Parallelization Overhead and HPC-Efficient Programming

Dongya Koh*

University of Arkansas

January 1, 2019

Abstract

An advent of high-performance computing (HPC) system enables parallel computing more handy and ubiquitous in numerically solving economic models. Nevertheless, it is surprising that very little is known about the full gain of modern computing power via parallel computing. Without knowing the parallel computer architectures, parallel computing with a large number of cores could turn out more harm than good. This study therefore shows several coding pitfalls that might cause significant slowdowns in HPC parallel computing and introduces easily-implementable HPC-efficient algorithms to prevent such slowdowns.

*This research is supported by the Arkansas High Performance Computing Center which is funded through multiple National Science Foundation grants and the Arkansas Economic Development Commission. We have also benefited from the participants at "Economic Research in High Performance Computing Environments" Workshop at Kansas City Fed.

1 Introduction

A scalable outcome (i.e., computational time) of solving a dynamic model with more than one core is possible only when a code is written in an efficient way. This is a good news for those who are solving a complex dynamic model that entails agonizing wait for the computation even with an efficient solution method. Yet, little is known about techniques to write an efficient code that achieves a speed-up through parallel computing. This study introduces a number of parallel-computing techniques to obtain a scalable outcome from efficiently solving dynamic macroeconomic models in high-performance computing (HPC) systems.

The recent development and provision of HPC, such as multi-core processors (CPUs and GPUs), clusters of a number of multi-core/multi-threading compute nodes, and cloud computing environments, enable economists to employ a number of compute elements in a concurrent way.[1] Parallel computing on HPC systems therefore becomes a hardware platform that helps speeding up the computational time and hence has recently been employed as a research tool in economics. Despite its rapid growth of computing power and of easily-implementable parallel programming, very little is known about the bottlenecks of parallel computing on HPC systems. Indeed, less experienced programmers tend to fall into coding pitfall that fails to take full advantage of HPC systems simply due to the lack of understanding in HPC architectures and parallel programming mechanisms. In this study, we explicitly demonstrate that simply parallelizing a serial program in HPC is not enough to generate a scalable outcome and that it requires more engineering effort to fine-tune the serial program to exploit the full capacity of HPC resources.

To fathom the size of inefficiency upon executing a serial code in parallel, we introduce parallelization overhead as a metric. Parallelization overhead is defined as an additional time incurred by parallelizing a chunk of tasks to multiple processors. If the total computational time of parallelizable part of program with a single core takes T seconds and if there is no overhead or latency of any kind upon parallelization, then N processors share the entire tasks to ideally complete the work in T/N seconds. This is a hypothetical case with perfect scalability. In reality, this ideal case is unreachable because extra time always incurs whenever parallelizing tasks to multiple processors. This extra additional time to parallelizing tasks is called "parallelization overhead."

The parallelization overhead in HPC mainly arises from two sources: communication overhead from memory access and data transfer and idling of workers from load imbalances.[2] The former latency is attributable to the computer architectures and memory management, while the latter is closely related to the parallelized tasks of a program. The memory access and data transfers are critical in parallel computing which may not be so critical in serial computing, since data in the main memory would be copied to a local memory of a pool of workers so that each worker can refer to the data stored in its local memory at a faster rate. As a matter of fact, this overhead arising from the memory access and data transfers varies by the distance of local memory from

[1] To prevent any confusions, we use the terms "cores", "processors", or "workers" synonymously indicating the multiple compute elements for parallel computing in HPC. We also use "CPU chips" and "sockets" interchangeably indicating a container for multiple cores.

[2] Parallelizing numerical evaluation of value/policy functions by grid points usually requires very little or almost no communication across processors until a job is done and synchronized.

the main CPU and the size of data to be transferred. This suggests that understanding internal designs and mechanics of computer system and programming languages at a certain degree is imperative to identify the sources of parallelization overhead and to write an efficient code.

We show the communication overhead from memory access and data transfers in two ways. First, we parallelize a block of works into multiple workers passing a class object instead of specific variables to be used. Passing an object that contains variables and functions as attributes is a fairly common coding style in object-oriented programming. Since the use of object-oriented programming languages such as C++ and Python is growing in the quantitative analysis of economics, we demonstrate that whether or not passing an entire object to a pool worker becomes harmful depends on the size of an object and the number of parallel workers to be employed. If the class object contains large variables that are not used in parallel processing, then passing the entire object to a large number of workers would trivially be very costly. This source of overhead also applies to GPU computing and distributed-memory computing where the local memory takes even farther distance with lower bandwidth from the main CPU than the shared-memory.

Second, we show that the right granularity—the amount of tasks assigned to each worker—depends on the number of parallel workers. It is trivial that as granularity rises total computational time and parallelization overhead both increase for any number of parallel workers. However, with a small granularity parallelization overhead becomes high when a large number of workers are used. This is because processing time of tasks by each worker is much quicker than creating a pool of workers and transferring data to and from each worker with small granularity. On the other hand, the amount of time required to create a pool of workers and to transfer data should be relatively quicker than processing time with a large granularity. This instructs us that the parallelization overhead relative to processing time by each worker becomes smaller either with small granularity and the small number of workers or with a large granularity and the large number of workers.

Another source of parallelization overhead arises from idling of workers. When each task taking different amount of time may cause idling of some workers waiting for other workers that are working on longer tasks. In our example, parallelizing by the grid points at which value/policy functions are evaluated may have some load imbalanced when the process contains root-finding and interpolation processes. This implies that eliminating such load imbalances may lead to an efficient-use of HPC resources. To get around this load imbalance problem, we suggest altering the chunk size of tasks that is assigned to each worker one at a time. One extreme is to divide the entire parallelizable work into the number of workers so that each worker takes on one big chunk of tasks. Another extreme is that one chunk contains only one task so that as a worker completes one task it asks for another task. If each task is about the same size of work that completes about the same execution time, then one big chunk of tasks per worker would be efficient. On the other hand, if there exists any load imbalances, then assigning smaller chunks of tasks would be faster. Our experiment shows that computing a typical dynamic model would be most efficient with more than one worker if the entire tasks are divided into 4-20 chunks of tasks per worker.

Fernández-Villaverde and Valencia (2018)'s practical guide to parallel computing in economics demonstrates parallel computing with different programming languages (Julia, Matlab, R, Python,

and C++) and parallelizing interfaces (OpenMP, MPI, CUDA, and OpenACC). Their work lays out a road map to parallel computation for economics research, while our work is more technically guiding the same audience how to write an efficient parallel program to make use of full capacity of HPC. Likewise, we haven't devoted even a page to examine the performance of parallel computing on GPUs because we would rather ask readers to refer to the work of Aldrich et al. (2011). Though complementary to these work, our contributions are threefold: first, we introduce a new metric of inefficiency, parallelization overhead, to measure how efficiently a program can run in HPC environment. Second, we investigate the parallelization overhead in shared-memory architectures in the first place so that we can single out two important overheads—communication overhead and idling of workers—arising as well in more complicated HPC architectures. Finally, we haven't presented any single code of a particular programming language (though we write the code in Python) since the interest of this study is to introduce general parallel-computing techniques that can easily be implemented in a program of any languages.[3]

The rest of the paper is organized as follows. In section 2, we describe a parallel-computing environment that we use for our experiment. In particular, we describe a dynamic model that we solve, a programming language to write a code, an HPC system that we employ, a solution method that we use, and a part of a program to be parallelized. In section 3, we define parallelization overhead as a metric for a code inefficiency and discuss the possible sources of the overhead. Then, in section 4, we examine how much and as to why inefficiencies may arise from a particular coding and investigate the design of HPC architectures and parallel programming. Finally, section 5 concludes.

2 Parallel Computing in HPC

For our demonstration of parallel computing on HPC environment, we need to decide an economic model, a solution method, a programming language, HPC environment, and which part of program to be parallelized. In this section, we specify the environment that we run a serial program in parallel to evaluate the performance of parallel computing.

2.1 Parallel Computing Environment

Benchmark Model: To assess the performance of parallel computing in HPC environment, we choose to numerically solve a stochastic consumption/saving life-cycle model for two reasons. First, a life-cycle model can be solved in a finite number of iterations (by age), while an infinite-horizon neoclassical growth model iterates until the convergence of value/policy functions which may vary by the choice of solution methods and numerical libraries. In addition, since the value/policy functions at the end of period T are deterministic, there is no dependency on an initial guess of value/policy functions.[4]

The benchmark model assumes that an individual lives until age T. The individual's problem is to choose an amount of saving/borrowing (a_{t+1}) and consumption (c_t) for each age $t > 0$ over

[3]The python codes that we use in this paper are posted in GitHub.

[4]Further, the life-cycle setting enables the results in this paper to be comparable in a consistent and complementary way with Fernández-Villaverde and Valencia (2018).

the life-cycle. The individual receives labor income and a return from the saving and encounters a borrowing limit ($\underline{a}$) every period. The individual's endowment (e_t) follows a Markov chain where a probability of receiving an endowment (e_{t+1}^j) at age $t+1$ conditional on the current endowment (e_t^k) is given by $P(e_{t+1}^j|e_t^k)$. The market prices (w, r) are taken as given. Then, the individual's problem at age $t < T$, for any given initial assets (a_0, e_0), is shown by the following Bellman equation:

$$V_t(a_t, e_t) = \max_{c_t, a_{t+1}} u(c_t) + \beta \mathbb{E}_t V_{t+1}(a_{t+1}, e_{t+1})$$

subject to

$$\begin{aligned} c_t + a_{t+1} &= we_t + (1+r)a_t \\ a_{t+1} &\geq \underline{a} \\ e_{t+1} &\sim P(e_{t+1}|e_t). \end{aligned}$$

where $V_t(\cdot)$ is a value function at age t, and we assume that the utility function takes an isoelastic preference, $u(c) = \frac{c^{1-\sigma}}{1-\sigma}$. In practice, this problem is solved backward assuming that the value function after life $(t > T)$ and saving at the end of life are zero.

Programming Language: There are many choices for programming languages. Low-level languages benefit from the computational speed-up while the coding and language designs could be more convoluted for less experienced programmers. Traditionally used low-level languages for quantitative analysis in economics were Fortran and C. On the other hand, coding with high-level languages is easier for less experienced programmers while a computation slowdown is unavoidable and some of the built-in programs are in a black box. Python and Julia are the two high-level languages with growing popularity in scientific computing. Yet, Matlab and R are still strongly supported by programmers due to their stable performance and rich supporting services and communities.

Aside from the trade-off between their computing speed and ease of learning, programming languages can also be classified by its programming style. Languages such as C++ and python are designed to be written in object-oriented, whereas other languages follow a procedural programming. Due to Python's growing popularity in computational economics, an objective-oriented style is more adopted to the quantitative analysis in economics than in the past.[5] One of the benefits of object-oriented programming (OOP) is its abstraction and inheritance. Variables and functions are considered as objects and encompassed in a class object. For example, OOP can create an abstract "human" class which contains walk, talk, and think as general functions that this object performs. Then, one can inherit this class and define more detailed characteristics, such as "gender" as an attribute, which characterizes an abstract "human" into more specific "male" and "female" classes. Although OOP is not a required feature of a language for the numerical computation of economic problems, we demonstrate an inefficient parallel code to which one is prone to write in OOP.

[5] QuantEcon projects provide open source code for economic modeling and tutorials for object-oriented programming in Python and Julia. https://quantecon.org/

In this exercise, we use Python 3.6.0-Anaconda, but our results are robust to other languages in general. The choice of Python is simply because of its flexible coding style (both OOP and POP), its user-friendliness to less experienced programmers, and its growing popularity in computational economics.

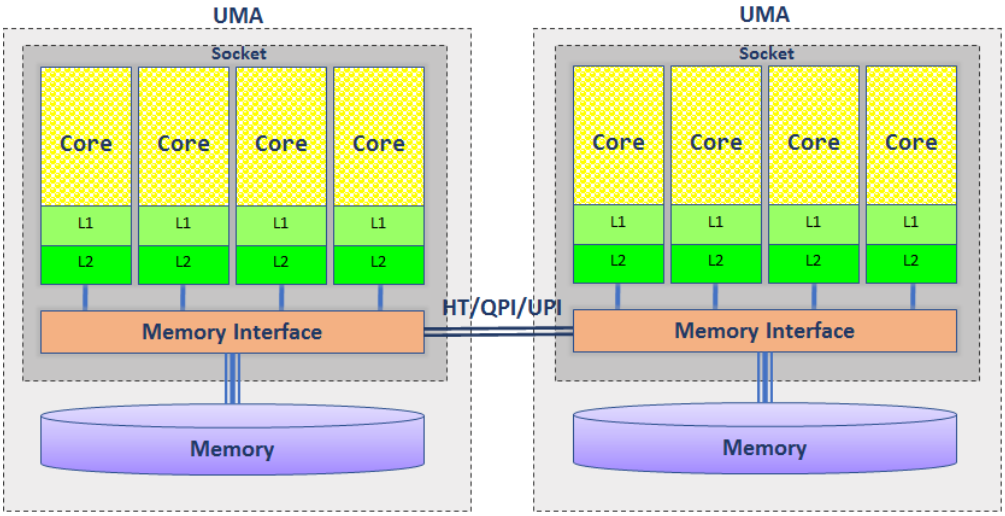
HPC architectures: A small-size memory, called cache (L1/L2 and sometimes L3), that holds copies of recently used data for processes is integrated on a CPU chip. Although a cache-sharing design is highly computer-dependent, L1 cache is usually attached to each core, while L2 cache can either be integrated on each core or shared by multiple cores. Cache-sharing among multiple cores can reduce communication overhead between cores and hence lowering the latency while improving the bandwidth. Due to its attachment in a CPU chip, an access to the cache should be very fast, despite its limited capacity. On the contrary, main memory (RAM) usually has a larger capacity in the order of magnitude. Despite its larger capacity as a storage, a farther distance of this memory from a CPU chip and its rate at which data can be transferred (bandwidth) cause longer time (latency) to access and transfer data. Finally, accessing both internal or external memory disk utilizes an IO interface, where the latency is high and bandwidth is low. Given the memory hierarchies and its associated latency and bandwidth, it is evident that the size and locality of memory are the keys for an efficient program.

Most common parallel computers are shared-memory computers where a space of memory is shared by multiple cores. It is very common these days that even our daily-use workstation comes with more than a single core. This kind of high-performance machines are called uniform memory access (UMA) systems. Since the pre-installed multi-cores (and caches) are connected to the same memory through a common front-side bus (FSB), latency and bandwidth of all the cores are the same. Even though the computing power of a multi-core workstation is improving day by day and year by year, a cluster of a number of multi-core processors multiplies the potential computing power. In general, the cluster system puts several UMA systems together to expand the number of cores, processors do not share the same memory. Therefore, this kind of system is called cache-coherent non-uniform memory access (ccNUMA) systems. Major distinctions between ccNUMA system and distributed memory machines are that the memories in ccNUMA system are physically distributed just like distributed memory, but are connected by high-speed network logic (QLogic?) that makes the whole memory systems virtually as one single address space. On the other hand, distributed memory machines are constructed by connecting a number of processing nodes (computers) via interconnecting networks. Therefore, a message passing interface (MPI) should be used to access and transfer data stored in a local memory of a node from different nodes. High-performance computing often refers to ccNUMA system via OpenMP, distributed memory via message passing interface (MPI), or GPU parallelization.

Figure 1 illustrates one example of ccNUMA systems. The ccNUMA system is comprised of two identical UMA systems where each UMA system consists of 4 cores with L1/L2 caches attached to each core[6] and are connected to the same memory. The two UMA systems are linked by a high-speed connection, HT/QPI/UPI, but are sharing two physically different memories

[6]Some systems may have an L2 cache shared by two or four cores in a socket. The design is highly computer-dependent.

Figure 1: HPC Architectures: UMA vs. ccNUMA



connected to each socket. This ccNUMA system enables programmers to use up to 8 physical cores with duplicated memory. This locality of memory to each socket will turn out to be the key for the parallelization overhead in the later sections.

For this exercise, we use one of AHPCC consists of three sub clusters, interconnected with a 324-port QDR 40 Gbps nonblocking QLogic Infiniband switch and supplementary switches, and is connected to an IBM GPFS shared file system with 88 TB of long-term storage and 35TB of scratch storage. The spec is Linux XX-XX-XX, quadruple Intel Xeon E5-4640 CPU, which contains 32 physical cores with multi-threading, and 768GB memory per node.

2.2 Profiling Solution Methods:

Value function iteration (VFI) and policy function iteration (PFI) are the two of typical solution methods to solve a dynamic problem. Besides VFI and PFI, there has been a development in solution methods to solve for a variety of dynamic problems. Most of the solution methods attempt to modify root-finding tasks, interpolation tasks, and tensor-product constructs of grid points of VFI. For instance, projection methods (Judd, 1992) modifies the interpolation part, endogenous grid methods (Carroll, 2005) modifies root-finding part, Smolyak sparse grid methods (Krueger and Kubler, 2004) modifies the tensor-product grid points in a hyper-cube. Further, recent development in perturbation method (Judd and Guu, 1997). Also, simulation algorithms such as stochastic simulation algorithm, ε-distinguishable set method, and cluster grid method attempt to deviate from fixed grid point algorithms. Although these solution methods are successful in reducing the computational time of a program with a single processor at a certain degree, we take VFI as our benchmark solution method in this exercise to assess the performance of parallel computing in HPC environment. It is worth noting that our results are robust with the choice of solution methods because a parallelization gain is solely determined by the amount of work to be parallelized which can vary by the complexity of model, the number of state space, and the grid size.

A pseudo-code for VFI algorithm is shown in Table 1. Since the stochastic endowment follows a Markov process in the life-cycle model, the functions should be evaluated at its current state space. To start with, we discretize each state space $\mathcal{A}, \mathcal{E}$ into N_a and N_e number of points (Step 1). We also discretize the stochastic process. We start solving a life-cycle problem from the end period by backward induction, since the saving decisions are trivially zero (Step 2). The value/policy functions at age T are evaluated at each grid point that we set at the beginning. Due to its triviality, no root finding and interpolation are involved in solving the problem at period T. Given the value function at end period, we solve a household problem for periods $t < T$. In each period other than $t = T$, at a given state $(a_t, e_t) \in \mathcal{A} \otimes \mathcal{E}$, the program uses a minimization routine to find saving that maximizes the lifetime utility (Step 3(a)). In the process of searching, the expected continuation value is calculated by interpolating the calculated next period's value function (Step 3(b)).

When the program is executed with a single processor, Step 3 accounts for 99.7% of total serial runtime. In particular, 96.5% of time is expensed for root-finding and interpolation process. This is the reason that the undergone development in solution methods in the literature has all replaced root-finding and interpolation in the program with more efficient algorithms. In the meantime, it is informative that this part of program should be parallelized across multiple processors for a further speed-up. In fact, efficiently parallelizing this part of program can potentially reduce 99.7% of total runtime.

2.3 Scalability and Granularity

Whether or not to parallelize tasks of the major bottleneck of a program depends on its scalability: how easily and effectively the tasks can be parallelized. Dynamic programming is an algorithm that value/policy functions are converging to one unique point after several (hundred or thousand) iterations. This algorithm critically hinges on updating value/policy functions from the previously computed value/policy functions. Even with the life-cycle model, time t value/policy functions are solved by the value/policy functions at time $t+1$ calculated at a previous iteration. Therefore, it is evident that the program cannot be easily scalable by age, i.e. we need to compute time $t + 1$ problem before we compute time t problem. On the other hand, value/policy functions in a typical dynamic problem are evaluated at selected finite discrete points in a state space. Since each grid point of state variables has no interactions with other grid points, evaluating a function at each grid point is easily parallelizable.

Then how much tasks should be assigned to each processor? To obtain a fair amount of accuracy on the approximation of value/policy functions, a large size of grid points in a state space hyper-cube should be taken. But this comes with the considerable amount of time cost. Therefore, granularity—the amount of tasks assigned in parallel to each processor—is controlled by the number of grid points on state variables. We will show in the later section that the right granularity depends on the number of processors used for parallel computing.

Table 1: Pseudo-VFI Code and the Percent of Total Runtime

Procedure	**Algorithm**	**% of Time**
Step 1.	Initialization	0.0538
a.	Set model parameters.	0.0002
b.	Construct grid points for $a_t \in \mathcal{A}$.	0.0001
c.	Construct grid points for $e_t \in \mathcal{E}$ with a transition matrix $P(e_{t+1}\|e_t)$.	0.0535
Step 2.	Computing a household problem at $t = T$	0.0224
	for $(a_T, e_T) \in \mathcal{A} \otimes \mathcal{E}$, **do**	
	$a_{T+1} = 0$	
	$c_T = we_T + (1 + r)a_T$	
	$V_T(a_T, e_T) = u(c_T)$	
	end for	
Step 3.	Computing a household problem at $t < T$	99.7448
	for $(a_t, e_t) \in \mathcal{A} \otimes \mathcal{E}$, **do**	
	Search for $a_{t+1} \in \mathcal{A}$ that maximizes	
	$W_t(a_t, e_t, a_{t+1}) = u(we_t + (1 + r)a_t - a_{t+1}) + \beta \mathbb{E}_t V_{t+1}(a_{t+1}, e_{t+1})$	
	using a bounded minimization routine.[7]	
	end for	
	In particular, each search process at a given state (a_t, e_t) takes the following sub-procedures:	
a.	For each a_{t+1} at a given state (a_t, e_t), interpolate an expected continuation value $\mathbb{E}_t V_{t+1}(a_{t+1}, e_{t+1})$.	96.5097
b.	For each a_{t+1} at a given state (a_t, e_t), compute $W_t(a_t, e_t, a_{t+1})$.	3.2351

3 Parallelization Overhead

Even though a performance by floating-point operation per second (Flops/sec) is often used as a measure of efficiency of parallel computing in computer science, our utmost interest is its computing speed and accuracy in computational economics. Therefore, we introduce another measure for efficiency—a parallelization overhead which is defined as an additional time cost by parallelizing a chunk of tasks to multiple processors.

3.1 Defining Parallelization Overhead

A series of program can be split into serial tasks which are executed by a single worker and parallel tasks that are distributed to multiple workers. Then the total runtime (execution time) of a program with n workers is given as:

$$T(n) = \frac{T_p}{n} + T_s + P(n)$$

where T_p is the runtime of parallelizable tasks with a single core, T_s is the runtime of a serial code, and $P(n)$ is a parallelization overhead. When a set of tasks is distributed to n workers, then the ideal computing time of the parallelized tasks should be T_p/n. However, the actual time to compute the tasks in parallel requires extra $P(n)$ seconds.

Though the shape of $P(n)$ is unknown, if $P(n)$ is a nice convex function of n, then the optimal number of cores (n^*) to minimize the total runtime is determined by the following equation:

$$P'(n^*) = \frac{T_p}{n^{*2}}.$$

Further assume that the parallelization overhead is an approximated function $P(n) = pn^\alpha$ with constant $p, \alpha > 0$. Then, the optimal number of cores must be

$$n^* = \left(\frac{T_p}{\alpha p}\right)^{\frac{1}{1+\alpha}}$$

This implies that when a constant part of parallelization overhead (p) is high, a gain from using HPC to parallelize tasks would be limited. On the contrary, if the granularity of the program is high (T_p) then the gain from HPC would be huge. This simple analysis insists on the importance of parallelization overhead to understand the potential gain from HPC parallelization and the optimal number of workers to be used. Further, this gives an idea that parallelization overhead is a key to write an efficient parallel program. Therefore, we calculate the actual parallelization overhead with n cores from the equation above:

$$P(n) = (T(n) - T_s) - \frac{T_p}{n},$$

provided that $P(1) = 0$.

After we define the parallelization overhead simply as a difference between the actual and

ideal runtime of parallelized tasks, our next question should be sources of the overhead. In fact, parallelizing tasks across multiple workers necessitates several extra procedures which create inevitable systemic overheads. For example, creating and closing a pool of workers even take several (milli-)seconds. Aside from such systemic overheads, we mainly focus on two preventable sources of overhead that substantially contribute to the parallelization overheads: (1) communication overhead from memory access and (2) idling overhead from load imbalances. The former latency is closely associated with the computer architectures, while the latter closely relates to the parallelized tasks of a program. Even though the full understanding of internal designs and mechanics of computer system and programming languages are not required, it is always helpful to grasp the main features of HPC architectures and performance of programming languages to identify the sources of parallelization overhead and to write an efficient code. We will start discussing the sources of overhead in more detail in the next sections.

3.2 Parallelization Overhead in Shared-Memory Programming

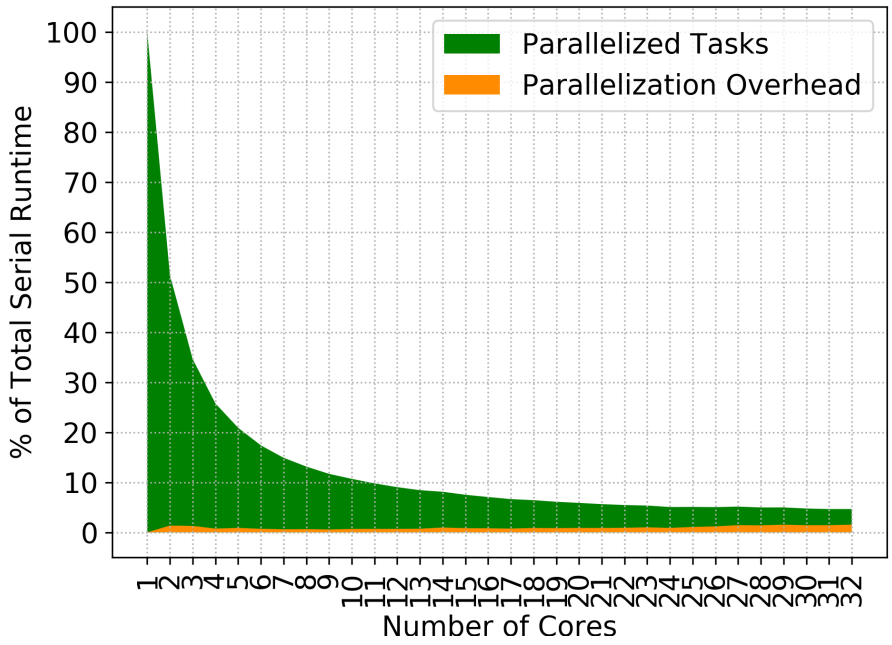
Parallelization scheme may depend on the available multi-core architectures. As discussed in the previous section, if multi-cores are designed in UMA or ccNUMA structure, then a shared-memory parallel programming with OpenMP should be used for parallel computing. On the other hand, if multi-cores in multiple nodes are interconnected by network and formulate a cluster, then a distributed-memory parallel programming with message passing interface (MPI) can also be used. Recently, there is a growing interest in GPU programming, but structurally, GPU programming is similar to distributed-memory programming since GPU memory is interconnected with CPU memory by a PCI bandwidth. Therefore, we first show the scalable outcome of parallel computing in shared-memory environment with OpenMP in this exercise.[8]

To experiment the efficiency of parallelization on the HPC environment specified above, we set standard values for model parameters in the life-cycle model. We assume that households live until $T = 10$ periods and use the VFI algorithm to solve the life-cycle model presented above with grid sizes for (a, e) to be $N_a = 1000$ and $N_e = 15$, respectively, as a benchmark. For the ease of exposition and comparison, we normalize the computational time of a program with n number of cores by the serial runtime with a single core. Therefore, changes in parameter values which affect the total runtime of a program have almost no effect on the normalized runtime of a program with multi-cores.

We first present an *optimistic* view of parallel computing with maximum 32 cores in shared-memory environment if we run a "carefully written (almost) efficient" program. Figure 2 shows the normalized runtime of the program run in parallel with different numbers of cores from 1 to 32 by serial runtime. If there is no overhead of any kind in parallelizing tasks, the ideal runtime with $n > 1$ cores must be $100/n$ percent of serial runtime. The figure shows that the total runtime

[8] In appendix, we demonstrate the parallelization overhead on GPUs for a growing interest on GPU computing and on distributed-memory programming with MPI. For the choice of shared-memory, distributed-memory, of GPUs, one has to consider an essential trade-off between communication overhead and the number of available cores. In particular, GPUs and distributed memory literally enables a thousand cores to be used, while a farther distance of their memory from a CPU chip and its rate at which data can be transferred (bandwidth) cause longer time (latency) to access and transfer data. As a consequence, the parallelization overhead should be higher than that in the share-memory.

Figure 2: Parallelization Overhead by Number of Cores



with 2 cores slightly above the half of serial runtime, and the total runtime reduces to 4.65% of serial runtime with the max 32 cores. Thus, if an actual computational time of the program takes 1000 seconds with a single core, then it is possible to speed up the time by 46.5 seconds with 32 cores. Whether this speed up turns out to be a good news or bad news depends on how many iterations this program needs to be run. If one were to run this program for 1000 iterations, then 1000 seconds of serial runtime are hopeless, but even 46.5 seconds for one iteration should take 46500 seconds with 32 cores. From this perspective, we must be able to take the parallelization speed-up to the limit, which critically hinges on the parallelization overhead.

As is shown in the figure, the parallelization overhead takes up on average 0.94% of total runtime and is very slightly rising as the number of cores increases. This implies that the parallelization overhead sets the lower bound for the runtime speedups one can achieve with a large number of cores. In other words, a hypothetical speedup of a program with infinite number cores (if available) should be equal to the limit of parallelization overhead with $n \to \infty$. With the shared-memory, an infinite number of cores is not available at the time of writing. Perhaps, GPUs and distributed-memory with more available cores are the natural transition for the further speed up. Be that as it may, but not so fast. In the shared-memory environment, we will examine the sources of parallelization overhead arising from HPC architectures and parallelizing tasks, which are definitely the issues arising in GPU and distributed memory.

4 HPC-Efficient Algorithms

Parallelizing a serial program does not require the full understanding of internal designs and systems of computer architectures and programming languages. In fact, most programming languages are designed to be able to parallelize a serial program with a dual-core laptop computer just by adding one or two simple lines of code, without paying any attentions on how tasks are parallelized across multi-cores.[9] Nevertheless, a better understanding of the HPC architectures is critical to make use of full computing power of high-performance computers. In this section, we demonstrate two main sources of parallelization overhead—communication overhead and idling of workers—in a typical shared-memory architecture which may essentially arise from the lack of understanding of HPC architectures and parallelization algorithms. We then introduce easily-implementable HPC-efficient coding recommendations.

4.1 Passing an Object or Variables?

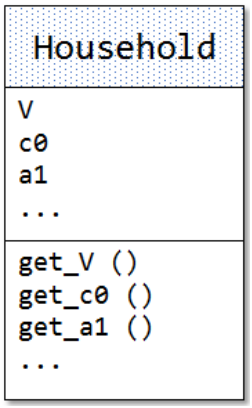
Object-oriented programming (OOP) as oppose to procedural programming (POP) has been adopted and used in recent programming languages such as Java, C++, Python, Ruby, and many more for quite some time due to its great advantages in reusing and compactifying codes.[10] On the contrary, it is quite recent that OOP is getting used in the quantitative analysis of economics. This is simply because of the commonly chosen languages in the economics profession (like Fortran or Matlab) and the overall length and types of program running for quantitative analysis. The idea of OOP is based on "object" in which variables and functions are contained as attributes. To apply OOP in an economic modeling, for instance, a household can be constructed as a class object, which contains variables such as V, $c0$, and $a1$ and functions to obtain those variables (see Figure 3). After defining the class in a program, we need to create an instance of household class to work with the object. The instance of household class can be distinguished by its attributes or initial values of a household (e.g. initial productivity, endowment, etc.).

Despite its coding benefit, we tend to overlook bottlenecks from memory access and data transfers with OOP which may not be an issue in serial case. Parallelizing tasks usually entail (without a specific memory control) data in the main memory that are used in parallelized work to be systemically copied to a local memory in each worker so as to reduce the amount of time accessing the data by each worker. In our example, value function at period $t+1$ is utilized to compute period t value function. Therefore, parallelizing tasks by grid points of state space to multiple workers needs a copy of the next period's value function stored in the local memory of the main processor to a local memory of each worker. With less attention to this memory allocation, however, we are prone to pass one single household object which contains next period's value function to each worker when lots of variables in the object need to be passed for the parallel process. This passing of an entire object in fact copies the whole attributes contained in this object to the memory in each worker. Therefore, if the size of object is large, passing the entire

[9] For example, OpenMP in C/C++ or Fortran needs to include `!$OMP PARALLEL` line to open up a parallelizing block of code and close with the line at the end of the tasks. Matlab can parallelize `for` loops by just replacing with `parfor`.

[10] Computational time difference of a serial program written in object-oriented programming and procedural programming is negligible. In our example of economic model, OOP is a few seconds slower than POP, but the difference is tiny.

Figure 3: Household Class in Object-Oriented Programming



object to a large number of workers may cause overhead from data transfer. In particular, if the class object contains large variables that are not used in parallel processing, then it is not efficient to pass the whole object to each worker. Then, an alternative to passing the object is simply passing a list of variables that are used for the computation.

Figure 4 shows that parallelization overhead when passing an object instead of variables starts rising after 12 workers and with 32 workers parallelization overhead with an object is XX% higher than passing variables. In terms of the total serial runtime, it is only around 0.6% with 4 to 10 parallel workers but this rate becomes 1.5% with 20 workers and 2.15% with 32 workers. On the contrary with variables, it is almost the same parallelization overhead with 7 to 10 workers, while this rate gradually rises to 1% with 23 workers and 1.5% with 32 workers. As described above, an object containing large size of data incurs more parallelization overhead with the object has to be copied to a large number of workers. Likewise, with a small number of workers like 2 to 4, the parallelization overhead is relatively smaller with an object than variables.

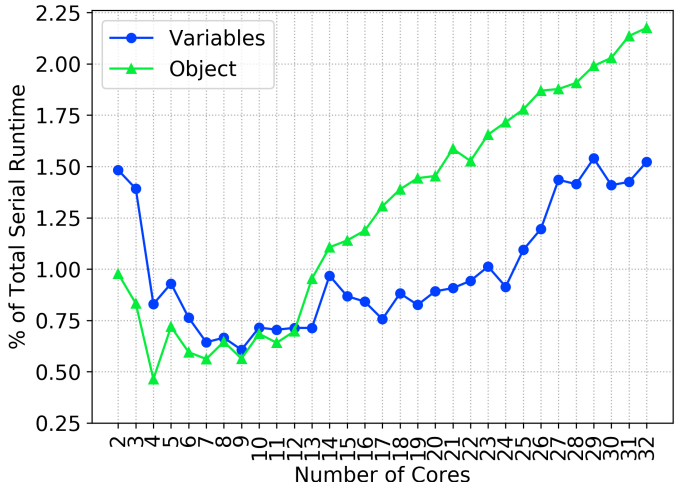
One might be tempted to use global variables to share the variable by multiple workers instead of passing variables. This could be efficient if communication overhead is less than data transfer overhead. Nevertheless, "global variable should be avoided in every way possible, esp. when the variable is non-constant" is one common advise that experienced programmers give. The reason is simply because the variable can be concurrently changed by different workers and detecting the changes is such a pain. Also often the parallel processors may have their own global memory and hence any defined global variables may not be shared by processors.

4.2 The Right Granularity for the Number of Cores

Granularity is defined as the amount of work in the parallel task. It is trivial that the smaller the granularity, the faster the computation, while the accuracy of policy functions and simulations undoubtedly benefits from the larger granularity. Therefore, a right size of granularity should be determined for an efficient HPC program.

In grid search methods that are often used to solve an economic model, granularity is controlled

Figure 4: Parallelization Overhead in Object-Oriented Programming



by the number of grids on state variables. Each state space is discretized into a finite number of points, and value/policy functions are evaluated at each point. Since each grid point in a hyper-cube of state space indicates an independent state from other states, value/policy evaluations can be independently conducted at each point in parallel. Therefore, when more cores are used one might be tempted to increase the number of grid points to improve the accuracy of value/policy functions.

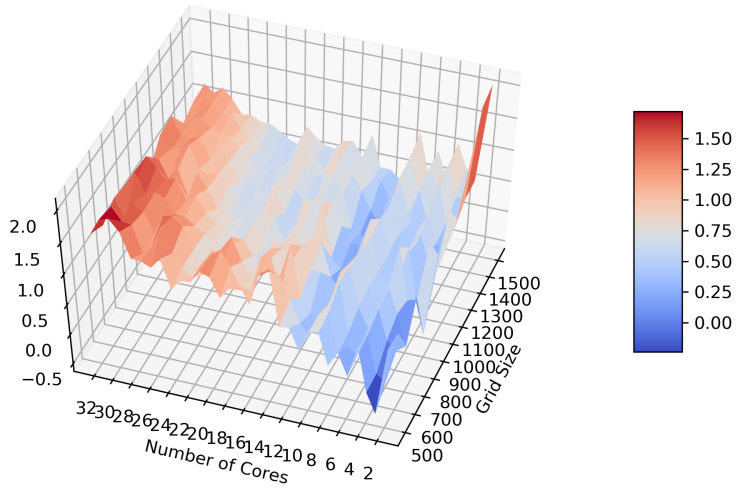
Figure 5 shows the parallelization overhead relative to the total serial runtime of selected grid sizes. The relative parallelization overhead gradually shrink as grid size rises. This is due to the fact that total serial runtime is rising as the grid size rises. In other words, the growth of overhead is slower than the growth of total serial runtime by an increase in grid size. It is clear from the figure that more cores can handle a large number of grids in relative measure. It should be re-emphasized that this does not mean that the absolute overhead is smaller as the grid size rises with more cores.

It is also inefficient to utilize a large number of cores for the smaller grid sizes. For example, the relative overhead rises to more than 1.5% of total serial runtime when 32 cores are used with 500 grid sizes. However, the relative overhead gradually shrinks to 1% as the granularity enlarges to 1500 grid points. There is an opposite performance for the small number of cores. For example, with only 2 cores, smaller granularity only takes less than 0.5% of total runtime, while the overhead rises up to 2% as the granularity rises. Therefore, it is instructive to state that the number of cores used and the size of granularity are in positive relation in terms of the relative overhead measure.

4.3 The Right Chunk of Tasks

Chunk size is related but is different from granularity. We distribute the entire tasks to multiple processors by a number of chunks of tasks. On the one extreme, we could divide the entire tasks

Figure 5: Parallelization Overhead by Granularity



(or grid points) into the number of cores and hence lump one chunk of tasks per node. On the other extreme, we could pass one task at a time to each processor as a processor is done with an assigned task. The former case with one large chunk minimizes the communication overhead due to one time passing of a chunk of tasks to each worker, while the latter case with smaller piece of chunks allows to prevent the idling overhead. If each task has different size of work, load imbalance occurs when a chunk of tasks is completed by some processors while other processors

If each task is about the same size of work that completes about the same execution time, a larger chunk should be benefited by reducing the communication overhead. However, a task containing root-finding or interpolation processes may often involve load imbalances across tasks, and hence smaller chunks can more flexibly avoid idling of cores waiting for cores that are working on longer processes. However, it is totally impossible to detect actual runtime of each task *ex ante* and therefore is impossible to calculate the accurate size of chunks to be dispatched to each worker for a complete load balancing.

Figure ?? shows the parallelization overhead by the size of chunk per core. We keep the grid size to be 1000 as before but only changing chunk size parameters of parallelization function. It is clear that assigning a large chunk of tasks per core is subject to the idling overhead which raises the parallelization overhead to 2-4% of total serial runtime. But as we increase the number of chunks per core, the flexibility rises that the parallelization overhead plummets by almost half. For most of the number of cores, parallelization overhead is moderately flat after 10 chunks of tasks. Of course, the result may change by the number of cores, but the overhead is relatively constant after 10 chunks. As the number of cores becomes 32 overhead gradually rises from 1.5% to 3%. This is simply the fact that communication overhead idlin goverhead dominates for a large number of cores.

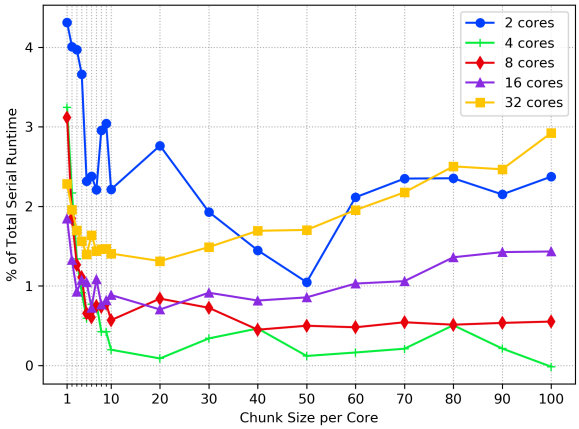
One interesting experiment that we conduct is when an object instead of variables is to be passed to each worker. In this case, communication overhead becomes more dominant that the parallelization overhead by 32 cores rise by more than 50% of total serial runtime. This is due to the memory copying of an entire object could be more costly when a large number of cores is used than passing necessary variables..

One interesting experiment that we conduct is when an object instead of variables is to be passed to each worker. In this case, communication overhead becomes more dominant that the parallelization overhead by 32 cores rise by more than 50% of total serial runtime. This is due to the memory copying of an entire object could be more costly when a large number of cores is used than passing necessary variables..

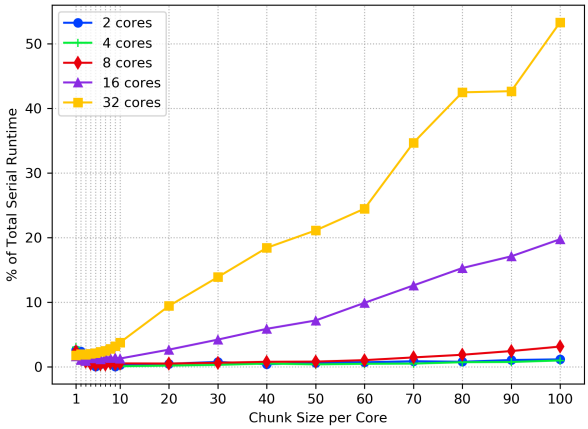
Static scheduling vs. dynamic scheduling

Figure 6: Parallelization Overhead by Chunk Size

(a) Passing Variables



(b) Passing Object



5 Conclusion

6 Bibliography

References

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

Carroll, C. D. (2005). The Method of Endogenous Gridpoints for Solving Dynamic Stochastic Optimization. *NBER Technical Working Paper 309*.

Fernández-Villaverde, J. and Valencia, D. Z. (2018). A Practical Guide to Parallelization in Economics. Working Paper.

Judd, K. L. (1992). Projection Methods for Solving Aggregate Growth Models. *Journal of Economic Theory*, 58:410–452.

Judd, K. L. and Guu, S.-M. (1997). Asymptotic methods for aggregate growth models. *Journal of Economic Dynamics and Control*, 21:1025–1042.

Krueger, D. and Kubler, F. (2004). Computing equilibrium in OLG models with stochastic production. *Journal of Economic Dynamics and Control*, 28(7):1411–1436.

7 Appendix

7.1 Parallelization Overhead on GPUs

another multi-core shared memory architecture is GPU. GPU programming is suited for a light-weight arithmetic computation with a mass of data calculation. Jesus's paper analyzed the performance of GPU parallelism, so I refer to the paper and minimize the discussion in this paper. Another reason to limit the discussion on GPU programming is that since GPUs have their own memory independent from CPU memory. So parallelism requires a copy of data to the GPU memory, which generates computational overhead. There are very few tricks for an efficient program on GPUs. GPUs are a hardware accelerator, physically separated from the CPU.

7.2 Overhead on Distributed Memory Parallelization