Arquitecturas Distribuidas
Trabajo Práctico Nº3 - Parte A
Montaje y uso de un Cluster Beowulf
Año 2025

Introducción

En este trabajo práctico se implementará un cluster Beowulf, en el cual todos los estudiantes deberán trabajar colaborativamente. Cada estudiante o grupo de estudiantes deberán configurar su computadora para que formen parte de un cluster de mayor tamaño, empleando las computadoras del laboratorio de informática.

Cada estudiante o grupo de estudiantes deberá crear una carpeta en su computadora local y el resto de las computadoras del cluster donde depositar sus programas, y permitir al resto de los estudiantes crear carpetas en su computadora local.

En este trabajo práctico se pide resolver problemas paralelizables a nivel de procesos empleando alguna implementación abierta de MPI (OpenMPI para C++ o MPI4Py para Python) para disminuir el tiempo de ejecución o incrementar el throughput. Los problemas a resolver en este trabajo práctico también fueron resueltos en el trabajo práctico N°1 mediante paralelización a nivel de hilos.

Objetivos

- Reconocer los componentes de un clúster de computadoras, el software intermedio y la red subyacente.
- Implementar y poner en marcha un cluster beowulf simple.
- Escribir y ejecutar programas que empleen paralelismo a nivel de procesos sobre MPI para resolver problemas paralelizables.
- Comprender cómo intervienen la configuración de la red subyacente, el software intermedio y componentes del sistema operativo en el funcionamiento de un clúster de computadoras.
- Analizar las ventajas y desventajas de diferentes tipos de paralelismo en cuanto a tiempo de ejecución, porcentaje de uso de los procesadores y utilización de la memoria.

Metodología

Trabajo individual o grupal. 2 estudiantes por grupo máximo.

Tiempo de realización aproximado: 8 - 10 horas (4 - 5 clases).

Aprobación

- Mostrar en clases el funcionamiento de los programas escritos.
- Elaborar un informe indicando las características del cluster en el cual ejecuten los programas (cantidad de computadoras y núcleos) y para cada ejercicio el speedup logrado. No incluir código en el informe. 2 carillas máximo.
- Subir a la plataforma Moodle el código de los programas escritos.

Materiales necesarios

- Herramientas de software:
  - Compilador de C++ versión C++11 o superior, preferentemente g++.
  - Interprete Python3.
  - OpenMPI para C++ (en el Anexo 1.1 se proveen instrucciones de instalación). Alternativamente puede utilizarse MPICH si hubiera problemas con el primero.
  - MPI4PY para Python3 (en el Anexo 1.2 se proveen instrucciones de instalación).
- Herramientas de hardware:
  - Dos o más computadoras con las herramientas mencionadas anteriormente instaladas (se utilizarán las computadoras del laboratorio de informática de la Facultad de Ingeniería).

Ejercicio 1 (Instalación y configuración del Cluster):

Configure su computadora para que pueda formar parte de un cluster de mayor tamaño. Todas las computadoras que formen parte del cluster deberán ser configuradas iguales, se sugiere realizar la siguiente configuración:

1a) Crear nuevo usuario de Linux

Cree un usuario llamado **mpiuser** con contraseña **Arquitecturas2025** (prestar atención a las mayúsculas) con permisos de administración (que sea parte del grupo **sudo**).

Para trabajar con las computadoras del laboratorio de informática de la facultad de Ingeniería crearemos un usuario de Linux diferente al usuario “estudiante” para no alterar la configuración de las máquinas. Este paso no es necesario si trabaja en su computadora personal (Se sugiere repasar los conceptos de usuarios y permisos vistos en la asignatura Sistemas Operativos).

Para crear un nuevo usuario de Linux en la terminal ingrese:

sudo adduser mpiuser

Donde mpiuser es el nombre de usuario del nuevo usuario a crear (se sugiere usar

ese nombre). Se pedirá una contraseña, se sugiere usar la indicada arriba para que todas las máquinas utilicen la misma contraseña. Se pedirán otros datos (Nombre completo, teléfono, etc.) ignore esos datos presionando Enter.
Para dar permisos de administrador al nuevo usuario ingrese:

sudo usermod -aG sudo mpiuser

Notar que -a indica *append* (agregar) y G *grupo*. De modo que está agregando al usuario *mpiuser* al grupo *sudo*.
Para cambiar de usuario ingrese:

su mpiuser

Se sugiere cambiar de usuario a través de la interfaz gráfica (ir a "apagar/cerrar sesión" y luego a "cambiar de usuario") para evitar errores entre la interfaz gráfica (probablemente la usará para copiar archivos) y la terminal.

1b) Instalar OpenMPI para C++ (ver Anexo 1.1). Instalar MPI4PY para Python3 (ver Anexo 1.2). Para instalar MPI4PY, necesita tener instalado y funcionando correctamente OpenMPI.
En este trabajo práctico podrá programar utilizando C++ o Python3. Pero deberá instalar ambos middlewares (OpenMPI y MPI4PY), para permitir que sus compañeros trabajen sobre su computadora pudiendo elegir cualquiera de los dos lenguajes.

1c) **Configurar carpetas de trabajo**
Todos los archivos ejecutables deben estar en la misma ruta (carpetas) en todas las computadoras del cluster. Cada estudiante deberá tener su propia carpeta en todas las máquinas del cluster. Se sugiere crear y utilizar la carpeta **/home/mpiuser/nombre_estudiante**.
Permita a sus compañeros crear sus carpetas de trabajo y copiar sus programas a su computadora.

comandos útiles:
mkdir nombre_carpeta crea una carpeta en el directorio actual.
ls lista los archivos y carpetas del directorio actual.
rm nombre_del_archivo borra un archivo
ssh usuario_remoto@ip para conectarse con una máquina remota a través de ssh
sshfs usuario_remoto@ip:/home/mpiuser/ /home/mpiuser/nombre_carpeta
monta el directorio */home/mpiuser/* del usuario remoto en el directorio local */home/mpiuser/nombre_carpeta* (útil para escribir el archivo de IPs o copiar ejecutables a otras máquinas).

1d) **Creación de clave pública para evitar que SSH pida la contraseña**

Cree un par clave pública y clave privada para SSH con:

ssh-keygen

Se le pedirán datos como: carpeta en la cual se guardará el archivo con la clave pública (por defecto será .ssh) y nombre (por defecto será id_rsa), una frase de seguridad (puede saltar este paso). Puede presionar Enter para elegir las opciones por defecto.

Puede ver los archivos con su par clave pública y clave privada en la carpeta .ssh, usualmente se llamarán id_rsa para la clave privada e id_rsa.pub para la clave pública. También verá un archivo llamado known_hosts, que contiene las claves públicas que su computadora ya conoce.

El siguiente paso es copiar su clave pública en el archivo known_hosts de la computadora con la cual se quiere comunicar. Esto puede hacerlo con la aplicación ssh-copy-id como:

ssh-copy-id user@ip

Donde *user* e *ip* es la dirección IP de la computadora a la cual quiere enviar y agregar su clave pública. Deberá realizar este paso con todas las máquinas que se vayan sumando al cluster (ver punto siguiente).

1e) Copie a su máquina el archivo /home/mpiuser/ips.txt que encontrará en la máquina **10.65.1.x** (puede hacerlo con scp, sshfs o ftp). En ese archivo están todas las computadoras que ya son parte del cluster.

Comparta su clave pública con cada una de las computadoras indicadas en ese archivo con **ssh-copy-id user@ip** (ver punto anterior).

Agregue su propia ip a dicho archivo en su máquina.

Debería poder conectarse por ssh con cada una de esas máquinas sin que pida contraseña.

Realice la prueba de funcionamiento (paso siguiente).

1f) Prueba rápida de funcionamiento

Ejecute el siguiente comando:

mpirun --hostfile ips.txt --mca btl_tcp_if_include ip_local/prefijo_red hostname

Notar que ips.txt es el archivo en el cual están las direcciones IPs de las máquinas del cluster. Notar que el parámetro *ip_local/prefijo_red* es la IP local, indicando el prefijo de red o tamaño de la máscara de red (repasar conceptos de redes de computadoras).

Si OpenMPI ha sido correctamente configurado, deberá ver una línea por cada núcleo agregado al cluster (número de IPs multiplicado por el número de núcleos en cada máquina).

Luego ejecute el comando:
mpirun *--hostfile ips.txt --mca btl_tcp_if_include ip_local/prefijo_red* python3 -m mpi4py.bench helloworld
Si MPI4PY ha sido correctamente configurado, deberá ver una línea por cada núcleo agregado al cluster.

1g) Cuando OpenMPI, MPI4PY y la configuración de carpetas esté lista en su máquina local, copie su IP en el archivo /home/mpiuser/ips.txt que encontrará en la máquina **10.65.1.120**. Su máquina ya es parte del cluster.
Verifique con frecuencia si nuevas máquinas se agregaron al archivo indicado, para agregarlas a su archivo local.

1h) **Primer programa**
Escriba un programa conformado por n procesos que se ejecuten en paralelo que escriba por pantalla la frase:

> *Hola Mundo! soy el proceso <X> de <TOTAL_PROCESOS> corriendo en la máquina <nombre de la máquina> IP= <direccion_IP>*

donde:

- <X> es el número ID del proceso.
- *<TOTAL_PROCESOS>* es el total de procesos.
- *<dirección_IP>* es la dirección IP de la máquina donde corre el proceso (debe ser la IP a través de la cual se accede a Internet, no la localhost).

Ejecute el programa creado en una computadora y luego en varias computadoras (ver en **Anexo 2** detalles de como escribir un programa en MPI).

Nota: Un método para saber la IP de la máquina donde corre un proceso es crear un socket TCP y conectarlo a un socket que provea algún servicio conocido (por ejemplo, el servidor web de la UNCuyo). Luego obtener la IP local a la cual está conectado dicho socket. Por último, no olvidar cerrar el socket.
En el aula abierta se proveen como ejemplos los archivos ***direccion_IP.cpp*** y ***direccion_IP.py*** con programas que permiten obtener la dirección IP en C++ y Python. Dichos códigos pueden agregarse como librerías a sus programas.

Para usar en C++, incluya en su código la librería #include "direccion_IP.cpp", y luego en el el código ejecute las siguientes instrucciones:

```
char mi_IP[100];
obtener_IP(mi_IP);
```

Estas instrucciones depositarán la IP local en la variable mi_IP.

Para usar en Python, en su código importe el módulo con import direccion_IP, y luego en el código ejecute:

direccion_IP.obtener_IP()

Esa función retornará la IP en forma de cadena de caracteres. Podrá imprimirla por pantalla directamente o almacenarla en alguna variable.

Ejercicio 2:

Escribir un programa que calcule el logaritmo natural de números mayores a 1500000 ($1.5*10^6$) en punto flotante de doble precisión largo (tipo **long double** en C++, **float** en Python) mediante serie de Taylor, empleando 10000000 (diez millones) de términos de dicha serie. El resultado debe imprimirse con 15 dígitos. Deberá utilizar N procesos que se ejecuten en paralelo, resolviendo cada proceso una parte de la sumatoria de la Serie de Taylor.

Serie de Taylor del logaritmo natural:

$$\ln(x) = 2\sum_{n=0}^{\infty} \frac{1}{2n+1}\left(\frac{x-1}{x+1}\right)^{2n+1}$$

a) Ejecute el programa empleando un solo proceso, luego en dos, tres, y así sucesivamente hasta alcanzar el triple del número máximo de núcleos disponibles. Seleccione el número de procesos con menor tiempo de ejecución, y para ese número de procesos, realice las siguientes operaciones.

b) Tome nota del tiempo de ejecución y calcule el speedup respecto a un solo proceso.

c) Compare los resultados con los obtenidos en el ejercicio N°1 del trabajo práctico N°1.

Ayuda: ln(1500000)=14.2209756660724

Ejercicio 3:

Se da un archivo de datos en forma de caracteres llamado “texto.txt” de 200 MB (200 millones de caracteres), que puede descargar desde: https://drive.google.com/file/d/19Bnah1DCzZu5wJvFeuppcx3voUOtOEFS/view?usp=sharing , y 32 patrones, cada uno en forma de cadena de caracteres almacenados una por línea en el archivo “patrones.txt” (que se encuentran en la carpeta del trabajo práctico N°3). Tanto el archivo “texto.txt” como el archivo “patrones.txt” son los mismos que se utilizaron en el Trabajo Práctico N°1. Cree un programa que busque la cantidad de veces que cada patrón aparece en el archivo “texto.txt”. El programa deberá generar una salida similar a la siguiente:

el patron 0 aparece 14 veces.
el patron 1 aparece 3 veces.

el patron 2 aparece 0 veces.
el patron 3 aparece 0 veces.
el patron 4 aparece 0 veces.
el patron 5 aparece 0 veces.
……..
……..

El archivo “texto.txt” no posee saltos de línea (es decir, posee solo una línea).
Resuelva el problema de dos formas:

- Empleando un solo proceso que busque todos los patrones (en el Trabajo Práctico N°1 se creó este programa).
- Empleando 32 procesos de modo que cada proceso busque un patrón.

Incluya código que permita obtener el tiempo de ejecución y calcule el speedup.
a) Ejecute el programa en una sola máquina (MPI distribuirá los procesos en los núcleos disponibles). Luego en todas las máquinas disponibles.
b) Tome nota del tiempo de ejecución y calcule el speedup respecto a un solo proceso.
c) Compare los resultados con los obtenidos en el ejercicio N°2 del trabajo práctico N°1.

Nota: Los archivos “texto.txt” y “patrones.txt” deben estar copiados en todas las computadoras donde se vayan a correr procesos.

Ayuda: El patrón 0 aparece 14 veces, el patrón 1 aparece 3 veces, el patrón 6 aparece 4 veces, el patrón 9 aparece 3622 veces, el patrón 11 aparece 2 veces, el patrón 13 aparece 6 veces, el patrón 16 aparece 2 veces, el patrón 18 aparece 6 veces, el patrón 21 aparece 2 veces, el patrón 27 aparece 6 veces, todos los demás patrones aparecen 0 veces.

Ejercicio 4:

Escriba un programa que realice la multiplicación de dos matrices de N*N elementos (siendo N un número grande, como 300, 1000 o 3000) del tipo **float** (número en punto flotante), y luego realice la sumatoria de todos los elementos de la matriz resultante. Muestre por pantalla los elementos de cada esquina de las matrices y el resultado de la sumatoria.
Escriba el programa de modo que múltiples procesos trabajen concurrentemente, resolviendo cada proceso un grupo de las N filas o columnas de la matriz resultado (nota, usar miles de procesos no es la forma más eficiente de resolver el problema, a menos que se disponga de miles de núcleos). Incluya código que permita obtener el

tiempo de ejecución.
Requisitos:

- El programa deberá permitir cambiar el número N de filas y columnas de las matrices.

Ayuda:
Si los elementos de la matriz1 son 0.1, y los elementos de la matriz2 son 0.2, suponiendo matrices de 300*300, la matriz resultante será (se muestran solo los elementos en los extremos):

|6.000 …… 6.000|
| ……|
|6.000 …… 6.000|

y el resultado de la sumatoria será 540000.

Si el número de elementos de las matrices es 1000*1000, la matriz resultante será (se muestran solo los elementos en los extremos):

|20.0003 …… 20.0003|
| ……..|
|20.0003 …… 20.0003|

y el resultado de la sumatoria será $2*10^7$.

Si el número de elementos de las matrices es 3000, la matriz resultante será (se muestran solo los elementos en los extremos):

|60.0012 …… 60.0012|
| ……|
|60.0012 …… 60.0012|

y el resultado de la sumatoria será $5.71526*10^8$.

Ejercicio N°5:
Escriba un programa que busque todos los números primos menores a un número N que se ingresará por teclado. Debe mostrar por pantalla los 10 mayores números primos y la cantidad de números primos menores que N. Utilice como datos números del tipo ***long long int***.
El problema debe resolverse de modo que el usuario pueda elegir cualquier número de procesos. El número N debe ingresarse por teclado. Cada proceso debe resolver una parte del problema.
a) Incluya código que permita obtener el tiempo de ejecución en cada programa y calcule el speedup.
b) Observe el porcentaje de uso de cada núcleo en cada implementación.

Ayuda:

Hay 78498 números primos menores que 10^6 (un millón), siendo los 5 mayores: 999953, 999959, 999961, 999979, 999983.

Hay 664579 números primos menores que 10^7 (diez millones), siendo los 5 mayores: 9999937, 9999943, 9999971, 9999973, 9999991.

Hay 5761455 números primos menores que 10^8 (cien millones), siendo los 5 mayores: 99999931, 99999941, 99999959, 99999971, 99999989.

Anexo 1: Instalación de OpenMPI y MPI4PY

1.1 Instalación de OpenMPI

OpenMPI y MPICH son implementaciones de MPI que trabaja sobre C, C++ o Fortran. Permiten comunicar procesos en diferentes computadoras mediante ssh (debe tener instalados cliente y servidor ssh de modo que no pida contraseña).
MPI4PY es un middleware para programar en MPI usando Python3. Requiere tener instalado y funcionando correctamente OpenMPI (o MPICH).

Requisitos previos

Antes de instalar cualquier paquete de software, se recomienda ejecutar:
sudo apt update y luego ***sudo apt upgrade*** (este comando puede demorar varios minutos en ejecutarse).
Cuidado: Si el sistema le pide instalar Grub (gestor de arranque de Linux) en las computadoras del laboratorio, no lo instale. Las computadoras del laboratorio utilizan un gestor de arranque diferente.

Se requiere tener instalados:

- g++: Instalado por defecto en Ubuntu y Debian. Para verificar instalación use ***g++ --version*** (deberá ver la versión de g++ instalada en su computadora). Puede instalarlo con ***sudo apt install g++***.
- Python3: Instalado en versiones actuales de Ubuntu y Debian. Para verificar instalación use ***python3 --version*** (deberá ver la versión de Python3 instalada en su computadora). Puede instalarlo con ***sudo apt install python3***.
- Un servidor y un cliente ssh. Puede instalar un servidor y un cliente de ssh con ***sudo apt install openssh-client openssh-server*** (Ubuntu y Debian tienen por defecto instalado un cliente ssh).

Instalación de OpenMPI y MPICH

Opción 1 (recomendado):

mpirun hostname

Para instalar OpenMPI desde los repositorios, ejecute en una terminal:
sudo apt install openmpi-bin libopenmpi-dev openmpi-common
openmpi-bin está instalado por defecto en la mayoría de las distribuciones de

Ubuntu. Es posible que libopenmpi-dev openmpi-common ya estén instaladas también.
En las computadoras del laboratorio se sugiere instalar OpenMPI. Si quisiera instalar MPICH en su computadora personal, desde los repositorios ejecute: ***sudo apt install mpich*** (se sugiere no instalar openmpi y mpich en la misma computadora, ya que pueden tener problemas de dependencias).

Nota: Para desinstalar un programa, puede usar "***sudo apt remove nombre_del_programa***"

Para verificar la instalación ejecute:

mpirun -n 2 hostname (Se ejecutarán dos procesos en dos núcleos de su computadora. Cada proceso muestra el nombre de su computadora).

mpirun hostname (Se ejecutarán tantos procesos como núcleos posea en su computadora. Cada proceso muestra el nombre de su computadora).

mpirun --version (verá la versión de OpenMPI o MPICH instaladas)

Opción 2:
Descargar los archivos de instalación desde la página web de OpenMPI: https://www.open-mpi.org (sección "Open MPI Software") o MPICH: https://www.mpich.org (sección "downloads"). Seguir las instrucciones de instalación (pueden encontrarse en un archivo llamado "INSTALL" o "README").

Opción 3**:**
Instalar OpenMPI o MPICH en una máquina virtual provista por un proveedor de Cloud Computing como Google Cloud Computing.
Prepare y encienda su máquina virtual provista por Google Cloud Computing (si posee una cuenta de Gmail, posee una máquina virtual gratuita en la infraestructura de Google Cloud Computing).
Siga los pasos explicados en "Opción 1."

Desactivar el firewall
Será necesario desactivar el firewall o agregar permisos de acceso a las IP del cluster. Para desactivar el firewall puede usar:

sudo ufw disable

1.2 Instalación de MPI4PY

MPI4PY (MPI for Phyton) provee middleware para escribir programas de MPI usando lenguaje Python. (Website oficial: https://mpi4py.readthedocs.io/en/stable/)

Requisitos previos

Tener instalado pip3 (instalador oficial de paquetes de Python). Para verificar instalación puede usar ***pip3 --version*** (deberá ver la versión instalada de pip3). Si no lo tiene instalado, instálelo con: ***sudo apt install python3-pip***

Tener instalado y funcionando correctamente OpenMPI o MPICH (ver Anexo 1) y Python3.

Para instalar MPI4PY en un entorno Python, ejecute: ***pip3 install mpi4py***

Si el comando anterior fallara (puede fallar si en su sistema hay múltiples instancias de Python instaladas), puede usar:

python3 -m pip install mpi4py

o bien, para instalar MPI4PY para todo el sistema (sin usar entornos Python), ejecute:

sudo apt install python3-mpi4py

Para verificar la instalación, ejecute:

mpirun -n 2 python3 -m mpi4py.bench helloworld

Deberá ver un mensaje de “helloword” en dos núcleos de su computadora.

o también:

mpirun python3 -m mpi4py.bench helloworld

Deberá ver tantos mensajes de “helloword” como núcleos haya en su computadora.

Anexo 2: Primitivas de MPI sobre C++

2.1 Cuestiones comunes a ambos lenguajes

Será necesario copiar su archivo ejecutable a todas las máquinas en las cuales necesite que se ejecute. Para ello puede usar alguna de las siguientes herramientas (las cuales fueron vistas en la asignatura Redes de Computadoras):

- opción 1:

scp /home/mpiuser/su_carpeta/ruta_al_archivo mpiuser@ip_remoto:/home/mpiuser/su_carpeta/

Deberá ejecutar dicho comando para copiar sus ejecutables a cada computadora en el cluster (puede ser más cómodo crear un script para este propósito, ChatGPT puede crear dicho scrip en segundos, siempre y cuando le proporcione el prompt adecuado).

- opción 2:

Usar sshfs para montar la copia de su carpeta en todas las máquinas remotas en su computadora local y copiar los archivos utilizando el explorador de archivos de Ubuntu (puede ser engorroso si hay muchas máquinas).

- opción 3:

Usar rsync (herramienta de sincronización entre máquinas. Similar a Dropbox, pero descentralizado).

rsync -r --delete /home/mpiuser/su_carpeta/ mpiuser@ip:/home/mpiuser/su_carpeta/

Deberá ejecutar dicho comando para sincronizar el contenido de su carpeta local con las copias en cada computadora en el cluster (puede ser más cómodo crear un script para este propósito).

Cuidado: Sincronice solo su directorio, no el de los demás usuarios.

Algunas banderas útiles del comando mpirun

-oversubscribe: Etiqueta necesaria si utiliza OpenMPI cuando se van a ejecutar

más procesos que núcleos disponibles (No necesaria si se utiliza MPICH).

--mca btl_tcp_if_include <interface o IP>,<interface o IP>/: En caso de tener más de una interfaz local, puede ser necesario especificar la interfaz local a utilizar (usual cuando hay dockers o virtualización).
--mca btl_tcp_if_exclude <interface o IP>,**<interface o IP>/**: En caso de tener más de una interfaz, puede ser necesario especificar la o las interfaz a excluir (usual cuando hay dockers o virtualización).

2.2 Primitivas de MPI en C++

Un listado completo de primitivas puede encontrarse en:
https://www.open-mpi.org/doc/v4.1/.
Las mismas se estudiarán en la asignatura Programación Paralela y Distribuida.
Abajo se provee un resumen de las 5 más utilizadas.

Librerías:
Su programa deberá incluir la librería **#include <mpi.h>**

Estructura general de un programa en MPI sobre C++

```
#include <mpi.h>
//……Otras librerías……
//……Otro código……
int rank, size;
if(MPI_Init(NULL, NULL)!=MPI_SUCCESS)
{
   cout<<"Error iniciando MPI"<<endl;
   exit(1);
}
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
//……Código del programa…….
if(MPI_Finalize()!=MPI_SUCCESS)
{
   cout<<"Error finalizando MPI"<<endl;
   exit(1);
}
```

A continuación se aclara que significan algunas de estas primitivas.

Inicialización y finalización

Todo el código MPI debe escribirse entre las primitivas **MPI_Init(int& argc, char**& argv)** y **MPI_Finalize()**. Estas funciones poseen la siguiente sintaxis:

MPI_Init(int& argc, char& argv)**: puede recibir argumentos. Los argumentos se pasan como un vector de argumentos. **argc** es un puntero a una variable donde se indica el número de argumentos. **argv** es un puntero al vector de argumentos. Si no se pasa ningún argumento, puede escribirse como **MPI_Init(NULL, NULL).**

MPI_Init puede retornar **MPI_SUCCESS** o **MPI_ERR_OTHER**, según MPI se haya inicializado correctamente o no (La fuente principal de errores es estar llamando dos veces a MPI_Init). Forma recomendada de uso:

```
if(MPI_Init(NULL, NULL)!=MPI_SUCCESS)
{
   cout<<"Error iniciando MPI"<<endl;
   exit(1);
}
```

MPI_Finalize() no recibe ningún argumento, y al igual que MPI_Init, puede devolver **MPI_SUCCESS** o **MPI_ERR_OTHER**. Forma usual de uso:

```
if(MPI_Finalize()!=MPI_SUCCESS)
{
   cout<<"Error finalizando MPI"<<endl;
   exit(1);
}
```

Obtención del número ID del proceso y el número total de procesos

MPI_Comm_rank(MPI_COMM_WORLD,&rank): Obtiene el número que identifica al proceso. El primer proceso se identifica con 0 (cero). Notar que el valor se escribe en una variable (“rank” en el ejemplo) que debe ser pasada por referencia. La variable rank es de tipo entero (***int***). La variable rank debe ser creada antes de llamar a la primitiva.

MPI_Comm_size(MPI_COMM_WORLD,&size): Obtiene el número total de procesos. La respuesta se escribirá en la variable del tipo ***int*** llamada “size”.

compilar un programa que utiliza MPI y C++

OpenMPI y MPICH disponen de 3 compiladores: mpiCC (debe utilizar las mayúsculas, porque mpicc es el compidador para C), mpicxx o mpic++. Para compilar usado mpicxx debe usar:

mpicxx -O3 -o nombre_ejecutable.out archivo_codigo_fuente.cpp

-O3 indica el nivel de optimización más alto. No es obligatorio, pero es necesario si desea obtener niveles altos de speedup.

Ejecutar un programa que utiliza MPI y C++:

mpirun -n 10 --hostfile ips.txt ./nombre_ejecutable.out

donde:

-n 10: Es el número de procesos a correr. Si se omite, se ejecutarán tantos procesos como núcleos disponibles.

--hostfile: Indica el archivo en el que se encuentran las IPs de las máquinas en las cuales se ejecutará la aplicación (en el ejemplo, el archivo se llama ips.txt). Si la aplicación se ejecuta en una sola máquina, no es necesario especificar este parámetro.

Primitivas más usadas

Todas las primitivas retornan MPI_SUCCESS si se ejecutaron correctamente o diferentes códigos de error si se producen errores.

MPI_Get_processor_name(name,&length);

Permite obtener el nombre de la computadora. Dicho nombre se almacenará en una variable llamada “name” (del tipo array de char), y la longitud del nombre se almacenará en una variable llamada “length” (del tipo int). Deberá crear previamente estas variables.

int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int rank_dest, int tag, MPI_Comm comm)

Donde:

buf: Dirección inicial del buffer con los datos a enviar. Note que el puntero es del tipo void, por lo que el buffer puede contener cualquier tipo de datos, incluso matrices, matrices de matrices, etc.

count: Número de elementos a enviar.

datatype: Tipos de datos a enviar, definidos por MPI. Puede tomar los siguientes valores: MPI_SHORT, MPI_INT, MPI_LONG, MPI_LONG_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_UNSIGNED_LONG_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE (para char)

rank_dest: proceso destino.

tag: Etiqueta del mensaje.

comm: Comunicador.

MPI_Send trabaja en conjunto con MPI_Recv.

int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag,

MPI_Comm comm, MPI_Status *status);
Los argumentos son muy similares a MPI_Send. Las diferencias son:
buf: Es un buffer donde se guardarán los datos recibidos.
count: Cantidad de datos máximos que se esperan recibir.
source: ID del proceso del que se esperan datos.
status: Estructura que indica el resultado de la operación. Tiene tres campos: status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR. Si no se desea utilizar, puede pasarse como argumento la constante MPI_STATUS_IGNORE.
MPI_Recv trabaja en conjunto con MPI_Send.

int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
Envía un dato por broadcast. Un proceso envía los datos, los demás lo reciben (incluyendo el proceso que realiza el envío).
buffer: buffer donde se almacenan los datos a enviar por el proceso que envía, y donde los procesos que reciben almacenarán el dato. Puede ser un solo dato, un vector, una matriz, etc.
count: Número de elementos a enviar.
datatype: Tipos de datos a enviar. Ver primitiva MPI_Send.
root: ID del proceso que envía los datos.

MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
Recoge o reune datos en un proceso, cuyo identificador se indica como **root**, desde el resto de los procesos, incluido el root (ver teoría).
sendbuf: Buffer donde cada proceso almacena los datos a enviar al root.
sendcount: Número de elementos a enviar por proceso.
sendtype: Tipos de datos a enviar, definidos por MPI. Ver primitiva MPI_Send.
recvbuf: Es un vector o colección de elementos donde se depositarán los datos recibidos. Debe contener un elemento para cada proceso.
recvcount: Número de elementos a recibir desde cada proceso.
recvtype: Tipos de datos a recibir, definidos por MPI. Ver primitiva MPI_Send.

MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
Esparce datos desde un proceso, cuyo identificador se indica como root, hacia todos los procesos. Los datos enviados a cada proceso son diferentes. (ver teoría).

Los argumentos tienen igual significado que la primitiva MPI_Gather.

2.3 Primitivas de MPI en Python3

Estructura general de un programa en MPI sobre Python

La estructura de un programa MPI en Python es la siguiente:

```
from mpi4py import MPI #Librería para usar MPI en un programa de Python
#.....Otras librerías.....
#.....Otro código.....
comm = MPI.COMM_WORLD #Obtenemos el comunicador
size = MPI.COMM_WORLD.Get_size()  #Número total de procesos
rank = MPI.COMM_WORLD.Get_rank()  #ID del proceso actual
#.....Código del programa.....
```

Ejecutar un programa que utiliza MPI sobre Python3:

mpirun -n 10 --hostfile ips.txt python3 nombre_programa.py

donde:

-n 10: Es el número de procesos a correr

--hostfile: Indica el archivo en el que se encuentran las IPs de las máquinas en las cuales se ejecutará la aplicación (en el ejemplo, el archivo se llama ips.txt). Si la aplicación se correrá en una sola máquina, no es necesario especificar este parámetro.

-oversubscribe: Etiqueta necesaria si utiliza OpenMPI cuando se van a ejecutar más procesos que núcleos disponibles (No necesaria si se utiliza MPICH).

Primitivas más utilizadas

name = MPI.Get_processor_name()

Permite obtener el nombre de la máquina en la cual corre el proceso.

comm.send(dato,dest=i,tag=11)
Envía un dato al proceso i.

dato=comm.recv(source=i,tag=11)
Recibe un dato desde el proceso i.

dato_a_recibir = comm.bcast(dato_a_enviar, root=i)
Envía un dato por broadcast, siendo i el ID del proceso que realiza el envío. Los procesos que reciben el dato (incluyendo el proceso root) lo almacenan en la variable “dato_a_recibir”.

data = comm.scatter(arreglo, root=i)
Esparce datos desde el proceso i al resto de los procesos. “arreglo” es un array que posee un dato para cada proceso (incluyendo al proceso root). Los datos a esparcir pueden ser variables, cadenas de texto, un list, lists anidados, etc. Notar que la cantidad de elementos de “arreglo” debe ser igual a la cantidad de procesos.

arreglo = comm.gather(data,root=i)
Recoge datos desde todos los procesos hacia un proceso. i es el proceso que recoge los datos. “data” contiene el dato que cada proceso enviará al proceso root (puede ser una variable, una cadena de caracteres, un list, list anidados, etc.). arreglo será una lista en la cual se almacenarán los datos recibidos.

Anexo 3: Medicion de tiempos en C++

3.1 Medir tiempos de ejecución con C++

Existen varios métodos. A continuación se muestra uno. Este método mide el tiempo entre dos ejecuciones de la instrucción ***gettimeofday***. Debe incluir la librería ***sys/time.h***:

```
timeval time1,time2;
gettimeofday(&time1,NULL);
 ……
 (Código cuyo tiempo de ejecución se desea medir)
 ……
gettimeofday(&time2,NULL);
cout << "Tiempo ejecución: " << double(time2.tv_sec - time1.tv_sec) +
+ double(time2.tv_usec-time1.tv_usec)/1000000 << endl;
```

(Las últimas dos líneas son una misma instrucción).

Anexo 4: Repaso de aritmética de los números primos

Un número primo es aquel que solo es divisible por 1 y por si mismo. Todo número puede factorizarse en números primos (a este enunciado se lo llama teorema fundamental del álgebra). Por ejemplo, los factores de 20 son 2, 2 y 5, ya que 20=2*2*5. Por lo tanto, 20 no es un número primo.
0 y 1 no son considerados primos.

Dificultades para resolver el ejercicio 4:
No existe un algoritmo simple y rápido para detectar si un número es primo. La única forma forma de saber si un número K es primo, es dividirlo por ***todos los números primos*** menores o iguales a $\sqrt{K}$, y verificar que el resto de la división sea diferente de cero para todas las divisiones.
Encontrar un algoritmo eficiente y confiable que resuelva este problema es un problema abierto hoy día.

Un algoritmo simple para hallar los números primos:
Para hallar todos los números primos entre 0 y N, propondremos un algoritmo simple que sigue tres pasos:
1) Por cada proceso, busca los números primos entre 0 y $N^{1/2}$. Almacenarlos en un vector auxiliar.
2) Cada proceso busca números primos en una parte del intervalo de números naturales entre 2 y N probando si los números impares “i” son divisibles o no por alguno de los *números primos* entre 2 y $N^{1/2}$ previamente encontrados (notar que los números pares no son primos).

Nota: este algoritmo es ineficiente por varias razones, entre estas: Los procesos que trabajen con números mayores tendrán mayor carga de trabajo.