MPI (Message Passing Interface)

Sources

- Introduction to the Message Passing Interface (MPI) using C
- MPI Foundation Course: 6 Hours!
- Using MPI with C
- https://github.com/Amagnum/Parallel-Dot-Product-of-2-vectors-MPI/blob/main/main.cpp
- https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
- https://github.com/mpitutorial/mpitutorial/blob/gh-pages/tutorials/mpi-scatter-gather-and-allgather/code/avg.c

Overview

- A C / C++ / Fortran library
- Serves to provide an efficient, multi-platform API to program in
- Follows the *Distributed Memory* paradigm
  - No shared variables
  - Processes communicate with each other by passing messages back and forth
- Header file
  - #include<mpi.h>
- Command to compile
  - mpicc program.c -o program
- Command to run (with runtime command line arguments)
  - mpirun -np [*num_processes*] program

Handles

Communicators

- Collection of process threads
- Each thread is assigned a unique ID, called it's *rank*
- In the beginning, all threads are grouped into one communicator
  - MPI_COMM_WORLD

MPI Type Handles

- Similar to regular data types, used for the same purpose when passing messages using MPI
- Common handles include
  - MPI_INT

- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE

Typical Program Structure

```
int main(int argc, char **argv)
{
	// Variables in MPI aren't shared, so each process has its own copy of these
	int process_rank, process_count;

	// Spawn processes, command line arguments used to specify number of processes
	MPI_Init(&argc, &argv);

	// Get rank and process count
	MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &process_count);

	// Check if current process is root (master)
	if(process_rank == 0)
	{
		// Ask for user input
		// Split workload among threads, including itself
		// Perform its task and generate partial output
		// Collect partial outputs
		// Combine partial outputs to yield final output
	}

	// If not, it is a slave process
	else
	{
		// Perform its task and generate partial output
		// Send partial output to root
	}

	MPI_Finalize();
}
```

Functions

General Syntax

error_code = MPI_Xxxx(args…);

Library Initialisation

Must be the first call to the MPI Library, spawns the processes

int MPI_Init(int *argc, char *argv);

Library Finalisation

Must be the last call to the MPI Library, shuts down all threads except the root (rank 0)

int MPI_Finalize();

Get Process Rank

MPI_Comm_rank(MPI_Comm comm, int *rank);

Eg.
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf(“Rank is %d\n“, rank);

Get Communicator Size

MPI_Comm_size(MPI_Comm comm, int *size);

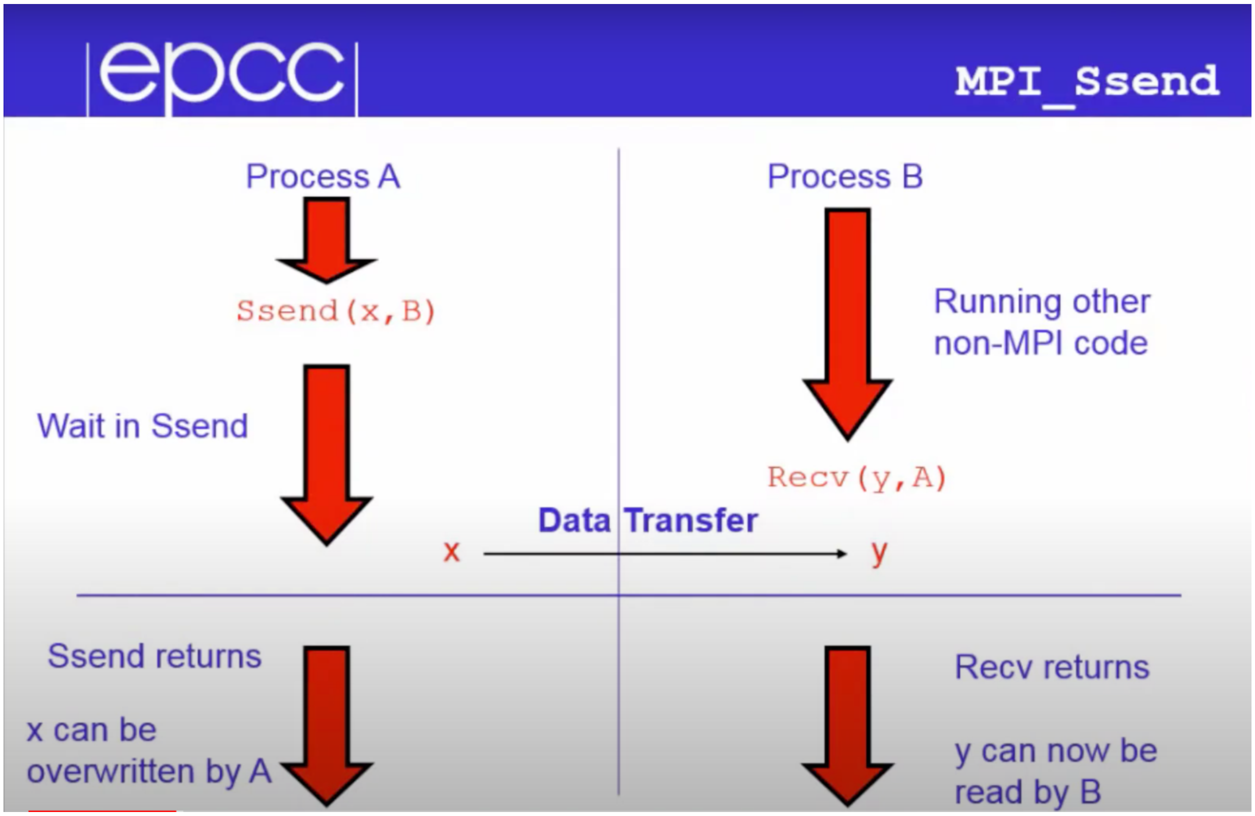
Send and Recv

Send

Ssend

- (Synchronous send)
- Waits until the message is received before proceeding
- Prone to deadlocks, if both processes attempt to send at the same time

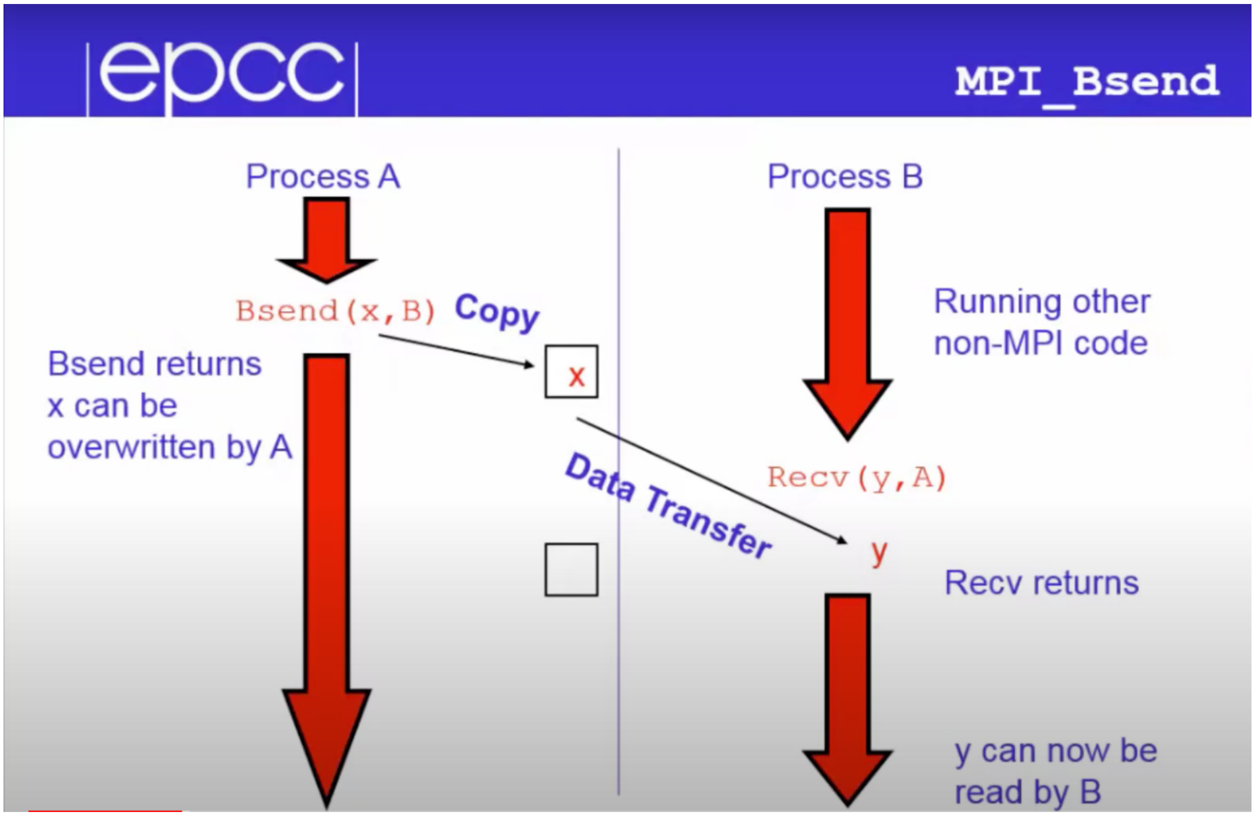
Eg.



Bsend

- (Buffered send)
- Asynchronous in nature
- Stores the message in a buffer, then proceeds with execution
- Sends the message when the receiving process runs *recv*
- Each process has a buffer space allocated to it for this purpose (size of 0 by default)
- Not prone to deadlocks
- Fails if buffer space is exhausted

Eg.



Send

- Defaults to Bsend if buffer space is available
- If not, switches to Ssend
    - Vulnerable to deadlocks while in this state

Syntax

int MPI_Send(...arguments);

The arguments

- (void *) *data_to_send*: address of a C variable that corresponds to *send_type* below
- (int) *send_count*: number of data elements to be sent
- (MPI type handle) *send_type*: datatype of *data_to_send*
- (int) *destination_ID*: rank of the receiving process
- (int) *tag*: message tag
- (MPI_Comm) *comm*: communicator

Recv

Only one function, acts in accordance with the type of send invoked

Syntax

int MPI_Recv(...arguments);

The arguments

- (void *) *received_data*: address of a C variable that corresponds to *receive_type* below
- (int) *receive_count*: number of data elements to be received
- (MPI type handle) *receive_type*: datatype of *received_data*
- (int) *sender_ID*: rank of the sending process
- (int) *tag*: message tag
- (MPI_Comm) *comm*: communicator
- (MPI_Status *) *status*: metadata corresponding to *received_data*

receive_count, sender_ID and *tag* can be used to select which message is to be received

Example

A program to send a number from one process to another

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

int main(int argc, char **argv)
{
	int process_rank, process_count, num;
	MPI_Status status;

	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &process_count);

	if(process_rank == 0)
	{
		printf("Enter the value of num\n");
		scanf("%d", &num);

		MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
	}

	else
	{
		MPI_Recv(&num, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);

		printf("Received %d by process %d", num, process_rank);
	}

	MPI_Finalize();
```

}

Barrier

Used for synchronisation, blocks progression until all processes have reached the barrier

Syntax

MPI_Barrier();

Broadcast, Scatter and Gather

These are collective operations that involve all processes

Broadcast

- One call, all must execute.
- No separate receive function.
- Other than the sender, all processes wait to receive and synchronise

Syntax

int MPI_Bcast(...arguments);

The arguments
- (void *) *data_to_send*: address of a C variable that corresponds to *send_type* below
- (int) *send_count*: number of data elements to be sent
- (MPI type handle) *send_type*: datatype of *data_to_send*
- (int) *sender_ID*: rank of the sending (broadcasting) process
- (MPI_Comm) *comm*: communicator

Scatter

- Split a variable into subsets of values
- Distribute them among processes in the group defined by the communicator
- Works similarly to broadcast

Syntax

int MPI_Scatter(...arguments); *// for a fixed subset size*
int MPI_Scatterv(...arguments); *// for a variable subset size (data not perfectly divisible by the number of processes)*

The arguments
- (void *) *data_to_scatter*: address of a C variable that corresponds to *send_type* below
- (int) *send_count*: number of data elements to send to each process
- (MPI type handle) *send_type*: datatype of *data_to_scatter*
- (void *) *data_to_receive*: subset of *data_to_scatter*
- (int) *receive_count*: size of the subset in *data_to_receive*

- (MPI type handle) *receive_type*: datatype of *data_to_receive*
- (int) *sender_id*: rank of sending process
- (MPI_Comm) *comm*: communicator

Gather

- Companion function to *Scatter*
- Gathers values from all sent variables into a single variable

Syntax

int MPI_Gather(...arguments); *// for a fixed subset size*
int MPI_Gatherv(...arguments); *// for a variable subset size (data not perfectly divisible by the number of processes)*

The arguments

- (void *) *data_to_gather*: address of the C variable to be sent by all processes that corresponds to *send_type* below
- (int) *send_count*: number of data elements to be sent by each process
- (MPI type handle) *send_type*: datatype of *data_to_scatter*
- (void *) *gathering_variable*: variable that holds all values of *data_to_gather*
- (int) *receive_count*: size of the variable to be received from each process
- (MPI type handle) *receive_type*: datatype of *data_to_receive*
- (int) *receiver_id*: rank of receiving process
- (MPI_Comm) *comm*: communicator

Reduce / Allreduce

- Reduce
  - Used to merge partial results by employing an operator
  - Places final result in the root process (rank = receiver_id)
  - Common operators include
    - MPI_SUM
    - MPI_PROD
    - MPI_MAX
    - MPI_MIN
- All_Reduce
  - Combines partial results into a variable held by all participating processes
  - Omit the last but one argument (receiver_id)

Syntax

int MPI_Reduce(...arguments); // for a variable subset size

The arguments

- (void *) *data_to_reduce*: address of a C variable that corresponds to *send_type* below, that contains a partial result to be merged
- (void *) *reduced_result*: address of a C variable that will contain the final result of the merge

- (int) *send_count*: number of data elements to send
- (MPI type handle) *send_type*: datatype of *data_to_reduce*
- (MPI operation handle) *operation_type*: type of operation to be performed
- (int) *receiver_id*: rank of receiving (destination of final result) process, typically the root
- (MPI_Comm) *comm*: communicator

Example

A program to compute the dot product of two vectors

```
#include<stdio.h>
#include<mpi.h>
#define MAX 10000

int main(int argv, char **argc)
{
	int process_rank, process_count, n, s_size, sum, dot_p;
	int a[MAX], b[MAX], a_s[MAX], b_s[MAX];

	MPI_Init(&argv, &argc);

	MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
	MPI_Comm_size(MPI_COMM_WORLD, &process_count);

	// Root process asks user for input vectors
	if(process_rank == 0)
	{
		printf("Enter the length of the vectors\n");
		scanf("%d", &n);

		printf("Enter the elements of the first vector\n");
		for(int i = 0; i < n; i++)
		{
			scanf("%d", &a[i]);
		}

		printf("Enter the elements of the second vector\n");
		for(int i = 0; i < n; i++)
		{
			scanf("%d", &b[i]);
		}

		s_size = n / process_count;
	}
```

```
    // Root process broadcasts length of vectors
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Root process broadcasts size of each subset
    MPI_Bcast(&s_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Root process scatters each vector among all processes
    MPI_Scatter(a, s_size, MPI_INT, a_s, s_size, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Scatter(b, s_size, MPI_INT, b_s, s_size, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process computes its partial sum
    for(int i = 0; i < s_size; i++)
    {
        sum += a_s[i] * b_s[i];
    }

    // Combining the partial sums
    MPI_Reduce(&sum, &dot_p, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if(process_rank == 0)
    {
        printf("The dot product is %d\n", dot_p);
    }

    MPI_Finalize();
}
```

Lab Programs

Trapezoidal Rule

Write an MPI program to compute the area under the curve using trapezoidal rule using MPI_Reduce and MPI_Allreduce.

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

 // Function to be integrated over
double f(double x)
{
    return (1 / (1 + (x * x)));
}

double trapezoidal(double a_p, double b_p, double h)
```

```
{
  // For each interval, i is the lower, and j is the upper bound
  double i, j;
  double y_i, y_j, area, sum_p = 0.0;

  for(i = a_p; i < b_p; i += h)
  {
     j = i + h;

     y_i = f(i);
     y_j = f(j);

     // Area of trapezium defined by this interval
     area = 0.5 * h * (y_i + y_j);

     // Calculate partial sum (sub-integral)
     sum_p += area;
  }

  return sum_p;
}

int main(int argc, char **argv)
{
  double a, b, n, a_p, b_p, n_p, sum, sum_p, h;

  int process_rank, num_processes;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_processes);

  // Only master process receives input from user
  if(process_rank == 0)
  {
     printf("Enter a, b and n\n");
     scanf("%lf", &a);
     scanf("%lf", &b);
     scanf("%lf", &n);

     // Number of intervals assigned to each process
     n_p = n / num_processes;

     // Length of each interval
```

```
    h = (b - a) / n;
  }

  // Master process broadcasts user input to slave processes
  MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&n_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // a = 0, b = 20, n = 10, h = 2, num_processes = 2, n_p = 5

  a_p = a + process_rank * h * n_p;
  b_p = a_p + h * n_p;

  sum_p = trapezoidal(a_p, b_p, h);

  // Combine partial sums
  MPI_Reduce(&sum_p, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  MPI_Allreduce(&sum_p, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  printf("The area under the curve is %lf (Process %d)\n", sum, process_rank);

  MPI_Finalize();
}
```

Sum of Vectors

Write an MPI program to read 2 vectors and print the sum vector using MPI_Scatter and MPI_Gather.

$$
\begin{aligned}
\mathbf{x} + \mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1})
\end{aligned}
$$

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

int main(int argc, char **argv)
{
  int process_rank, num_processes;
  int i, n, n_p;
  int a[100], b[100], sum[100], a_p[100], b_p[100], sum_p[100];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &num_processes);

if(process_rank == 0)
{
   printf("Enter the length of the vectors\n");
   scanf("%d", &n);

   printf("Enter the elements of vector 1\n");
   for(i = 0; i < n; i++)
   {
      scanf("%d", &a[i]);
   }

   printf("Enter the elements of vector 2\n");
   for(i = 0; i < n; i++)
   {
      scanf("%d", &b[i]);
   }

   n_p = n / num_processes;
}

MPI_Bcast(&n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Scatter(a, n_p, MPI_INT, a_p, n_p, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter(b, n_p, MPI_INT, b_p, n_p, MPI_INT, 0, MPI_COMM_WORLD);

for(i = 0; i < n_p; i++)
{
   sum_p[i] = a_p[i] + b_p[i];
}

MPI_Gather(sum_p, n_p, MPI_INT, sum, n_p, MPI_INT, 0, MPI_COMM_WORLD);

if(process_rank == 0)
{
   printf("Resultant vector :-\n");

   for(i = 0; i < n; i++)
   {
      printf("%d ", sum[i]);
   }
```

```
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Get Input

Write a function get_input(int rank, int comm_size, double *a, double *b, int *n) to read 3 values viz. a (double), b (double) and n (int) on process 0 and send it to other processes. Rewrite the same function using MPI_Bcast() method.

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

void get_input(double *a, double *b, int *n, int process_rank, int num_processes, int b_flag)
{
    MPI_Status status;
    double a_val = 0.0, b_val = 0.0;
    int n_val = 0;

    if(process_rank == 0)
    {
        printf("Enter a, b and n\n");
        scanf("%lf", &a_val);
        scanf("%lf", &b_val);
        scanf("%d", &n_val);

        a = &a_val;
        b = &b_val;
        n = &n_val;
    }

    if(b_flag == 0)
    {
        if(process_rank == 0)
        {
            for(int i = 1; i < num_processes; i++)
            {
                MPI_Send(a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
                MPI_Send(b, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
                MPI_Send(n, 1, MPI_INT, i, 2, MPI_COMM_WORLD);
            }
```

```
        }
        else
        {
            MPI_Recv(a, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Recv(b, 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Recv(n, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

            printf("a = %lf (Process - %d)\n", *a, process_rank);
            printf("b = %lf (Process - %d)\n", *b, process_rank);
            printf("n = %d (Process - %d)\n", *n, process_rank);
        }

    }

    else
    {
        MPI_Bcast(a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        MPI_Bcast(b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        MPI_Bcast(n, 1, MPI_INT, 0, MPI_COMM_WORLD);

        if(process_rank != 0)
        {
            printf("a = %lf (Process - %d)\n", *a, process_rank);
            printf("b = %lf (Process - %d)\n", *b, process_rank);
            printf("n = %d (Process - %d)\n", *n, process_rank);
        }
    }

}

int main(int argc, char **argv)
{
    double a, b;
    int n, b_flag;

    int process_rank, num_processes;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);

    if(process_rank == 0)
    {
        printf("Enter 1 to broadcast the values, 0 to not\n");
```

```
        scanf("%d", &b_flag);
    }

    // Broadcasting b_flag alone
    MPI_Bcast(&b_flag, 1, MPI_INT, 0, MPI_COMM_WORLD);

    get_input(&a, &b, &n, process_rank, num_processes, b_flag);

    MPI_Finalize();
}
```