Tutorial on MPI: The Message-Passing Interface

William Gropp



Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
gropp@mcs.anl.gov

Course Outline

- Background on Parallel Computing
- Getting Started
- MPI Basics
- Intermediate MPI
- Tools for writing libraries
- Final comments

Thanks to Rusty Lusk for some of the material in this tutorial.

This tutorial may be used in conjunction with the book "Using MPI" which contains detailed descriptions of the use of the MPI routines.



Material that beings with this symbol is 'advanced' and may be skipped on a first reading.

Background

- Parallel Computing
- Communicating with other processes
- Cooperative operations
- One-sided operations
- The MPI process

Parallel Computing

- Separate workers or processes
- Interact by exchanging information

Types of parallel computing

All use different data for each worker

Data-parallel Same operations on different data. Also called SIMD

SPMD Same program, different data

MIMD Different programs, different data

SPMD and MIMD are essentially the same because any MIMD can be made SPMD

SIMD is also equivalent, but in a less practical sense.

MPI is primarily for SPMD/MIMD. HPF is an example of a SIMD interface.

Communicating with other processes

Data must be exchanged with other workers

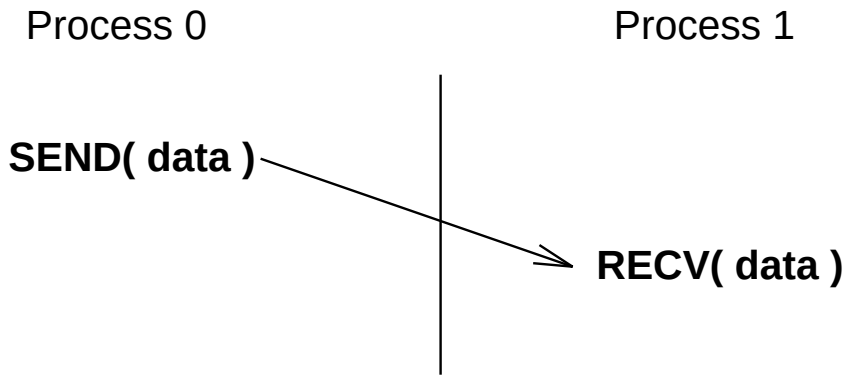
- Cooperative — all parties agree to transfer data
- One sided — one worker performs transfer of data

Cooperative operations

Message-passing is an approach that makes the exchange of data cooperative.

Data must both be explicitly sent and received.

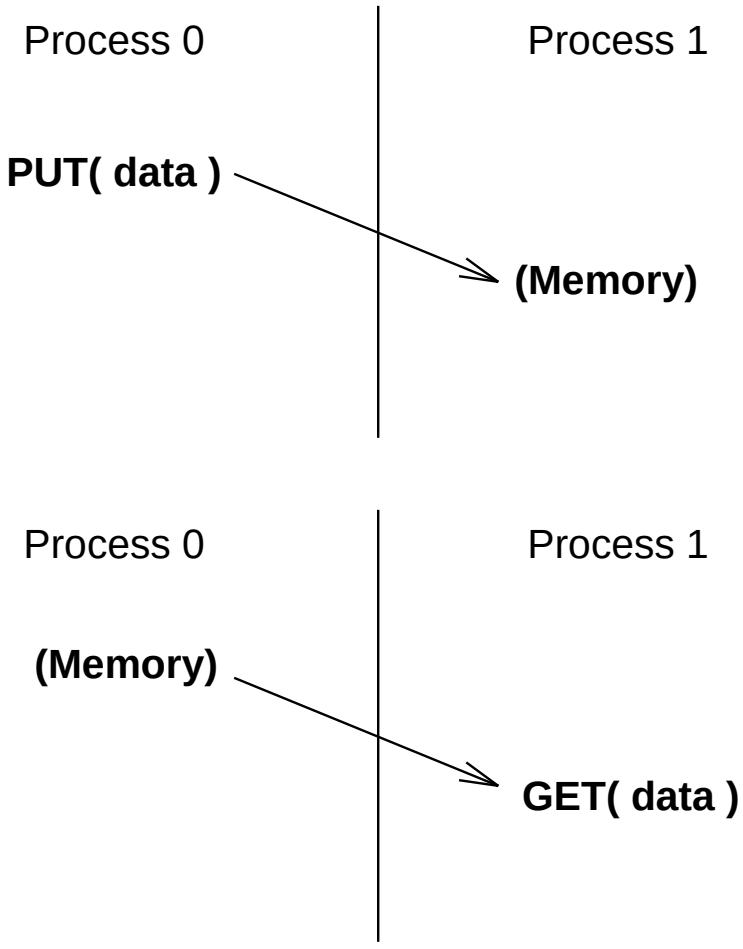
An advantage is that any change in the *receiver's* memory is made with the receiver's participation.



One-sided operations

One-sided operations between parallel processes include remote memory reads and writes.

An advantage is that data can be accessed without waiting for another process



Class Example

Take a pad of paper. Algorithm: Initialize with the number of neighbors you have

- Compute average of your neighbor's values and subtract from your value. Make that your new value.
- Repeat until done

Questions

1. How do you get values from your neighbors?
2. Which step or iteration do they correspond to? Do you know? Do you care?
3. How do you decide when you are done?

Hardware models

The previous example illustrates the hardware models by how data is exchanged among workers.

- Distributed memory (e.g., Paragon, IBM SPx, workstation network)
- Shared memory (e.g., SGI Power Challenge, Cray T3D)

Either may be used with SIMD or MIMD software models.



All memory is distributed.

What is MPI?

- A *message-passing library specification*
  - message-passing model
  - not a compiler specification
  - not a specific product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to permit (unleash?) the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - end users
  - library writers
  - tool developers

Motivation for a New Design

- Message Passing now mature as programming paradigm
  - well understood
  - efficient match to hardware
  - many applications
- Vendor systems not portable
- Portable systems are mostly research projects
  - incomplete
  - lack vendor support
  - not at most efficient level

Motivation (cont.)

Few systems offer the full range of desired features.

- modularity (for libraries)
- access to peak performance
- portability
- heterogeneity
- subgroups
- topologies
- performance measurement tools

The MPI Process

- Began at Williamsburg Workshop in April, 1992
- Organized at Supercomputing '92 (November)
- Followed HPF format and process
- Met every six weeks for two days
- Extensive, open email discussions
- Drafts, readings, votes
- Pre-final draft distributed at Supercomputing '93
- Two-month public comment period
- Final version of draft in May, 1994
- Widely available now on the Web, ftp sites, netlib (`http://www.mcs.anl.gov/mpi/index.html`)
- Public implementations available
- Vendor implementations coming soon

Who Designed MPI?

- Broad participation
- Vendors
  - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
- Library writers
  - PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
- Application specialists and consultants

Companies	Laboratories	Universities
ARCO	ANL	UC Santa Barbara
Convex	GMD	Syracuse U
Cray Res	LANL	Michigan State U
IBM	LLNL	Oregon Grad Inst
Intel	NOAA	U of New Mexico
KAI	NSF	Miss. State U.
Meiko	ORNL	U of Southampton
NAG	PNL	U of Colorado
nCUBE	Sandia	Yale U
ParaSoft	SDSC	U of Tennessee
Shell	SRC	U of Maryland
TMC		Western Mich U
		U of Edinburgh
		Cornell U.
		Rice U.
		U of San Francisco

Features of MPI

- General
  - Communicators combine context and group for message security
  - Thread safety
- Point-to-point communication
  - Structured buffers and derived datatypes, heterogeneity
  - Modes: normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), buffered
- Collective
  - Both built-in and user-defined collective operations
  - Large number of data movement routines
  - Subgroups defined directly or by topology

Features of MPI (cont.)

- Application-oriented process topologies
  - Built-in support for grids and graphs (uses groups)
- Profiling
  - Hooks allow users to intercept MPI calls to install their own tools
- Environmental
  - inquiry
  - error control

Features not in MPI

- Non-message-passing concepts not included:
  - process management
  - remote memory transfers
  - active messages
  - threads
  - virtual shared memory
- MPI does not address these issues, but has tried to remain compatible with these ideas (e.g. thread safety as a goal, intercommunicators)

Is MPI Large or Small?

- MPI is large (125 functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 functions)
  - Many parallel programs can be written with just 6 basic functions.
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.

Where to use MPI?

- You need a portable parallel program
- You are writing a parallel library
- You have irregular or dynamic data relationships that do not fit a data parallel model

Where *not* to use MPI:

- You can use HPF or a parallel Fortran 90
- You don't need parallelism at all
- You can use libraries (which may be written in MPI)

Why learn MPI?

- Portable
- Expressive
- Good way to learn about subtle issues in parallel computing

Getting started

- Writing MPI programs
- Compiling and linking
- Running MPI programs
- More information
  - *Using MPI* by William Gropp, Ewing Lusk, and Anthony Skjellum,
  - The LAM companion to "Using MPI..." by Zdzislaw Meglicki
  - *Designing and Building Parallel Programs* by Ian Foster.
  - A Tutorial/User's Guide for MPI by Peter Pacheco (ftp://math.usfca.edu/pub/MPI/mpi.guide.ps)
  - The MPI standard and other information is available at `http://www.mcs.anl.gov/mpi`. Also the source for several implementations.

Writing MPI programs

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

int main( argc, argv )
int argc;
char **argv;
{
MPI_Init( &argc, &argv );
printf( "Hello world\n" );
MPI_Finalize();
return 0;
}
```

Commentary

- `#include "mpi.h"` provides basic MPI definitions and types
- `MPI_Init` starts MPI
- `MPI_Finalize` exits MPI
- Note that all non-MPI routines are local; thus the `printf` run on each process

Compiling and linking

For simple programs, special compiler commands can be used. For large projects, it is best to use a standard Makefile.

The MPICH implementation provides the commands `mpicc` and `mpif77` as well as '`Makefile`' examples in '`/usr/local/mpi/examples/Makefile.in`'

Special compilation commands

The commands

```
mpicc -o first first.c
mpif77 -o firstf firstf.f
```

may be used to build simple programs when using MPICH.

These provide special options that exploit the profiling features of MPI

-mpilog Generate log files of MPI calls

-mpitrace Trace execution of MPI calls

-mpianim Real-time animation of MPI (not available on all systems)

There are specific to the MPICH implementation;

other implementations may provide similar commands

(e.g., `mpcc` and `mpxlf` on IBM SP2).

Using Makefiles

The file '`Makefile.in`' is a *template* Makefile. The program (script) '`mpireconfig`' translates this to a Makefile for a particular system. This allows you to use the same Makefile for a network of workstations and a massively parallel computer, even when they use different compilers, libraries, and linker options.

`mpireconfig Makefile`

Note that you must have '`mpireconfig`' in your `PATH`.

Sample Makefile.in

```
##### User configurable options #####

ARCH         = @ARCH@
COMM         = @COMM@
INSTALL_DIR = @INSTALL_DIR@
CC           = @CC@
F77          = @F77@
CLINKER      = @CLINKER@
FLINKER      = @FLINKER@
OPTFLAGS     = @OPTFLAGS@
#
LIB_PATH     = -L$(INSTALL_DIR)/lib/$(ARCH)/$(COMM)
FLIB_PATH    =
@FLIB_PATH_LEADER@$(INSTALL_DIR)/lib/$(ARCH)/$(COMM)
LIB_LIST     = @LIB_LIST@
#
INCLUDE_DIR = @INCLUDE_PATH@ -I$(INSTALL_DIR)/include

### End User configurable options ###
```

Sample Makefile.in (con't)

```
CFLAGS  = @CFLAGS@ $(OPTFLAGS) $(INCLUDE_DIR) -DMPI_$(ARCH)
FFLAGS = @FFLAGS@ $(INCLUDE_DIR) $(OPTFLAGS)
LIBS = $(LIB_PATH) $(LIB_LIST)
FLIBS = $(FLIB_PATH) $(LIB_LIST)
EXECS = hello

default: hello

all: $(EXECS)

hello: hello.o $(INSTALL_DIR)/include/mpi.h
        $(CLINKER) $(OPTFLAGS) -o hello hello.o \
        $(LIB_PATH) $(LIB_LIST) -lm

clean:
        /bin/rm -f *.o *~ PI* $(EXECS)

.c.o:
        $(CC) $(CFLAGS) -c $*.c
.f.o:
        $(F77) $(FFLAGS) -c $*.f
```

Running MPI programs

`mpirun -np 2 hello`

'`mpirun`' is not part of the standard, but some version of it is common with several MPI implementations. The version shown here is for the MPICH implementation of MPI.

 *Just as Fortran does not specify how Fortran programs are started, MPI does not specify how MPI programs are started.*

 *The option* `-t` *shows the commands that* `mpirun` *would execute; you can use this to find out how* `mpirun` *starts programs on yor system. The option* `-help` *shows all options to* `mpirun`.

Finding out about the environment

Two of the first questions asked in a parallel program are: How many processes are there? and Who am I?

How many is answered with `MPI_Comm_size` and who am I is answered with `MPI_Comm_rank`.

The rank is a number between zero and `size-1`.

A simple program

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

int main( argc, argv )
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 );
printf( "Hello world! I'm %d of %d\n",
        rank, size );
MPI_Finalize();
return 0;
}
```

Caveats



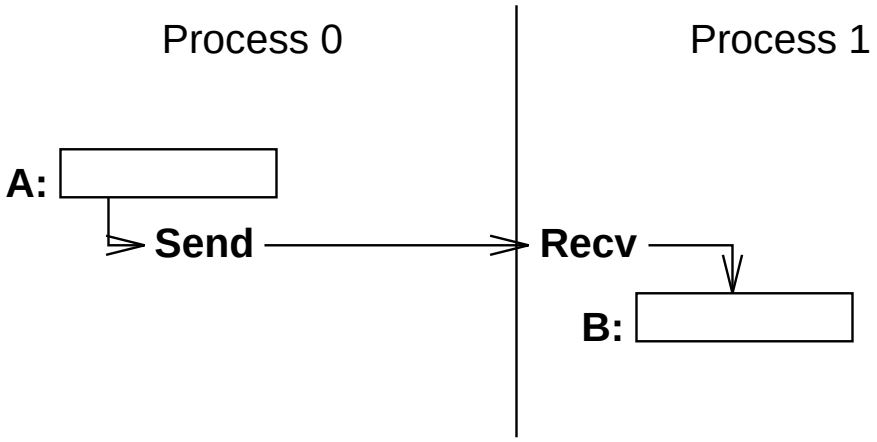
These sample programs have been kept as simple as possible by assuming that all processes can do output. Not all parallel systems provide this feature, and MPI provides a way to handle this case.

Exercise - Getting Started

Objective: Learn how to login, write, compile, and run a simple MPI program.

Run the "Hello world" programs. Try two different parallel computers. What does the output look like?

Sending and Receiving messages



Questions:

- To whom is data sent?
- What is sent?
- How does the receiver identify it?

Current Message-Passing

- A typical blocking send looks like

```
send( dest, type, address, length )
```

  where

  - `dest` is an integer identifier representing the process to receive the message.
  - `type` is a nonnegative integer that the destination can use to selectively screen messages.
  - (`address, length`) describes a contiguous area in memory containing the message to be sent.

  and

- A typical global operation looks like:

```
broadcast( type, address, length )
```

- All of these specifications are a good match to hardware, easy to understand, but too inflexible.

The Buffer

Sending and receiving only a contiguous array of bytes:

- hides the real data structure from hardware which might be able to handle it directly
- requires pre-packing dispersed data
  - rows of a matrix stored columnwise
  - general collections of structures
- prevents communications between machines with different representations (even lengths) for same data type

Generalizing the Buffer Description

- Specified in MPI by *starting address*, *datatype*, and *count*, where datatype is:
  - elementary (all C and Fortran datatypes)
  - contiguous array of datatypes
  - strided blocks of datatypes
  - indexed array of blocks of datatypes
  - general structure
- Datatypes are constructed recursively.
- Specifications of elementary datatypes allows heterogeneous communication.
- Elimination of length in favor of count is clearer.
- Specifying application-oriented layout of data allows maximal use of special hardware.

Generalizing the Type

- A single type field is too constraining. Often overloaded to provide needed flexibility.
- Problems:
  - under user control
  - wild cards allowed (`MPI_ANY_TAG`)
  - library use conflicts with user and with other libraries

Sample Program using Library Calls

`Sub1` and `Sub2` are from different libraries.

```
Sub1();
Sub2();
```

`Sub1a` and `Sub1b` are from the same library

```
Sub1a();
Sub2();
Sub1b();
```

Thanks to Marc Snir for the following four examples