UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

LABORATORIO MPI

Nifla Llallacachi, Manuel Angel

28 de Octubre de 2024

SOLUCIONES

Funciones MPI Utilizadas

- `MPI_Init`
  - **Descripción**: Inicializa el entorno MPI.
  - **Sintaxis**: `MPI_Init(argc, argv);`
- `MPI_Comm_rank`
  - **Descripción**: Obtiene el rango (identificador) del proceso en el comunicador.
  - **Sintaxis**: `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
- `MPI_Comm_size`
  - **Descripción**: Obtiene el número total de procesos en el comunicador.
  - **Sintaxis**: `MPI_Comm_size(MPI_COMM_WORLD, &numProces);`
- `MPI_Scatter`
  - **Descripción**: Distribuye partes de un arreglo desde el proceso raíz a todos los procesos en el comunicador.
  - **Sintaxis**: `MPI_Scatter(data, local_n, MPI_INT, local_data, local_n, MPI_INT, 0, MPI_COMM_WORLD);`
- `MPI_Reduce`
  - **Descripción**: Combina los datos de todos los procesos y los reduce a un solo resultado en el proceso raíz.
  - **Sintaxis**: `MPI_Reduce(histogram, global_histogram, valorMaximo, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);`
- `MPI_Finalize`
  - **Descripción**: Finaliza el entorno MPI y libera los recursos.
  - **Sintaxis**: `MPI_Finalize();`

0.1 Ejercicio 3.1.

Para este ejercicio se describe el funcionamiento de un programa escrito en C++ que utiliza la biblioteca MPI (Message Passing Interface) para calcular histogramas de manera distribuida. El programa genera un conjunto de datos aleatorios, los divide entre múltiples procesos y calcula un histograma local en cada proceso, que luego se combina para obtener un histograma global. Este enfoque permite la ejecución en paralelo, lo que mejora la eficiencia y reduce el tiempo de procesamiento.

0.1.1 Metodología

El programa se desarrolla utilizando los siguientes pasos:

1. **Inicialización de MPI**: Se inicia el entorno MPI y se determina el número total de procesos y el identificador de cada proceso (rank).
2. **Generación de Datos**: El proceso con rank 0 genera un conjunto de datos aleatorios, que se dividen posteriormente entre todos los procesos.
3. **Distribución de Datos**: Se utiliza la función `MPI_Scatter` para distribuir partes del conjunto de datos a cada proceso.
4. **Cálculo del Histograma Local**: Cada proceso calcula su propio histograma a partir de los datos que recibió.
5. **Reducción del Histograma Global**: Los histogramas locales se combinan en un histograma global utilizando la función `MPI_Reduce`.
6. **Impresión del Histograma Global**: Finalmente, el proceso con rank 0 imprime el histograma global.
7. **Liberación de Recursos**: Se liberan los recursos utilizados y se finaliza el entorno MPI.

0.1.2 Especificaciones de las Funciones

Funciones Personalizadas

- `generadorDatos`
  - **Descripción**: Genera un conjunto de datos aleatorios.
  - **Sintaxis**: `void generadorDatos(int *data, int N, int valorMaximo);`
- `histogramaLOcal`
  - **Descripción**: Calcula el histograma local a partir de los datos.
  - **Sintaxis**: `void histogramaLOcal(int *data, int N, int *histograma, int bins);`

0.2 Ejercicio 3.2.

Para este ejercicio se describe un programa escrito en C++ que utiliza la biblioteca MPI (Message Passing Interface) para estimar el valor de π mediante el método de Monte Carlo. El programa simula lanzamientos de "dardos" aleatorios en un cuadrado que circunscribe un cuarto de círculo, y calcula cuántos de estos dardos caen dentro del círculo. Esta proporción se utiliza para estimar el valor de π.

0.2.1 Metodología

El programa se desarrolla siguiendo estos pasos:

1. **Inicialización de MPI**: Se inicia el entorno MPI y se determina el número total de procesos y el identificador de cada proceso (rank).
2. **Definición de Lanzamientos Totales**: Se establece un número total de lanzamientos de dardos que se dividirá entre los procesos.
3. **Lanzamiento de Dardos**: Cada proceso calcula la cantidad de lanzamientos asignados y lanza los dardos, contando cuántos caen dentro del círculo.
4. **Reducción de Resultados**: Se utilizan funciones de reducción de MPI para sumar los resultados de los lanzamientos de todos los procesos.
5. **Estimación de Pi**: El proceso con rank 0 calcula y muestra la estimación final de π basada en los resultados combinados.
6. **Liberación de Recursos**: Se liberan los recursos utilizados y se finaliza el entorno MPI.

0.2.2 Especificaciones de las Funciones

Funciones Personalizadas

- `Dardos`
  - **Descripción**: Simula el lanzamiento de dardos y cuenta cuántos caen dentro del círculo.
  - **Sintaxis**: `double Dardos(int num);`
  - **Parámetros**:
    - `int num`: Número de dardos a lanzar.
  - **Retorno**: Retorna el número de dardos que caen dentro del círculo.

0.3 Ejercicio 3.3.

Se describe un programa escrito en C++ que utiliza la biblioteca MPI para realizar la suma de valores locales en múltiples procesos. El programa asigna a cada proceso un valor inicial basado en su rango y luego utiliza la comunicación entre procesos para calcular la suma total de esos valores de manera eficiente.

0.3.1 Metodología

El programa se desarrolla siguiendo los siguientes pasos:

1. **Inicialización de MPI**: Se inicia el entorno MPI y se determina el rango (identificador) de cada proceso, así como el número total de procesos.
2. **Asignación de Valores Locales**: Cada proceso asigna un valor local basado en su rango, de manera que el proceso 0 tiene un valor local de 1, el proceso 1 tiene un valor local de 2, y así sucesivamente.
3. **Comunicación entre Procesos**: Se utiliza un bucle para coordinar la comunicación entre procesos:
   - Si el proceso tiene un rango que es un múltiplo de $2 \times step$, espera recibir un valor de otro proceso.
   - Si el proceso puede enviar su suma local, la envía a otro proceso y se detiene.
4. **Cálculo de la Suma Global**: Al final del proceso de comunicación, el proceso con rango 0 imprime la suma total.
5. **Liberación de Recursos**: Se finaliza el entorno MPI y se liberan los recursos utilizados.

0.4 Ejercicio 3.5.

Se describe un programa escrito en C++ que utiliza la biblioteca MPI para realizar la multiplicación de una matriz por un vector en un entorno distribuido. El programa divide el trabajo de la multiplicación entre varios procesos para mejorar la eficiencia y reducir el tiempo de cómputo.

0.5 Metodología

El programa se desarrolla siguiendo los siguientes pasos:

1. **Inicialización de MPI**: Se inicia el entorno MPI y se determina el rango (identificador) de cada proceso, así como el número total de procesos.
2. **Validación de Tamaño**: Se verifica que el número de filas de la matriz m sea divisible por el número de procesos.
3. **Generación de Datos**: El proceso con rango 0 inicializa la matriz A y el vector x.
4. **Distribución de Datos**: La matriz A se distribuye entre todos los procesos utilizando `MPI_Scatter`, y el vector x se difunde a todos los procesos con `MPI_Bcast`.
5. **Multiplicación Local**: Cada proceso ejecuta la multiplicación de su parte local de la matriz A por el vector x.

6. **Recolección de Resultados**: Se reúnen los resultados parciales en el proceso 0 utilizando `MPI_Gather`.

7. **Liberación de Recursos**: Se finaliza el entorno MPI y se liberan los recursos utilizados.

0.6 Especificaciones de las Funciones

0.6.1 Funciones de Multiplicación

- `mult_matriz_vec`
  - **Descripción**: Multiplica una parte local de la matriz A por el vector x y almacena el resultado en y.
  - **Sintaxis**: `void mult_matriz_vec(int *L_A, int *Lx, int *Ly, int Lm, int n);`

0.7 Ejercicio 3.8.

En este último se describe un programa escrito en C++ que utiliza la biblioteca MPI para implementar un algoritmo de ordenamiento Merge Sort en un entorno distribuido. El programa divide un arreglo de números en partes que se procesan en paralelo, y finalmente combina los resultados para obtener un arreglo ordenado.

0.8 Metodología

El programa se desarrolla siguiendo los siguientes pasos:

1. **Inicialización de MPI**: Se inicia el entorno MPI y se determina el rango (identificador) de cada proceso, así como el número total de procesos.

2. **Generación de Datos**: El proceso con rango 0 genera un arreglo de números aleatorios y lo imprime.

3. **Distribución de Datos**: Se distribuye el arreglo original a todos los procesos utilizando `MPI_Scatter`, de modo que cada proceso recibe una parte del arreglo.

4. **Ordenamiento Local**: Cada proceso ejecuta el algoritmo de Merge Sort en su sección local del arreglo.

5. **Comunicación y Combinación de Resultados**: Se utilizan pasos de comunicación entre procesos para combinar los resultados parciales en un arreglo ordenado. Este proceso implica recibir datos de otros procesos y realizar la fusión de arreglos utilizando la función `merge`.

6. **Liberación de Recursos**: Se finaliza el entorno MPI y se liberan los recursos utilizados.

0.9 Especificaciones de las Funciones

0.9.1 Funciones de Ordenamiento

- `merge`
  - **Descripción**: Fusiona dos mitades de un arreglo ordenado.
  - **Sintaxis**: `void merge(int *arr, int *temp, int left, int mid, int right);`
- `merge_sort`
  - **Descripción**: Ordena un arreglo utilizando el algoritmo Merge Sort.
  - **Sintaxis**: `void merge_sort(int *arr, int *temp, int left, int right);`

0.10 GitHub

En el siguiente link se encuentra el repositorio de Github:

https://github.com/AngelNifla/EPCC$_{C}OMPUTACION_{P}ARALELA_{Y}{}_{D}ISTRIBUIDA_{2}$024/$tree/master/LABS/LAB$04