Práctica 10: Medida del rendimiento

Arquitecturas paralelas, Ingeniería informática, UBU.

Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde

14 diciembre 2023

Índice

Objetivos

Evaluar el rendimiento del sistema y adquirir la capacidad de prever la potencia del sistema y extraerla logrando un compromiso entre el esfuerzo de programación y la optimización del código.

Introducción

Esta práctica aborda el análisis del rendimiento de una implementación de MPI de un programa de multiplicación de matrices en paralelo. Se deben utilizar diferentes métricas para realizar una correcta evaluación.

Código

```
/*
* Nombre: PracticaARPA10
* Descripción: En esta práctica se realiza la multiplicación de matrices de forma paralela.
* Autores: Sergio Buil Laliena, Sandro Martín Rubio, Rocío Esteban Valverde
* Fecha: 12/12/2023
* Versión: 1.0
*/

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

// 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;

	    N = atoi(argv[1]);

	    if (N <= 0) {
	       printf("\nError. Las dimensiones de la matriz no pueden ser =< 0 ");
	       fflush(stdout);

	       // Finalización de MPI.
	       MPI_Finalize();
	       return 1;
	    }
	}
	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.
	if (mirango == 0) {
```

```
            initialize_matrix(A, N, N);
            initialize_matrix(B, N, N);
        }

        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("\nNúmero de procesos %d. \tTiempo de ejecución: %f segundos\n", world_size, end_time -
        start_time);
            fflush(stdout);
        }
```

```
    // Liberación de memoria.
    free(A_block);
    free(C_block);
    free(A);
    free(B);
    free(C);

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

Descripción del Código

Funciones Auxiliares:

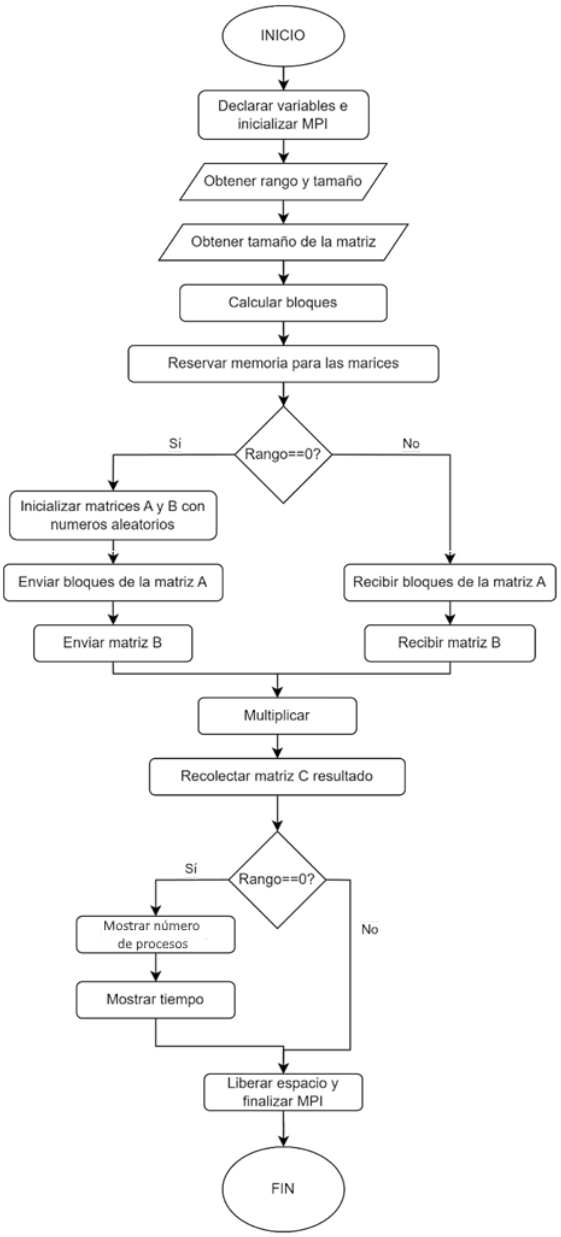
initialize_matrix: Inicializa una matriz con valores aleatorios.
multiply_matrices: Realiza la multiplicación de matrices por bloques.

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.
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 las añade a 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

No sabemos aún cómo responder porque no podemos comparar los tiempos de los resultados.

¿Está muy lejos el rendimiento obtenido del máximo teórico que se puede alcanzar?

Estimar las causas de la desviación observada.

Describir qué aspectos se deberían optimizar para obtener un mayor rendimiento.

Aún sin los resultados asumimos que nuestro programa podría ser optimizado repartiendo mejor las operaciones a realizar entre todos los procesos utilizando algunas de las técnicas explicadas en la anterior práctica. Sin embargo, no sabemos cuánto mejoraría eso los resultados del análisis y es importante que haya un equilibrio entre el esfuerzo de programación y la optimización del código.

Gráficas de rendimiento

Para obtener estas gráficas el proceso sería el siguiente:
Realizamos varias ejecuciones variando el número de procesos y el tamaño de las matrices.
Recopilamos los tiempos obtenidos y los ponemos en una tabla:

Nº Máquinas / NºFilas	1	2	3	4	5	6	7	8
1000								
2000								
3000								
4000								

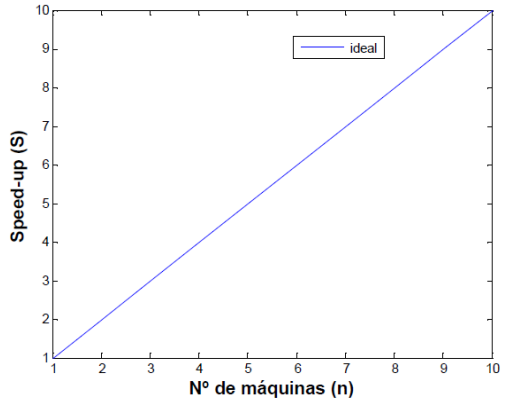
Los parámetros a evaluar son:
Speed-up (S): también llamado ganancia, que indica el grado de ganancia de velocidad de una computación paralela. Es el cociente entre T(1), tiempo empleado por un único procesador y T(n) el tiempo total T empleado por los n procesadores.

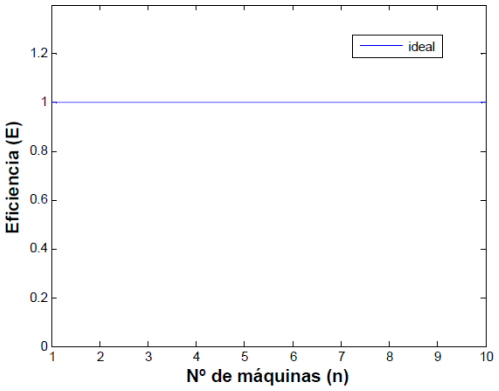
$$S = \frac{T(1)}{T(n)} \leq n \, .$$

y de la eficiencia (E). Determina el grado de aprovechamiento del sistema.

$$E = \frac{S}{n} = \frac{T(1)}{n \cdot T(n)} \leq 1 .$$

Si la eficiencia del sistema se mantiene constante y cercana a la unidad se dice que el sistema es escalable.
Las gráficas ideales serían:





Conclusión

La práctica destaca la utilidad de MPI en el desarrollo de aplicaciones paralelas, enfocándose en la multiplicación de matrices además se enfoca en la medida y el análisis del tiempo empleado. La introducción de 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 para minimizar el tiempo empleado en ejecutar el código con la mayor eficiencia usando el número de procesadores adecuado.