# Trabajo Práctico N° 1

**Fecha Límite de Entrega:** 31 de marzo
**Profesor:** Lic. Demian Barry
**Auxiliar:** Ing. Fernando Pap

### Condiciones de Aprobación:

En líneas generales el trabajo debe dar evidencia del desarrollo realizado. En casos puntuales en los que sea conveniente, incluir un archivo `readme.txt` con notas correspondientes. Entregas individuales subiendo contenido al classroom. Se recomienda utilizar github/Bitbucket/etc.
El trabajo debe ser entregado completo.

### Enunciados

## Parallel programming (MPI)

1. Requerimientos:
   - Instalar MPI (https://www.mpich.org – https://mpitutorial.com/tutorials/installing-mpich2).
   - Alternativamente, mediante Docker:

```
docker pull nlknguyen/alpine-mpich

docker run --rm -it -v $(pwd):/project nlknguyen/alpine-mpich
```

2. Compilar y ejecutar los programas del anexo.

   Compilación: mpicc <fuente> -o <ejecutable>
   Ejemplo: mpicc hello_world.c -o hello_world

   Ejecución: mpirun -np <nro de procesos> <ejecutable>
   Ejemplo: mpirun –np 4 ./hello_world

3. Realizar un programa que, dado un vector y un escalar, resuelva su producto. Se debe distribuir los elementos del vector a los procesos disponibles para realizar los productos de cada elemento del vector por el escalar. Mostrar el vector original y el vector resultante.

   Pueden utilizar como referencia la implementación del producto de una matriz por un vector:
   https://people.sc.fsu.edu/~jburkardt/c_src/mpi_test/matvec_mpi.c (¡Cuidado con el bug!).
   En https://people.sc.fsu.edu/~jburkardt/c_src/mpi_test/mpi_test.html se encuentran múltiples ejemplos.

# ANEXO

## Hello World

```
#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("Hola! Soy el proceso %d de %d\n", rank, size);

  MPI_Finalize();

  return 0;
}
```

**int MPI_Init(int *argc, char ***argv)**
**Entrada**
argc Puntero al número de argumentos
argv Puntero al vector de argumentos

**MPI_Comm**
tipo de dato que guarda toda la información relevante sobre un comunicador específico.

**int MPI_Comm_rank(MPI_Comm comm, int *rank)**
**Entrada**
comm Comunicador sobre el que se quiere conocer el identificador.
**Salida**
rank Entero que indica el rango del proceso.

**MPI_COMM_WORLD**
Identificador del comunicador al que pertenecen todos los procesos de una ejecución MPI.

**int MPI_Comm_size (MPI_Comm comm, int *size)**
**Entrada**
comm Comunicador sobre el que se quiere conocer el tamaño.
**Salida**
sizeTamaño del comunicador.

**int MPI_Finalize()**

**Send receive**

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

int main (int argc, char *argv[])
{
  int size, rank, dest, source, tag=1;
  char inmsg, outmsg='x';
  MPI_Status Stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    dest = 1;
    source = 1;
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    printf("Soy el proceso %d de %d y recibi %c\n", rank, size, inmsg);
  } else if (rank == 1) {
    dest = 0;
    source = 0;
    MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
    MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    printf("Soy el proceso %d de %d y recibi %c\n", rank, size, inmsg);
  }
  MPI_Finalize();
}
```

**int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
**Entrada**
buf Dirección inicial del buffer de envío. Esto significa que requiere un puntero. Si solo se pretende enviar un elemento, se puede enviar el puntero a este (&elemento).
count Numero de elementos a enviar (Debe ser un entero no negativo).
datatype Tipo de dato de cada elemento que se va a enviar. Acepta constantes definidas por MPI, por ejemplo MPI_INT.
dest Rango del proceso destino.
tag Entero que representa la etiqueta del mensaje. El significado de la etiqueta queda en manos del usuario, durante el proceso de envío no es modificado.
comm Comunicador utilizado para la comunicación

**int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**
**Entrada**
count Entero que indica el número máximo de elementos que se espera recibir en el buffer de entrada.
datatype Tipo de dato de cada elemento que se va a recibir. Acepta constantes definidas por MPI, por ejemplo MPI_INT.
source Rango del proceso de origen esperado, solo se recogen mensajes cuyo origen sea el especificado. Se acepta el valor MPI_ANY_SOURCE, el cual recoge de cualquier proceso origen.
tag Entero que representa la etiqueta del mensaje. Solo se recogerá un mensaje con la etiqueta especificada. Se acepta el valor MPI_ANY_TAG, que recoge con cualquier etiqueta. El significado de la etiqueta queda en manos del usuario, durante el proceso de envío no es modificado.
comm Comunicador utilizado para la comunicación. Solo se recogerán mensajes que han sido enviados por el comunicador seleccionado.
**Salida**
buf Buffer de entrada en el que se guarda el contenido del mensaje enviado. Recibe un puntero al comienzo del buffer.
status Objeto de tipo MPI_Status, contiene datos relevantes sobre el mensaje (como son el origen (MPI_SOURCE), la etiqueta (MPI_TAG) y el tamaño (size)).

**Hello World Send Receive**

```
/******************************************************************************
* FILE: mpi_helloBsend.c
* DESCRIPTION:
*   MPI tutorial example code: Simple hello world program that uses blocking
*   send/receive routines.
* AUTHOR: Blaise Barney
******************************************************************************/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER           0

int main (int argc, char *argv[])
{
  int  numtasks, taskid, len, partner, message;
  char hostname[MPI_MAX_PROCESSOR_NAME];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  /* need an even number of tasks  */
  if (numtasks % 2 != 0) {
    if (taskid == MASTER) {
     printf("Quitting. Need an even number of tasks: numtasks=%d\n", numtasks);
    }
  } else {
    if (taskid == MASTER)
      printf("MASTER: Number of MPI tasks is: %d\n",numtasks);

    MPI_Get_processor_name(hostname, &len);
    printf ("Hello from task %d on %s!\n", taskid, hostname);

    /* determine partner and then send/receive with partner */
    if (taskid < numtasks/2) {
      partner = numtasks/2 + taskid;
      MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
      MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
    } else if (taskid >= numtasks/2) {
      partner = taskid - numtasks/2;
      MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
      MPI_Send(&taskid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
    }
    /* print partner info and exit*/
    printf("Task %d is partner with %d\n",taskid,message);
  }
  MPI_Finalize();
}
```

**int MPI_Get_processor_name(char *name, int *resultlen)**
**Salida**

| | |
|---|---|
| name | Especificador único del nodo real. |
| resultlen | Longitud (en caracteres) del resultado retornado en name. |