PRÁCTICA 9: EJEMPLO DE APLICACIÓN PRÁCTICA

Arquitecturas paralelas, Ingeniería informática, UBU.

Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde

6 diciembre 2023

ÍNDICE

Objetivos

Demostrar los conocimientos adquiridos desarrollando una aplicación un poco más compleja que permita posteriormente evaluar el rendimiento del sistema.

Introducción

Esta práctica aborda la implementación de MPI mediante el desarrollo de un programa de multiplicación de matrices en paralelo. Se deberá utilizar el MPI_Recv, la reserva dinámica de memoria y la posibilidad de aprovechar la topología cartesiana.

Código

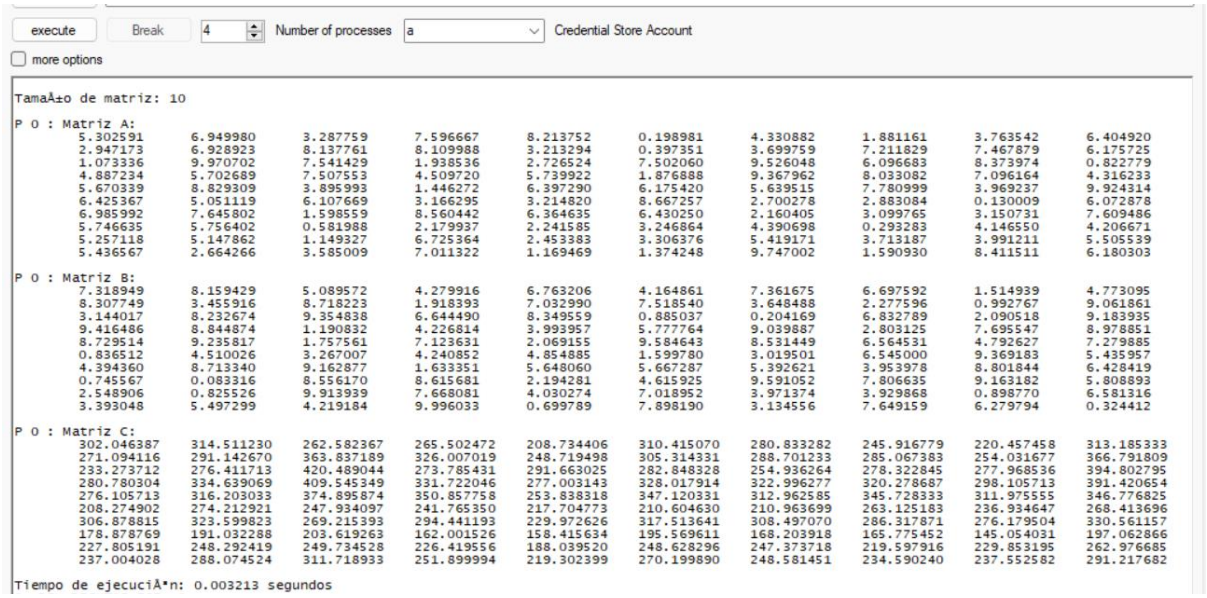
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <malloc.h>
#include <string.h>

void imprimir_matriz(float* matrix, int rows, int cols) {
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("\t %lf", matrix[i*cols+j]);
      }
      printf("\n");
          }
}
// Función para inicializar una matriz con valores aleatorios.
void initialize_matrix(float* matrix, int rows, int cols) {
          for (int i = 0; i < rows * cols; i++) {
      matrix[i] = (float)rand() / RAND_MAX * 10.0; // Valores aleatorios entre 0 y 10.
          }
}
// Función para realizar la multiplicación de matrices por bloques.
void multiply_matrices(float* A_block, float* B, float* C_block, int A_cols, int B_cols, int block_rows) {
          for (int i = 0; i < block_rows; i++) {
                for (int j = 0; j < B_cols; j++) {
                   float sum = 0.0;
                   for (int k = 0; k < A_cols; k++) {
                     sum += A_block[i * A_cols + k] * B[k * B_cols + j];
                   }
                   C_block[i * B_cols + j] = sum;
                }
          }
}
int main(int argc, char* argv[]) {
          int world_size, mirango;
          double start_time, end_time, total_time;

          // Inicialización de MPI.
          MPI_Init(&argc, &argv);
          MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &mirango);

srand(time(NULL) + mirango); // Semilla para números aleatorios.
int N;

if (mirango == 0) {// N= (argc > 1) ? atoi(argv[1]) : 0;
        printf("\nTamaño de matriz: ");
        fflush(stdout);
        scanf("%d", &N);

        while (N <= 0) {

            printf("\nPor favor, especifique un tamaño de matriz válido: ");
            fflush(stdout);
            scanf("%d", &N);
        }
}
MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

int rows_per_proc;
int extra_rows = 0;
if (mirango == 0) {
        // Cálculo del número de filas por proceso.
        if (world_size > N) {
            rows_per_proc = 1;
        }
       else {
            rows_per_proc = N / world_size;
            extra_rows = N % world_size; // Filas extra para manejar matrices no divisibles
exactamente entre los procesos.
       }
}
MPI_Bcast(&rows_per_proc, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&extra_rows, 1, MPI_INT, 0, MPI_COMM_WORLD);


// Reserva de memoria para bloques de la matriz A.
float* A_block = (float*)malloc(rows_per_proc * N * sizeof(float));
float* C_block = (float*)malloc(rows_per_proc * N * sizeof(float));
float* B = (float*)malloc(N * N * sizeof(float)); // Todos los procesos necesitan espacio para B.
float* A = (float*)malloc(N * N * sizeof(float));

// Inicialización de la matriz B por el proceso 0 y distribución a todos los procesos.
// Distribución de la matriz A entre todos los procesos.
if (mirango == 0) {

        initialize_matrix(A, N, N);
        printf("\nP %d : Matriz A: \n", mirango);
        imprimir_matriz(A, N, N);
        fflush(stdout);

        initialize_matrix(B, N, N);
        printf("\nP %d : Matriz B: \n", mirango);
        imprimir_matriz(B, N, N);
        fflush(stdout);
```

```
        }

        start_time = MPI_Wtime();

        // Dividir A entre los procesos, enviando a cada uno su bloque correspondiente.
         MPI_Scatter(A, rows_per_proc * N, MPI_FLOAT, A_block, rows_per_proc * N, MPI_FLOAT, 0,
MPI_COMM_WORLD);

        // Broadcast de la matriz B a todos los procesos.
         MPI_Bcast(B, N * N, MPI_FLOAT, 0, MPI_COMM_WORLD);

        // Multiplicación de matrices.
         multiply_matrices(A_block, B, C_block, N, N, rows_per_proc);

        // Recolectar los bloques de la matriz C en el proceso 0.
        float* C = NULL;
        if (mirango == 0) {
            C = (float*)malloc(N * N * sizeof(float));
        }

        // Usamos MPI_Gather para recolectar los resultados.
        MPI_Gather(C_block, rows_per_proc * N, MPI_FLOAT, C, rows_per_proc * N, MPI_FLOAT, 0,
MPI_COMM_WORLD);

        // Manejar las filas extras si las hay en el último proceso
        if (extra_rows > 0 && mirango == 0) {
            float* extra_A_block = (float*)malloc(extra_rows * N * sizeof(float));
            float* extra_C_block = (float*)malloc(extra_rows * N * sizeof(float));

            // Copiar las filas restantes al bloque extra
            memcpy(extra_A_block, A + rows_per_proc * world_size * N, extra_rows * N * sizeof(float));

            // Multiplicar las filas extra
            multiply_matrices(extra_A_block, B, extra_C_block, N, N, extra_rows);

            // Copiar los resultados de vuelta a c
            memcpy(C + (rows_per_proc * world_size * N), extra_C_block, extra_rows * N * sizeof(float));

            free(extra_A_block);
            free(extra_C_block);
        }

        end_time = MPI_Wtime();

        if (mirango == 0) {
            printf("\nP %d : Matriz C: \n", mirango);
            imprimir_matriz(C, N, N);

            printf("\nTiempo de ejecución: %f segundos\n", end_time - start_time);

            fflush(stdout);
        }


        // Liberación de memoria.
```

```
    free(A_block);
    free(C_block);
    free(B);
    free(C);

    // Finalización de MPI.
    MPI_Finalize();
    return 0;
}
```

Resultados



Descripción del Código

Funciones Auxiliares:

imprimir_matriz: Imprime una matriz en la salida estándar.
initialize_matrix: Inicializa una matriz con valores aleatorios.
multiply_matrices: Realiza la multiplicación de matrices por bloques.

Función Principal (main):

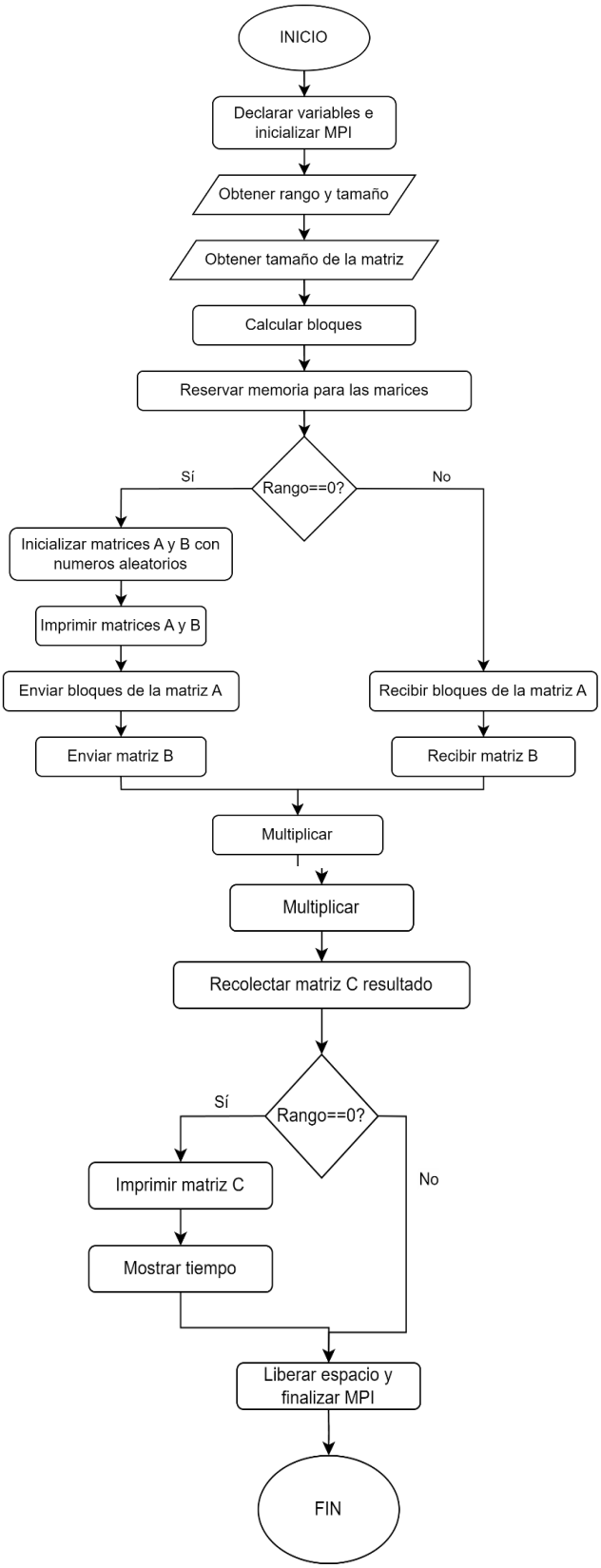
Inicia MPI y obtiene el tamaño del mundo y el rango del proceso.
Inicializa la semilla para la generación de números aleatorios.
En el proceso 0, solicita al usuario el tamaño de la matriz y realiza validaciones.
Utiliza MPI_Bcast para transmitir el tamaño de la matriz a todos los procesos.
Calcula el número de filas por proceso y las filas adicionales para manejar matrices no divisibles exactamente entre los procesos.

Reserva memoria para bloques de las matrices A y C, así como para la matriz B que es compartida por todos los procesos.
En el proceso 0, inicializa las matrices A y B con valores aleatorios y las imprime.
Inicia el tiempo de ejecución con MPI_Wtime.
Utiliza MPI_Scatter para distribuir la matriz A entre los procesos.
Utiliza MPI_Bcast para transmitir la matriz B a todos los procesos.
Realiza la multiplicación de matrices por bloques en cada proceso.
Utiliza MPI_Gather para recolectar los bloques de la matriz C en el proceso 0.
En el proceso 0, maneja las filas extras y muestra la matriz resultante C.
Imprime el tiempo de ejecución total.
Libera la memoria asignada y finaliza MPI.

Consideraciones Adicionales:

El código realiza una distribución de trabajo eficiente utilizando funciones MPI para la comunicación entre procesos.
Se utiliza la función MPI_Wtime para medir el tiempo de ejecución del programa.
La entrada y salida están principalmente concentradas en el proceso 0 para evitar duplicación.
Se utiliza la función memcpy para copiar bloques de memoria, facilitando la manipulación de submatrices.

Diagrama de flujo



Cuestiones

¿Sería posible aprovechar la potencia de la topología cartesiana para facilitar la resolución de este problema?

Sí, podría ser beneficioso aprovechar la topología cartesiana en la resolución del problema de multiplicación de matrices en paralelo. La topología cartesiana de MPI permite organizar los procesos en una cuadrícula lógica, lo que podría reflejar la estructura de las matrices a multiplicar. Cada proceso podría representar un bloque de la matriz y, mediante la topología cartesiana, se podría establecer una comunicación eficiente entre los procesos vecinos.

Plantear diferentes alternativas para resolver el problema y comparar su rendimiento previsible con el de la aplicación que se ha desarrollado.

Enfoque por Bloques:

Dividir cada matriz en bloques y asignar cada bloque a un proceso.
Utilizar la topología cartesiana para organizar los procesos en una cuadrícula 2D que refleje la estructura de los bloques.
Implementar comunicación entre procesos vecinos para la transmisión de bloques necesarios.

Enfoque de Fila-Fila:

Cada proceso maneja una fila completa de la matriz resultante.
Utilizar la topología cartesiana para organizar los procesos en una fila lógica.
La multiplicación de cada fila se realiza independientemente en paralelo.

Enfoque de Columna-Columna:

Cada proceso maneja una columna completa de la matriz resultante.
Utilizar la topología cartesiana para organizar los procesos en una columna lógica.
La multiplicación de cada columna se realiza independientemente en paralelo.

Enfoque Híbrido:

Combinar enfoques anteriores según la estructura de las matrices y la topología cartesiana. Por ejemplo, si las matrices son grandes, se podrían dividir en bloques y asignar a procesos, y luego usar la topología cartesiana para organizar esos bloques.

Conclusión

La práctica destaca la utilidad de MPI en el desarrollo de aplicaciones paralelas, enfocándose en la multiplicación de matrices. La introducción de la topología cartesiana y estrategias de asignación de trabajo proporciona perspectivas clave. La importancia de la

reserva dinámica de memoria se subraya para manejar eficientemente matrices grandes. En resumen, la práctica promueve la aplicación práctica de conceptos MPI avanzados y habilidades en programación paralela.