MPI version of the Serial Code With One-Dimensional Decomposition

Timothy H. Kaiser, Ph.D.
tkaiser@mines.edu
Overview

We will choose one of the two dimensions and subdivide the domain to allow the distribution of the work across a group of distributed memory processors.

We will focus on the principles and techniques used to do the MPI work in the model.
Examples at

http://hpc.mines.edu/examples

or enter the commands:

mkdir examples

cd examples

wget http://hpc.mines.edu/examples/examples.tgz

For this session go to the “stommel” directory
STEP1: introduce the MPI environment

- Need to include “mpif.h” or use mpi to define MPI constants
- Need to define our own constants
  - numnodes - how many processors are running
  - myid - Which processor am I
  - mpi_err - error code returned by most calls
  - mpi_master - the id for the master node
STEP1: introduce the MPI environment

module mympi
  use mpi
  ! include "mpif.h"
  integer numnodes,myid,mpi_err
  integer, parameter::mpi_master=0
end module
STEP1: Start the MPI environment

- Add the following to your program
  - call MPI_INIT( mpi_err )
  - call MPI_COMM_SIZE(MPI_COMM_WORLD,numnodes, mpi_err)
  - call MPI_COMM_RANK(MPI_COMM_WORLD, myid, mpi_err)
  - write(*,*)’from ‘, myid,’numnodes=‘,numnodes

- To stop, add the following next
  - call MPI_Finalize(mpi_err)
We read the data on processor 0 and send to the others

```fortran
if(myid .eq. mpi_master)then
    read(*,*) nx, ny
    read(*,*) lx, ly
    read(*,*) alpha, beta, gamma
    read(*,*) steps
endif
```

We use MPI_BCAST to send the data to the other processors. We use 8 calls. Can you do it in 2?
Domain Decomposition (1d)

Physical domain is sliced into sets of columns so that computation in each set of columns will be handled by different processors. Why do columns and not rows?

<table>
<thead>
<tr>
<th>Serial Version</th>
<th>Parallel Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cells on one processor</td>
<td>node 0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Domain Decomposition (1d)

- We set our array bounds differently on each processor so that:
  - We take our original grid and break it into numnodes subsections of size nx/numnodes
  - Each processor calculates for a different subsection of the grid
- No two processors calculate psi for the same (l,j)
- We add special boundary cells for each subsection of the grid called ghost cells
- The values for the ghost cells are calculated on neighboring processors and sent using MPI calls.
Domain Decomposition (1d)

With ghost cells our decomposition becomes...

<table>
<thead>
<tr>
<th>Serial Version</th>
<th>Parallel Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cells on one processor</td>
<td>node 0</td>
</tr>
</tbody>
</table>

*Illustration showing the distribution of cells across processors.*
Domain Decomposition (1d)

How and why are ghost cells used?

Node 0 allocates space for psi(0:7,0:3) but calculates psi(1:6,1,2)
Node 1 allocates space for psi(0:7,2:5) but calculates psi(1:6,3,4)
Node 2 allocates space for psi(0:7,4:7) but calculates psi(1:6,5,6)

To calculate the value for psi(4,4) node1 requires the value from psi(4,3),psi(5,4),psi(3,4),psi(4,5)

Where does it get the value for psi(4,5)? From node2, and it holds the value in a ghost cell
Domain Decomposition (1d)

Source code for setting up the distributed grid with ghost cells

! we stripe the grid across the processors

\[
\begin{align*}
i_1 &= 1 \\
i_2 &= n_y \\
dj &= \frac{\text{real}(n_x, b_8)}{\text{real}(\text{numnodes}, b_8)} \\
&= \text{nint}(1.0_b8 + \text{myid} \times dj) \\
&= \text{nint}(1.0_b8 + (\text{myid} + 1) \times dj) - 1 \\
\text{write}(*, 101) &\text{myid, i1, i2, j1, j2}
\end{align*}
\]

101 format("myid= ", i3, 3x, &
" (" , i3, " <= i <= " , i3, ") , ", &
" (" , i3, " <= j <= " , i3, ")")

! allocate the grid to \((i_1-1:i_2+1, j_1-1:j_2+1)\) this includes boundary cells

\[
\text{allocate}(\text{psi}(i_1-1:i_2+1, j_1-1:j_2+1))
\]

Try adding this to your program. What do you get?
Ghost cell updates

When do we update ghost cells?

Each trip through our main loop we call do_transfer to update the ghost cells

Our main loop becomes...

```fortran
  do i=1,steps
    call do_jacobi(psi,new_psi,mydiff,i1,i2,j1,j2)
    call do_transfer(psi,i1,i2,j1,j2)
    write(*,*)i,diff
  enddo
```
How do we update ghost cells?

Processors send and receive values to and from neighbors.

Need to exchange with left and right neighbors except processors on far left and right only transfer in 1 direction.

Trick 1 to avoid deadlock:

Even # processors
send left
receive from left
send right
receive from right

Odd # processors
receive from right
send to right
receive for left
send to left

Trick 2 to handle the end processors
Send to MPI_PROC_NULL instead of a real processor.
How do we update ghost cells?

! How many cells are we sending
num_x=i2-i1+3

! Where are we sending them
myleft=myid-1
myright=myid+1
if(myleft .le. -1)myleft=MPI_PROC_NULL
if(myright .ge. numnodes)myright=MPI_PROC_NULL
How do we update ghost cells?

For even-numbered processors...

if(even(myid))then
    ! send to left
    call MPI_SEND(psi(:,j1), num_x,MPI_DOUBLE_PRECISION,myleft, &
                 100,MPI_COMM_WORLD,mpi_err)
    ! rec from left
    call MPI_RECV(psi(:,j1-1),num_x,MPI_DOUBLE_PRECISION,myleft, &
                 100,MPI_COMM_WORLD,status,mpi_err)
    ! rec from right
    call MPI_RECV(psi(:,j2+1),num_x,MPI_DOUBLE_PRECISION,myright, &
                 100,MPI_COMM_WORLD,status,mpi_err)
    ! send to right
    call MPI_SEND(psi(:,j2), num_x,MPI_DOUBLE_PRECISION,myright, &
                 100,MPI_COMM_WORLD,mpi_err)
else
How do we update ghost cells?

For odd-numbered processors...

Else ! we are on an odd column processor
! rec from right
   call MPI_RECV(psi(:,j2+1),num_x,MPI_DOUBLE_PRECISION,myright, &
                 100,MPI_COMM_WORLD,status,mpi_err)
! send to right
   call MPI_SEND(psi(:,j2), num_x,MPI_DOUBLE_PRECISION,myright, &
                  100,MPI_COMM_WORLD,mpi_err)
! send to left
   call MPI_SEND(psi(:,j1), num_x,MPI_DOUBLE_PRECISION,myleft, &
                  100,MPI_COMM_WORLD,mpi_err)
! rec from left
   call MPI_RECV(psi(:,j1-1),num_x,MPI_DOUBLE_PRECISION,myleft, &
                 100,MPI_COMM_WORLD,status,mpi_err)
endif
How do we update ghost cells?
It’s a 4-stage operation

Example with 4 nodes:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td>Send left to MPI_PROC_NULL</td>
<td>Receive right from Proc 2</td>
<td>Send left to Proc 1</td>
<td>Receive right from MPI_PROC_NULL</td>
</tr>
<tr>
<td>Stage 2</td>
<td>Receive left from MPI_PROC_NULL</td>
<td>Send right to Proc 2</td>
<td>Receive left from Proc 1</td>
<td>Send right to MPI_PROC_NULL</td>
</tr>
<tr>
<td>Stage 3</td>
<td>Receive right from Proc 1</td>
<td>Send right to Proc 0</td>
<td>Receive right from Proc 3</td>
<td>Send right to Proc 2</td>
</tr>
<tr>
<td>Stage 4</td>
<td>Send right to Proc 1</td>
<td>Receive right from Proc 0</td>
<td>Send right to Proc 3</td>
<td>Receive right from Proc 2</td>
</tr>
</tbody>
</table>
Only a few other modifications

Force and do_jacobi are not modified
We modify the boundary condition routine only
to set value for true boundaries and ignore ghost cells

```fortran
subroutine bc(psi,i1,i2,j1,j2)
! sets the boundary conditions
! input is the grid and the indices for the interior cells
    use numz
    use mympi
    use input, only : nx,ny
    implicit none
    real(b8),dimension(i1-1:i2+1,j1-1:j2+1):: psi
    integer,intent(in):: i1,i2,j1,j2
! do the top edges
    if(i1 .eq. 1) psi(i1-1,:)=0.0_b8
! do the bottom edges
    if(i2 .eq. ny) psi(i2+1,:)=0.0_b8
! do left edges
    if(j1 .eq. 1) psi(:,j1-1)=0.0_b8
! do right edges
    if(j2 .eq. nx) psi(:,j2+1)=0.0_b8
end subroutine bc
```
Residual

• In our serial program, the routine do_jacobi calculates a residual for each iteration

• The residual is the sum of changes to the grid for a jacobi iteration

• Now the calculation is spread across all processors

• To get the global residual, we can use the MPI_Reduce function

```fortran
call MPI_REDUCE(mydiff,diff,1,MPI_DOUBLE_PRECISION, & MPI_SUM,mpi_master,MPI_COMM_WORLD,mpi_err)
if(myid .eq. mpi_master)write(*,*)i,diff
```
Our main loop is now…

Call the do_jacobi subroutine
Update the ghost cells
Calculate the global residual

```fortran
  do i=1,steps
      call do_jacobi(psi,new_psi,mydiff,i1,i2,j1,j2)
      call do_transfer(psi,i1,i2,j1,j2)
      call MPI_REDUCE(mydiff,diff,1,MPI_DOUBLE_PRECISION, &
                      MPI_SUM,mpi_master,MPI_COMM_WORLD,mpi_err)
      if(myid .eq. mpi_master)write(*,*)i,diff
  enddo
```
Final change

We change the write_grid subroutine so that each node writes its part of the grid to a different file.

Function unique returns a file name based on an input string and the node number.

We change the open statement in write_grid to:

```fortran
open(18, file=unique("out1d_")), recl=max(80, 15*((jend-jstart)+3)+2))
```
Unique

We add an interface to unique in the module face

Unique is the function:

```fortran
function unique(name)
use numz
use mympi
character (len=*) name
character (len=20) unique
character (len=80) temp
if(myid .gt. 99)then
   write(temp,"(a,i3)")trim(name),myid
else
   if(myid .gt. 9)then
      write(temp,"(a,'0',i2)")trim(name),myid
   else
      write(temp,"(a,'00',i1)")trim(name),myid
   endif
endif
unique=temp
return
end function unique
```
Unique (easier)

We add an interface to unique in the module face

Unique is the function:

```fortran
function unique(name)
  use numz
  use mympi
  character (len=*) name
  character (len=20) unique
  character (len=80) temp
  write(temp, "(a,i4.4)") trim(name), myid
  unique=temp
return
end function unique
```
Try it!

- Compile
- make stf_00
- Run (something like this)
  - stf_00 < stommel.in
  - mpiexec -np 4 ./stf_01 < stommel.in
Suggested exercises

Study, compile, and run the program st_01 on various numbers of processors

Change it to use 2 or 1 MPI_bcast calls instead of 8
Hint: (The "correct" way to do it with 1 call is to use F90- and MPI-derived data types)

Do the decomposition in rows

Do periodic boundary conditions

Modify the write_grid routine to output the whole grid from node 0
2d decomposition

The program is almost identical

We now have our grid distributed in a block fashion across the processors instead of striped

We can have ghost cells on 1, 2, 3 or 4 sides of the grid held on a particular processor
Example 2d Decomposition
50 x 50 grid on 4 processors

Grid on each processor is allocated to:

- pid= 0  ( 0 <= i <= 25), ( 0 <= j <= 26)
- pid= 1  ( 0 <= i <= 26), ( 25 <= j <= 51)
- pid= 2  ( 25 <= i <= 51), ( 0 <= j <= 26)
- pid= 3  ( 25 <= i <= 51), ( 25 <= j <= 51)

But each processor calculates only for:

- pid= 0  ( 1 <= i <= 25), ( 1 <= j <= 25)
- pid= 1  ( 1 <= i <= 25), ( 26 <= j <= 50)
- pid= 2  (26 <= i <= 50), ( 1 <= j <= 25)
- pid= 3  (26 <= i <= 50), ( 26 <= j <= 50)

Extra cells are ghost cells

Grid Distributed across
4 processors
Only three changes need to be made to our program

Given an arbitrary number of processors, find a good topology (number of rows and columns of processors)

Make new communicators to allow for easy exchange of ghost cells
  Set up communicators so that every processor in the same row is in a given communicator
  Set up communicators so that every processor in the same column is in a given communicator

Add the up/down communication
Given an arbitrary number of processors, find a good topology (number of rows and columns of processors)

\[
nrow = \text{nint}(\sqrt{\text{numnodes}}) \\
ncol = \text{numnodes} / nrow \\
\text{do while (nrow*ncol .ne. numnodes)} \\
\hspace{1cm} nrow = nrow + 1 \\
\hspace{1cm} ncol = \text{numnodes} / nrow \\
\text{enddo} \\
\text{if(nrow .gt. ncol)then} \\
\hspace{1cm} i = ncol \\
\hspace{1cm} ncol = nrow \\
\hspace{1cm} nrow = i \\
\text{endif} \\
\text{myrow = myid/ncol + 1} \\
\text{mycol = myid - (myrow-1)*ncol + 1}
\]
Make new communicators to allow for easy exchange of ghost cells

! make the row and col communicators
! all processors with the same row will be in the same ROW_COMM
call MPI_COMM_SPLIT(MPI_COMM_WORLD,myrow,mycol,ROW_COMM,mpi_err)
call MPI_COMM_RANK( ROW_COMM, myid_row, mpi_err )
call MPI_COMM_SIZE( ROW_COMM, nodes_row, mpi_err )

! all processors with the same col will be in the same COL_COMM
call MPI_COMM_SPLIT(MPI_COMM_WORLD,mycol,myrow,COL_COMM,mpi_err)
call MPI_COMM_RANK( COL_COMM, myid_col, mpi_err )
call MPI_COMM_SIZE( COL_COMM, nodes_col, mpi_err )

! find id of neighbors using the communicators created above
mytop =myid_col-1;if( mytop .lt. 0 )mytop =MPI_PROC_NULL
mybot =myid_col+1;if( mybot .eq. nodes_col)mybot =MPI_PROC_NULL
myleft =myid_row-1;if( myleft .lt. 0 )myleft =MPI_PROC_NULL
myright=myid_row+1;if( myright .eq. nodes_row)myright=MPI_PROC_NULL
if(even(myid_row)) then
    ! send to top
    call MPI_SEND(psi(i1,:),num_y,MPI_DOUBLE_PRECISION,mytop,  
    &     10, COL_COMM,mpi_err)
    ! rec from top
    call MPI_RECV(psi(i1-1,:),num_y,MPI_DOUBLE_PRECISION,mytop,  
    &     10,COL_COMM,status,mpi_err)
    ! rec from bot
    call MPI_RECV(psi(i2+1,:),num_y,MPI_DOUBLE_PRECISION,mybot,  
    &     10,COL_COMM,status,mpi_err)
    ! send to bot
    call MPI_SEND(psi(i2,:),num_y,MPI_DOUBLE_PRECISION,mybot,  
    &     10, COL_COMM,mpi_err)
else
Communication up/down (continued)

! rec from bot
    call MPI_RECV(psi(i2+1,:),num_y,MPI_DOUBLE_PRECISION,mybot, &
                 10,COL_COMM,status,mpi_err)

! send to bot
    call MPI_SEND(psi(i2,:),num_y,MPI_DOUBLE_PRECISION,mybot, &
                  10,COL_COMM,mpi_err)

! send to top
    call MPI_SEND(psi(i1,:),num_y,MPI_DOUBLE_PRECISION,mytop, &
                  10,COL_COMM,mpi_err)

! rec from top
    call MPI_RECV(psi(i1-1,:),num_y,MPI_DOUBLE_PRECISION,mytop, &
                  10,COL_COMM,status,mpi_err)

endif