ARQUITECTURA DE SISTEMAS DISTRIBUIDOS. 3º GII-TI. PRÁCTICA 7
Departamento de Arquitectura y Tecnología de Computadores
Universidad de Sevilla

PROGRAMACIÓN DE COMPUTADORES DE MEMORIA DISTRIBUIDA USANDO MPI

PREPARACIÓN

OBJETIVOS

En esta práctica se pretende que el alumno conozca la metodología a emplear para desarrollar aplicaciones paralelas para computadores paralelos de memoria distribuida y se familiarice con las particularidades que comporta este tipo de plataforma distribuida para la ejecución de los programas, su depuración y su rendimiento.

METODOLOGÍA

Se estudiarán los fundamentos de la programación utilizando el paradigma más empleado hoy en día en este tipo de plataforma: la biblioteca de funciones MPI (Message-Passing Interface), junto con el lenguaje de programación C.

Se realizarán programas para un PC con sistema operativo Linux, usando el compilador gcc junto con la biblioteca de funciones MPI. El alumno empleará un único PC para aprender los mecanismos de desarrollo y ejecución de programas paralelos que se emplean en plataformas distribuidas tales como un cluster. La ejecución en un PC de la aplicación con distinto número de procesos podrá beneficiarse de los múltiples núcleos del procesador.

ESTUDIO PREVIO

Estudiar los contenidos relacionados con la programación paralela de multiprocesadores con memoria distribuida impartidos en teoría.

Leer atentamente la presentación de diapositivas titulada "Introducción a la programación de computadores de memoria distribuida usando MPI" que forma parte del material de esta práctica.

Ambos contenidos podrán ser evaluados en un test de conocimientos previos al comienzo de esta práctica.

REALIZACIÓN EN LABORATORIO

Tras iniciar sesión en linux, abrir una consola de terminal y crear un directorio de trabajo para esta práctica, llamado `practica-mpi`.

Como documentación básica para trabajar usando el sistema operativo Linux, el estudiante puede emplear resúmenes de “Comandos UNIX” como:

Este más breve:

http://es.ccm.net/contents/311-comandos-de-linux

O este más completo:

https://es.wikipedia.org/wiki/Comandos_Bash

Crear o editar los archivos de código fuente empleando cualquier editor de texto de Linux (por ejemplo “gedit” o “nano”). Los comandos para compilar y ejecutar programas que usan MPI son los siguientes:

Compilación: `mpicc [-lm] -o nombre_archivo nombre_archivo.c`

Compilación: `mpic++ [-lm] -o nombre_archivo nombre_archivo.cpp`

El corchete no ha de ponerse, significa que es opcional.

El modificador opcional -lm hace que el compilador enlace el programa con las librerías matemáticas.

Ejecución: `mpirun -np X nombre_archivo`

Donde X es un número que indica el número de procesos que debe lanzar la aplicación.

EJERCICIO 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 PC-13-73: !Hola mundo!
Yo soy el proceso 1 de 3, corriendo en PC-13-73.
Yo soy el proceso 2 de 3, corriendo en PC-13-73.
```

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.

EJERCICIO 2:

Crear un archivo de código fuente C conteniendo el programa “saludos.c” (presentado en las transparencias incluidas en el material de esta práctica), compilarlo y ejecutarlo sobre distinto número de procesadores. ¿Qué ocurre al ejecutar el programa sólo para un procesador? ¿Por qué?

EJERCICIO 3:

Partiendo del programa “saludos.c”, crear un programa llamado “saludos_en_anillo.c” en el cual el proceso i le envía un saludo al proceso de su “derecha” (en forma cíclica). Piense en estas fórmulas: (i+1+p)%p, (i-1+p)%p , (donde el operador % es el módulo matemático o resto de la división entera). Prestar atención a cómo el proceso i calcula de quién debe recibir un mensaje. Contestar además:

a) El proceso i, ¿debe enviar primero su mensaje al proceso i+1 y después recibir el mensaje del proceso i-1, o a la inversa (primero recibir y luego enviar)? ¿Importa el orden?

b) ¿Qué ocurre cuando se ejecuta el programa sobre un único procesador (con un único proceso)?

Por ejemplo, para 3 procesos la salida por pantalla podría ser algo como:

```
Yo soy el proceso 1 y he recibido:   Saludos desde el proceso 0.
Yo soy el proceso 0 y he recibido:   Saludos desde el proceso 2.
Yo soy el proceso 2 y he recibido:   Saludos desde el proceso 1.
```

EJERCICIO 4:

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. (ver transparencias del material de esta práctica), y colocando las barreras adecuadas (piense en como debe medirse el tiempo correctamente), con MPI_Barrier(MPI_COMM_WORLD).

IMPORTANTE: cuando mida tiempos, comente todos las funciones de I/O (los printf () o similares) que están dentro del código a medir, ya que la llamada a I/O tarda mucho y es secuencial. Puede evitar que se compilen y ejecuten comentando tales líneas.

Para imprimir tiempos use algo como esto:

```
if (mi_rango == 0)
        printf ("TIEMPO DE EJECUCION = %lf s\n", tiempo_total);
```

Medir el tiempo de ejecución del programa al variar el número de procesos con el que es lanzado desde 1 hasta 6 procesos. Utilice la opción mpirun -np 5 -oversubscribe fichero.exe si el Sistema operativo le impide lanzar más de 4 procesos.

¿Son lógicos los tiempos de ejecución que se observan?

Programa prod_escalar_mpi.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   (2*2*2*2*3*3*5*7*11*13*50) // vale  720720*50
#define ELEMENTOS   (6) // una prueba con pocos elementos
/* ¡ATENCION: EL Nº DE ELEMENTOS DEBE SER DIVISIBLE POR EL Nº DE PROCESOS (p)! */

#define  PRINTF_ENABLE

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

/* FUNCION QUE CALCULA EL PRODUCTO ESCALAR LOCAL (DE UN TROZO DE LOS VECTORES) */
float prod_escalar_serie(
       float a[], // ENTRADA
       float b[], // ENTRADA
       float n   /* ENTRADA: NUMERO DE ELEMENTOS */ );

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 VAN A 0
    int         etiqueta = 0;
    int        i;
    MPI_Status  status;

    /* INICIALIZA LOS VECTORES */
    for (i = 0; i < n; i++)
          x[i] = y[i] = i%5;

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

    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
    n_local = n/p;
    inicio_vector_local = mi_rango * n_local;
    suma_local = prod_escalar_serie(&x[inicio_vector_local], &y[inicio_vector_local], n_local);
#ifdef PRINTF_ENABLE
    printf("MI RANGO = %d , SUMA LOCAL = %f\n", mi_rango, suma_local);
#endif
    /* SUMA LAS CONTRIBUCIONES CALCULADAS POR CADA PROCESO */
    if (mi_rango == 0) {
        suma_total = suma_local;
        for (fuente = 1; fuente < p; fuente++) {
            MPI_Recv(&suma_local, 1, MPI_FLOAT, fuente, etiqueta, MPI_COMM_WORLD, &status);
           suma_total = suma_total + suma_local;
        }
    } else {
        MPI_Send(&suma_local, 1, MPI_FLOAT, dest, etiqueta, MPI_COMM_WORLD);
    }

    /* MUESTRA EL RESULTADO POR PANTALLA */
    if (mi_rango == 0) {
#ifdef PRINTF_ENABLE
       printf("PRODUCTO ESCALAR USANDO p=%d TROZOS DE LOS VECTORES X E Y = %f\n", p, suma_total);
#endif
    }
    MPI_Finalize();                                 // CIERRA EL UNIVERSO MPI */
return 0;
} /* FIN DEL MAIN  */

/* FUNCION QUE CALCULA EL PRODUCTO ESCALAR LOCAL (DE UN TROZO DE LOS VECTORES) */
float prod_escalar_serie(float a[], float b[], 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;
} /* FIN DE LA FUNCION PROD_ESCALAR_SERIE  */
```

Nombre:	
Fecha y horario del grupo:	

ARQUITECTURA DE SISTEMAS DISTRIBUIDOS. 3º GII-TI. PRÁCTICA 7
PROGRAMACIÓN DE COMPUTADORES DE MEMORIA DISTRIBUIDA USANDO MPI.

RESULTADOS OBTENIDOS POR EL ALUMNO

Ejercicio 1:

Líneas adicionales añadidas al programa o líneas que se han modificado:	
¿Se puede lanzar un número de procesos mayor que el número de núcleos del procesador utilizado? ¿Qué ocurre en ese caso?	

Ejercicio 2:

¿Qué ocurre al ejecutar el programa "saludos" sólo para un proceso? ¿Por qué?	

Ejercicio 3:

Código fuente del programa:	
a) El proceso i, ¿debe enviar primero su mensaje al proceso i+1 y después recibir el mensaje del proceso i-1, o a la inversa (primero recibir y luego enviar)? ¿Importa el orden?	
b) ¿Qué ocurre cuando se ejecuta el programa sobre un único procesador (con un único proceso)?	

Ejercicio 4:

a) Funcionamiento del programa con nº elementos = 6:

Nº de procesos: p =1	Sumas parciales=	Producto escalar =
Nº de procesos: p =2	Sumas parciales=	Producto escalar =
Nº de procesos: p =3	Sumas parciales=	Producto escalar =

b)

Líneas adicionales añadidas al programa o líneas que se han modificado:	

Tiempo de ejecución del programa (con nº elementos = 2*2*2*2*3*3*5*7*11*13* 50 = 720720* 50) al variar el número de procesos (p) con el que es lanzado desde 1 hasta 6 procesos:

Nº de procesos (p)	Tiempo de ejecución	Aceleración
1		1.00
2		
3		
4		
5		
6		

¿Son lógicos los tiempos de ejecución que se observan?

¿Cuáles son los tiempos que intervienen?
