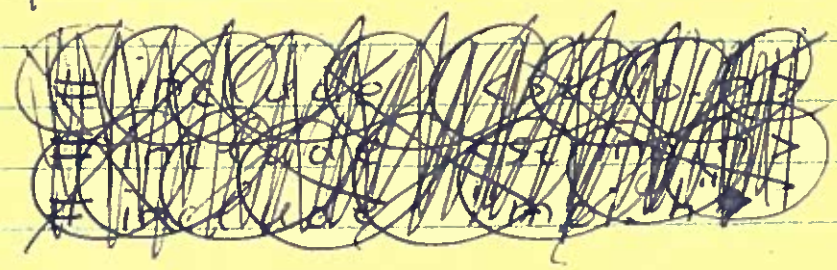
(PPMPI CH 3) L5

INTRODUCTION TO MPI

variation of the parallel hello world program, each process other than zero, sends a msg to process 0.

the processes involved in the execution of a parallel program are identified by a sequence of integers 0, 1, ..., p-1

processes 1, ... p-1 will send a msg to process 0. we need to know that process 0 received the messages.



→ parallel-hello_word-msg.c

```
/* greetings.c -- greetings program
 *
 * Send a message from all processes with rank != 0 to process 0.
 *    Process 0 prints the messages received.
 *
 * Input: none.
 * Output: contents of messages received by process 0.
 *
 * See Chapter 3, pp. 41 & ff in PPMPI.
 */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

main(int argc, char* argv[]) {
    int         my_rank;       /* rank of process      */
    int         p;             /* number of processes  */
    int         source;        /* rank of sender       */
    int         dest;          /* rank of receiver     */
    int         tag = 0;       /* tag for messages     */
    char        message[100];  /* storage for message  */
    MPI_Status  status;        /* return status for    */
                               /* receive              */

    /* Start up MPI */
    MPI_Init(&argc, &argv);

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR,
            dest, tag, MPI_COMM_WORLD);
    } else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag,
                MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    /* Shut down MPI */
    MPI_Finalize();
} /* main */
```

sprintf // write formatted data to string

compile/run with 4 procs:

Greetings from process 1!
" " " 2!
" " " 3!
} in order!

inner mechanics of program execution:

(1) a directive is issued to the OS that has the effect of placing a copy of the executable program on each proc.

(2) each proc. begins execution of its copy of the executable

(3) different processes can execute different statements by branching within the program, based on their process ranks.

in general: each process runs a different program (MIMD programming)

in practice: this is achieved by branching statements within a single program (SPMD programming)

(single-program multiple-data)

```
if (my_rank != 0)
    ...
else
    ...
```

MPI/C program consists of conventional C statements and preprocessor directives.

MPI is a library of definitions & functions that can be used in C/Fortran programs.

General MPI/C program

1) must contain the preprocessor directive
   #include "mpi.h"

2) consistent naming scheme: start with "MPI_"
   (for MPI-defined identifiers)
   remaining part: CHAR for constants
   Init for functions

3) before any other MPI fct's can be called, the fct MPI_Init must be called, only ONCE.
   its param's are pointers to the main's argc argv

4) after a program has finished using the MPI library, it must call MPI_Finalize, "clean-up"
   e.g. free memory allocated by MPI

typical MPI program layout:

```
#include "mpi.h"
   ⋮
main (int argc, char* argv[]) {
   ⋮
MPI_Init (&argc, &argv);
   ⋮                          ← MPI fct calls
MPI_Finalize ();
   ⋮
}
```

Communicators

the flow of control in an SPMD program depends on the rank of a process.

MPI fct MPI_Comm_rank returns the rank of a process in its 2nd parameter.
its first parameter is a communicator
i.e. (a collection of processes that can send msg's to each other)

the predefined comm/tor MPI_COMM_WORLD consists of all processes running when program execution begins.

MPI fct MPI_Comm_size returns #processes in a comm/tor, executing the program.

MSG: Data + Envelope

The actual msg passing is carried out by MPI fct^s MPI_Send, MPI_Recv.

MPI_Send sends a msg to a designated process.

MPI_Recv receives a msg from a process.

(I) suppose process A wants to send a msg to process B.

(1) the msg must be addressed "enclose it in an envelope" add the address — so that the msg passing system knows where to deliver it

(2) the system needs to determine the size of the msg. OR the end of the msg. — msg passing systems enclose msg^s to envelopes

(3) destination + size should be enough for B to receive the msg, by calling MPI_Recv

(II) A sends a msg to B, asking for data
C sends a msg to B, containing values
D sends a msg to B, that should be printed

(4) add the address of the source process to the envelope, so that B can act accordingly

(III) B receives floats from several processes. {some are to be printed, others to be stored in an array, and a single process can send both kinds. How does B distinguish btw

sol: use tags or msg types

a tag is an integer from 0 to $2^{15}-1$ specified by the programmer, and added to the msg envelope.

floats to be printed: tag 0
" " " stored: tag 1

when B receives the msg, it will check the value of the tag and act accordingly.

(IV) suppose that a program uses a library to solve systems of linear eq.s and that fct's in the library need to do msg passing.

how can we distinguish btw msg's a process A sends and msg's sent by fct's in the library?
(the might use the same tags)

sol: add a comm/tor to the msg envelope.

2 processes using distinct comm/tors cannot receive msg from each other.

summary: the msg envelope contains (at least)

1. rank of the sender/receiver
2. tag
3. comm/tor.

SENDING MESSAGES (8)

syntax for MPI_Send, MPI_Recv

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

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

(1) contents of msg are stored in a block of memory referenced by "message"

(2) the msg contains a seq of "count" values, each having MPI type "datatype" (RTM for correspondence btw MPI & C data types)

NOTE: amount of space allocated for the receiving buffer does not have to match the exact amount of space of the msg being received.

(3) "dest" rank of the receiving process
"source" " " " sending process

MPI allows "source" to be a wildcard. predefined constant "MPI_ANY_SOURCE" for a process to be able to receive a msg from any sending process.

(4) "tag" is an int
"comm" is a comm/tor, predefined MPI_COMM_WORLD

(mechanisms to partition the msg space)

MPI_Recv can use the wildcard MPI_ANY_TAG for a tag.

for process A to send a msg to process B, the arg. comm that A uses in its MPI_Send call must be identical to the arg that B uses in its call to MPI_Recv.

A must use a tag, to send
B can use either an identical tag, or MPI_ANY_TAG, to receive

NOTE: using wildcards for the args "source" and "tag" by MPI_Recv, but not by MPI_Send, indicates a PUSH com/tion mechanism, i.e. data transfer is effectively carried out by the sender (PULL, receiver)

(5) "status" returns info on the data actually received

it references a struct with at least 3 members: { MPI_SOURCE, MPI_TAG, MPI_ERROR }

if the source of the received msg was MPI_ANY_SOURCE, then status->MPI_SOURCE will contain the rank of the process that sent the msg.

to determine the size of the msg received, we call: MPI_Get_count

RTM
```
int MPI_Get_count(
        MPI_Status*   status
        MPI_Datatype  datatype
        int*          count_ptr )
```

(6) MPI_Send & MPI_Recv have integer return values, that are error codes