Scientific Computing on Parallel Machines

Cálculo científico en ordenadores paralelos

Markus Uhlmann

CIEMAT

www.ciemat.es/sweb/comfos/personal/uhlmann

UCM – Abril 2008

Schedule

Part I	Introduction to parallel programming	lecture 1
Part II	Introduction to MPI	
	General introduction	
	& MPI program structure	lecture 2
	Point-to-point communication	lecture 3,4
	Collective communication	lecture 5
	2D Poisson example	lecture 6,7
	Non-contiguous data & Mixed datatypes	lecture 7
	Virtual topologies & Communication subsets	lecture 8
	Use of linear algebra libraries	lecture 9
	Extensions	lecture 10

Point-to-point Communication

Basic communication between PAIRS of processors

- single message
- sent from source processor
- to destination processor

Components of a message

$$\boxed{\text{Message} = \text{Envelope} + \text{Body}}$$

Envelope

- source process id
- destination process id
- communicator id
- tag (to classify messages)

Body

- buffer (message data)
- datatype of message data
- count (number of items)

Sending a message

MPI syntax

- `call MPI_SEND(buffer,icount,dtype,idest,`
  `itag,comm,ierr)`
- `comm` – the communicator
- `buffer` – start address of the buffer to be sent
- `icount` – number of elements to be sent
- `dtype` – datatype contained in `buffer`
- `idest` – the destination processor rank
  valid: 0 $\leq$ `idest` $\leq$ `size-1`
- `itag` – the message tag (integer)
  0 $\leq$ `itag` $\leq$ `maxtag`, (at least 32767)
  `MPI_ATTR_GET(comm,MPI_TAG_UB,maxtag,lflag,ierr)`

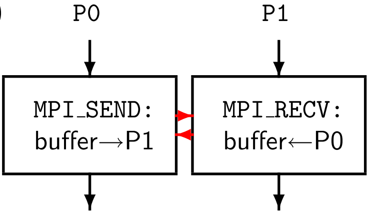
Receiving a message

MPI syntax

- `call MPI_RECV(buffer,icount,dtype,isource,`
  `itag,comm,istat,ierr)`
- `buffer` – start address of the buffer to be received
- `icount` – max. number of elements to be received
- `dtype` – datatype contained in `buffer`
- `isource` – the source processor rank
  valid: $0 \leq$ `isource` $\leq$ `size-1` $\cup$ `MPI_ANY_SOURCE`
- `itag` – the message tag (also: MPI_ANY_TAG)
- `istat` – status of the receive operation

Example: Simple Send & Receive

```
  integer  istat(MPI_STATUS_SIZE)
  ...
  isource=0
  idest=1
  itag=99
  icount=10
  if(myrank.eq.isource)then
    call  MPI_SEND(buffer,icount,MPI_REAL,
&     idest,itag,MPI_COMM_WORLD,ierr)
  elseif  (myrank.eq.idest)then
    call  MPI_RECV(buffer,icount,MPI_REAL,
&     isource,itag,MPI_COMM_WORLD,istat,ierr)
  endif
```



(source)

What happens when Sending/Receiving?

Operation "blocks" until completion

- `MPI_RECV` completion:
  $\longrightarrow$ a message with matching envelope was received
  $\Rightarrow$ data in buffer is available
- `MPI_SEND` completion:
  $\longrightarrow$ message is handed off to MPI
  (either copied to internal buffer or already transferred)
  $\Rightarrow$ buffer can be overwritten

$\rightsquigarrow$ <u>careful organisation</u> of communication patterns!

Send-Receive: Matching of Messages

myrank1:
```
MPI_SEND(buffer,icount1,
        dtype1,idest,itag1,
               icomm1,ierr)
```

myrank2:
```
MPI_RECV(buffer,icount2,
      dtype2,isource,itag2,
        icomm2,istat,ierr)
```

Conditions for matching:

- icomm2 = icomm1
- idest = myrank2
- isource = myrank1 OR isource = MPI_ANY_SOURCE
- itag2 = itag1 OR itag2 = MPI_ANY_TAG
- icount2 $\geq$ icount1
- datatypes are not checked!

Status of Completed Receive Operation

```
integer istat(MPI_STATUS_SIZE)
MPI_RECV(buffer,icount,dtype,MPI_ANY_SOURCE,MPI_ANY_TAG,
                                        icomm,istat,ierr)
```

- entries of status array after completion:

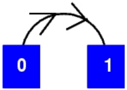
  `istat(MPI_SOURCE)` $\longrightarrow$ source process id
  `istat(MPI_TAG)` $\longrightarrow$ message 'tag'
  `istat(MPI_ERROR)` $\longrightarrow$ error code
  `call MPI_GET_COUNT(istat,dtype,icount2,ierr)`
  $\longrightarrow$ `icount2` is the actual number of received entries

Communication deadlock

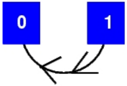
Non-matching communication operations $\longrightarrow$ freeze execution

Safe Communication

(source code)



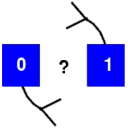
then




```
  if(myrank.eq.0)then
     call MPI_SEND(buffer1,icount,MPI_REAL,1,itag1,
&              MPI_COMM_WORLD,ierr)
     call MPI_RECV(buffer2,icount,MPI_REAL,1,itag2,
&              MPI_COMM_WORLD,istat,ierr)
  elseif(myrank.eq.1)then
     call MPI_RECV(buffer2,icount,MPI_REAL,0,itag1,
&              MPI_COMM_WORLD,istat,ierr)
     call MPI_SEND(buffer1,icount,MPI_REAL,0,itag2,
&              MPI_COMM_WORLD,ierr)
  endif
```

Communication Deadlock

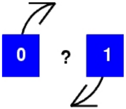
(source code)




```fortran
  if(myrank.eq.0)then
    call MPI_RECV(buffer2,icount,MPI_REAL,1,itag,
&              MPI_COMM_WORLD,istat,ierr)
    call MPI_SEND(buffer1,icount,MPI_REAL,1,itag,
&              MPI_COMM_WORLD,ierr)
  elseif(myrank.eq.1)then
    call MPI_RECV(buffer2,icount,MPI_REAL,0,itag,
&              MPI_COMM_WORLD,istat,ierr)
    call MPI_SEND(buffer1,icount,MPI_REAL,0,itag,
&              MPI_COMM_WORLD,ierr)
  endif
```

Possible Communication Deadlock

(source code)




```
  if(myrank.eq.0)then
     call MPI_SEND(buffer1,icount,MPI_REAL,1,itag,
&             MPI_COMM_WORLD,ierr)
     call MPI_RECV(buffer2,icount,MPI_REAL,1,itag,
&             MPI_COMM_WORLD,istat,ierr)
  elseif(myrank.eq.1)then
     call MPI_SEND(buffer1,icount,MPI_REAL,0,itag,
&             MPI_COMM_WORLD,ierr)
     call MPI_RECV(buffer2,icount,MPI_REAL,0,itag,
&             MPI_COMM_WORLD,istat,ierr)
  endif
```

Send Modes

- standard mode MPI_SEND
  → direct transmission or buffering (depends on size)
- synchronous mode MPI_SSEND
  → completes when reception has begun
- buffered mode MPI_BSEND
  → always buffering
  (add buffer space:
  MPI_BUFFER_ATTACH(buff,size_in_bytes))
- ready mode MPI_RSEND
  → assumes matching MPI_RECV has been posted

(source code)

Ordering of multiple messages

MPI messages are non-overtaking

- subsequent messages between the same pair of processors are sent/received in order

Still, global order can be non-deterministic:

Ordering of multiple messages (continued)

Example:

```
IF(rank.EQ.0) THEN
  CALL MPI_SEND(buf1,count,..,2,tag,..)
  CALL MPI_SEND(buf2,count,..,1,tag,..)
ELSE IF(rank.EQ.1) THEN
  CALL MPI_RECV(buf2,count,..,0,tag,..)
  CALL MPI_SEND(buf2,count,..,2,tag,..)
ELSE IF(rank.EQ.2) THEN
  CALL MPI_RECV(buf1,count,..,MPI_ANY_SOURCE,tag,..)
  CALL MPI_RECV(buf2,count,..,MPI_ANY_SOURCE,tag,..)
END IF
```

→ order of receive at `rank=2` not predetermined

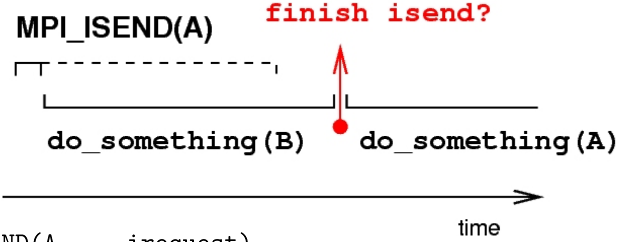
MPI SendRecv function

Grouping of send and receive operation

- a process sends and receives one indepentend message each
- source and destination rank can be identical
- buffers of send and receive must be different mem locations
- syntax:
```
MPI_SENDRECV(sbuff,scount,stype,idest,stag,
             rbuff,rcount,rtype,isource,rtag,
             comm,ierr)
```
- use `MPI_SENDRECV_REPLACE` for identical buffers

Non-blocking Communication

Overlapping communication & computation




```
call MPI_ISEND(A,...,irequest)
call do_something(B)
call MPI_WAIT(irequest,...)
call do_something(A)
```

Non-blocking Communication Syntax

- send/receive operations:

```
call MPI_ISEND(buffer,count,dtype,dest,tag,comm,
  request,err)
call MPI_IRECV(buffer,count,dtype,source,tag,comm,
  request,err)
```

  `request` → 'handle' identifying the posted operation

- waiting for completion:

```
call MPI_WAIT(request,status,err)
```

- testing for completion:

```
call MPI_TEST(request,flag,status,err)
```

  `flag` → 'logical' indicating completion

Non-blocking Communication Caveat

Contents of posted isend/irecv buffer should not be accessed before completion!

Non-blocking Communication Example

(source code)

```
  if(myrank.eq.0)then
    call MPI_IRECV(buffer2,icount,MPI_REAL,1,itag,
&            MPI_COMM_WORLD,irequest,ierr)
    buffer1(1)=0.0
    call MPI_WAIT(irequest,istatus,ierr)
    buffer2(icount)=buffer2(icount)*0.25
  elseif (myrank.eq.1)then
    call MPI_ISEND(buffer1,icount,MPI_REAL,0,itag,
&            MPI_COMM_WORLD,irequest,ierr)
    buffer2(4)=1.5
    call MPI_WAIT(irequest,istatus,ierr)
    buffer1(icount)=0.0
  endif
```

The 10 Essential MPI Statements

(1) `include 'mpif.h'`

(2) `integer istat(MPI_STATUS_SIZE)`

(3) `call MPI_INIT(...)`

(4) `call MPI_COMM_RANK(...)`

(5) `call MPI_COMM_SIZE(...)`

(6) `call MPI_[I]SEND(...)`

(7) `call MPI_[I]RECV(...)`

(8) `call MPI_WAIT(...)`

(9) `call MPI_BARRIER(...)`

(10) `call MPI_FINALIZE(...)`

Code example: parallel search

Problem statement

- find indices of matching entries in a large table

Program steps

1. read data from file
2. define target entry
3. scan array for matching entries
   & immediate output of indices found

   (sequential code) (data file)

Parallel algorithm

Individual steps

1. identify concurrency
2. decomposition: (domain/functional)
3. design of communication patterns
4. algorithm implementation
5. validation against sequential results
6. timing/optimization of parallel execution

Decomposition

Considerations

- incremental results required a.s.a.p.
  $\Rightarrow$ one dedicated "master" task for I/O
- "worker" tasks search on sub-arrays
  $\Rightarrow$ functional AND domain decomposition

Communication

Considerations

- workers need to send each result immediately
- workers need to signal end of search
- master needs to listen for incoming messages
- source and content of messages unknown to master

$\rightsquigarrow$ need to define a protocol

Possible solution

Master

```
i=0
end_tag=99
do while(i.ne.nproc-1)
   CALL MPI_RECV(...,
        MPI_ANY_SOURCE,
        MPI_ANY_TAG,...)
   if(status(MPI_TAG).eq.
        end_tag)then
      i=i+1
   else
      output_result
   end if
end do
```

Workers

```
end_tag=99
do i=1,nloc
   if(a(i) == target)then
     call MPI_SEND(...,
          tag=0,...)
   end if
end do
call MPI_SEND(...,
     tag=end_tag,...)
```

More considerations

Need to decompose and distribute input data

Need conversion of local to global indices

Timing of sections of the code

`t1=MPI_WTIME()`

`double precision t1`, in seconds since arbitrary origin

- correct output:

```
P 1 62
P 3 271
P 3 291
P 3 296
P 2 183
```

(parallel code)