Práctica 4
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 aplicar **distintas** estrategias de paralelismo en 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 se va a continuar trabajando sobre el problema tratado en la práctica anterior. Deberán, por tanto, paralelizar la aplicación secuencial dada para mejorar el desempeño de esta en máquinas paralelas de memoria distribuida. Para ello se dispone de la biblioteca **MPI**.

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 de MPI:

1. Instrucciones para abrir, gestionar y cerrar las comunicaciones entre procesos corriendo en diferentes nodos.
2. Instrucciones para transferir datos entre 2 procesos (uno a uno).
3. Instrucciones 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++, Fortran, Python etc.. En este enlace podéis encontrar un tutorial con ejemplos de uso de MPI del laboratorio Lawrence-Livermore.

Desarrollo:
En este caso, se desarrollará la práctica en múltiples computadores (**mínimo 3**), y se usarán las estrategias de paralelización en entornos multicomputador que más se adecúen a la estrategia particular que siga cada grupo. Es importante argumentar todos los cambios necesarios para

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

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...). El informe debe incluir los aspectos estudiados en la **Unidad 3** de teoría: modo, estilo y estructura del programa paralelo, características de la comunicación, etc.

El análisis incluirá al menos los siguientes aspectos (con los gráficos necesarios):

- **Tiempos y aceleración** en términos de nodos y tamaño del problema/umbrales de error
- **Eficiencia paralela** en términos de los parámetros anteriores → ¿Cuál es la implementación más eficiente?
- **Comparación** con las implementaciones de la práctica anterior: secuencial y con OpenMP.

Caso especial: (opcional para subir nota)

Al igual que en la práctica 3, os planteamos el mismo caso especial para tratar de ejecutar en paralelo la parte de fine tuning y la de evaluación final del clasificador K-NN. El escenario multicomputador planteado en esta práctica es aún más propicio para implementar esa estrategia de paralelismo funcional.

Si se consigue implementar este caso de ejecución especulativa para explotar el paralelismo funcional, será muy útil comparar métricas obtenidas frente a la versión que no incluya esta característica (Tcpu, Speed-Up o Ganancia, etc.).

Notas generales a la práctica:

- La implementación realizada tendrá que poder ejecutarse bajo el sistema operativo Linux del laboratorio. Se recuerda que en conserjería se reparten toallitas hidroalcohólicas que permiten higienizar los equipos informáticos y el mobiliario.
- Se debe calcular la ganancia en **velocidad** y la **eficiencia** obtenida para varias configuraciones (distinto número de nodos).
- **Los grupos que implementen satisfactoriamente el paralelismo siguiendo el modo de programación paralela MPMD o híbrido tendrán hasta 1 punto extra en la práctica.**
- **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**: durante la semana del 20 de diciembre.
- 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 (actualizado a noviembre de 2021):

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();
}
```