Arquitecturas Distribuidas
Trabajo Práctico Nº3
Año 2022
Tema: Paralelismo a nivel de procesos usando MPI

Introducción

En este trabajo práctico se pide resolver problemas paralelizables a nivel de procesos empleando alguna implementación abierta de MPI (OpenMPI, MPICH o MPI4Py) 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. En el Anexo 1 se provee información sobre las primitivas básicas de MPI tanto en C++ como Python.

Cada estudiante o grupo de estudiantes deberán configurar al menos dos computadoras para que formen parte de un cluster de mayor tamaño, empleando las computadoras del laboratorio de informática. En el Anexo 2 se provee información de cómo preparar las computadoras para que sean parte de un cluster, instalar y configurar el middleware necesario y probar su funcionamiento.

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. Se sugiere que las carpetas tengan la ruta **/home/mpiuser/nombre_carpeta**.

En todos los ejercicios se pide obtener datos sobre tiempos de ejecución para calcular speedups y visualizar porcentajes de uso de cada núcleo de la computadora local. En el Anexo 3 se provee información sobre cómo medir tiempos con C++ y Python. En el Trabajo práctico N°1 se provee un repaso de C++ y un repaso de aritmética (puede ser necesario para algunos ejercicios).

Puede utilizar C++ o Phyton. Con C++ obtendrá menores tiempos de ejecución. Con Python el tiempo de desarrollo será menor y más simple.

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: 6-8 horas (3-4 clases).

Aprobación
- Mostrar en clases el funcionamiento de los programas escritos.
- Elaborar un informe indicando las características del cluster en la 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++.
  - OpenMPI o MPICH para C++ (en el Anexo 2 se proveen instrucciones de instalación).
  - En caso de utilizar Python: Python versión 3 y librería MPI4PY (en el Anexo 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 0 (Ejercicio Introductorio):
a) Configure dos computadoras para que puedan formar parte de un cluster de mayor tamaño. Instale el software necesario para usar MPI sobre C++ y Python (si prefiere utilizar C++, igual instale las herramientas necesarias para trabajar con Python, para permitir a otros estudiantes que elijan trabajar con Python puedan usar su computadora). Todas las computadoras que formen parte del cluster deberan ser configuradas iguales, se sugiere realizar la siguiente configuración:
- Cree un usuario llamado **mpiuser** con contraseña **Arquitecturas2022** con permisos de administración (que sea parte del grupo **sudo**). Ver Anexo 2.1.
- Configure las computadoras para que puedan comunicarse a través de ssh sin necesidad de ingresar contraseña. Ver Anexo 2.2.

- Cree una carpeta en todas las computadoras del cluster donde copiará el código ejecutable de sus programas. Cada estudiante o grupo deberá tener su propia carpeta en todas las máquinas del cluster. Se sugiere crear y utilizar la carpeta **/home/mpiuser/nombre_carpeta**.
- Permita a sus compañeros crear sus carpetas de trabajo y copiar sus programas a su computadora.
- Instale MPICH o OpenMPI y luego MPI4PY (ver Anexo 2.3)

b) 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.

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.

Ejercicio 1:

Escribir un programa que calcule el logaritmo natural de un número menor 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 doble 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 2:

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/1A7hmHJ60ahudNIw7gRm6DezJIBdLHyfE/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. Buscado por <IP>
el patron 1 aparece 3 veces. Buscado por <IP>
el patron 2 aparece 0 veces. Buscado por <IP>
el patron 3 aparece 0 veces. Buscado por <IP>
el patron 4 aparece 0 veces. Buscado por <IP>
el patron 5 aparece 0 veces. Buscado por <IP>
……..
……..

Donde <IP> es la IP de la máquina que buscó el patrón.
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 creo 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 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 3:

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 $5.71526*10^8$.

Nota: Si trabaja con Python, la biblioteca ***numpy*** puede ser de gran ayuda (puede instalarse como *pip3 install numpy*. Para más información, vaya a https://numpy.org/).

Ejercicio N°4: (ejercicio de desafío)
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*** si trabaja con C++. Con Python no es necesario indicar el tipo de variable.
El problema debe resolverse de modo que el usuario pueda elegir cualquier número de procesos. 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.

Anexo 1: Primitivas de MPI sobre C++ y Python

1.1 Primitivas de MPI en C++

Un listado completo de primitivas puede encontrarse en: https://www.open-mpi.org/doc/v4.1/.
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……
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 machinesfile.txt  ./nombre_ejecutable.out
```

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 machinefile.txt). Si la aplicación se ejecuta 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 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.
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.

1.2 Primitivas para usar MPI en Python

Un listado completo de primitivas puede encontrarse en:
https://mpi4py.readthedocs.io/en/stable/
A continuación se provee un resumen de las 5 más utilizadas.

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 machinesfile.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 machinefile.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 2: Instalación de OpenMPI o MPICH y MPI4PY

2.1 Creación de usuarios en Linux

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.

Creación de usuarios a través de la terminal con permiso de administrador

Para crear un nuevo usuario de Linux 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 **Arquitecturas2022** 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.

Creación de usuarios desde la interfaz gráfica

Seguir instrucciones en https://help.ubuntu.com/stable/ubuntu-help/user-add.html.es

Configure el usuario como administrador, elija como nombre de usuario **mpiuser** y elija como contraseña **Arquitecturas2022**.

2.2 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, junto con los datos de dichas computadoras, entre ellas, nombre de usuario, IP, etc.
El siguiente paso es enviar el archivo con la clave pública a la computadora con la cual se quiere comunicar, y agregar su computadora al archivo known_hosts de dicha computadora. Esto puede hacerlo con la aplicación ssh-copy-id como:

ssh-copy-id user@ip_host

Donde *user* e *ip_host* son los datos de la computadora a la cual quiere enviar la clave pública.

2.3 Instalación de OpenMPI, MPICH y MPI4Py

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 una interfaz (wrapper) que permite crear programas que utilicen MPI utilizando Python. MPI4Py requiere tener OpenMPI o MPICH instalados correctamente y Python o Python3 (recomendado).

Requisitos previos

Requiere tener instalados:

- g++: Instalado por defecto en Ubuntu y Debian. Puede instalarlo con ***sudo apt install g++***.
- Un servidor y un cliente ssh. Ubuntu y Debian tienen por defecto insalado un cliente ssh y usualmente un servidor (puede instalar un servidor y un cliente de ssh con ***sudo apt install openssh-client openssh-server***).

Instalación de OpenMPI y MPICH

Opción 1 (recomendado):
Para instalar OpenMPI desde los repositorios: "***sudo apt install openmpi-bin"***. Instalado por defecto en la mayoría de las distribuciones de Ubuntu.
Para instalar MPICH desde los repositorios: "***sudo apt install mpich***"

Nota 1: Ubuntu tiene instalado por defecto OpenMPI. OpenMPI posee problemas de incompatibilidad con VMWare que al día de la fecha no han sido resueltos. Por este

motivo, en los laboratorios de la Facultad de Ingeniería, se sugiere trabajar con MPICH.
Nota 2: puede que le pida instalar algunas librerías como *libopenmpi-dev* o *openmpi-common.* En ese caso puede instalarlas con *sudo apt install*.
Nota 3: Para desinstalar un programa, puede usar "***sudo apt remove openmpi-bin***"

Para verificar la instalación ejecute:

mpirun -n 2 hostname (se ejecutará un programa que muestra el nombre de su computadora en dos núcleos 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."

Instalación de MPI4Py
Requiere tener instalado y funcionando correctamente OpenMPI o MPICH y Python3. Para instalar, ejecute: ***python3 -m pip install mpi4py***
Nota: Si no tiene pip instalado, instálelo con "***sudo apt-get install python3-pip***"

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.

2.4 Desactivar el firewall

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

sudo ufw disable

Anexo 3: Medicion de tiempos en C++ y Python3

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).

3.2 Medir tiempos de ejecución con Python3

Existen varios métodos. A continuación se muestra uno. Este método mide el tiempo entre dos ejecuciones del método time(). Debe importar el módulo *time* (import time).

```
start_time=time.time()
…….
(Código cuyo tiempo de ejecución se desea medir)
…….
end_time=time.time()
print("Tiempo total de ejecucion " + str(end_time - start_time))
```

Notar que se utiliza la función str() para transformar una variable numérica en una cadena de caracteres, ya que la función print solo permite concatenar cadena de caracteres.