Multiplicación de Matrices en Paralelo: Un Enfoque Integrado de OpenMP

Alejandro Castro Martínez
Departamento de Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia
alejandrocastro@javeriana.edu.co

***Abstract*— In the field of high-performance computing, parallelizing computationally intensive tasks is essential to harness the full potential of modern computer systems. Matrix multiplication, a fundamental operation in various scientific and engineering domains, can significantly benefit from parallelization techniques. This article presents a study on parallel matrix multiplication using an integrated approach that combines OpenMP. Through extensive experimentation, execution times and speedup achieved with different matrix sizes and processor counts are analyzed. The findings provide valuable insights into the effectiveness of an integrated OpenMP approach for accelerating matrix multiplication and its implications for high-performance computing applications.**

Keywords*— *Parallel Computing, Matrix Multiplication, High-Performance Computing, OpenMP

***Resumen*— En el campo de la informática de alto rendimiento, la paralelización de tareas computacionalmente intensivas es esencial para aprovechar todo el potencial de los sistemas informáticos modernos. La multiplicación de matrices, una operación fundamental en diversos dominios científicos y de ingeniería, puede beneficiarse significativamente de técnicas de paralelización. Este artículo presenta un estudio sobre la multiplicación de matrices en paralelo utilizando un enfoque integrado que combina OpenMP. A través de una experimentación extensa, se analizan los tiempos de ejecución y la aceleración lograda con diferentes tamaños de matrices y recuentos de procesadores. Los hallazgos ofrecen ideas valiosas sobre la eficacia de un enfoque integrado OpenMP para acelerar la multiplicación de matrices y sus implicaciones en aplicaciones de informática de alto rendimiento.**

Palabras Clave*— *Computación Paralela, Multiplicación de Matrices, Computación de Alto Rendimiento, OpenMP

I. Introducción

En la era actual de la informática de alto rendimiento, la aceleración de cálculos computacionalmente intensivos se ha convertido en un imperativo para abordar problemas científicos y de ingeniería de envergadura. La multiplicación de matrices, una operación elemental en numerosos campos, es uno de los desafíos computacionales más comunes y fundamentales.

Su eficiente resolución no solo impulsa la investigación y la innovación en áreas tan diversas como la simulación climática, la bioinformática y la física cuántica, sino que también se encuentra en la base de muchas aplicaciones prácticas, desde el procesamiento de imágenes hasta el aprendizaje automático [1].

Este artículo se sumerge en el fascinante mundo de la multiplicación de matrices en un entorno paralelo, donde la explotación de múltiples núcleos y la colaboración entre procesadores se convierten en la clave para desbloquear todo el potencial de los modernos sistemas de cómputo. La investigación se centra en la combinación de dos enfoques de programación paralela ampliamente adoptados: OpenMP. Al integrar estas tecnologías, se abre un abanico de posibilidades para acelerar significativamente la multiplicación de matrices y otros cálculos matriciales complejos.

En este artículo, se presenta un estudio de la multiplicación de matrices en paralelo y se explora el impacto de esta estrategia en el rendimiento y la escalabilidad. La multiplicación de matrices es una tarea intrínsecamente paralelizable y, como tal, es un campo fértil para la experimentación y la optimización en la búsqueda de un cómputo más rápido y eficiente [1].

II. Marco Teórico

A. *Multiplicación de matrices*

La multiplicación de matrices es una subrutina fundamental de álgebra lineal, cuya implementación convencional se muestra en las ecuaciones presentadas a continuación [2].

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{1}$$

$$AB = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{12} + A_{22}B_{22} & A_{11}B_{12} + A_{12}B_{22} \end{bmatrix} \tag{2}$$

Esta fórmula matemática se puede describir como que cada posición de la matriz resultante se obtiene multiplicando término por término las entradas de la p-ésima fila de A y la q-ésima columna de B, y sumando estos n productos como se muestra a continuación [3].

$$[AB]_{p,q} = \sum_{i=0}^{n} a_{p,i} b_{i,q} \tag{3}$$

En términos más prácticos, para calcular cualquier elemento de la matriz resultante, se lleva a cabo la operación de

multiplicación entre todos los elementos de una fila en una matriz y los elementos correspondientes de la columna en la otra matriz, seguido por la suma de estos productos.

1) Implementación de un algoritmo computacional para la multiplicación de matrices

Al implementar este algoritmo en un programa computacional, el proceso inicial implica la formulación de un pseudocódigo, que se define de la siguiente manera:

```
1:  para p=0 hasta n hacer
2:      para q=0 hasta n hacer
3:          s = 0
4:          para i=0 hasta n hacer
5:              s = A[p][i] * B[i][q] + s
6:          fin para
7:          C[p][q] = s
8:      fin para
9:  fin para
```

Fig. 1. Pseudocódigo de la multiplicación de matrices tradicional. Adaptado de [4].

Basándonos en este pseudocódigo, procedemos a la etapa de codificación. En el contexto de este estudio, hemos optado por implementar el algoritmo en el lenguaje de programación C, conocido por su eficacia y versatilidad en el ámbito de la programación de alto rendimiento.

a) Implementación del algoritmo tradicional en el lenguaje de programación C.

La elección de este lenguaje se debe a su capacidad para gestionar la manipulación de matrices y la programación paralela de manera efectiva. La implementación en C nos permitirá explorar en detalle el rendimiento y la escalabilidad de la multiplicación de matrices en un entorno paralelo, donde la optimización y la eficiencia son fundamentales [5].

```c
/*
    Multiplicación de Matrices
*/
// Recorremos las filas de la primera matriz A
for (int p = 0; p < FILAS_MATRIZ_A; p++) {
    // Recorremos las columnas de la matriz B
    for (int q = 0; q < COLUMNAS_MATRIZ_B; q++) {
        int s = 0;
        // Iteramos a través de cada columna de la primera matriz A
        for (int i = 0; i < COLUMNAS_MATRIZ_A; i++) {
            // Realizamos la multiplicación y acumulación del resultado
            s += matrizA[p][i] * matrizB[i][q];
        }
        // Almacenamos el resultado en la matriz de resultado
        C[p][q] = s;
    }
}
```

Fig. 2. Código de la multiplicación de matrices tradicional implementado en el lenguaje de programación C. Adaptado de [4].

Dado que el análisis que se presenta a lo largo de este artículo se centrará en la multiplicación de matrices cuadradas, es relevante destacar que las variables **FILAS_MATRIZ_A**, **COLUMNAS_MATRIZ_B** y **COLUMNAS_MATRIZ_A** se considerarán como representativas de un número idéntico.

Esta suposición simplifica el análisis al asumir que las matrices de entrada tienen el mismo número de filas y columnas, lo que es común en aplicaciones que involucran operaciones matriciales fundamentales.

b) Implementación del algoritmo tradicional en el lenguaje de programación C con uso de apuntadores.

Un aspecto notable y enriquecedor dentro del contexto del lenguaje de programación C es el uso de apuntadores. Los apuntadores son herramientas fundamentales que permiten el acceso directo a posiciones de memoria en un programa, ofreciendo un alto grado de control y eficiencia en la manipulación de datos. En este artículo, exploramos la posibilidad de implementar el mismo algoritmo de multiplicación de matrices, pero incorporando el concepto de apuntadores [6].

La utilización de apuntadores en la multiplicación de matrices es un enfoque avanzado que puede resultar en un código más eficiente y compacto. En lugar de acceder a los elementos de las matrices mediante índices, como se hizo en la implementación previa, se aprovecha la capacidad de los apuntadores para acceder a las direcciones de memoria directamente.

```c
/*
    Multiplicación de Matrices con Uso de Punteros
*/
// Recorremos las filas de la primera matriz A
for (i = 0; i < SZ; i++) {
    // Recorremos las columnas de la matriz B
    for (j = 0; j < SZ; j++) {
        // Creación de apuntadores y variable acumuladora
        double *pA, *pB, S;
        S = 0.0;

        // Establecer punteros a la fila i de la matriz A y
        // la columna j de la matriz B
        pA = a + (i * SZ);
        pB = b + j;

        // Realizar la multiplicación de elementos correspondientes
        // y acumular el resultado
        for (k = SZ; k > 0; k--, pA++, pB += SZ) {
            S += (*pA * *pB);
        }

        // Almacenar el resultado en la matriz C
        c[i * SZ + j] = S;
    }
}
```

Fig. 3. Código de la multiplicación de matrices tradicional implementado en el lenguaje de programación C con uso de apuntadores.

c) Caracterización del algorítmo tradicional en el lenguaje de programación C con uso de apuntadores.

El fragmento de código presentado es una implementación detallada del algoritmo de multiplicación de matrices utilizando un enfoque basado en punteros en el lenguaje de programación C. Este enfoque está diseñado para optimizar la eficiencia del cálculo al reducir la sobrecarga asociada con el acceso a elementos de matriz utilizando índices.

El algoritmo opera sobre matrices de tamaño '**SZ x SZ**' y calcula cada elemento de la matriz resultante **C** mediante una serie de bucles anidados. En cada iteración de los bucles, se establecen punteros '***pA***' y '***pB***' que apuntan a la fila '***i***' de la matriz **A** y la columna '***j***' de la matriz **B**, respectivamente.

La multiplicación de elementos correspondientes se lleva a cabo en el ciclo *for* interno controlado por la variable '***k***'. Aquí, el puntero '***pA***' avanza a través de la fila de la matriz **A**, mientras que el puntero '***pB***' se desplaza verticalmente en la columna de la matriz **B**.

Luego, se realiza la multiplicación de elementos correspondientes y se acumula el resultado en la variable '***S***'. Este proceso se repite hasta completar todas las iteraciones, y el resultado se almacena en la matriz **C**.

Además, antes de ingresar al ciclo *for* interno, la variable '***S***' se inicializa en **0.0**. Este paso es esencial para garantizar una acumulación precisa de los productos de la multiplicación. A medida que se realizan las operaciones dentro del ciclo *for*, '***S***' se actualiza con la suma de los productos.

El uso de punteros en lugar de índices de matriz permite un acceso eficiente a la memoria y reduce la necesidad de cálculos repetitivos, lo que es esencial para mejorar el rendimiento en aplicaciones que involucran matrices de gran tamaño.

2) Implementación de un nuevo algoritmo computacional para la multiplicación de matrices

En el proceso de elaboración de este artículo comparativo, se ha concebido un nuevo algoritmo destinado a la multiplicación de matrices cuadradas. El propósito central de este algoritmo radica en optimizar el recorrido de los punteros, lo cual implica seguir las filas de las matrices cuadradas a multiplicar.

Con el fin de ilustrar este enfoque de manera más esclarecedora, se procede a expandir el ejemplo inicial presentado en la ecuación (1), a una multiplicación de matrices cuadradas de tamaño 3x3, como se describe a continuación.

$$AB = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = C \qquad (4)$$

$$C = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \qquad (5)$$

Cada uno de los elementos contenidos en la matriz resultante C, como se expuso previamente en la ecuación precedente, se deriva de un proceso que involucra tres operaciones de multiplicación y dos operaciones de suma, tal como se detalla en las ecuaciones subsiguientes.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} \qquad (6)$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} \qquad (7)$$
$$C_{13} = A_{11}B_{13} + A_{12}B_{23} + A_{13}B_{33} \qquad (8)$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} + A_{23}B_{31} \qquad (9)$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} \qquad (10)$$
$$C_{23} = A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33} \qquad (11)$$
$$C_{31} = A_{31}B_{11} + A_{32}B_{21} + A_{33}B_{31} \qquad (12)$$
$$C_{32} = A_{31}B_{12} + A_{32}B_{22} + A_{33}B_{32} \qquad (13)$$
$$C_{33} = A_{31}B_{13} + A_{32}B_{23} + A_{33}B_{33} \qquad (14)$$

Una vez hemos detallado exhaustivamente cada operación requerida para calcular cada elemento de la matriz resultante, el paso subsiguiente consiste en describir el proceso de indexación y cálculo necesario para la formulación del algoritmo que se ajuste al enfoque previamente delineado.

En la concepción de este algoritmo, resulta esencial el concepto de que los punteros de ambas matrices deben avanzar a través de ellas siguiendo el recorrido de las filas. Considerando esta premisa y las ecuaciones previamente presentadas, se deduce que no es factible efectuar todas las operaciones requeridas para una celda de la matriz resultante antes de continuar con la siguiente posición.

Al examinar detenidamente cada una de las ecuaciones expuestas, se percibe una estrategia de cálculo sistemática que puede facilitar la comprensión de la multiplicación de matrices en el contexto de este enfoque.

Para ilustrar este proceso, consideremos el cálculo de los valores de C_{11}, C_{12} y C_{13}, donde el objetivo es calcular el primer término de cada celda, manteniendo el apuntador de la matriz **A** fijo en la primera celda, mientras que el apuntador de la matriz **B** se desplaza horizontalmente a lo largo de la primera fila.

Este paso se repite en un ciclo, donde el apuntador de la matriz **A** se desplaza horizontalmente a la siguiente posición, mientras que el apuntador de la matriz **B** se desplaza verticalmente a la fila inmediatamente inferior y realiza el recorrido horizontal nuevamente.

Este proceso continúa hasta que el apuntador de la matriz **A** alcance la última posición horizontal, al mismo tiempo que el apuntador de la matriz **B** desciende a la última fila y realiza el recorrido horizontal.

Durante este proceso, se acumulan los resultados de las operaciones en la matriz resultante. Al finalizar la ejecución de todo el algoritmo, se habrán calculado y acumulado todos los valores requeridos en la matriz resultante.

Este método de cálculo es meticuloso y se basa en la coordinación precisa de las operaciones, lo que permite la obtención del resultado deseado en la multiplicación de las dos matrices cuadradas.

a) Implementación del nuevo algoritmo en el lenguaje de programación C con uso de apuntadores.

Teniendo en consideración la explicación previamente proporcionada, se procedió a la implementación de un código que se fundamenta en el algoritmo descrito, el cual busca optimizar el proceso de multiplicación de matrices mediante el uso de punteros. El código fuente original, que efectuaba una multiplicación de matrices de manera convencional, sirvió como punto de partida para esta implementación.

Las adaptaciones necesarias se realizaron con el propósito de dar cumplimiento a la estrategia detallada anteriormente. A continuación, se presenta el código resultante, desarrollado en el lenguaje de programación C.

```c
/*
    Multiplicación de Matrices con Uso de Punteros
    Nuevo algoritmo
*/
// Recorremos las filas de la primera matriz A
for (i = 0; i < SZ; i++) {
    double *pA;
    // Establecer puntero a la fila i de la matriz A
    pA = a + (i * SZ);

    // Iteramos a través de las filas de la matriz B
    for (j = 0; j < SZ; j++) {
        double *pB, S;
        // Establecer puntero a la fila j de la matriz B
        pB = b + (j * SZ);

        // Iteramos a través de los elementos de las matrices A y B
        for (k = 0; k < SZ; k++, pB++) {
            // Realizamos la multiplicación de elementos correspondientes
            // y acumulamos el resultado
            c[i*SZ+k] += (*pA * *pB);
        }

        // Incrementamos la posición del apuntador de la matriz A
        pA++;

    }
}
```

Fig. 4. Código de la nueva multiplicación de matrices implementado en el lenguaje de programación C con uso de apuntadores.

b) Caracterización del nuevo algoritmo en el lenguaje de programación C con uso de apuntadores.

El código proporcionado es un ejemplo de multiplicación de matrices utilizando punteros en el lenguaje de programación C. Este algoritmo se centra en la optimización del cálculo de la multiplicación de dos matrices cuadradas de tamaño '**SZ x SZ**', donde **SZ** es una constante definida previamente que indica el tamaño en una dimensión de las matrices cuadradas.

En este algoritmo, se utiliza un enfoque de puntero para recorrer las filas y columnas de las matrices **A** y **B** de manera eficiente. Se inicia con un bucle que recorre las filas de la matriz **A** (índice '***i***'). Para cada fila de **A**, se establece un puntero '***pA***' que apunta a la fila correspondiente en la matriz **A**. Luego, se itera a través de las filas de la matriz **B** (índice '***j***') y se establece un puntero '***pB***' que apunta a la fila correspondiente en la matriz **B**.

Dentro del bucle anidado, se realiza un tercer bucle para iterar a través de los elementos de las matrices **A** y **B** (índice '***k***'). En este bucle, se multiplican los elementos correspondientes de las matrices **A** y **B** y se acumulan en la matriz de resultado **C**. Este enfoque minimiza la necesidad de acceder repetidamente a las matrices originales, lo que mejora la eficiencia del cálculo.

Finalmente, se incrementa el puntero '***pA***' para avanzar a la siguiente fila de la matriz **A** y se repiten los pasos anteriores hasta completar el cálculo de la multiplicación de matrices.

B. OpenMP

OpenMP, conocido como Open Multi-Processing, es una poderosa API de programación que ha revolucionado la forma en que los programadores abordan la programación paralela en C y otros lenguajes [7].

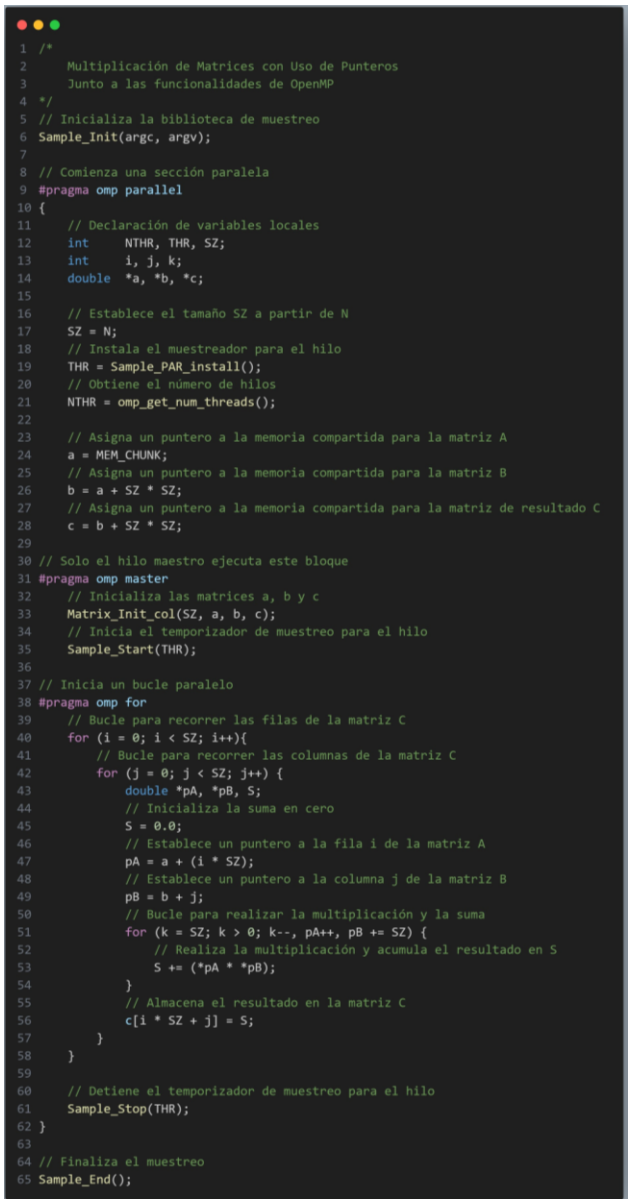
En un mundo donde la eficiencia computacional es esencial, OpenMP se ha convertido en una herramienta invaluable. Permite a los desarrolladores aprovechar al máximo los recursos de hardware disponibles, particularmente en sistemas con múltiples núcleos de CPU [7].

Con el propósito de garantizar la rigurosidad y precisión de los resultados obtenidos en este artículo, el proceso de experimentación desplegado incorpora de manera sistemática estas capacidades en cada uno de los algoritmos previamente expuestos.

A continuación, se presenta una caracterización detallada de dichos algoritmos, destacando la inclusión de los procesos de OpenMP.

a) Caracterización del algorítmo tradicional en el lenguaje de programación C con uso de apuntadores y funcionalidades de OpenMP.

A partir del algoritmo expuesto en la Fig. 3, se procede a realizar el proceso de adición de las funcionalidades proporcionadas por OpenMP como se detalla en la figura subsecuente.



```c
/*
    Multiplicación de Matrices con Uso de Punteros
    Junto a las funcionalidades de OpenMP
*/
// Inicializa la biblioteca de muestreo
Sample_Init(argc, argv);

// Comienza una sección paralela
#pragma omp parallel
{
    // Declaración de variables locales
    int     NTHR, THR, SZ;
    int     i, j, k;
    double  *a, *b, *c;

    // Establece el tamaño SZ a partir de N
    SZ = N;
    // Instala el muestreador para el hilo
    THR = Sample_PAR_install();
    // Obtiene el número de hilos
    NTHR = omp_get_num_threads();

    // Asigna un puntero a la memoria compartida para la matriz A
    a = MEM_CHUNK;
    // Asigna un puntero a la memoria compartida para la matriz B
    b = a + SZ * SZ;
    // Asigna un puntero a la memoria compartida para la matriz de resultado C
    c = b + SZ * SZ;

// Solo el hilo maestro ejecuta este bloque
#pragma omp master
    // Inicializa las matrices a, b y c
    Matrix_Init_col(SZ, a, b, c);
    // Inicia el temporizador de muestreo para el hilo
    Sample_Start(THR);

// Inicia un bucle paralelo
#pragma omp for
    // Bucle para recorrer las filas de la matriz C
    for (i = 0; i < SZ; i++){
        // Bucle para recorrer las columnas de la matriz C
        for (j = 0; j < SZ; j++) {
            double *pA, *pB, S;
            // Inicializa la suma en cero
            S = 0.0;
            // Establece un puntero a la fila i de la matriz A
            pA = a + (i * SZ);
            // Establece un puntero a la columna j de la matriz B
            pB = b + j;
            // Bucle para realizar la multiplicación y la suma
            for (k = SZ; k > 0; k--, pA++, pB += SZ) {
                // Realiza la multiplicación y acumula el resultado en S
                S += (*pA * *pB);
            }
            // Almacena el resultado en la matriz C
            c[i * SZ + j] = S;
        }
    }

    // Detiene el temporizador de muestreo para el hilo
    Sample_Stop(THR);
}

// Finaliza el muestreo
Sample_End();
```

Fig. 5. Código de la multiplicación de matrices tradicional implementado en el lenguaje de programación C con uso de apuntadores y las funcionalidades de OpenMP.

En este código, se presenta un ejemplo de multiplicación de matrices con el uso de la API OpenMP (Open Multi-Processing). El objetivo principal es explotar la capacidad de OpenMP para paralelizar el cálculo de la multiplicación de matrices, lo que puede acelerar significativamente el proceso en sistemas con múltiples núcleos de CPU.

El código comienza con la inicialización de la biblioteca de muestreo utilizando la función ***Sample_Init***, que puede admitir argumentos de línea de comandos (**argc** y **argv**).

A continuación, se entra en una sección paralela definida por ***#pragma omp parallel***, lo que permite la ejecución simultánea de múltiples hilos. En esta sección, se declaran varias variables locales que son utilizadas en el cálculo, incluyendo el número de hilos, el tamaño de las matrices y los punteros a las matrices **A**, **B** y **C**.

El código emplea directivas de OpenMP, como ***#pragma omp master***, para asegurarse de que ciertas tareas se ejecuten solo en el hilo maestro, como la inicialización de las matrices. Luego, se inicia el temporizador de muestreo con ***Sample_Start*** para medir el tiempo de ejecución.

El bucle paralelo ***#pragma omp for*** se encarga de distribuir la carga de trabajo de la multiplicación de matrices entre los hilos disponibles. Cada hilo calcula una porción de la matriz de resultados **C**.

Los punteros '***pA***' y '***pB***' se utilizan para acceder a los elementos de las matrices **A** y **B**, mientras que se realiza la multiplicación y acumulación en la variable **S**. Una vez que se completa la multiplicación y acumulación de una porción de la matriz **C**, se almacenan los resultados en la posición correspondiente.

Finalmente, se detiene el temporizador de muestreo con ***Sample_Stop*** y se finaliza la sección paralela. El código concluye con ***Sample_End***, que finaliza el muestreo y proporciona datos relevantes sobre el tiempo de ejecución del algoritmo en paralelo.

Esta implementación ilustra cómo OpenMP se puede utilizar para optimizar algoritmos de cómputo intensivo, como la multiplicación de matrices, al distribuir eficientemente el trabajo entre varios hilos de ejecución.

b) Caracterización del nuevo algoritmo en el lenguaje de programación C con uso de apuntadores y funcionalidades de OpenMP.

A partir del algoritmo expuesto en la Fig. 4, se procede a realizar el proceso de adición de las funcionalidades proporcionadas por OpenMP como se detalla en la figura subsecuente.

```c
/*
    Multiplicación de Matrices con Uso de Punteros
    Nuevo algoritmo junto a las funcionalidades
    OpenMP
*/
// Inicializa la biblioteca de muestreo
Sample_Init(argc, argv);

// Inicia una sección paralela
#pragma omp parallel
{
    // Declaración de variables locales
    int     NTHR, THR, SZ;
    int     i, j, k;
    double  *a, *b, *c;

    // Establece el tamaño SZ a partir de N
    SZ    = N;
    // Instala el muestreador para el hilo actual
    THR  = Sample_PAR_install();
    // Obtiene el número de hilos en la sección paralela
    NTHR = omp_get_num_threads();

    // Asigna un puntero a la memoria compartida para la matriz A
    a = MEM_CHUNK;
    // Asigna un puntero a la memoria compartida para la matriz B
    b = a + SZ * SZ;
    // Asigna un puntero a la memoria compartida para la matriz de resultado C
    c = b + SZ * SZ;

// Solo el hilo maestro ejecuta este bloque
#pragma omp master
    // Inicializa las matrices a, b y c
    Matrix_Init_col(SZ, a, b, c);

    // Inicia el temporizador de muestreo para el hilo actual
    Sample_Start(THR);

// Inicia un bucle paralelo que divide la iteración entre los hilos
#pragma omp for
    // Bucle para recorrer las filas de la matriz A
    for (i = 0; i < SZ; i++) {
        double *pA;
        // Establece un puntero a la fila i de la matriz A
        pA = a + (i * SZ);
        // Bucle para recorrer las filas de la matriz B
        for (j = 0; j < SZ; j++) {
            double *pB, S;
            // Establece un puntero a la columna j de la matriz B
            pB = b + (j * SZ);
            // Bucle para realizar la multiplicación y la suma
            for (k = 0; k < SZ; k++, pB++) {
                // Realiza la multiplicación y acumula el resultado en la matriz C
                c[i*SZ+k] += (*pA * *pB);
            }
            // Avanza al siguiente elemento en la fila de la matriz A
            pA++;
        }
    }

    // Detiene el temporizador de muestreo para el hilo actual
    Sample_Stop(THR);
}

// Finaliza el muestreo
Sample_End();
```

Fig. 6. Código de la nueva multiplicación de matrices implementado en el lenguaje de programación C con uso de apuntadores y las funcionalidades de OpenMP.

Este código demuestra el uso de OpenMP para paralelizar el cálculo de la multiplicación de matrices. Comienza con la inicialización de la biblioteca de muestreo a través de ***Sample_Init***. Luego, inicia una sección paralela con ***#pragma omp parallel***, lo que permite que múltiples hilos realicen cálculos de forma concurrente. En esta sección, se declaran variables locales y se establecen los tamaños y punteros de las matrices.

El uso de OpenMP se hace evidente con ***#pragma omp master***, que garantiza que ciertas tareas, como la inicialización de las matrices, se ejecuten solo en el hilo maestro. El temporizador de muestreo se inicia con ***Sample_Start*** para medir el tiempo de ejecución.

La sección paralela incluye un bucle ***#pragma omp for*** que divide la iteración entre los hilos disponibles. Cada hilo calcula una porción de la matriz de resultados C.

Se utilizan punteros, como '***pA***' y '***pB***', para acceder a los elementos de las matrices **A** y **B**, y la multiplicación y acumulación se realizan en la matriz **C**. OpenMP permite una distribución eficiente de la carga de trabajo, lo que resulta en un rendimiento mejorado en sistemas multiprocesador.

El código concluye con ***Sample_Stop*** para detener el temporizador de muestreo, finaliza la sección paralela y utiliza ***Sample_End*** para proporcionar información sobre el tiempo de ejecución.

En resumen, este código ejemplifica cómo OpenMP facilita la paralelización de algoritmos de cómputo intensivo, como la multiplicación de matrices, al distribuir eficientemente la carga de trabajo entre múltiples hilos de ejecución, lo que puede acelerar significativamente el procesamiento en sistemas con varios núcleos de CPU.

III. Ambiente de pruebas

El entorno informático empleado para llevar a cabo las pruebas se fundamenta en la infraestructura tecnológica proporcionada por la Facultad de Ingeniería de la Pontificia Universidad Javeriana, con el propósito de llevar a cabo una práctica exhaustiva y rigurosa.

Inicialmente, se procedió a la ejecución de los programas configurados con las funcionalidades de OpenMP en un equipo que cumplió con las siguientes especificaciones:

- **Sistema Operativo**: Linux Ubuntu 20.04
- **Arquitectura**: x86_64
- **Procesador**: Intel(R) Xeon(R) W-1290 CPU @ 3.20GHz
- **Número de CPU(s)**: 20
- **Memoria Total**: 65 GB

Este equipo proporciona un ambiente propicio y robusto para llevar a cabo las pruebas y experimentaciones con el objetivo de evaluar el rendimiento y la eficiencia de las implementaciones que hacen uso de OpenMP.

Las características técnicas detalladas del sistema subrayan su capacidad de cómputo, lo cual resulta fundamental para lograr resultados precisos y significativos en el contexto de la evaluación de algoritmos y procesos paralelos.

Además, para la ejecución de los programas relacionados tanto con OpenMP, se hizo uso de una herramienta de gran relevancia en el ámbito de la computación distribuida. Se trata de un clúster de procesamiento facilitado por el Departamento de Tecnologías de la Pontificia Universidad Javeriana y conocido como "**Condor**".

Este clúster de procesamiento de alto rendimiento se encuentra configurado con un total de 38 procesadores, los cuales operan en conjunto como un clúster, administrados mediante el software HTCondor.

Este enfoque en paralelismo y cómputo distribuido permite abordar tareas de gran envergadura y complejidad, lo que resulta esencial para la evaluación de algoritmos y procesos que requieren una escalabilidad significativa y un alto nivel de procesamiento.

Por último, se realizó la ejecución de los mismos programas en una máquina propia del investigador la cual cuenta con las siguientes características de hardware

- **Sistema Operativo**: Windows 11
- **Arquitectura**: x86_64
- **Procesador**: AMD Ryzen 7 5700X 8-Core Processor 3.40 GHz
- **Número de CPU(s)**: 16
- **Memoria Total**: 32 GB

Estos tres entornos de pruebas confirieron al proceso investigativo, expuesto a lo largo del presente artículo, una dimensión significativa que será objeto de análisis en las secciones subsiguientes.

En las secciones subsiguientes, se asignarán denominaciones específicas a los entornos de ejecución con el fin de facilitar su identificación precisa:

- ✓ Plataforma Linux Ubuntu 20.04: Designada como "***Equipo Ubuntu***"
- ✓ Clúster de Alto Rendimiento Condor: Identificado como "***Clúster Condor***"
- ✓ Estación de Trabajo Personal con sistema operativo Windows 11: Referido como "***Equipo Windows***".

IV. Metodología

En el curso de la investigación, la metodología adoptada se enfocó en la ejecución de dos programas codificados en C, aprovechando la interfaz OpenMP para la distribución eficiente de tareas y procesos.

La implementación de esta metodología se llevó a cabo en tres entornos de hardware diferentes, empleando scripts previamente desarrollados con el propósito de simplificar y estandarizar el proceso de ejecución.

La recopilación de datos se ejecutó de manera sistemática durante las ejecuciones, utilizando la salida estándar de los mencionados scripts y programas.

Este enfoque meticuloso se alinea directamente con el objetivo primordial de obtener resultados robustos y comparables en el análisis de rendimiento, respaldando así la coherencia de los hallazgos.

A continuación, se detalla la secuencia de pasos adoptada para la ejecución precisa de los programas y la captura acertada de información pertinente.

A. Compilación de los programas en C

El primer paso en este proceso es la compilación de los programas fuente. Para estandarizar este proceso se creó un archivo ***"Makefile"*** el cual realiza la compilación conjunta de los distintos archivos fuente.

Este archivo se encuentra en la ruta de la carpeta del proyecto: "./***Matrix_Multiplication/src/Makefile***". Para realizar la ejecución de este solo se debe ejecutar el comando: "***make***" estando ubicado en la carpeta "***src***" del proyecto.

La ejecución este comando mencionado resulta en la generación de ejecutables correspondientes a diversos programas desarrollados en C.

Dichos archivos ejecutables serán almacenados en el directorio "***BIN***", situado en la raíz del proyecto, siguiendo la siguiente nomenclatura:

- ➢ ***MM1c***: Archivo ejecutable correspondiente al algoritmo tradicional de multiplicación de matrices de filas por columnas.
- ➢ ***MM1f***: Archivo ejecutable correspondiente al algoritmo nuevo de multiplicación de matrices de filas por filas.

B. Ejecución de los archivos ejecutables individuales

Los dos archivos ejecutables generados demandan la especificación de ciertos parámetros para garantizar su ejecución adecuada:

- **Tamaño de las matrices a multiplicar**: Debe ser un número entero comprendido entre 0 y 10,240, representando el tamaño de las matrices cuadradas sujetas a la operación de multiplicación.
- **Cantidad de procesadores a utilizar**: Se espera un valor numérico entero que indique la cantidad de procesadores a emplear en el proceso de multiplicación. Este número está limitado por la capacidad de procesadores disponible en la máquina donde se ejecuta el programa.
- **Número 0**: Este parámetro, aunque carece de significado en el contexto de la operación, es esencial para la correcta ejecución del programa. Debe ser siempre el número 0 y se incorpora en la programación como requisito indispensable.

Al llevar a cabo una instancia de ejecución para ambos programas previamente compilados, el procedimiento se configura de la siguiente manera "***./MM1c 1000 10 0***".

Este comando activaría el programa que implementa el algoritmo tradicional de multiplicación de matrices. La ejecución se efectuaría sobre dos matrices cuadradas, cada una compuesta por 1,000 filas y 1,000 columnas, haciendo uso de 10 procesadores.

La ejecución de este programa produce un resultado de fácil interpretación. Cada línea de salida está compuesta por dos valores delimitados por el carácter "**:**". El primer número indica el procesador utilizado, mientras que el siguiente indica el tiempo de utilización de dicho procesador en microsegundos.

A continuación, se presenta una ilustración que exhibe la salida del comando ejecutado en una de las máquinas de prueba.

Fig. 7. Resultados de la ejecución del comando en una máquina.

C. *Ejecución de los experimentos*

En respuesta a la necesidad de realizar múltiples ejecuciones de diversos programas, se ha desarrollado un conjunto de scripts en el lenguaje de programación **Perl**. Este enfoque se ha adoptado para gestionar eficientemente la ejecución repetida de pruebas asociadas con algoritmos específicos de multiplicación de matrices.

Se han diseñado tres scripts distintos para abordar la diversidad de programas en cuestión. El primero, denominado "***lanzadorMM1c.pl***", se dedica exclusivamente a la ejecución de pruebas para el programa de multiplicación de matrices mediante el método tradicional de filas por columnas.

En paralelo, el segundo script, "***lanzadorMM1f.pl***", se especializa en ejecutar pruebas para el programa de multiplicación de matrices mediante el método de filas por filas. Por último, el tercer script, denominado "***lanzadorMM1.pl***", engloba y coordina la ejecución de pruebas para ambos algoritmos.

Las pruebas emprendidas por estos scripts abarcan diversas combinaciones de matrices de tamaño, incluyendo dimensiones como "100", "200", "400", "600", "800", "1000", "1500", "2000", "3000", "4000", "5000", "6000", "7000" y "8000". Además, se han considerado diferentes cantidades de procesadores para evaluar el rendimiento de los algoritmos, con opciones que van desde un solo procesador ("1") hasta configuraciones más avanzadas, como "2", "4", "8", "10", "14", "16" y "20".

En caso de que se requiera ajustar los parámetros, tanto el tamaño de las matrices a multiplicar como la cantidad de procesadores a emplear, se puede realizar dicha modificación de manera precisa y eficiente mediante la manipulación de las listas presentadas en la ilustración que sigue a continuación:

```
# Definición de listas de ejecutables, núcleos y tamaños de vector
@Ejecutables = ("MM1c","MM1f");
@cores = ("1","2","4","8","10","14","16");
@VectorSize= ("100","200","400","600","800","1000","1500","2000","3000","4000");
```

Fig. 8. Definición de las listas modificables de los scripts desarrollados en Perl.

En el fragmento de código proporcionado, se identifican tres listas, cada una desempeñando un papel específico en el proceso:

- ✓ Ejecutables:
  - Esta lista contiene los nombres de los programas ejecutables que servirán como base para la ejecución de las pruebas. Cada elemento de esta lista representa un programa específico destinado a realizar la multiplicación de matrices.
- ✓ Cores:
  - La lista denominada "Cores" almacena la cantidad de procesadores que se emplearán en cada una de las pruebas. Cada elemento de esta lista representa una configuración particular de núcleos de procesador utilizados durante la ejecución de las pruebas.
- ✓ VectorSize:
  - La lista "VectorSize" detalla el tamaño de las matrices cuadradas que serán sometidas a la operación de multiplicación. Cada valor en esta lista representa las dimensiones de las matrices, estableciendo la base para las variadas combinaciones de tamaños que serán objeto de prueba.

Los scripts mencionados están ubicados en la ruta "./***Matrix_Multiplication/scripts/execution/***". Para su ejecución, se requiere encontrarse en dicha carpeta, y además, es imperativo que exista la carpeta "./***Matrix_Multiplication/Soluciones/***".

Esta última carpeta actúa como el destino para almacenar las soluciones generadas por cada ejecución de los programas, representando así un componente esencial del proceso de experimentación.

D. *Generación de archivos para el clúster Condor*

En la fase de preparación para la ejecución de trabajos en el entorno del Clúster Condor, se llevó a cabo la generación de archivos "*submit*" destinados a cada ejecución requerida.

Estos archivos encapsulan directivas y parámetros esenciales que permiten al Clúster Condor orquestar la ejecución distribuida de programas específicos de manera eficiente. Un ejemplo paradigmático de la estructura de uno de estos archivos puede observarse a continuación.

```
executable = ../../BIN/MM1c
universe = vanilla
output = output/salida_1000_16_$(Process).txt
error = error/error_1000_16_$(Process).txt
log = log/log_1000_16_$(Process).txt
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
notification = never

arguments = 1000 16 0
queue 30
```

Fig. 9. Archivo submit de ejemplo para la ejecución del programa en el Clúster Condor.

Este archivo de envío para HTCondor se estructura en diversas secciones que desempeñan roles específicos en la configuración y ejecución del trabajo distribuido. A

continuación, se detallan y explican cada una de estas secciones para una comprensión integral:

- ***executable = ../../BIN/MM1c***:
    - Este parámetro define la ruta del ejecutable que se utilizará durante la ejecución. En este caso, se especifica que el programa ejecutable "**MM1c**" se encuentra en la ruta relativa "**../../BIN/**".
- ***universe = vanilla***:
    - El parámetro "universe" establece el entorno de ejecución. En este caso, se utiliza el valor "**vanilla**", que es el entorno estándar de HTCondor.
- ***output, error, log***:
    - Estos parámetros definen la ubicación y el formato de los archivos de salida, error y registro (log). Las expresiones "$(Process)" son variables que toman el valor del número del proceso en la ejecución.
- ***should_transfer_files = YES***:
    - Este parámetro indica que los archivos necesarios para la ejecución deben transferirse al sistema de archivos de ejecución del trabajador.
- ***when_to_transfer_output = ON_EXIT***:
    - Determina cuándo se deben transferir los archivos de salida de vuelta al sistema de archivos del solicitante. En este caso, la transferencia se realiza al finalizar la ejecución del trabajo.
- ***notification = never***:
    - Configura las notificaciones sobre el estado de la ejecución. En este caso, se establece en "never", indicando que no se recibirán notificaciones.
- ***arguments = 1000 16 0***:
    - Especifica los argumentos que se pasarán al programa ejecutable. En este ejemplo, se proporcionan los valores "1000", "16" y "0".
- ***queue 30***:
    - La instrucción "queue" se utiliza para encolar trabajos para ejecución. En este caso, se encolan 30 trabajos con la misma configuración definida anteriormente.

Este archivo submit para HTCondor configura de manera precisa los parámetros esenciales para la ejecución distribuida del programa "**MM1c**", permitiendo la ejecución de múltiples instancias con variaciones en los argumentos especificados.

Dado que los argumentos experimentales varían en cada ejecución, como se evidenció en el script desarrollado en el lenguaje Perl, se procede a la generación de un archivo por cada combinación posible de argumentos, según la combinatoria previamente delineada.

Con el objetivo de optimizar esta tarea de manera automatizada, se han desarrollado dos scripts de consola *Bash Shell*. Estos scripts se encargan de generar los archivos "*submit*" correspondientes a cada configuración de ejecución requerida y los someten a la ejecución en el entorno del Clúster Condor.

Este enfoque automatizado no solo agiliza el proceso de preparación y envío de trabajos al Clúster Condor, sino que también garantiza la exhaustividad en la cobertura de las diversas configuraciones experimentales, facilitando así la obtención de resultados representativos y comparables en el contexto de la evaluación de rendimiento.

En el contexto de la automatización del proceso, el script designado como "***createFiles.sh***" se ubica en el directorio "***./Matrix_Multiplication/scripts/condor_X***", donde la variable "**X**" adopta el valor de "**c**" para denotar archivos vinculados a la multiplicación tradicional de matrices y "**f**" para aquellos relacionados con el algoritmo de filas por filas.

La generación de archivos "*submit*" específicos para la multiplicación tradicional de matrices es ilustrada en la imagen adjunta. En el caso del algoritmo de filas por filas, la única variación reside en el ejecutable que se debe invocar.

Este enfoque modular y diferenciado según el algoritmo permite una adaptabilidad eficiente a las distintas configuraciones experimentales, otorgando flexibilidad y precisión en la preparación de los trabajos destinados al Clúster Condor.

```bash
#!/bin/bash

# Definir las listas de valores para los parámetros
VectorSize=("100" "200" "400" "600" "800" "1000" "1500" "2000" "3000" "4000"
            "5000" "6000" "7000" "8000")
cores=("1" "2" "4" "8" "10" "14" "16" "20")

# Directorio donde se generarán los archivos de sumisión
output_dir="submit_files"

# Crear el directorio si no existe
mkdir -p "$output_dir"

# Crear directorios de error, output y log para almacenar
# los respectivos archivos
mkdir -p "error"
mkdir -p "output"
mkdir -p "log"

# Número de ejecuciones por combinación
num_executions=30

# Iterar sobre las combinaciones de parámetros
for vs in "${VectorSize[@]}"; do
    for c in "${cores[@]}"; do
        # Nombre del archivo de sumisión
        submit_file="$output_dir/submit_${vs}_${c}.sub"

        # Crear el archivo de sumisión
        cat <<EOF > "$submit_file"
executable = ../../BIN/MM1c
universe = vanilla
output = output/salida_${vs}_${c}_\$(Process).txt
error = error/error_${vs}_${c}_\$(Process).txt
log = log/log_${vs}_${c}_\$(Process).txt
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
notification = never

arguments = $vs $c 0
queue $num_executions
EOF

        echo "Archivo de sumisión creado: $submit_file"
    done
done
```

Fig. 10. Script de consola bash shell para la creación de los archivos submit con todas las posibles combinaciones para el algoritmo tradicional de multiplicación de matrices.

Resulta fundamental destacar que este script no solo genera los archivos "*submit*" necesarios, sino que también configura las carpetas esenciales para garantizar la ejecución efectiva de los trabajos, incluyendo las carpetas "*error*", "*output*" y "*log*".

Asimismo, todos los archivos "*submit*" resultantes se almacenan de manera organizada en la carpeta "*submit*_files". Para adaptar el script a un mayor número de experimentos en cada ejecución, se realiza la modificación de la variable "*num_executions*" según el número de repeticiones deseado.

E. Ejecución de los jobs en el clúster Condor

En el contexto de la ejecución de trabajos en el Clúster Condor, la acción de enviar un archivo "*submit*" para su procesamiento se realiza mediante el comando "***condor_submit <archivo submit>***".

No obstante, considerando la generación de 115 archivos de este tipo por cada programa a ejecutar según las combinaciones mencionadas previamente, llevar a cabo la ejecución de manera individual sería un proceso laborioso y proclive a errores.

En respuesta a esta consideración, se ha desarrollado un script adicional en el lenguaje *Bash Shell*. Este script, diseñado con la finalidad de optimizar el proceso, ejecuta de manera automatizada todos los archivos contenidos en la carpeta "*submit_files*".

Este enfoque no solo simplifica el procedimiento operativo, sino que también agiliza la ejecución de trabajos en el entorno del Clúster Condor, contribuyendo así a una gestión más eficiente y efectiva de los recursos computacionales disponibles.

```bash
#!/bin/bash
# Directorio donde se encuentran los archivos de sumisión
submit_dir="submit_files"

# Iterar sobre los archivos de sumisión y enviarlos a Condor
for submit_file in "$submit_dir"/*.sub; do
    condor_submit "$submit_file"
    echo "Archivo de sumisión enviado: $submit_file"
done
```

Fig. 11. Script de consola bash shell para la ejecución de los archivos submit presentes en la carpeta “submit_files”.

V. RESULTADOS

Tras haber completado la ejecución de los diversos programas en cada uno de los tres entornos previamente delineados, se han obtenido los resultados que se exponen a continuación. Es esencial destacar que los resultados se centran en el tiempo de ejecución, medido en microsegundos, de cada uno de los algoritmos en los respectivos ambientes de pruebas.

Los resultados en su forma original están alojados en el directorio "***./Matrix_Multiplication/results/***". Este directorio se segmenta en seis subdirectorios, los cuales contienen los resultados generados por la ejecución de cada uno de los algoritmos en los distintos entornos de pruebas.

Estos archivos de resultados son de naturaleza textual, y su estructura sigue el formato de salida detallado en la ección IV.B, referente a la Ejecución de los archivos ejecutables individuales.

Cada iteración de prueba realizada con cada algoritmo y conjunto de parámetros experimentales se ejecutó de manera repetida en 30 ocasiones. En consecuencia, cada archivo de resultados almacenado refleja los tiempos de ejecución derivados de estos 30 experimentos.

Para gestionar este conjunto extenso de datos, se implementó un proceso de postprocesamiento mediante un programa desarrollado en Python. Este programa recorre sistemáticamente cada archivo en el directorio de resultados y genera archivos nuevos que contienen valores consolidados.

La consolidación de valores se efectúa seleccionando el tiempo máximo registrado en cada ejecución, ya que este representa el tiempo de ejecución total del algoritmo en el respectivo experimento.

Esta operación se automatiza mediante el programa Python mencionado, produciendo archivos con 30 valores, cada uno correspondiente al tiempo en microsegundos empleado en el experimento “***N***”. Este enfoque de postprocesamiento proporciona una visión más clara y representativa de los resultados, permitiendo un análisis más preciso y significativo.

La ubicación del script de Python es: "***./Matrix_Multiplication/scripts/getMaxTimes.py***" y los archivos consolidados se encuentran dentro de la carpeta "***./Matrix_Multiplication/consolidated/***"

En la etapa subsiguiente, utilizando los resultados consolidados, se efectúa el cálculo de valores promedio en conjunción con la medida de desviación estándar para cada ejecución.

Al igual que en la fase anterior, esta operación se ha automatizado mediante un script desarrollado en Python. Dicho script aglutina los archivos pertinentes asociados a un ambiente de pruebas específico y los compila en archivos de formato CSV para facilitar su análisis posterior.

A continuación, se exhibe un fragmento de uno de estos archivos, presentado en formato de tabla para una interpretación más accesible.

TABLE I. RESULTADOS DE LA EJECUCIÓN DEL ALGORITMO DE MULTIPLICACIÓN DE MATRICES TRADICIONAL EN EL AMBIENTE UBUNTU

Resultados de la ejecución del algoritmo de multiplicación de matrices tradicional en el ambiente Ubuntu			
Tamaño	***Procesadores***	***Tiempo Promedio***	***Desviación estándar***
100	1	677.03	29.51
100	10	115.77	3.82
100	14	103.17	9.82
100	16	100.03	3.38
100	2	356.8	34.37
100	20	88.1	8.98
100	4	192.2	26.98

Como se detalla en la tabla, la presentación de resultados se estructura de acuerdo con el tamaño de las matrices a multiplicar, seguido por la cantidad de procesadores utilizados. A continuación, se destacan el tiempo promedio de ejecución y, finalmente, la desviación estándar vinculada a esta métrica.

Dado que el análisis y la comparación efectiva de datos numéricos suelen beneficiarse considerablemente mediante representaciones gráficas, la próxima sección exhibirá diversas gráficas de interés derivadas de los archivos de resultados generados.

Este enfoque visual proporcionará una perspectiva más intuitiva y comprensible de las tendencias, patrones y variaciones presentes en los resultados, enriqueciendo así la interpretación de los hallazgos experimentales.

A. *Cálculo de la métrica speedup*

Otra métrica interesante para analizar es el "*speedup*". el "*speedup*" en el contexto de algoritmos distribuidos se refiere a la mejora en el rendimiento que se logra al ejecutar un algoritmo en un entorno distribuido en comparación con su ejecución en un sistema no distribuido o en una sola máquina.

El "*speedup*" se calcula típicamente como la relación entre el tiempo de ejecución del algoritmo en un sistema distribuido y el tiempo de ejecución en un sistema no distribuido. La fórmula general es:

$$Speedup = \frac{T_{nodistribuido}}{T_{distribuido}} \quad (15)$$

Donde:

- $T_{nodistribuido}$ es el tiempo de ejecución en un sistema no distribuido.
- $T_{distribuido}$ es el tiempo de ejecución en un sistema distribuido.

Un speedup mayor que 1 indica que el algoritmo distribuido es más rápido que su contraparte no distribuida. Sin embargo, es importante señalar que el speedup ideal no siempre es alcanzable debido a la presencia de sobrecarga de comunicación, problemas de sincronización y otros factores relacionados con la distribución de la computación.

El speedup es una medida clave para evaluar la eficiencia de los algoritmos distribuidos, especialmente en entornos como sistemas de procesamiento distribuido, clústeres de computadoras y entornos de cómputo en la nube, donde la paralelización puede conducir a mejoras significativas en el rendimiento.

El calculo correspondiente junto a los cálculos estadísticos del valor promedio y de la desviación estándar se encuentran en el script: "***./Matrix_Multiplication/scripts/StatisticsValues.py***"

B. *Análisis gráfico*

La primera fase de análisis se centra en la observación de los tiempos de ejecución de cada algoritmo en los diversos ambientes de prueba.

En este contexto, se presentan las siguientes representaciones gráficas derivadas de los experimentos realizados. Estas gráficas constituyen una herramienta visual clave para evaluar y comparar de manera efectiva el rendimiento de los algoritmos en los entornos especificados, proporcionando una perspectiva más accesible para interpretar las diferencias y similitudes en los resultados experimentales.

Es preciso señalar que, debido a limitaciones técnicas, no fue factible ejecutar las pruebas con tamaños de matrices superiores a 6000 en el entorno del equipo de prueba con sistema operativo Windows.

No obstante, se procederá a realizar el análisis con los datos correspondientes a tamaños menores, maximizando la utilización de los datos disponibles para proporcionar un entendimiento significativo del rendimiento en ese entorno específico.

1) *Rendimiento vs. Procesadores*

Se inicia el análisis gráfico presentando las representaciones visuales del rendimiento en relación con la cantidad de procesadores para cada uno de los algoritmos.

Se destaca la diversidad de tamaños de matrices que han sido objeto de análisis en estas visualizaciones, buscando resaltar las variaciones y tendencias discernibles en función de dichos parámetros.

Estas gráficas constituyen una herramienta esencial para la interpretación y comparación efectiva del rendimiento de los algoritmos bajo distintas configuraciones, proporcionando una visión detallada y comprensible de su comportamiento.

a) Algoritmo tradicional de multiplicación de matrices

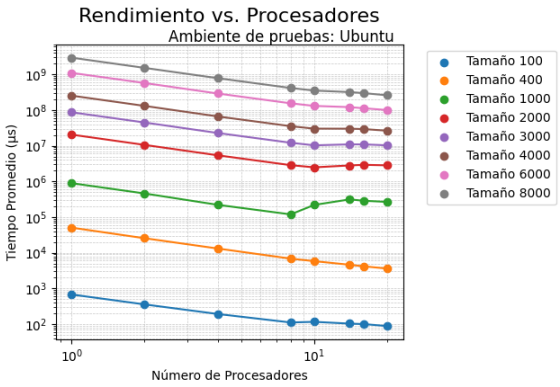


Fig. 12. Gráfica de Rendimiento vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en el ambiente de pruebas Ubuntu.

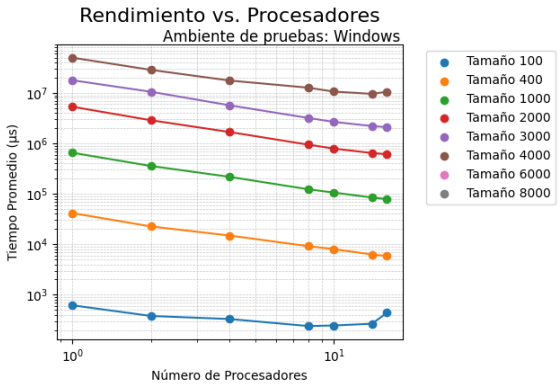


Fig. 13. Gráfica de Rendimiento vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en el ambiente de pruebas Windows.

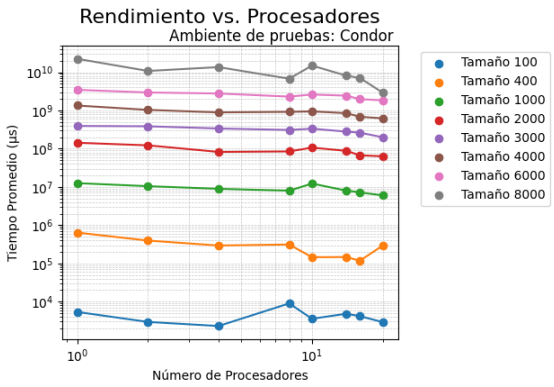


Fig. 14. Gráfica de Rendimiento vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en el ambiente de pruebas Condor.

En las representaciones gráficas correspondientes a los ambientes de prueba de Ubuntu y Windows, se evidencia una reducción significativa de aproximadamente un orden de magnitud al aumentar la cantidad de procesadores de uno a diez. No obstante, este beneficio en términos de tiempo no se observa en el caso del entorno Condor.

En el Clúster Condor, se destaca un comportamiento particular del tiempo promedio de ejecución, que exhibe una tendencia lineal en las diversas ejecuciones, a pesar de presentar un tiempo promedio mayor en comparación con los dos entornos de ejecución restantes.

Sin embargo, para la matriz de mayores dimensiones probada, se observa una reducción significativa del tiempo de ejecución con un incremento en la cantidad de procesadores.

En el contexto de los tres entornos de prueba, se observa que, al incrementar el tamaño de la matriz, también se incrementa el tiempo promedio de procesamiento. Es importante destacar que, a pesar de este aumento progresivo, las ejecuciones con tamaños de matrices en el rango de miles no experimentan variaciones sustanciales en el orden de magnitud del tiempo promedio de ejecución.

Este comportamiento sugiere una relativa estabilidad en el rendimiento de los algoritmos para tamaños de matrices dentro de ese rango específico. Dicha consistencia podría indicar una capacidad razonable de los algoritmos para gestionar conjuntos de datos de tamaño moderado sin incurrir en aumentos significativos en los tiempos de ejecución.

b) Algoritmo filas por filas de multiplicación de matrices

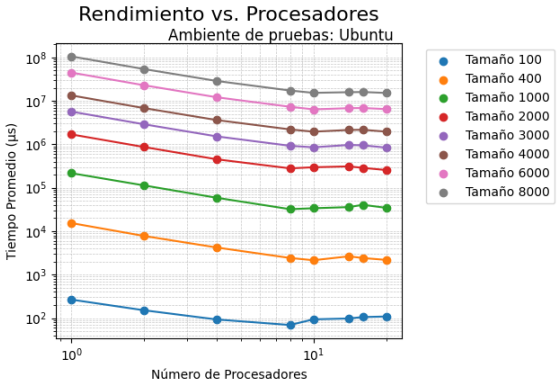


Fig. 15. Gráfica de Rendimiento vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en el ambiente de pruebas Ubuntu.

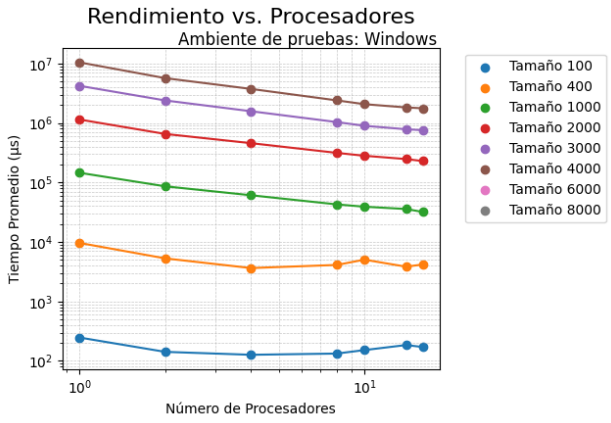


Fig. 16. Gráfica de Rendimiento vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en el ambiente de pruebas Windows.

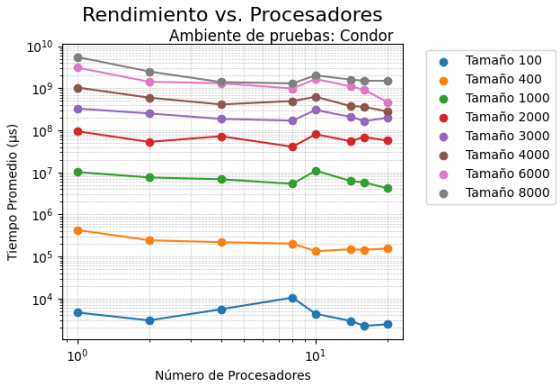


Fig. 17. Gráfica de Rendimiento vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en el ambiente de pruebas Condor.

El algoritmo de multiplicación de filas por filas, al igual que su contraparte tradicional, exhibe una tendencia de reducción en el tiempo promedio de ejecución conforme se incrementa el número de procesadores asignados. Este fenómeno es evidente en los entornos de prueba de Ubuntu y Windows. En el entorno Condor, el tiempo promedio de ejecución sigue un comportamiento lineal, respaldando la observación previa realizada para el algoritmo convencional.

Al analizar las gráficas de los entornos de Ubuntu y Windows, también se evidencia una estabilización del tiempo promedio de ejecución alrededor de la asignación de diez procesadores. Esta constancia en el rendimiento sugiere una optimización alcanzada en términos de paralelización y destaca la influencia significativa de la cantidad de procesadores en el desempeño del algoritmo.

En el contexto general de los tres entornos de prueba, se advierte que, al aumentar el tamaño de la matriz, se observa un consecuente incremento en el tiempo promedio de procesamiento. Sin embargo, es crucial señalar que, a pesar de este aumento gradual, las ejecuciones con tamaños de matrices en el rango de miles no experimentan variaciones sustanciales en el orden de magnitud del tiempo promedio de ejecución.

Este fenómeno subraya la capacidad de los algoritmos para gestionar conjuntos de datos de tamaño moderado con una relativa estabilidad en los tiempos de ejecución, proporcionando una perspectiva valiosa sobre su eficacia en escenarios específicos.

2) Speedup

La serie subsiguiente de gráficas a examinar se centra en la métrica de "speedup". Se enfatiza la variedad de tamaños de matrices que han sido considerados en estas representaciones visuales, con la intención de resaltar las variaciones y tendencias identificables en relación con dichos parámetros.

Estas gráficas proporcionan una perspectiva fundamental para evaluar el rendimiento relativo de los algoritmos en términos de escalabilidad y eficiencia, permitiendo una comprensión más profunda de su capacidad para aprovechar recursos adicionales en función del tamaño de la matriz.

a) Algoritmo tradicional de multiplicación de matrices

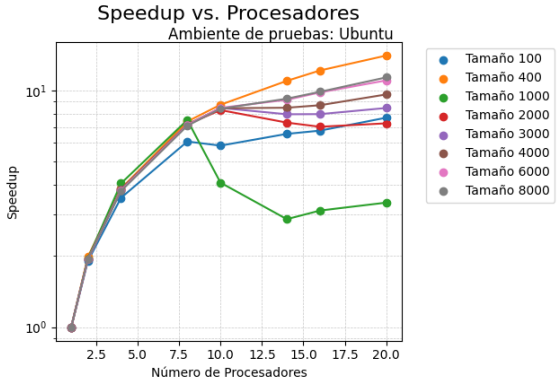


Fig. 18. Gráfica de Speedup vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en el ambiente de pruebas Ubuntu.

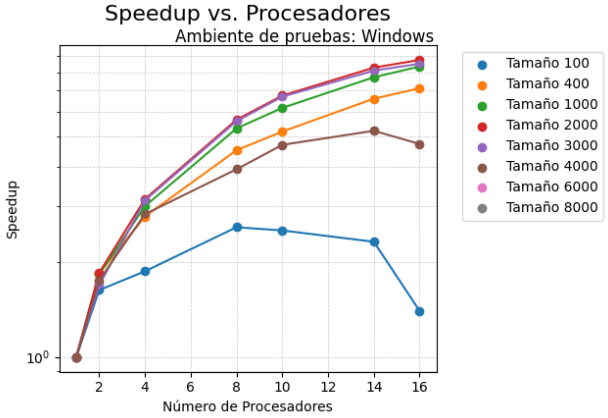


Fig. 19. Gráfica de Speedup vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en el ambiente de pruebas Windows.

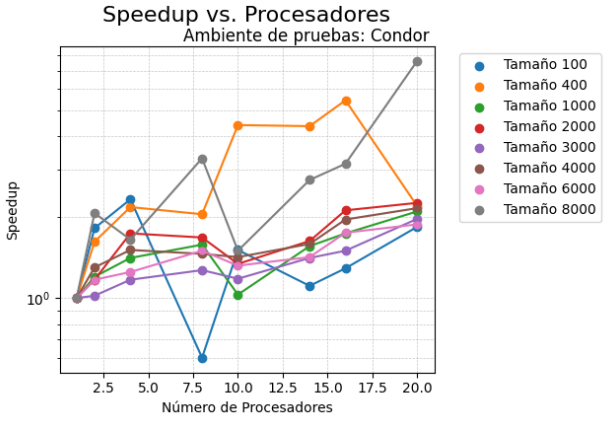


Fig. 20. Gráfica de Speedup vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en el ambiente de pruebas Condor.

Estas gráficas corroboran el patrón de mejora en la ejecución de los algoritmos a medida que se incrementa la cantidad de procesadores, especialmente evidente en los entornos de Ubuntu y Windows.

No obstante, para el entorno del Clúster Condor, el comportamiento de mejora se presenta de manera más irregular

y no exhibe mejoras sustanciales en cuanto a la métrica de "*speedup*".

En el contexto del entorno de Ubuntu, se destaca que la única ejecución que desvía la tendencia general es aquella asociada a matrices de tamaño 1000. A pesar de esta excepción, se observa un aumento positivo en el "*speedup*" en comparación con la ejecución no distribuida del trabajo, indicando una contribución favorable de la paralelización en términos de eficiencia.

b) Algoritmo filas por filas de multiplicación de matrices

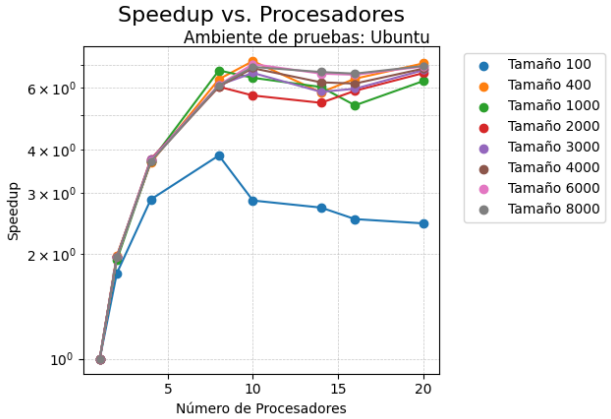


Fig. 21. Gráfica de Speedup vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en el ambiente de pruebas Ubuntu.

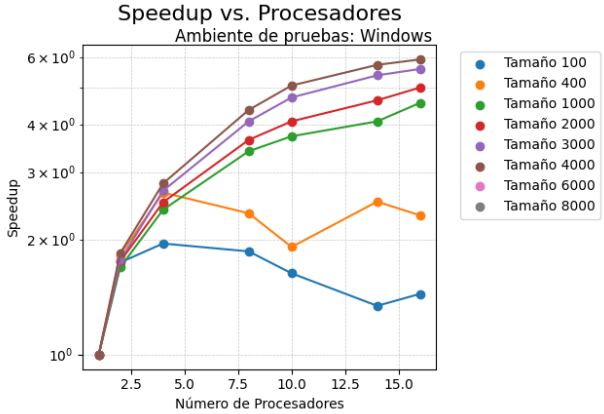


Fig. 22. Gráfica de Speedup vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en el ambiente de pruebas Windows.

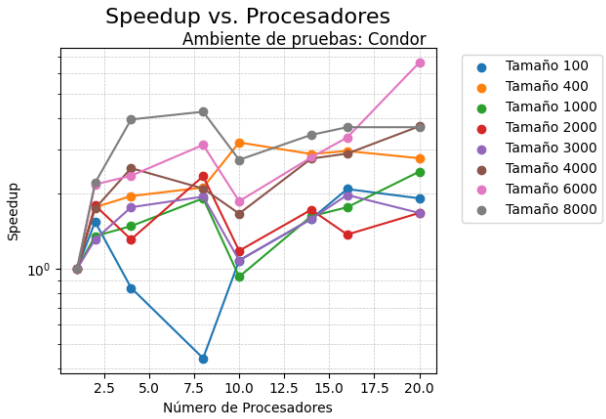


Fig. 23. Gráfica de Speedup vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en el ambiente de pruebas Condor.

En el caso de este algoritmo de multiplicación de matrices por filas por filas, se evidencia un comportamiento análogo al anteriormente descrito. En los entornos de Ubuntu y Windows, se aprecia un incremento significativo en la métrica de "*speedup*" conforme se incrementa la cantidad de procesadores.

En contraste, para el entorno de Condor, se observa un comportamiento más lineal en comparación con los otros dos ambientes. A pesar de que se registra una mejora en la métrica de "speedup", esta no alcanza la misma significancia que en los otros entornos.

Este fenómeno podría atribuirse a las características particulares del algoritmo de multiplicación de matrices por filas por filas, que, a pesar de mostrar mejoras al aumentar la cantidad de procesadores, podría no capitalizar completamente el potencial de paralelización presente en el entorno del Clúster Condor.

C. Comparativa de tiempos de ejecución entre ambientes de prueba

Habiendo adquirido una comprensión exhaustiva y observado el comportamiento de los algoritmos en diversos entornos de ejecución, el siguiente paso consiste en la comparativa entre estos.

En consecuencia, la siguiente sección se dedica a contrastar los tiempos promedio de ejecución de los distintos algoritmos considerando matrices de tamaños variables a multiplicar.

Este análisis comparativo permitirá discernir patrones consistentes y divergencias en el rendimiento de los algoritmos en función de las dimensiones de las matrices, brindando una perspectiva integral sobre su desempeño relativo.

a) Algoritmo tradicional de multiplicación de matrices

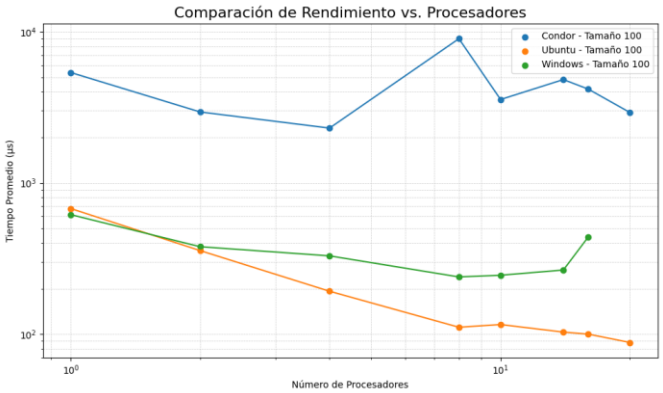


Fig. 24. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en los distintos ambientes de prueba para matrices de tamaño 100.

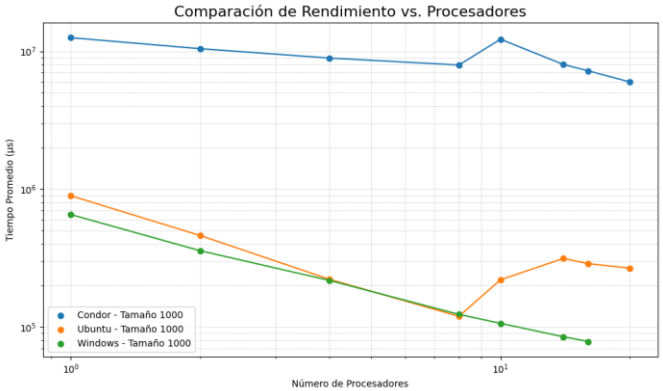


Fig. 25. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en los distintos ambientes de prueba para matrices de tamaño 1000.

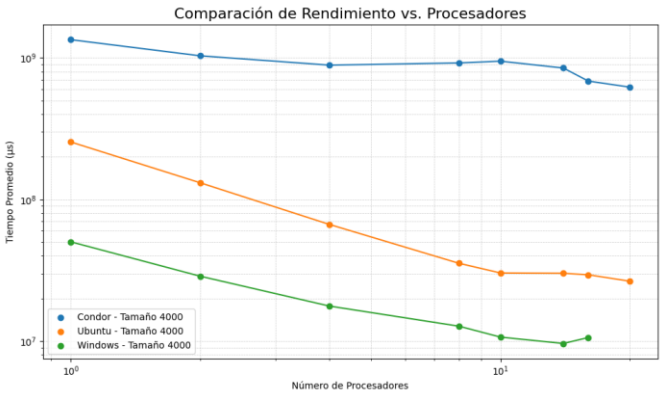


Fig. 26. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para el algoritmo tradicional de multiplicación de matrices en los distintos ambientes de prueba para matrices de tamaño 4000.

Para el algoritmo convencional de multiplicación de matrices, las gráficas anteriores evidencian que, en los casos de tamaños de matrices 100, 1000 y 4000, los entornos de prueba Ubuntu y Windows exhiben tiempos promedio de ejecución inferiores en comparación con el entorno Condor.

Un análisis más detallado de la ejecución en los ambientes Ubuntu y Windows revela un comportamiento similar en términos de tiempos de ejecución.

La disparidad más destacada entre estos dos entornos se manifiesta en el caso del tamaño de matriz 4000. A pesar de la notable diferencia en el tiempo promedio de ejecución, ambos ambientes comparten la característica de reducir el tiempo a medida que se incorporan más procesadores a la ejecución.

Este análisis subraya la importancia de considerar el tamaño específico de la matriz al evaluar el rendimiento relativo de los algoritmos en distintos entornos.

b) Algoritmo filas por filas de multiplicación de matrices

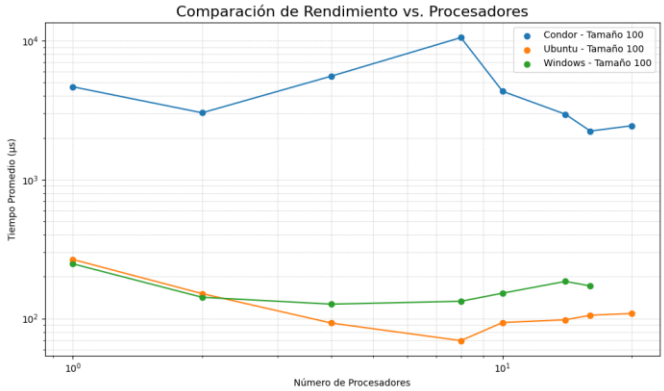


Fig. 27. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en los distintos ambientes de prueba para matrices de tamaño 100.

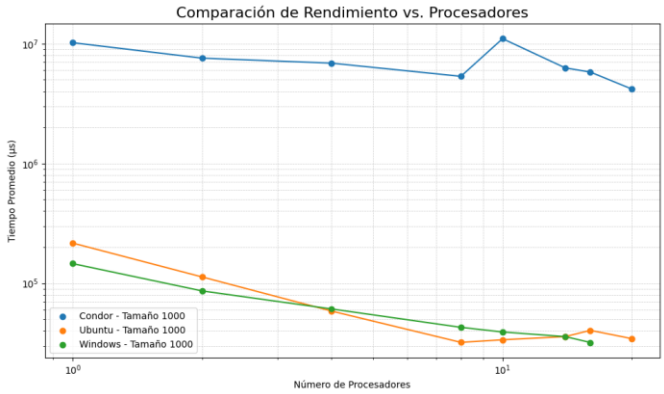


Fig. 28. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en los distintos ambientes de prueba para matrices de tamaño 1000.

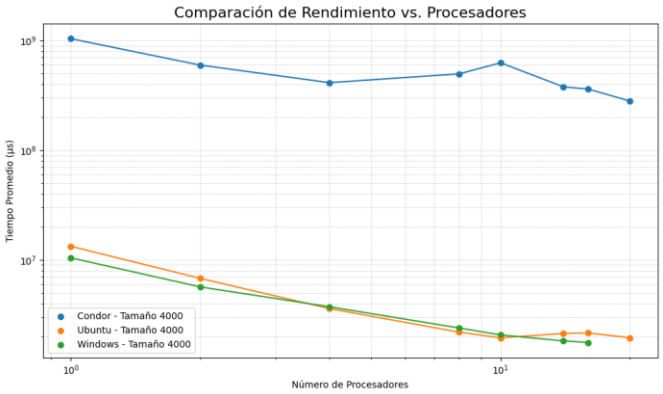


Fig. 29. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para el algoritmo filas por filas de multiplicación de matrices en los distintos ambientes de prueba para matrices de tamaño 4000.

El escenario para el algoritmo de multiplicación de matrices por filas por filas refleja un comportamiento análogo al previamente descrito.

En este caso, los entornos de Windows y Ubuntu exhiben tiempos promedio de ejecución inferiores al entorno Condor. Notablemente, no se observan comportamientos sustancialmente distintos entre los entornos de Windows y Ubuntu para este algoritmo; ambos siguen una tendencia descendente en cuanto al tiempo promedio de ejecución.

Este análisis refuerza la consistencia en el rendimiento relativo de los algoritmos en entornos similares y respalda la relevancia de considerar el contexto específico al interpretar los resultados obtenidos.

D. Comparativa de tiempos de ejecución entre algoritmos

En vista de la similitud en las características entre los ambientes de ejecución en cuanto a la tendencia de los tiempos de ejecución promedio para cada algoritmo, la siguiente etapa consiste en identificar cuál algoritmo presenta los tiempos de ejecución promedio más bajos en cada entorno.

A continuación, se presentan gráficas detalladas que desglosan los tiempos promedio de ejecución de los dos algoritmos analizados, discriminados por ambiente de ejecución.

Estas representaciones visuales facilitarán la evaluación comparativa de los algoritmos en función de sus rendimientos relativos en cada entorno específico.

a) Ambiente Condor

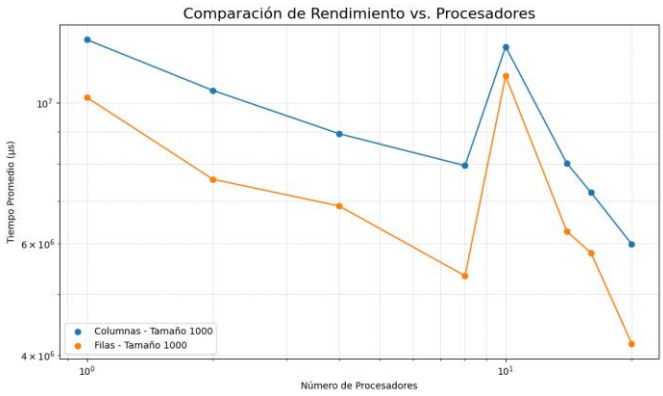


Fig. 30. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Condor para matrices de tamaño 1000.

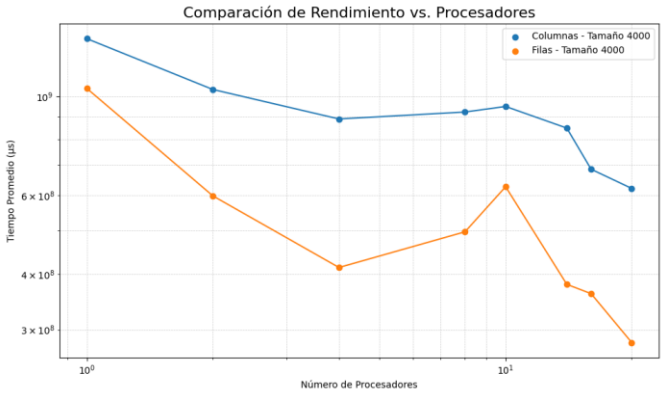


Fig. 31. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Condor para matrices de tamaño 4000.

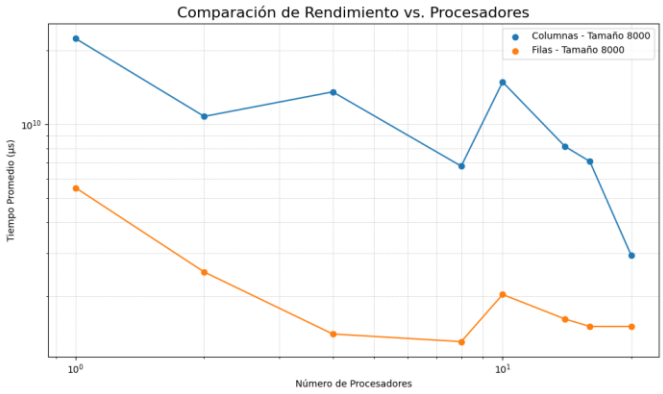


Fig. 32. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Condor para matrices de tamaño 8000.

La ejecución en el entorno de pruebas Condor revela una mejora sustancial en el tiempo promedio de ejecución al emplear el algoritmo de multiplicación de matrices por filas por filas en comparación con el algoritmo tradicional.

Este patrón se mantiene consistente para matrices de diversos tamaños. Aunque en las gráficas presentadas el comportamiento es similar para un tamaño de matriz de 1000, la ventaja del algoritmo de filas por filas se torna más notable a medida que aumenta el tamaño de la matriz.

Adicionalmente, tanto el algoritmo tradicional como el de filas por filas exhiben una disminución en el tiempo promedio de ejecución conforme se incrementa la cantidad de procesadores disponibles para la ejecución del programa.

Este fenómeno sugiere que ambos algoritmos son escalables a medida que se amplía la capacidad de procesamiento. Estas observaciones destacan la eficacia del algoritmo de filas por filas en el entorno Condor, subrayando su capacidad para capitalizar la infraestructura de procesamiento distribuido de manera óptima.

b) Ambiente Ubuntu

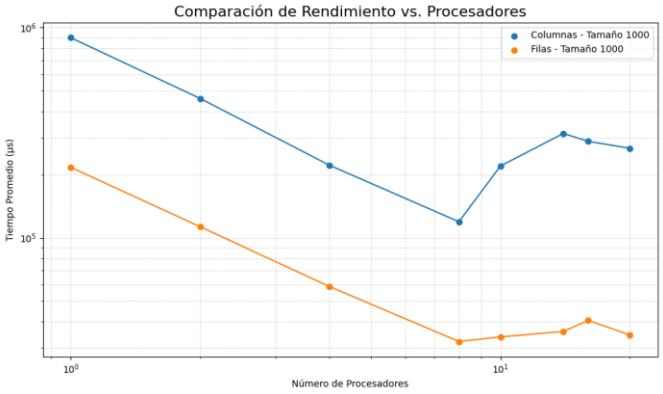


Fig. 33. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Ubuntu para matrices de tamaño 1000.

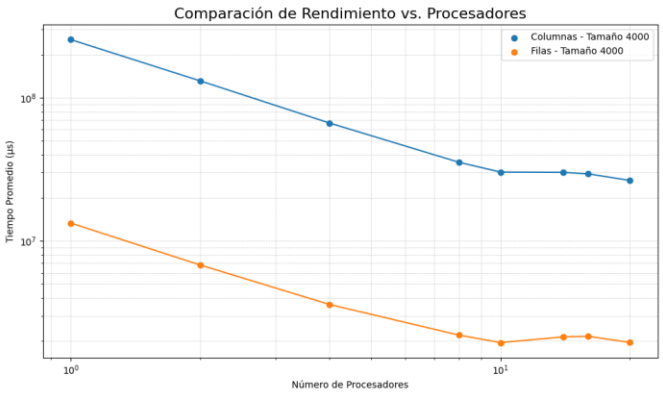


Fig. 34. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Ubuntu para matrices de tamaño 4000.

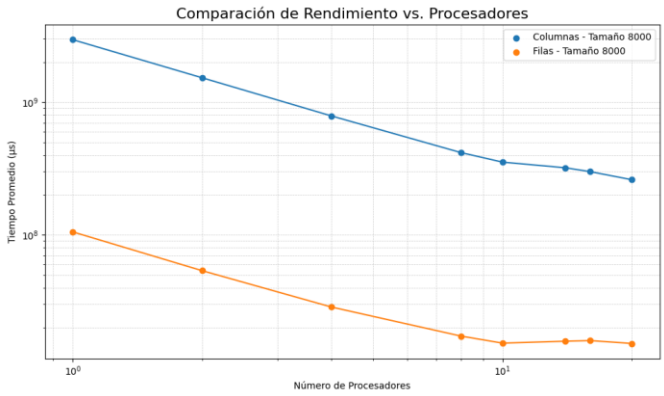


Fig. 35. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Ubuntu para matrices de tamaño 8000.

En el entorno de prueba Ubuntu, se observa un fenómeno similar al caso anterior. El algoritmo de multiplicación de matrices por filas por filas exhibe una mejora sustancial en la duración promedio de la tarea de ejecución.

En este ambiente de pruebas, la mejora es notoria incluso con matrices de tamaño 1000, alcanzando su diferencia más significativa en términos de tiempo de ejecución en el caso de matrices de tamaño 8000.

Esta discrepancia llega a ser aproximadamente de un orden de magnitud a lo largo de todas las pruebas realizadas con diversas cantidades de procesadores. Estos resultados subrayan la eficacia del algoritmo de filas por filas en el entorno Ubuntu, resaltando su capacidad para optimizar la ejecución en sistemas operativos de tipo Linux.

c) Ambiente Windows

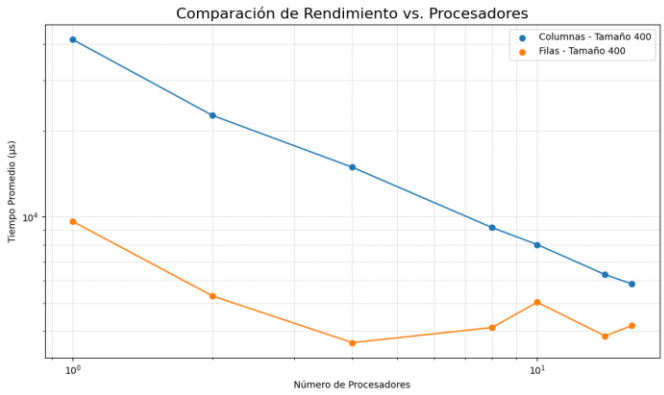


Fig. 36. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Windows para matrices de tamaño 400.

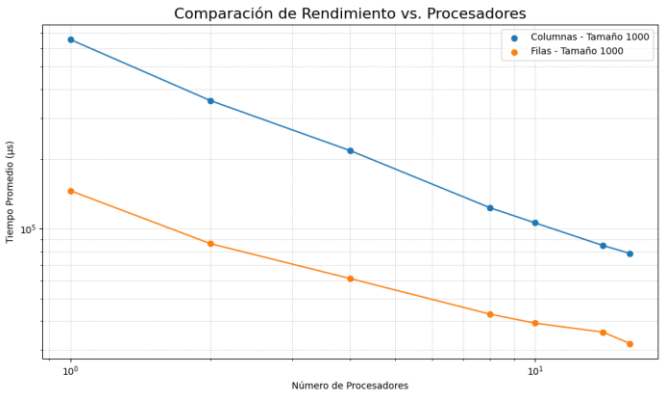


Fig. 37. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Windows para matrices de tamaño 1000.

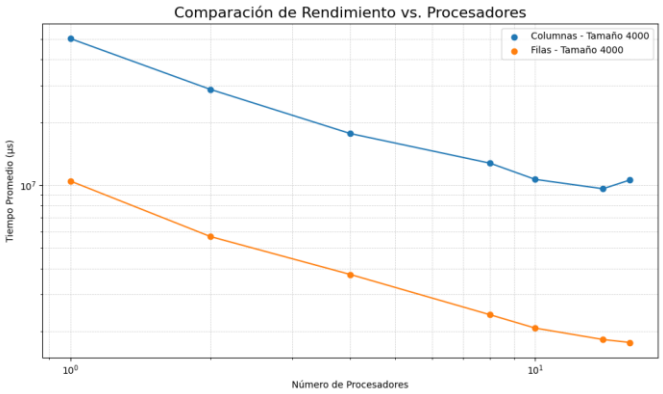


Fig. 38. Gráfica de comparación de Rendimiento vs. Cantidad de procesadores para ambos algoritmos de multiplicación de matrices en el ambiente de prueba Windows para matrices de tamaño 4000.

En el entorno de pruebas Windows, se evidencia un comportamiento similar al observado en el ambiente Ubuntu. No obstante, la diferencia entre los dos algoritmos no es tan pronunciada como en el caso de Ubuntu.

A pesar de ello, tras analizar estos dos entornos de prueba y el anteriormente mencionado, se confirma que el algoritmo de multiplicación de matrices por filas por filas presenta una mejora sustancial en términos de tiempos promedio de ejecución.

Estos hallazgos respaldan la consistencia en el rendimiento del algoritmo de filas por filas, resaltando su capacidad para optimizar la ejecución en ambientes Windows y reafirmando su eficacia en diferentes sistemas operativos.

E. Comparativa de tiempos de ejecución entre algoritmos para el mismo tamaño de matrices y la misma cantidad de procesadores

Con el objetivo de evaluar la magnitud del impacto tanto del tipo de algoritmo como del entorno de pruebas, se presenta un análisis más detallado mediante gráficas de barras.

Estas representaciones visualizan el tiempo promedio de ejecución de cada algoritmo en distintos ambientes, considerando una cantidad fija de uno, diez y dieciséis procesadores y diferentes tamaños de matrices: 100, 1000 y 4000.

Estas gráficas proporcionan una visión más específica y detallada de las diferencias en el rendimiento de los algoritmos en contextos específicos.

1) Cantidad fija de un procesador y variando el tamaño de la matriz.

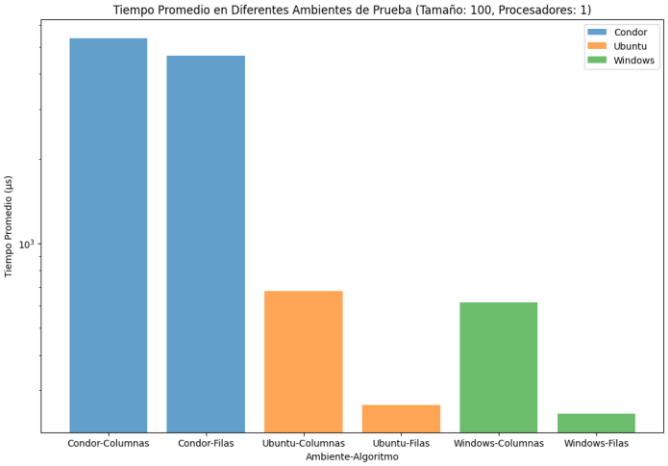


Fig. 39. Gráfica de comparación de Tiempos promedio de ejecución de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 100 usando un solo procesador.

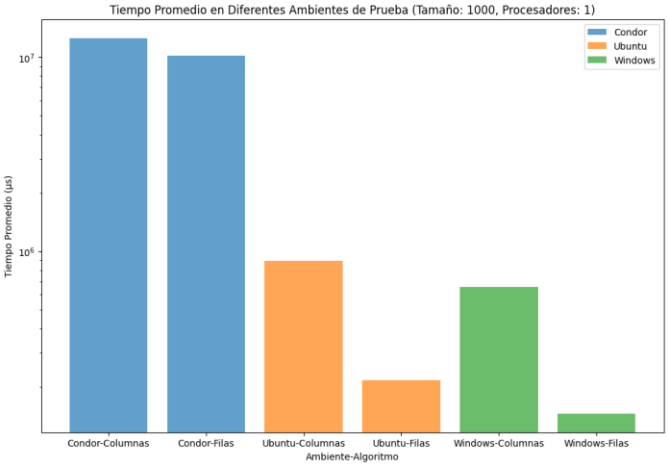


Fig. 40. Gráfica de comparación de Tiempos promedio de ejecución de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 1000 usando un solo procesador.

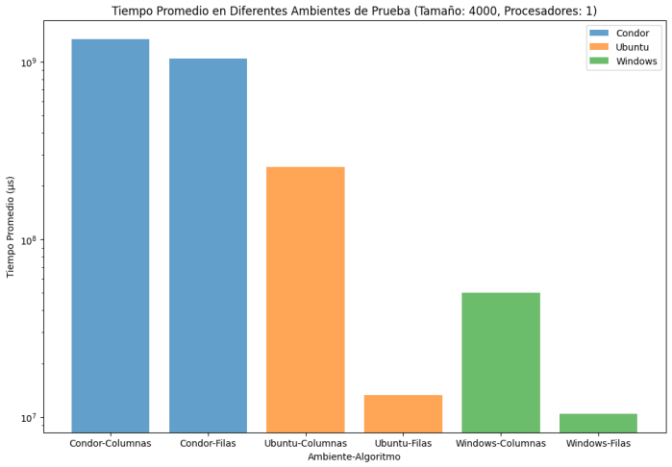


Fig. 41. Gráfica de comparación de Tiempos promedio de ejecución de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 4000 usando un solo procesador.

Las gráficas revelan que el entorno de pruebas Condor exhibe el mayor tiempo promedio de ejecución para ambos algoritmos, con una brecha significativa en comparación con los otros ambientes. En contraste, los entornos Ubuntu y Windows muestran tiempos promedio de ejecución similares, aunque el ambiente Windows registra tiempos ligeramente inferiores.

Al analizar los tipos de algoritmos, se destaca claramente la mejora en el tiempo promedio de ejecución del algoritmo de filas por filas respecto al algoritmo tradicional en todos los entornos de prueba.

Esta mejora es particularmente notable en los entornos Ubuntu y Windows, donde el tiempo promedio de ejecución disminuye casi en un orden de magnitud en matrices de tamaños 100, 1000 y 4000.

Estos resultados subrayan la eficacia del algoritmo de filas por filas en la mejora del rendimiento, especialmente en entornos de ejecución basados en sistemas operativos Linux y Windows.

2) Variando la cantidad de procesadores y dejando fijo el tamaño de la matriz

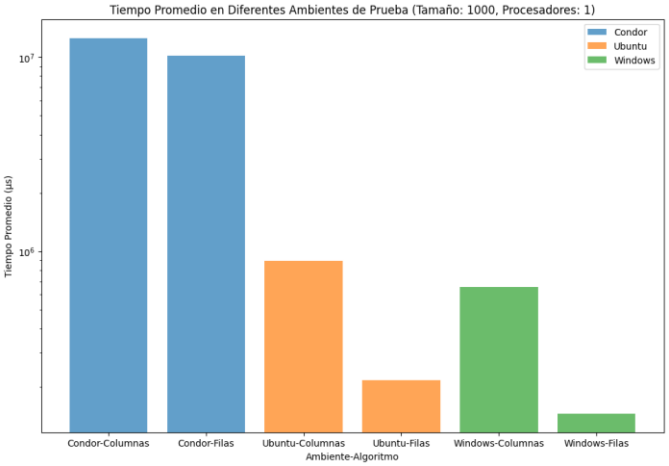


Fig. 42. Gráfica de comparación de Tiempos promedio de ejecución de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 1000 usando un solo procesador.

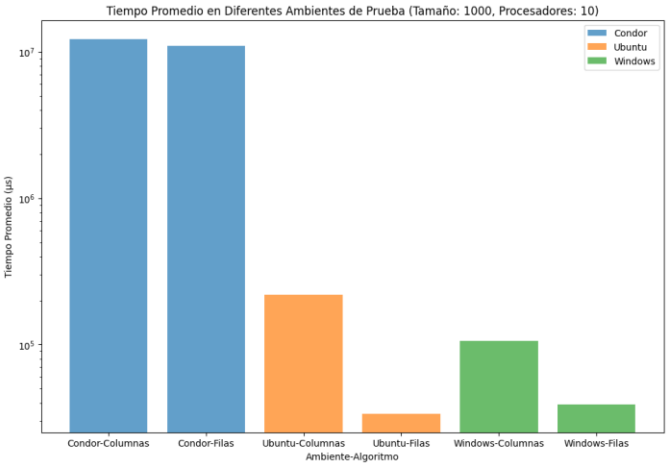


Fig. 43. Gráfica de comparación de Tiempos promedio de ejecución de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 1000 usando 10 procesadores.

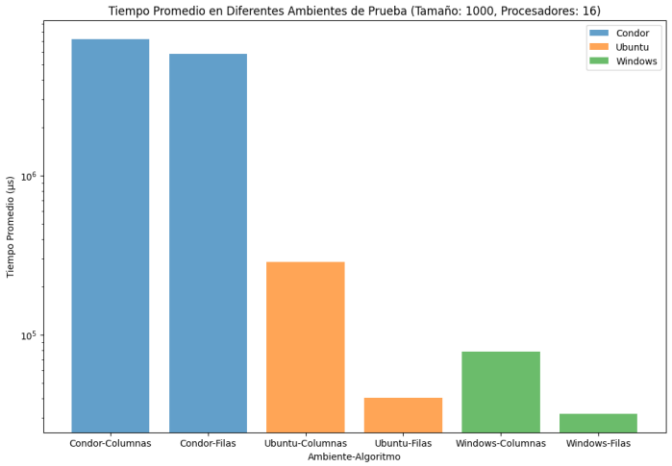


Fig. 44. Gráfica de comparación de Tiempos promedio de ejecución de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 1000 usando 16 procesadores.

Al mantener constante el tamaño de la matriz en 1000 y variar la cantidad de procesadores, persiste la tendencia de que el algoritmo de filas por filas exhibe un tiempo promedio de ejecución inferior al algoritmo tradicional.

Esta mejora continúa siendo significativa en los entornos de prueba Ubuntu y Windows, aunque no se presenta de manera tan pronunciada en el entorno Condor.

Estos resultados refuerzan la consistencia en la mejora del rendimiento del algoritmo de filas por filas en entornos específicos y sugieren que esta ventaja se mantiene incluso con cambios en la cantidad de procesadores, al menos para matrices de tamaño 1000.

F. Comparativa del speedup en los distintos ambientes de prueba

La evaluación del impacto de la adición de procesadores en el conjunto de soluciones se aborda a través del análisis del "speedup", una métrica crucial para comprender la eficiencia del rendimiento en paralelo.

En las siguientes gráficas, se presenta un análisis detallado de esta métrica en los tres ambientes de prueba, abarcando los tres algoritmos estudiados.

Estas visualizaciones permiten discernir cómo el incremento de procesadores afecta la velocidad de ejecución y si los algoritmos demuestran un comportamiento escalable en diferentes entornos.

Este análisis, cabe aclarar se centrará sobre los entornos de prueba Ubuntu y Windows ya que estos son los que presentan mayor "*speedup*" según lo analizado en gráficas anteriores.

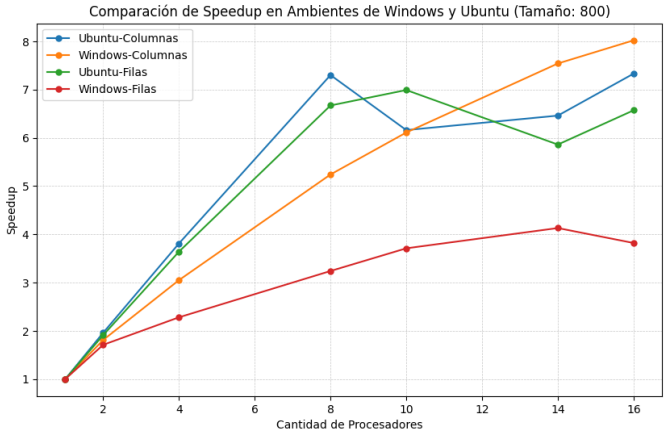


Fig. 45. Gráfica de comparación de speedup vs. Cantidad de procesadores de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 800.

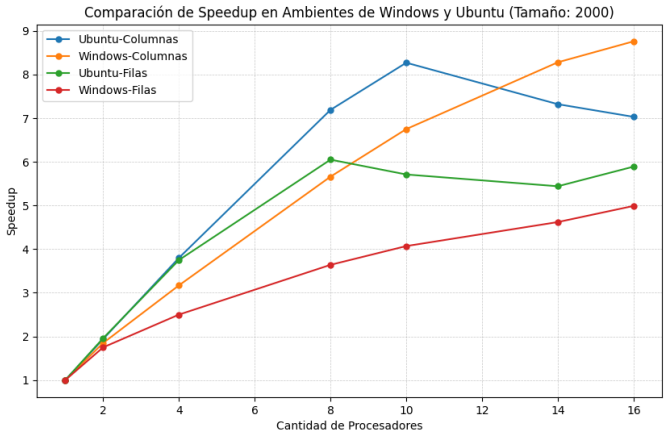


Fig. 46. Gráfica de comparación de speedup vs. Cantidad de procesadores de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 2000.

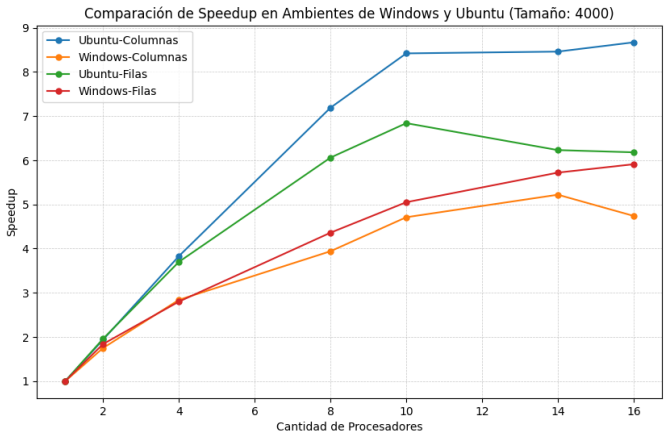


Fig. 47. Gráfica de comparación de speedup vs. Cantidad de procesadores de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 4000.

Las gráficas revelan que tanto el algoritmo tradicional como el de filas por filas, en los entornos de prueba de Ubuntu y

Windows, experimentan un "speedup" positivo a medida que se incrementa la cantidad de procesadores. Este hallazgo sugiere que ambos algoritmos son paralelizables y pueden aprovechar eficientemente múltiples procesadores.

Al examinar matrices de tamaño reducido, inferiores a 2000, se observa que el algoritmo tradicional de multiplicación de matrices es más beneficiado por la adición de procesadores. Estas visualizaciones confirman que la ganancia, al agregar procesadores a ambos algoritmos, es aproximadamente de 5 con 20 procesadores para cada uno.

Además, a partir de 10 procesadores, se infiere que la mejora adicional no es significativa. Estos resultados proporcionan una perspectiva valiosa sobre la escalabilidad de los algoritmos en función de la cantidad de procesadores utilizados.

Ahora bien, no es que en el ambiente de pruebas Condor no se encuentre una ganancia a medida que se adicionan procesadores al proceso de solución, sino que el comportamiento de agregar mas procesadores es un poco más errático.

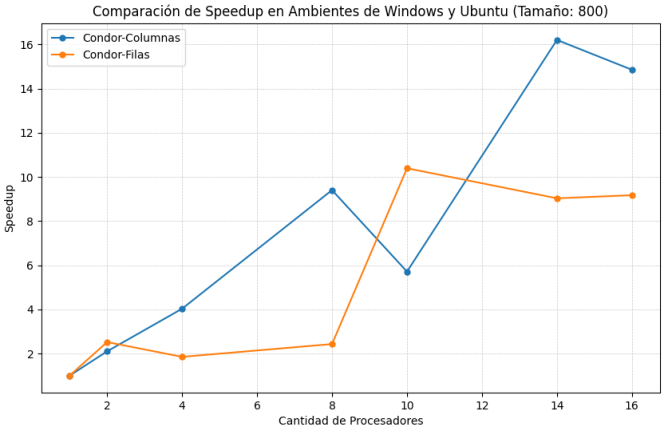


Fig. 48. Gráfica de comparación de speedup vs. Cantidad de procesadores de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 800 para el ambiente Condor.

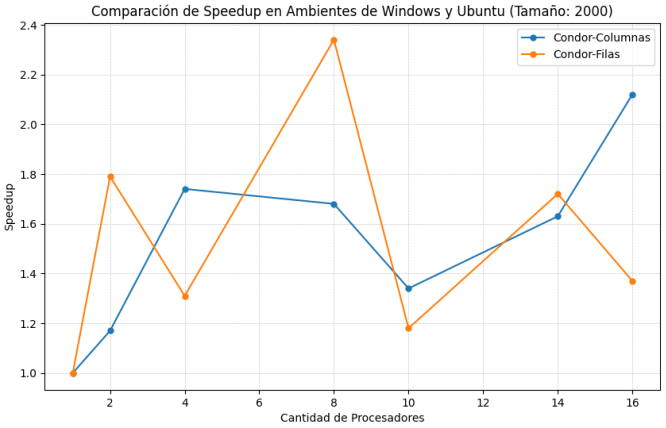


Fig. 49. Gráfica de comparación de speedup vs. Cantidad de procesadores de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 2000 para el ambiente Condor.

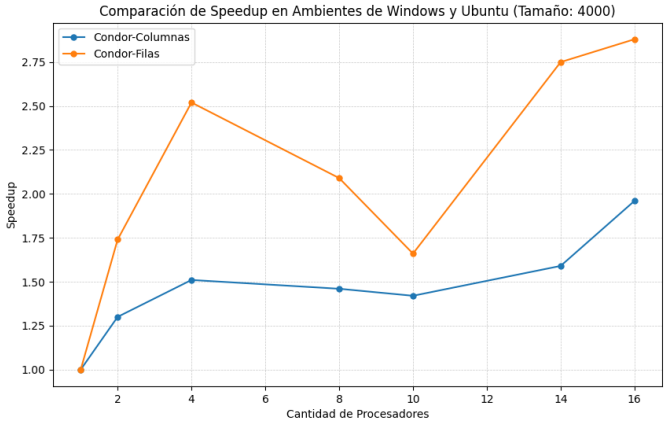


Fig. 50. Gráfica de comparación de speedup vs. Cantidad de procesadores de ambos algoritmos de multiplicación de matrices en los distintos ambientes de pruebas para matrices de tamaño 4000 para el ambiente Condor.

En estos ejemplos de "speedup", se observa que la métrica no sigue una línea de tendencia incremental, a diferencia de lo observado en los ambientes de prueba de Ubuntu y Windows.

Además, la ganancia de ambos algoritmos para matrices pequeñas, como se muestra en la gráfica de tamaño 800, es del orden de 10 para 20 procesadores. Sin embargo, para matrices más grandes, la ganancia no supera el valor de 5.

Este contraste destaca la influencia del tamaño de la matriz en la mejora de rendimiento al aumentar la cantidad de procesadores, mostrando resultados más modestos en comparación con los ambientes de prueba anteriores.

VI. Conclusiones

La multiplicación de matrices se revela como un problema computacional esencial, cuyo abordaje ofrece valiosas lecciones en el ámbito de la computación distribuida.

Dentro de la diversidad de enfoques existentes para resolver este problema, este proyecto se centró en dos estrategias particulares: el método tradicional y el algoritmo de multiplicación por filas.

Este análisis, aunque específico en sus elecciones, brinda una perspectiva inicial sobre la eficacia de estos algoritmos en diferentes entornos y abre la puerta a futuras exploraciones.

El algoritmo de multiplicación por filas destaca como una muestra de cómo la innovación en las estrategias algorítmicas puede traducirse en mejoras sustanciales en los tiempos de ejecución.

No obstante, es fundamental reconocer que este proyecto aborda solo una pequeña fracción del vasto panorama de enfoques existentes para la multiplicación de matrices.

La investigación en este campo continúa avanzando, con numerosas técnicas aún por explorar, lo que resalta la riqueza y complejidad de este problema fundamental.

Cada algoritmo seleccionado representa un punto en el continuo espectro de soluciones posibles. En la realidad, la elección del algoritmo más adecuado dependerá de múltiples

factores, como la naturaleza de los datos, la infraestructura del sistema y los objetivos específicos de rendimiento.

El presente proyecto no solo proporciona una evaluación comparativa de los algoritmos elegidos, sino que también destaca la importancia de la adaptabilidad y la elección contextualizada de estrategias algorítmicas.

En relación con los ambientes de prueba, los resultados revelan que tanto Ubuntu como Windows exhiben un rendimiento sobresaliente en términos de tiempo medio de ejecución y "*speedup*".

Esta observación respalda la idea de que estos algoritmos están particularmente optimizados para funcionar eficientemente en entornos multiprocesador no distribuidos.

Por otro lado, el análisis del ambiente Condor revela una dinámica más compleja, donde la mejora al aumentar la cantidad de procesadores puede no ser tan pronunciada como en entornos multiprocesador. Este hallazgo invita a profundizar en la comprensión de cómo se distribuye y gestiona el trabajo en este entorno específico.

En conclusión, este proyecto proporciona una visión detallada y focalizada de los algoritmos de multiplicación de matrices en diferentes contextos. No obstante, es solo un paso inicial en la exploración de un dominio vasto y diverso.

La computación distribuida continuará desempeñando un papel crucial en la resolución eficiente de problemas computacionales fundamentales, y este proyecto contribuye a la comprensión de cómo distintas estrategias se desempeñan en este contexto dinámico y desafiante.

VII. Trabajo Futuro

El camino hacia investigaciones más profundas y completas en el ámbito de la multiplicación de matrices en entornos distribuidos y multiprocesador ofrece diversas oportunidades para ampliar el conocimiento y mejorar las estrategias existentes. Algunas áreas clave que podrían explorarse en futuros proyectos son:

1. **Evaluación Exhaustiva de Algoritmos**: Considerar una variedad más amplia de algoritmos para la multiplicación de matrices, explorando no solo su eficiencia sino también su capacidad de adaptación a distintos tipos y tamaños de matrices.
2. **Escalabilidad y Tolerancia a Fallos**: Abordar cuestiones de escalabilidad y tolerancia a fallos en entornos distribuidos, analizando cómo los algoritmos se comportan frente a sistemas que enfrentan interrupciones o fallas.
3. **Optimización para Ambiente Condor**: Investigar estrategias específicas para optimizar el rendimiento de los algoritmos en el entorno Condor, considerando la posibilidad de incorporar bibliotecas de computación distribuida como MPI para mejorar la eficiencia.
4. **Exploración de Nuevos Algoritmos**: Considerar algoritmos adicionales, como el algoritmo de Strassen, para analizar sus ventajas y desventajas en comparación con los algoritmos tradicionales y de filas por filas.
5. **Inclusión de Plataformas en la Nube**: Ampliar la evaluación a entornos de nube y sistemas distribuidos heterogéneos para comprender cómo los algoritmos se desempeñan en contextos más diversos.
6. **Impacto de la Memoria RAM**: Incrementar el tamaño de las matrices para evaluar el impacto en el rendimiento, teniendo en cuenta la capacidad de la memoria RAM de los sistemas involucrados en el procesamiento.
7. **Comparativa con Librerías Estándar**: Realizar comparativas detalladas con bibliotecas estándar de multiplicación de matrices, como BLAS o cuBLAS, para entender el rendimiento relativo y las posibles áreas de mejora.
8. **Incorporación de Dispositivos Especializados**: Analizar el rendimiento de los algoritmos al ser ejecutados en dispositivos especializados, como unidades de procesamiento gráfico (GPU) o aceleradores, para explorar posibles ganancias de eficiencia.

Estos enfoques ofrecen una visión más completa y detallada del rendimiento de los algoritmos de multiplicación de matrices en diversos contextos, proporcionando así contribuciones significativas al campo de la computación distribuida y la optimización algorítmica.

VIII. Repositorio de Código

La totalidad del código, incluyendo los scripts y los resultados obtenidos, se halla archivada en un repositorio de acceso público, disponible en la siguiente dirección: https://github.com/Akamiz96/Matrix_Multiplication.

Referencias

[1] X. Liao, S. Li, W. Yu, and Y. Lu, "Parallel matrix multiplication algorithms in Supercomputing," 2021 IEEE 6th International Conference on Intelligent Computing and Signal Processing (ICSP 2021). Accessed: Nov. 01, 2023. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9409013

[2] Q.-L. Kao and C.-R. Lee, "Design Fast Matrix Algorithms on High-Performance Cloud Platforms," 2012 IEEE 4th International Conference on Cloud Computing Technology and Science, 2012, Accessed: Nov. 01, 2023. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6427556

[3] A. J. Stothers, "On the Complexity of Matrix Multiplication," 2010.

[4] R. Enrique, A. Lira, A. C. May, R. Armando González García, V. Alberto Gómez Pérez, and C. De Jesús, "MulCoMa un algoritmo concurrente para multiplicar matrices," Journal of Basic Sciences, vol. 3, no. 7, pp. 1–11, 2017, [Online]. Available: http://revistas.ujat.mx/index.php/jobs

[5] A. : Daniel and G. Stelzner, "Análisis de tecnologías HPC en entornos de computación heterogéneos," 2013.

[6] [T. Jensen, "Tutorial sobre apuntadores y arreglos en C," Feb. 2000, Accessed: Nov. 01, 2023. [Online]. Available: https://www.cimat.mx/~alram/cpa/pointersC.pdf

[7] A. Qawasmeh, A. M. Malik, and B. M. Chapman, "OpenMP task scheduling analysis via OpenMP runtime API and tool visualization," in Proceedings - IEEE 28th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2014, IEEE Computer Society, Nov. 2014, pp. 1049–1058. doi: 10.1109/IPDPSW.2014.116.