Práctica 2
Paralelismo en sistemas de memoria distribuida

Objetivos:

- Aprender a paralelizar una aplicación en un sistema de memoria distribuida utilizando el estilo de programación paralela mediante "*paso de mensajes*".
- Estudiar el API de MPI y aprender a aplicar **distintas** estrategias de paralelismo con su aplicación.
- Aplicar métodos y técnicas propios de esta asignatura para estimar las ganancias máximas y las eficiencias del proceso de paralelización.

Introducción
En esta práctica vamos a estudiar e implementar el **paralelismo en memoria distribuida**, para ello utilizaremos el estándar **MPI**. Se deberá, por tanto, aprender a paralelizar una aplicación secuencial para mejorar el desempeño de esta en máquinas paralelas de memoria distribuida.

Especificación MPI:
MPI (*Message-passing interface)* es una especificación que establece las funciones de una biblioteca para el paso de mensajes entre múltiples procesadores. Esto permite la comunicación vía paso de mensajes entre nodos de un clúster de computadores que constituyen sistemas multicomputador de memoria distribuida[1]. En este escenario los datos utilizados por un proceso se mueven desde el espacio de direcciones de este al de otro proceso, de forma cooperativa, mediante las instrucciones recogidas en el estándar MPI.

Podemos distinguir **4 tipos de instrucciones** definidas en la especificación o API de MPI:

1. Para abrir, gestionar y cerrar las comunicaciones entre procesos corriendo en diferentes nodos.
2. Para transferir datos entre 2 procesos (uno a uno).
3. Para transferir datos entre múltiples procesos.
4. Instrucciones que permiten al usuario definir tipos de datos.

Las principales ventajas del uso de MPI son que permite establecer un estándar de paso de mensajes en arquitecturas multicomputadora portable, fácil de utilizar y que permite abstraer los detalles de bajo nivel propios de la gestión de este tipo de comunicaciones.

Existen multitud de librerías que implementan el estándar MPI (mpich, OpenMPI, …) en diferentes lenguajes como C, C++, Rust, Fortran, Python, Matlab, etc.. En este enlace pueden encontrar un tutorial con ejemplos de uso de MPI del laboratorio Lawrence-Livermore.

1 Consultad diapositivas de la Unidad 1 sobre la Taxonomía de Flynn.

Tarea 2.1: Estudio previo y entrenamiento con MPI (1 sesión)

En esta tarea se deberá buscar información acerca de las distintas implementaciones del estándar MPI sobre C/C++/Fortran y otros lenguajes como Rust, Python, Matlab, etc. Hay empresas importantes, como IBM, Cray o SGI, que tienen su propia implementación de MPI para ser usado en sus máquinas paralelas. También hay distintos proyectos de software libre que implementan MPI y para los que su uso está muy extendido tanto en la academia como en la industria (mpich, OpenMPI, …).

Tarea 2.1.1 Comente las características diferenciadoras de cada implementación.

En estos enlaces pueden encontrar tutoriales con ejemplos de uso de MPI.

- https://lsi2.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php *(Universidad de Granada)*
- https://www.youtube.com/watch?v=uiZU05CfZUg *(UPV)*
- https://www.youtube.com/watch?v=0FxDKhRxQqU *(UPV)*
- https://www.youtube.com/watch?v=y1yB7LTn6oA *(UPV)*

Tarea 2.1.2 Vamos a realizar un sencillo programa para comprobar el correcto funcionamiento de MPI entre, al menos, 2 nodos del laboratorio. El programa deberá realizar algún tipo de **comunicación interactiva** entre los dos nodos. Proponemos el siguiente programa, aunque se puede complicar/mejorar como estimen apropiado:

Ejemplo de programa: Un nodo, que llamaremos N1, solicita un valor de tipo entero a otro nodo (N2) en la red. Para ellos, N1 informará en la salida estándar lo siguiente “Solicitando la longitud del vector al nodo N2...”. N2, en este momento, debe preguntar al usuario algo como “Mensaje desde N1: Introduzca tamaño del vector: ”. Después de que N1 reciba correctamente dicho número (N), deberá calcular la suma de los N primeros números naturales y volver a enviar el resultado a N2. N2 por su parte, deberá imprimir el resultado en pantalla y avisar a N1 de que ha finalizado su tarea. N1 finaliza el programa paralelo.

Tarea 2.1.3 Implemente los siguientes programas en secuencial y en paralelo con MPI sobre una máquina paralela de memoria distribuida:

- Cálculo de PI como la integral definida entre 0 y 1 de la derivada del arcotangente. Mirad el enlace: https://lsi2.ugr.es/jmantas/ppr/tutoriales/tutorial_mpi.php?tuto=03_pi
- Cálculo de PI mediante el Método de Montecarlo. Véase el enlace https://www.geogebra.org/m/cF7RwK3H

Responda a las siguientes preguntas:

- ¿Es el problema totalmente paralelizable?
- ¿Cuál es la máxima ganancia en velocidad esperada?
- ¿Y la eficiencia?
-

Calcule los siguientes tiempos para cada programa:

- Tiempo secuencial del programa sin paralelizar.
- Tiempo paralelo del programa paralelizado ejecutado en 1 sólo nodo (N=1) y suponiendo que P=1,2,3,4,5,6 … Es decir, el programa está paralelizado, pero estamos simulando en la

máquina del laboratorio el comportamiento que tendrá una vez desplegado en varios nodos de la red. Represente gráficamente estos tiempos y comente los resultados. ¿Se observa ganancia en velocidad? ¿Por qué?

- Tiempo paralelo del programa ejecutado en paralelo en varios nodos del laboratorio. ¿Se observa ganancia en velocidad? ¿Por qué?

Tarea 2.1.3 Revise las transparencias de la **Unidad 3** que ya tiene en Moodle (de la número 28 en adelante). Concretamente, las que hablan de las distintas **alternativas de comunicaciones** entre varios procesos.

Busque cómo implementa MPI (concretamente mpich, la versión que tienen instalada en el laboratorio) las siguientes alternativas de comunicación. Explique también qué se hace en cada una de ellas:

- uno-a-todos:
  - Difusión
  - Dispersión
- todos-a-uno:
  - Reducción
  - Acumulación
- todos-a-uno, etc.

Desarrollo uno o varios programas **sencillos** para comprobar el correcto funcionamiento de estas funciones. Al menos deberán usar **6 alternativas de comunicación** de las proporcionadas por el API de mpich.

Tarea 2.1.4 Desarrolle 2 programas paralelos siguiendo el **modo de programación SPMD** y **MPMD**, respectivamente. Revise las transparencias de la Unidad 3 para recordar qué es un modo de programación paralela. Estos programas deben calcular, dado un vector de N elementos de tipo `double` o `float`, el máximo, el mínimo y la media de todos los elementos del vector. Compruebe en **nodos reales** que el programa funciona correctamente.

Tarea 2.2: Estudio y paralelización de una aplicación con MPI (1 sesión)

Adjunto a esta práctica encontrará el archivo *lab2_files.zip*, que contiene 3 ficheros:

- `main.c`: Código que deberá paralelizar
- `processing.h`: Fichero de cabeceras en C que usa *main.c*
- `libprocessing.so`: biblioteca dinámica con la implementación de varias funciones.

En el archivo `main.c` encontrará un sencillo código que llama a 4 funciones, de manera secuencial, que procesan un vector de 100 elementos (puede cambiar la longitud del vector si lo desea). La signatura de cada una de las 4 funciones a que se llaman tiene la siguiente forma:

```
void process_block1 (int grupo, int* vector, int LEN, int inicio, int fin);
```

donde:

- `grupo` es el número de grupo de prácticas al que pertenecen (Ej. Para el grupo de los lunes de 15-17h grupo vale 5, para el de los lunes de 17-19h vale 6, etc.),
- `vector` e `int_elementos` son el puntero a un vector de enteros de 32 bits y su longitud, respectivamente.
- Por último, `inicio` y `fin` son los índices (`0..LEN-1`) entre los que puede procesar el vector. Por ejemplo, si los quiero procesar sólo el elemento 8, indicaría `process_block1(GRUPO, vector, LEN, 7,7);`
- El procesamiento de cada elemento del vector (que está oculto) no depende de cálculos previos, sólo del elemento en cuestión del vector.
- Las funciones no devuelven nada, sólo realizan un cierto procesamiento que deberá acelerar.

NOTA: Debe compilar teniendo en cuenta la dependencia dinámica `libprocessing.so` (que no se encontrará por defecto en la ruta del sistema para estas bibliotecas). Deberá actualizar la variable de entorno `LD_LIBRARY_PATH` para **compilar** y **ejecutar** correctamente.

Por ejemplo, pruebe a compilar así (suponiendo que los 3 ficheros están en el mismo directorio):

```
gcc -Wall main.c -L. -lprocessing -o main.elf
```

Realice el **perfilado** que estime oportuno para estudiar el comportamiento en el tiempo del programa sin paralelizar (t_{seq}). Paralelice el código con MPI suponiendo que consta de, al menos, 2 y 3 nodos del laboratorio para calcular el *speedup*.

Se pide:

2.2.1. Calcule el tiempo secuencial de la versión sin paralelizar (t_{seq}).

2.2.2. Proponga una alternativa óptima para optimizar la velocidad usando MPI. Calcule, con esta implementación, el tiempo para P=1, 2 y 3 nodos **en la misma máquina del laboratorio**. ¿Se observa alguna ganancia?

2.2.3. Calcule la ganancia en velocidad ($S_p(p)$) para, al menos, 2 y 3 nodos distintos del laboratorio.

2.2.4. Calcule la eficiencia paralela en la ejecución paralela con nodos reales y usando varios procesos dentro del mismo nodo (como hizo en 2.2.2).

Notas a tener en cuenta:
Es importante argumentar todos los cambios necesarios para paralelizar la solución secuencial de partida. El **análisis del rendimiento debe ser exhaustivo y detallado** (diferentes tallas del problema, diferente número de nodos en el clúster...).

Notas generales a la práctica:

- La implementación realizada tendrá que poder ejecutarse **bajo el sistema operativo Linux del laboratorio**, y tomará 4 sesiones (8h)
- **Es obligatorio** entregar un ***Makefile*** con las reglas oportunas para compilar y limpiar su programa (`make clean`) de manera sencilla.
- Las/los estudiantes entregarán, además de la aplicación desarrollada, una memoria, estructurada según indicaciones del profesor, con la información obtenida.
- **Entrega**: la semana del 4 de noviembre antes del inicio de su sesión de prácticas correspondientes.
- Los trabajos teóricos/prácticos realizados han de ser originales. La detección de copia o plagio supondrá la calificación de "0" en la prueba correspondiente. Se informará la dirección de Departamento y de la EPS sobre esta incidencia. La reiteración en la conducta en esta u otra asignatura conllevará la notificación al vicerrectorado correspondiente de las faltas cometidas para que estudien el caso y sancionen según la legislación.

Ayuda: Pasos para compilar y ejecutar en el laboratorio (esta información puede variar de un curso a otro):

1. Todas las máquinas deben tener el **mismo usuario**. Todos los ejecutables deben tener el **mismo nombre** en cada máquina y estar en la **misma carpeta**.
2. Compilar empleando mpicc o mpic++:
   `mpicc programa.c –o programa`
   Recuerde que el programa, junto con sus dependencias, deben copiarse y ser accesibles en cada nodo de su multicomputador.
3. Crear el **archivo de hosts** [2]con las IPs o nombres de cada nodo de la red y copiar al resto de máquinas en el mismo directorio de trabajo donde se encuentran los ejecutables.
4. Cree un certificado de clave pública ssh en uno de los nodos:
   `ssh-keygen –t rsa` (crea la carpeta oculta ~/.ssh)
5. Copiar el archivo *`~/.ssh/id_rsa.pub`* en la carpeta *`~/.ssh`* del resto de máquinas y cambiar su nombre a *`authorized_keys`*. Si no existe la carpeta oculta `~/.ssh` en una máquina, crearla previamente ejecutando el comando *`ssh-keygen –t rsa`*.
6. Ejecutar el programa en la máquina donde se creó el certificado:
   `mpirun –mca plm_rsh_no_tree_spawn 1 –hostfile <archivo_hosts>`
   `–n <núm_procesos> ./programa`

Observaciones:

- Usar el mismo directorio en todas las máquinas: p.ej. Carpeta personal (`$HOME`)
- Abrir sesión con **el mismo usuario en todos los nodos** del cluster
- Puede ser necesario modificar los permisos de los archivos:
  - Archivo de hosts: `chmod 600 <archivo_hosts>`
  - Archivo ejecutable para permitir ejecución remota: `chmod o+x programa`

Ejemplo 1.

```
$ mpirun -hostfile hosts -n 4 ./Ejemplo1
Hola. Soy el procesador 2 de un total de 4
Hola. Soy el procesador 1 de un total de 4
Hola. Soy el procesador 0 de un total de 4
Hola. Soy el procesador 3 de un total de 4

Hola desde el procesador 1
```

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

// Ejemplo 'Hola' donde cada procesador
// se identifica
int main(int argc, char **argv)
{
   int id, nprocs;

   MPI_Init(&argc,&argv);
   MPI_Comm_size( MPI_COMM_WORLD, &nprocs );  // Devuelve el numero de procesos
                                              // en el COMM_WORLD (comunicador)
   MPI_Comm_rank( MPI_COMM_WORLD, &id );      // Identifica el id asignado

   printf("Hola. Soy el procesador %d de un total de %d\n", id, nprocs);

   if (id == 1)       // Selección de procesamiento en el procesador 1
   {
      // Solo se ejecuta en el procesador 1
      printf("\nHola desde el procesador %d\n", id);
   }

   MPI_Finalize();

   return 0;
}
```

2**Archivo de hosts**: archivo de texto plano que especifica el nombre o la IP de las máquinas que forman el supercomputador. Cada nombre/IP en una línea.

Ejemplo 2.

```c
#include <stdio.h>
#include "mpi.h"
#include <unistd.h>
#include <stdlib.h>

int sched_getcpu();

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);
  printf(" PID: %d Hello from process %d out of %d on %s processor %d \n",
           getppid(), rank, numprocs, processor_name, sched_getcpu());
  if (rank==2)
  {
     printf("¡Hola! desde el procesador %d\n", rank);
  }
  MPI_Finalize();
}
```