MPI Workshop - I

Introduction to Point-to-Point

and

Collective Communications

AHPCC Research Staff

Week 1 of 3

Table of Contents

- **Introduction**
- **MPI Course Map**
- **Background**
- **MPI Routines/Exercises**
  - point-to-point communications
  - blocking versus non-blocking calls
  - collective communications
- **How to run MPI routines at AHPCC (Blackbear)**
- **References**

Parallelism is done by:

- Breaking up the task into smaller tasks
- Assigning the smaller tasks to multiple workers to work on simultaneously
- Coordinating the workers
- Not breaking up the task so small that it takes longer to tell the worker what to do than it does to do it
  - ***Buzzwords: latency, bandwidth***

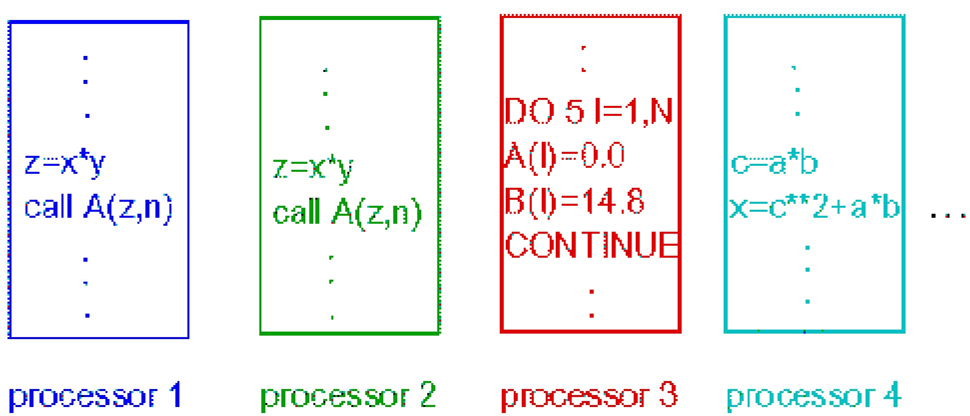
- **All parallel computers use multiple processors**
  - There are several different methods used to classify computers
  - No single scheme fits all designs
  - Flynn's scheme uses the relationship of program instructions to program data.
    - **SISD - Single Instruction, Single Data Stream**
    - **SIMD - Single Instruction, Multiple Data Stream**
    - **MISD - Multiple Instruction, Single Data Stream (no practical examples)**
    - **MIMD - Multiple Instruction, Multiple Data Stream**
      - **SPMD - Single program, Multiple Data Stream**
        - special case, typical MPI model

Underlying model - MIMD/SPMD

- Parallelism achieved by connecting multiple processors together
- Includes all forms of multiprocessor configurations
- Each processor executes its own instruction stream independent of other processors on unique data stream
- Advantages
  - ✔ **Processors can execute multiple job streams simultaneously**
  - ✔ **Each processor can perform any operation regardless of what other processors are doing**

- Disadvantages
  - ✔ **Load balancing overhead - synchronization needed to coordinate processors at end of parallel structure in a single application**

MIMD Model



MPI Memory Model - Distributed Memory

- Multiple processors operate independently but each has its own private memory
- Data is shared across a communications network using message passing
- User responsible for synchronization using message passing
- Advantages
  - ✔ **Memory scalable to number of processors. Increase number of processors, size of memory and bandwidth increases.**
  - ✔ **Each processor can rapidly access its own memory without interference**

- Disadvantages
  - ✔ **Difficult to map existing data structures to this memory organization**
  - ✔ **User responsible for sending and receiving data among processors**
  - ✔ **To minimize overhead and latency, data should be blocked up in large chunks and shipped before receiving node needs it**

Message Passing

- The message passing model is defined as:
  - **set of processes using only local memory**
  - **processes communicate by sending and receiving messages**
  - **data transfer requires cooperative operations to be performed by each process (a send operation must have a matching receive)**
- Programming with message passing is done by linking with and making calls to libraries which manage the data exchange between processors. Message passing libraries are available for most modern programming languages.

- Flexible, it supports multiple programming schemes including:
  - ✔ Functional parallelism - different tasks done at the same time.
  - ✔ Master-Slave parallelism - one process assigns subtask to other processes.
  - ✔ SPMD parallelism - Single Program, Multiple Data - same code replicated to each process

Message Passing Implementations

- MPI - Message Passing Interface
- PVM - Parallel Virtual Machine
- MPL - Message Passing Library

Message Passing Interface - MPI

- A standard portable message-passing library definition developed in 1993 by a group of parallel computer vendors, software writers, and application scientists.
- Available to both Fortran and C programs.
- Available on a wide variety of parallel machines.
- Target platform is a distributed memory system such as the SP.
- All inter-task communication is by message passing.
- All parallelism is explicit: the programmer is responsible for parallelism the program and implementing the MPI constructs.
- Programming model is SPMD

MPI Standardization Effort

- **MPI Forum initiated in April 1992: Workshop on Message Passing Standards.**
  - Initially about 60 people from 40 organizations participated.
    - **Defines an interface that can be implemented on many vendor's platforms with no significant changes in the underlying communication and systemsoftware.**
    - **Allow for implementations that can be used in a heterogeneous environment.**
    - **Semantics of the interface should be language independent.**
  - Currently, there are over 110 people from 50 organizations who have contributed to this effort.

MPI-Standard Release

- **May, 1994 MPI-Standard version 1.0**
- **June, 1995 MPI-Standard version 1.1***
  - includes minor revisions of 1.0
- **July, 1997 MPI-Standard version 1.2 and 2.0**
  - with extended functions
    - ✔ **2.0 - support real time operations, spawning of processes, more collective operations**
    - ✔ **2.0 - explicit C++ and F90 bindings**
- **Complete postscript and HTML documentation can be found at: http://www.mpi-forum.org/docs/docs.html**

*** Currently available at AHPCC**

MPI Implementations

- **Vendor Implementations**
  - IBM-MPI *
  - SGI *
- **Public Domain Implementations**
  - MPICH (ANL and MSU)*
  - Other implementations have largely died off.

* Available at AHPCC.

Language Binding (version 1.1)

- **Fortran 77**

  include 'mpif.h'

  ```
  call MPI_ABCDEF(list of arguments, IERROR)
  ```

- **Fortran 90 via Fortran 77 Library**
  - F90 strong type checking of arguments can cause difficulties
  - cannot handle more than one object type
  - include 'mpif90.h'
- **ANSI C**

  #include 'mpi.h'

  ```
  IERROR=MPI_Abcdef(list of arguments)
  ```

- **C++ via C Library**
  - via extern "C" declaration, #include 'mpi++.h'

Examples to Be Covered

	Week 1 Point to Point Basic Collective	Week 2 Collective Communications	Week 3 Advanced Topics
MPI functional routines	MPI_SEND (MPI_ISEND) MPI_RECV (MPI_IRECV) MPI_BCAST MPI_SCATTER MPI_GATHER	MPI_BCAST MPI_SCATTERV MPI_GATHERV MPI_REDUCE MPI_BARRIER	MPI_DATATYPE MPI_HVECTOR MPI_VECTOR MPI_STRUCT MPI_CART_CREATE
MPI Examples	Helloworld Swapmessage Vector Sum	Pi Matrix/vector multiplication Matrix/matrix mulplication	Poisson Equation Passing Structures/ common blocks Parallel topologies in MPI

Program examples/MPI calls

- **Hello - Basic MPI code with no communications.**
  - MPI_INIT - starts MPI communications
  - MPI_COMM_RANK - get processor id
  - MPI_COMM_SIZE - get number of processors
  - MPI_FINALIZE - end MPI communications
- **Swap - Basic MPI point-to-point messages**
  - MPI_SEND - blocking send
  - MPI_RECV - blocking receive
  - MPI_IRECV, MPI_WAIT - non-blocking receive
- **Vecsum - Basic collective communications calls**
  - MPI_SCATTER - distribute an array evenly among processors
  - MPI_GATHER - collect pieces of an array from processors

Get MPI sample codes

- **Download MPI example codes from**
  - ~acpineda/public_html/mpi-sept2k/workshop_1
  - http://www.ahpcc.unm.edu/~acpineda/mpi-sept2k/workshop_1
- **Example codes**
  - hello.f, hello.c
  - swap.f, swap.c
  - vecsum.f, vecsum.c

A Basic MPI Program

```
program helloworld

include 'mpif.h'
integer comm, rank, numproc, ierror


call MPI_INIT(ierror)

call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierror)


call MPI_COMM_SIZE(MPI_COMM_WORLD,numproc,ierror)
```

Include declarations of MPI functions and constants.

Begin parallel execution of code.

Find out which process we are from the set of processes defined by the communicator MPI_COMM_WORLD which is MPI's shorthand for all the processors running your program. This value is stored in rank.

Returns the number of processes in numproc.

A Basic MPI Program - cont'd

```
print *,"Hello World from Processor ",rank," of ",numproc
```

This line is printed by all processes.

```
if(rank.eq.0) then
  print *,"Hello again from processor ", rank
endif
```

This line is printed only by the process of rank equal to 0.

```
call MPI_FINALIZE(ierror)
```

End parallel execution.

```
end program helloworld
```

Compiling your code

- **You invoke your compiler via scripts that tack on the appropriate MPI include and library files:**
  - mpif77 -o <progname> <filename>.f
    - **mpif77 -c <filename>.f**
    - **mpif77 -o progname <filename>.o**
  - mpif90 -o <progname> <filename>.f90
  - mpicc -o <progname> <filename>.c
  - mpiCC -o <progname> <filename>.cc
- **The underlying compiler, NAG, PGI, etc. is determined by how MPIHOME and PATH are set up in your environment.**

How to compile and run MPI on Blackbear

- **MPICH**
  - **Two choices of communications networks:**
    - ✔eth - FastEthernet (~100Mb/sec)
    - ✔gm - Myrinet (~1.2 Gb/sec)
  - **Many compilers**
    - ✔NAG F95 - f95
    - ✔PGI - pgf77, pgcc, pgCC, pgf90
    - ✔GCC, G77
  - **Combination is determined by your environment.**

How to compile and run MPI on Blackbear

- **MPICH** - Two ways to setup your environment
  - ✔ **http://www.ahpcc.unm.edu/Systems/Documentation/BB-UG.html**
    - setup_env.bb - appends to your .cshrc
    - .rhosts file -lists nodes on bb
    - cshrc_bb
    - bashrc_bb
  - ✔ **Copy ACP's experimental version**
    - **copy from ~acpineda**
    - .prefs.ARCH (ARCH=BB, AZUL, RCDE, etc.)
      - **set compiler/network options for your platform here**
    - .cshrc
    - .cshrc.ARCH

PBS (Portable Batch Scheduler)

- **To submit job use**
  - **qsub file.pbs**
    - **file.pbs is a shell script that invokes mpirun**
  - **qsub -I**
    - **Interactive session**
- **To check status**
  - **qstat, qstat -an** (see man qstat for details)
- **To cancel job**
  - **qdel job_id**

PBS command file (file.pbs)

- **Just a shell script**

#PBS -l nodes=4,walltime=12:00:00

#!/bin/csh

…

source $HOME/BLACKBEAR/cshrc_bb

gmpiconf2 $PBS_NODEFILE

mpirun -np 8 -arch ch_gm -machinefile $PBS_NODEFILE <executable or script>

…

-l is also an option to qsub

Myrinet

gmpiconf - 1 process per node

gmpiconf2 - 2 processes per node

Message Exchange

```
     if(numproc > 1) then
     if(rank == root) then

            message_sent='Hello from processor 0'
```

MPI_SEND is the standard blocking send operation. Depending upon whether the implementers of the particular MPI library you are using buffer the message in a global storage area, this call may or may not block until a matching receive has been posted. Other flavors of send operations exist in MPI that allow you to force buffering, etc.

Messages are tracked by source id/rank, destination id/rank, message tag, and communicator.

Destination

Message Tag

```
            call MPI_SEND(message_sent, 80, MPI_CHARACTER, 1, 1, &
MPI_COMM_WORLD, ierror)
```

Buffer containing the data

The number of elements in the data buffer

The type of the data being sent. In this case character.

Message Exchange - cont’d

The root process then stops at MPI_RECV until processor 1 sends its message back.

```
       call MPI_RECV( message_received, 80, MPI_CHARACTER, 1, 1, &
     MPI_COMM_WORLD, status, ierror)

     else if (rank.eq.1) then

 ! Processor 1 waits until processor 0 sends its message

       call MPI_RECV(message_received, 80, MPI_CHARACTER, 0, 1, &
               MPI_COMM_WORLD, status, ierror)

  ! It then constructs a reply.
        message_sent='Proc 1 got this message: '//message_received
  ! And sends it....
        call MPI_SEND( message_sent, 80, MPI_CHARACTER, 0, 1, &
               MPI_COMM_WORLD,ierror)
     endif
     print *,"Processor ",rank," sent '",message_sent,"'"
    print *,"Processor ",rank," received '",message_received,"'"
   else
     print *,"Not enough processors to demo message passing"
  endif
```

Sender Id (points to the source argument, 1)

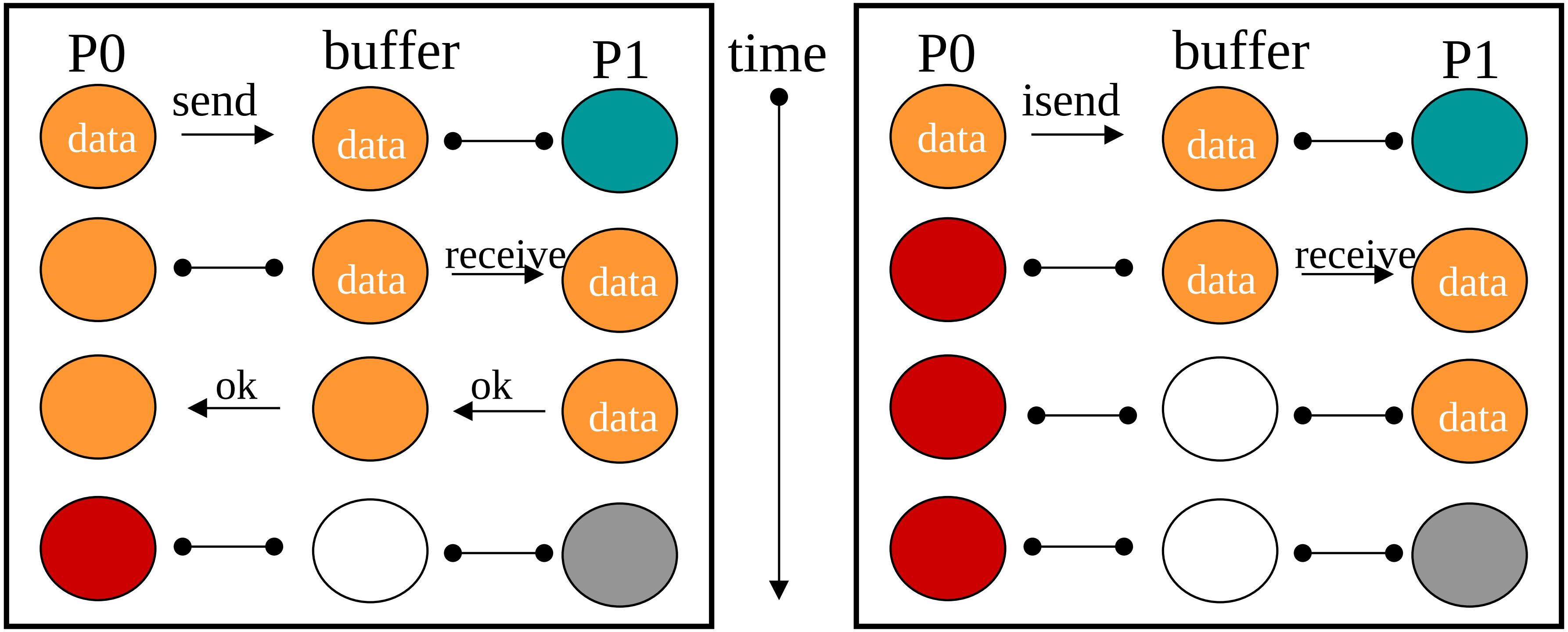
Message Tag (points to the tag argument, 1)

Matching Sends to Receives

- **Message Envelope - consists of the source, destination, tag, and communicator values.**
- **A message can only be received if the specified envelope agrees with the message envelope.**
- **The source and tag portions can be wildcarded using MPI_ANY_SOURCE and MPI_ANY_TAG. (Useful for writing client-server applications.)**
- **Source=destination is allowed except for blocking operations.**
- **Variable types of the messages must match.**
- **In heterogeneous systems, MPI handles data conversions, e.g. big-endian to little-endian.**
- **Messages (with the same envelope) are not overtaking.**

Blocking vs. non-blocking calls/Buffering

- Non-blocking calls can be used to avoid “DEADLOCK”.
- Non-blocking calls can be used to overlap computation and communications.



Non-blocking call

```
if(rank.eq.root) then

   message_sent='Hello from processor 0'
```

Begin the receive operation by letting the world know we are expecting a message from process 1. We then return immediately.

```
   call MPI_IRECV( message_received, 80, MPI_CHARACTER, 1, 1, &
         MPI_COMM_WORLD, request, ierror)
```

Now send the message as before.

```
   call MPI_SEND(message_sent, 80, MPI_CHARACTER, 1, 1, &
        MPI_COMM_WORLD, ierror)
```

Now wait for the receive operation to complete.

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

else if (rank.eq.1) then
```

Non-blocking call

- Can use MPI_TEST in place of MPI_WAIT to periodically check on a message rather than blocking and waiting.
- Client-server applications can use MPI_WAITANY or MPI_TESTANY.
- Can peek ahead at messages with MPI_PROBE and MPI_IPROBE.

Collective Communications

Broadcast the coefficients to all processors.

$$\tilde{z} = a\tilde{x} + b\tilde{y}$$

Scatter the vectors among N processors as
zpart, xpart, and ypart.

Calls can return as soon as their participation is complete.

Vector Sum

MPI_SCATTER distributes blocks of array x from the root process to the array xpart belonging to each process in MPI_COMM_WORLD. Likewise, blocks of the array y are distributed to the array ypart.

Array x and the number of elements of type real to send to each process. Only meaningful to root.

Array xpart and the number of elements of type real to receive.

```
call  MPI_SCATTER( x, dim2, MPI_REAL, xpart, dim2, MPI_REAL, root, &
                   MPI_COMM_WORLD, ierr )
```

Array y and the number of elements of type real to send to each process. Only meaningful to root.

Array ypart and the number of elements of type real to receive.

```
call  MPI_SCATTER( y, dim2, MPI_REAL, ypart, dim2, MPI_REAL, root, &
                   MPI_COMM_WORLD, ierr )
```

Vector Sum - cont'd

The coefficients, *a* and *b*, are stored in an array of length 2, coeff, that is broadcast to all processes via MPI_BCAST from the process root.

```
call  MPI_BCAST( coeff, 2, MPI_REAL, root, MPI_COMM_WORLD, ierr )
```

Now each processor computes the vector sum on its portion of the vector. The blocks of the vector sum are stored in zpart.

```
do i = 1, dim2
   zpart(i) = coeff(1)*xpart(i) + coeff(2)*ypart(i)
 enddo
```

Now we use MPI_GATHER to collect the blocks back to the root process.

The array zpart to be gathered and the number of elements each process sends to root.

For the root process, the array z contains the collected blocks from all processes on output. MPI_GATHER needs to know how much data to collect from each process.

```
call  MPI_GATHER( zpart, dim2, MPI_REAL, z, dim2, MPI_REAL, root, &
             MPI_COMM_WORLD, ierr )
```

References - MPI Tutorial

- **PACS online course**
  - http://webct.ncsa.uiuc.edu:8900/
- **CS471 - Andy Pineda**
  - http://www.arc.unm.edu/~acpineda/CS471/HTML/CS471.html
- **MHPCC**
  - http://www.mhpcc.edu/training/workshop/html/workshop.html
- **Edinburgh Parallel Computing Center**
  - http://www.epcc.ed.ac.uk/epic/mpi/notes/mpi-course-epic.book_1.html
- **Cornell Theory Center**
  - http://www.tc.cornell.edu/Edu/Talks/topic.html#mess
- **Argonne**
  - http://www-unix.mcs.anl.gov/mpi/