie = ind % ne;
  // ...
}
```

The `MPI_Gatherv` function collects information from each worker in the pool and stacks it in a matrix. At the end of the `for` loop, the master thread needs to collect the respective computations of the value function made by each worker, and stack them in a matrix that stores the value function. This value function is then broadcast in the next iteration to workers, and so on:

```
MPI_Gatherv(Valp, ..., MPI_COMM_WORLD);
```

The last step is finalizing the multithreading with the function `MPI_Finalize`.

```
MPI_Finalize();
```

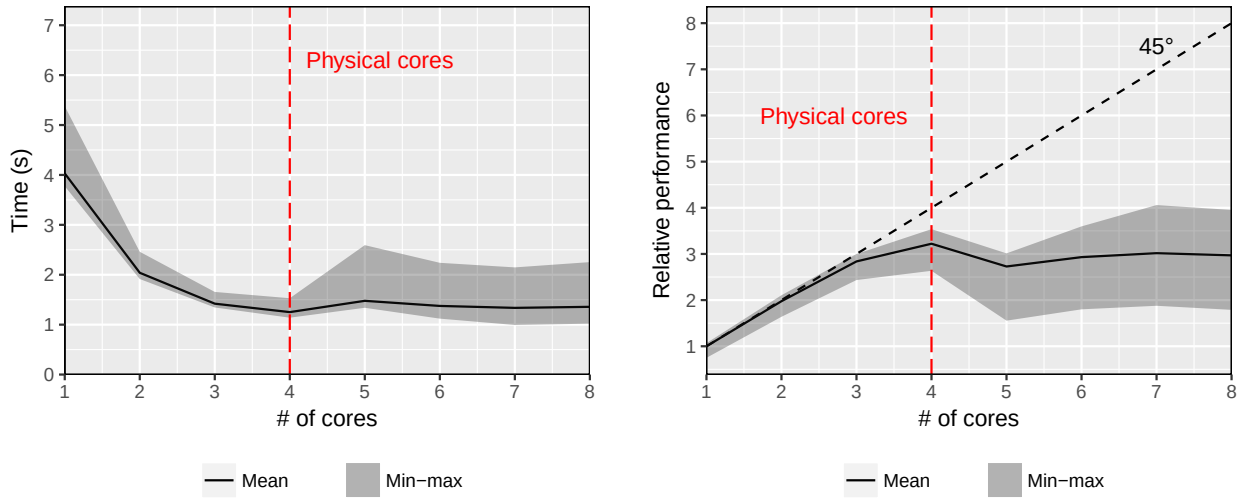
The complete code can be found on the Github repository of this paper and deserves some attention to fully understand our paragraphs above.

When applied to our life-cycle model, the performance of `MPI` with `C++` is excellent. `MPI` takes advantage of all 8 processors and computes the problem nearly four times faster than in serial mode (see Figures 19a and 19b). At the same time, the performance is not outstandingly better than `OpenMP`'s. Up to 4 cores, the performance is almost equivalent, being linear on average up to the third core, and the variance of performance gains growing with the fourth core. The only significant difference is that with MPI, the variance of performance across experiments is smaller and the average larger when using more than four threads.

But our example was not ideal for `MPI`: we run the code in a machine with shared memory and only 8 cores. If we wanted to run the code with dozens of cores, `MPI` would shine and show all its potential, while `OpenMP` would reach its limits.

Our results highlight the trade-off between `OpenMP` and `MPI`: `OpenMP` is much simpler to code and its performance is nearly as good as `MPI`, but it requires shared memory. `MPI` can handle even the most difficult problems (state-of-the-art numerical problems in natural sciences and engineering are computed with `MPI`), but it is much harder to employ.

Our educated guess is that, for most of the computations performed in economics (and the machines many researchers will have access to), economists will find `OpenMP` more attractive



(a) Computing time (s)

(b) Performance gains.

Figure 19: Results in `C++` with `MPI` with different number of processors. Number of experiments: 20.

than `MPI`, but with a sufficient number of exceptions as to justify some basic familiarity with the latter.

8 Parallelization Schemes on GPUs

Over the last ten years, parallel computation on GPUs has gained much popularity among researchers and in industry. Although initially developed for graphics rendering (thus, their name), GPUs are particularly convenient for operations on matrices, data analysis, and machine learning. The reason is that tasks involved in these algorithms require operations with array manipulation similar to the ones for which GPUs are optimized. In these problems, the serial part of the code is run in the CPU, and the GPU is used only when the user explicitly states the part of the code to be computed in parallel. Furthermore, GPUs are comparatively cheap and come pre-installed on nearly all computers for sale nowadays. See Aldrich, Fernández-Villaverde, Gallant, and Rubio-Ramírez (2011) and Aldrich (2014) for detailed explanations of how GPUs work and for the application of their computational power to economics.

The advantage of GPU computing is that, unlike CPUs, which have at most a couple of dozen processing cores, GPUs have thousands of computing cores. Table 1 presents the number of computing cores and memory of some of the available desktop `NVIDIA` GPUs (`NVIDIA` is the leading vendor of GPUs). And just as you can stack several CPUs in one machine, it

GPU	CUDA Cores	Memory
GeForce GTX TITAN Z	5760	12 GB
NVIDIA TITAN Xp	3840	12 GB
GeForce GTX 1080 Ti	3584	11 GB
GeForce GTX 690	3072	4 GB
GeForce GTX 780 Ti	2880	3 GB
GeForce GTX 1080	2560	8 GB
GeForce GTX 780	2304	3 GB
GeForce GTX 980	2048	4 GB
GeForce GTX 1070	1920	8 GB
GeForce GTX 970	1664	4 GB
GeForce GTX 770	1536	2 GB

Table 1: Number of CUDA cores and memory in NVIDIA GPUs.

is also possible to install several GPUs in the same computer, which increases performance even more (although it also complicates programming). Thus, through GPU computing, the user can achieve much larger speed gains than with CPU parallel computing. Depending on the type of computations performed and on the memory handling and communication management between the CPU and the GPU, the performance gains range from 10 times as fast as the serial code, up to hundreds of times faster. Examples of applications in economics well-suited for GPUs include dense linear algebra, value function iteration, Monte Carlos, and sequential Monte Carlos.

But the thousands of cores of a GPU come at a cost: GPUs devote more transistors to arithmetic operations rather than flow control (i.e., for branch prediction, as when you execute an "if" conditional) and data caching. Thus, not all applications are well-suited for computation on a GPUs (or the corresponding algorithms may require some considerable rewriting). For instance, applications involving searching or sorting of vectors typically require too much branching for effective GPU computing. Also, memory management can be difficult.

Figure 20 illustrates the architecture of an NVIDIA GPU.[26] The CPU is referred to as the "host," and the GPU is referred to as the "device." The device is composed of grids, where each grid is composed of two-dimensional blocks. The dimensions of the grids and blocks are defined by the user, with the only restriction being that each block can handle up

[26]Source for the figure: NVIDIA CUDA Compute Unified Device Architecture, Programming Guide 1.0, http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

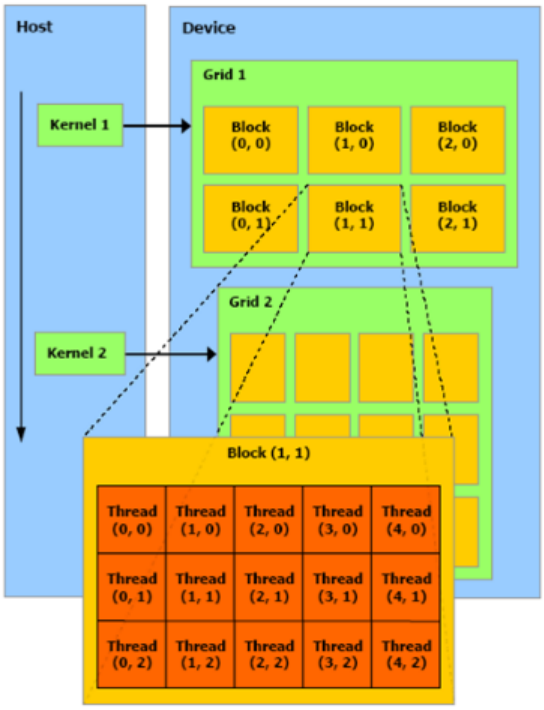


Figure 20: CUDA GPU architecture.

to 512 threads. This limitation means that the programmer must carefully divide the tasks such that each of these blocks does not have more than 512 jobs. The user must also take care in regard to communication and memory handling. More concretely, at every parallel region of the code, the user must explicitly transfer variables from the CPU to the GPU by appropriately managing the memory available in the GPU.

In this section, we will present two approaches to parallelization in GPUs: `CUDA`, in Subsection 8.1, and `OpenACC`, in Subsection 8.2. `CUDA` is the traditional environment for coding in GPUs, with plenty of examples and a wide user base. The drawback is that `CUDA` requires the codes to include many low-level instructions to communicate between the CPU and GPU and for memory management.[27] `OpenACC` is a more recent, but rather promising, avenue to simplify computation in GPUs. `OpenACC` requires much less effort by the coder and can work across different platforms and mixed processing environments, but at the potential cost of slower code. We will skip a description of `OpenCL`, an open standard for computing in heterogeneous platforms. While `OpenCL` has the advantage of not being tied to `NVIDIA` GPUs, its user base is thinner and less dynamic, and its performance is generally inferior to

[27]This disadvantage can partially be addressed by the use of libraries such as `ArrayFire` (https://arrayfire.com/), which allow high-level instructions and function calls. In our assessment, however, going with `OpenACC` seems at this moment a better idea for the long run.

CUDA's. Similarly, one can program in the GPU with the Matlab parallel toolbox and in R with the gpuR package. However, if a researcher needs enough speed as to require the use of a GPU, perhaps moving first to a language more efficient than Matlab or R before diving into GPUs seems a wiser course of action.[28]

8.1 CUDA

CUDA, which stands for Computer Unified Device Architecture, is a parallel computing model with extensions for C, C++, Fortran, and Python, among others, that allows the implementation of parallel computing on the GPU of a computer. The CUDA model was designed by NVIDIA. It was first released in 2007 and the current stable version, as of April 2018, is 9.2, with an extensive range of applications and a deep user base.[29]

The first step for parallel computing in CUDA is to explicitly define the functions that are going to be executed from within the GPU, as opposed to those that are executed from the CPU.[30] The former should be preceded by the directive __device__, while the latter should be preceded by __global__:

```
// 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, ...){
  // ...
}
```

Next, the user must allocate memory in the device (GPU) for the variables that are going to be transferred from the host (CPU) at the beginning of the parallel computation.

[28] The repository https://github.com/JuliaGPU lists several alternatives for GPU computing in Julia, but the packages are still far from easy to employ by less experienced users.

[29] An excellent exposition of CUDA is offered by Kirk and Hwu (2014). Some tutorials and links are https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf, https://developer.nvidia.com/cuda-education, and https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/.

[30] The reader will benefit much from following the next pages while checking the code posted on the Github repository.

Analogous to the `malloc` function in `C++`, the `cudaMalloc` allocates the required memory in the GPU. This function has as input the name of the variable to be copied and its size, which is defined as an element of type `size_t`. In our case, we are going to initialize the value function in the GPU with the name `V`:

```
float *V;
size_t sizeV = T*ne*nx*sizeof(float);
cudaMalloc((void**) &V, sizeV);
```

Since we now have to handle memory in the CPU and memory in the GPU, we must assign different names to objects that live in both. For example, the value function is going to be kept in the host memory, but will be transferred to the device and updated at each iteration. Hence, we must assign a name to the value function in the host, and a different name for the value function in the device. With the `cudaMalloc` function above, we named the value function living in the device as `V`.

Now we have to initialize the value function that lives in the host. For ease of reference, we will precede the name of variables in the host with `h`, so the value function in the host is called `hV`:

```
float *hV;
hV = (float *)malloc(sizeV);
```

Once memory is allocated in the GPU, the user must transfer from host to device the variables that need to be accessed and/or modified in the parallel computations. The function used to transfer information between host and device is `cudaMemcpy`, which uses as input the object in the device memory that was already initialized (`V`), the object in the host memory that is going to be copied to the device (`hV`), the size of the object (`sizeV`) and the reserved word `cudaMemcpyHostToDevice` that specifies the direction of the transfer:

```
cudaMemcpy(V, hV, sizeV, cudaMemcpyHostToDevice);
```

The user should initialize and transfer any other variables that will be used in the device for every computation in the same way.

Next, we must take care of defining the dimensions of the blocks and grids for parallelization in the GPU. Each thread in the GPU will have a 5-dimensional "coordinate" given by the number of its grid, the 2-dimensional coordinate of its block, and its 2-dimensional coordinate within the block. This 5-dimensional "coordinate" facilitates parallelization. For

example, if the user only wants to parallelize along one dimension –such as the capital x dimension– she can define only one grid and one block of dimensions $1 \times n_x$. In this way, the thread i will be assigned the task of computing the value function at the i-th point in the grid for capital. Here, we must be careful not to violate the restriction that any given block can have at most 512 threads, which means that $n_x \leq 512$ must hold.

Similarly, if the user wants to parallelize across 2 dimensions –say, the productivity e and capital x dimensions– she can define one grid with one block of dimensions $n_e \times n_x$. Here, the thread with coordinates (i, j) inside the block will be assigned the task of computing the value function at the i-th entry of the productivity grid and the j-th entry of the capital grid. Again, we must ensure that $n_e \cdot n_x \leq 512$.

The dimensions of blocks and grids are defined as objects of type `dim3`, using the `CUDA` constructors `dimBlock` and `dimGrid`. Remember not to assign more than 512 threads per grid!

```
int block_height = 20;
int block_width = ne;
dim3 dimBlock(block_height, block_width);
dim3 dimGrid(nx/block_size, 1);
```

Given that it is very likely that the restriction $n_x \cdot n_e \leq 512$ will be violated, we define only one grid with multiple blocks. Each block has a width of size n_e and a height of size 20. We define as many blocks as necessary to cover all points in the productivity and capital grids.

As already explained, the job assigned to each thread should be a function of its coordinates inside the 5-dimensional structure defined by the user within the device. The variables `blockIdx.x`, `blockIdx.y`, `blockIdx.z`, `blockDim.x`, ..., are used to access the coordinates of each thread. Given these coordinates, the user should define the tasks assigned to every thread. For example, in the case where we want to parallelize across the productivity and capital dimensions, the function that computes the value function will start as follows:

```
__global__ float value(parameters params, float* V, ...){
  int ix = blockIdx.x * blockDim.x + threadIdx.x;
  int ie = threadIdx.y;
  // ...
}
```

This function will compute the value function at the `ix` point in the capital grid and `ie` point in the productivity grid. Here, `ie` is simply given by the y-coordinate inside the block, `threadIdx.y`, but `ix` is given both by the coordinate of the block, `blockIdx.x`, and the x-coordinate within the block, `threadIdx.x`.

Finally, to compute the value function in parallel, the user can call the function `value`, explicitly stating the dimensions defined above within `<<<...>>>`. After calling the function, we must check that all workers finish their job before iterating to $t-1$. For this, we use the function `cudaDeviceSynchronize` right after calling the function `value`:

```
value<<<dimGrid, dimBlock>>>(params, V, ...);
cudaDeviceSynchronize();
```

The function `value` is executed in the device, but is called from the host, so it is defined as `__global__`. Only functions that are called by the device, within parallel executions, should be preceded by `__device__`.

The function `value` is called at every iteration $t \in \{1, \ldots, T\}$. Once the value function is computed for a given t, the value function must be copied back to the CPU, again using the `cudaMemcpy` function. However, the last input must be set as `cudaMemcpyDeviceToHost`, to specify that `V` must be copied from device to host:

```
cudaMemcpy(hV, V, sizeV, cudaMemcpyDeviceToHost);
```

Finally, the function `cudaFree` is used to free the memory in the device at the end of the computation:

```
cudaFree(V);
```

We run our `CUDA` code on the same machine as all the previous experiments. The computer is equipped with a `GeForce GTX 860M` GPU with 640 CUDA cores and 4GiB of memory. The average run time in 20 experiments is 1.86 seconds, slightly better, for instance, than `C++-OpenMP` in 8 cores. As in the case with `MPI`, our problem is not complex enough to show all the power of `CUDA`, but the results give, at least, a flavor of how to use it and its potentialities.

8.2 OpenACC

`OpenACC` is a model designed to allow parallel programming across different computer architectures with minimum effort by the developer. It is built around a very simple set of directives, very similar in design to `OpenMP`. To implement parallel computing, it suffices to add a couple of lines of code, or directives, before a section of the code the researcher wants to parallelize. Moreover, it is explicitly designed to be portable across different computer architectures. This means that 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. In this way, `OpenACC` can be used to perform the same task as `OpenMP` and `MPI`, with a similar performance, but can also be used to perform GPU computing when available. Given the similarities to its implementation in `OpenMP`, but the clear advantage of allowing the user to execute the code in the GPU in addition to the CPU, `OpenACC` represents a more powerful tool, although one still in relatively early stages of development. There are only a couple of books written about it, for example, Farber (2017) and Chandrasekaran and Juckeland (2017), neither of which is fully satisfactory.

`OpenACC`, like `OpenMP`, executes the code serially using the `fork-join` paradigm, until a directive for parallel computing is found. The section of the code that is parallelizable is then computed with the use of multiple CPU (or GPU) threads. 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. After the parallel region is executed, the code is serially executed until a new directive is found. As was the case with `OpenMP`, Figure 13 illustrates code execution in `OpenACC`.

The OpenACC website describes multiple compilers, profilers, and debuggers for `OpenACC`. We use the `PGI Community Edition` compiler, which is a free release of `PGI`. The reason for using `PGI` rather than `GCC`, as in Section 7.6 with `OpenMP`, is that 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.

To implement the parallelization, we must include the directive `#pragma acc parallel loop` right before the `for` loop:

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

This directive has multiple options, among which the `private(...)` option allows the user to specify which variables are private to each worker. The programmer can also explicitly set barriers that synchronize every thread, synchronize the memory in the CPU and the GPU, and so on. If we want to perform parallelization only with the CPU, in which the memory is shared by workers, this directive and any options included are all that is required. We can go on to compile the code, as explained below, and execute the program. If we want to execute the code with the GPU, a data movement must be specified between the host memory (CPU) and the device memory (GPU), given that the CPU and GPU do not share memory.

Once compiled, the `OpenACC` code works as follows. The program is executed in serial by the CPU until the first parallelization directive is encountered in the code. At this point, the variables and objects required for each computation must be copied from the host memory (CPU) to the device memory (GPU), before executing the code in parallel. Once the parallel region is executed, for the program to work correctly, the objects that were stored in the device memory must be copied back to the host memory. After that, the program is executed by the CPU in serial, until another parallel directive is encountered. For this reason, the user must specify the points at which data movements should occur between the CPU and the GPU.

The main directive for data movements is `#pragma acc data copy(...)`, where the variables that should be copied back and forth from CPU to GPU are specified inside the `copy(...)` option. There are different options for data movements, such as `copyin(...)`, which only moves data from CPU to GPU, `copyout(...)`, which only moves data from GPU to CPU, and so on. This directive should be specified right before the `#pragma acc parallel loop` that declares the beginning of a parallel region:

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

If we include the directive `#pragma acc data copy(...)`, but compile the code to be executed only with the CPU, this data movement directive is ignored, a convenient feature for debugging and testing.

At this point, by choosing the appropriate compilation flag, the user can choose to 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, the user wants 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
```

Importantly, we do not need to modify the code to parallelize with a multicore CPU or with a GPU. This makes `OpenACC` a very portable parallel computing extension, and reduces significantly the barriers to GPU computing.

When we run the code 20 times on the same computer and GPU described in previous sections, we measure an average run time of 1.42 seconds, a record among all the options presented so far. This result is impressive: given how easy it is to code in `OpenACC` and the portability of the code among machines and processing units, `OpenACC` seems to be one of the best available avenues for parallelization.

9 Beyond CPUs and GPUs

There are several alternatives for parallel computation beyond using CPUs and GPUs. First, there is an intermediate option, the bootable host processors `Intel Xeon Phi`™, which can pack, in the current generation, up to 72 cores. In this guide, it is not necessary to cover these processors, since `MPI` and `OpenACC` can easily handle them with straightforward adaptations of the material presented in previous pages.

Similarly, we will not discuss how to use tensor processing units (TPUs) and field-programmable gate arrays (FPGA). The former have just come on the market and are still in an early stage of development, and we are not aware of any application in economics that uses them. The latter are probably not efficient for most problems in economics due to the hardware development cost. Nevertheless, for examples of this approach in finance, where the cost considerations might be different, see the papers in Schryver (2015).

10 Conclusion

As a summary of our discussion, Figure 21 illustrates the absolute performance of the computation with each of the programming languages, according to the number of threads used in the parallel pool (for codes in the GPU, we report a constant line, since, by construction, we do not change the number of cores). The computation times for `Julia` are calculated with the `pmap` function. `Matlab`, `Python`, and `R` are not included, because they take much longer than the rest of the languages.

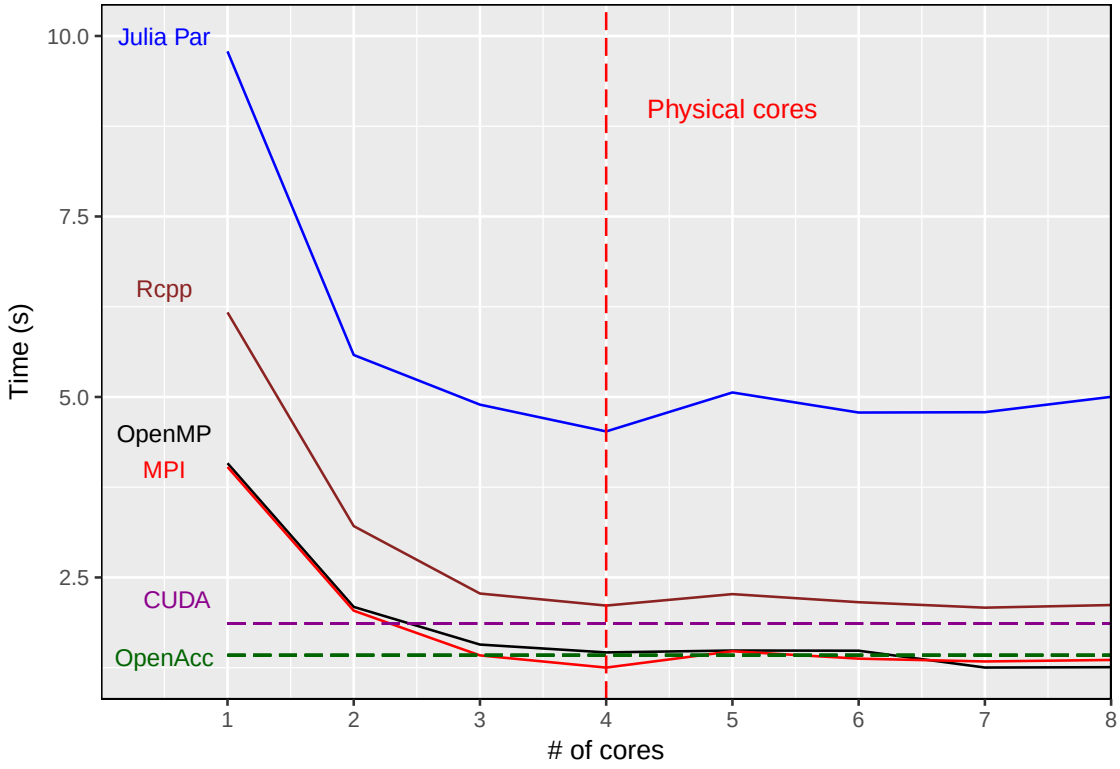


Figure 21: Computing time (s) by language with different number of processors.

The best performance is attained when computing the model with `CUDA` and `OpenACC` (when the latter performs the computations with the GPU). When looking only at parallel CPU computing, the best performance is obtained with `C++-OpenMP` and `C++-MPI`. `Rcpp` does an excellent job, nearly matching the speed of `C++-OpenMP` and `C++-MPI`. `Julia` falls somewhat behind, pointing perhaps to the need to reconfigure our code to take advantage of the peculiarities of the language. Our results show also that employing `Matlab`, `Python`, and `R` for this class of problems is not recommended and that parallelization does not fix the speed problem of these languages.

We conclude our guide by reiterating that parallelization offers a fantastic avenue to improve the performance of computations in economics. Furthermore, parallelization is relatively easy to implement once one understands the basic concepts. Even in `Julia`, where parallelization is trivial, and with a standard laptop, one can cut in half the run time of a realistic model. If the researcher is willing to jump to `OpenMP` and `OpenACC`, the gains are potentially much larger, especially if she has access to a more powerful computer or cloud services.

Appendix

To understand the role that optimization flags play in the performance of the `C++` codes, we plot in Figure 22 the run times for a different number of processors with no optimization flags and with three flags (from less aggressive, -O1, to most aggressive, -O3) in the case of `C++-OpenMP`.

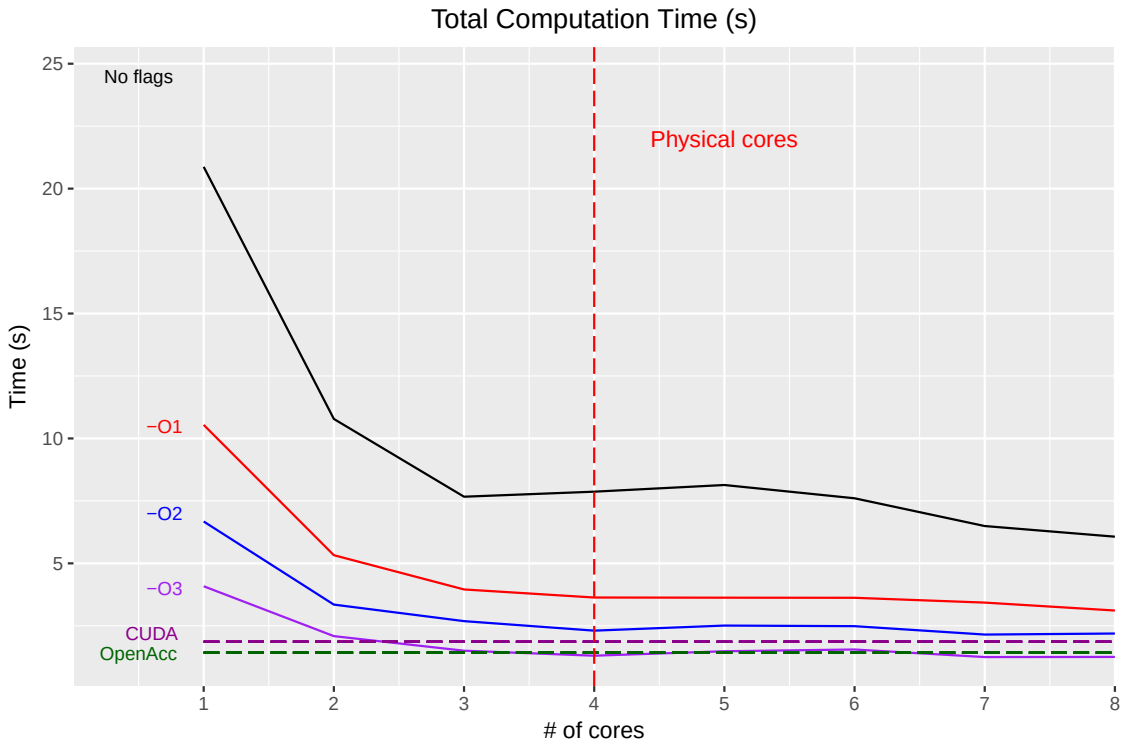


Figure 22: Computing time(s) in `C++` with different number of processors and optimization flags.

References

Aldrich, E. M. (2014): "GPU Computing in Economics," in *Handbook of Computational Economics Vol. 3*, ed. by K. Schmedders, and K. L. Judd, vol. 3 of *Handbook of Computational Economics*, pp. 557 – 598. Elsevier.

Aldrich, E. M., J. Fernández-Villaverde, A. R. Gallant, and J. F. Rubio-Ramírez (2011): "Tapping the Supercomputer under your Desk: Solving Dynamic Equilibrium Models with Graphics Processors," *Journal of Economic Dynamics and Control*, 35(3), 386–393.

Aruoba, S. B., and J. Fernández-Villaverde (2015): "A Comparison of Programming Languages in Macroeconomics," *Journal of Economic Dynamics and Control*, 58, 265 – 273.

Ben-Porath, Y. (1967): "The Production of Human Capital and the Life Cycle of Earnings," *Journal of Political Economy*, 75(4), 352–365.

Bertsekas, D. P. (2012): *Dynamic Programming and Optimal Control, Vol. 2*. Athena Scientific, 4nd edn.

Calderhead, B. (2014): "A General Construction for Parallelizing Metropolis-Hastings Algorithms," *Proceedings of the National Academy of Sciences of the United States of America*, 111(49), 17408–17413.

Chandrasekaran, S., and G. Juckeland (2017): *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley.

Chang, M. (2017): "Marriage, Portfolio Choice, and Labor Supply," *Working Paper, University of Pennsylvania*.

Chapman, B., G. Jost, and R. v. d. Pas (2007): *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press.

Cocco, J. F. (2005): "Consumption and Portfolio Choice over the Life Cycle," *Review of Financial Studies*, 18(2), 491–533.

Conesa, J. C., and D. Krueger (1999): "Social Security Reform with Heterogeneous Agents," *Review of Economic Dynamics*, 2(4), 757–795.

Devroye, L. (1986)): *Non-Uniform Random Variate Generation*. Springer-Verlag.

Farber, R. (ed.) (2017): *Parallel Programming with OpenACC*. Morgan Kaufmann.

Fernández-Villaverde, J., and D. Krueger (2011): "Consumption and Saving over the Life Cycle: How Important are Consumer Durables?," *Macroeconomic Dynamics*, 15(5), 725–770.

Fernández-Villaverde, J., and J. F. Rubio-Ramírez (2007): "Estimating Macroeconomic Models: A Likelihood Approach," *Review of Economic Studies*, 74(4), 1059–1087.

Flamm, K. (2018): "Measuring Moore's Law: Evidence from Price, Cost, and Quality Indexes," Working Paper 24553, National Bureau of Economic Research.

Gorelick, M., and I. Ozsvald (2014): *High Performance Python: Practical Performant Programming for Humans*. O'Reilly Media.

Gourieroux, C., and A. Monfort (1997): *Simulation-based Econometric Methods*. Oxford University Press.

Gropp, W., E. Lusk, and A. Skjellum (2014): *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 3rd edn.

Hager, G., and G. Wellein (2011): *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press.

Ingvar, S. (2010): "Efficient Parallelisation of Metropolis-Hastings Algorithms using a Prefetching Approach," *Computational Statistics & Data Analysis*, 54(11), 2814 – 2835.

Kirk, D. B., and W. W. Hwu (2014): *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 3rd edn.

Lin, Y. C., and L. Snyder (2008): *Principles of Parallel Programming*. Pearson/Addison Wesley.

Martino, L., V. Elvira, D. Luengo, J. Corander, and F. Louzada (2016): "Orthogonal parallel MCMC methods for sampling and optimization," *Digital Signal Processing*, 58, 64 – 84.

McCool, M. D., A. D. Robison, and J. Reinders (2012): *Structured Parallel Programming Patterns for Efficient Computation*. Elsevier/Morgan Kaufmann.

Moore, G. E. (1965): "Cramming More Components onto Integrated Circuits," *Electronics*, 38(8).

——— (1975): "Progress in Digital Integrated Electronics," *Electron Devices Meeting*, 21, 11–13.

Murray, C. J. (1997): *The Supermen: The Story of Seymour Cray and the Technical Wizards Behind the Supercomputer*. John Wiley & Sons, Inc.

Nishiyama, S., and K. Smetters (2014): "Analyzing Fiscal Policies in a Heterogeneous-Agent Overlapping-Generations Economy," in *Handbook of Computational Economics*, ed. by K. Schmedders, and K. L. Judd, vol. 3, pp. 117–160. Elsevier.

Pacheco, P. (2011): *An Introduction to Parallel Programming*. Morgan Kaufmann.

Robert, C. (2007)): *The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation*. Springer-Verlag, 2nd edn.

Roberts, G. O., A. Gelman, and W. R. Gilks (1997): "Weak Convergence and Optimal Scaling of Random Walk Metropolis Algorithms," *The Annals of Applied Probability*, 7(1), 110–120.

Rünger, G., and T. Rauber (2013): *Parallel Programming - for Multicore and Cluster Systems*. Springer, 2nd edn.

Schryver, C. d. (2015): *FPGA Based Accelerators for Financial Applications*. Springer.

Wickham, H. (2014): *Advanced R*, Chapman & Hall/CRC The R Series. Taylor & Francis.

Zarruk-Valencia, D. (2017): "Wall Street or Main Street: Who to Bail Out?," *Job market paper, University of Pennsylvania*.