Parallelization Overhead and HPC-Efficient Algorithm

Dongya Koh*

University of Arkansas

November 25, 2018

Abstract

An advent of high-performance computing (HPC) system enables parallel computing more handy and ubiquitous in numerically solving economic models. In fact, despite the introduction of efficient solution methods and the use of efficient programming languages, parallel computing has yet been the last resort for time-consuming complex models to be solved. Nevertheless, it is surprising that very little is known about the full gain of modern computing power via parallel computing. This study therefore shows under what conditions we gain a full benefit from solving an economic model using parallel computing in a HPC system. In particular, we show the source of bottlenecks inherent to the parallel computing in HPC architectures and introduce easily-implementable HPC-efficient algorithms to get around with the overhead.

*dkoh@walton.uark.edu; https://sites.google.com/site/dongyakoh/

1 Introduction

A lifecycle income profile is non-stationary. Hence the state space in old age should be a super set of that in young age. If one chooses the largest state space for all age, then there are some points that young age never reach. On the other hand, if one uses age-dependent state space with uniform grid size for all age, then a wider state space should have coarse grid points.

- HPC environment: AMD cluster, Intel cluster, GPU, shared memory, distributed memory
- High language and low language: Python, C++
- Profiling memory usage, Memory architecture, batch size, pre-dispatch

2 Profiling Solution Methods

VFI as an example: VFI compute a value function over a hyper-cube state space using optimization routine and interpolation. Most of the alternative solution methods compute value and polity functions over a hyper-cube state space and a certain degree of optimization and interpolation. Therefore, profiling VFI applies to all other solution methods.

- How many seconds each code takes?
- Access to memory, CPU use?
- Where to parallelize?
- how to parallelize?

Model: Consumption/Saving

The baseline model assumes that an individual lives until age T, the individual's problem is to choose an amount of saving and consumption:

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

subject to

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

The utility function takes isoelastic preference, $u(c) = \frac{c^{1-\sigma}}{1-\sigma}$

Solution Method: Value Function Iteration

Low Dimension	High Dimension
Value function iteration (VFI)	Smolyak sparse grid method (Krueger & Kubler, 2004, etc.)
Policy function iteration (PFI)	Adaptive grid method (Brumm & Scheidegger, 2017)
Projection method (Judd, 1992, etc.)	Stochastic simulation algorithm (Den Haan & Marcet, 1990, etc.)
Endogenous grid method (EGM) (Carroll, 2005, etc.)	ε-distinguishable set method (Judd, Maliar, Maliar, 2015)
Envelope condition method (Maliar & Maliar, 2013)	Cluster grid method (Judd, Maliar, Maliar, 2015)
Precomputation method (Judd, Maliar, Maliar, & Tsener, 2017)	Perturbation method (Judd & Guu, 1993, etc.)

Table 1: VFI Algorithm and 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 each state $(a_T, e_T) \in \mathcal{A} \otimes \mathcal{E}$, $a_{T+1} = 0$, and compute $c_T = we_T + (1+r)a_T$, $V_T(a_T, e_T) = u(c_T)$	
Step 3.	Computing a household problem at $t < T$	99.7448
	For each state $(a_t, e_t) \in \mathcal{A} \otimes \mathcal{E}$, search for a maximizer a_{t+1} 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.[1] 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 Computing Parallelization Overheads

- The total runtime of a program with n cores:

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

- The optimal number of cores to minimize the runtime:

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

- If the parallelization overhead is approximated as $P(n) = pn^\alpha \quad (p, \alpha > 0)$:

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

- Sources of parallelization overhead:
  1. Communication overhead in the form of synchronization and data communications
  2. Idling due to load imbalances
- Overheads vary by the implemented parallel algorithms and problems to be solved
- Parallelization overhead with n cores can be calculated as

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

  provided that $P(1) = 0$

- Profile parallelization overhead in HPC, GPU, cloud computing (communication overhead, idling, memory use etc.)
- Estimate the parallelization overhead and compute the optimal number of cores, showing it in the graphs.
- What are the conditions under which a large number of cores are not needed? language, solution methods, etc.

- Figure1. Runtime by cores with parallelization overhead and optimal number of cores
- Figure2. 3D graph of parallelization overhead to show the changes in concavity of P(n) to Tp.

When the convexity of $P(n)$ increases, n^* decreases. When the concavity of $P(n)$ increases, n^* increases. When $P(n)$ is not increasing fast by n, then n^* will be higher. When $P(n)$ is increasing fast, Then $T(n)$ will be convex and will start rising after certain n.

4 HPC-Efficient Algorithms

How much we gain from using multiple cores

Task scheduling:

Reduction:

Language	Type	Rel. Exec. Time
C/C++	low-level, fastest with gcc	1.00
Fortran	low-level	1.05
Python	high-level, open-source, growing in popularity	$44\times \sim 270\times$
Julia	high-level, new open-source	$2.64\times \sim 2.70\times$
R	high-level, open-source	$281\times \sim 475\times$
Matlab	high-level, not free, license issue	$9\times \sim 11\times$
Mathematica	high-level	$809\times$

Source: Arouba & Fernandez-Villaverde (2014)

Memory use: