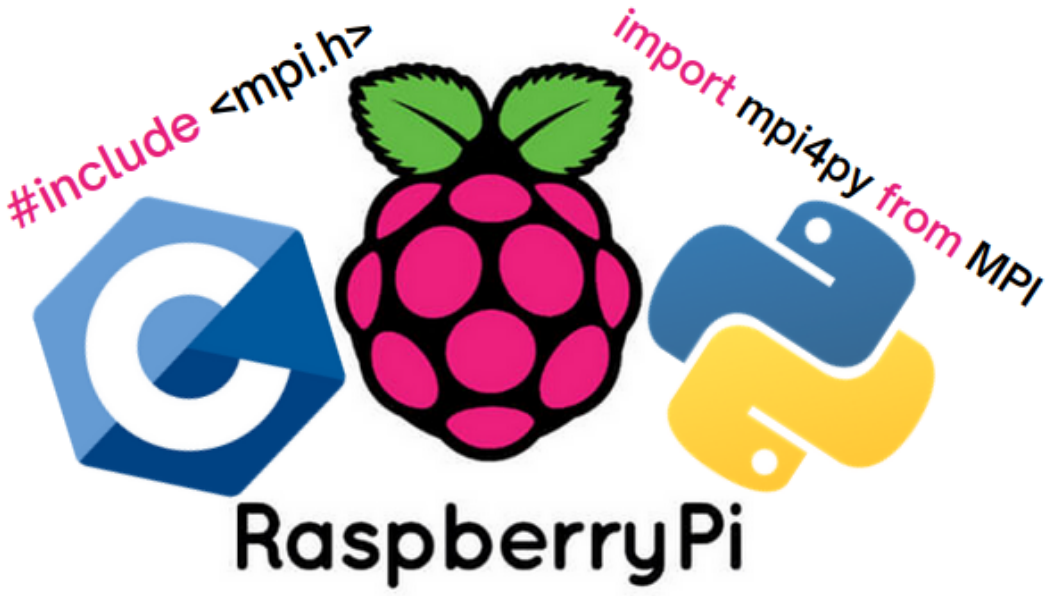

Grado en Ingeniería Informática
Arquitecturas Paralelas




**Autores:**

| | |
|---|---|
| Álvaro Palacios Criado | p02pacra@uco.es |
| Adolfo Fernandez Gil | i92fegia@uco.es |
| Rafael Carlos Díaz Mata | i02dimar@uco.es |

**Profesores**

Ezequiel Herruzo Gómez

Córdoba, 8 de junio de 2023

# Índice

# Índice de figuras

# 1. Introducción

En este proyecto final de la asignatura de Arquitecturas Paralelas (Repositorio en GitHub[1]), vamos a tratar diferentes temas vistos en la asignatura y a lo largo de la carrera. El proyecto consiste en realizar un cluster [2] de con tres dispositivos Raspberry Pi, formando una red IOT solamente en local, para de esta manera poder ejecutar las prácticas de la asignatura. Algunos aspectos de los que vamos a tratar en el proyecto son:

- **Redes**: configuración de red y hosts para poder formar una red local y tener conexión por SSH. Además montaremos un NFS (Network File System).
- **Programación**: programaremos usando librerías de C como en la prácticas de clase como OpenMP y OpenMPI. Además implementaremos estos mismos programas de clase, pero en Python.
- **Administración de Sistemas**: realizaremos configuraciones de varios archivos del entorno UNIX y sacaremos datos de ejecución de los programas para mantener una monitorización de los recursos.
- **Arquitecturas Paralelas**: tocaremos de cerca la caché de estos dispositivos, realizando pruebas con dos vectores de ejecución para la multiplicación de matrices, uno optimizado **vector i,k,j** y otro sin optimizar (**vector i,j,k**).

# 2. Hardware

En este apartado vamos a hablar sobre los elementos hardware de nuestro cluster. Los principales elementos son los dispositivos Raspberry Pi [3], comenzaremos haciendo una introducción a ellos.

### ¿Que es una Raspberry Pi?

La descripción se centra en una placa base que brinda soporte a diversos componentes de un ordenador, como un procesador ARM de hasta 1500 MHz, un chip gráfico y una memoria RAM de hasta 8 GB. Además, presenta múltiples posibilidades adicionales.

Gracias a sus puertos y entradas, se habilita la conexión de dispositivos periféricos, como una pantalla táctil, un teclado e incluso un televisor. Incorpora un procesador gráfico VideoCore IV, lo que permite la reproducción de vídeos, incluso en alta definición.

En términos de conectividad, facilita la conexión a la red mediante un puerto Ethernet, y algunos modelos ofrecen opciones de conexión Wifi y Bluetooth.

Asimismo, incluye una ranura SD que posibilita la instalación de sistemas operativos libres mediante una tarjeta microSD.

### ¿Cuáles son los usos de las Raspberry Pi de forma personal y de forma empresarial?

A **nivel personal**, la Raspberry Pi ofrece una amplia gama de posibles usos. Algunas de las aplicaciones más comunes incluyen:

1. Centro multimedia: Puede utilizarse como un dispositivo de reproducción multimedia para transmitir contenido de vídeo, música o imágenes en una pantalla o televisor.
2. Servidor doméstico: La Raspberry Pi puede actuar como un servidor doméstico para almacenar y compartir archivos, alojar un sitio web personal o incluso configurar un servidor de juegos.
3. Proyectos de domótica: Es posible utilizar la Raspberry Pi como una central de domótica para controlar dispositivos del hogar como luces, termostatos, sistemas de seguridad, entre otros.
4. Aprendizaje de programación: La Raspberry Pi es una herramienta ideal para aprender a programar, ya que ofrece un entorno de desarrollo accesible y versátil para principiantes.
5. Proyectos de robótica: Puede utilizarse como el cerebro de robots y proyectos de automatización, permitiendo el control y la interacción con diferentes sensores y actuadores.

A **nivel empresarial**, la Raspberry Pi también tiene varias aplicaciones potenciales, como:

1. Señalización digital: Puede utilizarse para crear displays y carteles digitales en tiendas, restaurantes o ferias comerciales, para mostrar información relevante o promociones.
2. Servidores de datos: La Raspberry Pi puede funcionar como un servidor de datos local para pequeñas empresas, permitiendo el almacenamiento y acceso a archivos compartidos de forma segura.
3. Soluciones de Internet de las Cosas (IoT): La Raspberry Pi es ampliamente utilizada en proyectos de IoT, permitiendo la conexión y gestión de dispositivos inteligentes y recopilando datos para su análisis.
4. Control de automatización: Puede utilizarse para automatizar procesos en la línea de producción, monitorear y controlar equipos industriales o gestionar sistemas de iluminación y climatización..

## 2.1. Modelos Raspberry Pi

Los modelos que hemos usado para nuestro cluster son los siguientes:

- Raspberry pi 4 Model B 4GB [4]
- Raspberry pi 2 Model B [5]
- Raspberry pi 1 Model A+ [6]

### 2.1.1. Raspberry pi 4 Model B 4GB

La Raspberry Pi 4 Model B de 4GB (figura 1) es una placa de desarrollo de tamaño reducido que ofrece un alto rendimiento y versatilidad para diversas aplicaciones. A continuación, se presentan sus especificaciones técnicas:

1. **Procesador**: Incorpora un procesador Broadcom BCM 2711 de cuatro núcleos ARM Cortex-A72, con una frecuencia de reloj de 1.5 GHz. Este potente procesador proporciona un rendimiento mejorado en comparación con modelos anteriores.
2. **Memoria RAM**: Cuenta con 4GB de memoria LPDDR4-3200, lo que permite un rápido acceso a los datos y un funcionamiento fluido del sistema.
3. **Conectividad**: Dispone de dos puertos USB 3.0, dos puertos USB 2.0 y un conector USB-C para la alimentación. Además, incorpora un puerto Ethernet Gigabit para la conexión a la red cableada. También ofrece conectividad inalámbrica mediante Wi-Fi 802.11ac de doble banda (2.4 GHz y 5 GHz) y Bluetooth 5.0.
4. **Salida de vídeo**: Proporciona dos salidas micro-HDMI que admiten resoluciones de hasta 4K a 60 Hz. Esto permite una calidad de imagen nítida y la posibilidad de conectar dos pantallas simultáneamente.
5. **Almacenamiento**: Integra una ranura para tarjetas microSD, que permite expandir el almacenamiento mediante tarjetas de memoria de hasta 512 GB. Además, dispone de dos puertos USB 3.0 y un puerto USB 2.0 para conectar dispositivos de almacenamiento externos.
6. **Sistema Operativo**: Es compatible con una amplia gama de sistemas operativos, como Raspbian (basado en Linux), Ubuntu, Windows 10 IoT Core y muchos otros. Esto brinda flexibilidad para adaptarse a diferentes necesidades y proyectos.
7. **GPIO**: Incluye un conector de 40 pines GPIO (General Purpose Input/Output) que permite la conexión y control de sensores, actuadores y otros componentes electrónicos externos.

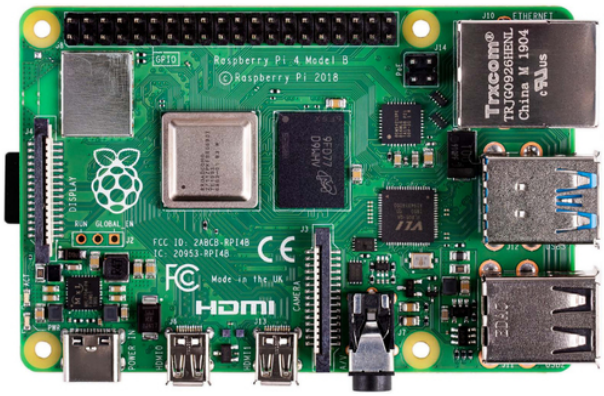



Figura 1: Raspberry 4 Model B 4GB

### 2.1.2. Raspberry pi 2 Model B

La Raspberry Pi 2 Modelo B (figura 2) es la segunda generación de Raspberry Pi. Sustituyó a la Raspberry Pi 1 Modelo B+ original en febrero de 2015.

1. **Procesador**: Incorporaba un procesador Broadcom BCM2836 de cuatro núcleos ARM Cortex-A7, con una frecuencia de reloj de 900 MHz. Este procesador ofrecía un aumento significativo en el rendimiento en comparación con el modelo anterior, la Raspberry Pi 1.
2. **Memoria RAM**: Contaba con 1GB de memoria LPDDR2 SDRAM, lo que permitía un funcionamiento más rápido y eficiente del sistema en comparación con la Raspberry Pi 1.
3. **Conectividad**: Disponía de cuatro puertos USB 2.0 para la conexión de dispositivos externos, como teclados, ratones, unidades de almacenamiento, entre otros. También incluía un puerto Ethernet para la conexión a la red.
4. **Salida de vídeo**: Proporcionaba una salida HDMI para conectar un monitor o televisor y un conector de vídeo compuesto (RCA) para la conexión a pantallas más antiguas. Soportaba resoluciones de hasta 1080p.
5. **Almacenamiento**: Ofrecía una ranura para tarjetas microSD para el almacenamiento del sistema operativo y los datos.
6. **GPIO**: Incluía un conector de 40 pines GPIO (General Purpose Input/Output) que permitía la conexión y control de componentes electrónicos externos, como sensores y actuadores.

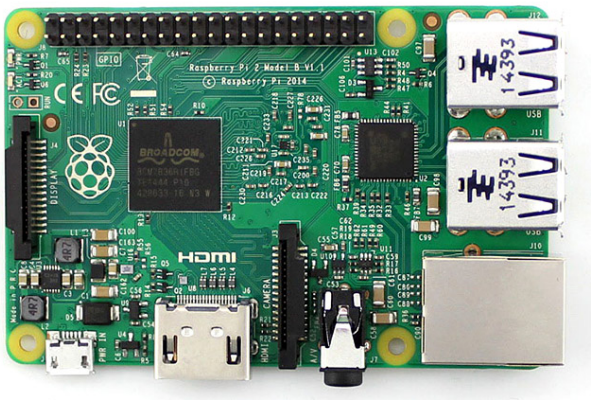



Figura 2: Raspberry 2 Model B

### 2.1.3. Raspberry Pi 1 Model A+

La Raspberry Pi 1 es un dispositivo de computación de bajo costo y tamaño reducido que fue lanzado en febrero de 2012. A continuación, se presentan las especificaciones técnicas de la Raspberry Pi 1 Model A (figura 3):

- **Procesador**: Broadcom BCM2835 de un solo núcleo a 700 MHz.
- **Memoria RAM**: 512 MB.
- **Conectividad**: 1 puerto USB 2.0, puerto de video compuesto, conector de cámara CSI, conector HDMI para salida de video de alta definición.
- **Factor de forma compacto y liviano**.
- **Bajo consumo de energía**.
- **Almacenamiento**: Ranura para tarjeta microSD (no incluye memoria interna).
- Capacidad de ejecutar **varios sistemas operativos**, incluyendo Raspbian y otros basados en Linux.
- **Amplia compatibilidad** con periféricos como teclados, ratones y unidades de almacenamiento externo.
- Ideal para proyectos portátiles y con espacio limitado.
- **Versatilidad** para proyectos de bricolaje, educación y aplicaciones industriales de bajo costo.

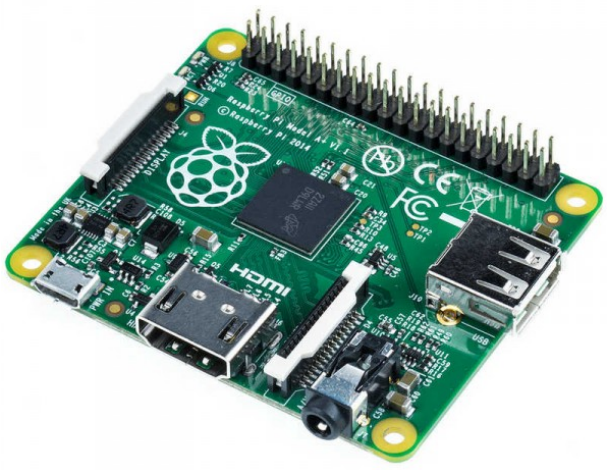


Figura 3: Raspberry 1 Model A+

## 2.2. Red local

**¿Qué es una red local?**

Una red local, también conocida como LAN (Local Area Network, por sus siglas en inglés), hace referencia a una infraestructura de comunicación que interconecta dispositivos electrónicos en un ámbito geográficamente restringido, como una oficina, un edificio o un campus universitario. Esta infraestructura posibilita el intercambio de datos y recursos entre los dispositivos conectados, lo que a su vez facilita la comunicación y la transferencia de información.

Las redes locales se caracterizan por diversos aspectos particulares. En primera instancia, presentan dimensiones limitadas y se diseñan con el propósito de abarcar un área geográfica reducida, lo cual simplifica la gestión y el mantenimiento de la red. Además, suelen ser de propiedad privada, lo que implica que la entidad u organismo responsable ejerce control sobre la seguridad y los recursos compartidos.

**Posibles casos donde se debe usar una red local**

Una red local puede ofrecer diversos beneficios y casos de uso. En un entorno empresarial, una LAN puede permitir la compartición de recursos, como impresoras, archivos y bases de datos, lo que facilita la colaboración y mejora la eficiencia operativa. Además, las LANs suelen proporcionar acceso a Internet compartido, lo que permite a los usuarios acceder a recursos en línea y comunicarse con el mundo exterior.

**¿Cómo configurar una red local en sistemas linux?**

En cuanto a la configuración de una red local, existen dos enfoques comunes: la configuración gráfica y la configuración desde la terminal en sistemas Linux. Para configurar una red local de forma gráfica, la mayoría de los entornos de escritorio en Linux ofrece herramientas intuitivas que facilitan la tarea. Por lo general, estas herramientas permiten configurar parámetros como la dirección IP, la máscara de subred, la puerta de enlace predeterminada y los servidores DNS. Además, es posible administrar aspectos de seguridad, como el cortafuegos y la autenticación de red.

Por otro lado, la configuración desde la terminal en Linux ofrece un enfoque más flexible y potente para la configuración de redes. Mediante comandos como ifconfig, ip, route y iptables, es posible manipular directamente las configuraciones de red. Esto proporciona un mayor nivel de control y permite realizar ajustes más específicos según las necesidades de la red.

En nuestro caso para la red local hemos usado un router TP-LINK con el cual realizar el enrutamiento entre las raspberry. Cada raspberry tenía una ip estática la cual configuramos de forma gráfica. Las ips para cada raspberry eran:

- **Raspberry pi4**: 192.168.1.104
- **Raspberry pi2**: 192.168.1.102
- **Raspberry pi1**: 192.168.1.101

Debido al cortafuegos de la Universidad de Córdoba teníamos conectividad entre los dispositivos pero no recibíamos una ip con la cual poder obtener acceso a internet, por lo cual nuestra red estaba aislada al acceso externo sino se conectaba a nuestro router. Para terminar este punto vamos a mostrar una foto del cluster (figura 4):

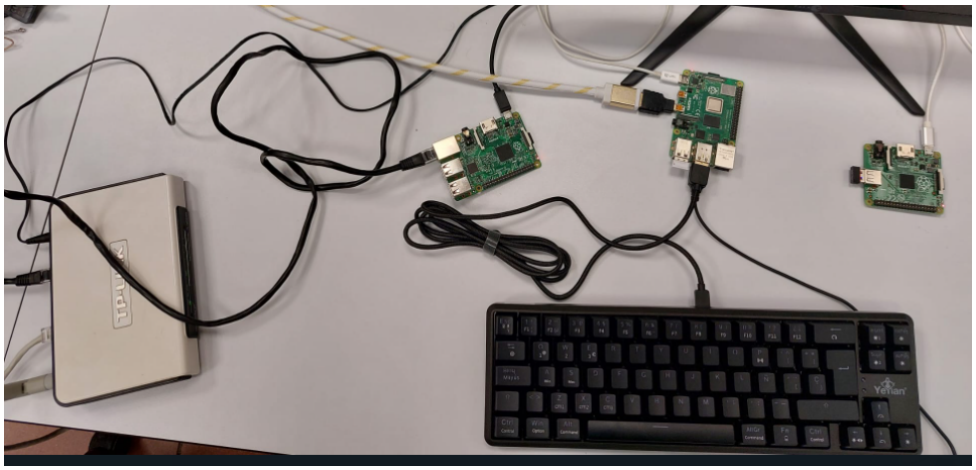


Figura 4: Cluster

# 3. Software y Entorno

## 3.1. Sistemas Operativos

Para instalar los sistemas operativos de Raspberry seguiremos las siguientes instrucciones:

1. Instalamos en nuestra máquina Raspberry Pi Imager [7] este software nos permitirá cargar en nuestras tarjetas SD el Sistema Operativo.
2. Para instalar el programa usaremos (figura 5)

```
sudo apt install ./imager_1.7.4_amd64.deb
```

Figura 5: Comando de instalación de Imager

3. Instalamos en cada una de las tarjetas los S.O comentados anteriormente (figura 6).
   *a*) Operating System: Sistema operativo a elegir.
   *b*) Storage: almacenamiento, en nuestro caso SD .
   *c*) Configuración adicional: indicamos zona horaria.
   *d*) Write: escribimos el S.O en la SD

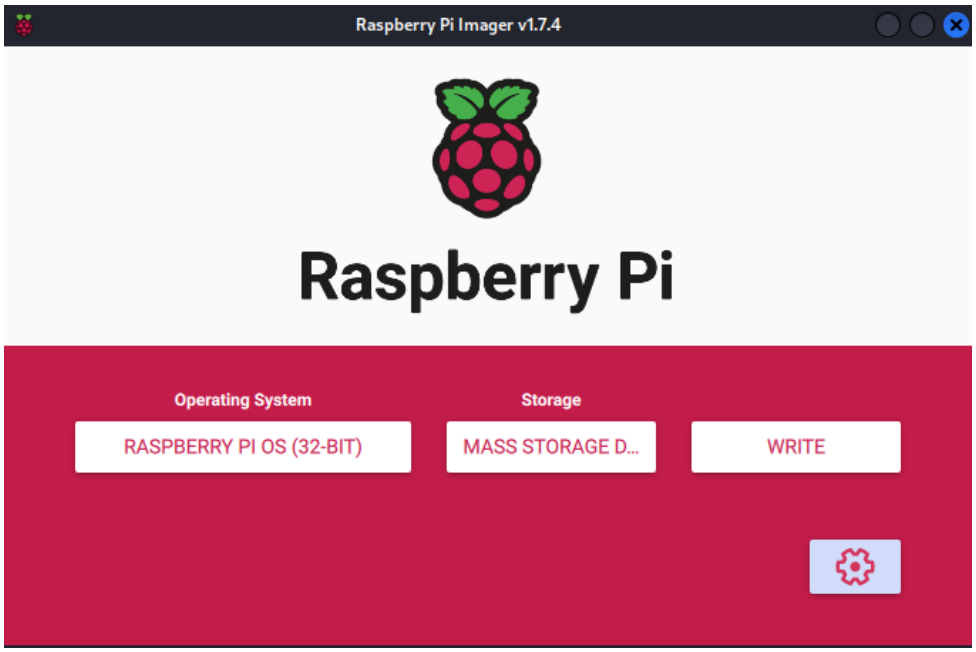



Figura 6: Ejemplo de instalación

4. Una vez terminada la instalación del S.O, procedemos encender nuestra Raspberry y configurar el sistema.
5. Una vez encendida nuestra Raspberry con la SD introducida y un teclado puesto, podemos configurar parámetros básicos como zona horaria, teclado y usuario/contraseña:
   - Usuario: ap
   - Password: ap2023

## 3.2. Entorno

Una vez configurado todos los Sistemas Operativos de nuestras Raspberrys Pi, procederemos a la configuración del entorno de trabajo, es decir, a la instalación de los paquetes necesarios y configuración de ficheros necesarios para una conexión en nuestra red local de una manera más amigable.

### 3.2.1. Paquetes

Para comenzar instalaremos los paquetes necesarios para poder ejecutar los programas con **MPI** tanto en C como en Python sin ningún problema:

- **Generales (Para C)**: (figura 7) mpich, mpich-doc, libmpich2-3, libmpich-dev, openmpi-bin y mpi. (**Nota**: instalar de uno en uno)

```
sudo apt-get install mpich mpich-doc libmpich2-3 libmpich-dev openmpi-bin mpi
```

Figura 7: Instalación de los paquetes para MPI y C

- **Python**:(figura 8) usaremos el instalador de paquetes nativo de python **pip**.

```
pip install mpi4py
```

Figura 8: Instalación de los paquetes para MPI y Python

### 3.2.2. Fichero /etc/hosts

Para comenzar vamos a cambiar los nombres de las máquinas, para que sean mucho más fácil la configuración. Para ello modificaremos el hostname de la maquina en el fichero **/etc/hostname** (figura 9).

```
sudo /etc/hostname
```

Figura 9: Cambio del nombre de hostname

El siguiente paso es configurar el fichero **/etc/hosts** de cada una de las Raspberrys, donde el formato del fichero será el siguiente (figura 10), y un ejemplo de la Raspberry 4 (figura 11), En el ejemplo vemos que la línea **127.0.0.1 raspberrypi** es comentada, es muy importante para poder dejar salir los mensajes a los otros dispositivos, esto hay que realizarlo en todas las máquinas:

```
<ip>:<nombre_host>
<ip>:<nombre_host>
#no incluir el mismo
```

Figura 10: Formato fichero /etc/hosts

```
ap@pi4:~ $ cat /etc/hosts
127.0.0.1       localhost
::1             localhost ip6-localhost ip6-loopback
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters

#127.0.1.1              raspberrypi

# Raspberry pi
192.168.1.102 pi2
192.168.1.101 pi1
192.168.1.103 alvaro
```

Figura 11: Fichero Etc/hosts de la pi4

## 3.3. Configuración

Para este proyecto hay dos aspectos cruciales para su funcionamiento, uno es SSH para la conexión entre las maquinas y el servidor NFS para compartir los ficheros en un mismo directorio.

### 3.3.1. SSH

Para la configuración de SSH usaremos la terminal de los dispositivos Raspberry Pi y seguiremos los siguientes pasos:

1. Entrar a la configuración nativa de los dispositivos Raspberry, con el comando **sudo raspi-config**.
2. Una vez dentro seleccionamos la opción **Interface Options** (figura 12).

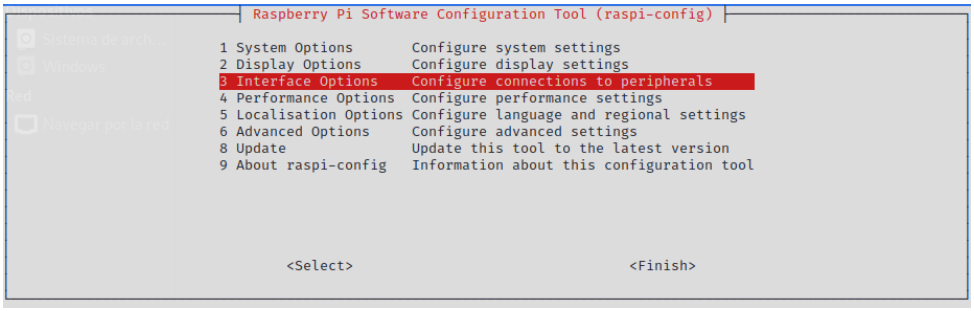



Figura 12: Interface Option de la configuración nativa

3. Una vez dentro habilitamos SSH (figura 13).

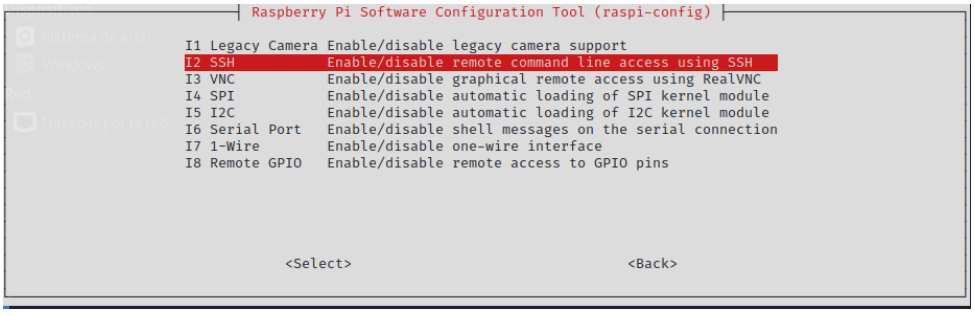



Figura 13: Habilitación de SSH

4. Una vez activado comprobamos con el comando **sudo systemctl status ssh**. Si hubiera algún problema reiniciamos el dispositivo.

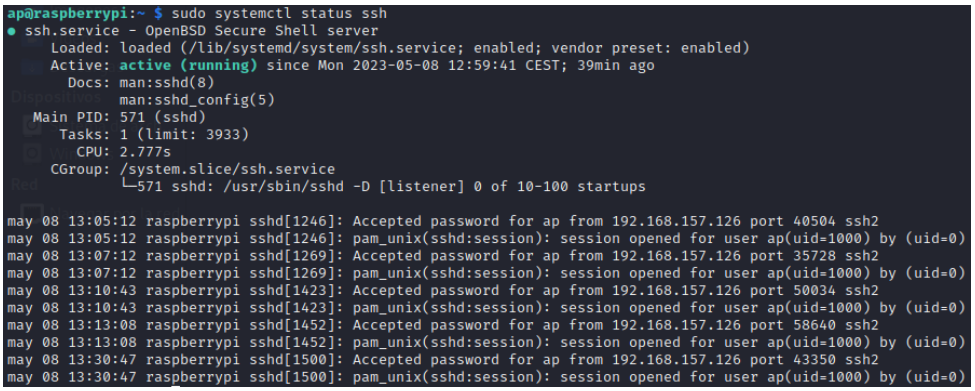



Figura 14: Estado de SSH

Una vez completada la activación del servicio ssh, nos disponemos a generar las contraseñas ssh y a compartirlas con el resto de hosts. En nuestro caso la contraseña ssh la hemos dejado vacía. Para realizar este procesos hemos usado dos comandos (figura 15). En el primer comando usamos nuestro nombre de host y usuario, y en el segundo para pasarlo a los demás usamos el nombre de los demas hosts. Por ejemplo si comfiguramos desde la Raspberry Pi 4, usaremos **ssh-keygen -t rsa -C pi4:ap** y **ssh-copy-id pi1** y **ssh-copy-id pi2**.

```
ssh-keygen -t rsa -C <hostname>:<username>
ssh-copy-id <remotehostname> (Para pasar a los demas hosts)
```

Figura 15: Generar y compartir contraseñas ssh

### 3.3.2. NFS

Para poder compartir los ficheros y tener un punto donde todas las Raspberry puedan acceder vamos a instalar y configurar un NFS [8] (Network File System). El servidor de este NFS estrá montado en la Raspberry Model 4 y los clientes en las Raspberry Model 1 y 2.

**Servidor NFS:** Comencemos instalando y configurando el servidor NFS, para ello vamos a seguir los siguientes pasos (figura 16):

1. Instalamos el paquete necesario para poder tener y utilizar el servidor, con el comando **sudo apt-get install nfs-kernel-server**.
2. Creamos una carpeta que será nuestro punto compartido del NFS, con el comando **mkdir /cloud**.
3. A continuación vamos a crear una entrada en el fichero **/etc/exports**, de esta manera le daremos permisos para exportar contenido a la carpeta anteriormente creada. Primero con el comando **sudo nano /etc/exports** entramos al fichero y finalmente añadimos la siguiente línea al final **/home/ap/cloud *(rw,sync,no_root_squash,no_subtree_check)**.
4. Para terminar exportamos aplicamos los cambios con **sudo exportfs -a** o simplemente reiniciamos el servicio con **sudo service nfs-kernel-server restart**

```
sudo apt-get install nfs-kernel-server
mkdir /cloud
sudo nano /etc/exports #añadimos la línea
sudo exportfs -a
```

Figura 16: Pasos para montar el servidor NFS

**Clientes NFS:** a conitnuación instalaremos y configuraremos los clientes NFS, para ello vamos a seguir los siguientes pasos (figura 17):

1. Instalamos el paquete necesario para poder tener y configurar el equipo como cliente de NFS, con el comanod **sudo apt-get install nfs-common**.
2. Creamos una carpeta que será nuestro punto compartido del NFS, con el comando **mkdir /cloud**.
3. Por ultimo tenemos que realizar el montado de la unidad del NFS, hacemos un montado cada vez que el cliente quiera usar el NFS, con el comando **sudo mount -t nfs ¡serverhostname¿:/home/ap/cloud /cloud**, en nuestro caso el serverhostname es **pi4**.

```
sudo apt-get install nfs-common
mkdir /cloud
sudo mount -t nfs pi4:/home/ap/cloud /cloud
```

Figura 17: Pasos para montar el cliente NFS

# 4. Código

## 4.1. Mpi con C

El código que hemos realizado consta de 1 función y el main de dicho programa. Comencemos explicando la función y cabeceras del código (figura 18).

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

        #define M_SIZE 1024
        int A[M_SIZE][M_SIZE];
        int B[M_SIZE][M_SIZE];
        int C[M_SIZE][M_SIZE];
        int global_C[M_SIZE][M_SIZE];

        void Rellenar_Matrices(int A[M_SIZE][M_SIZE], int B[M_SIZE][M_SIZE], int
↪   C[M_SIZE][M_SIZE]) {
            int i, j;
            for (i = 0; i < M_SIZE; i++) {
                for (j = 0; j < M_SIZE; j++) {
                    A[i][j] = rand() % 10;
                    B[i][j] = rand() % 10;
                    C[i][j] = 0;
                }
            }
        }
```

Figura 18: Código de las cabeceras y función de rellanar matriz en código C

En este apartado del código se declaran las librerías necesarias para el funcionamiento del código, la variable **M_SIZE** para el tamaño de las matrices A, B, C, global_C y la función que rellena estas matrices con números aleatorios entre 0 y 100.

Ahora vamos a continuar explicando el código main (figura 19). En el código del main se realizan las siguientes funciones o apartados a explicar:

1. Inicializa la semilla para la generación de números aleatorios utilizando srand y time.
2. Declara las variables **process_rank** y **comunicator_size** para almacenar el rango del proceso y el tamaño del comunicador MPI respectivamente.
3. Inicializa el entorno MPI con **MPI_Init**.
4. Obtiene el rango del proceso actual y el tamaño del comunicador MPI con **MPI_Comm_rank** y **MPI_Comm_size**.
5. Calcula el tamaño local de cada proceso dividiendo **M_SIZE** entre el tamaño del comunicador MPI.
6. Declara una matriz local local_C de tamaño local_size x M_SIZE para almacenar el resultado local de la multiplicación de matrices.
7. Creamos un control para evitar que el tamaño del grupo asociado al comunicado sea menor a 2.
8. Mide el tiempo de inicio de ejecución utilizando **MPI_Wtime**.
9. Si el rango del proceso es 0, llama a la función Rellenar_Matrices para llenar las matrices A, B y C con valores aleatorios, e imprime un mensaje indicando la multiplicación en paralelo con el número de procesos utilizados.
10. Sincroniza todos los procesos utilizando **MPI_Barrier**.

11. Transmite la matriz B a todos los procesos utilizando **MPI_Bcast**.
12. Distribuye la matriz A entre los procesos utilizando **MPI_Scatter**.
13. Cada proceso realiza la multiplicación local de matrices en su subconjunto de filas y columnas de local_C.
14. Recopila los resultados parciales de cada proceso en la matriz global_C utilizando **MPI_Gather**.
15. Mide el tiempo de finalización de ejecución utilizando **MPI_Wtime**.
16. Finaliza el entorno MPI con **MPI_Finalize**.

```
    int main(int argc, char *argv[]) {
        srand(time(NULL));

        int process_rank, comunicator_size;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &comunicator_size);
        int local_size = M_SIZE / comunicator_size;
        int local_C[local_size][M_SIZE];
        if(comunicator_size<2){
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
        double start = MPI_Wtime();
        if (process_rank == 0) {
            Rellenar_Matrices(A, B, C);
            printf("\nRealizando la multiplicación con %d procesos ... \n",
↪   comunicator_size);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Bcast(B, M_SIZE * M_SIZE, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Scatter(A, local_size * M_SIZE, MPI_INT, local_C, local_size *
↪   M_SIZE, MPI_INT, 0, MPI_COMM_WORLD);

        for (int i = 0; i < local_size; i++) {
            for (int j = 0; j < M_SIZE; j++) {
                for (int k = 0; k < M_SIZE; k++) {
                    local_C[i][j] += local_C[i][k] * B[k][j];
                }
            }
        }

        MPI_Gather(local_C, local_size * M_SIZE, MPI_INT, global_C, local_size *
↪   M_SIZE, MPI_INT, 0, MPI_COMM_WORLD);

        double end = MPI_Wtime();

        if (process_rank == 0) {
            double time = end - start;
            printf("\nTiempo de ejecución: %f segundos\n", time);
            printf("C_global[10][10] = %d\n", global_C[10][10]);
        }

        MPI_Finalize();
        return 0;
}
```

Figura 19: Código del main en C

Para comparar el rendimiento del programa que hemos realizado con MPI, lo compararemos con un programa secuencial ejecutado en la Raspberry 1 Model A+. (figura 20).

Para realizar este código hemos usado las siguientes páginas para buscar información [9], [10], [11]

```c
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define MAX 1024
int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];

int main (){

    for(int i=0; i<MAX; i++){
        for(int j=0; j<MAX;j++){
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
            C[i][j] = 0;
        }
    }

    double start = omp_get_wtime();

    for(int i=0; i<MAX; i++){
        for(int j=0; j<MAX; j++){
            for(int k=0; k<MAX; k++){
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    double end = omp_get_wtime();

    double time = (double) (end - start);

    printf("%d\n", C[10][10]);
    printf("El tiempo de ejecución es: %f\n", time);
    return 0;
}
```

Figura 20: Código secuencial en C

## 4.2. MPI con Python

Este código utiliza la biblioteca mpi4py para realizar una multiplicación de matrices utilizando el paradigma de programación paralela llamado "Message Passing Interface"(MPI) (figura 21):

1. Importar las bibliotecas necesarias:
   - **from mpi4py import MPI** importa la biblioteca mpi4py que proporciona funciones y herramientas para la programación paralela basada en MPI.
   - **import numpy as np** importa la biblioteca NumPy para trabajar con matrices y arreglos multidimensionales.
   - **import random** importa la biblioteca random para generar números aleatorios.
2. Definir el tamaño de la matriz: $\mathbf{M}_SIZE = 1024 define el tamaño de la matriz cuadrada.$
3. Definir una función para rellenar las matrices: **Rellenar_Matrices(A, B, C)** es una función que recibe tres matrices (A, B y C) y las rellena con valores aleatorios entre 0 y 9. Utiliza un bucle anidado para iterar sobre los índices de la matriz y asignar un número aleatorio a cada elemento.
4. Crear las matrices globales:
   - global_A = np.zeros((M_SIZE, M_SIZE), dtype=np.float32) crea una matriz global A de tamaño M_SIZE x M_SIZE, inicializada con ceros y tipo de datos de punto flotante de 32 bits.
   - global_B = np.zeros((M_SIZE, M_SIZE), dtype=np.float32) crea una matriz global B de tamaño M_SIZE x M_SIZE, inicializada con ceros y tipo de datos de punto flotante de 32 bits.

- global_C = np.zeros((M_SIZE, M_SIZE), dtype=np.float32) crea una matriz global C de tamaño M_SIZE x M_SIZE, inicializada con ceros y tipo de datos de punto flotante de 32 bits.

5. Inicializar la comunicación MPI:
   - **comm = MPI.COMM_WORLD** crea un comunicador MPI.
   - **rank = comm.Get_rank()** obtiene el rango (identificador) del proceso actual.
   - **rank = comm.Get_rank()** obtiene el número total de procesos.
6. Calcular el tamaño local: **local_size = M_SIZE // size** calcula el tamaño de cada bloque de la matriz local dividiendo el tamaño total de la matriz entre el número de procesos.
7. Crear la matriz local: local_C = np.zeros((local_size, M_SIZE), dtype=np.float32) crea una matriz local de tamaño local_size x M_SIZE, inicializada con ceros y tipo de datos de punto flotante de 32 bits.
8. Medir el tiempo de ejecución: start_time = MPI.Wtime() registra el tiempo de inicio de la ejecución.
9. Proceso del rango 0: Si el rango es 0, se llama a la función Rellenar_Matrices para rellenar las matrices globales A, B y C con valores aleatorios. **comm.Barrier()** se utiliza para sincronizar todos los procesos antes de continuar.
10. Difundir la matriz global B: **comm.Bcast(global_B, root=0)** se utiliza para difundir la matriz global B a todos los procesos desde el proceso.
11. Dividir la matriz global A en matrices locales: **comm.Scatter(global_A, local_C, root=0)** se utiliza para dividir la matriz global A en submatrices y distribuir cada submatriz en los procesos locales. Cada proceso recibe una parte de la matriz global A que se almacena en su matriz local C.
12. Realizar la multiplicación de matrices local: **local_C = np.dot(global_A, global_B)** realiza la multiplicación de matrices local en cada proceso, multiplicando la matriz local C por la matriz global B.
13. Recopilar las matrices locales en la matriz global C: **comm.Gather(local_C, global_C, root=0)** se utiliza para recopilar las matrices locales de cada proceso y combinarlas en la matriz global C en el proceso de rango 0.
14. Medir el tiempo de ejecución final: **end_time = MPI.Wtime()** registra el tiempo de finalización de la ejecución.
15. Proceso del rango 0: Si el rango es 0, se calcula el tiempo transcurrido restando el tiempo de finalización del tiempo de inicio. Se imprime el tiempo de ejecución y se muestra el valor del elemento en la posición [10][10] de la matriz global C.
16. Finalizar MPI: **MPI.Finalize()** finaliza la comunicación MPI.

```python
from mpi4py import MPI
import numpy as np
import random

M_SIZE = 1024

def Rellenar_Matrices(A, B, C):
    for i in range(M_SIZE):
        for j in range(M_SIZE):
            global_A[i][j] = random.randint(0, 9)
            global_B[i][j] = random.randint(0, 9)
            global_C[i][j] = 0

global_A = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)
global_B = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)
#C = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)
global_C = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)

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

local_size = M_SIZE // size
local_C = np.zeros((local_size, M_SIZE), dtype=np.float32)

start_time = MPI.Wtime()
if rank == 0:
    Rellenar_Matrices(global_A, global_B, global_C)

comm.Barrier()
comm.Bcast(global_B, root=0)
comm.Scatter(global_A, local_C, root=0)

global_C = np.dot(global_A, global_B)

comm.Gather(local_C, global_C, root=0)

end_time = MPI.Wtime()
if rank == 0:
    elapsed_time = end_time - start_time
    print("\nTiempo de ejecución: %f segundos" % elapsed_time)
    print("C_global[10][10] =", global_C[10][10])

MPI.Finalize()
```

Figura 21: Código MPI con python

Para poder comparar los tiempos y sacar conclusiones entre la paralelización y los tiempos secuenciales usaremos el código implementado con MPI en Python de manera secuencial (figura 22).

Para esta parte en la que hemos trabajado con python, hemos usado las siguientes fuentes de información [12] [13]

```python
import numpy as np
import time
import random

M_SIZE = 1024

def Rellenar_Matrices(A, B, C):
    for i in range(M_SIZE):
        for j in range(M_SIZE):
            global_A[i][j] = random.randint(0, 9)
            global_B[i][j] = random.randint(0, 9)
            global_C[i][j] = 0

global_A = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)
global_B = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)
#C = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)
global_C = np.zeros((M_SIZE, M_SIZE), dtype=np.float32)

Rellenar_Matrices(global_A, global_B, global_C)

start_time = time.time()

global_C = np.dot(global_A, global_B)

end_time = time.time()

elapsed_time = end_time - start_time
print("\nTiempo de ejecución: %f segundos" % elapsed_time)
print("C_global[10][10] =", global_C[10][10])
```

Figura 22: Código secuencial con python

## 4.3. OpenMP con C

Para este apartado vamos a realizar la mejor opción de paralelización de cada vector (externa, intermedia e interna) y realizaremos un estudio con ejecución de 2,4,8,16,32,64 hilos. Para ello vamos a usar un código muy parecido al del apartado A. La mecánica será igual, tres códigos cada uno un nivel de paralelización y en cada código un switch que da opción a usar cada vector de ejecución. La única diferencia es el uso de **omp_set_num_thread** en vez de **omp_get_num_thread** para en el main poder introducir por pantalla el número de hilos y asignarlos a la región paralela (figura 23).

La explicación del código (figura 23) podemos verla a continuación:

1. Incluye las bibliotecas necesarias: **time.h, stdlib.h, stdio.h y omp.h**, que proporcionan funciones y tipos de datos necesarios para el programa.

2. Define una constante **MAX** con el valor 1024. Esta constante se utiliza para definir el tamaño de las matrices.

3. Declara tres matrices de tamaño **MAXxMAX**: A, B y C. Estas matrices se utilizarán para almacenar los datos de entrada y el resultado de la multiplicación.

4. Declara las variables **num_thread**, **start**, **end** y **t**. **num_thread** almacenará el número de hilos que se utilizarán en la ejecución del programa. **start, end y t** se utilizarán para medir el tiempo de ejecución del programa.

5. Define la función **MultiplicarMatriz_i_k_j().** Esta función utiliza OpenMP para realizar la multiplicación de matrices de manera paralela. Primero, se establece el número de hilos utilizando la función **omp_set_num_threads()** y se pasa el valor de **num_thread**. Luego, se utiliza la directiva **pragma omp parallel** para iniciar una región paralela. Dentro de esta región, se utiliza la directiva **pragma omp for** con la cláusula **schedule(static) nowait** para distribuir el bucle externo de la multiplicación de matrices entre los hilos. Los bucles internos se ejecutan de manera secuencial. Cada hilo realiza la multiplicación de las filas correspondientes de las matrices A y B y acumula los resultados en la matriz C.

6. Define la función **main().** Esta función es el punto de entrada del programa. Primero, muestra un mensaje para que el usuario introduzca el número de hilos que desea utilizar. Luego, utiliza la función scanf() para leer el número de hilos y almacenarlo en la variable num_thread. A continuación, se llama a la función Rellenar_Matrices() para inicializar las matrices A y B con valores aleatorios. Después, se registra el tiempo de inicio de la ejecución utilizando la función **omp_get_wtime().** A continuación, se llama a la función MultiplicarMatriz_i_k_j() para realizar la multiplicación de matrices. Después de la multiplicación, se registra el tiempo de finalización de la ejecución y se calcula la duración total de la ejecución. Finalmente, se muestra el valor de la celda [10][10] de la matriz C y el tiempo de ejecución en segundos.

```c
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define MAX 1024
int A[MAX][MAX], B[MAX][MAX], C[MAX][MAX];
int num_thread=0;
double start=0.0, end=0.0, t=0.0;

//Vector i,k,j
void MultiplicarMatriz_i_k_j(){
    omp_set_num_threads(num_thread);
    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait
        for(int i=0; i<MAX; i++){
            for(int j=0; j<MAX; j++){
                for(int k=0; k<MAX; k++){
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }
}
void main (){
    printf("Introduce el numero de hilos que desee (2,4,8,16,32,64)\n");
    scanf("%d", &num_thread);
    srand(time(NULL));
    Rellenar_Matrices();

    start = omp_get_wtime();
    MultiplicarMatriz_i_k_j();
    end = omp_get_wtime();
    t =(end - start);
    printf("%d\n", C[10][10]);
    printf("El tiempo de ejecución es: %f\n", t);
```

Figura 23: Código de OpenMP con C

La mejor opción de nivel de paralelización para cada vector es **Vector i,k,j**: la mejor opción es externo, ya que estamos paralelizando el nivel externo, es decir las filas con lo que no creamos ningún conflicto y podemos repartir el trabajo de los 4000 elementos entre el número de hilos, por ejemplo 8, es decir, 500 elementos por hilos. Esta opción de paralelización se estudio en clase, en la práctica 4 con OpenMp, que era una de las mejores.

Siempre mejor paralelizar i o en su defecto k, ya que de esta manera paralelizamos lo más externo que no depende de lo interno y repartimos de mejor manera los elementos a los recursos (hilos).

# 5. Mediciones

## 5.1. MPI con C

A continuación vamos a centrarnos en los tiempos de ejecución obtenidos durante la ejecución de los códigos en C. Para ello hemos ejecutado 5 veces los programas con distintos números de hilos y hemos hecho una media aritmética de los tiempos (figura 24).

| **Num Hilos** | **Tiempo C** |
|---|---|
| Secuencial | 128,589437 |
| 2 | 88,502392 |
| 4 | 41,754691 |
| 8 | 78,100495 |
| 16 | 134,355219 |
| 32 | 174,255278 |
| 64 | 201,578934 |

Figura 24: Tiempos de ejecución del programa MPI con C

A continuación para poder apreciar realmente si es necesario el uso de la paralelización en estos casos, vamos a calcular el Speed-Up (figura 25). Para calcular el Speed-Up en tanto por ciento usaremos la siguiente fórmula:

$$Speed\_up = [(Tiempo\_original \ - \ Tiempo\_mejorado)/Tiempo\_original] * 100 \quad (1)$$

| **Num Hilos** | **SP % C** |
|---|---|
| 2 | 31,17444631 |
| 4 | 67,52867734 |
| 8 | 39,26367762 |
| 16 | -4,483869075 |
| 32 | -35,5129022 |
| 64 | -56,76165842 |

Figura 25: Speed Up de ejecución del programa MPI con C

## 5.2. MPI con Python

A continuación vamos a centrarnos en los tiempos de ejecución obtenidos durante la ejecución de los códigos en Python. Para ello hemos ejecutado 5 veces los programas con distintos números de hilos y hemos hecho una media aritmética de los tiempos (figura 26).

| **Num Hilos** | **Tiempo Python** |
|---|---|
| Secuencial | 7,4369821 |
| 2 | 2,214578 |
| 4 | 2,350501 |
| 8 | 4,59763 |
| 16 | 8,811508 |
| 32 | 16,399719 |
| 64 | 29,789567 |

Figura 26: Tiempos de ejecución del programa MPI con Python

A continuación para poder apreciar realmente si es necesario el uso de la paralelización en estos casos, vamos a calcular el Speed-Up (figura 27). Para calcular el Speed-Up en tanto por ciento usaremos la siguiente fórmula:

$$Speed\_up = [(Tiempo\_original \ - \ Tiempo\_mejorado)/Tiempo\_original] * 100 \quad (2)$$

| **Num Hilos** | **SP % Python** |
|---|---|
| 2 | 70,2220878 |
| 4 | 68,3944244 |
| 8 | 38,17882122 |
| 16 | -18,48230749 |
| 32 | -120,5157788 |
| 64 | -300,5598857 |

Figura 27: Speed Up de ejecución del programa MPI con Python

## 5.3. OpenMP con C

A continuación vamos a centrarnos en los tiempos de ejecución obtenidos durante la ejecución de los códigos en C. Para ello hemos ejecutado 5 veces los programas con distintos números de hilos y hemos hecho una media aritmética de los tiempos, estos tiempos ha sido sacados ejecutando el código en la Raspberry Model 4 (figura 28).

| **Num Hilos** | **Tiempo OpenMP** |
|---|---|
| Secuencial | 13,761133 |
| 2 | 6,973904 |
| 4 | 3,4666023 |
| 8 | 3,496053 |
| 16 | 3,511393 |
| 32 | 3,679254 |
| 64 | 3,786742 |

Figura 28: Tiempos de ejecución del programa OpenMP con C

A continuación para poder apreciar realmente si es necesario el uso de la paralelización en estos casos, vamos a calcular el Speed-Up (figura 29). Para calcular el Speed-Up en tanto por ciento usaremos la siguiente fórmula:

$$Speed\_up = [(Tiempo\_original \ - \ Tiempo\_mejorado)/Tiempo\_original] * 100 \quad (3)$$

| **Num Hilos** | **SP % OpenMP** |
|---|---|
| 2 | 49,32173099 |
| 4 | 74,80874358 |
| 8 | 74,59472995 |
| 16 | 74,48325657 |
| 32 | 73,26343696 |
| 64 | 72,48233848 |

Figura 29: Speed Up de ejecución del programa OpenMP con C

# 6. Conclusiones y Análisis

Para comenzar con las conclusiones, trataremos el análisis que hemos podido obtener experimentalmente, con la ejecución de los diferentes códigos y tecnologías en equipos reales como son las Raspberry Pi y no en un entorno simulado como en las prácticas:

- **MPI con C**: como podemos ver tanto en las tablas de tiempos como de SpeedUp (figura 24, 25) los tiempos disminuyen con el aumento de hilos hasta llegar a 4 hilos que es el pico de ganancia que proporcionan las Raspberry Pi. Apartir de este punto la ganancia se vuelve negativa, ya que intentan crear recursos (hilos) que no pueden abastecer por deficiencia del hardware de los dispositivos. Con lo que la paralelización es viable con MPI en C, sabiendo gestionar los recursos hardware y no sobre pasando los limites de este. Podemos ver los resultando en la gráfica siguiente (figura 30)

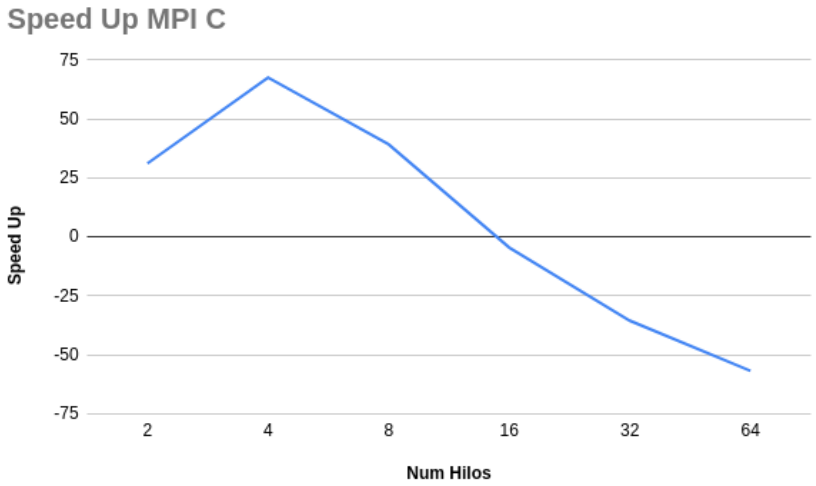



Figura 30: Gráfica de Speed Up de MPI con C

- **MPI con Python**: en este apartado podemos apreciar lo mismo que en el acaso anterior con el código en C, tenemos buena ganancia hasta llegar el máximo de recursos hardware que vuelve a ser 4 hilos. Con lo que tenemos la misma justificación que en el caso anterior. Si es verdad que hemos experimentado algo extraño, el código en C debería ser más rápido que en Python ya que en C nos quitamos toda la parte interpretativa del código. Los resultados los podemos ver en las tablas (figura 26, 27) y gráfica (figura 31) correspondientes.

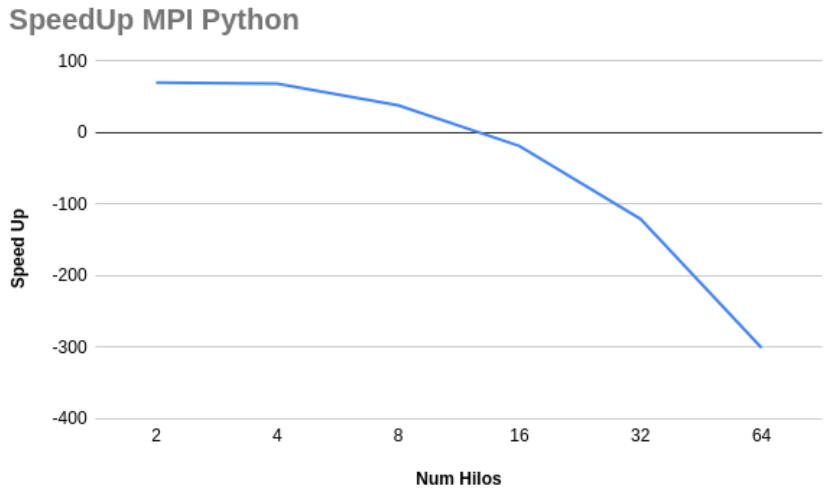



Figura 31: Gráfica de Speed Up de MPI con Python

- **OpenMP con C**: con OpenMP en las Raspberry Model 4 ha ido todo a la perfección, hemos experimentado un comportamiento parecido que en las prácticas de clase, simplemente con las limitaciones hardware que experimentan las Raspberry Pi, siendo su pico máximo de ganancia con 4 hilos y manteniéndose o disminuyéndose levemente mientras se aumentaban el número de hilos. Los resultados los podemos ver en las tablas (figura 28, 29) y gráfica (figura 32) correspondientes.

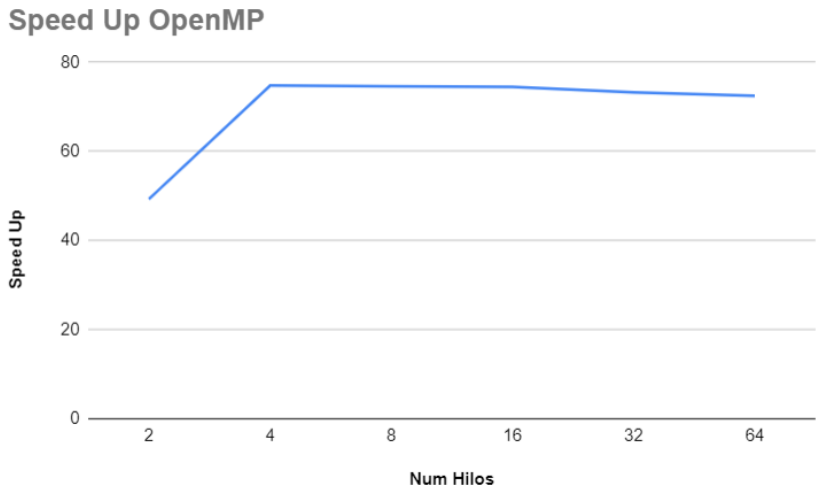



Figura 32: Gráfica de Speed Up de OpenMP con C

En resumen, el proyecto nos ha parecido muy interesante, hemos visto el funcionamiento de todo lo explicando en clase con equipos reales y de carácter IoT. Hemos tenido problemas de versiones, algo que en clase al ser simulado no pasaba. Concretamente este error fue uno muy importante ya que para ejecutar los programas con la flag **-f machinefile** hemos tenido que ejecutar los programas lanzándolos desde la Raspberry Pi 1 Model A+ ya que era la única en la que podíamos instalar la versión de MPI 3, mientras que en el resto la versión era la 4. Sinceramente el proyecto ha sido muy enriquecedor ya que hemos visto una situación real empresarial trabajando en equipo y con dispositivos reales.

# Referencias

[1] *Repositorio del Proyecto*. URL: https://github.com/Alvar04/CLUSTER_RASPBERRY.git (vid. pág. 3).

[2] *Blog de ClusterPi*. URL: https://blog.anthares101.com/magi-project/ (vid. pág. 3).

[3] *Documentación Raspberry*. URL: https://www.raspberrypi.com/documentation/ (vid. pág. 3).

[4] *Especificaciones de Raspberry 4*. URL: https://www.pccomponentes.com/caracteristicas-raspberry-pi-4 (vid. pág. 4).

[5] *Especificaciones de Raspberry 2*. URL: https://raspberryparatorpes.net/hardware/raspberry-pi-2-model-b/ (vid. pág. 4).

[6] *Especificaciones de Raspberry 1*. URL: https://www.ediciones-eni.com/open/mediabook.aspx?idR=14eb0dda346e32b9d304%20d7908f5fcc38 (vid. pág. 4).

[7] *Software Pi Imager*. URL: https://www.raspberrypi.com/software/ (vid. pág. 8).

[8] *NFS Information*. URL: https://fortinux.gitbooks.io/humble_tips/content/administrar_gnulinux/tutorial_montar_particiones_nfs_en_una_red_gnulinux.html (vid. pág. 11).

[9] *MPI Universidad de Granada*. URL: https://lsi2.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php (vid. pág. 14).

[10] *MPI Tutorial*. URL: https://mpitutorial.com/tutorials/ (vid. pág. 14).

[11] *MPI Information*. URL: https://www.mpich.org/static/docs/v3.3/www3/ (vid. pág. 14).

[12] *Python Information*. URL: https://docs.python.org/es/3.9/c-api/memory.html (vid. pág. 17).

[13] *Cluster Raspberry con MPI*. URL: https://www.meccanismocomplesso.org/en/clusters-and-parallel-programming-with-mpi-and-raspberry-pi/ (vid. pág. 17).