Parallel Processing Lab: MPI

```
MPI_Init(&argc, &argv);
```

Initialize MPI depending on parameter given on cmd

argc [in, optional]
A pointer to the number of arguments for the program. This value can be NULL.

argv
A pointer to the argument list for the program. This value can be NULL.

```
MPI_Finalize();
```

Terminates the calling MPI process's execution environment.

```
MPI_Comm_rank(
MPI_COMM_WORLD, &myrank);
```

update value `myrank` to match MPI this process rank

myrank [out]
On return, a pointer to the identifier of the calling process within the group of the communicator.

```
MPI_Comm_size(
MPI_COMM_WORLD, &numprocs);
```

update value `numprocs` to match MPI number of process

size [out]
On return, indicates the number of processes in the group for the communicator.

```
MPI_Get_processor_name(
processor_name, &namelen);
```

resultlen [out]
Length (in characters) of the name.

name
A unique specifier for the actual (as opposed to virtual) node. This must be an array of size at least **MPI_MAX_PROCESSOR_NAME**.

```
mpicc hello.c -o hello
mpirun -np 4 ./hello
```

Parallel Processing Lab: Lab 2

Why do the numbers in the sample hello.c come out of order?

MPI codes are not guaranteed to complete in any specific order.

MPI codes allow all the process to be executed at the same time. You cannot make MPI processes *launch* in a certain order, but if you need *a certain part* of your application to run in the order of rank number, you can use a barrier.

Name the two commands to initialize and finalize a MPI program.

```
MPI_Init(&argc, &argv);
```

```
MPI_Finalize();
```

Describe the function of MPI_Comm_rank.

```
MPI_Comm_rank(
MPI_COMM_WORLD, &myrank);
```

update value `myrank` to match MPI this process rank

rank [out]
On return, a pointer to the identifier of the calling process within the group of the communicator.

Explain the function of MPI_Comm_World.

Constants	
MPI_COMM_NULL	The value used for invalid communicator handles.
MPI_COMM_WORLD	An initial intra-communicator of all processes.
MPI_COMM_SELF	Communicator that contains only the calling process.

Parallel Processing Lab: MPI

Performs a standard mode send operation and returns when the send buffer can be safely reused.

```
MPI_Send(&buf, BUFSIZE,
    MPI_CHAR, 0, 19,
    MPI_COMM_WORLD);
```

buf [in, optional]
A pointer to the buffer that contains the data to be sent.

count
The number of elements in the buffer. If the data part of the message is empty, set the *count* parameter to 0.

datatype
The data type of the elements in the buffer.

send to process 0, message tag 19, value of &buf slicing into 0-BUFSIZE, type char

dest
The rank of the destination process within the communicator that is specified by the *comm* parameter.

tag
The message tag that can be used to distinguish different types of messages.

comm
The handle to the communicator.

Performs a receive operation and does not return until a matching message is received.

```
MPI_Recv(&buf, BUFSIZE,
MPI_CHAR, MPI_ANY_SOURCE,
    19, MPI_COMM_WORLD);
```

buf [in, optional]
A pointer to the buffer that contains the data to be sent.

count
The number of elements in the buffer. If the data part of the message is empty, set the *count* parameter to 0.

datatype
The data type of the elements in the buffer.

listen to MPI COMM and wait until any message with tag 19 to be update into buf

source
The rank of the sending process within the specified communicator. Specify the **MPI_ANY_SOURCE** constant to specify that any source is acceptable.

tag
The message tag that is used to distinguish different types of messages. Specify the **MPI_ANY_TAG** constant to indicate that any tag is acceptable.

comm
The handle to the communicator.

Broadcasts data from one member of a group to all members of the group

```
MPI_Bcast(&buf, BUFSIZE,
    MPI_INT, 0,
    MPI_COMM_WORLD);
```

buffer [in, out]
The pointer to the data buffer. On the process that is specified by the *root* parameter, the buffer contains the data to be broadcast. On all other processes in the communicator that is specified by the *comm* parameter, the buffer receives the data broadcast by the root process.

```
process 0 -> MPI_Send(&buf,
BUFSIZE, MPI_INT, 0, xx,
    MPI_COMM_WORLD);
```

count [in]
The number of data elements in the buffer. If the *count* parameter is zero, the data part of the message is empty.

datatype [in]
The MPI data type of the elements in the send buffer.

```
other process ->
MPI_Recv(&buf, BUFSIZE,
    MPI_INT, 0, xx,
    MPI_COMM_WORLD);
```

root [in]
The rank of the process that is sending the data.

comm [in]
The MPI_Comm communicator handle.

Parallel Processing Lab: Example

Simplified

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

int main(int argc, char **argv) {
    int myrank, numprocs;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    // ANY CODE RUN HERE WILL BE
    // EXECUTE IN ALL PROCESSOR

    printf(myrank); // print 0-n {n: numprocs}

    if (myrank == 0) {
        // ANY CODE RUN IN HERE WOULD BE
        // EXECUTED ONLY ON PROCESSOR 0

        printf(myrank); // print 0
    }

    MPI_Finalize();
}
```

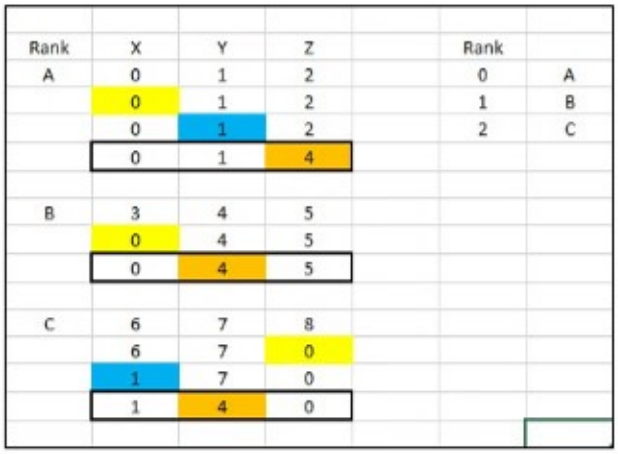
Full

```
// function definition [ basic c ]
int f(int nPartition) {
    return 0;
};

// Example taken from lab5q3
// Make sure to write the prototype correctly

double serialCompute(int nPartition) {
    double sum=0;

    // overall different is this line
    // so we calculate each and every step
    // from 0 till number of partition reach
    for(int i=0; i<nPartition; i++) {
        sum += f((a+i*w)+middle);
    }
    return sum * w;
}

double parallelCompute(int nPartition) {
    double sum=0;

    // overall different is this line
    // we start from current rank and
    // skips by numprocs
    // say like total 2 rank and 10k partition
    // 0, 2, 4, 6, 8, ....
    // 1, 3, 5, 7, 9, ....
    for(int i=my_rank; i<nPartition; i+=numprocs) {
        sum += f((a+i*w)+middle);
    }
    return sum * w;
}
```

```
// Import Library as usual
#include <stdio.h>    // Standard I/O
#include <mpi.h>    // To Use MPI

// function prototype [ basic c ]
int f(int);

int main(int argc, char **argv) {
    int myrank, numprocs;

    // variable declaration
    int nPartition;
    double result, localResult, start, end;
    double btime;

    // For Initialization
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    if (!my_rank) {
        // DO SOMETHING SERIAL HERE

        // Example Get Input from User
        printf("Enter Partitions: ");
        fflush(stdout);
        scanf("%d", &nPartition);

        // Example serial calculation
        double singleResult = serialCompute(nPartition);
        printf("Result is %1.20f\n", singleResult);

        // Example Serial timing
        printf("Serial time: %1.20f\n",
                MPI_Wtime() - btime);

        // Example Time before parallel executed
        btime = MPI_Wtime();
    }

    // DO SOMETHING PARALLEL HERE

    // Example Broadcast
    MPI_Bcast(&nPartition, 1,
              MPI_INT, 0,
              MPI_COMM_WORLD);

    // Example Local Computation
    localResult = parallelCompute(nPartition);

    // Example Total Computation
    MPI_Reduce(&localResult,
               &result, 1, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);

    if (!my_rank) {
        // DO SOMETHING SERIAL HERE

        // Example Display time taken
        printf("Parallel time: %1.20f\n",
                MPI_Wtime() - btime);
        printf("\n");
    }

    // To Make it Work
    MPI_Finalize();
}
```

Parallel Processing Lab: Lab 4

```
int x, y, z;
switch (my_rank){
    case 0: x=0; y=1; z=2;
    MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);
    MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD);
    break;
    case 1: x=3; y=4; z=5;
    MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
    break;
    case 2: x=6; y=7; z=8;
    MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD, &status);
    MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
    break;
}
printf("%d: x=%d y=%d z=%d\n", rank, x, y, z);
```



Rank	X	Y	Z	Rank	
A	0	1	2	0	A
	0	1	2	1	B
	0	1	2	2	C
	0	1	4		
B	3	4	5		
	0	4	5		
	0	4	5		
C	6	7	8		
	6	7	0		
	1	7	0		
	1	4	0		

Processor 0

`MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);`
Send broadcast of x value to all and wait for all to receive

`MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);`
Send value of y to processor 2 with tag 43 and wait for receive

`MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD);`
Wait broadcast from process 1 and store it in z

Processor 1

`MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);`
Wait broadcast from process 0 and store it in x

`MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);`
Send broadcast of y value to all and wait for all to receive

Processor 2

`MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);`
Wait broadcast from process 0 and store it in z

`MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD, &status);`
Receive tag 43 from processor 0 and store it in x

`MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);`
Wait broadcast from process 1 and store it in y

Serial

```
double serialCompute(int nPartition) {
    double step = 1.0/nPartition;
    double x, total=0;
    int i;

    // overall different is this line
    // so we calculate each and every step
    // from 0 till number of partition reach
    for (i=0; i<nPartition; i++) {
        x=(step/2) + step*i;
        total = total + f(x);
    }
    return total*step;
}
```

Parallel

```
double parallelCompute(int nPartition) {
    double step = 1.0/nPartition;
    double x, total=0;
    int i;

    // overall different is this line
    // we start from current rank and skips by numprocs
    // say like total 2 rank and 10k partition
    // 0, 2, 4, 6, 8, ....
    // 1, 3, 5, 7, 9, ....
    for (i=my_rank; i<nPartition; i+=numprocs) {
        x=(step/2) + step*i;
        total = total + f(x);
    }
    return total*step;
}
```

`for (i=0; i<nPartition; i++)`
Compute everything from 0 to n

`for (i=my_rank; i<nPartition; i+=numprocs)`
Compute everything from 0 to n of each processor. It skips the job of other processor.

Parallel Processing Lab: Lab 5

Determine the route taken in a 6 dimensional Hypercube network from node 8 to node 53

	32	16	8	4	2	1
8	0	0	1	0	0	0
53	1	1	0	1	0	1
(XOR) r =	1	1	1	1	0	1

Step 1: Convert Decimal To Binary

Step 2: Do XOR with both of the number and make it as r

r: indicator to flip bit [if it 1, flip]

	32	16	8	4	2	1
8	0	0	1	0	0	0
8+32 = 40	1	0	1	0	0	0
40+16 = 56	1	1	1	0	0	0
56-8 = 48	1	1	0	0	0	0
48+4 = 52	1	1	0	1	0	0
52	1	1	0	1	0	0
52+1 = 53	1	1	0	1	0	1

Step 3: Start from 8; Left to Right, flip the required bit

Part 3: Calculating something else than Pi

```
#include<stdio.h>
#include<math.h>
float f(float x){
    return (sin (x/2) +1);
}
int main(){
    double a,b,w,sum=0,middle;
    int i,n;
    printf("Enter the number of intervals : ");
    scanf("%d",&n);
    a=M_PI/2;
    b=2*M_PI;
    //step length (width of each slice)
    w=(b-a)/n;
    middle = w/2;
    for(i=0; i<n; i++)
    { sum += f((a+i*w)+middle);
    }
    sum = sum * w;
    printf("Answer : %f \n",sum);
    return 0;
}
```

```
// locally compute the pi value for respective
    localResult = parallelCompute(nPartition, a,
w, middle);

    // merging all the data
    // one could say &result is global result
that sum all the &localResult
    MPI_Reduce(&localResult, &result, 1,
MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // uncomment this if you want to see each
local result
    /* printf("Rank %d: Pi is %1.20f\n", my_rank,
localResult);  */

    // !my_rank also mean my_rank == 0
    if (!my_rank) {
        printf("\n");
        printf("Result is %1.20f\n", result);
        printf("Parallel time: %1.20f\n",
MPI_Wtime() - btime);
        printf("\n");
    }
```

```
double parallelCompute(int nPartition, double a,
double w, double middle) {
    double sum=0;

    // overall different is this line
    // we start from current rank and skips by
numprocs
    // say like total 2 rank and 10k partition
    // 0, 2, 4, 6, 8, ....
    // 1, 3, 5, 7, 9, ....
    for(int i=my_rank; i<nPartition; i+=numprocs)
    {
        sum += f((a+i*w)+middle);
    }
    return sum * w;
}
```