Introduction to MPI

Makoto Nakajima
Department of Economics
University of Pennsylvania

April 2002
Plan of Talk

• Motivation (1): Typical Computation Problem in Macro

• Motivation (2): Why MPI?

• Style of Fortran Code with MPI

• Sample Code (1): One-to-one communications

• Sample Code (2): Collective communications

• Back to the Motivating Problem
Motivation (1): A Typical Quantitative Macro Program

- Initialization
- Guess equilibrium prices
- Loop for value functions
  - Set a guess for value function
  - For each state, find optimal decision rule
  - Derive a new guess of value function
  - If the new guess is close to the old guess, get out of the loop
- Set initial type distribution
- Loop for type distribution
  - Set a guess for type distribution
  - Update the type distribution using optimal decision rules
  - If the new type distribution is close to the old one, get out of the loop
- Calculate the new prices, based on the updated type distribution
- If the new prices are close to the old prices, you are done
- Finalization
Motivation (1)

- For the program like above, loops of value functions and type distribution are usually the hot spots (the most time-consuming parts).

- Inside these loops, the computer is implementing the same procedure with different state variables (Parallel structure).

- If a hot spot has parallel structure, we can make the program faster by dividing the job and letting processors to do the different parts of the job simultaneously.

- This is what parallel programming is for (This is called loop parallelism and it's just a basic part of parallel programming).
Motivation (2)

- Standard Fastest Environment
  - Hardware: PC with very fast processor
  - Software: C or Fortran

- In order to have even faster environment, need to have:
  - Hardware: non standard processor: Better to have Cost performance and Scalability,
  - Software: non standard language: Better to have Speed and Portability

- What is MPI (Message Passing Interface)?
  - A library for C or Fortran which enables processors that do not share memory to send and receive data each other
Why MPI?

There are several choices of faster environment. The most popular ones are the followings:

- Vector machine (supercomputer) with High Performance Fortran: Fast, Simple, Super-expensive, non-portable, non-scalable

- Parallel processors: Less expensive
  - Shared memory with OpenMP: Less fast, Simple, Portable, Less scalable
  - Non-shared memory with MPI: Fast, Less simple, Portable, Scalable
Style of Fortran 90 Code with MPI

How does a Fortran 90 code with MPI look like?

- Some subroutines are added to standard Fortran 90 code.
- All the processors implement the same program.
- Each of the processors is given her id number.
- A task can be divided by indicating which processor does which part.
- Data are separately held in each processor. All the transfers of data have to be explicitly implemented using MPI subroutines.
From Sample Code 1: Key Elements

- **nproc**: number of processors in the current MPI environment

- **id**: identification number given to each processor (0 to nproc-1)

- **communicator**: set of processors

- **tag**: identifier for each operation

- **ierr**: error code (just for Fortran version of MPI)

- **status(MPI_STATUS_SIZE)**: status of each processor

- **data type**: MPI_INTEGER, MPI_DOUBLE_PRECISION, etc
From Sample Code 1: Key Routines

• MPI_INIT(ierr): initialize MPI environment

• MPI_COMM_SIZE(comm,nproc,ierr): Find out how many processors there are in the environment

• MPI_COMM_RANK(comm,id,ierr) Find out which processor I am (0 to nproc-1)

• MPI_SEND(data, count, type, dest, tag, comm, ierr): Send data to other processor

• MPI_RECV(data, count, type, root, tag, comm, status, ierr): Receive data from other processor

• MPI_FINALIZE(ierr): finalize MPI environment
Subroutines of collective operations (notice that so far we have one-to-one operations)

- **MPI_REDUCE**(send\_data, recv\_data, count, type, operation, root, comm, ierr): (in case of summation, of course, there are many other operations available) Sum up ”send\_data” of all the processors and save the data into ”recv\_data” in ”root” processor

- **MPI_BCAST**(data, count, type, root, comm, ierr): Broadcast the ”data” of ”root” to all the processors
  
  - Notice that this one subroutine includes both sending and receiving procedure
  
  - Therefore, this subroutine must be called by all the processors
Back to Our Problem (1)

With MPI, loops for value functions and distribution can be programmed as follows and it is much faster

- Loop for value functions
  - Set a guess for value functions
  - Divide the state space and assign a subspace for each processor
  - Each processor solves optimal decision rule of the assigned state space
  - Each processor derives a new guess of value function
  - Send the updated value functions each other so that all the processors have the updated guess of the entire state
  - If the new guess is close to the old guess, get out of the loop
Back to Our Problem (2)

- Loop for type distribution
  - Set a guess for type distribution
  - Assign subset of agents for each processor
  - Each processor updates the type of the assigned agents
  - Send the updated types of agents each other
  - If the new distribution is close to the old one, end
PROGRAM sum_1_100
   include 'mpif.h'

   !******* variable declaration *******
   integer:: proc_no     !specify the number of processor
   integer:: partial_sum !store partial sums
   integer:: total_sum   !store total sum
   integer:: int_top     !top integer of summation
   integer:: int_end     !last integer of summation
                         !(different for each processor)
   integer:: int_counter !used to count the integer to be summed up
   integer, dimension(4):: recv_sum
                         !used for id=0 to receive data from proc 1-4

   !******* variables related MPI *******
   integer:: ierr  !return error message from MPI subroutines
   integer:: id    !identification number of each processor
   integer:: nproc !number of processors
   integer:: tag   !id number for particular jobs

   integer, dimension(MPI_STATUS_SIZE):: STATUS
                           !used to record the status of each processors.
                           !MPI_STATUS_SIZE is defined in the MPI library

   !******* initialize MPI system *******
   !initialization of MPI environment
   call MPI_INIT(ierr)

   !returns id number for each processor
   !(different for each processor)
   call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)

   !returns the number of processors minus 1
   !(id starts from 0)
   call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)

   !******* self introduction *******
print *, 'Hello! I am processor ',id
!Notice that each processor prints different id number

******* summing up separately *******
total_sum=0
partial_sum=0

!Be careful! The value of int_top and int_end are different
!for each processor because different processors have different
!id number.
int_top=20*id+1
int_end=20*(id+1)

!Of course, the sums are also different among processors
if (id<=4) then
  do int_counter=int_top, int_end
    partial_sum=partial_sum+int_counter
  end do
end if

******* calculate total sum by sending the partial sums *******
!Processors 1 to 4 send the sums to processor 0
if (id>=1 .and. id<=4) then
  tag=100+id
  call MPI_SEND(partial_sum,1,MPI_INTEGER,0,tag,&
    MPI_COMM_WORLD,ierr)
end if

!Processor 0 receives the partial sums from processors
!1 to 4, sums them up, and send back to processors 1 to 4
if (id==0) then
  do proc_no=1, 4
    tag=100+proc_no
    call MPI_RECV(recv_sum(proc_no),1,MPI_INTEGER,proc_no,&
      tag,MPI_COMM_WORLD,STATUS,ierr)
  end do
  total_sum=sum(recv_sum(1:4))+partial_sum
  do proc_no=1, 4
    tag=proc_no+200
    call MPI_SEND(total_sum,1,MPI_INTEGER,proc_no,tag,&
      MPI_COMM_WORLD,ierr)
end do

end if

!Processes 1 to 4 receives the result
! id>=1 .and. id<=4 ) then
    tag=id+200
    call MPI_RECV(total_sum,1,MPI_INTEGER,0,tag,&
    MPI_COMM_WORLD,STATUS,ierr)
end if

!******** write out the result ********
!Notice the processors after 5 is also running this program
!but does not do anything significant in the program.
print *,'This is a report from processor ',id
print *,'  my partial sum is',partial_sum
print *,'  my total   sum is',total_sum

!******* finalization of MPI environment
call MPI_FINALIZE(ierr)

END PROGRAM sum_1_100
Hello! I am processor 0
This is a report from processor 0
  my partial sum is 210
  my total sum is 5050

Hello! I am processor 4
This is a report from processor 4
  my partial sum is 1810
  my total sum is 5050

Hello! I am processor 8
This is a report from processor 8
  my partial sum is 0
  my total sum is 0

Hello! I am processor 2
This is a report from processor 2
  my partial sum is 1010
  my total sum is 5050

Hello! I am processor 1
This is a report from processor 1
  my partial sum is 610
  my total sum is 5050

Hello! I am processor 5
This is a report from processor 5
  my partial sum is 0
  my total sum is 0

Hello! I am processor 3
This is a report from processor 3
  my partial sum is 1410
  my total sum is 5050

Hello! I am processor 6
This is a report from processor 6
  my partial sum is 0
  my total sum is 0

Hello! I am processor 7
This is a report from processor 7
  my partial sum is 0
  my total sum is 0
PROGRAM sum_1_100
  include 'mpif.h'

  !******** variable declaration ********
  integer:: partial_sum, total_sum
  integer:: int_top, int_end, int_counter

  !******** variables related MPI ********
  integer:: ierr  !return error message from MPI subroutines
  integer:: id    !identification number of each processor
  integer:: nproc !number of processors
  integer, dimension(MPI_STATUS_SIZE):: STATUS

  !******** initialize MPI system ********
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, id, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)

  !******** self introduction ********
  print *, 'Hello! I am processor number ',id

  !******** summing up separately ********
  total_sum=0
  partial_sum=0
  int_top=20*id+1
  int_end=20*(id+1)
  if (id<=4) then
    do int_counter=int_top, int_end
      partial_sum=partial_sum+int_counter
    end do
  end if

  !******** calculating total sum directly by MPI_REDUCE ********
  call MPI_REDUCE(partial_sum,total_sum,1,MPI_INTEGER,MPI_SUM,0,&
    MPI_COMM_WORLD,ierr)

  !******** broadcast the total sum to all the processors ********
call MPI_BCAST(total_sum,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

!******* write out the result *******
print *, 'This is a report from processor ', id
print *, ' my partial sum is', partial_sum
print *, ' my total sum is', total_sum

!******* finalization of MPI environment
call MPI_FINALIZE(ierr)

END PROGRAM sum_1_100
SAMPLE OUTPUT FROM SAMPLE PROGRAM 2

Hello! I am processor number 0
This is a report from processor 0
  my partial sum is 210
  my total   sum is 5050

Hello! I am processor number 1
This is a report from processor 1
  my partial sum is 610
  my total   sum is 5050

Hello! I am processor number 5
This is a report from processor 5
  my partial sum is 0
  my total   sum is 5050

Hello! I am processor number 3
This is a report from processor 3
  my partial sum is 1410
  my total   sum is 5050

Hello! I am processor number 7
This is a report from processor 7
  my partial sum is 0
  my total   sum is 5050

Hello! I am processor number 2
This is a report from processor 2
  my partial sum is 1010
  my total   sum is 5050

Hello! I am processor number 4
This is a report from processor 4
  my partial sum is 1810
  my total   sum is 5050

Hello! I am processor number 6
This is a report from processor 6
  my partial sum is 0
  my total   sum is 5050

Hello! I am processor number 8
This is a report from processor 8
  my partial sum is 0
  my total   sum is 5050