Message Passing Paradigm
MPI : Distributed memory systems

MPI

MPI stands for **Message Passing Interface.**

- Library of subroutines/functions - not a language.
- Programmer insert appropriate MPI subroutine/ function calls, compile and finally link with MPI message passing library.

It is possible to write fully-functional message-passing programs by using **only six routines**. The first three examples.

Typical MPI Program Structure

MPI include file (header)

Initialize MPI environment

Message Passing Calls

Terminate MPI Environment

All MPI routines, data-types, and constants are prefixed by "`MPI_`".

The return code for successful completion is `MPI_SUCCESS`.

Six MPI Functions

These six functions allows us to write many programs:

```
MPI_Init()
MPI_Finalize()
MPI_Comm_size()
MPI_Comm_rank()
MPI_Send()
MPI_Recv()
```

MPI Processes

- MPI is process oriented - program consists of multiple processes, each corresponding to one processor.
- In practice - runs its own copy of the same code (SPMD).

- MPI process and threads - MPI process can contain a single thread or multiple threads.

- MPI processes are identified by their **ranks**
  - *If total `nprocs` processes in computation, rank ranges from `0, 1, …, nprocs-1`.*
  - *`nprocs` does not change during computation.*

Example

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

int main(int argc, char **argv)
{
  int my_rank, nprocs;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf(“This is process %d among %d processes\n”,
         my_rank, nprocs);
  MPI_Finalize();
  return 0;
}
```

MPI Environment

- MPI_Init
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Finalize

- MPI_Wtime

  Returns an elapsed wall clock time in seconds on the calling processor

Initialization and Termination

- `MPI_Init()` initializes MPI environment
- Must be called before any other MPI routine
- Called only once; subsequent calls are erroneous.

- `MPI_Finalize()` cleans up MPI environment
  - *Must be called before exits.*
  - *No other MPI routine can be called after this call*
  - *Exception: `MPI_Initialized()`; `MPI_Get_version()`, `MPI_Finalized()`.*

- Abnormal termination: `MPI_Abort()`
  - *Makes a best attempt to abort all tasks*

Communicators and Process Groups

- A communicator defines a *communication domain* - a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type `MPI_Comm`.
- Communicators are used as arguments to all message transfer MPI routines.
- A process can belong to many different communication domains.
- MPI defines a default communicator called `MPI_COMM_WORLD` which includes all the processes.
- MPI_Comm_size

  Determines the number of processes in the group associated with a communicator
- MPI_Comm_rank

  Determines the rank of the calling process within the communicator

Communicators

- ➢ How many CPUs: `MPI_COMM_SIZE()`
- ➢ Who am I: `MPI_COMM_RANK()`

- ➢ Can be used for data decomposition etc.
  - ▪ *Suppose we know total number of grid points, total number of cpus and current cpu id.*
  - ▪ *We can calculate which portion of data current cpu is to work on.*

- ➢ Ranks used to specify source and destination of communications.

`my_rank` value different on different processors !

Second Example

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

int main(int argc, char **argv)
{
  char message[256];
  int my_rank;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  if(my_rank==0){
    strcpy(message,"Hello");
    MPI_Send(message,strlen(message)+1,MPI_CHAR,1,99,MPI_COMM_WORLD);
  }
  else if(my_rank==1) {
    MPI_Recv(message,256,MPI_CHAR,0,99,MPI_COMM_WORLD,&status);
    printf("Process %d received: %s\n",my_rank,message);
  }
  MPI_Finalize();
  return 0;
}
```

6 MPI functions:

MPI_Init()
MPI_Finalize()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Send()
MPI_Recv()

MPI Communications

- Point-to-point communications
  - *Involves a sender and a receiver, one processor to another processor*
  - *Only the two processors participate in communication*

- Collective communications
  - *All processors within a communicator participate in communication e.g. Barrier, reduction operations, gather, …*

Send / Receive

```
…
MPI_Send(message,strlen(message)+1,MPI_CHAR,
1,99,MPI_COMM_WORLD);
MPI_Recv(message,256,MPI_CHAR,0,99,MPI_COMM_WORLD,&status);
…
```

- Message data: what to send, what to receive?
  - *Where is the message? Where to put it?*
  - *What kind of data is it? What is the size?*
- Message envelope: where to send/receive?
  - *Sender, receiver*
  - *Communication context*
  - *Message tag.*

Send

```
int MPI_Send(void *buf,int count,MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
MPI_SEND(BUF,COUNT,DATATYPE,DEST,TAG,COMM,IERROR)
```

- `buf` – memory address of start of message
- `count` – number of data items
- `datatype` – what type each data item is (integer, character, double, float …)
- `dest` – rank of receiving process
- `tag` – additional identification of message
- `comm` – communicator, usually MPI_COMM_WORLD

```
char message[256];
MPI_Send(message,strlen(message)+1,MPI_CHAR,
1,99,MPI_COMM_WORLD);
```

Receive

**int MPI_Recv(void *buf,int count,MPI_Datatype datatype,int source,int tag,
MPI_Comm comm,MPI_Status *status)**

MPI_RECV(BUF,COUNT,DATATYPE,SOURCE,TAG,COMM,STATUS,IERROR)

- `buf` – initial address of receive buffer
- `count` – number of elements in receive buffer (size of receive buffer)
- `datatype` – data type in receive buffer
- `source` – rank of sending process
- `tag` – additional identification for message
- `comm` – communicator, usually MPI_COMM_WORLD
- `status` – object containing additional info of received message
- `ierror` – return code

```
char message[256];
MPI_Recv(message,256,MPI_CHAR,0,99,MPI_COMM_WORLD,&status);
```

Actual number of data items received can be queried from status object; it may be smaller than count, but cannot be larger (if larger → overflow error).

Basic MPI Data Types

MPI datatype	C datatype
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`
`MPI_DOUBLE`	`double`
`MPI_FLOAT`	`float`
`MPI_LONG_DOUBLE`	`long double`
`MPI_BYTE`	
`MPI_PACKED`	

Essentials of communication

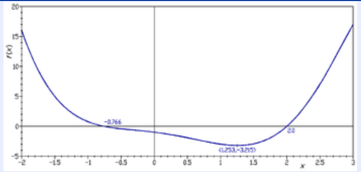
- Sender must specify valid destination.
- Sender and receiver data type, tag, communicator must match.
- Receiver returns extra (status) parameter to report info regarding message received.
- Sender specifies size of sendbuf; receiver specifies *upper bound* of recvbuf.

Message Passing

- Both standard send and receive functions are *blocking*
- `MPI_Recv` returns only after receive buffer contains requested message
- `MPI_Send` may or may not block until message received (usually blocks)
- May lead to deadlock

MPI :
Integration using trapezoidal rule

Parallelizing – Integration using Trapezoidal Rule



- *Function f over Problem interval* ***[A,B]***
- *Create and Distribute chunks of workload. Each workload consists of its own subinterval of* ***[a,b]*** *which is assigned to each process.*
- *Calculate* ***f*** *for each subinterval*
- *Finally add the* ***f*** *calculated for all the sub intervals to produce result for the complete problem* ***[A,B]***

➢ Issues to consider

- *Load balancing - Number of trapezoids (n) are equally divisible across (p).*

➢ Each process needs the following information

- *Rank of the process*
- *Ability to derive the workload per processor as a function of rank*

Finally any process can do the summation

Parallelizing Trapezoidal Rule (SPMD)

- Algorithm
  Assuming Number of trapezoids *n* is evenly divisible across *p* processors
  - *Calculate:*

$$h = \frac{(b-a)}{n}$$

  - ***Each process calculates its own workload***
    - **local number of trapezoids (local_n) = *n/p***
    - **local starting point (local_a) = a+(process_rank *local_n* h)**
    - **local ending point (local_b) = (local_a + local_n * h)**
  - *Each process calculates its own integral for the local intervals*
    - For each of the local_n trapezoids calculate area and sum the area for local_n trapezoids
  - *If PROCESS_RANK == 0*
    - Receive messages (sub-interval area sum) from all processors
    - ADD all sub-interval areas
  - *If PROCESS_RANK > 0*
    - Send sub-interval area to PROCESS_RANK(0)

.

Basic MPI Calls

- The 6 main MPI calls:
  - *MPI_Init*
  - *MPI_Finalize*
  - *MPI_Comm_size*
  - *MPI_Comm_rank*
  - *MPI_Send*
  - *MPI_Recv*
- Include MPI Header file
  - *#include “mpi.h”*
- Basic MPI Datatypes
  - *MPI_INT, MPI_FLOAT, ….*

Parallel Trapezoidal Rule

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

main(int argc, char** argv) {
    int         my_rank;   /* My process rank           */
    int         p;         /* The number of processes   */
    float       a = 0.0;   /* Left endpoint             */
    float       b = 1.0;   /* Right endpoint            */
    int         n = 1024;  /* Number of trapezoids      */
    float       h;         /* Trapezoid base length     */
    float       local_a;   /* Left endpoint my process  */
    float       local_b;   /* Right endpoint my process */
    int         local_n;   /* Number of trapezoids for my calculation
*/
    float       integral;  /* Integral over my interval */
    float       total;     /* Total integral            */
    int         source;    /* Process sending integral  */
    int         dest = 0;  /* All messages go to 0      */
    int         tag = 0;
    MPI_Status  status;
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco

Parallel Trapezoidal Rule

```
    float Trap(float local_a, float local_b, int local_n, float h);     /* Calculate local
integral  */

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

    /* Get my process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many processes are being used */
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    h = (b-a)/n;     /* h is the same for all processes */
    local_n = n/p;  /* So is the number of trapezoids */

    /* Length of each process' interval of * integration = local_n*h.  So my interval
     * starts at: */

    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
    integral = Trap(local_a, local_b, local_n, h);
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pach

Parallel Trapezoidal Rule

```
 /* Add up the integrals calculated by each process */
    if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,MPI_COMM_WORLD,
&status);
            total = total + integral;
        }
    } else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
    }
    /* Print the result */
    if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n",n);
        printf("of the integral from %f to %f = %f\n",
            a, b, total);
    }
    /* Shut down MPI */
    MPI_Finalize();
} /*  main  */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco

MPI Summary

- MPI Programs are made up of communicating processes
- Each process has its own address space containing its own attributes such as rank, size etc.

- Default communicator is MPI_COMM_WORLD
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it