Parallelization Overhead and HPC-Efficient Programming

Dongya Koh[*]

University of Arkansas

January 4, 2019

Abstract

An easier access to high-performance computing (HPC) systems enables parallel computing more commonplace in the research of economics. Nevertheless, it is surprising that very little is known about the full gain of modern computing power via parallel computing. Without knowing the computer architectures and parallel algorithms, parallel computing with a large number of cores could result in a significant slowdown than a speed-up. To measure the size of inefficiency in parallel computing and to identify the sources of overhead, this study introduces a new metric, parallelization overhead. It turns out as a consequence that reducing the parallelization overhead to its minimum automatically leads to writing an HPC-efficient parallel program.

[*]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 potential (scalable) speed-up of solving a dynamic problem in economics with more than one core is possible only when a program is written in an efficient way for parallel-computing. 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 in 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 is becoming a benchmark 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 shared memory architectures mainly arises from two sources: communication overhead from 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

[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.

fact, this overhead arising from the memory access and data transfers varies by the distance of local memory from 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 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 gets coarse, total computational time and parallelization overhead both increase for any number of parallel workers. However, with a finer granularity, parallelization overhead becomes relatively high when a large number of workers are used. This is because the processing time for computation by each worker is much quicker than the processing time for creating a pool of workers and transferring data to and from each worker. On the other hand, the amount of time required to create a pool of workers and to transfer data should be relatively faster than processing time for computation with a coarse granularity. This instructs us a rule of thumb that the parallelization overhead becomes smaller when granularity is fine with the small number of workers or when granularity is coarse with the large number of workers. Thus, the right granularity and the right size of parallelism should be jointly determined.

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-purpose 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]

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

[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 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 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^j_{t+1}) at age $t+1$ conditional on the current endowment (e^k_t) is given by $P(e^j_{t+1}|e^k_t)$. 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, the low-level languages used for quantitative analysis in economics were Fortran and C. On the other hand, coding with high-level languages is much easier for less experienced programmers while a fine-grained control of inner work would be lost and a computation slowdown is unavoidable. Python and Julia are the recent 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 manner, whereas other languages follow a procedural programming (POP). 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 present efficient OOP coding

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

techniques to prevent any parallel-computing slowdowns.

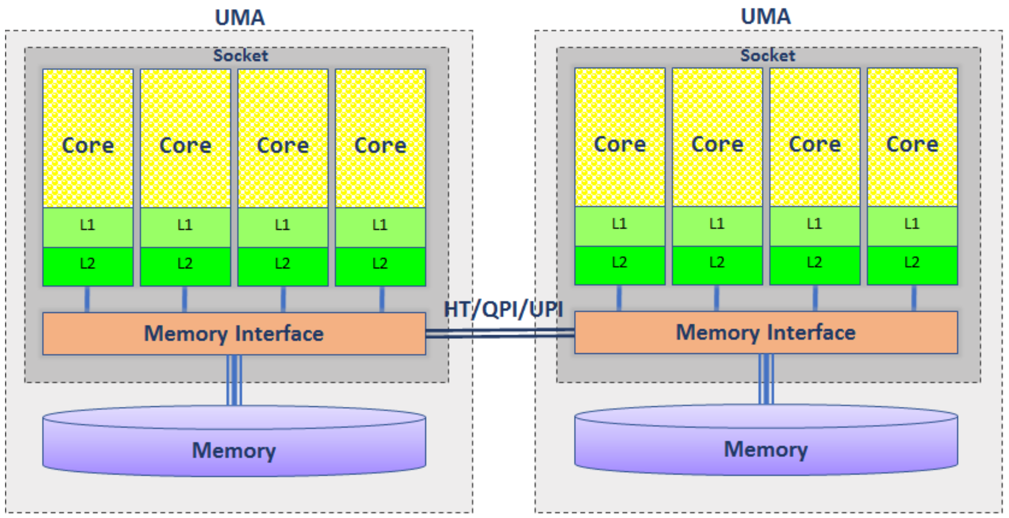
In this exercise, we use Python 3.6.0-Anaconda, but our results are robust to other languages in general. The choice of Python is simply because of its flexible coding style (both OOP and POP), its user-friendliness to less experienced programmers, and its growing popularity in computational economics.

HPC architectures: A very common parallel computer to which we have an easy-access is our daily-use laptop computer or a desktop workstation which comes these days with more than a single core. In general, parallel computers in HPC systems combine multiple CPUs/cores/threads so that multiple instructions or multiple data can be distributed to those available resources and executed concurrently. There are many ways to combine CPUs and cores so that a large size of instructions and data can be executed in parallel. For example, a dual-core laptop simply has two cores in one CPU socket. Or, multi-core CPU sockets are combined together to increase the number of available cores. Or, multi-core multi-CPU computers are connected by network to multiply the available cores. Or, even graphic processors can be utilized to incorporate thousands of cores.

Regarding the design of parallel computers, the key is how to share memory across multiple processors. 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 socket. 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 processors and hence lowering the latency while improving the bandwidth. Due to its attachment to physical cores, 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 processing units 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 memory organization of HPC architectures is the keys for parallel computing.

The most basic memory organization of parallel computers are shared-memory computers where an address space is shared by multiple CPUs or cores. 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 desktop workstation is improving day by day and year by year, a cluster of a number of multi-core processors multiplies the potential computing power to the order of magnitude. 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 bus that makes the whole memory systems virtually as one single address space. On the other hand, distributed memory machines

Figure 1: HPC Architectures: UMA vs. ccNUMA



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 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 a ccNUMA shared-memory queue provided in the Arkansas High Performance Computing Center (AHPCC) cluster system which 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 operating system of the queue is CentOS 6.10, and the queue has quadruple Intel Xeon E5-4640 CPU, which contains 32 physical cores with multi-threading, and 768GB memory per node. Later in the appendix, we utilize two nodes of this environment, which essentially provide max 64 cores, to experiment distributed-memory parallel programming.

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

2.2 Profiling Solution Methods:

Value function iteration (VFI) and policy function iteration (PFI) are the two primitive solution methods to solve a dynamic programming problem. However, since these solution methods exhibit slow computational time, low accuracy, and the curse of dimensionality, there has been a development in solution methods, most of which attempt to modify root-finding algorithms, interpolation methods, or tensor-product constructs of grid points of VFI/PFI.[7] For instance, projection methods (Judd, 1992) and perturbation method (Judd and Guu, 1993) both modify the approximation of value/policy functions. The solution methods that simplify root-finding are endogenous grid methods (Carroll, 2005, Barillas and Fernández-Villaverde, 2007) and envelop condition methods (Maliar and Maliar, 2013). Also, there has been a number of improvements in replacing tensor-product grid constructs which causes the well-known "curse of dimensionality" (e.g. parameterized expectation algorithm (Marcet, 1988, Haan and Marcet, 1990), Smolyak sparse grid methods (Krueger and Kubler, 2004, Judd et al., 2014), stochastic simulation algorithm (Judd et al., 2011), cluster grid method (Judd et al., 2010), ε-distinguishable set method (Judd et al., 2012), and adaptive sparse grid method (Brumm and Scheidegger, 2017)) 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 scaling effects from parallel computing are obtain when granularity is coarse, which we discuss in the later section in more details.

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 $\mathcal{A}$ and $\mathcal{E}$ 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 proceed to 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 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. The fact that the serial part of the program accounts for very little of

[7] Aruoba et al. (2006) document the performance comparison of widely used solution methods. Maliar and Maliar (2014) review a broader set of solution methods.

its total computation implies that Amdahl's concern (i.e. "Amdahl's Law") about a limit on the potential gain from parallelization is minimized.

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**	
	$\quad a_{T+1} = 0$	
	$\quad c_T = we_T + (1+r)a_T$	
	$\quad V_T(a_T, e_T) = u(c_T)$	
	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.[8]	
	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

2.3 Whether or Not to Parallelize: Scalability and Granularity

Whether or not to parallelize a serial 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 for each grid point of state variables the same set of tasks are computed independently from the tasks at other grid points, evaluating a function at each grid point is easily parallelizable. This implies that `for` loop over each grid point in Step 3 should easily be parallelized to multi-cores.

In fact, parallelizing the dynamic problem is a simple *data parallelism* that executes the same instructions on different parts of the large data.

Another dimension to be explored for a potential benefit from parallelization is granularity. Granularity is defined as the ratio between the amount of computation time and the amount of communication time in a parallelized task. When the granularity of parallelized tasks is coarse (fine), more (less) computation is required than communication. A rule of thumb is that the coarser the granularity, the more speed ups we can obtain from parallelization of tasks. In our example, the granularity of Step 3 could be coarse if we set a large number of grid points for state variables since the communication (synchronization) takes place after multiple workers complete evaluating value/policy functions at all grid points at each age. Therefore, a rule of thumb is that a program with a large number of grid points (a coarse granularity) should be good candidate for parallel computing in HPC. One caveat, however, is that as the number of grid points increases, the accuracy on the approximation of value/policy functions improves, while it comes with the considerable amount of time for computation. Given such a trade-off, we will show in the later section that the right granularity (the grid size) depends on the number of processors employed for parallel computing.

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.

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 the sources of overhead. In fact, parallelization overhead arises from several procedures to parallelize into multiple workers where some of the overheads from those procedures are inevitable systemic overheads. For example, creating and closing a pool of workers take several (milli-)seconds. In addition, there are many more sources of overheads if memory organizations changes (e.g. distributed memory or GPUs). Aside from such unavoidable overheads, the numerical computation of dynamic problem may suffer from two major sources of overhead that substantially contribute to the parallelization overheads in shared memory architectures: (1) communication overhead from data transfer and (2) idling overhead from load imbalances. The former latency is closely associated with the computer architectures, while the latter is closely related to the parallelized tasks of a program. Usually, the communication overhead is not incredibly large on shared-memory environment due to low latency and high bandwidths (Hager and Wellein, 2011). Nevertheless, this overhead should still be critical since OOP and a flexible control over granularity in our example alter the size of transferring data, and it would become a major obstacle in GPU and distributed-memory parallelization. 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 details in the next sections.

3.2 Parallelization Overhead with OpenMP

Parallelization scheme may depend on the available HPC architectures. As discussed in the previous section on HPC environment, if multi-cores are designed in UMA or ccNUMA structure, then a shared-memory parallel programming with Open Multi-Processing (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 in terms of overheads, GPU programming is similar to distributed-memory programming since GPUs and distributed memory literally enables a thousand cores to be used, while a farther distance of their memory from the main 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. Also, the share-memory environment can single out the most important two sources of parallelization overhead for our economic computation. Therefore, we show the scalable outcome of parallel computing in shared-memory environment with OpenMP in this exercise.[9]

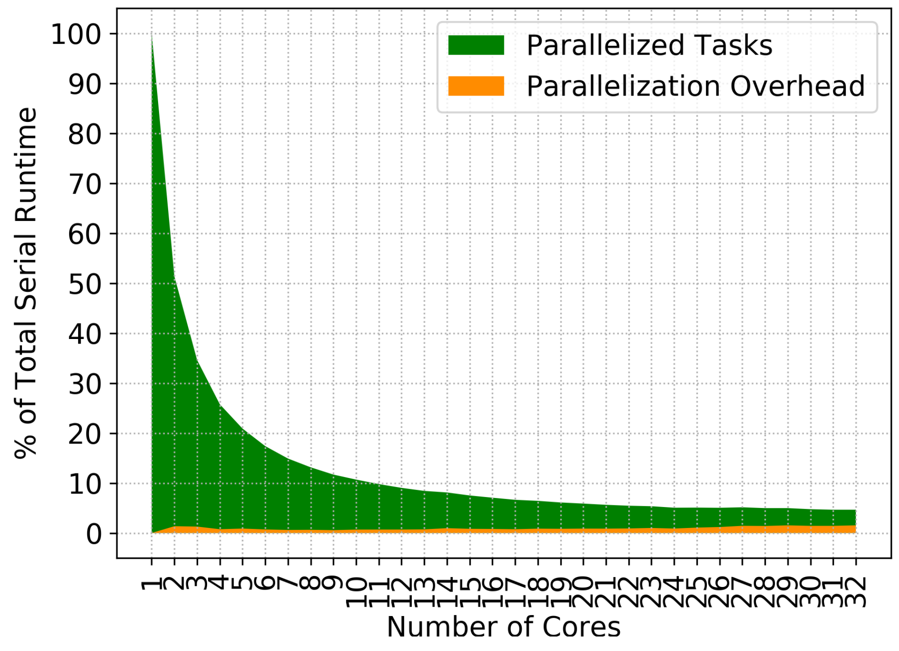
To experiment the efficiency of parallelization on the HPC environment specified above, we set standard values for model parameters in the life-cycle model. We assume that households live until $T = 10$ periods and use the VFI algorithm to solve the life-cycle model presented above with grid sizes for (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 that affect the total runtime of a program have almost no effect on the normalized runtime of a program with multi-cores.

We first present a *promising* result 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. 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 with 2 cores is slightly above the half of serial runtime, and the total runtime reduces to 4.65% of serial runtime with the max 32 cores. 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 or not this speed up turns out to be a good news depends on how many iterations this program needs to be run. If one were to run this program for 100 iterations, then 1000 seconds of serial runtime are hopeless, but even 46.5 seconds for one iteration should take 4650 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 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 speed-ups 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 $P(n)$ as $n \to \infty$. 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

[9]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.

Figure 2: Parallelization Overhead by Number of Cores



are definitely the issues arising in GPU and distributed memory.

3.3 Parallelization Overhead with MPI

OpenMP that we use as an parallel computing interface in shared memory cannot be used in distributed memory architectures. Message passing interface (MPI) is indeed a counterpart interface in the distributed memory.

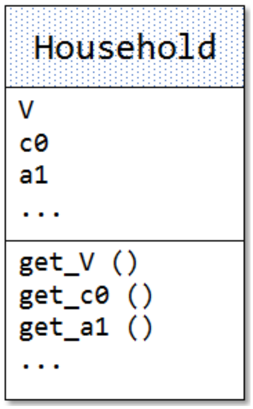
[To Be Completed]

4 HPC-Efficient Programming

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.[10] 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

[10]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`.

Figure 3: Household Class in Object-Oriented Programming



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.[11] On the contrary, it is just 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 overheads 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

[11]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.

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 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 cores and the overhead becomes 43% higher than passing variables with 32 cores. By passing an object, it is only around 0.62% of total serial runtime with 4 to 12 parallel workers, but this rate becomes 1.45% with 20 workers and 2.18% with 32 workers. On the contrary with variables, it is almost the same parallelization overhead with 8 to 12 workers, while this rate gradually rises to 1% with 23 workers and 1.5% with 32 workers. As described above, an object containing a large size of data incurs more parallelization overhead when the object has to be copied to a large number of workers. Thus, one critical source of parallelization overhead arises from the data transfers to a pool of workers.[12] Note that Figure 2 is the result from passing variables instead of object to a pool of workers.

Given the fact that data transfers create overhead, 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 to the global memory is less costly than overhead from data transfers. Nevertheless, "global variable should be avoided in every way possible, esp. when the variable is non-constant" is one common advise that experienced programmers would tell. The reason is simply because the variable can be concurrently changed by different workers and detecting the changes is such a pain. Also, the parallel workers may often have their own global memory and hence any defined global variables may not be globally shared by processors.

There are many other ways to get around this data transfer problem. Compressing data could be one way to downsize large data before transferring to each worker. Also, selecting data types that efficiently store data in the main memory could be another. These are language-specific techniques that we pursue for the future research.

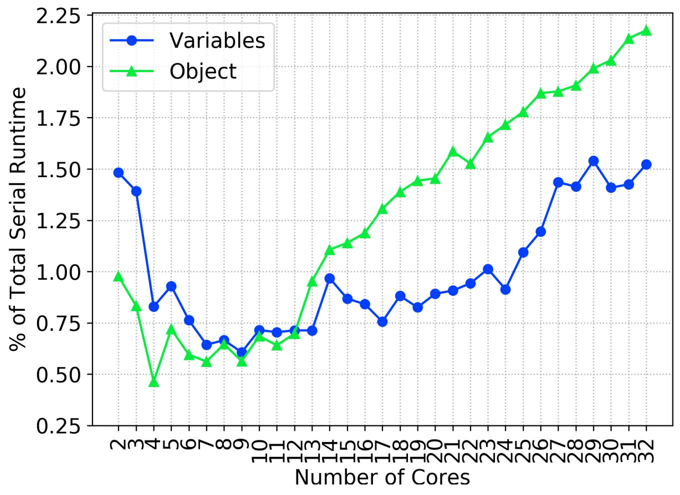
4.2 The Right Granularity for the Number of Cores

Granularity is defined as the ratio between the amount of computation time and the amount of communication time in a parallelized task. When the granularity of parallelized tasks is coarse (fine), more (less) computation of tasks is required than communication across processors and for data transfers. A rule of thumb with granularity is that parallel programming benefits when parallelizing tasks of a program comprises a coarse granularity.

In our example, granularity is controlled by the finite number of grid points where each

[12]Note that with a small number of workers like 2 to 4, the parallelization overhead is relatively smaller passing an object than variables. This is in fact due to Python's systemic feature called "pickle" that creates extra overhead. When it "pickles" a function with many arguments in parallelization, it takes a bit more time and memory space for a function with more arguments. In our example, it is more than ten variables passed to the function as arguments, while it is just one argument when passing an object.

Figure 4: Parallelization Overhead in Object-Oriented Programming

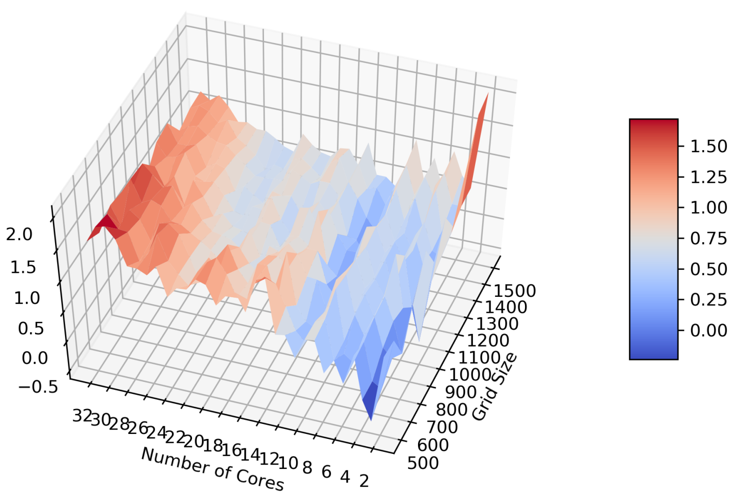


state space is discretized and value/policy functions are evaluated at those points. This implies that the granularity becomes coarse if we set a large number of grid points for state variables since the communication (synchronization) takes place after multiple workers complete evaluating value/policy functions at all grid points at each age. As granularity gets coarse (with a large number of grid points), it is trivial that total computational time and parallelization overhead both increase for any number of parallel workers. But, at the same time, a fair amount of accuracy on the approximation of value/policy functions can be obtained from a large size of grid points. Then our natural question is: what is the right granularity for the number of cores so that parallel computing of a program results in scaling outcome?

Figure 5 shows the parallelization overhead relative to the total serial runtime of selected grid sizes. With a small number of grid points, say N_a =500, relative parallelization overhead becomes almost 2% of total serial runtime with 32 cores. This is because the amount of work that each worker takes on is fairly small that the computing time would be faster than the processing time for creating a pool of workers and transferring data to and from each worker. On the contrary, the relative overhead gradually shrinks to 1% as the grid size rises with a large number of cores. This is because the amount of time required to create a pool of workers and to transfer data should be relatively faster than processing time for computation with a coarse granularity. There is an opposite performance for the small number of cores as well. For example, with only 2 cores, smaller number of grid points only takes less than 0.5% of total runtime, while the overhead rises up to 2% as the number of grid points rises. Therefore, it is instructive to state that the number of cores used and the size of granularity are in a positive relation. This result implies that the parallelization overhead becomes smaller when granularity is fine with the small number of workers or when granularity is coarse with the large number of workers.

Employing more efficient solution methods, which is designed to substantially reduce a serial runtime, definitely make granularity finer with fixed number of grid size and hence using parallel computing may not yield scaling outcome. For example, The VFI solution method has been modified by an endogenous grid method (Carroll (2005) and Barillas and Fernández-Villaverde (2007)) which solves the same problem at 1/100 of VFI runtime. Even with an efficient solution method, however, our result is still informative that the grid size can be further expanded thereby improving the accuracy of approximation as well as yielding a scaling effect from parallel computing. Thus, the right granularity and the right size of parallelism should be jointly determined conditional on the choice of solution method.

Figure 5: Parallelization Overhead by Granularity

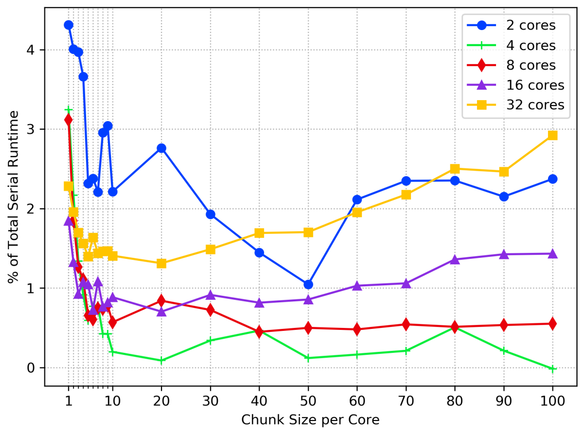


4.3 The Right Chunk of Tasks

Chunk size is related but is different from granularity. We distribute the entire tasks to multiple workers by a number of chunks of tasks. On one extreme, we could divide the entire tasks (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 worker as a worker finishes an assigned task. The former case with one large chunk minimizes the overhead due to one time assigning of a chunk of tasks to each worker, while the latter case with smaller piece of chunks allows to prevent the idling of workers. If each task is about the same size of work that completes at about the same execution time (i.e. load balancing), a larger chunk should be benefited by reducing the overhead from fetching chunks of tasks one at a time from the queue. 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 workers waiting for other workers that are working on longer processes. However, it is not easy to detect actual load of each task *ex ante* and therefore is difficult to calculate the accurate size of chunks dispatched to each worker for a perfect load balancing before execution.

Figure 6 shows the parallelization overhead by the size of chunks per core. We keep the grid

Figure 6: Parallelization Overhead by Chunk Size



size to be N_a =1000 as before but only changing chunk size in a parallelization algorithm. It is clear that assigning one large chunk of tasks per core is subject to the idling of workers which raises the parallelization overhead to 2-4% of total serial runtime. As we increase the number of chunks per core, the flexibility rises that the parallelization overhead plummets almost by half. For most of the number of cores, parallelization overhead is moderately flat after 10 chunks of tasks per core. As we employ a large number of cores the overhead gradually rises as the number of chunks increases. While the large number of chunks per core reduces a waiting time of workers, fetching a chunk of tasks from the queue of entire tasks by each worker one at a time may sum up to increase the overhead. Thus, the trade-off between idling of workers and fetching tasks informs us that the parallelization overhead is suppressed to its minimum at around 4-20 chunks of takes per core when a large number of cores is employed. Note that our efficient result in Figure 2 in fact assigns 4 chunks of tasks per core.

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.

Aruoba, S. B., Fernández-Villaverde, J., and Rubio-Ramírez, J. F. (2006). Comparing Solution Methods for Dynamic Equilibrium Economies. *Journal of Economic Dynamics and Control*, 30:2477–2508.

Barillas, F. and Fernández-Villaverde, J. (2007). A Generalization of the Endogenous Grid Method. *Journal of Economic Dynamics and Control*, 31:2698–2712.

Brumm, J. and Scheidegger, S. (2017). Using Adaptive Sparse Grids to Solve High-Dimensional Dynamic Models. *Econometrica*, 85(5):1575–1612.

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.

Haan, W. J. D. and Marcet, A. (1990). Solving the Stochastic Growth Model by Parameterizing Expectations. *Journal of Business & Economic Statistics*, 8(1):31–34.

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

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. (1993). *Perturbation Solution Methods for Economic Growth Models*, pages 80–103. Springer, New York, NY.

Judd, K. L., Maliar, L., and Maliar, S. (2010). A Cluster-Grid Production Method: Solving Problems with High Dimensionality. *NBER Working Paper 15965*.

Judd, K. L., Maliar, L., and Maliar, S. (2011). Numerically Stable and Accurate Stochastic Simulation Approaches for Solving Dynamic Economic Models. *Quantitative Economics*, 2:173–210.

Judd, K. L., Maliar, L., and Maliar, S. (2012). Merging Simulation and Projection Approaches to Solve High-Dimensional Problems. *NBER Working Paper 18501*.

Judd, K. L., Maliar, L., Maliar, S., and Valero, R. (2014). Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain. *Journal of Economic Dynamics and Control*, 44:92–123.

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

Maliar, L. and Maliar, S. (2013). Envelope Condition Method versus Endogenous Grid Method for Solving Dynamic Programming Problems. *Economics Letters*, 120(2):262–266.

Maliar, L. and Maliar, S. (2014). *Chapter 7 - Numerical Methods for Large-Scale Dynamic Economic Models*, volume 3.

Marcet, A. (1988). Solving Non-linear Stochastic Models by Parameterizing Expectations. Working Paper, Carnegie Mellon University.

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