Parallel Programming with MPI

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

Modificado por Ángel de Vicente, para el curso de Técnicas Avanzadas de Programación

Table of Contents

Setting the Stage

- Overview of parallel computing
- Parallel architectures
- Parallel programming models
- Hardware
- Software

Overview of Parallel Computing

- Parallel computing is when a program uses concurrency to either
  - decrease the runtime needed to solve a problem
  - increase the size of problem that can be solved
- The direction in which high-performance computing is headed!
- Mainly this is a price/performance issue
  - Vector machines (e.g., Cray X1) very expensive to engineer and run
  - Commodity hardware/software - Clusters!

Writing a Parallel Application

- Decompose the problem into tasks
  - Ideally, these tasks can be worked on independently of the others
- Map tasks onto "threads of execution" (processors)
- Threads have shared and local data
  - Shared: used by more than one thread
  - Local: Private to each thread
- Write source code using some parallel programming environment
- Choices may depend on (among many things)
  - the hardware platform to be run on
  - the level performance needed
  - the nature of the problem

Parallel Architectures

- Distributed memory (Pentium 4 and Itanium 2 clusters)
  - Each processor has local memory
  - Cannot directly access the memory of other processors
- Shared memory (Cray X1, SGI Altix, Sun COE)
  - Processors can directly reference memory attached to other processors
  - Shared memory may be *physically* distributed
    - The cost to access remote memory may be high!
  - Several processors may sit on one memory bus (SMP)
- Combinations are very common, e.g. Itanium 2 Cluster:
  - 258 compute nodes, each with 2 CPUs sharing 4GB of memory
  - High-speed Myrinet interconnect between nodes.

Parallel Programming Models

- Distributed memory systems
  - For processors to share data, the programmer must explicitly arrange for communication - "**Message Passing**"
  - Message passing libraries:
    - MPI ("Message Passing Interface")
    - PVM ("Parallel Virtual Machine")
    - Shmem (Cray only)
- Shared memory systems
  - "Thread" based programming
  - Compiler directives (OpenMP; various proprietary systems)
  - Can also do explicit message passing, of course

Parallel Computing: Hardware

- In very good shape!
- Processors are cheap and powerful
  - Intel, AMD, IBM PowerPC, …
  - Theoretical performance approaching 10 GFLOP/sec or more.
- SMP nodes with 8-32 CPUs are common
- Clusters with tens or hundreds of nodes are common
- Affordable, high-performance interconnect technology is available - clusters!
- Systems with a few hundreds of processors and good inter-processor communication are not hard to build

Parallel Computing: Software

- Not as mature as the hardware
- The main obstacle to making use of all this power
  - Perceived difficulties with writing parallel codes outweigh the benefits
- Emergence of standards is helping enormously
  - MPI
  - OpenMP
- Programming in a shared memory environment generally easier
- Often better performance using message passing
  - Much like assembly language vs. C/Fortran

Brief History of MPI

- What is MPI
- MPI Forum
- Goals and Scope of MPI
- MPI on OSC Parallel Platforms

What Is MPI

- **M**essage **P**assing **I**nterface

- What is the message?

  DATA

- Allows data to be passed between processes in a distributed memory environment

MPI Forum

- First message-passing interface standard
  - Successor to PVM
- Sixty people from forty different organizations
- International representation
- MPI 1.1 Standard developed from 92-94
- MPI 2.0 Standard developed from 95-97
- Standards documents
  - http://www.mcs.anl.gov/mpi/index.html
  - http://www.mpi-forum.org/docs/docs.html (postscript versions)

Goals and Scope of MPI

- MPI's prime goals are:
  - To provide source-code portability
  - To allow efficient implementation
- It also offers:
  - A great deal of functionality
  - Support for heterogeneous parallel architectures
- Acknowledgements
  - Edinburgh Parallel Computing Centre/University of Edinburgh for material on which this course is based
  - Dr. David Ennis of the Ohio Supercomputer Center who initially developed this course

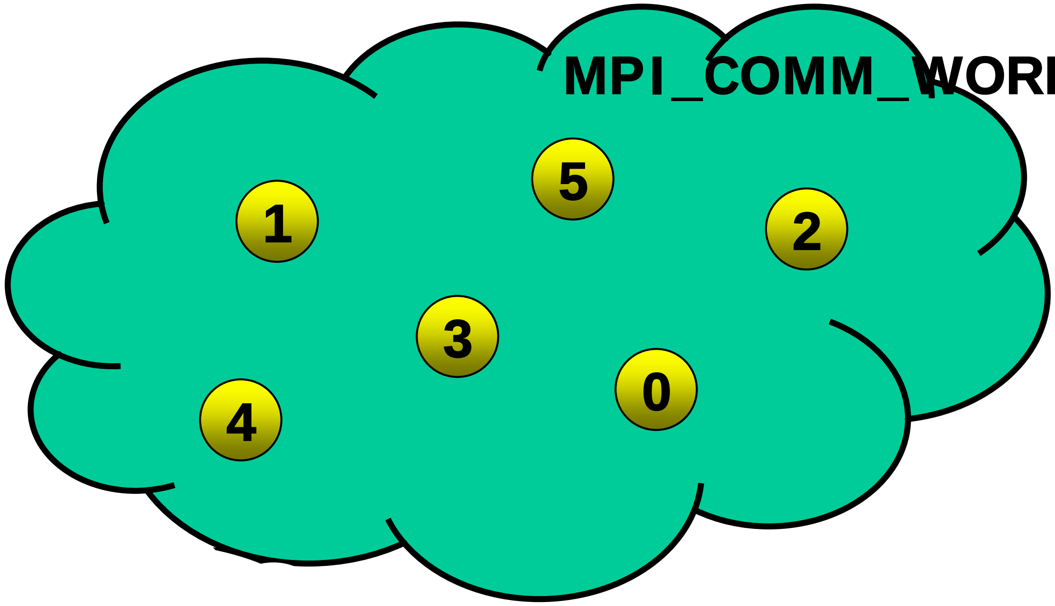
MPI Program Structure

- MPI Communicator
- MPI_Comm_world
- Header files
- MPI function format
- Initializing MPI
- Communicator Size
- Process Rank
- Exiting MPI

MPI Communicator

- Programmer view: group of processes that are allowed to communicate with each other
- All MPI communication calls have a communicator argument
- Most often you will use `MPI_COMM_WORLD`
  - Defined when you call `MPI_Init`
  - It is all of your processors...

MPI_COMM_WORLD Communicator



Header Files

- MPI constants and handles are defined here

C:

```
#include <mpi.h>
```

Fortran:

```
use mpi (ó estilo Fortran 77: include 'mpif.h')
```

MPI Function Format

C:

```
error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

Fortran:

```
CALL MPI_XXXXX(parameter,...,IERROR)
```

Initializing MPI

- Must be the first routine called (only once)

C:

```
int MPI_Init(int *argc, char ***argv)
```

Fortran:

```
CALL MPI_INIT(IERROR)

INTEGER IERROR
```

Communicator Size

- How many processes are contained within a communicator

C:

```
MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran:

```
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)

INTEGER COMM, SIZE, IERROR
```

Process Rank

- Process ID number within the communicator
  - Starts with zero and goes to (n-1) where n is the number of processes requested

- Used to identify the source and destination of messages

C:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran:

```
CALL MPI_COMM_RANK(COMM, RANK, IERROR)

INTEGER COMM, RANK, IERROR
```

Exiting MPI

- Must be called last by “all” processes

C:

```
MPI_Finalize()
```

Fortran:

```
CALL MPI_FINALIZE(IERROR)
```

Bones.c

```
#include<mpi.h>

void main(int argc, char *argv[]) {
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

/* … your code here … */

   MPI_Finalize ();
}
```

Bones.f

```
   PROGRAM skeleton
   use mpi
   INTEGER ierror, rank, size
   CALL MPI_INIT(ierror)
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
   CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

C  … your code here …

   CALL MPI_FINALIZE(ierror)
   END
```

Using MPI on the Clusters at IAC

- Compile with the MPI wrapper scripts (`mpicc`, `mpiCC`, `mpif77`, `mpif90`)
- Examples:

```
$ mpicc myprog.c
$ mpif90 myprog.f
```

- To run:

```
Ver documento "Cómo usar el cluster del IAC"
```

Exercise #1: Hello World

- Write a minimal MPI program which prints “hello world”
- Run it on several processors in parallel
- Modify your program so that only the process ranked 2 in MPI_COMM_WORLD prints out “hello world”
- Modify your program so that each process prints out its rank and the total number of processors

What's in a Message

- Messages
- MPI Basic Datatypes - C
- MPI Basic Datatypes - Fortran
- Rules and Rationale

Messages

- A message contains an array of elements of some particular MPI datatype

- MPI Datatypes:
  - Basic types
  - Derived types

- Derived types can be build up from basic types

- C types are different from Fortran types

MPI Basic Datatypes - C

MPI Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed log int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

MPI Basic Datatypes - Fortran

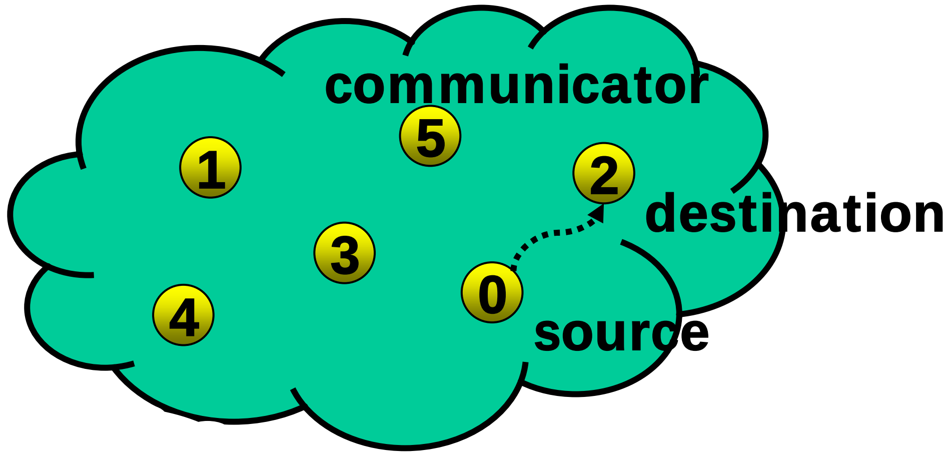
MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Rules and Rationale

- Programmer declares variables to have “normal” C/Fortran type, but uses matching MPI datatypes as arguments in MPI routines
- Mechanism to handle type conversion in a heterogeneous collection of machines
- General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send

Point-to-Point Communications

Point-to-Point Communication



- Communication between two processes
- Source process *sends* message to destination process
- Destination process *receives* the message
- Communication takes place within a communicator
- Destination process is identified by its rank in the communicator

Sending a Message

C:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm)
```

Fortran:

```
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<type>        BUF(*)
INTEGER       COUNT, DATATYPE, DEST, TAG
INTEGER       COMM, IERROR
```

Arguments

`buf`	starting <u>address</u> of the data to be sent
`count`	number of elements to be sent
`datatype`	MPI datatype of each element
`dest`	rank of destination process
`tag`	message marker (set by user)
`comm`	MPI communicator of processors involved

```
MPI_SEND(data,500,MPI_REAL,6,33,MPI_COMM_WORLD,IERROR)
```

Receiving a Message

C:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, \
     int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Fortran:

```
CALL MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
              STATUS, IERROR)

<type>        BUF(*)
INTEGER       COUNT, DATATYPE, DEST, TAG
INTEGER       COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

Wildcarding

- Receiver can wildcard
- To receive from any source

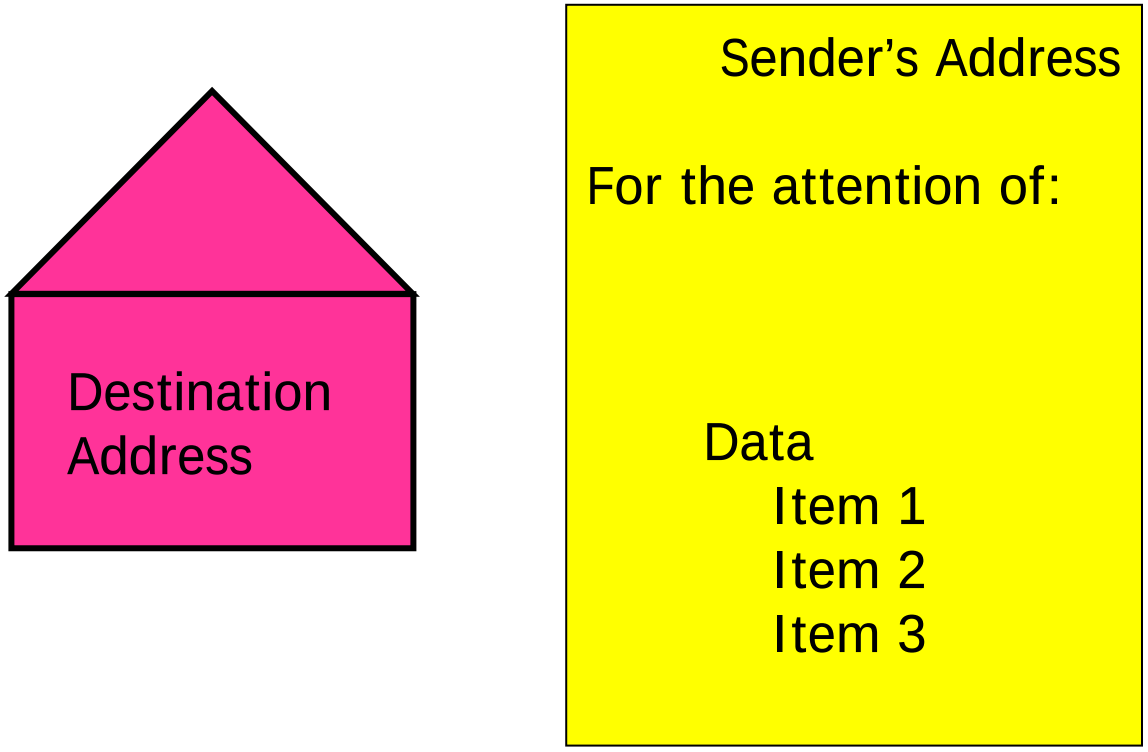
```
MPI_ANY_SOURCE
```

To receive with any tag

```
MPI_ANY_TAG
```

- Actual source and tag are returned in the receiver's `status` parameter

Communication Envelope



Communication Envelope Information

- Envelope information is returned from `MPI_RECV` as `status`

- Information includes:
  - Source: `status.MPI_SOURCE` or `status(MPI_SOURCE)`
  - Tag: `status.MPI_TAG` or `status(MPI_TAG)`
  - Count: `MPI_Get_count` or `MPI_GET_COUNT`

Received Message Count

- Message received may not fill receive buffer
- `count` is number of elements actually received

C:

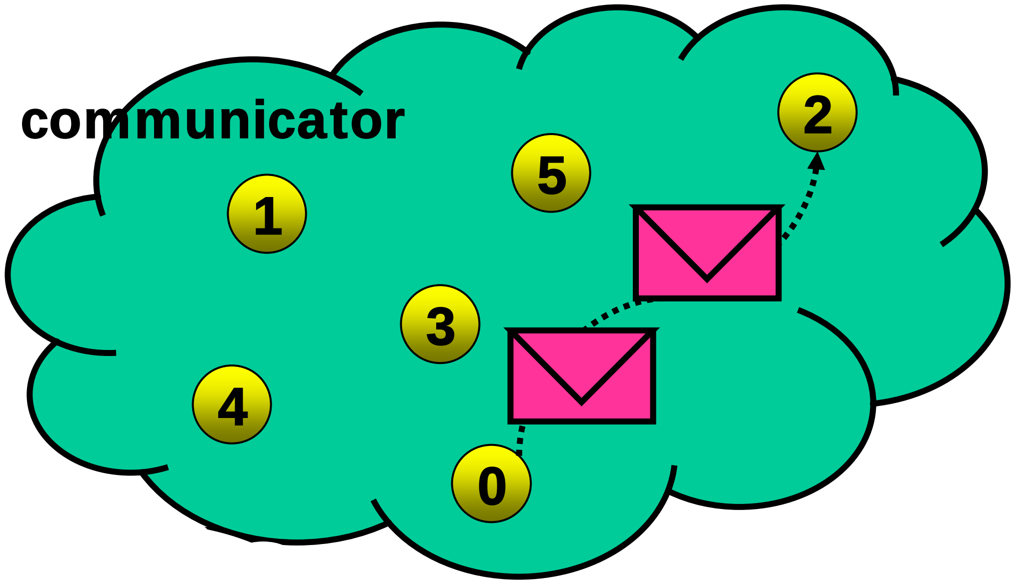
```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype, int *count)
```

Fortran:

```
CALL MPI_GET_COUNT(STATUS,DATATYPE,COUNT,IERROR)

INTEGER       STATUS(MPI_STATUS_SIZE), DATATYPE
INTEGER       COUNT,IERROR
```

Message Order Preservation



- Messages do no overtake each other
- Example: Process 0 sends two messages
  Process 2 posts two receives that match either message
  Order preserved

Sample Program #1 - Fortran

```fortran
      PROGRAM p2p
C Run with two processes
      INCLUDE 'mpif.h'
      INTEGER err, rank, size
      real data(100)
      real value(200)
      integer status(MPI_STATUS_SIZE)
      integer count
      CALL MPI_INIT(err)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
      if (rank.eq.1) then
         data=3.0
         call MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
       else
         call MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55,
     &                 MPI_COMM_WORLD,status,err)
         print *, "P:",rank," got data from processor ",
     &                 status(MPI_SOURCE)
         call MPI_GET_COUNT(status,MPI_REAL,count,err)
         print *, "P:",rank," got ",count," elements"
         print *, "P:",rank," value(5)=",value(5)
      end if
      CALL MPI_FINALIZE(err)
      END
```

```
         Program Output
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=3.
```