



MPI introduction

EUDAT – PRACE Summer School on managing scientific data from analysis to long term archiving, 23-27 September 2019, Trieste, Italy

Leon Kos, University of Ljubljana

The Message-Passing Model

- Unlike the shared memory model, resources are local
- MPI is for communication among processes, which have separate address spaces.
- Interprocess communication consists of
 - Synchronization
 - Movement of data from one process's address space to another's.

Why MPI

- Scalable to thousands of processes
- MPI provides a powerful, efficient, and *portable* way to express parallel programs
- Many libraries use MPI and thus programs eliminate the need of knowing programming in MPI.

Minimal MPI

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

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

Try to run it with SLURM

1. `module load gnu openmpi`
2. `mpicc hello-mpi.c`
3. `srun -n 6 --partition=gll_usr_gpuprod
--account=train_ceudat19 a.out`
4. Also `--reservation=s_tra_eudat`
and `--partition=gll_usr_gpuprod` available

```
program main  
  use mpi  
  integer ierr
```

- Fortran example
uses
`mpif90 hello-mpi.f90`
instead

```
call MPI_INIT( ierr )  
print *, 'Hello, world!'  
call MPI_FINALIZE( ierr )  
end
```

Rank and communicator

- A process is identified by its *rank* in the group associated with a communicator
- **MPI_Comm_size** reports the number of processes.
- **MPI_Comm_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**.

Updated hello-mpi.{c,f90}

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

int 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 );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}

program main
use mpi
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Point-To-Point Message Passing – Data transfer and Synchronization

- The sender process cooperates with the destination process
- The communication system must allow the following three operations
 - send(message)
 - receive (message)
 - synchronisation

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
 - `MPI_INIT`
 - `MPI_FINALIZE`
 - `MPI_COMM_SIZE`
 - `MPI_COMM_RANK`
 - `MPI_SEND`
 - `MPI_RECV`
- Point-to-point (send/recv) isn't the only way

Send/Receive P-t-P

```
program main
implicit none
use mpi
integer ierr, rank, size
integer status(MPI_STATUS_SIZE)
real data(2)

call MPI_INIT( ierr )
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
if (rank .eq. 0) then
  data(1)=1
  data(2)=2
  call MPI_SEND(data, 2, MPI_REAL, 1, 2929, MPI_COMM_WORLD, ierr)
else if (rank.eq.1) then
  call MPI_RECV(data, 2, MPI_REAL, 0, 2929, MPI_COMM_WORLD, status, ierr)
  print *, data(1), data(2)
endif
call MPI_FINALIZE( ierr )
end
```

Standard Send and Receive in C

- `int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`
- `int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status);`

C example

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char * argv[])
{
    int err, size, rank;
    MPI_Status status;
    float data[2];
    err = MPI_Init(&argc, &argv);
    err = MPI_Comm_size(MPI_COMM_WORLD, &size);
    err = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if( rank == 0 ) {
        data[0] = 1.0, data[1] = 2.0;
        MPI_Send(data, 2, MPI_FLOAT, 1, 1230, MPI_COMM_WORLD);
    } else if( rank == 1 ) {
        MPI_Recv(data, 2, MPI_FLOAT, 0, 1230, MPI_COMM_WORLD, &status);
        printf("%d: a[0]=%f a[1]=%f\n", rank, a[0], a[1]);
    }
    err = MPI_Finalize();
}
```

Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency

Summary

- MPI is a **standard** for message-passing and has numerous implementations (OpenMPI, IntelMPI, MPICH, etc)
- MPI uses send and receive calls to manage communications between two processes (point-to-point)
- The calls can be blocking or non-blocking.
- Non-blocking calls can be used to overlap communication with computation but wait routines are needed for synchronization.
- Deadlock is a common error and is due to incorrect order of send/receive

THANK YOU FOR YOUR
ATTENTION

www.prace-ri.eu