Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Grado en Ingeniería Informática

Arquitectura y Computación de Altas Prestaciones

Practica 1

Autora:
DE LA VIEJA LAFUENTE, CLAUDIA

Curso:
2023-2024

Índice

1. Ejercicio 1

Muestra cómo compilas el ejemplo 1 del seminario MPI en ATCGrid. Después, ejecútalo dentro de la cola de Slurm de forma asíncrona (comando sbatch con script inyectado –wrap) y dentro de la partición "acap". Juega con los parámetros N y n de forma que se ocupe primero sólo un nodo y luego dos. Pon además algún caso que Slurm vea inviable

```
[acap7@atcgrid ~]$ cd Practica_1/
[acap7@atcgrid Practica_1]$ ls
ej1  ejemplo1.c
[acap7@atcgrid Practica_1]$ sbatch -p acap -Aacap -N 1 -n 2 --wrap "mpirun -n 2 ./ej1"
Submitted batch job 228063
[acap7@atcgrid Practica_1]$ sbatch -p acap -Aacap -N 2 -n 2 --wrap "mpirun -n 2 ./ej1"
Submitted batch job 228064
[acap7@atcgrid Practica_1]$ ls
ej1  ejemplo1.c  slurm-228063.out  slurm-228064.out
```

Figura 1: Ejecución ejercicio 2

2. Ejercicio 2

El producto escalar de dos vectores, u y v, se define como la siguiente operación algebraica que devuelve un escalar

$$\mathbf{u} \cdot \mathbf{v} = u_1 \cdot v_1 + u_2 \cdot v_2 + \ldots + u_n \cdot v_n$$

Modifica el ejemplo 4 del seminario de MPI para convertirlo en un cálculo del producto escalar. Esta vez haz que todos los procesos trabajen, incluido el proceso 0 ó maestro. Explica los cambios y pon unos cuantos ejemplos de ejecución.

Para poder realizar este ejercicio he tenido que modificar la funcion masterTask que es la encargada de hacer el producto escalar entre dos vectores.

```
void masterTask(int rank, int numProcs, int dataSize){
    int* v = malloc(sizeof(int)*dataSize);
    int* u = malloc (sizeof(int) * dataSize);

    // Inicializar vectores u y v
    for(int i = 0; i < dataSize; i++){
        u[i] = i + 1;
        v[i] = i + 2;
    }

    // Enviar los trabajos a los trabajadores
    for(int i = 1; i < numProcs; i++){
        MPI_Send(u, dataSize, MPI_INT, i, NORMAL_TAG, MPI_COMM_WORLD);
        MPI_Send(v, dataSize, MPI_INT, i, NORMAL_TAG, MPI_COMM_WORLD);
    }

    // Calcular producto escalar local
    int resultado = 0;
    for(int i = 0; i < dataSize; i++){
        resultado += u[i] * v[i];
    }

    // Recibir resultados parciales de los trabajadores y sumarlos
    int buffer;
    for(int i = 1; i < numProcs; i++){
        MPI_Recv(&buffer, 1, MPI_INT, i, NORMAL_TAG, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
        resultado += buffer;
    }

    printf("El resultado es: %d\n", resultado);
    //------------------
    free(u);
    free(v);
}
```

Listing 1: Código de la función masterTask

Para poder hacer que todos los procesos trabajen he tenido que modificar el main para quitar el filtro que habia antes de que solo "trabajaban"los procesos master.

```c
int main(int argc, char* argv[]){
    int rank, numProcs, dataSize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

    if(argc != 2){
        printf("Error: Tama o de argumentos insuficiente");
        shutDown(numProcs);
    }else{
        dataSize = atoi(argv[1]);
        if(dataSize <= 0){
            printf("Error: Tama o de datos invalido\n");
            shutDown(numProcs);
        }else{
            if(rank == 0){ // Si el proceso es el maestro
                masterTask(rank, numProcs, dataSize);
            }else{
                worker(rank, numProcs, dataSize);
            }
        }
    }

    MPI_Finalize();
    return 0;
}
```

Listing 2: Código de la función main

```
claudia@claudia-laptop:~/Escritorio/UNIVERSIDAD/Curso 23-24/Segundo cuatri/ACAP/Practica 1$ mpiexec -n 4 ./ej2  0
Error: Tamaño de datos invalido
Error: Tamaño de datos invalido
Error: Tamaño de datos invalido
Error: Tamaño de datos invalido
claudia@claudia-laptop:~/Escritorio/UNIVERSIDAD/Curso 23-24/Segundo cuatri/ACAP/Practica 1$ mpiexec -n 4 ./ej2  1
El resultado es: 5
claudia@claudia-laptop:~/Escritorio/UNIVERSIDAD/Curso 23-24/Segundo cuatri/ACAP/Practica 1$ mpiexec -n 4 ./ej2  2
El resultado es: 17
claudia@claudia-laptop:~/Escritorio/UNIVERSIDAD/Curso 23-24/Segundo cuatri/ACAP/Practica 1$ mpiexec -n 4 ./ej2  3
El resultado es: 38
claudia@claudia-laptop:~/Escritorio/UNIVERSIDAD/Curso 23-24/Segundo cuatri/ACAP/Practica 1$ mpiexec -n 4 ./ej2  4
El resultado es: 70
claudia@claudia-laptop:~/Escritorio/UNIVERSIDAD/Curso 23-24/Segundo cuatri/ACAP/Practica 1$ mpiexec -n 4 ./ej2  5
El resultado es: 115
```

Figura 2: Ejecución ejercicio 2

3. Ejercicio 3

Se va a crear un servicio distribuido mediante MPI. Haz por tanto un programa que sólo conciba su ejecución con 4 procesos, cada uno con una funcionalidad distinta. El proceso 0 centralizará toda la entrada/salida y la interacción con el usuario. En concreto, solicitará en bucle infinito un comando, que será un número (en el rango [0,4]). Dependiendo del número indicado en el comando, se realizará una de estas actividades:

- **Si el número es 0, se saldrá del programa correctamente, acabando todos los procesos su ejecución.**
- **Si el número es 1, se le pedirá al usuario la longitud de un vector, y el proceso 0 pedirá entonces al proceso 1 un vector de números decimales aleatorios entre 0 y 1 con la longitud indicada. Dicho vector será mostrado por consola.**
- **Si el número es 2, se le pedirá al usuario una palabra, y el proceso 0 la enviará entonces al proceso 2. Éste sumará el valor ASCII de cada carácter y se lo devolverá al proceso 0, que mostrará dicho resultado.**
- **Si el número es 3, se le pedirá al usuario una palabra, y el proceso 0 la enviará entonces al proceso 3. Éste convertirá la cadena de caracteres a mayúscula y se la devolverá al proceso 0, que la mostrará por consola.**
- **Si el número es 4, se harán todas las acciones anteriores (menos la 0).**

Para la solución del siguiente ejercicio he creado 3 funciones auxialiares, cuales voy a explicar brevemente acontinuación. [utf8]inputenc

Función `vector_random`

Esta función genera un vector de números decimales aleatorios en el rango [0, 1]. Toma dos parámetros: `length` para la longitud del vector y `vector` como un array de tipo `double` donde se almacenan los números generados.

Función `suma_ascii`

Calcula la suma de los valores ASCII de los caracteres en una cadena de texto dada. Toma un parámetro: `palabra`, un puntero a una cadena de caracteres (string).

Función `pasar_mayusculas`

Convierte todos los caracteres de una cadena de texto a mayúsculas. Toma un parámetro: `palabra`, un puntero a una cadena de caracteres (string).

4. Ejercicio 4

El número es la relación entre la longitud de una circunferencia y su diámetro. Es un número irracional y una de las constantes más usada en Matemáticas, Física e Ingeniería. Hay distintos métodos para aproximar su valor, y uno de ellos es el siguiente: Se puede demostrar que: A su vez, esa integral se puede aproximar como una suma del área de N rectángulos: Donde cada rectángulo tiene una base fija x y una altura dada por F(x) = 4/(1+x²). Se proporciona un programa C que implementa este cálculo. Se pide que paralelices dicho cálculo en MPI para un número cualquiera de procesos, asumiendo que sólo el proceso 0 recibe parámetros, balanceando la carga equitativamente, y minimizando la interacción entre procesos. Incluye una medición del tiempo real ("wall time") empleado donde estimes oportuno y que ésta se muestre por consola. Prueba algunos valores y compara los tiempos con respecto a la versión secuencial. ¿Crees que merece la pena la versión paralela?

La función implementa la aproximación de pi mediante la programación paralela con MPI. El trabajo se distribuye entre diferentes procesos, identificados por su rango (rank) y su número total (numProcs). Cada proceso calcula su contribución local al sumatorio de la integral utilizando la fórmula que se nos ha proporcionado en el enunciado. La anchura de los rectángulos se define como la inversa de la cantidad de intervalos.

Posteriormente, se lleva a cabo una reducción mediante la función MPI_Reduce, donde cada proceso aporta su resultado parcial (almacenado en 'sum'). El proceso con rango 0 suma todos estos resultados parciales para obtener la suma total, almacenada en 'totalSum'.

Finalmente, se multiplica 'totalSum' por la anchura de los rectángulos para obtener la estimación final de pi, que se retorna como el resultado de la función.

Este enfoque permite una distribución eficiente del trabajo entre múltiples procesos, facilitando el cálculo paralelo y la mejora del rendimiento en entornos distribuidos.

```c
double piRectangles(int intervals, int rank, int numProcs) {
    double ancho = 1.0 / intervals;
    double sum = 0.0, x;

    // Calcular la parte local del sumatorio
    for (int i = rank; i < intervals; i += numProcs) {
        x = (i + 0.5) * ancho;
        sum += 4.0 / (1.0 + x * x);
    }

    // Reducci n: sumar los resultados parciales de todos los
       procesos
    double totalSum;
    MPI_Reduce(&sum, &totalSum, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);

    return totalSum * ancho;
}
```

Listing 3: Código para aproximar π con rectángulos y MPI.