

## PRACTICA 2 – Paralelización con MPI

### EJERCICIO 1.1

Modificar el programa “hola\_mundo\_avanzado.c” (presentado en las transparencias incluidas en el material de esta práctica) para que en el mensaje de salida que muestra por pantalla cada proceso incluya al final el nombre del procesador en que está corriendo ese proceso. Por ejemplo, una salida típica podría ser:

```
[practica@hal practica-mpi]$ mpirun -np 3 hola_mundo_avanzado2
```

Soy el proceso 0 de 3 corriendo en hal: !Hola mundo!

Yo soy el proceso 1 de 3, corriendo en hal.

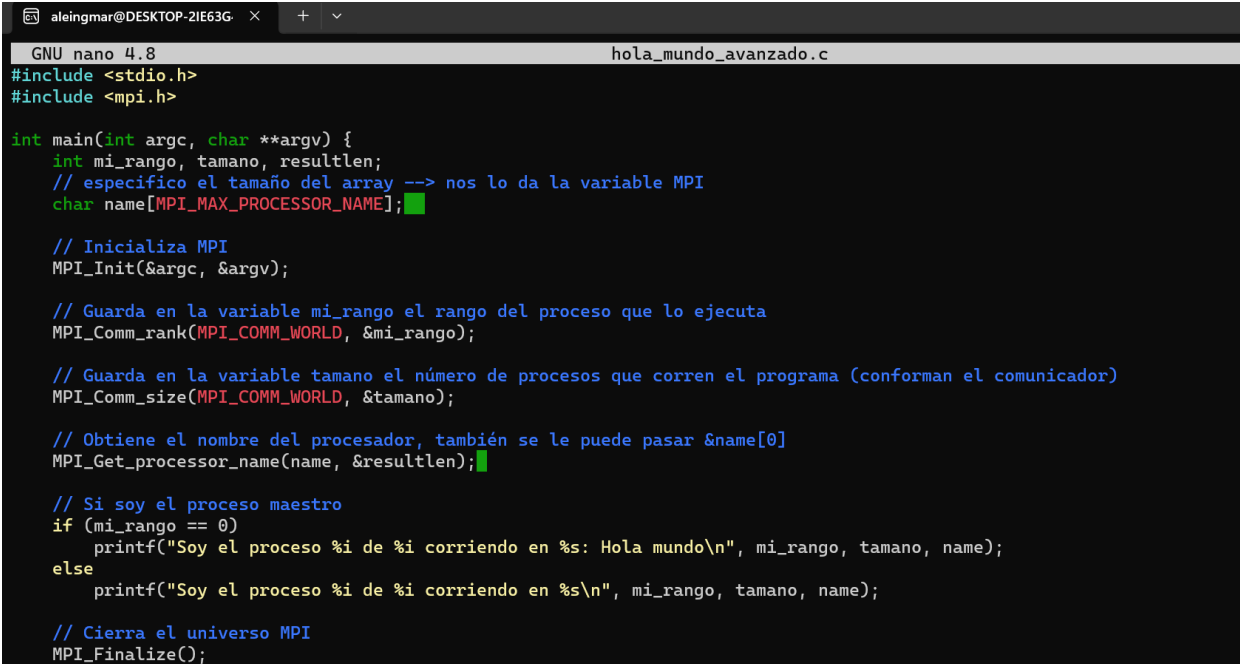
Yo soy el proceso 2 de 3, corriendo en hal.

Pista: Utilizar la función `MPI_Get_processor_name`.

Investigar si se puede lanzar un número de procesos que sea mayor que el número de núcleos del procesador utilizado y qué ocurre en ese caso.

El computador donde se ejecuta tiene 8 cores lógicos (4 físicos)

Programa `hola_mundo_avanzado` modificado:



```
GNU nano 4.8 hola_mundo_avanzado.c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
    int mi_rango, tamano, resultlen;
    // especifico el tamaño del array --> nos lo da la variable MPI
    char name[MPI_MAX_PROCESSOR_NAME];

    // Inicializa MPI
    MPI_Init(&argc, &argv);

    // Guarda en la variable mi_rango el rango del proceso que lo ejecuta
    MPI_Comm_rank(MPI_COMM_WORLD, &mi_rango);

    // Guarda en la variable tamano el número de procesos que corren el programa (conforman el comunicador)
    MPI_Comm_size(MPI_COMM_WORLD, &tamano);

    // Obtiene el nombre del procesador, también se le puede pasar &name[0]
    MPI_Get_processor_name(name, &resultlen);

    // Si soy el proceso maestro
    if (mi_rango == 0)
        printf("Soy el proceso %i de %i corriendo en %s: Hola mundo\n", mi_rango, tamano, name);
    else
        printf("Soy el proceso %i de %i corriendo en %s\n", mi_rango, tamano, name);

    // Cierra el universo MPI
    MPI_Finalize();
}
```

¿Se puede lanzar un número de procesos mayor que el número de núcleos del procesador utilizado? ¿Qué ocurre en ese caso?

Ejecución:

```
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpicc hola_mundo_avanzado.c -o hola_mundo_avanzado2
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 4 hola_mundo_avanzado2
Soy el proceso 1 de 4 corriendo en DESKTOP-2IE63G4
Soy el proceso 2 de 4 corriendo en DESKTOP-2IE63G4
Soy el proceso 0 de 4 corriendo en DESKTOP-2IE63G4: Hola mundo
Soy el proceso 3 de 4 corriendo en DESKTOP-2IE63G4
```

Cuando se intenta lanzar la ejecución en 5 procesos, sale este aviso en la terminal:

```
-----
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 5 hola_mundo_avanzado2
-----
There are not enough slots available in the system to satisfy the 5
slots that were requested by the application:

    hola_mundo_avanzado2

Either request fewer slots for your application, or make more slots
available for use.

A "slot" is the Open MPI term for an allocatable unit where we can
launch a process. The number of slots available are defined by the
environment in which Open MPI processes are run:

1. Hostfile, via "slots=N" clauses (N defaults to number of
   processor cores if not provided)
2. The --host command line parameter, via a ":N" suffix on the
   hostname (N defaults to 1 if not provided)
3. Resource manager (e.g., SLURM, PBS/Torque, LSF, etc.)
4. If none of a hostfile, the --host command line parameter, or an
   RM is present, Open MPI defaults to the number of processor cores

In all the above cases, if you want Open MPI to default to the number
of hardware threads instead of the number of processor cores, use the
--use-hwthread-cpus option.

Alternatively, you can use the --oversubscribe option to ignore the
number of available slots when deciding the number of processes to
launch.
```

Siguiendo estrictamente la teoría no se debería poder lanzar un mayor número de procesos que cores físicos tiene el computador donde se ejecuta, ya que MPI se basa en la programación de memoria distribuida, donde cada proceso tiene su propio espacio de memoria separado del resto. Requisito que cumplen los cores físicos de un computador.

Los cores lógicos se obtienen gracias a la tecnología SMT, que consiste en lanzar dos hilos de ejecución por núcleo físico, compartiendo el mismo espacio de memoria del chip. Por lo tanto, en la teoría estricta no se podría hacer uso de esta tecnología, ya que se basa en la compartición de recursos entre ambos hilos. Sin embargo, también se puede ejecutar programas paralelizados con MPI en cores lógicos de un computador con rendimiento satisfactorio, ya que, aunque compartan espacio de memoria con otro (gracias a la tecnología SMT), estos se comportan como si fueran distintos espacios de memoria. Aunque físicamente los dos hilos compartan memoria, en práctica y conceptualmente funcionan como dos hilos con memorias separadas.

Para poder desplegar más procesos que cores físicos tenga el computador sin que salte el aviso anterior, hace falta manejar el fichero hostfile que se menciona en la práctica.

Si se intenta ejecutar en más cores que cores lógicos tiene el computador donde se ejecuta (cores virtuales), el rendimiento del programa experimentará una pérdida de rendimiento y una variación y fluctuación constantes.

## EJERCICIO 1.2

A continuación se proporciona un programa ("prod\_escalar\_mpi.c") que es un ejemplo de cálculo simple en paralelo que realiza el producto escalar de dos vectores.

Estudiar el funcionamiento del programa. Para ello, cambiar inicialmente el número de elementos de los vectores a un valor muy pequeño (por ejemplo 6). Ejecutar el programa variando el número de procesos lanzados desde 1 hasta 3, comprobar que los resultados son correctos y estudiar el código para entender cómo se realiza el cálculo.

Modificar el programa añadiéndole una medición del tiempo de ejecución, usando para ello la función `MPI_Wtime`.

Medir el tiempo de ejecución del programa al variar el número de procesos con el que es lanzado desde 1 hasta 6 procesos.

¿Son lógicos los tiempos de ejecución que se observan?

El estudio detallado de código se comenta directamente en el script de programación con comentarios línea por línea explicando el proceso. De todas formas, como resumen general, el programa calcula de forma paralelizada la multiplicación escalar de dos vectores, y esta carga de trabajo la reparte entre los distintos elementos de computación (procesos) que hay en el clúster MPI definido por el comunicador por defecto. Una vez que cada nodo calcula de forma local su propio resultado, lo manda como mensaje al nodo maestro, que se encarga de sumar todos los resultados parciales y presentar el resultado final por pantalla al usuario.

Código fuente del programa modificado:

```
/* prod_escalar_mpi.c
 * CALCULO DEL PRODUCTO ESCALAR DE DOS VECTORES
 *
 * ENTRADA: NINGUNA.
 * SALIDA: PRODUCTO ESCALAR
 */
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define ELEMENTOS 6
/* ¡ATENCIÓN: EL N° DE ELEMENTOS DEBE SER DIVISIBLE POR EL N° DE PROCESOS (p)! */

float x[ELEMENTOS], y[ELEMENTOS]; // VECTORES (VARIABLES GLOBALES)

////////////////////////////////////

/* FUNCION QUE CALCULA EL PRODUCTO ESCALAR LOCAL (DE UN TROZO DE LOS VECTORES) --> se reparte el trabajo en cada proceso del cluster */
float prod_escalar_serie(
    float a[], // ENTRADA
    float b[], // ENTRADA
    float n /* ENTRADA: NUMERO DE ELEMENTOS */ ){
    int i;
    float suma = 0.0;
    for (i = 0; i < n; i++)
        suma = suma + a[i] * b[i];
    return suma;
}

////////////////////////////////////
```

[ Read 93 lines ]

```

// PROD_ESCALAR_SERIE */
}

int main(int argc, char** argv) {
    int mi_rango; // RANGO DE MI PROCESO
    int p; // NUMERO DE PROCESOS
    int n = ELEMENTOS; // NUMERO DE ELEMENTOS DE LOS VECTORES
    int n_local; // NUMERO DE ELEMENTOS DE CADA FRAGMENTO
    int inicio_vector_local; // INDICE DE INICIO DE CADA FRAGMENTO
    float suma_local; // PRODUCTO ESCALAR SOBRE MI INTERVALO
    float suma_total; // PRODUCTO ESCALAR TOTAL
    int fuente; // PROCESO QUE ENVIA RESULTADO DE SUMA
    int dest = 0; // DESTINATARIO: TODOS LOS MENSAJES con los resultados VAN A 0
    int etiqueta = 0;
    int i;
    MPI_Status status;

    MPI_Init(&argc, &argv); // INICIALIZA MPI

    /* INICIALIZA LOS VECTORES, ambos vectores tendrán los mismos valores (0,1,2,3,4...) */
    for (i = 0; i < n; i++)
        x[i] = y[i] = i%5;

    MPI_Comm_rank(MPI_COMM_WORLD, &mi_rango); // AVERIGUA EL RANGO DE MI PROCESO
    MPI_Comm_size(MPI_COMM_WORLD, &p); // AVERIGUA CUANTOS PROCESOS HAY
    //divide el número de elementos de los vectores entre el número de procesos -->
    //tocandole a cada proceso n_local elementos
    n_local = n/p;
    // el proceso 0, se encargará de los primeros n/p elementos, el proceso 1, de los siguientes n/p elementos, (empieza en n/p),
    //proceso 2 empieza por n/p*2...
    inicio_vector_local = mi_rango * n_local;
    //cada uno de los procesos ejecuta esta función, pasándoles distinto valores de parámetros
    suma_local = prod_escalar_serie(&x[inicio_vector_local], &y[inicio_vector_local], n_local);
    printf("MI RANGO = %d , SUMA LOCAL = %f\n", mi_rango, suma_local);
}

```

```

GNU nano 4.8 prod_escalar_mpi.c
//cada uno de los procesos ejecuta esta función, pasándoles distinto valores de parámetros
suma_local = prod_escalar_serie(&x[inicio_vector_local], &y[inicio_vector_local], n_local);
printf("MI RANGO = %d , SUMA LOCAL = %f\n", mi_rango, suma_local);

/* SUMA LAS CONTRIBUCIONES CALCULADAS POR CADA PROCESO */

// IMPORTANTE --> la suma de todos los resultados de los procesos se mandan al nodo maestro (rank=0) y se suman en él
//solo se ejecuta esto en el nodo maestro
if (mi_rango == 0) {
    //inicio el total, con el resultado local de este nodo
    suma_total = suma_local;
    //bucle donde recibo por mensaje MPI, los resultados de los demás procesos y los acumulo
    for (fuente = 1; fuente < p; fuente++) {
        //recibo los mensajes del resto d enodos
        MPI_Recv(&suma_local, 1, MPI_FLOAT, fuente, etiqueta, MPI_COMM_WORLD, &status);
        suma_total = suma_total + suma_local;
    }

    //IMPORTANTE --> si NO soy el nodo maestro, envío mi resultado por mensaje
} else {
    MPI_Send(&suma_local, 1, MPI_FLOAT, dest, etiqueta, MPI_COMM_WORLD);
}

/* MUESTRA EL RESULTADO POR PANTALLA, en el nodo maestro*/
if (mi_rango == 0) {
    printf("PRODUCTO ESCALAR USANDO p=%d TROZOS DE LOS VECTORES X E Y = %f\n", p, suma_total);
}

MPI_Finalize(); // CIERRA EL UNIVERSO MPI */
} /* MAIN */

```

Como debe ser, independientemente del número de procesos el resultado final es el mismo, lo único que varía es la carga de trabajo de cada proceso y por lo tanto los resultados locales de cada nodo.

La ejecución con los distintos números de procesos y sus resultados se pueden ver aquí:

```

aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ nano prod_escalar_mpi.c
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpicc prod_escalar_mpi.c -o prod_escalar_mpi
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 1 prod_escalar_mpi
MI RANGO = 0 , SUMA LOCAL = 30.000000
PRODUCTO ESCALAR USANDO p=1 TROZOS DE LOS VECTORES X E Y = 30.000000
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 2 prod_escalar_mpi
MI RANGO = 0 , SUMA LOCAL = 5.000000
PRODUCTO ESCALAR USANDO p=2 TROZOS DE LOS VECTORES X E Y = 30.000000
MI RANGO = 1 , SUMA LOCAL = 25.000000
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 3 prod_escalar_mpi
MI RANGO = 0 , SUMA LOCAL = 1.000000
MI RANGO = 1 , SUMA LOCAL = 13.000000
PRODUCTO ESCALAR USANDO p=3 TROZOS DE LOS VECTORES X E Y = 30.000000
MI RANGO = 2 , SUMA LOCAL = 16.000000

```

Nº de procesos: p=1	Producto escalar = 30.00
Nº de procesos: p=2	Producto escalar =30.00
Nº de procesos: p=3	Producto escalar =30.00

## Versión del programa midiendo el tiempo de ejecución:

```

GNU nano 4.8                                prod_escalar_mpi_tiempo.c
/* prod_escalar_mpi.c
 * CALCULO DEL PRODUCTO ESCALAR DE DOS VECTORES
 *
 * ENTRADA: NINGUNA.
 * SALIDA: PRODUCTO ESCALAR
 */

#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define ELEMENTOS 6
/* ¡ATENCIÓN: EL N° DE ELEMENTOS DEBE SER DIVISIBLE POR EL N° DE PROCESOS (p)! */

float      x[ELEMENTOS], y[ELEMENTOS];      // VECTORES (VARIABLES GLOBALES)

//////////

/* FUNCION QUE CALCULA EL PRODUCTO ESCALAR LOCAL (DE UN TROZO DE LOS VECTORES) --> se reparte el trabajo en cada proceso del cluster:
float prod_escalar_serie(
    float a[], // ENTRADA
    float b[], // ENTRADA
    float n /* ENTRADA: NUMERO DE ELEMENTOS */ ){
    int i;
    float suma = 0.0;
    for (i = 0; i < n; i++)
        suma = suma + a[i] * b[i];
    return suma;
}

```

```

GNU nano 4.8                                prod_escalar_mpi_tiempo.c

return suma;

//////////
} /* PROD_ESCALAR_SERIE */

int main(int argc, char** argv) {
    int      mi_rango;          // RANGO DE MI PROCESO
    int      p;                // NUMERO DE PROCESOS
    int      n = ELEMENTOS;    // NUMERO DE ELEMENTOS DE LOS VECTORES
    int      n_local;          // NUMERO DE ELEMENTOS DE CADA FRAGMENTO
    int      inicio_vector_local; // INDICE DE INICIO DE CADA FRAGMENTO
    float     suma_local;       // PRODUCTO ESCALAR SOBRE MI INTERVALO
    float     suma_total;       // PRODUCTO ESCALAR TOTAL
    int      fuente;           // PROCESO QUE ENVIA RESULTADO DE SUMA
    int      dest = 0;         // DESTINATARIO: TODOS LOS MENSAJES con los resultados VAN A 0
    int      etiqueta = 0;
    int      i;
    double    tiempo_inicio, tiempo_fin;
    MPI_Status status;

    MPI_Init(&argc, &argv);      // INICIALIZA MPI
    //inicializa marca de tiempo, para medir tiempo de ejecución
    tiempo_inicio = MPI_Wtime();

    /* INICIALIZA LOS VECTORES, ambos vectores tendrán los mismos valores (0,1,2,3,4...) */
    for (i = 0; i < n; i++)
        x[i] = y[i] = i%5;
}

```

```

aleingmar@DESKTOP-2IE63G  x  +  v
GNU nano 4.8                                prod_escalar_mpi_tiempo.c

x[i] = y[i] = i%5;

MPI_Comm_rank(MPI_COMM_WORLD, &mi_rango);    // AVERIGUA EL RANGO DE MI PROCESO
MPI_Comm_size(MPI_COMM_WORLD, &p);           // AVERIGUA CUANTOS PROCESOS HAY
//divide el número de elementos de los vectores entre el número de procesos -->
//tocandole a cada proceso n_local elementos
n_local = n/p;
// el proceso 0, se encargará de los primeros n/p elementos, el proceso 1, de los siguientes n/p elementos, (empieza en n/p),
//proceso 2 empieza por n/p*2...
inicio_vector_local = mi_rango * n_local;
//cada uno de los procesos ejecuta esta función, pasándoles distintos valores de parámetros
suma_local = prod_escalar_serie(&x[inicio_vector_local], &y[inicio_vector_local], n_local);
printf("MI RANGO = %d , SUMA LOCAL = %f\n", mi_rango, suma_local);

/* SUMA LAS CONTRIBUCIONES CALCULADAS POR CADA PROCESO */

// IMPORTANTE --> la suma de todos los resultados de los procesos se mandan al nodo maestro (rank=0) y se suman en él
//solo se ejecuta esto en el nodo maestro
if (mi_rango == 0) {
    //inicio el total, con el resultado local de este nodo
    suma_total = suma_local;
    //bucle donde recibo por mensaje MPI, los resultados de los demás procesos y los acumulo
    for (fuente = 1; fuente < p; fuente++) {
        //recibo los mensajes del resto de nodos
        MPI_Recv(&suma_local, 1, MPI_FLOAT, fuente, etiqueta, MPI_COMM_WORLD, &status);
        suma_total = suma_total + suma_local;
    }
}

```



```

GNU nano 4.8                                prod_escalar_mpi_tiempo.c
    MPI_Recv(&suma_local, 1, MPI_FLOAT, fuente, etiqueta, MPI_COMM_WORLD, &status);
    suma_total = suma_total + suma_local;
}

//IMPORTANTE --> si NO soy el nodo maestro, envío mi resultado por mensaje
} else {
    MPI_Send(&suma_local, 1, MPI_FLOAT, dest, etiqueta, MPI_COMM_WORLD);
}

/* MUESTRA EL RESULTADO POR PANTALLA, en el nodo maestro*/
if (mi_rango == 0) {
    printf("PRODUCTO ESCALAR USANDO p=%d TROZOS DE LOS VECTORES X E Y = %f\n", p, suma_total);
}
//marca de tiempo --> para medir tiempo ejecución
tiempo_fin = MPI_Wtime();
MPI_Finalize();                                // CIERRA EL UNIVERSO MPI */

//calcula el tiempo total de ejecución, en función de las dos marcas de tiempo
if(mi_rango==0) {
    double tiempo_total = tiempo_fin - tiempo_inicio;
    printf("Tiempo total de ejecución: %f segundos\n", tiempo_total);
}
} /* MAIN */

```

Resultados de la ejecución del programa con vectores de 6 elementos midiendo el tiempo de ejecución:

```

aleingmar@DESKTOP-2IE63G4: ~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 1 prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 30.000000
PRODUCTO ESCALAR USANDO p=1 TROZOS DE LOS VECTORES X E Y = 30.000000
Tiempo total de ejecución: 0.000029 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 2 prod_escalar_mpi_tiempo
MI RANGO = 1 , SUMA LOCAL = 25.000000
MI RANGO = 0 , SUMA LOCAL = 5.000000
PRODUCTO ESCALAR USANDO p=2 TROZOS DE LOS VECTORES X E Y = 30.000000
Tiempo total de ejecución: 0.000081 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 3 prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 1.000000
PRODUCTO ESCALAR USANDO p=3 TROZOS DE LOS VECTORES X E Y = 30.000000
MI RANGO = 1 , SUMA LOCAL = 13.000000
MI RANGO = 2 , SUMA LOCAL = 16.000000
Tiempo total de ejecución: 0.000080 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 4 prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 0.000000
MI RANGO = 1 , SUMA LOCAL = 1.000000
MI RANGO = 2 , SUMA LOCAL = 4.000000
MI RANGO = 3 , SUMA LOCAL = 9.000000
PRODUCTO ESCALAR USANDO p=4 TROZOS DE LOS VECTORES X E Y = 14.000000
Tiempo total de ejecución: 0.000256 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 5 prod_escalar_mpi_tiempo
-----
There are not enough slots available in the system to satisfy the 5
slots that were requested by the application:

    prod_escalar_mpi_tiempo

Either request fewer slots for your application, or make more slots
available for use.

```

```

aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ cp ../ej2/hostfile.conf ./hostfile.conf
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 5 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 0.000000
MI RANGO = 1 , SUMA LOCAL = 1.000000
MI RANGO = 4 , SUMA LOCAL = 16.000000
MI RANGO = 3 , SUMA LOCAL = 9.000000
PRODUCTO ESCALAR USANDO p=5 TROZOS DE LOS VECTORES X E Y = 30.000000
MI RANGO = 2 , SUMA LOCAL = 4.000000
Tiempo total de ejecución: 0.000356 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 6 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 0.000000
MI RANGO = 2 , SUMA LOCAL = 4.000000
MI RANGO = 4 , SUMA LOCAL = 16.000000
MI RANGO = 3 , SUMA LOCAL = 9.000000
MI RANGO = 5 , SUMA LOCAL = 0.000000
MI RANGO = 1 , SUMA LOCAL = 1.000000
PRODUCTO ESCALAR USANDO p=6 TROZOS DE LOS VECTORES X E Y = 30.000000
Tiempo total de ejecución: 0.000296 segundos

```

¿Son lógicos los tiempos de ejecución que se observan?

En el par de imágenes anteriores se puede observar como a medida que vamos aumentando en número de procesos en los que se paraleliza el programa, va aumentando también el tiempo de ejecución. Esto es lógico, ya que el paso de mensaje entre procesos se penaliza en tiempo, y la tarea que realiza cada nodo es tan pequeña, que en términos de tiempos no compensa. Se tarda más en compartir los mensajes entre procesos, que en que un solo proceso corra él solo toda la carga de trabajo.

Tiempo de ejecución del programa (con nº elementos =  $2*2*2*2*3*3*5*7*11*13 = 720720$ ) al variar el número de procesos (p) con el que es lanzado desde 1 hasta 6 procesos:

```

aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpicc prod_escalar_mpi_tiempo.c -o prod_escalar_mpi_tiempo
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 1 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 4324320.000000
PRODUCTO ESCALAR USANDO p=1 TROZOS DE LOS VECTORES X E Y = 4324320.000000
Tiempo total de ejecución: 0.004145 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 2 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 0 , SUMA LOCAL = 2162160.000000
PRODUCTO ESCALAR USANDO p=2 TROZOS DE LOS VECTORES X E Y = 4324320.000000
MI RANGO = 1 , SUMA LOCAL = 2162160.000000
Tiempo total de ejecución: 0.003846 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 3 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 1 , SUMA LOCAL = 1441440.000000
MI RANGO = 2 , SUMA LOCAL = 1441440.000000
MI RANGO = 0 , SUMA LOCAL = 1441440.000000
PRODUCTO ESCALAR USANDO p=3 TROZOS DE LOS VECTORES X E Y = 4324320.000000
Tiempo total de ejecución: 0.004267 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 4 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 1 , SUMA LOCAL = 1081080.000000
MI RANGO = 3 , SUMA LOCAL = 1081080.000000
MI RANGO = 0 , SUMA LOCAL = 1081080.000000
PRODUCTO ESCALAR USANDO p=4 TROZOS DE LOS VECTORES X E Y = 4324320.000000
MI RANGO = 2 , SUMA LOCAL = 1081080.000000
Tiempo total de ejecución: 0.004792 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 5 --hostfile hostfile.conf prod_escalar_mpi_tiempo
MI RANGO = 3 , SUMA LOCAL = 864869.000000
MI RANGO = 1 , SUMA LOCAL = 864861.000000
MI RANGO = 4 , SUMA LOCAL = 864879.000000
MI RANGO = 2 , SUMA LOCAL = 864866.000000
MI RANGO = 0 , SUMA LOCAL = 864854.000000
PRODUCTO ESCALAR USANDO p=5 TROZOS DE LOS VECTORES X E Y = 4324320.000000
Tiempo total de ejecución: 0.007040 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej1$ mpirun -np 6 --hostfile hostfile.conf prod_escalar_mpi_tiempo

```

Nº de procesos (p)	Tiempo de ejecución	Aceleración
1	0.004145	1.00
2	0.003846	1.08
3	0.004267	0.97
4	0.004792	0.86
5	0.007040	0.59
6	0.007784	0.53

¿Son lógicos los tiempos de ejecución que se observan?

De forma similar que en los resultados anteriores en la ejecución con 6 elementos, creemos que los tiempos tienen sentido:

El paso de información entre procesos es costoso a nivel de tiempo y la carga de trabajo de cada proceso no es suficiente para que compense de forma clara esto. Sin embargo, a diferencia de la ejecución con vectores de 6 elementos, la carga de trabajo es algo más alta, lo que hace que en el caso de 2 procesos si compense mínimamente la paralelización, llegando a producirse una aceleración de 1.08. En cambio, cuando fraccionamos más la carga de trabajo y añadimos más paso de mensaje, empieza a verse reducida esa rentabilidad y con ello, la aceleración del sistema.

## EJERCICIO 2.1

A continuación se presenta un programa en C que realiza una estimación numérica de la integral o área encerrada entre la gráfica de una función no negativa  $f(x)$ , dos líneas verticales situadas en posiciones  $x=a$  y  $x=b$  y el eje  $x$ , utilizando el “método de los trapecios”.

A partir de este programa, se pide que realice una versión paralela del mismo utilizando MPI. Para paralelizar el programa, se deben distribuir los datos entre los diferentes procesadores. En este caso, los datos son los trapecios dentro del intervalo  $[a,b]$ . La idea es que cada proceso (menos el de rango 0) estime el valor de la integral sobre un subintervalo y envíe el resultado al proceso de rango 0, que se encargará de sumar todos los resultados (y también computará el área del primer subintervalo).

Si tenemos en total  $n$  trapecios y  $p$  procesos (suponemos que  $n$  es mayor que  $p$  y además, por simplificar, que  $n$  es divisible por  $p$ ), asignaremos  $n/p$  trapecios a cada proceso. Así, cada proceso tendrá que aplicar la fórmula de los trapecios sobre un trozo del área total, de longitud  $h * n/p$ . El proceso de rango 0 calculará la suma de los trapecios que van desde  $x=a$  hasta  $x=a+h*n/p$ . El proceso de rango 1, la suma de los trapecios que van desde  $x=a+h*n/p$  hasta  $x=a+2h*n/p$ . Y así sucesivamente.

Un proceso de rango genérico  $i$  calculará la suma de los trapecios que van desde  $x=a+i*h*n/p$  hasta  $x=a+(i+1)*h*n/p$ . Así pues, para este proceso su valor “ $a\_local$ ” sería  $a+i*h*n/p$  y su valor “ $b\_local$ ” sería  $a+(i+1)*h*n/p$ . Asimismo, su número de trapecios a sumar, “ $n\_local$ ”, sería  $n/p$ .

El algoritmo paralelo sería por tanto:

1. Cada proceso calcula cuál es su intervalo de integración, en función del valor de su rango  $i$ .
2. Cada proceso estima la integral de  $f(x)$  sobre su intervalo usando la regla de los trapecios.
3. Cada proceso distinto del 0 envía su integral al proceso 0.
4. El proceso 0 suma los valores recibidos de los procesos individuales y muestra el resultado final.

AYUDA: Partir del código proporcionado (“integracion\_trapecios\_pi\_esqueleto.c”) completarlo.

Probar el programa obteniendo una estimación del valor de  $\pi$  a partir del área de medio círculo. Es decir, calcular el área de medio círculo de radio unidad. La función a integrar se obtiene teniendo en cuenta que la ecuación de un círculo centrado en el origen es  $x^2 + y^2 = r^2$ . Despejando  $y$ , resulta la función a integrar, que es:  $y = \sqrt{r^2 - x^2}$ . Sabemos que dicho área debe ser igual a  $\pi/2$  (ya que el área de un círculo de radio



unidad es  $\pi * r^2 = \pi$ ). Por lo tanto, multiplicando por 2 el valor calculado por nuestro programa obtendremos una estimación de  $\pi$ .

La explicación del código viene desarrollada en líneas de comentarios en el propio código fuente del programa.

```
GNU nano 4.8                                integracion_trapecios_pi.c
/* INTEGRACION NUMERICA POR EL METODO DE LOS TRAPECIOS
 *
 * ENTRADA: NINGUNA.
 * SALIDA: ESTIMACION DE LA INTEGRAL DESDE a HASTA b DE f(x)
 * USANDO EL METODO DE LOS TRAPECIOS CON n TRAPECIOS
 */
#include <stdio.h>
#include <math.h>
#include <mpi.h>

////////////////////////////////////

/* FUNCION QUE CALCULA LA INTEGRAL LOCAL de cada proceso (area del trapezio de su parte) */

float Trapecio(float local_a, float local_b, int local_n, float h) {
    /* ALMACENA EL RESULTADO EN integral */
    float integral = 0.0;
    float x;
    int i;
    /* Definimos la FUNCION QUE ESTAMOS INTEGRANDO (que llamamos para calcular el area local) */
    float f(float x);

    // CALCULA EL VALOR DE integral DESDE local_a HASTA local_b
    /*siguiendo la fórmula matemática de la práctica, el extremo inicial y final, se dividen entre dos en el sumando
    por lo que inicializamos la variable acumuladora con esto*/
    integral = f(local_a)/2.0 + f(local_b)/2.0;
}
```

```
GNU nano 4.8                                integracion_trapecios_pi.c
por lo que inicializamos la variable acumuladora con esto*/
integral = f(local_a)/2.0 + f(local_b)/2.0;

/* recorremos cada trapezio que le toca a cada proceso (depende del numero de procesos) y
vamos calculando el valor de la función pasandole cada uno de los valores de x, de cada uno de esos trapecios
(sin contar con los extremos, por eso empieza por i=1 y no l=0, y no llega a ejecutarse el i para el valor de x final)*/

for (i = 1; i < local_n; i++) {
    x = local_a + i * h;
    integral += f(x);
}

integral = integral * h;
return integral;
} /* TRAPECIO */

////////////////////////////////////

/* FUNCION QUE ESTAMOS INTEGRANDO */

float f(float x) {
    float return_val;
    /* CALCULA f(x) Y DEVUELVE SU VALOR */

    // la función a integrar es esta --> y=sqrt(1-x*x) (semicirculo de radio=1)
}
```

```
GNU nano 4.8                                integracion_trapecios_pi.c
// la función a integrar es esta --> y=sqrt(1-x*x) (semicirculo de radio=1)

return_val = sqrt(1.0 - x*x);
return return_val;
} /* f */

////////////////////////////////////

int main(int argc, char** argv) {
    int mi_rango; // RANGO DE MI PROCESO
    int p; // NUMERO DE PROCESOS
    float a = -1.0; // EXTREMO IZQUIERDO
    float b = 1.0; // EXTREMO DERECHO
    int n = 10000; // NUMERO DE TRAPECIOS
    float h; // LONGITUD DE LA BASE DEL TRAPECIO
    float local_a; // EXTREMO IZQUIERDO PARA MI PROCESO
    float local_b; // EXTREMO DERECHO PARA MI PROCESO
    int local_n; // NUMERO DE TRAPECIOS PARA EL CALCULO
    float suma_local = 0.0; // INTEGRAL SOBRE MI INTERVALO
    float suma_total = 0.0; // INTEGRAL TOTAL
    int fuente; // PROCESO QUE ENVIA RESULTADO DE INTEGRAL
    int dest = 0; // DESTINATARIO: TODOS LOS MENSAJES VAN A 0
    int etiqueta = 0;
    MPI_Status status;

    MPI_Init(&argc, &argv); // INICIALIZA MPI

    MPI_Comm_rank(MPI_COMM_WORLD, &mi_rango); // mi_rango <-- RANGO (id) DE MI PROCESO
```

```

GNU nano 4.8                                integracion_trapecios_pi.c
MPI_Comm_rank(MPI_COMM_WORLD, &mi_rango); // mi_rango <-- RANGO (id) DE MI PROCESO
MPI_Comm_size(MPI_COMM_WORLD, &p); // p <-- NUMERO TOTAL DE PROCESOS QUE PARTICIPAN

// CALCULA h: ANCHO DE CADA TRAPECIO --> divido todo el ancho de la figura por el numero de trapecios
h = (b - a) / n;

// CALCULA local_n: EL NUMERO DE TRAPECIOS POR PROCESO --> divido el n trapecios entre la cantidad de procesos que hay
local_n = n / p;

/* LONGITUD DEL INTERVALO DE INTEGRACION DE CADA PROCESO = local_n*h
 * ASI QUE MI INTERVALO COMIENZA EN: */
// cada proceso se encargará del ancho medido anteriormente, ya para repartirlo
// hay que definir donde empieza y termina cada proceso que será por donde empieza el siguiente
local_a = a + local_n * h * mi_rango;
local_b = a + local_n * h * (mi_rango + 1);

// cada proceso pasa estos datos y llama a la función trapezio (datos distintos para cada proceso)
suma_local = Trapecio(local_a, local_b, local_n, h);

/* EL PROCESO 0 RECIBE Y SUMA LAS INTEGRALES CALCULADAS POR CADA PROCESO */
// IMPORTANTE --> la suma de todos los resultados de los procesos se mandan al nodo maestro (rank=0) y se suman en él
// solo se ejecuta esto en el nodo maestro
if (mi_rango == 0) {

```

```

GNU nano 4.8                                integracion_trapecios_pi.c
if (mi_rango == 0) {

    // inicio el total, con el resultado local de este nodo
    suma_total = suma_local;

    // bucle donde recibo por mensaje MPI, los resultados de los demás procesos y los acumulo
    for (fuente = 1; fuente < p; fuente++) {

        // recibo los mensajes del resto de nodos
        MPI_Recv(&suma_local, 1, MPI_FLOAT, fuente, etiqueta, MPI_COMM_WORLD, &status);
        suma_total = suma_total + suma_local;
    }

    // IMPORTANTE --> si NO soy el nodo maestro, envío mi resultado por mensaje
} else {

    MPI_Send(&suma_local, 1, MPI_FLOAT, dest, etiqueta, MPI_COMM_WORLD);
}

/* MUESTRA EL RESULTADO POR PANTALLA */

if (mi_rango == 0) {
    printf("ESTIMACION USANDO n=%d TRAPECIOS,\n", n);

```

```

/* MUESTRA EL RESULTADO POR PANTALLA */

if (mi_rango == 0) {
    printf("ESTIMACION USANDO n=%d TRAPECIOS,\n", n);
    printf("DE LA INTEGRAL DESDE %f HASTA %f = %f\n", a, b, suma_total);
    printf("PI = %f\n", 2 * suma_total);
}

MPI_Finalize(); // CIERRA EL UNIVERSO MPI

return 0;
} /* MAIN */

```

Ejecución del programa:

```

aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ nano integracion_trapecios_pi.c
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpicc integracion_trapecios_pi.c -o integracion_trapecios_pi -lm
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 4 integracion_trapecios_pi
ESTIMACION USANDO n=10000 TRAPECIOS,
DE LA INTEGRAL DESDE -1.000000 HASTA 1.000000 = 1.570795
PI = 3.141590
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 1 integracion_trapecios_pi
ESTIMACION USANDO n=10000 TRAPECIOS,
DE LA INTEGRAL DESDE -1.000000 HASTA 1.000000 = 1.570796
PI = 3.141591

```

## EJERCICIO 2.2

Modificar el programa realizado en el ejercicio anterior añadiéndole una medición del tiempo de ejecución, usando la función `MPI_Wtime`.

Modificar también los datos de ejecución en el código fuente de modo que se evalúe:

- a) Extremo izquierdo:  $a = 1.0$
- b) Extremo derecho:  $b = 1000000.0$
- c) Número de trapecios:  $n = 300300000$ . (Es el resultado de  $2*3*5*7*11*13*10000$ ).
- d) Función que se integra:  $\log(\text{pow}(x, 10))$

El resultado del área debe ser un número de uno a varios centenares de millones, aunque los valores variarán en función del número de trapecios y de procesos usados debido a los errores de truncamiento y a que se ha usado una precisión pequeña (variables de tipo float). Esto es un ejemplo de que en los cálculos hay que prestar mucha atención a los errores de truncamiento. En este caso no lo haremos porque en este ejercicio nos vamos a centrar en medir el tiempo de ejecución con precisión.

Con estos datos, ejecutar el programa y anotar el tiempo de ejecución obtenido para un número de procesos desde 1 hasta NP, donde NP es igual al doble de núcleos físicos del procesador utilizado más dos ( $NP = 2 \times \text{cores\_físicos} + 2$ ). Representar una gráfica del tiempo de ejecución respecto al número de procesos.

La aceleración de un programa paralelo se define como el cociente entre el tiempo de ejecución del programa para un sólo procesador y el tiempo de ejecución con n procesadores. Con los datos anteriores, calcular la aceleración del programa para n desde 2 hasta NP procesos. Representar una gráfica de la aceleración respecto al número de procesos. Interpretar ambas gráficas.

La ejecución del programa paralelizado implementando los cambios mencionados en el enunciado sobre 1 a 4 procesos:

```
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ nano int*tiempo.c
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpicc integracion_trapecios_pi_tiempo.c -o integracion_trapecios_pi_tiempo -lm
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 1 integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 14302241.000000
Tiempo total de ejecución: 13.679714 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 2 integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 28604482.000000
Tiempo total de ejecución: 7.294148 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 3 integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 35755604.000000
Tiempo total de ejecución: 5.421489 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 4 integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 50057844.000000
Tiempo total de ejecución: 4.366750 segundos
aleingmar@DESKTOP-2IE63G4:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 5 integracion_trapecios_pi_tiempo
-----
There are not enough slots available in the system to satisfy the 5
slots that were requested by the application:
```

Ejecución en 5 a 10 procesos:

```

aleingmar@DESKTOP-2IE63G: ~/CARRERA/ASSB/practica-mpi/ej2$ nano hostfile.conf
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 5 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 64360084.000000
Tiempo total de ejecución: 3.616471 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 6 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 71511208.000000
Tiempo total de ejecución: 3.172835 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 7 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 85813448.000000
Tiempo total de ejecución: 2.854936 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 8 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 96236984.000000
Tiempo total de ejecución: 2.852808 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 9 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 107266808.000000
Tiempo total de ejecución: 2.770481 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 10 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 121569048.000000
Tiempo total de ejecución: 2.842286 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 9 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 107266808.000000
Tiempo total de ejecución: 2.848181 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 9 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 107266808.000000
Tiempo total de ejecución: 2.728256 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 9 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 107266808.000000
Tiempo total de ejecución: 2.703625 segundos
aleingmar@DESKTOP-2IE63G:~/CARRERA/ASSB/practica-mpi/ej2$ mpirun -np 10 --hostfile hostfile.conf integracion_trapecios_pi_tiempo
ESTIMACION USANDO n=300300000 TRAPECIOS,
DE LA INTEGRAL DESDE 1.000000 HASTA 1000000.000000 = 121569048.000000

```

Tiempos:

```

GNU nano 4.8
1 13.68
2 7.29
3 5.42
4 4.37
5 3.61
6 3.17
7 2.86
8 2.85
9 2.77
10 2.84

```

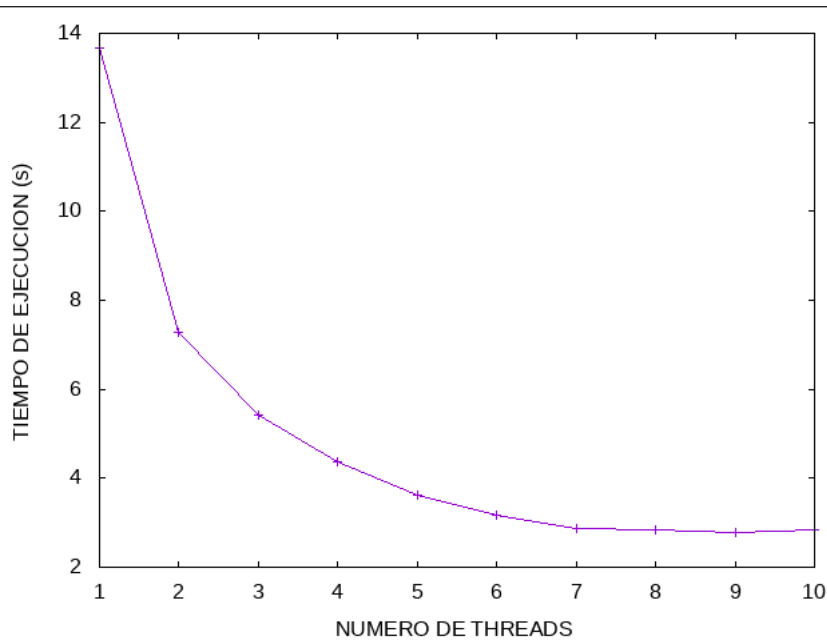
Aceleración:

```

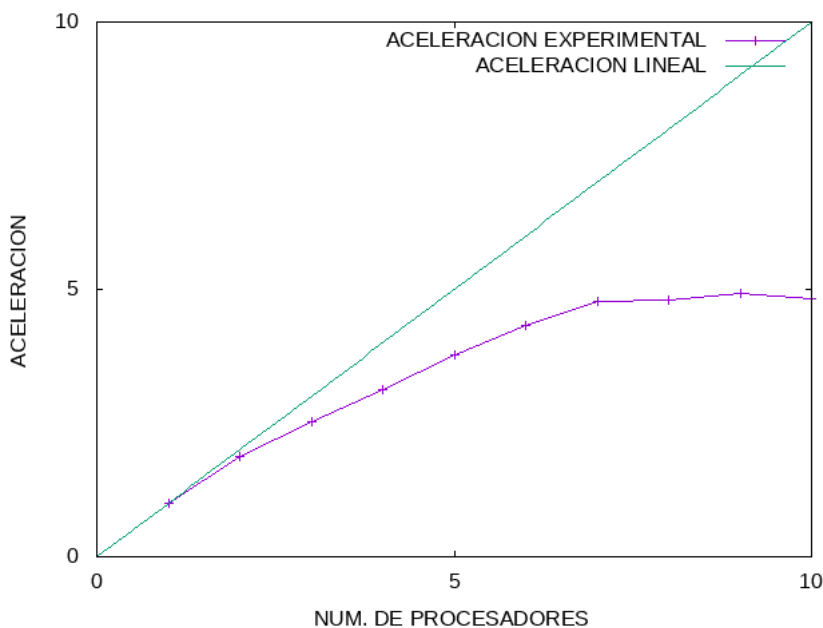
[1/2]
1 1
2 1.68
3 2.20
4 2.58
5 2.79
6 2.90
7 3.25
8 3.64
9 3.58
10 3.54

```

Representar una gráfica del tiempo de ejecución respecto al número de procesos



Representar una gráfica de la aceleración respecto al número de procesos.



Interpretar ambas gráficas

En las gráficas de aceleración y tiempo del programa en función del número de procesos paralelizados, se puede observar dos sucesos:

1. Que como tendencia clara, al dividir la carga de trabajo paralelizando este programa en sus cores lógicos (de 2 a 8 cores), vemos una reducción considerable en tiempos de ejecución y un aumento en la aceleración. Lo que implica claramente, una mejora del rendimiento del programa.
2. Que al superar el número de núcleos lógicos del ordenador en el que se ejecuta (en este caso 8), el rendimiento tiende a fluctuar, experimentando subidas y bajadas,

aunque generalmente se observa una tendencia a la reducción de rendimiento. Esto se debe a que el programa comienza a ejecutarse en núcleos virtuales y la CPU debe hacer un trabajo extra para “simular” la existencia y ejecución en estos cores extras.