Solving Heterogeneous Agent Model with MPI

Makoto Nakajima

Federal Reserve Bank of Philadelphia

April 2009

Outline of Topics

1. Introduction
2. The Model
3. Solution Method
4. Parallelization
5. Parallel Code
6. Benchmark
7. Concluding Remarks

Introduction

In this lecture, I will do the followings:

1. Show an example of using MPI with Fortran 90.
2. Use a standard heterogeneous agent model à la Aiyagari (1994) as an example.
3. Start with the serial (non-parallel) version of the code, and show how the code is parallelized.
4. Show how much the code runs faster if parallelized.

The Model: Aiyagari (1994)

- Standard incomplete market economy with mass of atomless infinitely-lived agents.
- Individual earnings shock follows a Markov chain.
- An agent receives stochastic earnings each period, and chooses how much to save and consume.
- No labor-leisure choice.
- Can save only in the form of physical capital. Ad-hoc borrowing constraint (set at zero). No state-contingent security allowed to be traded.
- Representative firm has an access to CRS technology.
- Focus on the steady state where interest rate and wage are constant over time.

Solution Method: Discretization

- Asset space is discretized into $na (= 2000)$ grid points.
- Productivity shock can take one of the 7 values ($ne = 7$)
- Each agent is characterized by (e, a), where:
  - $e \in \{1, 2, 3, 4, 5, 6, 7\}$
  - $a \in \{1, 2, 3, ..., 2000\}$
- Therefore, total number of individual states is ni=na*ne=14000.
- Restrict the choice to be on the asset grids. Therefore, the optimization problem for an agent of each type is just choosing the grid point $a' \in \{1, 2, 3, ..., 2000\}$ associated with the highest value.

Solution Method: Main Loop

1. Beginning of Loop 0 Guess $\frac{K}{Y}$. This gives the guess for r and w (remember CRS production technology).
2. Loop 1 Given the prices, using the value function iteration, find the optimal value function $V(e,a)$ and associated optimal decision rule $a' = g_a(e,a)$.
3. Loop 2 Using $g_a(e,a)$ and the Markov transition matrix $p(e,e')$, compute the ergodic distribution of agent types.
4. Using the ergodic distribution, compute the aggregate labor supply and the aggregate capital stock.
5. Compute $\frac{K}{Y}$ associated with the computed aggregate capital stock and aggregate labor supply.
6. End of Loop 0 Compare the guess of $\frac{K}{Y}$ and newly obtained $\frac{K}{Y}$. If not close enough, update the guess of $\frac{K}{Y}$ and go back to the top.

Solution Method: Inside Loop 1

1. Beginning of Loop 1 Take r and w as given. Guess the value function $V_0(e,a)$.
2. Given the prices, and $V_0(e,a)$, update the value function using the Bellman operator. Notice that the optimization problem for each type (e,a) is just choosing the optimal grid point a' out of 2000 grid points.
3. Call the updated value function $V_1(e,a)$.
4. End of Loop 1 Compare $V_0(e,a)$ and $V_1(e,a)$. If not close enough, set $V_0 = V_1$ and go back to the top.

Solution Method: Inside Loop 2

1. Beginning of Loop 2 Take the optimal decision rule $g_a(e, a)$ as given. Guess the type distribution of agents $d_0(e, a)$.
2. Using $g_a(e, a)$ and the Markov transition matrix $p(e, e')$, update the distribution and denote the new distribution as $d_1(e, a)$.
3. End of Loop 2 Compare $d_0(e, a)$ and $d_1(e, a)$. If not close enough, set $d_0 = d_1$ and go back to the top.

Parallelization: Basic Idea

- Suppose we use $nproc = 10$ nodes: $id = 0, 1, ..., 9$
- Remember there are $ni = na * ne = 14000$ types.
- We assign $1400 (= 14000/10)$ for each node.
- Each node updates value only for 1400 types.
- Each node updates distribution only for 1400 types.

Parallelization: Preparation

- Function *iafun*: converts i to a
- Function *iefun*: converts i to e
- Variable $itop = ni/nproc * id + 1$
- Variable $iend = ni/nproc * (id + 1)$
- Each node is in charge of $i = itop, itop + 1, ..., iend$
- Confirm that the entire type space is covered.
- If ni is not a multiple of $nproc$, a bit tricky but the idea is the same.

Parallelization: Pseudo Code [1]

Initialization Block

1. (Start of the code)
2. include 'mpif.h' ($\rightarrow$ link mpi library)
3. (Variable declaration)
4. call MPI_INIT ($\rightarrow$ initialization)
5. call MPI_COMM_RANK ($\rightarrow$ id assigned)
6. call MPI_COMM_SIZE ($\rightarrow$ nproc set)
7. if (id==0) open 'output file'
8. (set iafun, iefun, itop, iend)

- Be careful not to allow multiple nodes to access to the same file simultaneously.

Parallelization: Pseudo Code [2]

Loop 0 (main loop for prices)

1. All nodes compute k_0 (initial guess for $\frac{K}{Y}$)
2. Beginning of loop 0
3. (Loop 1: obtain optimal value function)
4. (Loop 2: obtain ergodic distribution)
5. All nodes compute k_1 (updated $\frac{K}{Y}$)
6. If k_0 and k_1 are close, get out of the loop
7. Update k_0
8. End of loop 0
9. call MPI_FINALIZE

- Make sure that all nodes have the same prices at any point of time!

Parallelization: Pseudo Code [3]

Loop 1 (value function iteration)

1. All nodes set the initial guess V_0
2. Beginning of loop 1
3. Each node computes updated value V_1 for $i = itop, ..., iend$
4. call MPI_ALLGATHER ($\rightarrow$ sharing updated value V_1)
5. If V_0 and V_1 are close, get out of the loop
6. Update $V_0 = V_1$
7. End of loop 1

- Make sure that V_0 is shared among all nodes all the time!
- Notice that each node *id* has optimal decision rule only for $i = itop, ..., iend$.

Parallelization: Pseudo Code [4]

Loop 2 (Ergodic Distribution)

1. All nodes set the initial guess d_0
2. Beginning of loop 2
3. Each node updates the distribution and obtain d_1 only for agents of type $i = itop, ..., iend$
4. call MPI_ALLREDUCE ($\rightarrow$ sum up d_1 across all nodes)
5. If d_0 and d_1 are close, get out of the loop
6. Update $d_0 = d_1$
7. End of loop 2

- Make sure that V_0 is shared among all nodes all the time!
- Notice that each node *id* has optimal decision rule only for $i = itop, ..., iend$.

Parallel Code: Line-by-Line [1]

See the accompanying code aiyagari_disc_mpi.f90. Some remarks below:

- Line 19: Start with 'mpif.h' (in case of Fortran). This adds to the code a header that is necessary to use MPI.
- Line 67: Designate the node (number 0) which does all the file handling.
- Line 68: ierr stores the error code which is returned whenever MPI subroutines are called. This is only for MPI for Fortran. No need if C is used. Common mistake.
- Line 68: nproc stores the number of nodes used.

Parallel Code: Line-by-Line [2]

- Line 68: id stores the id number for each node. Takes the value between 0 and (nproc-1)
- Line 68: nii stores the number of individual states allocated to each node. This is set at the smallest value satisfying nii*nproc≥ni
- Line 68: itop and iend store the position where the individual states allocated to the particular node starts and ends.
- Line 69: val_ind and val_agg are used to share updated value across all the nodes.
- Line 70: dist_temp is used to share updated distribution across all the nodes.

Parallel Code: Line-by-Line [3]

- Line 93: mpi_init has to be called at first. Don't think.
- Line 94: mpi_rank returns each node its id number. Different for each node.
- Line 95: mpi_size returns the number of nodes currently used.
- Line 98: Allowing only the id=0 to write to an output file. Otherwise, all the nodes will try to open and write to the same output file. Remember all the nodes share the same hard drive.

Parallel Code: Line-by-Line [4]

- Line 112: compute nii, which is the number of jobs (individual states) allocated to each node. It's important that approximately the same number of jobs are allocated to each node, as all the node has the same computational power, and all the jobs require approximately the same amount of time to finish.
- Line 114:115: There are ni individual states. Each node is in charge of the state between itop:iend. These numbers are different for each node and together cover all of 1:ni.
- Line 182: Instead of covering 1:ni, each node updates value function only for states i=itop:iend.

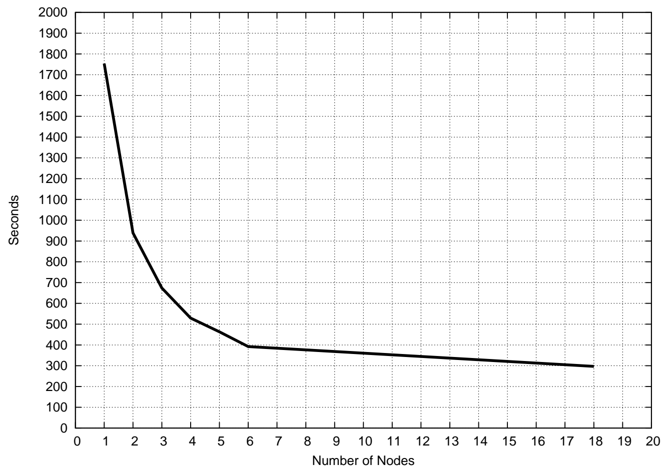
Parallel Code: Line-by-Line [5]

- Line 199:201: Updated value function is shared across all the nodes, using collective communication command mpi_allgather. Notice that the decision rule need not be shared for this algorithm. Don't send unnecessary data.
- Line 248: Instead of covering i=1:ni, each node updates the distribution for i=itop:iend.
- Line 256:258: Using collective communication command mpi_allreduce to share the updated distribution.
- Line 348: Always end your program with mpi_finalize.

Benchmark: Overview

- Use my `susquehanna` cluster for assessing performance of the parallel code.
- `susquehanna` consists of 7 nodes (1 frontend + 6 compute nodes).
- Each node is equipped with Athlon2600 (1.8Ghz)... Machine from the 20th century.
- Gigabit ethernet network.
- Compiled with Intel Compiler For Linux Version 7. only $-O3$ option is used.
- First of all, I ran the sequential code using (of course) one node. Then ran the parallel code with 2-6 nodes.
- I also ran the parallel code using In-Koo Cho's 18 nodes cluster (9 dual-Opteron) at UIUC.

Benchmark: Result: Figure



Benchmark: Result: Table

Number of Nodes	Time (Minutes:Seconds)	(Seconds)	Normalized
1 (Sequential code)	29:14	1754	1.00
2 (Susquehanna)	15:39	939	0.54
3 (Susquehanna)	11:14	674	0.38
4 (Susquehanna)	8:49	529	0.30
5 (Susquehanna)	7:43	463	0.26
6 (Susquehanna)	6:32	392	0.22
18 (Cho)	4:57	297	0.17

- Pretty large gain from parallelization.
- The marginal gain is diminishing, as the time spent for message passing gets larger and larger.

Concluding Remarks

- Gain from parallel:
  - Task parallelism (Distribute large tasks across nodes)
  - Data parallelism (Distribute large data across nodes)
- Loss from parallel: Time spent for passing data.
- Always compare the gain and loss from parallelization!
- For example, actually, the code runs faster if the loop for computing ergodic distribution is not parallelized.
- Try to get the fastest and robust code before parallelizing.
- Study the optimization option of your compiler. The gain from using an appropriate compiling option is huge.

References

Aiyagari, S. Rao, "Uninsured Idiosyncratic Risk and Aggregate Saving," *Quarterly Journal of Economics*, 1994, *109* (3), 659–684.