MPI (Message Passing Interface)

The MPI standard is used to write and run parallel programs across MIMD PEs, be they cores inside a processor or nodes in a cluster — debugging runs do not even require a parallel machine. We generally use OpenMPI, **www.open-mpi.org**, with the C programming language.

MPI Environment

- **#include <mpi.h>**
- int **MPI_Init**(int **argc, char ***argv)
- int **MPI_Finalize**(void)

MPI Common Constants

- **MPI_SUCCESS**, **MPI_ERR**, **MPI_ANY_TAG**, **MPI_ANY_SOURCE**, **MPI_COMM_WORLD**, **MPI_NULL_WINDOW**
- **MPI_Datatype** values: **MPI_BYTE**, **MPI_CHAR**, **MPI_SHORT**, **MPI_INT**, **MPI_LONG**, **MPI_UNSIGNED_CHAR**, **MPI_UNSIGNED_SHORT**, **MPI_UNSIGNED**, **MPI_UNSIGNED_LONG**, **MPI_FLOAT**, **MPI_DOUBLE**
- **MPI_Op** values: **MPI_MAX**, **MPI_MIN**, **MPI_SUM**, **MPI_PROD**, **MPI_BAND**, **MPI_BOR**, **MPI_BXOR**, **MPI_LAND**, **MPI_LOR**, **MPI_LXOR**, **MPI_REPLACE**

MPI Message Passing

- int **MPI_Send**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm c)
- int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm c, MPI_Status *s)
- int **MPI_Sendrecv**(void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int src, int recvtag, MPI_Comm c, MPI_Status *s)
- int **MPI_Sendrecv_replace**(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int src, int recvtag, MPI_Comm c, MPI_Status *s)
- int **MPI_Get_count**(MPI_Status *s, MPI_Datatype datatype, int *count)
- int **MPI_Iprobe**(int src, int tag, MPI_Comm c, int *flag, MPI_Status *s)

MPI Collective Communications

- int **MPI_Bcast**(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm c)
- int **MPI_Allgather**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm c)
- int **MPI_Gather**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm c)
- int **MPI_Scatter**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm c)
- int **MPI_Alltoall**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm c)
- int **MPI_Reduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm c)
- int **MPI_Allreduce**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm c)
- int **MPI_Scan**(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm c)
- int **MPI_Barrier**(MPI_Comm c)

MPI Remote Memory Access

- int **MPI_Win_create**(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm c, MPI_Win *w)
- int **MPI_Win_free**(MPI_Win *w)
- int **MPI_Put**(void *orgaddr, int orgcount, MPI_Datatype orgtyp, int targrank, MPI_Aint targdisp, int targcount, MPI_Datatype targtyp, MPI_Win w)
- int **MPI_Get**(void *orgaddr, int orgcount, MPI_Datatype orgtyp, int targrank, MPI_Aint targdisp, int targcount, MPI_Datatype targtyp, MPI_Win w)
- int **MPI_Accumulate**(void *orgaddr, int orgcount, MPI_Datatype orgtyp, int targrank, MPI_Aint targdisp, int targcount, MPI_Datatype targtyp, MPI_Op op, MPI_Win w)
- int **MPI_Win_fence**(int assert, MPI_Win w)

MPI Commands

- To compile:
  mpicc *file.c* -o *file*
- To run on *N* PEs in nodes listed in *names*:
  mpirun -n *N* -hostfile *names file*

MPI Sample Program

Compute approximate value of π; algorithm from M. J. Quinn, *Parallel Computing Theory And Practice*, McGraw Hill, 1994.

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

main(int argc, char **argv)
{
  register double width;
  double sum, lsum;
  double rsum = 0.0;
  register int intervals, i;
  int nproc, iproc;

  if (MPI_Init(&argc, &argv) !=
      MPI_SUCCESS) exit(1);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
  intervals = atoi(argv[1]);
  width = 1.0 / intervals;
  lsum = 0;
  for (i=iproc; i<intervals; i+=nproc) {
    double x = (i + 0.5) * width;
    lsum += 4.0 / (1.0 + x * x);
  }
  lsum *= width;
  MPI_Reduce(&lsum, &sum, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
  if (iproc == 0)
    printf("Pi=%f or %f\n", sum, rsum);
  MPI_Finalize();
  return(0);
}
```