ALBUQUERQUE
High Performance Computing Center

The University of New Mexico

MPI Workshop - I

Introduction to Point-to-Point and Collective Communications

AHPCC Research Staff

Week 1 of 2

Table of Contents

Table of Contents

- **Processor Communication**
  - Shared Memory
  - Distributed Memory
- **Parallel Programming Paradigms**
  - Various Methods
  - Message Passing
  - Data Parallel

Implementations

- Message Passing
  - ✔**Message Passing Interface**
  - ✔**Message Passing Library**
- Data Parallel
  - ✔**F90 / High Performance Fortran**

What is Parallelism?

- A strategy for performing large, complex tasks faster.
- A large task can either be performed serially, one step following another, or can be decomposed into smaller tasks to be performed simultaneously, i.e., in parallel.

- **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***

Sequential Programming

- Traditionally, programs have been written for serial computers:
  - ✔ **One instruction executed at a time**
  - ✔ **Using one processor**
  - ✔ **Processing speed dependent on how fast data can move through hardware**
  - ✔ **Speed of Light = 30 cm/nanosecond**
  - ✔ **Limits of Copper Wire = 9 cm/nanosecond**
  - ✔ **Fastest machines execute approximately 1 instruction in 9-12 billionths of a second**
  - ✔ **This is not fast enough for many problems**

Traditional Supercomputer Technology

- Fastest possible single processors.
- Peak performance was achieved with good memory bandwidth.
- Benefits
  - ✔**Supports sequential programming (well understood)**
  - ✔**30+ years of compiler and tool development**
  - ✔**I/O is relatively simple**
- Limitations
  - ✔**Extremely expensive & has significant cooling requirements**
  - ✔**Single processor performance is reaching its asymptotic limit**

- **Parallel Supercomputer Technology**
  - Applies many smaller cost efficient processors to work on a part of the same task
  - Capitalizes on work done in the microprocessor and networking markets
- **Benefits**
  - Ability to achieve performance and work on problems impossible with traditional computers.
  - Exploit "off the shelf" processors, memory, disk and tape systems.

- Ability to scale to problem.
- Ability to quickly integrate new elements into systems thus capitalizing on improvements made by other markets.
- Commonly much cheaper.

Limitations

- New technology. Programmers need to learn parallel programming approaches.
- Standard sequential codes will not "just run".
- Compilers and tools are often not mature.
- I/O is not as well understood yet.

Parallel Programming

- Parallel programming involves:
  - ✔**Decomposing an algorithm or data into parts**
  - ✔**Distributing the parts as tasks which are worked on by multiple processors simultaneously**
  - ✔**Coordinating work and communications of those processors**
- Parallel programming considerations:
  - ✔**Type of parallel architecture being used**
  - ✔**Type of processor communications used**

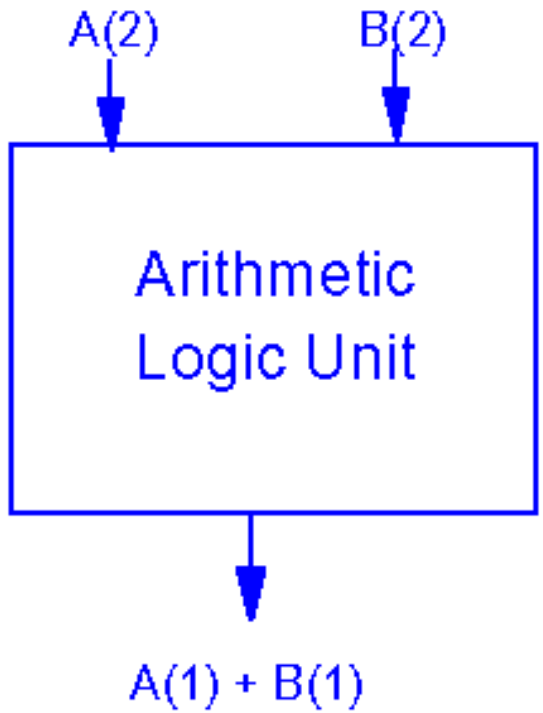
All parallel computers use multiple processors

- 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**

SISD Model

- Conventional serial, scalar von Neumann computer
- One instruction stream
- A single instruction is issued each clock cycle
- Each instruction operates on a single (scalar) data element
- Limited by the number of instructions that can be issued in a given unit of time
- Performance frequently measured in MIPS (million of instructions per second)

SISD Model



SIMD Model

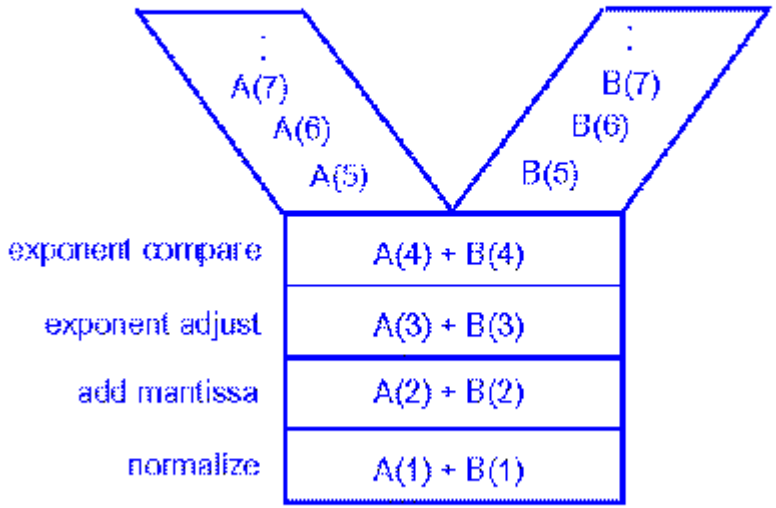
- Also von Neumann architectures but more powerful instructions
- Each instruction may operate on more than one data element
- Usually intermediate host executes program logic and broadcasts instructions to other processors
- Synchronous (lockstep)
- Rating how fast these machines can issue instructions is not a good measure of their performance

- Performance is measured in MFLOPS (millions of floating point operations per second)
- Two major types:
  - ✔**Vector SIMD**
  - ✔**Parallel SIMD**

Vector SIMD

- Single instruction results in multiple operands being updated
  - ✔ **Scalar processing operates on single data elements. Vector processing operates on whole vectors (groups) of data at a time.**
  - ✔ **Examples:**
    - ✸ **Cray 1, NEC SX-2, Fujitsu VP, Hitachi S820**
    - ✸ **Single processor of:**
      - Cray C 90, Cray2, NEC SX-3, Fujitsu VP 2000, Convex C-2

Vector SIMD Model



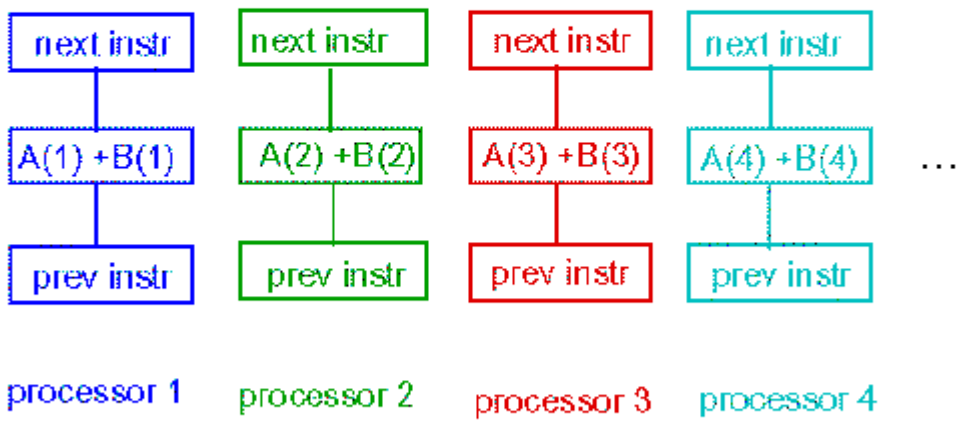
Parallel SIMD

- Processor arrays - single instruction is issued and all processors execute the same instruction, operating on different sets of data.
- Processors run in a synchronous, lockstep fashion
- Advantages
  - ✔**DO loops conducive to SIMD parallelism**

```
do 100 i= 1, 100
    c(i) = a(i) + b(i)
100  continue
```

  - ✔**Synchronization trivial - all processors operate in lock-step**

Parallel SIMD Model



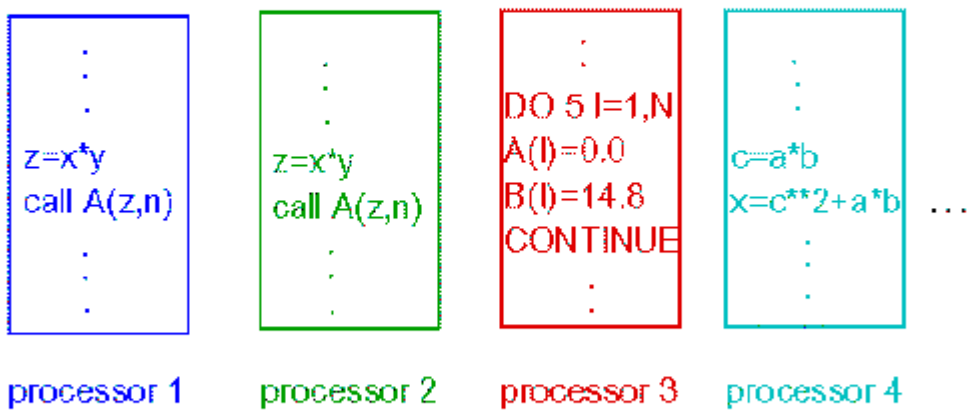
- Disadvantages
  - ✔**Decisions within DO loops can result in poor execution by requiring all processes to perform the operation controlled by decision whether results are used or not**
  - ✔**Examples:**
    - ✷**Connection Machine CM-2; Maspar MP-1, MP-2**

MIMD Model

- 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


z=x*y
call A(z,n)
z=x*y
call A(z,n)
DO 5 I=1,N
A(I)=0.0
B(I)=14.8
CONTINUE
c=a*b
x=c**2+a*b
...
processor 1
processor 2
processor 3
processor 4

- Examples
  - ✔**MIMD Accomplished via Parallel SISD machines:**
    - ✷**Sequent, nCUBE, Intel iPSC/2, IBM RS6000 cluster**
  - ✔**MIMD Accomplished via Parallel SIMD machines:**
    - ✷**Cray C 90, Cray 2, NEC SX-3, Fujitsu VP 2000, Convex C-2,**
    - ✷**Intel Paragon, CM 5, KSR-1, IBM SP1, IBM SP2**

Processor Communications

- Task coordination of multiple nodes requires some form of inter-processor communications:
  - ✔**Convey information and data between processors**
  - ✔**Synchronize node activities**
- Communication is dependent upon memory architecture, which, in turn, will affect how you write your parallel program
- The two primary memory architectures are:
  - ✔**Shared Memory and Distributed Memory**

Shared Memory

- Multiple processors operate independently but share the same memory resources
- Only one processor can access the shared memory location at a time
- Synchronization achieved by controlling tasks' reading from and writing to the shared memory
- Advantages
  - ✔Easy for user to use efficiently
  - ✔Data sharing among tasks is fast (speed of memory access)

- Disadvantages
  - ✔**Memory is bandwidth limited. Increase of processors without increase of bandwidth can cause severe bottlenecks**
  - ✔**User is responsible for specifying synchronization, e.g., locks**
- Examples:
  - ✔**Cray Y-MP, Convex C-2, Cray C-90**

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**
- Examples:
  - ✔**nCUBE Hypercube, Intel Hypercube, TMC CM-5,**
  - ✔**IBM SP1, SP2, Intel Paragon**

- **There are many methods of programming parallel computers. Two of the most common are message passing and data parallel.**
  - Message Passing - the user makes calls to libraries to explicitly share information between processors.
  - Data Parallel - data partitioning determines parallelism
- **An effective implementation is one which closely matches its target hardware and provides the user ease in programming.**

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

Data Parallel

- The data parallel model is defined as:
  - ✔ **Each process works on a different part of the same data structure**
  - ✔ **Global name space**
  - ✔ **Commonly a Single Program Multiple Data (SPMD) approach**
  - ✔ **Data is distributed across processors**
  - ✔ **All message passing is done invisibly to the programmer**
  - ✔ **Commonly built "on top of" one of the common message passing libraries**

- Programming with data parallel model is accomplished by writing a program with data parallel constructs and compiling it with a data parallel compiler.
- The compiler converts the program into standard code and calls to a message passing library to distribute the data to all the processes.

Parallel Paradigm Implementations

- Message Passing
  - ✔**MPI - Message Passing Interface**
  - ✔**PVM - Parallel Virtual Machine**
  - ✔**MPL - Message Passing Library**
- Data Parallel
  - ✔**Fortran 90 / High Performance Fortran**

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
- Covered in three workshops of this series.

- Message Passing Library
  - ✔**Message Passing Library often called MPL.**
  - ✔**Part of IBM's SP Parallel Environment.**
  - ✔**IBM's proprietary message passing routines.**
  - ✔**Designed to provide a simple and efficient set of well understood operations for coordination and communication among processors in a parallel application.**
  - ✔**Target platform is a distributed memory system such as the SP. Has also been ported to RS/6000 clusters.**

- F90 / High Perfomance Fortran
  - ✔**Fortran 90 (F90) - (ISO / ANSI standard extensions to Fortran 77).**
  - ✔**High Performance Fortran (HPF) - extensions to F90 to support data parallel programming.**
  - ✔**Compiler directives allow programmer specification of data distribution and alignment.**
  - ✔**New compiler constructs and intrinsics allow the programmer to do computations and manipulations on data with different distributions.**
  - ✔**To be covered in the next workshop of this series.**

Course Map

	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

What is MPI ?

- **Message Passing Interface is a standardized communication protocol used on massively parallel machines with distributed-memory architectures.**
  - Also available on some shared memory architectures.
- **Implemented as a set of library calls that can be called from different programming languages. (But mixed language programs are not portable.)**
- **It is currently the most popular message passing parallel computing tool relative to others that also exist today (e.g. PVM, MPL).**
- **Very flexible - allows the use of many parallel programming paradigms**

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)*
  - LAM (Ohio State) for all except SP1.
  - CHIMP (EPCC)

* 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’

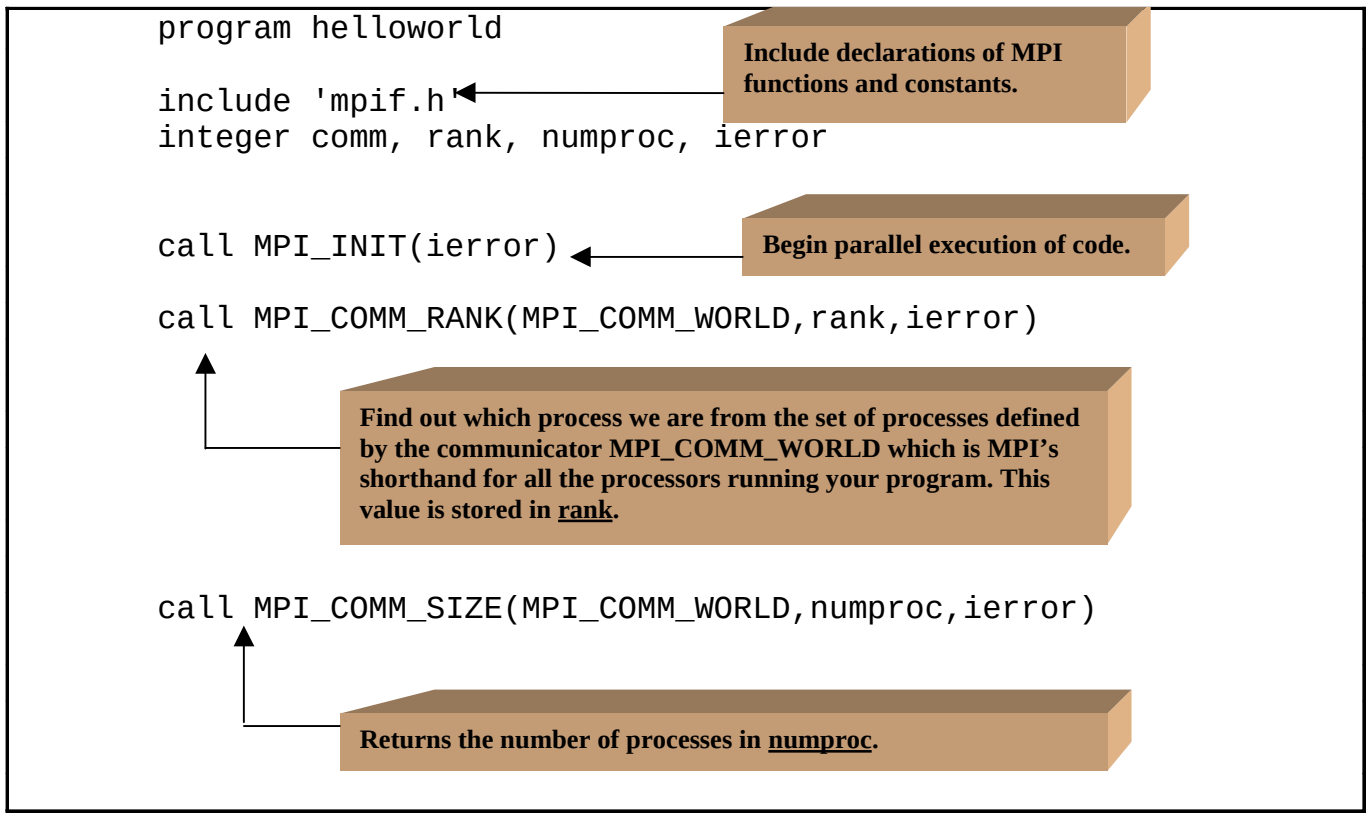
  ```
  IERROR=MPI_Abcdef(list of arguments)
  ```

- **C++ via C Library**
  - via extern “C” declaration, #include ‘mpi++.h’

Program examples/MPI calls

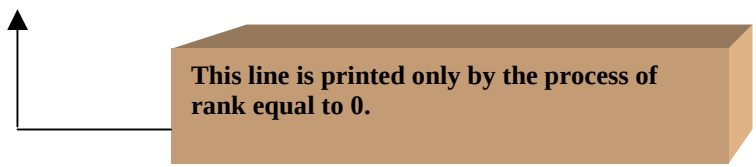
- **Hello - Basic MPI code with no communications.**
  - MPI_INIT - starts MPI communications
  - MPI_COMM_RANK - get processor id
  - MPI_COMM_SIZE - get number of processors
  - MPI_FINALIZE - end MPI communications
- **Swap - Basic MPI point-to-point messages**
  - MPI_SEND - blocking send
  - MPI_RECV - 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

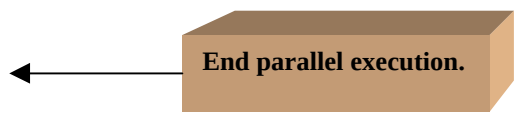
A Basic MPI Program



A Basic MPI Program - cont'd

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

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

call MPI_FINALIZE(ierror)

end program helloworld
```





How to Compile MPI at the AHPCC

- **MPICH**
  - MPICH script
  - Compiling and linking objects directly
- **IBM**
  - Interactively on the SP2s (use for code development)
  - Loadleveler

MPICH Script

- **To compile and link your MPI code in one step, use the appropriate command from the following:**
  - mpif77 -i mpich -c ch_p4 -- -o <progname> <filename>.f
  - mpif90 -i mpich -c ch_p4 -- -o <progname> <filename>.f90
  - mpicc -i mpich -c ch_p4 -- -o <progname> <filename>.f
  - mpiCC -i mpich -c ch_p4 -- -o <progname> <filename>.f

Process Group File

- **Before running the program you must construct a process group file in the directory where your executable is. This file is called <progname>.pg and should look something like this:**

```
local 0
turing01 1 /complete/path/to/your/executable
turing02 1 /complete/path/to/your/executable
turing03 1 /complete/path/to/your/executable
```

Direct Compiling and Linking

- **To compile**
  - **f95 -free -c -I/usr/local/mpich/include <file.f>**
- **To link objects**
  - **f95 -o <file> <file.o> -L/usr/local/mpich/lib/LINUX/ch_p4 -lmpi**

MPI on IBM

- **Compiling on IBM**
  - mpxlf90 -qnolm -o <file> <file.f>

- **Set necessary environment variables**

setenv MP_PROCS n (when using n processors. Max - 4.)
setenv MP_EUIDEVICE en0 (to use ethernet for communication)
setenv MP_EUILIB ip (to use IP protocol for communication)
setenv MP_RMPOOL 0 (to specify Resource Manager pool)

Load Leveler

- **Compile like before using mpxlf90 script**
- **To submit job use**
  - llsubmit <file.cmd>
- **To check status**
  - llq or showq
- **To cancel job**
  - llcancel job_id

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

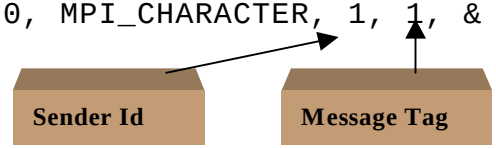
The number of elements in the data buffer

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
```

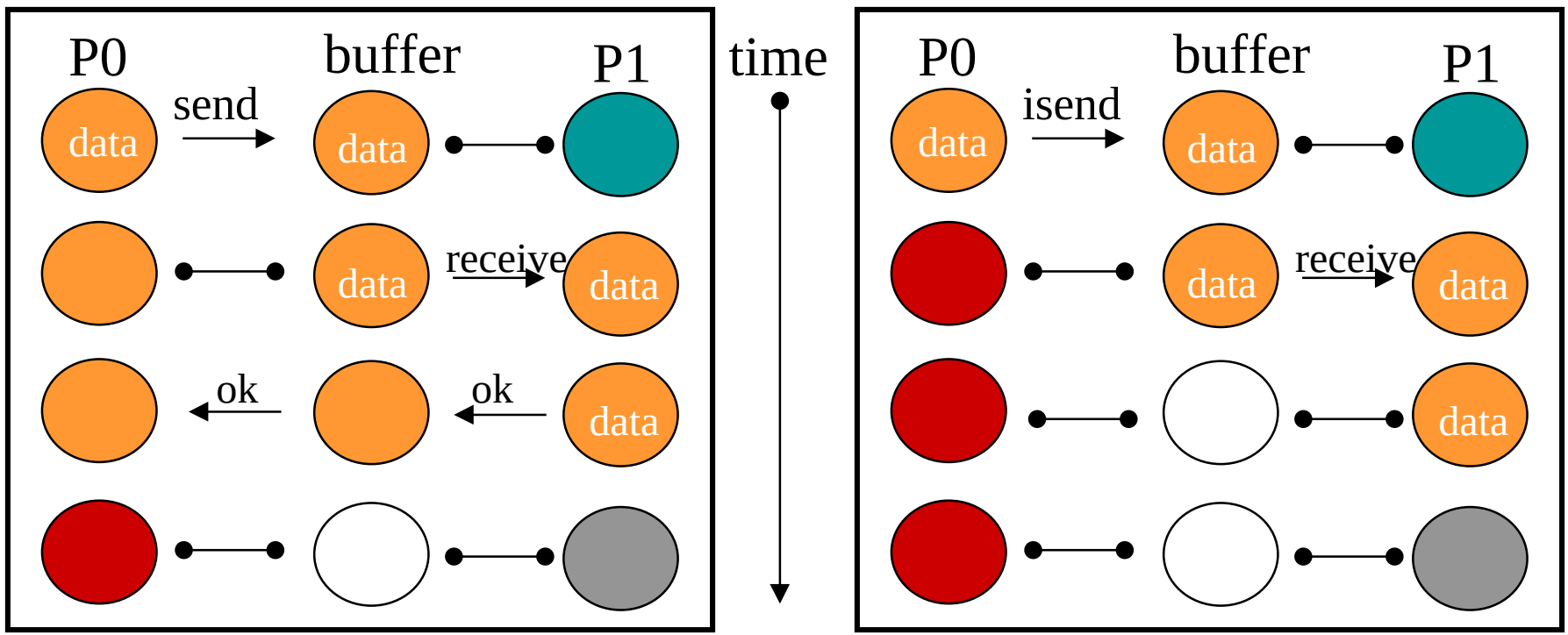


Matching Sends to Receives

- **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.

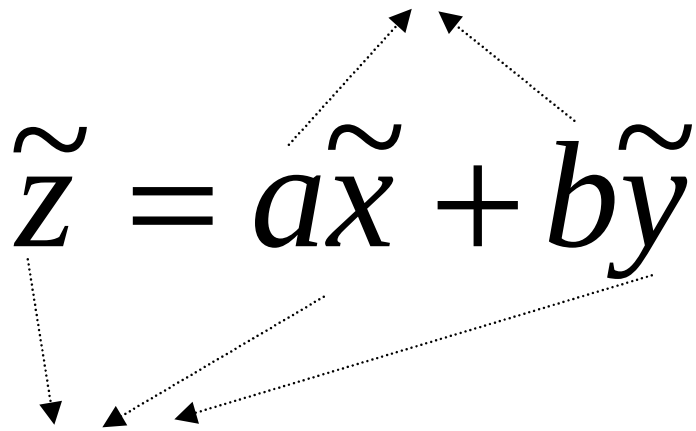
```
   call MPI_WAIT(request, status, ierror)

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.



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

- **Handout**
- **CS471 - Andy Pineda**
  - http://www.arc.unm.edu/~acpineda/CS471/HTML/CS471.html
- **MHPCC**
  - http://www.mhpcc.edu/training/workshop/html/mpi/MPIIntro.html
  - 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

References - IBM Parallel Environment

- **POE - Parallel Operating Environment**

  http://www.mhpcc.edu/training/workshop/html/poe/poe.html

  http://ibm.tc.cornell.edu/ibm/pps/doc/primer/

- **Loadleveler**

  http://www.mhpcc.edu/training/workshop/html/loadleveler/LoadLeveler.html

  http://ibm.tc.cornell.edu/ibm/pps/doc/LlPrimer.html

  http://www.qpsf.edu.au/software/ll-hints.html