# UNIVERSIDAD DE TARAPACÁ

## FACULTAD DE INGENIERÍA

Departamento de Ingeniería en Computación e Informática

# INFORME TÉCNICO

# IMPLEMENTACIÓN DE MPI CON PYTHON

**Autores: Andoni Flores Balsebre**

**Ignacio Rojas Jorquera**

**Francisco Venegas Aguilera**

**Curso: E.F.P Alto Desempeño Computacional y Big Data**

**Profesor: Diego Aracena Pizarro**

ARICA, 20 de octubre de 2019

# RESUMEN

El presente informe técnico detalla todo el proceso de implementar una arquitectura compatible con Interfaz por Paso de Mensajes (MPI).

Esto se realizó empleando un nodo maestro externo al servidor y n-nodos workers dentro de un servidor con máquinas virtuales.

Para esta solución se optó por el lenguaje de programación Python dada las ventajas que presenta y su reciente crecimiento en el mercado. Además, la librería utilizada para MPI es llamada MPI4PY.

El documento concluye con pruebas y análisis de resultados obtenidos durante y al finalizar el proceso de implementación.

# CONTENIDO

**I. INTRODUCCIÓN** ..... 5
**II. OBJETIVOS** ..... 6
2.1. Objetivo General ..... 6
2.2. Objetivos Específicos ..... 6
**III. DESARROLLO** ..... 7
3.1. Interfaz de Paso de Mensaje (MPI) ..... 7
3.2. Implementación ..... 7
3.2.1. Arquitectura Beowulf ..... 7
3.2.2. Características recursos utilizados ..... 8
3.2.2. MPI4PY ..... 10
3.2.3. Instalación de MPI4PY ..... 10
3.2.4. Conexión SSH ..... 11
3.2.4.1. Cliente y Servidor SSH ..... 12
3.2.4.1. SSH y claves públicas ..... 12
3.2.5. Ejecutando un programa en Python MPI ..... 15
3.2.5.1. Ejemplo “Helloworld” ..... 15
3.2.5.2. Ejecutando el programa ..... 16
3.2.5.3. Código para pruebas “Phi” ..... 18
3.3. Pruebas y Resultados ..... 23
**IV. CONCLUSIONES** ..... 25
**V. REFERENCIAS BIBLIOGRÁFICAS** ..... 26

# ÍNDICE DE FIGURAS

Figura 1: Arquitectura Beowulf. .......... 7
Figura 2: Nodo Maestro. .......... 8
Figura 3: Switch. .......... 8
Figura 4: Router. .......... 8
Figura 5: Servidor. .......... 9
Figura 6: Arquitectura Beowulf y recursos utilizados. .......... 10
Figura 7: Generando clave pública. .......... 13
Figura 8: Clave pública para los nodos workers. .......... 14
Figura 9: Comprobando funcionamiento clave pública. .......... 14
Figura 10: Código de ejemplo helloworld.py. .......... 15
Figura 11: Contenido del archivo Machinefile - Información de nodos Workers. .......... 16
Figura 12: Ejecución helloworld.py para un solo nodo. .......... 16
Figura 13: Ejecución helloworld.py para n-nodo. .......... 17
Figura 14: Fórmula de Leibniz. .......... 18
Figura 15: Código Python para el cálculo del número Pi. .......... 19
Figura 16: Resultado ejecución programa “phi.py” parte 3. .......... 20
Figura 17: Resultado ejecución programa phi.py - Parte 2. .......... 21
Figura 18: Resultado ejecución programa phi.py - Parte 3. .......... 22
Figura 19: Gráfica de resultados 50 slices. .......... 23
Figura 20: Gráfica de resultados 1280 slices. .......... 24

# ÍNDICE DE TABLAS

Tabla 1: Asignaciones IP. .......... 9
Tabla 2: Resultados 50 slices. .......... 23
Tabla 3: Resultados 128 slices. .......... 24

# I. INTRODUCCIÓN

Los clúster de HPC en la actualidad su gran mayoría se construyen con máquinas físicas. Académicamente nos interesa estudiar HPC en profundidad por lo que es importante estudiar el comportamiento y desempeño de clústeres HPC construidos con máquinas virtuales. Si bien los clúster HPC de máquinas virtuales se utilizan para proteger a las máquinas físicas de fallos de hardware y software, vale la pena estudiar su potencial más allá de su tarea principal.

Es por esto que en este proyecto estudiaremos el desempeño de un clúster HPC enfocado a Big Data, en este documento específicamente veremos la configuración inicial del clúster y además un par de pruebas de desempeño y sus resultados.

# II. OBJETIVOS

## 2.1. Objetivo General

Estudiar e implementar un clúster con PC (Notebook, RaspberryPi, Xbox, PS4, Máquinas Virtuales), servidores reales y virtualizado, utilizando los recursos remotos y los locales en DICI AZUFRE que se tengan disponibles.

## 2.2. Objetivos Específicos

- Seleccionar arquitectura para clúster compatible con MPI.
- Seleccionar Middleware MPI y lenguajes asociados.
- Armar clúster con recursos disponibles empleando máquinas virtuales.
- Utilizar una aplicación MPI con datos masivos.

# III. DESARROLLO

## 3.1. Interfaz de Paso de Mensaje (MPI)

Librería estándar de paso de mensajes para las comunicaciones entre procesos desarrollada por un grupo de académicos y socios industriales para fomentar un uso más generalizado y portabilidad de programas.

## 3.2. Implementación

En esta sección se describirán los pasos a seguir para una implementación de MPI en Python. Dicha implementación será realizada sobre un Sistema Operativo Linux y mediante la versión 3 de Python. Como dato adicional, el equipo cuenta con conexión a internet para una instalación de paquetes de manera más sencilla.

### 3.2.1. Arquitectura Beowulf

Es un clúster masivamente paralelo de altas prestaciones que ejecuta un sistema operativo de libre distribución como Linux, se interconecta mediante una red privada de gran velocidad y sale al mundo exterior por un solo nodo. Generalmente se compone de un grupo de PC's o estaciones de trabajo dedicados a ejecutar tareas que precisan una alta capacidad de cálculo. Utilizando librerías como MPI particionas tareas en paralelo para su comunicación. En la Figura 1 se observa un ejemplo de clúster Beowulf.

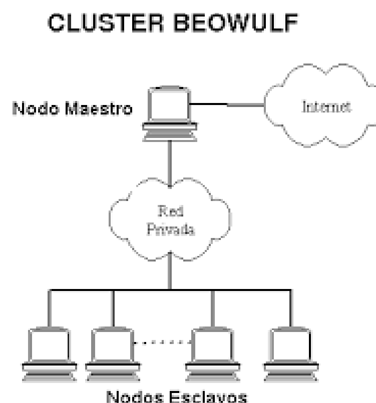



*Figura 1: Arquitectura Beowulf.*

### 3.2.2. Características recursos utilizados

Para el proyecto se configuró en base a la arquitectura Beowulf y los recursos utilizados se describen a continuación.

- **Nodo Maestro:** Notebook Toshiba procesador i5, memoria RAM 8GB y Sistema Operativo Ubuntu 16.04, ver Figura 2.

*Figura 2: Nodo Maestro.*

- **Switch:** D-Link DES-1024D 24 puertos 10/100 Mbps, observar Figura 3.

*Figura 3: Switch.*

- **Router:** D-Link DIR-600 inalámbrico de 150Mbps con 4 puertos de red 10/100 Mbps, observar Figura 4.

*Figura 4: Router.*

- **Servidor 1, 2 y 3:** DELL EMC PowerEdge R430, procesador Intel Xeon E5-2600 v4 con hasta 22 núcleos, RAM de 64 GB, ver Figura 5.

*Figura 5: Servidor.*

Cabe mencionar que los nodos workers serán virtualizados mediante el software hipervisor Citrix XenServer 7.5 (servidor 2 y 3) y XenServer 7.6 (servidor 1). Además, cada recurso cuenta con una dirección IP asociada para realizar las conexiones respectivas utilizando una red privada, las direcciones IP de los recursos utilizados son las siguientes, ver Tabla 1.

*Tabla 1: Asignaciones IP.*

| Recurso | IP |
|---|---|
| Nodo Maestro | 10.0.0.100 |
| Router | 10.0.0.1 y 146.83.102.X |
| PC 1 | 10.0.0.35 |
| PC 2 | 10.0.0.21 |
| PC 3 | 10.0.0.15 |
| Servidor 1 | 10.0.0.30 |
| Servidor 2 | 10.0.0.20 |
| Servidor 3 | 10.0.0.40 |
| Nodo Esclavo 1 Servidor 1 | 10.0.0.31 |
| Nodo Esclavo 2 Servidor 1 | 10.0.0.32 |
| Nodo Esclavo 3 Servidor 1 | 10.0.0.33 |
| Nodo Esclavo 4 Servidor 1 | 10.0.0.34 |
| Nodo Esclavo 1 Servidor 2 | 10.0.0.22 |
| Nodo Esclavo 2 Servidor 2 | 10.0.0.23 |
| Nodo Esclavo 3 Servidor 2 | 10.0.0.23 |
| Nodo Esclavo 1 Servidor 3 | 10.0.0.41 |
| Nodo Esclavo 2 Servidor 3 | 10.0.0.42 |
| Nodo Esclavo 3 Servidor 3 | 10.0.0.43 |

En la Figura 6 se observa la arquitectura empleada con los recursos utilizados.

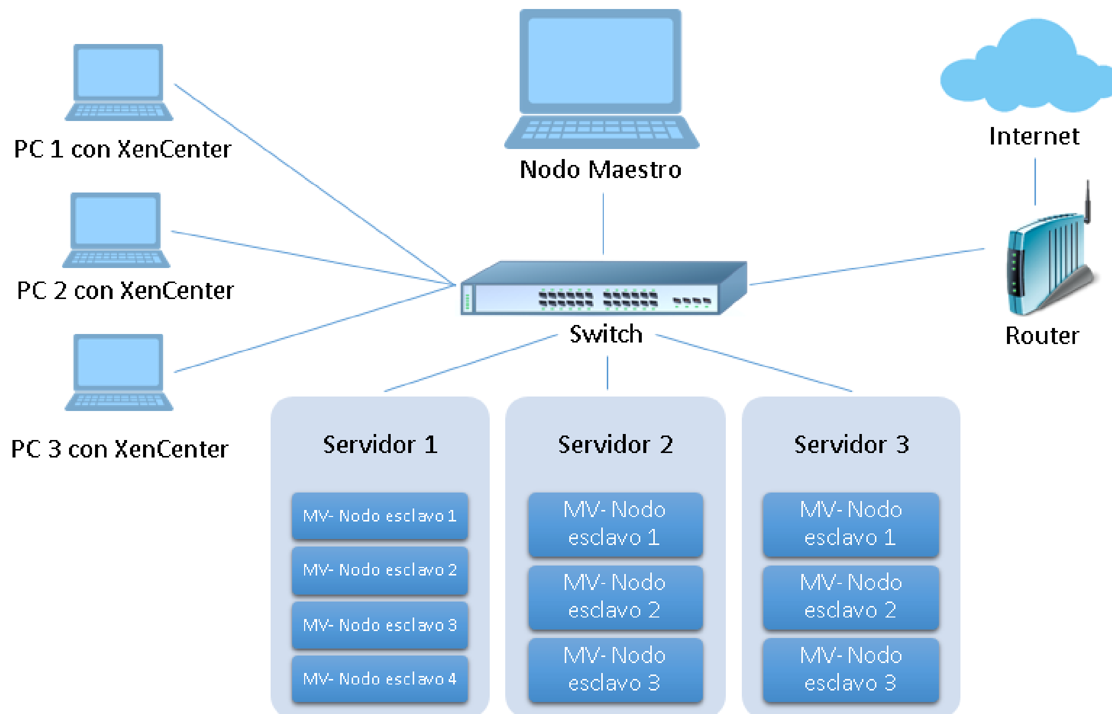



*Figura 6: Arquitectura Beowulf y recursos utilizados.*

### 3.2.2. MPI4PY

MPI4PY es una librería que permite a Python emplear métodos del estándar de Interfaz de Paso de Mensajes (MPI). Esta se implementa por encima de la especificación MPI-1/2/3 y expone una API que se basa en los enlaces MPI-2 C ++ estándar.

### 3.2.3. Instalación de MPI4PY

Como paso previo a la instalación de MPI4PY, es necesario instalar la librería de OpenMPI en nuestro sistema operativo Linux. Para aquello, se debe ejecutar el siguiente comando en la terminal del sistema (ingresar sin los “>>” iniciales):

***>> sudo apt-get install libopenmpi-dev***

Este paso es necesario debido a que se instalan librerías relacionadas a OpenMPI y que MPI4PY utilizará recurrentemente. Una vez terminada la instalación de esa librería, se procede con la instalación del paquete MPI4PY mediante el uso del Administrador de Paquetes PIP en su versión 3:

***>> pip3 install mpi4py***

En caso de no contar con PIP en su equipo, este puede instalarse mediante el siguiente comando:

***>> sudo apt-get install python3-pip***

La instalación de estas librerías es obligatoria para cada nodo, tanto maestro como workers.

### 3.2.4. Conexión SSH

El paso de mensajes es el objetivo de esta actividad y es realizado mediante conexiones SSH, por lo que requerimos de clientes y servidores SSH para aquello.

El enfoque de esta solución es que el nodo maestro invoca a todos los servidores que ejecutarán archivos Python a través de MPI. Para esto es necesario que el maestro pueda conectarse con cada uno de los trabajadores (o "workers"), es decir, se basa en un esquema de invocación lineal, donde el maestro posee todas las conexiones ssh con los trabajadores.

Cabe destacar que MPI4PY por defecto tiene un esquema de invocación basado en una estructura de árbol, esto se refiere a que el maestro inicializa el sistema, ejecutando dos servidores que corresponden a trabajadores y luego estos dos trabajadores invocan a otros 4 trabajadores y así sucesivamente. Se decidió por un esquema de invocación lineal por sobre una estructura de árbol, debido a que en la estructura de invocación de árbol es necesario que los trabajadores que invoquen nuevos servidores puedan conectarse a través de SSH con los trabajadores que invocaran (esto incluye generar y conocer las claves públicas) por lo que en un proyecto de gran escala esto no sería muy eficiente. Sin embargo, sí se puede automatizar la creación de estas claves públicas el esquema de árbol si se pudiese considerar.

### 3.2.4.1. Cliente y Servidor SSH

La conexión entre el host "maestro" y los hosts "workers" se realiza mediante conexión SSH. Por lo que es necesario que el host maestro cuente con un cliente SSH y los equipos "workers" deberán contar con un servidor SSH.

Los clientes SSH actualmente ya se encuentran dentro de todo sistema operativo Linux, no así los servidores SSH. Para instalar un servidor SSH (en este caso OpenSSH), se realizará en un nodo worker mediante el siguiente comando:

***>> sudo apt-get install openssh-server***

Una vez instalado y funcionando el servidor SSH (verificable con el comando "***systemctl ssh***"), se establecerá una conexión SSH desde el maestro hacia un nodo worker. En el nodo maestro, deberá ejecutar el siguiente comando:

***>> ssh <username@ip-worker>***

, donde *username* es el nombre host de la máquina y el campo *ip-worker* es la dirección IP de dicha máquina. Luego, como en toda conexión SSH, se le pedirá la contraseña.

Si las credenciales fueron correctas entonces la conexión SSH terminará con éxito. Además, hay que mencionar que este proceso deberá repetirse para cada nodo worker de tal forma que finalmente exista un nodo maestro con cliente SSH y n-nodos servidor con servidores SSH para cada uno de ellos.

### *3.2.4.1. SSH y claves públicas*

Como se habrá dado cuenta, para establecer una conexión SSH, se solicita una contraseña. Esto está bien, pero tiene sus consecuencias si estamos hablando de una n-cantidad de nodos worker. En este caso, para cada nodo, tendría que ingresar la contraseña cada vez que se establezca y restablezca una conexión SSH. Esto es totalmente inviable.

Ya explicado el problema se plantea la solución, hacer uso de una clave pública, la cual será generada por el nodo maestro y compartida hacia los nodos worker. Situándonos en el nodo maestro, generamos una clave pública mediante el siguiente comando:

***>> ssh-keygen***

Se preguntará acerca del nombre del archivo y de su localización en el equipo, además de una ***passphrase*** o frase secreta que deberá introducirse dos veces para generar una clave RSA (ver Figura 7). Como resultado se obtendrán 2 archivos en su respectiva: ***id_rsa.pub - id_rsa***.

```
aici@aici-TECRA-Z40-C:~$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/aici/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/aici/.ssh/id_rsa.
Your public key has been saved in /home/aici/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:x4MEGPCwDb7Doz56o3oOUS7rNFf9eoYQA9ty2ulHBd0 aici@aici-TECRA-Z40-C
The key's randomart image is:
+---[RSA 2048]----+
|  +..o. . .      |
| . B.  o . E     |
|  + *   o        |
| + + =.. +       |
|o * =.+.S +      |
| = +.+ ... .     |
|+o .. o ..       |
|+o*  . o.o       |
|*Bo.  ..o        |
+----[SHA256]-----+
aici@aici-TECRA-Z40-C:~$ ssh-copy-id vm1@10.0.0.31
```

*Figura 7: Generando clave pública.*

Lo siguiente será llevar la clave pública hacia los nodos workers. Esto mediante el comando **ssh-copy-id**, el cual tomará la clave pública en la ubicación por defecto (la generada con el comando previo) y será llevada hacia el host worker. El comando es el siguiente:

***>> ssh-copy-id <username@ip-worker>***

Ingresado el comando se solicitará una contraseña. Si el proceso terminó correctamente, entonces habremos copiado la clave pública correctamente en un nodo worker (ver Figura 8). Además, existen situaciones en las que es necesario forzar la copia añadiéndole el parámetro ***-f*** al comando. Hay que recordar que se deberá iterar este paso para cada nodo worker.

```
aici@aici-TECRA-Z40-C:~/.ssh$ ssh-copy-id -f basededatos@10.0.0.22
basededatos@10.0.0.22's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'basededatos@10.0.0.22'"
and check to make sure that only the key(s) you wanted were added.

aici@aici-TECRA-Z40-C:~/.ssh$ ssh-copy-id -f ubuntuhost1@10.0.0.43
The authenticity of host '10.0.0.43 (10.0.0.43)' can't be established.
ECDSA key fingerprint is SHA256:75fsTQf6Eh+cSzn4fKe6L2yNtuEG21pvnd2C9UjzLNg.
Are you sure you want to continue connecting (yes/no)? yes
ubuntuhost1@10.0.0.43's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'ubuntuhost1@10.0.0.43'"
and check to make sure that only the key(s) you wanted were added.
```

*Figura 8: Clave pública para los nodos workers.*

Para probar el funcionamiento de la clave pública se recomienda establecer una conexión SSH de prueba, donde solo bastará con ingresar el comando de conexión SSH respectivo para que la conexión se realice manera exitosa, sin necesidad de ingresar una contraseña, como se presenta en la Figura 9.

```
aici@aici-TECRA-Z40-C:~$ ssh vm1@10.0.0.31
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.15.0-45-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

269 packages can be updated.
201 updates are security updates.

Last login: Thu Oct 10 11:55:05 2019 from 10.0.0.100
vm1@esclavo01:~$
```

*Figura 9: Comprobando funcionamiento clave pública.*

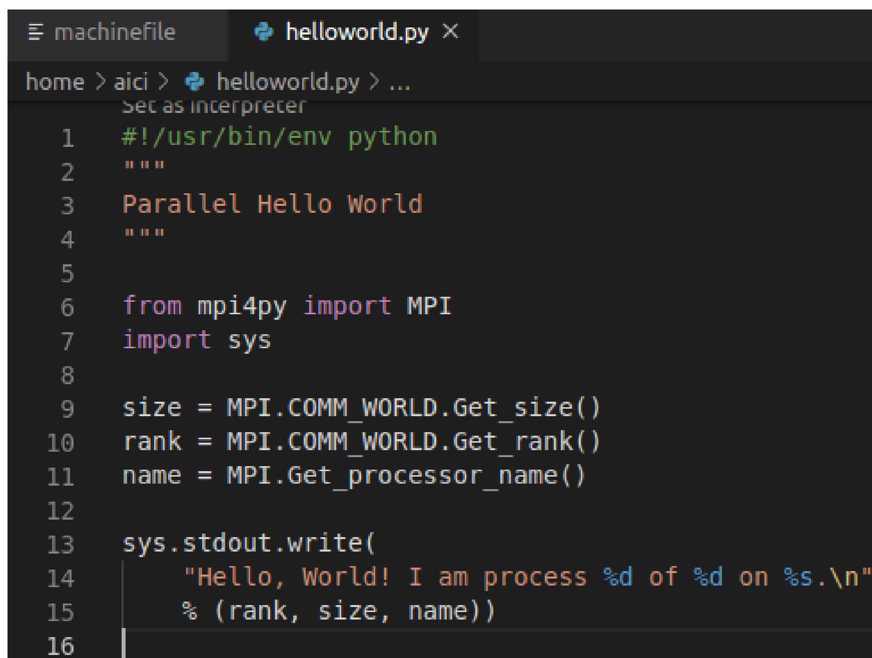
### 3.2.5. Ejecutando un programa en Python MPI

En esta sección, nos enfocaremos en describir los pasos para iniciar un código de pruebas que funcione con Python MPI desde la terminal.

#### *3.2.5.1. Ejemplo "Helloworld"*

La Figura 10 muestra una implementación del clásico "Helloworld" en Python, pero con la característica de ejecutarse mediante MPI, identificando cada nodo. Este código de prueba se encuentra disponible en el siguiente enlace:

https://bitbucket.org/mpi4py/mpi4py/downloads/




```python
#!/usr/bin/env python
"""
Parallel Hello World
"""

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

*Figura 10: Código de ejemplo helloworld.py.*

La tarea que realiza el código es imprimir en pantalla el número de proceso y la máquina remota que está ejecutando la rutina. Pero, antes que nada, es necesario llevar dicho código Python a cada uno de los nodos workers. Esto se puede realizar de manera sencilla mediante el Secure Copy (**SCP**), que es un medio de transferencia segura de archivos entre un host local y uno remoto, usando el protocolo Secure Shell (SSH). Situándonos en el nodo maestro, ejecutaremos el siguiente comando:

***>> scp <username@ip-worker> <directorio remoto para la ubicación> <nombre del archivo a copiar>***

Además, se debe contar con un archivo de texto plano que contenga los parámetros de conexión SSH para cada nodo worker con el que se quiera trabajar. Un ejemplo del contenido de este archivo (llamado **machinefile**) se puede apreciar en la Figura 11.

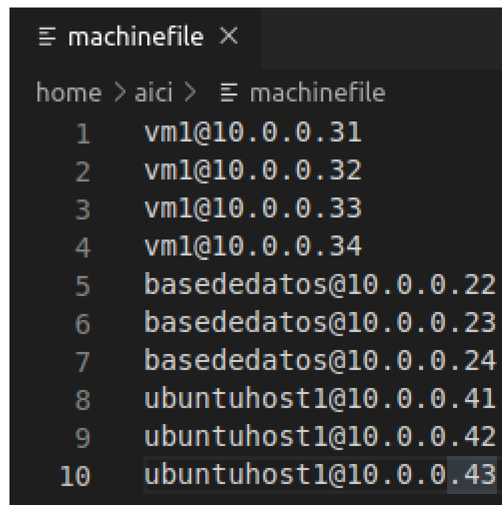



*Figura 11: Contenido del archivo Machinefile - Información de nodos Workers.*

### *3.2.5.2. Ejecutando el programa*

Para iniciar el programa se debe ingresar el siguiente comando en el nodo maestro:

***>> mpirun.openmpi -np <# procesos> -machinefile <ruta del archivo machinefile> <directorio remoto para la ubicación> <nombre del archivo a copiar>***

, donde **mpirun.ompenmpi** o **mpiexec** indica que es un programa MPI, **-n** o **-np** indica el número de procesos, -**machinefile** ***<ruta del archivo>*** indica la ruta del archivo "**machinefile**" en la máquina maestra, y **python3** ***<código py>*** indica que es una rutina de Python 3 acompañado del archivo a ejecutar. La Figura 12 muestra la ejecución del Holamundo.py" mediante MPI de manera local (solo en un nodo).

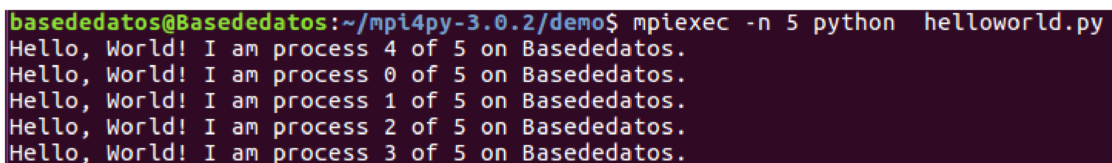



*Figura 12: Ejecución helloworld.py para un solo nodo.*

La Figura 13 muestra la ejecución del “holamundo.py” mediante MPI de manera remota (*n-nodos*).

```
Hello, World! I am process 31 of 64 on esclavo06.
Hello, World! I am process 61 of 64 on esclavo09.
Hello, World! I am process 41 of 64 on esclavo06.
Hello, World! I am process 17 of 64 on esclavo04.
Hello, World! I am process 34 of 64 on esclavo06.
Hello, World! I am process 62 of 64 on esclavo09.
Hello, World! I am process 33 of 64 on esclavo06.
Hello, World! I am process 63 of 64 on esclavo09.
Hello, World! I am process 36 of 64 on esclavo06.
Hello, World! I am process 39 of 64 on esclavo06.
Hello, World! I am process 60 of 64 on esclavo09.
Hello, World! I am process 4 of 64 on esclavo01.
Hello, World! I am process 18 of 64 on esclavo04.
Hello, World! I am process 13 of 64 on esclavo03.
Hello, World! I am process 42 of 64 on esclavo10.
Hello, World! I am process 47 of 64 on esclavo10.
Hello, World! I am process 48 of 64 on esclavo10.
Hello, World! I am process 49 of 64 on esclavo10.
Hello, World! I am process 51 of 64 on esclavo10.
Hello, World! I am process 46 of 64 on esclavo10.
Hello, World! I am process 50 of 64 on esclavo10.
Hello, World! I am process 45 of 64 on esclavo10.
Hello, World! I am process 43 of 64 on esclavo10.
Hello, World! I am process 3 of 64 on esclavo01.
Hello, World! I am process 44 of 64 on esclavo10.
Hello, World! I am process 21 of 64 on esclavo05.
Hello, World! I am process 22 of 64 on esclavo05.
Hello, World! I am process 23 of 64 on esclavo05.
Hello, World! I am process 24 of 64 on esclavo05.
Hello, World! I am process 27 of 64 on esclavo05.
Hello, World! I am process 30 of 64 on esclavo05.
Hello, World! I am process 20 of 64 on esclavo05.
Hello, World! I am process 25 of 64 on esclavo05.
Hello, World! I am process 29 of 64 on esclavo05.
Hello, World! I am process 28 of 64 on esclavo05.
Hello, World! I am process 37 of 64 on esclavo06.
Hello, World! I am process 26 of 64 on esclavo05.
Hello, World! I am process 9 of 64 on esclavo02.
Hello, World! I am process 8 of 64 on esclavo02.
Hello, World! I am process 7 of 64 on esclavo02.
Hello, World! I am process 5 of 64 on esclavo02.
Hello, World! I am process 6 of 64 on esclavo02.
aici@aici-TECRA-Z40-C:~$
```

*Figura 13: Ejecución helloworld.py para n-nodo.*

Como se puede apreciar en la figura previa, este ejemplo muestra el número de proceso, el total (en este caso 64 dado que fue ingresado con **-np 64**) y el nombre del equipo que atendió dicho proceso.

#### *3.2.5.3. Código para pruebas "Phi"*

La Figura 15 muestra una implementación del código de Phi, mediante la fórmula de Leibniz la cual podemos ver en la Figura 14. Como podemos observar la fórmula de Leibniz es una serie de sumas y restas que equivalen a π/4, es por esto por lo que en la línea 44 del código Python (ver figura 15) se imprime el resultado multiplicado por cuatro.

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}.$$

*Figura 14: Fórmula de Leibniz.*

Además, al ser un cálculo de π basado en series es más fácil dividirlo en partes iguales para que los trabajadores lo procesen paralelamente. Dicho código fue obtenido desde el siguiente enlace y modificado para mostrar el tiempo de ejecución del programa:

https://gist.github.com/jcchurch/930276

```python
import time
from mpi4py import MPI

start_time = time.time()
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

slice_size = 1000000
total_slices = 128

# This is the master node.
if rank == 0:
    pi = 0
    slice = 0
    process = 1

    print (size)

    # Send the first batch of processes to the nodes.
    while process < size and slice < total_slices:
        comm.send(slice, dest=process, tag=1)
        print ("Sending slice",slice,"to process",process)
        slice += 1
        process += 1

    # Wait for the data to come back
    received_processes = 0
    while received_processes < total_slices:
        pi += comm.recv(source=MPI.ANY_SOURCE, tag=1)
        process = comm.recv(source=MPI.ANY_SOURCE, tag=2)
        print ("Recieved data from process", process)
        received_processes += 1

        if slice < total_slices:
            comm.send(slice, dest=process, tag=1)
            print ("Sending slice",slice,"to process",process)
            slice += 1

    # Send the shutdown signal
    for process in range(1,size):
        comm.send(-1, dest=process, tag=1)

    print ("Pi is ", 4.0 * pi)
    print("Tiempo de Ejecución: ", time.time() - start_time)

# These are the slave nodes, where rank > 0. They do the real work
else:
    while True:
        start = comm.recv(source=0, tag=1)
        if start == -1: break

        i = 0
        slice_value = 0
        while i < slice_size:
            if i%2 == 0:
                slice_value += 1.0 / (2*(start*slice_size+i)+1)
            else:
                slice_value -= 1.0 / (2*(start*slice_size+i)+1)
            i += 1

        comm.send(slice_value, dest=0, tag=1)
        comm.send(rank, dest=0, tag=2)
```

*Figura 15: Código Python para el cálculo del número Pi.*

Hay que recordar que el código deberá ser copiado a cada nodo worker mediante el comando SCP. Una vez hecho esto, se procede a ingresar el mismo comando previo, pero ahora ejecutando el código “phi.py”. La Figura 16, 17 y 18 muestran el resultado de la ejecución del programa:

```
aici@aici-TECRA-Z40-C:~$ mpirun --mca plm_rsh_no_tree_spawn 1 -np 64 -machinefile /home/aici/machinefile python3 phi.py
64
Sending slice 0 to process 1
Sending slice 1 to process 2
Sending slice 2 to process 3
Sending slice 3 to process 4
Sending slice 4 to process 5
Sending slice 5 to process 6
Sending slice 6 to process 7
Sending slice 7 to process 8
Sending slice 8 to process 9
Sending slice 9 to process 10
Sending slice 10 to process 11
Sending slice 11 to process 12
Sending slice 12 to process 13
Sending slice 13 to process 14
Sending slice 14 to process 15
Sending slice 15 to process 16
Sending slice 16 to process 17
Sending slice 17 to process 18
```

*Figura 16: Resultado ejecución programa “phi.py” parte 3.*

Los resultados muestran los workers a los que el nodo maestro envía los “slices”, cuando estos son retornados y el resultado final, acompañado del tiempo de ejecución.

```
Recieved data from process 1
Sending slice 65 to process 1
Recieved data from process 4
Sending slice 66 to process 4
Recieved data from process 10
Sending slice 67 to process 10
Recieved data from process 55
Sending slice 68 to process 55
Recieved data from process 62
Sending slice 69 to process 62
Recieved data from process 11
Sending slice 70 to process 11
Recieved data from process 63
Sending slice 71 to process 63
Recieved data from process 53
Sending slice 72 to process 53
Recieved data from process 56
Sending slice 73 to process 56
Recieved data from process 59
Sending slice 74 to process 59
Recieved data from process 58
Sending slice 75 to process 58
Recieved data from process 13
Sending slice 76 to process 13
Recieved data from process 61
Sending slice 77 to process 61
Recieved data from process 12
Sending slice 78 to process 12
Recieved data from process 9
Sending slice 79 to process 9
Recieved data from process 3
Sending slice 80 to process 3
Recieved data from process 18
```

*Figura 17: Resultado ejecución programa phi.py - Parte 2.*

```
Recieved data from process 61
Recieved data from process 60
Recieved data from process 11
Recieved data from process 54
Recieved data from process 10
Recieved data from process 4
Recieved data from process 9
Recieved data from process 3
Recieved data from process 19
Recieved data from process 55
Recieved data from process 8
Recieved data from process 17
Recieved data from process 52
Recieved data from process 6
Recieved data from process 57
Recieved data from process 18
Recieved data from process 7
Recieved data from process 12
Recieved data from process 16
Recieved data from process 15
Recieved data from process 1
Recieved data from process 20
Recieved data from process 28
Recieved data from process 44
Recieved data from process 46
Recieved data from process 35
Recieved data from process 24
Recieved data from process 34
Recieved data from process 45
Recieved data from process 27
Recieved data from process 29
Recieved data from process 43
Recieved data from process 22
Recieved data from process 23
Recieved data from process 38
Recieved data from process 42
Recieved data from process 21
Recieved data from process 30
Recieved data from process 37
Recieved data from process 25
Pi is  3.141592645777273
Tiempo de Ejecución:  4.153278827667236
aici@aici-TECRA-Z40-C:~$
```

*Figura 18: Resultado ejecución programa phi.py - Parte 3.*

## 3.3. Pruebas y Resultados

Estos son los resultados de las pruebas realizadas con el código "Phi" la cual se divide en 50 como muestra la Tabla 2 y 128 slices (cortes que realiza el programa para el cálculo) en la Tabla 3, en cantidad de procesos; cada uno con tres ejecuciones y el promedio de estos resultados. Además, las Figuras 17 y 18 plasman de manera gráfica los resultados obtenidos.

*Tabla 2: Resultados 50 slices.*

| Para 50 slice | | | | |
|---|---|---|---|---|
| **Procesos** | **Resultado 1** | **Resultado 2** | **Resultado 3** | **Promedio** |
| 4 | 9.026 | 9.26 | 8.84 | 9.04 |
| 8 | 5.64 | 6.6 | 6.58 | 6.27 |
| 16 | 3.71 | 3.82 | 3.74 | 3.76 |
| 32 | 2.25 | 2.34 | 2.33 | 2.31 |
| 64 | 2.62 | 2.9 | 2.7 | 2.74 |
| 128 | 4.38 | 4.27 | 4.34 | 4.33 |

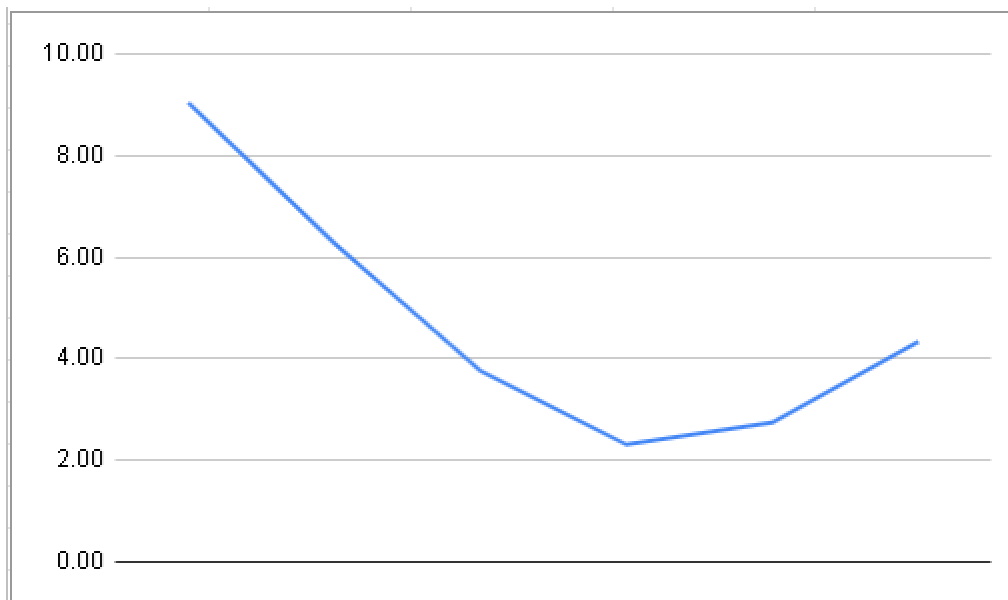



*Figura 19: Gráfica de resultados 50 slices.*

*Tabla 3: Resultados 128 slices.*

| Para 128 slice | | | | |
|---|---|---|---|---|
| Procesos | Resultado 1 | Resultado 2 | Resultado 3 | Promedio |
| 4 | 24.45 | 25.9 | 22.75 | 24.37 |
| 8 | 12.29 | 12.89 | 11.92 | 12.37 |
| 16 | 9 | 8.86 | 9.19 | 9.02 |
| 32 | 4.7 | 4.89 | 4.86 | 4.82 |
| 64 | 4.47 | 4.38 | 4.28 | 4.38 |
| 128 | 5.28 | 5.1 | 5.42 | 5.27 |

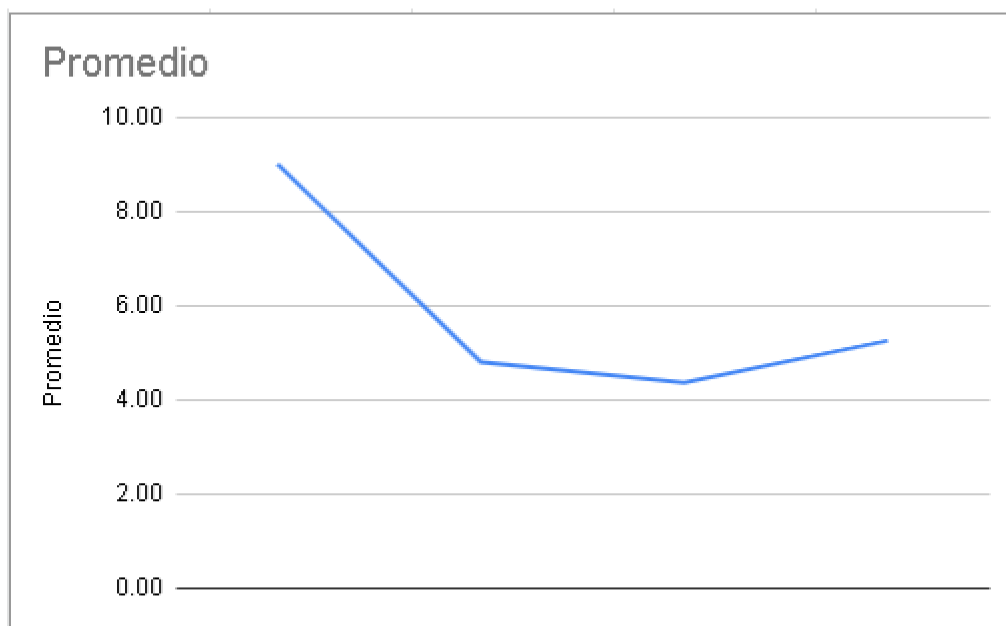



*Figura 20: Gráfica de resultados 1280 slices.*

# IV. CONCLUSIONES

Tras la realización del taller se concluye los siguientes puntos:

- Resaltar la poca documentación existente acerca de una implementación de MPI utilizando la librería MPI4PY.
- Una implementación mediante máquinas virtuales abarata mucho los costos de implementación de un clúster para ciertas prestaciones.
- En la arquitectura Beowulf los nodos que conforman el clúster se encuentran dedicados únicamente a tareas del clúster.
- Los nodos workers físicamente pueden ser diferentes tipos de recursos que tengan capacidad de procesamiento, pero a la hora de virtualizar ganamos características importantes como mejor tiempo de conexión y seguridad.
- La arquitectura Beowulf la ser implementada con nodos virtualizados se obtiene una escalabilidad muy alta, siempre que los recursos físicos lo permitan.
- Para la virtualización de nodos se hace muy flexible y adaptable a la solución de nuestras necesidades, además el ahorro de estructura física, monetaria y tiempo se hace importante a para la administración y mantención de la estructura.
- El cálculo de phi es muy versátil a la hora de medir rendimiento. Esto sirvió al equipo de trabajo para medir, conocer y comparar resultados de rendimiento finales basados en la cantidad de slices y cantidad de procesos.
- Respecto al último código, hay que destacar el comportamiento (en ambos slices) de rendimiento resultante, donde se aprecia que, un mayor número de procesos representa un mayor aumento en el rendimiento, lo que se traduce en tiempos menores de ejecución. Sin embargo, cuando el número sobrepasa los 64 procesos, entonces la cantidad de máquinas virtuales no puede atender en una primera iteración dicha cantidad, por lo que deberá esperar a que los procesos que son atendidos terminen, traduciéndose en mayor tiempo para esperas sumado a lo que se pierde en comunicaciones (que en este caso fueron ínfimos, pero existen). Por lo anterior podemos concluir que existe una configuración de números de procesos y slices las cuales podrían minimizar el tiempo de ejecución, esto es sumamente importante al estudiar los desempeños de sistemas, especialmente el desempeño de un clúster HPC.

# V. REFERENCIAS BIBLIOGRÁFICAS

[1] F. Segarra, «Hostinger,» 16 05 2017. [En línea]. Available: https://www.hostinger.es/tutoriales/llaves-ssh. [Último acceso: 20 10 2019].

[2] L. Dalcin, «MPI for Python,» 11 07 2019. [En línea]. Available: https://mpi4py.readthedocs.io/en/stable/. [Último acceso: 20 10 2019].

[3] «MIP4PY Bitbucket,» [En línea]. Available: https://bitbucket.org/mpi4py/mpi4py/downloads/. [Último acceso: 20 10 2019].

[4] J. Church, «GitHub - MPI4PY Phi,» [En línea]. Available: https://gist.github.com/jcchurch/930276. [Último acceso: 20 10 2019].