1 MPI Hands-On - Introduction to MPI

1.1 Parallel Computing

- Separate workers or processes.
- Interact by exchanging information.
- Data-Parallel. Same operations on different data. Also called SIMD.
- SPMD. Same program, different data.
- MIMD. Different programs, different data.

1.2 Communicating with other processes

Data must be exchanged with other workers;

- **Cooperative** — all parties agree to transfer data.
  - Message-passing is an approach that makes the exchange of data cooperative.
  - Data must both be explicitly sent and received.

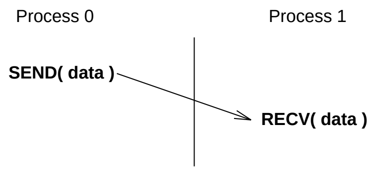

Figure 1: Cooperative–Communicating with other processes.

- **One sided** — one worker performs transfer of data.
  - 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.

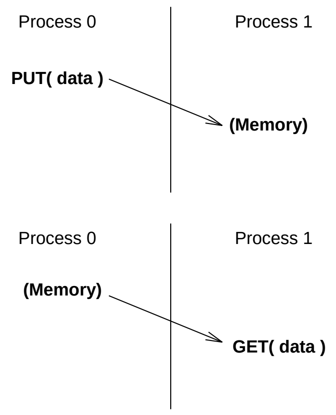

Figure 2: One sided–Communicating with other processes.

1.3 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.
- Designed to provide access to advanced parallel hardware for
  - end users.
  - library writers.
  - tool developers.

1.4 MPI Implementations

- Open MPI (a project combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI))
- MPICH (Argonne National Laboratory).
- UNIFY (Mississippi State University).
- CHIMP (Edinburgh Parallel Computing Centre).
- LAM (Ohio Supercomputer Center).

- MPI for the Fujitsu AP1000 (Australian National University).
- Cray MPI Product for the T3D (Cray Research and the Edinburgh Parallel Computing Center).
- IBM's MPI for the SP.
- SGI's MPI for 64-bit mips3 and mips4.
- PowerMPI for Parsytec Systems.
- HP's MPI implementation.

1.5 Is MPI Large or Small?

- MPI is large (125 functions)(See this link)
  - MPI's extensive functionality requires many functions.
  - Number of functions not necessarily a measure of complexity.
- MPI is small. Many parallel programs can be written with just 6 basic functions.
  - **MPI_Init**– Initialise MPI.
  - **MPI_Comm_size**– Find out how many processes there are.
  - **MPI_Comm_rank**– Find out which process I am.
  - **MPI_Send**– Send a message.
  - **MPI_Recv**– Receive a message.
  - **MPI_Finalize**– Terminate MPI.
- MPI is just right
  - One can access flexibility when it is required.
  - One need not master all parts of MPI to use it.

1.6 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).

1.7 How To Use MPI? Essential!!

1. When possible, start with a debugged serial version.
2. Design parallel algorithm.
3. Write code, making calls to MPI library.
4. Compile and run using implementation specific utilities.
5. Run with a few nodes first, increase number gradually.

1.8 Getting started

1.8.1 Writing MPI programs I

First program with MPI (hello.c). Write the following code and study the response.

```
#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;
}
```

- `#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.

```
mpicc -o hello hello.c
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.

1.8.2 Writing MPI programs II

Another Example (Again no messsage-passing) (hello1.c):

```
#include <stdio.h>
#include <mpi.h>
main(argc, argv)
int argc;
char *argv[];
{
char name[BUFSIZ];
int length;
MPI_Init(&argc, &argv);
MPI_Get_processor_name(name, &length);
printf("%s: hello world\n", name);
MPI_Finalize();
}
```

```
#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;
}
```

1.8.3 Writing MPI programs III

Another Example (Again hello and again no messsage-passing) (hello2.c): Two of the first questions asked in a parallel program are:

1. How many processes are there? Answered with ***MPI_Comm_size***
2. Who am I? Answered with ***MPI_Comm_rank***. **The rank** is a number between zero and **size**-1.

1.8.4 Exercise - Getting Started

- Designing, compiling, and runing a simple MPI program.
  - Write a program that combines all the "Hello world" programs above.
  - Execute several times and/or try different number of nodes. What does the output look like? Why it does differ?