2 Greetings!

The first C program that most of us saw was the "Hello, world!" program in Kernighan and Ritchie's classic text, *The C Programming Language* [3]. It simply prints the message "Hello, world!" A variant that makes some use of multiple processes is to have each process send a greeting to another process.

In MPI, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers. If there are p processes executing a program, they will have ranks 0, 1, ..., $p-1$. The following program has each process other than 0 send a message to process 0, and process 0 prints out the messages it received.

```
#include <stdio.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 = 50;       /* Tag for messages */
    char message[100];  /* Storage for the message */
    MPI_Status status;  /* Return status for receive */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    if (my_rank != 0) {
        sprintf(message, "Greetings from process %d!",
            my_rank);
        dest = 0;
        /* Use strlen(message)+1 to include '\0' */
        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);
        }
    }

    MPI_Finalize();
} /* main */
```

The details of compiling and executing this program depend on the system you're using. So ask your local guide how to compile and run a parallel program that uses MPI. We discuss the freely available systems in an appendix.

When the program is compiled and run with two processes, the output should be

```
Greetings from process 1!
```

If it's run with four processes, the output should be

```
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
```

Although the details of what happens when the program is executed vary from machine to machine, the essentials are the same on all machines, provided we run one process on each processor.

1. The user issues a directive to the operating system which has the effect of placing a copy of the executable program on each processor.

2. Each processor begins execution of its copy of the executable.

3. Different processes can execute different statements by branching within the program. Typically the branching will be based on process ranks.

So the Greetings program uses the *Single Program Multiple Data* or *SPMD* paradigm. That is, we obtain the *effect* of different programs running on different processors by taking branches within a single program on the basis of process rank: the statements executed by process 0 are different from those

executed by the other processes, even though all processes are running the same program. This is the most commonly used method for writing MIMD programs, and we'll use it exclusively in this *Guide.*

2.1 General MPI Programs

Every MPI program must contain the preprocessor directive

```
#include "mpi.h"
```

This file, mpi.h, contains the definitions, macros and function prototypes necessary for compiling an MPI program.

Before any other MPI functions can be called, the function MPI_Init must be called, and it should only be called once. Its arguments are pointers to the main function's parameters — argc and argv. It allows systems to do any special set-up so that the MPI library can be used. After a program has finished using the MPI library, it must call MPI_Finalize. This cleans up any "unfinished business" left by MPI — e.g., pending receives that were never completed. So a typical MPI program has the following layout.

```
          ⋮
#include "mpi.h"
          ⋮
main(int argc, char** argv) {
          ⋮
    /* No MPI functions called before this */
    MPI_Init(&argc, &argv);
          ⋮
    MPI_Finalize();
    /* No MPI functions called after this */
          ⋮
} /* main */
          ⋮
```

2.2 Finding Out About the Rest of the World

MPI provides the function MPI_Comm_rank, which returns the rank of a process in its second argument. Its syntax is

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

The first argument is a *communicator*. Essentially a communicator is a collection of processes that can send messages to each other. For basic programs, the only communicator needed is MPI_COMM_WORLD. It is predefined in MPI and consists of all the processes running when program execution begins.

Many of the constructs in our programs also depend on the number of processes executing the program. So MPI provides the function MPI_Comm_size for determining this. Its first argument is a communicator. It returns the number of processes in a communicator in its second argument. Its syntax is

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

2.3 Message: Data + Envelope

The actual message-passing in our program is carried out by the MPI functions MPI_Send and MPI_Recv. The first command sends a message to a designated process. The second receives a message from a process. These are the most basic message-passing commands in MPI. In order for the message to be successfully communicated the system must append some information to the data that the application program wishes to transmit. This additional information forms the *envelope* of the message. In MPI it contains the following information.

1. The rank of the receiver.
2. The rank of the sender.
3. A tag.
4. A communicator.

These items can be used by the receiver to distinguish among incoming messages. The *source* argument can be used to distinguish messages received

from different processes. The *tag* is a user-specified int that can be used to distinguish messages received from a single process. For example, suppose process A is sending two messages to process B; both messages contain a single float. One of the floats is to be used in a calculation, while the other is to be printed. In order to determine which is which, A can use different tags for the two messages. If B uses the same two tags in the corresponding receives, when it receives the messages, it will "know" what to do with them. MPI guarantees that the integers 0–32767 can be used as tags. Most implementations allow much larger values.

As we noted above, a communicator is basically a collection of processes that can send messages to each other. When two processes are communicating using MPI_Send and MPI_Receive, its importance arises when separate modules of a program have been written independently of each other. For example, suppose we wish to solve a system of differential equations, and, in the course of solving the system, we need to solve a system of linear equations. Rather than writing the linear system solver from scratch, we might want to use a *library* of functions for solving linear systems that was written by someone else and that has been highly optimized for the system we're using. How do we avoid confusing the messages *we* send from process A to process B with those sent by the library functions? Before the advent of communicators, we would probably have to partition the set of valid tags, setting aside some of them for exclusive use by the library functions. This is tedious and it will cause problems if we try to run our program on another system: the other system's linear solver may not (probably won't) require the same set of tags. With the advent of communicators, we simply create a communicator that can be used exclusively by the linear solver, and pass it as an argument in calls to the solver. We'll discuss the details of this later. For now, we can get away with using the predefined communicator MPI_COMM_WORLD. It consists of all the processes running the program when execution begins.

2.4 MPI_Send and MPI_Receive

To summarize, let's detail the syntax of MPI_Send and MPI_Receive.

```
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)
```

Like most functions in the standard C library most MPI functions return an integer error code. However, like most C programmers, we will ignore these return values in most cases.

The contents of the message are stored in a block of memory referenced by the argument message. The next two arguments, count and datatype, allow the system to identify the end of the message: it contains a sequence of count values, each having *MPI* type datatype. This type is not a C type, although most of the predefined types correspond to C types. The predefined MPI types and the corresponding C types (if they exist) are listed in the following table.

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long 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	

The last two types, MPI_BYTE and MPI_PACKED, don't correspond to standard C types. The MPI_BYTE type can be used if you wish to force the system to perform no conversion between different data representations (e.g., on a heterogeneous network of workstations using different representations of data). We'll discuss the type MPI_PACKED later.

Note that the amount of space allocated for the receiving buffer does not have to match the exact amount of space in the message being received. For

example, when our program is run, the size of the message that process 1 sends, strlen(message)+1, is 28 chars, but process 0 receives the message in a buffer that has storage for 100 characters. This makes sense. In general, the receiving process may not know the exact size of the message being sent. So MPI allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage, an overflow error occurs [4].

The arguments dest and source are, respectively, the ranks of the receiving and the sending processes. MPI allows source to be a "wildcard." There is a predefined constant MPI_ANY_SOURCE that can be used if a process is ready to receive a message from *any* sending process rather than a particular sending process. There is *not* a wildcard for dest.

As we noted earlier, MPI has two mechanisms specifically designed for "partitioning the message space:" tags and communicators. The arguments tag and comm are, respectively, the tag and communicator. The tag is an int, and, for now, our only communicator is MPI_COMM_WORLD, which, as we noted earlier is predefined on all MPI systems and consists of all the processes running when execution of the program begins. There is a wildcard, MPI_ANY_TAG, that MPI_Recv can use for the tag. There is *no* wildcard for the communicator. In other words, in order for process A to send a message to process B, the argument comm that A uses in MPI_Send must be identical to the argument that B uses in MPI_Recv.

The last argument of MPI_Recv, status, returns information on the data that was actually received. It references a record with with two fields — one for the source and one for the tag. So if, for example, the *source* of the receive was MPI_ANY_SOURCE, then status will contain the rank of the process that sent the message.