Introducción al Cómputo en Paralelo en el Laboratorio Nacional de Supercómputo del Sureste de México: Uso de MPI

Poulette Mayoral Orueta y Luis M. Villaseñor Cendejas
Benemérita Universidad Autónoma de Puebla, Puebla, Mexico
E-mail: lvillasen@gmail.com

9 de Noviembre de 2016

1 Introducción

Una modelo popular para hacer cómputo en paralelo en un clúster consiste en usar MPI [1] (del inglés Message Passing Interfase). A diferencia de otros modelos como OpenMP [2], MPI tiene la ventaja de que se aplica por igual a sistemas de memoria compartida, es decir procesos que corren en paralelo dentro de un mismo nodo, que a sistemas de memoria distribuida, es decir procesos que corren en paralelo en varios nodos dentro de un clúster o en varias computadoras dentro de una red.

En un clúster de varios nodos, o en uno de varias computadoras dentro de una red, uno puede optar por hacer cómputo paralelo basado únicamente en MPI o mediante un modelo mixto OpenMP/MP. Ambos modelos tiene ventajas y desventajas que se pueden consultar por ejemplo en [3].

En este tutorial vamos a describir, desde un punto de vista práctico, el uso de MPI en el clúster del Laboratorio Nacional de Supercómputo del Sureste de México (LNS). Ilustrando con ejemplos el uso de las principales funciones de MPI en cuatro lenguajes: C, Fortran, Julia y Python, de modo que cualquier usuario puede correr estos ejemplos siguiendo las instrucciones detalladas que se dan a continuación y de ese modo adquirir en poco tiempo experiencia que le puede permitir aplicar MPI a sus programas de cómputo secuenciales para obtener un mayor beneficio del LNS.

MPI consta de alrededor de 125 funciones, sin embargo basta con manejar el uso de unas pocas para resolver la mayoría de los problemas de paralelización. Por esta razón, en este tutorial ilustramos con ejemplos el uso de las funciones más usadas que son las siguientes:

- MPI_INIT I
- MPI_COMM_RANK
- MPI_COMM_SIZE
- MPI_GET_PROCESSOR_NAME
- MPI_Reduce
- MPI_Send
- MPI_Recv
- MPI_Barrier
- MPI_FINALIZE

El comando básico de MPI para correr un programa en paralelo a través de N procesos es

```
mpirun -np N programa-a-ejecutar
```

Es importante notar que N puede ser cualquier número, ya que varios procesos se pueden correr en paralelo aún en un único procesador. Por otro lado, los procesadores modernos consisten de varios núcleos. Para obtener el número de núcleos de una computadora, o de un nodo en un clúster, se puede usar el comando de Linux

Listado 1: Ejemplo de un archivo de comandos básico para usar MPI con SLURM.

```
#!/bin/bash
#SBATCH --job-name mpi-simple
#SBATCH --nodes 3
#SBATCH --ntasks-per-node 24
#SBATCH --time 00:10:00
#SBATCH --output mpi-simple.out
module load tools/intel/impi/5.0.2.044
mpirun mpi-simple
```

```
nproc
```

Por ejemplo si en el clúster del LNS nos conectamos al nodo *master*1 con *ssh* y ahí tecleamos `nproc` obtenemos 32, mientras que si nos conectamos al nodo `cn0103-2` vemos que tiene 24 núcleos.

Para obtener el máximo rendimiento de cómputo en paralelo de una computadora, o de un nodo, hay que usar el comando `mpirun` con N igual al número de núcleos. En el caso de tener acceso a un clúster, como el del LNS, podemos correr un programa en los procesadores de varios nodos, para tal fin basta con usar la opción `machinefile` a través del comando

```
mpirun -np N -machinefile lista-de-nodos programa-a-ejecutar
```

donde el archivo de texto `lista-de-nodos` contiene los nombres de los nodos en cuestión. El sistema operativo se encarga, a través de la implementación de MPI, de distribuir la ejecución de los programas en forma óptima entre los núcleos que contengan los nodos listados en el archivo `lista-de-nodos`.

Por ejemplo para correr un programa en los 72 núcleos de los nodos cn0103-1.lns.buap.mx, cn0103-2.lns.buap.mx y cn0103-3.lns.buap.mx hacemos lo siguiente:

```
cat > nodos
cn0103-2
cn0103-3
cn0103-4
crtl c
```

para crear el archivo `nodos` y posteriormente ejecutamos el comando

```
mpirun -np 72 -machinefile nodos programa-a-ejecutar
```

En la Sección 2.1 mostramos un ejemplo concreto.

El comando `mpirun` se puede usar en el LNS solo para pruebas rápidas, para correr un programa que tarde más de unos segundos se requiere usar el manejador de cargas de trabajos SLURM [4, 5]. En el Listado 1 se ilustra un archivo de comandos básico para el uso de MPI con SLURM. En la Subsección 2.1 mostramos en detalle cómo ejecutar este archivo que se llama Ejemplo-SLURM.sh y su resultado.

Finalmente, cuando se usa MPI es muy importante balancear las cargas de los procesos que corren en paralelo. Esto se debe a que el tiempo de ejecución está limitado por el tiempo que tarda en correr el proceso que tiene mayor carga de cómputo.

Todos los programas que se listan en este tutorial se pueden descargar del repositorio github [6] usando el comando siguiente

```
git clone https://github.com/lvillasen/TutorialMPI
```

Listado 2: Listado del programa `mpi-simple.c` en C

```
#include <stdio.h>
 #include <mpi.h>
 main(int argc, char **argv)
 {
    int ierr, rank, size;
  char hostname[30];
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &size);
    gethostname(hostname,30);
   int a= rank;
    printf("a=%i en  proceso %i de un total de %i en %s\n",
       a,rank, size,hostname);
    ierr = MPI_Finalize();
 }
```

2 Uso de MPI en C en el LNS

Como prerrequisito para usar MPI con C en el LNS debemos cargar los módulos siguientes:

```
module load tools/intel/impi/5.0.2.044
module load compilers/intel/parallel_studio_xe_2015/15.0.1
```

Es importante no tener cargado ningún otro módulo de mpi que pueda interferir con estas librerías. Para ver qué módulos están disponibles se usa el comando

```
module avail
```

y para ver qué módulos están cargados se usa el comando

```
module list
```

si es necesario, un módulo que esté cargado, por ejemplo el `tools/openmpi/intel/1.8.4`, se puede remover con el comando

```
module unload tools/openmpi/intel/1.8.4
```

Para que los siguientes ejemplos en C compilen y se ejecuten exitosamente es necesario que el resultado del comando

```
module list
```

incluya en el listado de módulos activos al menos los siguientes:

```
Currently Loaded Modulefiles:
  1) compilers/intel/parallel_studio_xe_2015/15.0.1
  2) tools/intel/impi/5.0.2.044
```

2.1 Ejemplo básico de uso de MPI en C en el LNS

El primer ejemplo se llama mpi-simple.c y se muestra en el Listado 2.

En este ejemplo se ilustra el uso de las funciones siguientes:

- int MPI_Init(&argc, &argv)

- int MPI_Comm_rank(MPI_COMM_WORLD, &rank)
- int MPI_Comm_size(MPI_COMM_WORLD, &size)
- int MPI_Finalize()

La primera función inicia MPI, la segunda obtiene el número de proceso y lo guarda en la variable *rank*, la tercera obtiene el número de procesos que se corren simultáneamente y lo guarda en la variable *size*. La cuarta función finaliza MPI. Todas las funciones arrojan como reultado un entero que vale 1 si la ejecución de la función se realizó con éxito.

Este código se compila con el comando

```
mpicc mpi-simple.c -o mpi-simple
```

y se corre con el comando siguiente:

```
mpirun -n 4 mpi-simple
```

Este código produce como resultado lo siguiente:

```
a=1 en  proceso 1 de un total de 4
a=2 en  proceso 2 de un total de 4
a=3 en  proceso 3 de un total de 4
a=0 en  proceso 0 de un total de 4
```

A manera de ejemplo del poder de MPI para trabajar en sistemas de memoria distribuida, este mismo programa lo podemos correr en paralelo en los 72 núcleos de los nodos `cn0103-2, cn0103-3 y cn0103-4` que previamente guardamos en el archivo `nodos`. Vemos que en efecto el comando

```
cat nodos
```

nos arroja

```
cn0103-2
cn0103-3
cn0103-4
```

Enseguida ejecutamos este programa con el comando

```
mpirun -np 72 -machinefile nodos  mpi-simple
```

para producir las 72 líneas siguientes:

```
   a=11 en proceso 11 de 72 en cn0103-4.lns.buap.mx
   a=14 en proceso 14 de 72 en cn0103-4.lns.buap.mx
.....
   a=36 en proceso 36 de 72 en cn0103-2.lns.buap.mx
   a= 0 en proceso  0 de 72 en cn0103-2.lns.buap.mx
```

que indican que en efecto el programa se ejecutó simultáneamente a través de 72 procesos distribuidos uniformemente en los 3 nodos indicados.

Este mismo resultado lo podemos obtener ejecutando `mpirun` a través de SLURM. Para esto ejecutamos el comando

```
sbatch Ejemplo-SLURM.sh
```

donde el archivo Ejemplo-SLURM.sh se muestra en el Listado 1. Para ver el estado de este trabajo usamos el comando

```
squeue
```

Después de ejecutarse el resultado se muestra en el archivo `mpi-simple.out` de modo que si hacemos

```
cat mpi-simple.out
```

obtenemos

```
a=48 en proceso 48 de 72 en cn0211-2.lns.buap.mx
a=24 en proceso 24 de 72 en cn0211-1.lns.buap.mx
....
a=16 en proceso 16 de 72 en cn0210-4.lns.buap.mx
a=60 en proceso 60 de 72 en cn0211-2.lns.buap.mx
```

es decir que obtenemos el mismo resultado que obtuvimos previamente cuando corrimos `mpirun` en forma directa.

2.2 Ejemplo de uso de la función *reduce* de MPI en C en el LNS

En el siguiente ejemplo ilustramos el uso de la función *reduce* que está definida en C por

```
int MPI_Reduce(
    void* send_data,
    void* recv_data,
    int cuenta,
    MPI_Datatype tipo,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

donde *cuenta* es el número de variables del tipo $MPI\_Datatype$ que se van a sumar. La variable `MPI_Datatype` indica el tipo de datos cuya correspondencia con el tipo de datos en C se muestra en la Tabla 1. La explicación de lo qué significan las demás variables se da a través de un ejemplo del uso de esta función.

Tipo en MPI	Tipo en C	Número de Bits
MPI_CHAR	carácter	8
MPI_SHORT	entero corto con signo	8
MPI_INT	entero	16
MPI_LONG	entero largo	32
MPI_UNSIGNED_CHAR	carácter sin signo	8
MPI_UNSIGNED_SHORT	entero corto sin signo	8
MPI_UNSIGNED	entero sin signo	16
MPI_UNSIGNED_LONG	entero largo sin signo	32
MPI_FLOAT	punto flotante	32
MPI_DOUBLE	punto flotante doble	64
MPI_LONG_DOUBLE	punto flotante largo doble	128
MPI_BYTE	carácter sin signo	8
MPI_PACKED	(ninguno)	

Tabla 1: Correspondencia entre tipos de variables entre MPI y C.

En este ejemplo se ilustra la manera de sumar tanto números escalares como arreglos de números. El código respectivo de llama `mpi-reduce.c`, y se muestra en el Listado 3.

Listado 3: Listado del programa `mpi-reduce.c` en C

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv){
int size, rank, ierr;
int a, a_sum,i;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
a=rank;
int A[3] = { rank,rank,rank};
int A_sum[3] = { 0,0,0};
  printf("En proceso %i a = %i \n",rank,a);
     MPI_Reduce(&a, &a_sum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
     MPI_Reduce(&A, &A_sum,3,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
if( rank == 0 ){
  printf("La suma de a es = %i \n",a_sum);
  printf("La suma de A es [");
for( i = 0; i < 3; i++) {
        printf("%d ", A_sum[i]);
    }
printf("]\n");}
 ierr = MPI_Finalize();}
```

En el caso del arreglo A hacemos de suma de 3 elementos del arreglo. El destino de la suma de a y del arreglo A se escoge como el proceso 0.

Este código se compila con el comando

```
mpicc mpi-reduce.c -o mpi-reduce
```

Una vez compilado se corre con el comando siguiente:

```
mpirun -n 4  mpi-reduce
```

y produce como resultado lo siguiente:

```
En proceso 2 a = 2
En proceso 3 a = 3
En proceso 0 a = 0
En proceso 1 a = 1
Suma de a de todos los procesos = 6
Suma de A de todos los procesos = [6 6 6 ]
```

2.3 Ejemplo de uso de las funciones *send* y *receive* de MPI en C en el LNS

La función *send* está definida en C por

```
int MPI_Send(
  void *data,
  int cuenta,
```

```
  MPI_Datatype tipo,
  int dest,
  int etiqueta,
  MPI_Comm comm
);
```

mientras que la función *receive* está definida por

```
MPI_Recv(
    void* data,
    int cuenta,
    MPI_Datatype tipo,
    int source,
    int etiqueta,
    MPI_Comm communicator,
    MPI_Status* status)
```

A continuación ilustramos el uso de las funciones *send* y *receive* en C con un ejemplo . El código respectivo se llama `mpi-send-rcv.c` y se muestra en el Listado 4.

Este código se compila con el comando

```
mpicc mpi-send-rcv.c -o mpi-send-rcv
```

Una vez compilado se corre con el comando siguiente:

```
mpirun -n 4  mpi-send-rcv
```

y produce como resultado lo siguiente:

```
Datos recibidos en proceso 1 con etiqueta 11:
 11 11 11
Datos recibidos en proceso 1 con etiqueta 12:
 12 12 12
```

2.4 Ejemplo de paralelización de un programa en C

A manera de ejemplo ilustrativo del uso de MPI usamos el método de integración Monte Carlo para calcular el valor de π. El programa mostrado en el Listado 5 genera pares de números aleatorios entre 0 y 1 y cuenta el número de puntos que caen dentro de un círculo de radio unitario en el primer cuadrante. Dado los números aleatorios son uniformes en todo el primer cuadrante, la fracción de los pares que están dentro del cículo es proporcional a el área del círculo unitario que se intercepta con el primer cuadrante, es decir, esta fracción es proporcional a $\pi/4$.

Este programa se compila con el comando

```
gcc pi.c -o pi -lm
```

y al correrlo con el comando

```
pi
```

produce como resultado lo siguiente:

```
Numero de puntos = 100000000
Numero de puntos dentro del circulo = 78542448 +- 8862.42
Pi = 4 N_dentro/N_total= 3.1417 +- 0.000354497
```

Listado 4: Listado del programa `mpi-send-rcv.c` en C

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv){
int size, rank, rc;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Status Stat;
int i,a, a_sum;
if( rank == 0 ){
int datos11[3] = { 11,11,11};
int datos12[3] = { 12,12,12};
rc = MPI_Send(&datos11, 3, MPI_INT, 1, 11, MPI_COMM_WORLD);
rc = MPI_Send(&datos12, 3, MPI_INT, 1, 12, MPI_COMM_WORLD);
}
if( rank == 1 ){
int datos11_r[3] = { 0,0,0};
int datos12_r[3] = { 0,0,0};
rc = MPI_Recv(&datos11_r, 3, MPI_INT, 0, 11, MPI_COMM_WORLD,&Stat );
rc = MPI_Recv(&datos12_r, 3, MPI_INT, 0, 12, MPI_COMM_WORLD,&Stat);
   printf("Datos recibidos en proceso %i con etiqueta 11:\n", rank);
for(i = 0; i < 3; i++) {
   printf("%d ", datos11_r[i]);
    }
   printf("\nDatos recibidos en proceso %i con etiqueta 12:\n", rank);
for(i = 0; i < 3; i++) {
   printf("%d ", datos12_r[i]);
    }
}
rc = MPI_Finalize();}
```

Este problema es del tipo conocido como "trivialmente paralelizable" ya que para acelerarlo basta usar MPI para correr varios procesos iguales en diferentes núcleos, y hacer la suma mediante la función *reduce*.

Ejercicio. Modificar el programa del Listado 5 para paralelizar el cálculo de π con MPI y compruebar su correcto funcionamiento.

3 Uso de MPI en Fortran en el LNS

Como prerrequisito para usar MPI con Fortran en el LNS debemos cargar los módulos siguientes:

```
module load compilers/intel/parallel_studio_xe_2015/15.0.1
module tools/intel/impi/5.0.2.044
```

Es importante no tener cargado ningún otro módulo de openmp que pueda interferir con estas librerías, por ejemplo asegurarse que no esté cargado el módulo `tools/openmpi/intel/1.8.4`.

Listado 5: Listado del programa secuencial que calcula π en C.

```
/* Programa para calcular Pi usando el metodo Monte Carlo */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
int main()
{
   int N=100000000; int i,count=0;
   double x,y,z,pi;
   srand(1);
   for ( i=0; i<N; i++) {
      x = (double)rand()/RAND_MAX;
      y = (double)rand()/RAND_MAX;
      z = x*x+y*y;
      if (z<=1) count++;
      }
   printf ("Numero de puntos = %d \n",N);
   printf ("Numero de puntos dentro del circulo \
  = %d +- %g \n",count,sqrt(count));
   printf ("Pi = 4 N_dentro/N_total= %g +- %g \n"\
   ,(double)count/N*4,sqrt(count)/N*4);
}
```

Listado 6: Listado del programa `mpi-simple.f` en Fortran

```
      Program simple
      include 'mpif.h'
      integer size,rank,ie,a,len
      character(MPI_MAX_PROCESSOR_NAME) host
      len=20
      call MPI_INIT(ie)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ie)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ie)
      call MPI_GET_PROCESSOR_NAME(host, len, ie)
      a=rank
      WRITE(*,10) 'a=', a,' en proceso ',
     &rank,' de ', size,' en ',host
 10   Format(1x,A4,I2,A12,I2,A4,I2,A4,A20)
      call MPI_FINALIZE(ie)
      end
```

3.1 Ejemplo básico de uso de MPI en Fortran en el LNS

El primer ejemplo se llama mpi-simple.c y se muestra en el Listado 6.

Este código, escrito en Fortran 90, se compila con el comando

```
mpif90 mpi-simple.f -o mpi-simple
```

y se corre con el comando siguiente:

```
mpirun -n 4 mpi-simple
```

Este código produce como resultado lo siguiente:

```
a= 3 en proceso  3 de  4 en master1.lns.buap.mx
a= 1 en proceso  1 de  4 en master1.lns.buap.mx
a= 0 en proceso  0 de  4 en master1.lns.buap.mx
a= 2 en proceso  2 de  4 en master1.lns.buap.mx
```

Tipo en MPI	Tipo en Fortran
MPI_INTEGER	entero
MPI_REAL	real
MPI_DOUBLE PRECISION	precisión doble
MPI_COMPLEX	complejo
MPI_LOGICAL	lógico
MPI_CHARACTER	carácter
MPI_BYTE	(ninguno)
MPI_PACKED	(ninguno)

Tabla 2: Correspondencia entre tipos de variables entre MPI y Fortran en forma general. Nótese que MPI para Fortran 90 cuenta con un conjunto más amplio de tipos [7].

3.2 Ejemplo de uso de la función *reduce* de MPI en Fortran en el LNS

La función *reduce* está definida en Fortran por

```
call MPI_Reduce(sendbuf, recvbuf, cuenta, tipo, operacion, raiz, comm, ierr)
```

La correspondencia entre los tipos de variables en MPI y en Fortran está descrita en forma genérica en la Tabla 2.

A continuación mostramos un ejemplo del uso de la función *reduce* en Fortran. El código respectivo de llama `mpi-reduce.f`, y se muestra en el Listado 7.

Este código se compila con el comando

```
mpif90 mpi-reduce.f -o mpi-reduce
```

Una vez compilado se corre con el comando siguiente:

```
mpirun -n 4  mpi-reduce
```

y produce como resultado lo siguiente:

```
 En proceso    3  a= 3
 En proceso    0  a= 0
 La suma de a es    6
 En proceso    1  a= 1
 En proceso    2  a= 2
```

Listado 7: Listado del programa `mpi-reduce.f` en Fortran

```
      Program reduce
      include 'mpif.h'
      integer size, rank, count,a,a_sum, etiqueta, ierr
      integer stat(MPI_STATUS_SIZE)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
      a=rank
      write(*,10) 'En proceso ', rank, ' a=',a
10    Format(1x,A15,2x,I2, A4,I2)
      call MPI_REDUCE(a,a_sum,1,MPI_INTEGER,
     & MPI_SUM,0,MPI_COMM_WORLD,ierr)
      if (rank .eq. 0) then
      write(*,20) 'La suma de a es ',a_sum
20    Format(1x,A20,2x,I5)
      endif
      call MPI_FINALIZE(ierr)
      end
```

3.3 *Ejemplo de uso de las funciones send y receive de MPI en Fortran en el LNS*

La función *send* está definida en Fortran por

```
CALL MPI_SEND(bufer,cuenta,tipo,destino,etiqueta,MPI_COMM_WORLD,ierr)
```

mientras que la función *receive* está definida por

```
Call MPI_Recv(bufer, cuenta, tipo,fuente,etiqueta, MPI_COMM_WORLD, estado,ierr)
```

A continuación ilustramos el uso de estas funciones *send* y *receive* en Fortran con un ejemplo . El código respectivo se llama `mpi-send-rcv.f` y se muestra en el Listado 8.

Este código se compila con el comando

```
mpif90 mpi-send-rcv.f -o mpi-send-rcv
```

Una vez compilado se corre con el comando siguiente:

```
mpirun -n 4  mpi-send-rcv
```

y produce como resultado lo siguiente:

```
 Datos enviados desde proceso 0 con etiqueta 11 = 11          11          11
 Datos enviados desde proceso 0 con etiqueta 12 = 12          12          12
 Datos recibidos en proceso 1 con etiqueta 11 = 11          11          11
 Datos recibidos en proceso 1 con etiqueta 12 = 12          12          12
```

4 Uso de MPI en Julia en el LNS

El lenguaje Julia [8] es relativamente reciente ya que data de 2012. Julia es un lenguaje de software abierto del tipo "compilado en el momento" que está orientado al cómputo científico. Tiene la ventaja de ser muy simple de aprender y por lo tanto fácil para usar en pruebas de desarrollo de software con una velocidad de ejecución que es casi tan rápida como C. Otra

Listado 8: Listado del programa `mpi-send-rcv.f` en Fortran

```
      Program send y receive
      include 'mpif.h'
      integer size, rank, count,o,in, etiqueta, ie
      integer stat(MPI_STATUS_SIZE)
      INTEGER, DIMENSION(0:2) :: datos11,datos12,data11_r,data12_r
      datos11 = (/11,11,11/)
      datos12 = (/12,12,12/)
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ie)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ie)
      if (rank .eq. 0) then
      call MPI_SEND(datos11,3, MPI_INTEGER,1,11,MPI_COMM_WORLD,ie)
      call MPI_SEND(datos12, 3, MPI_INTEGER,1,12,MPI_COMM_WORLD,ie)
      write(*,*) 'Datos enviados desde proceso 0 con etiqueta 11 ='
     & ,datos11
      write(*,*) 'Datos enviados desde proceso 0 con etiqueta 12 ='
     & ,datos12
      endif
      if (rank .eq. 1) then
      data11_r = (/0,0,0/)
      data12_r = (/0,0,0/)
      call MPI_RECV(data11_r,3,MPI_INTEGER,
     & 0,11,MPI_COMM_WORLD,stat,ie)
      call MPI_RECV(data12_r,3,MPI_INTEGER,
     & 0,12,MPI_COMM_WORLD,stat,ie)
      write(*,*) 'Datos recibidos en proceso 1 con etiqueta 11 ='
     & ,data11_r
      write(*,*) 'Datos recibidos en proceso 1 con etiqueta 12 ='
     & ,data12_r
      endif
      call MPI_FINALIZE(ie)
      end
```

ventaja de Julia es que, al igual que Python, cuenta con una gran comunidad de desarrolladores. Una desventaja obvia es que por su corta edad se encuentra en una fase de menor madurez con respecto a los demás lenguajes usados en este documento. Para aprender Julia existen bastantes recursos libres, ver por ejemplo [9].

Como prerrequisito para usar MPI con Julia en el LNS debemos cargar los módulos siguientes:

```
module load tools/intel/impi/5.0.2.044
module load applications/julia/0.4.6
```

de modo que al ejecutar el comando

```
module list
```

obtenemos lo siguiente:

Listado 9: Listado del programa `mpi-simple.jl` en Julia

```
using MPI
MPI.Init()
comm = MPI.COMM_WORLD
const rank = MPI.Comm_rank(comm)
const size = MPI.Comm_size(comm)
a=rank
print("a=$a en proceso $rank de un total de $size\n")
MPI.Finalize()
```

```
Currently Loaded Modulefiles:
  1) tools/intel/impi/5.0.2.044   2) applications/julia/0.4.6
```

Para agregar el paquete de MPI a Julia basta con teclear lo siguiente:

```
julia
Pkg.add("MPI")
```

Este procedimiento se hace solo una vez pero es importante hacerlo cuando esté cargado el módulo `tools/intel/impi/5.0.2.044`.

4.1 Ejemplo básico de uso de MPI en Julia en el LNS

El primer ejemplo se llama mpi-simple.jl y se muestra en el Listado 9.

Este código se corre con el comando

```
mpirun -n 4 julia mpi-simple.jl
```

y produce como resultado lo siguiente:

```
a=3 en proceso 3 de un total de 4
a=0 en proceso 0 de un total de 4
a=1 en proceso 1 de un total de 4
a=2 en proceso 2 de un total de 4
```

4.2 Ejemplo de uso de la función *reduce* de MPI en Julia en el LNS

El segundo ejemplo en Julia ilustra el uso de la función *reduce*. El código respectivo de llama `mpi-reduce.jl` y se muestra en Listado 10.

Este código se corre con el comando

```
mpirun -n 4 julia mpi-reduce.jl
```

y produce como resultado lo siguiente:

```
a = 2 en proceso 2 de un total de 4
a = 0 en proceso 0 de un total de 4
a = 3 en proceso 3 de un total de 4
a = 1 en proceso 1 de un total de 4
La suma = 6 en el proceso 0
```

Listado 10: Listado del programa `mpi-reduce.jl` en Julia

```
using MPI
MPI.Init()
comm = MPI.COMM_WORLD
const rank = MPI.Comm_rank(comm)
const size = MPI.Comm_size(comm)
a=rank
print("a = $rank en proceso $rank de un total de $size\n")
Sum = MPI.Reduce(a, MPI.SUM, 0, comm)
if rank == 0
 print("La suma = $Sum en el proceso $rank \n",);
end
MPI.Finalize()
```

Listado 11: Listado del programa `mpi-send-rcv.jl` en Julia

```
using MPI
MPI.Init()
comm = MPI.COMM_WORLD
const rank = MPI.Comm_rank(comm)
const size = MPI.Comm_size(comm)
if rank == 0
 data11 = ones(3)*1
    data12 = ones(3)*2
    MPI.Send(data11,  1, 11, comm)
    MPI.Send(data12,  1, 12, comm)
elseif rank == 1
    data_recibida11 =zeros(3)
    data_recibida12 =zeros(3)
    MPI.Irecv!(data_recibida11, 0, 11, comm)
    MPI.Irecv!(data_recibida12, 0, 12, comm)
end
MPI.Barrier(comm)
if rank == 1
println("En proceso ",rank,"  data_recibida11 =",data_recibida11)
println("En proceso ",rank,"  data_recibida12 =",data_recibida12)
end
MPI.Finalize()
```

4.3 Ejemplo de uso de las funciones *send* y *receive* de MPI en Julia en el LNS

El tercer ejemplo en Julia muestra el uso de las funciones *send* y *receive*. El código respectivo se llama `mpi-send-rcv.jl` y se muestra en el Listado 11.

Este código se corre con el comando

```
mpirun -n 4 julia mpi-send-rcv.jl
```

y produce como resultado lo siguiente:

```
En proceso 1  data_recibida11 =[1.0,1.0,1.0]
En proceso 1  data_recibida12 =[2.0,2.0,2.0]
```

Nótese que si se comenta la línea `MPI.Barrier(comm)` se pierde la sincronización y se incurre en errores.

5 Uso de MPI en Python en el LNS

Aunque Python es más lento que los lenguajes anteriores, tiene la ventaja de que existe una gran cantidad de código libre para su aplicación en casi todas las áreas del conocimiento, con el atenuante adicional de que están surgiendo paquetes que permiten acelerar la ejecución de Python, como por ejemplo Numba [10], que permite compilar en el momento el código de las subrutinas que el usuario seleccione. Otra ventaja de Python es la rapidez con que se puede programar un sistema de prueba que eventualmente se puede reescribir en C o en Fortran una vez que se ha depurado y optimizado.

Como prerrequisito para usar MPI con Pyjthon en el LNS debemos cargar los módulos siguientes:

```
module load tools/intel/impi/5.0.2.044
module load applications/anaconda/3.19.1
```

de modo que al ejecutar el comando

```
module list
```

obtenemos lo siguiente:

```
Currently Loaded Modulefiles:
  1) tools/intel/impi/5.0.2.044
  2) applications/anaconda/3.19.1
```

El módulo de Python que contiene la implementación de MPI se llama `mpi4py` [11] y se instala en el clúster del LNS al cargar el módulo *applications/anaconda*/3.19.1.

5.1 Ejemplo básico de uso de MPI en Python en el LNS

El primer ejemplo se llama mpi-simple.py y se muestra en el Listado 12.

Este código se corre con el comando

```
mpirun -n 4 python mpi-simple.py
```

y produce como resultado lo siguiente:

```
a=0 en proceso 0 de 4 en master1.lns.buap.mx
a=2 en proceso 2 de 4 en master1.lns.buap.mx
a=3 en proceso 3 de 4 en master1.lns.buap.mx
a=1 en proceso 1 de 4 en master1.lns.buap.mx
```

Listado 12: Listado del programa `mpi-simple.py` en Python

```
from mpi4py import MPI;import socket
comm = MPI.COMM_WORLD
size = MPI.COMM_WORLD.Get_size()
rank = comm.Get_rank()
host=socket.gethostname()
a=rank
for i in range(size):
    if i==rank:
        print("a=%d en proceso %d de %d en %s"%(a,rank, size,host))
MPI.Finalize()
```

5.2 Ejemplo de uso de la función *reduce* de MPI en Python en el LNS

El segundo ejemplo en Python ilustra el uso de la función *reduce*. El código respectivo de llama `mpi-reduce.py` y se muestra en el Listado 13.

Listado 13: Listado del programa `mpi-reduce.py` en Python

```
from mpi4py import MPI;import numpy ;import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
a = random.randint(0,3)
A=numpy.ones(3)*random.randint(0,3)
print("En proceso %d a = %s A= %s" % (rank,a,A))
sum_a = comm.reduce(a, op=MPI.SUM, root =0)
sum_A = comm.reduce(A, op=MPI.SUM, root =0)
if rank == 0:
    print("La suma de a es %s y la de A es %s" % (sum_a,sum_A))
MPI.Finalize()
```

Este código se corre con el comando

```
mpirun -n 4 python mpi_reduce.py
```

y produce como resultado lo siguiente:

```
En proceso 1 a = 0 A= [ 1.  1.  1.]
En proceso 0 a = 3 A= [ 1.  1.  1.]
En proceso 3 a = 3 A= [ 0.  0.  0.]
En proceso 2 a = 3 A= [ 3.  3.  3.]
La reduccion de a es 9 y la de A es [ 5.  5.  5.]
```

5.3 Ejemplo de uso de las funciones *send* y *receive* de MPI en Python en el LNS

El tercer ejemplo en Python muestra el uso de las funciones *send* y *receive*. El código respectivo se llama `mpi-send-rcv.py` y se muestra en el Listado 14

Este código se corre con el comando

```
mpirun -n 4 python mpi-send-rcv.py
```

y produce como resultado lo siguiente:

```
En proceso 1, data11_recibido =[ 11.  11.  11.]
En proceso 1, data12_recibido =[ 12.  12.  12.]
```

Listado 14: Listado del programa `mpi-send-rcv.py` en Python

```
from mpi4py import MPI; import numpy as np ; import random
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
if rank == 0:
    data11 = np.ones(3)*11
    data12 = np.ones(3)*12
    comm.send(data11, dest=1, tag=11)
    comm.send(data12, dest=1, tag=12)
elif rank == 1:
    data11_r = np.zeros(3)
    data12_r = np.zeros(3)
    data11_r = comm.recv(source=0, tag=11)
    data12_r = comm.recv(source=0, tag=12)
    print("En proceso %d, data11_recibido =%s"%(rank, data11_r))
    print("En proceso %d, data12_recibido =%s"%(rank, data12_r))
MPI.Finalize()
```

Bibliografía

[1] http://dl.acm.org/citation.cfm?id=898758
[2] http://dl.acm.org/citation.cfm?id=898758
[3] http://pawangh.blogspot.mx/2014/05/mpi-vs-openmp.html
[4] http://www.ceci-hpc.be/slurm_tutorial.html
[5] https://www.rc.colorado.edu/support/user-guide/batch-queueing.html
[6] https://github.com/lvillasen/TutorialMPI
[7] http://mpi-forum.org/docs/mpi-2.2/mpi22-report/node353.htm
[8] http://julialang.org/
[9] http://julialang.org/learning/
[10] http://numba.pydata.org/
[11] https://pythonhosted.org/mpi4py/apiref/index.html