Análisis de Rendimiento del Nodo de Cómputo con OpenMP y MPI

Nestor Aparicio[1*], Manuel Beltran[2*], Gisell Cristiano[2*] and Daniel Prada[1*]

[1]Ingeniería de Sistemas y Computación, Facultad de Ingeniería, Pontificia Universidad Javeriana, 110231, Bogotá, Colombia.
[2]Maestría en Inteligencia Artificial, Facultad de Ingeniería, Pontificia Universidad Javeriana, 110231, Bogotá, Colombia.

*Corresponding author(s). E-mail(s): nestorj_aparicioh@javeriana.edu.co - https://github.com/nextore28; beltranom@javeriana.edu.co - https://github.com/IaManBel; gn_cristiano@javeriana.edu.co - https://github.com/GisellNatalia; elkind.prada@javeriana.edu.co - https://github.com/Elkin77;

Abstract

Los lenguajes de programación paralelos representan un tema común en la evolución de los sistemas informáticos de alto rendimiento (HPC). Así mismo los procesadores multinúcleo ofrecen un potencial creciente de paralelismo, pero plantean un desafío de desarrollo de programas para lograr un alto rendimiento en las aplicaciones. El objetivo de este estudio es analizar y comparar el rendimiento de cuatro algoritmos de multiplicación de matrices estándar (fila por columna) con diferentes técnicas de optimización (MM1c, MM1f, MM1fu, MM2f) implementados con dos modelos de programación paralela (OpenMPI y OpenMP) en un nodo de cómputo con procesadores multinúcleo. Para ello, se evaluó el tiempo de ejecución de la multiplicación de matrices para diferentes tamaños de matrices y número de hilos en una arquitectura de memoria compartida. Los resultados muestran que en la implementación OpenMP el código MM2f fue el más eficiente en términos de rendimiento, mientras que en la implementación MPI, los códigos MM2f, MM1f y MM1fu lograron obtener los tiempos de ejecución más cortos, lo que significa que son los códigos más eficientes en términos de rendimiento. Además, es importante destacar que el algoritmo MM2f presenta una complejidad reducida de $\boldsymbol{O(N^{3/2})}$ en comparación con los otros algoritmos, lo cual contribuye a su mayor eficiencia en términos de rendimiento.

Keywords: programación paralela, algoritmos de multiplicación de matrices, OpenMPI, OpenMP, HPC.

1 Introdución

En los últimos años, la tendencia ha sido optimizar las soluciones informáticas paralelas y distribuidas utilizando diseños de hardware escalables y lenguajes de programación paralelos escalables [1]. Además, el uso de OpenMP [2] ha permitido aprovechar los procesadores multinúcleo [3, 4] y arquitecturas multinúcleo [5] para el procesamiento paralelo mediante la utilización de subprocesos. Cada núcleo de una máquina es una unidad que lee y ejecuta una instrucción de programa de longitud fija llamada Word [6]. Suele ser de 8, 16, 32, 64 o trozos de bits de longitud variable. Las instrucciones de un programa indican la operación a realizar, puede ser la lectura de los datos del teclado, la visualización de los resultados en el dispositivo, la obtención de datos de un archivo o la escritura de la salida en un archivo. Aunque una máquina de un solo núcleo ejecuta una instrucción a la vez, el procesamiento paralelo en arquitecturas multinúcleo se ha convertido en una estrategia importante para mejorar el rendimiento de las aplicaciones informáticas [7].

Las CPU en las computadoras paralelas actuales son más rápidas que las de 1985, es importante destacar que ha habido avances significativos en los entornos de programación paralela en los últimos años. En la actualidad, existen dos estándares importantes que se utilizan ampliamente en la programación paralela: MPI y OpenMP [8]. OpenMP es una herramienta de programación que permite la técnica de ejecución fork-and-join, en la que un programa comienza a ejecutarse como un único proceso o subproceso. Durante la ejecución, el subproceso se ejecuta secuencialmente hasta que se detecta una directiva de paralelización para una región paralela. En este punto, el subproceso crea un grupo de subprocesos y se convierte en el subproceso maestro del nuevo grupo. Todos los hilos ejecutan el programa hasta el final de este segmento paralelo. Además de la paralelización dentro de un nodo, se puede lograr otra paralelización del programa a través de la programación de la interfaz de paso de mensajes (MPI, por sus siglas en ingles) [9], que se puede emplear dentro y entre varios nodos. MPI es un estándar ampliamente aceptado para escribir programas de paso de mensajes y proporciona al usuario un modelo de programación donde los procesos se comunican entre sí llamando a las rutinas de la biblioteca para enviar y recibir mensajes. Esta combinación de técnicas de programación ofrece una gran capacidad de paralelización y escalabilidad en sistemas de alto performance.

En resumen, tanto MPI como OpenMP son herramientas importantes para la programación paralela. MPI es una biblioteca que permite la comunicación y coordinación entre procesadores que no comparten memoria, lo que permite la realización de cálculos en paralelo. Por otro lado, OpenMP es un conjunto de directivas de compilación que ayudan a generar código multiproceso que aprovecha múltiples procesadores dentro de un multiprocesador de memoria compartida. Empleando

ambos, MPI y OpenMP, es posible programar un cluster de multiprocesadores para realizar tareas de manera más rápida y eficiente que empleando un solo procesador.

El objetivo de este documento es analizar, implementar y comparar cuatro programas en OpenMP y MPI para multiplicar matrices cuadradas. Esto responde a la necesidad de evaluar y comprender cómo diferentes técnicas de programación paralela afectan el rendimiento de las aplicaciones en este contexto. Cada programa se basa en una distribución diferente de los elementos de la matriz. Por lo tanto, cada uno de los cuatro programas es significativamente diferente de los otros tres.

2 Background

La Unidad Central de Procesamiento (CPU) es el circuito electrónico que ejecuta las instrucciones que componen un programa de computadora. Realiza operaciones aritméticas y lógicas básicas instruidas por el programa. Un core es el cerebro de la CPU que recibe instrucciones y realiza operaciones. Puede tener multi-cores, por ejemplo, puede tener 2 cores, 4 cores, 8 cores, etc. [10].

La arquitectura de memoria compartida es adecuada para la computación paralela en varios procesadores que comparten una sola memoria [11]. Para los sistemas de memoria compartida que tienen múltiples procesadores de varios núcleos, todos los procesadores pueden conectarse a la memoria principal directamente o cada procesador puede conectarse a un bloque de memoria mediante interconexión. Aunque cada procesador está conectado a un bloque de memoria, los procesadores pueden tener acceso a otros bloques de memoria a través del hardware integrado en el procesador. La arquitectura de memoria compartida para computación paralela brinda un mejor rendimiento en comparación con la computación secuencial, pero no en comparación con la computación distribuida cuando la carga de trabajo es demasiado grande para caber en una sola máquina.

Hasta el momento se ha desarrollado una amplia gama de modelos de programación paralela basados en memoria compartida. Se pueden clasificar principalmente en tres tipos, como se describe a continuación [12].

1. *Modelos de subprocesos:* estos modelos se basan en la biblioteca de subprocesos que proporciona rutinas de bajo nivel para paralelizar la aplicación. Estos modelos utilizan bloqueos de exclusión mutua y variables condicionales para establecer comunicaciones y sincronización entre subprocesos. Los modelos de subprocesos son adecuados para aplicaciones basadas en la multiplicidad de datos y proporcionan una flexibilidad muy alta al programador.
2. *Modelos basados en directivas:* estos modelos utilizan las directivas del compilador de alto nivel para paralelizar las aplicaciones. Estos modelos son una extensión de los modelos basados en hilos. Los modelos basados en directivas se encargan de las características de bajo nivel, como la partición, la gestión de trabajadores, la sincronización y la comunicación entre los subprocesos. La principal ventaja de

los modelos de directivas es que es fácil escribir aplicaciones paralelas y el programador no necesita considerar problemas como carreras de datos, intercambio falso, interbloqueos.

3. *Modelos de asignación de tareas:* los modelos se basan en el concepto de especificar tareas en lugar de subprocesos como lo hacen otros modelos.Esto se debe a que las tareas son de corta duración y más ligeras que los subprocesos. Una diferencia entre las tareas y los subprocesos es que las tareas siempre se implementan en modo usuario.

Por otro lado, el modelo de memoria distribuida es conocido como un conjunto de múltiples computadoras con múltiples pares de procesadores y memorias conectados entre sí a través de una red de interconexión. Básicamente, describe un gran conjunto de computadoras en red, cada una con su propia memoria privada. Se utiliza para aplicaciones que son demasiado grandes para caber en una sola máquina.

Los modelos empleados en este paper son:

- OpenMP como modelo de memoria compartida basado en directivas:

  El paso de mensajes abierto o la especificación abierta para la multiprocesamiento es una interfaz de programa de aplicación que define un conjunto de directivas de programa, rutinas de biblioteca en tiempo de ejecución y variables de entorno que se utilizan para expresar explícitamente la paralelismo de memoria compartida con subprocesos múltiples directos [13]. Se puede especificar en C/C++/FORTRAN. OpenMP se sitúa en un alto nivel de abstracción que facilita el desarrollo de aplicaciones paralelas desde la perspectiva del desarrollador. OpenMP oculta e implementa por sí mismo detalles como la partición de carga de trabajo, la gestión de trabajadores, la comunicación y la sincronización. Los desarrolladores solo necesitan especificar las directivas para paralelizar la aplicación.

- MPI como modelo de memoria distribuida y compartida:

  El Estándar de Interfaz de Paso de Mensajes (MPI) es un estándar de biblioteca de paso de mensajes diseñado para funcionar en una amplia variedad de computadoras paralelas [14]. Se han definido especificaciones de interfaz para programas en C/C++ y Fortran. Mientras que otros modelos trabajan sólo en sistemas de multiprocesamiento simétrico (SMP), MPI funciona en sistemas de memoria compartida y distribuida. Es portable, no es necesario modificar el código fuente al portar la aplicación a una plataforma diferente que admita (y sea compatible con) el estándar MPI. MPI utiliza objetos llamados comunicadores y grupos para definir qué colección de procesos pueden comunicarse entre sí. La mayoría de las rutinas de MPI requieren especificar un comunicador como argumento. Dentro de un comunicador, cada proceso tiene su propio identificador único de entero asignado por el sistema cuando el proceso se inicializa, llamado rango. Los rangos son utilizados por el programador para especificar la fuente y el destino de los mensajes.

3 Configuración de experimentos

En este documento se ejecuto el algoritmo de multiplicación de matrices, se realizó un número significativo de experimentos con varios tamaños de matrices, a saber, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1200, 1400, 1600, 1800, 2000, 2400, 2800, 3000. También con distintas cantidades de threads, a saber, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. Para obtener los resultados finales, se calcula la media del tiempo de ejecución en segundos para cada combinación de tamaño de matriz y número de hilos.

Se emplean cuatro programas de multiplicación de matrices estándar implementados con dos modelos de programación paralela (OpenMPI y OpenMP). Los programas difieren en la forma como se almacenan las matrices en memoria, lo que afecta el orden en que se acceden a las entradas [14]:

1. *MM1c:*

   El siguiente código es un programa en C que realiza la multiplicación de matrices empleando OpenMP. La multiplicación de matrices se realiza utilizando un enfoque estándar de bucles anidados triples, donde el bucle interno calcula el producto punto de una fila de la primera matriz con una columna de la segunda matriz. El elemento resultante de la matriz de producto se almacena en una tercera matriz.

   La función principal toma el tamaño de la matriz como argumento e inicializa las matrices utilizando la función Matrix_Init_col. Las matrices se almacenan en un bloque continuo de memoria de tamaño DATA_SZ, que se asigna estáticamente al principio del archivo utilizando la matriz _CHUNK.

   La paralelización se implementa utilizando la construcción "parallel for" de OpenMP, que distribuye las iteraciones de los dos bucles externos entre los hilos disponibles. El número de hilos se obtiene utilizando la función omp_get_num_threads(), y cada hilo tiene sus propias copias privadas de las matrices a, b y c.

   Las funciones Sample_Init, Sample_Start, Sample_Stop y Sample_End forman parte de un marco de medición de rendimiento llamado "Sample". Estas funciones se utilizan para medir el tiempo de ejecución de la multiplicación de matrices e imprimir los resultados.

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>
# include "sample.h"

# ifndef MIN
# define MIN(x,y) ((x)<(y)?(x):(y))
# endif
```

```
# define DATA_SZ (1024*1024*64*3)

static double  MEM_CHUNK[DATA_SZ];

void Matrix_Init_col(int SZ, double *a, double *b, double *c)
{
  int j,k;

  for (k=0; k<SZ; k++)
    for (j=0; j<SZ; j++) {
      a[j+k*SZ] = 2.0*(j+k);
      b[j+k*SZ] = 3.2*(j-k);
      c[j+k*SZ] = 1.0;
    }
}

int main (int argc, char **argv)
{
  int  N;

  if (argc < 2) {
    printf("MM1c MatrixSize [Sample arguments ...]\n");
    return -1;
  }

  N  = (int) atof(argv[1]); argc--; argv++;

  if (N > 1024*8)
  {
    printf("Unvalid MatrixSize\n");
    return -1;
  }

  Sample_Init(argc, argv);

#pragma omp parallel
{
  int     NTHR, THR, SZ;
  int     i, j, k;
  double  *a, *b, *c;

  SZ    = N;
  Sample_PAR_install();
  NTHR = omp_get_num_threads();
```

```
  a = MEM_CHUNK;
  b = a + SZ*SZ;
  c = b + SZ*SZ;

#pragma omp master
  Matrix_Init_col(SZ, a, b, c);

  Sample_Start(THR);

#pragma omp for
  for (i=0; i<SZ; i++)
    for (j=0; j<SZ; j++) {
      double *pA, *pB, S;
      S=0.0;
      pA = a+(i*SZ); pB = b+j;
      for (k=SZ; k>0; k--, pA++, pB+=SZ)
        S += (*pA * *pB);
      c[i*SZ+j]= S;
    }

  Sample_Stop(THR);
}

  Sample_End();
}
```

El algoritmo presentado tiene una complejidad de $O(N^3)$, donde N es el tamaño de la matriz. Esto se debe a que se realizan tres bucles anidados para recorrer los elementos de las matrices y realizar las operaciones de multiplicación y suma. En cada iteración de los bucles, se realiza una operación de multiplicación y una operación de suma, lo cual implica un total de N^3 operaciones para completar la multiplicación de matrices. Por lo tanto, la complejidad del algoritmo es cúbica en función del tamaño de la matriz.

2. *MM1f:*

   La diferencia entre los códigos MM1c y MM1f está en la forma en que se accede a la matriz b en el bucle más interno. En MM1c, se accede a los elementos de b usando pB += SZ, lo que incrementa el puntero pB en elementos SZ en cada iteración. Esto significa que se accede a los elementos de b en orden de columna principal, lo que puede ser más eficiente en algunos casos porque conduce a una mejor localidad de caché.

Por el contrario, MM1f accede a los elementos de b en orden de fila principal, que es el orden natural en el que se almacena la matriz en la memoria. Esto se hace incrementando el puntero pB en 1 en cada iteración del bucle más interno.

```
# include <stdlib.h>
# include <stdio.h>
//# include <omp.h>
# include "sample.h"

# ifndef MIN
# define MIN(x,y) ((x)<(y)?(x):(y))
# endif

# define DATA_SZ (1024*1024*64*3)

static double  MEM_CHUNK[DATA_SZ];

void Matrix_Init_col(int SZ, double *a, double *b, double *c)
{
  int j,k;

  for (k=0; k<SZ; k++)
    for (j=0; j<SZ; j++) {
      a[j+k*SZ] = 2.0*(j+k);
      b[k+j*SZ] = 3.2*(j-k);
      c[j+k*SZ] = 1.0;
    }
}

int main (int argc, char **argv)
{
  int  N;

  if (argc < 2) {
    printf("MM1c MatrixSize [Sample arguments ...]\n");
    return -1;
  }

  N  = (int) atof(argv[1]); argc--; argv++;

  if (N > 1024*8)
  {
    printf("Unvalid MatrixSize\n");
    return -1;
  }
```

```
  Sample_Init(argc, argv);

#pragma omp parallel
{
  int     NTHR, THR, SZ;
  int     i, j, k;
  double  *a, *b, *c;

  SZ    = N;
  THR = Sample_PAR_install();
  NTHR = omp_get_num_threads();

  a = MEM_CHUNK;
  b = a + SZ*SZ;
  c = b + SZ*SZ;

#pragma omp master
  Matrix_Init_col(SZ, a, b, c);

  Sample_Start(THR);

#pragma omp for
  for (i=0; i<SZ; i++)
    for (j=0; j<SZ; j++) {
      double *pA, *pB, S;
      S=0.0;
      pA = a+(i*SZ); pB = b+(j*SZ);
      for (k=SZ; k>0; k--, pA++, pB++)
        S += (*pA * *pB);
      c[i*SZ+j]= S;
    }

  Sample_Stop(THR);
}

  Sample_End();
}
```

El algoritmo presentado tiene una complejidad de $O(N^3)$, donde N es el tamaño de la matriz. Al igual que en el algoritmo anterior, se realizan tres bucles anidados para recorrer los elementos de las matrices y realizar las operaciones de multiplicación y suma. Aunque se ha eliminado la directiva de OpenMP para la

programación paralela, el orden de complejidad sigue siendo el mismo. La complejidad cúbica se mantiene debido a la naturaleza del problema de multiplicación de matrices, que requiere visitar cada elemento de las matrices de entrada una vez. Por lo tanto, la complejidad del algoritmo sigue siendo $O(N^3)$.

3. *MM1fu:*

   El código MM1fu es una versión optimizada del algoritmo de multiplicación de matrices MM1c que utiliza un enfoque de vectorización para mejorar el rendimiento. En lugar de recorrer las matrices elemento por elemento, la implementación vectoriza los cálculos utilizando operaciones en bloques de cuatro elementos a la vez, lo que puede aprovechar mejor las características de la arquitectura del procesador.

   En MM1fu, se agregó un bucle adicional en el que se realizan los cálculos en bloques de cuatro elementos y se suman los resultados de cada bloque. Esto reduce el número de operaciones de acceso a la memoria necesarias, lo que a su vez reduce el tiempo de ejecución y aumenta la eficiencia.

   Además, se utiliza OpenMP para paralelizar el bucle exterior y aprovechar el poder de procesamiento de múltiples núcleos de CPU. Al igual que en MM1c, se utiliza la biblioteca Sample para medir el tiempo de ejecución y la utilización de CPU.

```
# include <stdlib.h>
# include <stdio.h>
# include <omp.h>
# include "sample.h"

# ifndef MIN
# define MIN(x,y) ((x)<(y)?(x):(y))
# endif

# define DATA_SZ (1024*1024*64*3)

static double  MEM_CHUNK[DATA_SZ];

void Matrix_Init_col(int SZ, double *a, double *b, double *c)
{
  int j,k;

  for (k=0; k<SZ; k++)
    for (j=0; j<SZ; j++) {
      a[j+k*SZ] = 2.0*(j+k);
      b[k+j*SZ] = 3.2*(j-k);
      c[j+k*SZ] = 1.0;
```

```
    }
}

int main (int argc, char **argv)
{
  int  N;

  if (argc < 2) {
    printf("MM1c MatrixSize [Sample arguments ...]\n");
    return -1;
  }

  N  = (int) atof(argv[1]); argc--; argv++;

  if (N > 1024*8)
  {
    printf("Unvalid MatrixSize\n");
    return -1;
  }

  Sample_Init(argc, argv);

#pragma omp parallel
{
  int     NTHR, THR, SZ;
  int     i, j, k;
  double  *a, *b, *c;

  SZ    = N;
  THR = Sample_PAR_install();
  NTHR = omp_get_num_threads();

  a = MEM_CHUNK;
  b = a + SZ*SZ;
  c = b + SZ*SZ;

#pragma omp master
  Matrix_Init_col(SZ, a, b, c);

  Sample_Start(THR);

#pragma omp for
  for (i=0; i<SZ; i++)
    for (j=0; j<SZ; j++) {
      double *pA, *pB, c0, c1, c2, c3;
```

```
        c0=c1=c2=c3=0.0;
        pA = a+(i*SZ);
        pB = b+(j*SZ);
        k=SZ;
        while (k&3) { // in case SZ is not a multiple of 4
          c0 += (*pA * *pB);
          k--; pA++; pB++;
        }
        for (; k>0; k-=4, pA+=4, pB+=4) {
          c0 += (*pA * *pB);
          c1 += (*(pA+1) * *(pB+1));
          c2 += (*(pA+2) * *(pB+2));
          c3 += (*(pA+3) * *(pB+3));
        }
        c[i*SZ+j]= c0+c1+c2+c3;
      }

    Sample_Stop(THR);
  }

    Sample_End();
  }
```

El algoritmo presentado tiene una complejidad de $O(N^3)$, donde N es el tamaño de la matriz. Aunque se ha introducido una optimización en forma de instrucciones SIMD (Single Instruction, Multiple Data), el orden de complejidad no se ve afectado. La optimización SIMD permite realizar operaciones de multiplicación y suma en paralelo para un conjunto de elementos contiguos en la matriz. Sin embargo, el número total de operaciones requeridas para multiplicar todas las entradas de la matriz sigue siendo proporcional a N^3. Por lo tanto, la complejidad del algoritmo sigue siendo $O(N^3)$.

4. *MM2f:*

El algoritmo de multiplicación de matrices empleado es una variación del algoritmo clásico de multiplicación de matrices, donde las matrices de entrada se dividen en bloques de tamaño 2x2 y cada multiplicación de bloques se realiza utilizando solo 8 operaciones de punto flotante (FLOP). El algoritmo utiliza el desenrollado de bucles y el bloqueo de registros para maximizar la eficiencia del cálculo.

El programa toma un argumento de línea de comando para el tamaño de la matriz (N), que debe ser un múltiplo de 2 y no mayor que 8192. El programa inicializa tres matrices de tamaño N*N de números de punto flotante de doble precisión (a, b, c

) con valores aleatorios. La multiplicación de matriz se realiza en paralelo usando OpenMP con cada subproceso trabajando en un subconjunto de los bloques de matriz.

```
# include <omp.h>
# include "sample.h"

# ifndef MIN
# define MIN(x,y) ((x)<(y)?(x):(y))
# endif

# define DATA_SZ (1024*1024*64*3)

static double  MEM_CHUNK[DATA_SZ];

void Matrix_Init_col(int SZ, double *a, double *b, double *c)
{
  int j,k;

  for (k=0; k<SZ; k++)
    for (j=0; j<SZ; j++) {
      a[j+k*SZ] = 2.0*(j+k);
      b[k+j*SZ] = 3.2*(j-k);
      c[j+k*SZ] = 1.0;
    }
}

int main (int argc, char **argv)
{
  int  N;

  if (argc < 2) {
    printf("MM1c MatrixSize [Sample arguments ...]\n");
    return -1;
  }

  N = (int) atof(argv[1]); argc--; argv++;

  if (N > 1024*8 | N&1)
  {
    printf("Unvalid MatrixSize (must be multiple of 2)\n");
    return -1;
  }

  Sample_Init(argc, argv);
```

```
#pragma omp parallel
{
  int     NTHR, THR, SZ;
  int     i, j, k;
  double  *a, *b, *c;

  SZ    = N;
  THR = Sample_PAR_install();
  NTHR = omp_get_num_threads();

  a = MEM_CHUNK;
  b = a + SZ*SZ;
  c = b + SZ*SZ;

#pragma omp master
  Matrix_Init_col(SZ, a, b, c);

  Sample_Start(THR);

#pragma omp for
  for (i=0; i<SZ; i+=2)
    for (j=0; j<SZ; j+=2) {
      double *pA, *pB, c0, c1, c2, c3;
      c0=c1=c2=c3=0.0;
      pA = a+(i*SZ); pB = b+(j*SZ);
      for (k=SZ; k>0; k-=2, pA+=2, pB+=2)
      {
        double a0, a1, a2, a3;
        double b0, b1, b2, b3;

        a0 = *pA;  a1 = *(pA+1);  a2 = *(pA+SZ);  a3 = *(pA+SZ+1);
        b0 = *pB;  b1 = *(pB+1);  b2 = *(pB+SZ);  b3 = *(pB+SZ+1);

        c0 += a0*b0 + a1*b1;
        c1 += a0*b2 + a1*b3;
        c2 += a2*b0 + a3*b1;
        c3 += a2*b2 + a3*b3;
      }
      pB = c+i*SZ+j;
      *pB= c0;  *(pB+1)= c1;  *(pB+SZ)= c2; *(pB+SZ+1)= c3;
    }

  Sample_Stop(THR);
}
```

```
  Sample_End();
}
```

El algoritmo presentado tiene una complejidad de $O(N^{3/2})$. En el bucle más interno del algoritmo, se realiza la multiplicación y suma de cuatro elementos de las matrices en cada iteración. A diferencia del algoritmo clásico de multiplicación de matrices, donde se realizan operaciones individuales para cada elemento, este algoritmo aprovecha la propiedad de que los elementos se agrupan en bloques de 2x2.

El bucle intermedio, que itera sobre las columnas de la matriz, se incrementa de 2 en 2 (j+=2). De manera similar, el bucle externo, que itera sobre las filas de la matriz, también se incrementa de 2 en 2 (i+=2).

Dado que los bucles se incrementan en pasos de 2, el número total de iteraciones se reduce a la mitad en comparación con un algoritmo de multiplicación de matrices tradicional.

En resumen los tres primeros algoritmos tienen una complejidad de $O(N^3)$, mientras que el cuarto algoritmo tiene una complejidad de $O(N^{3/2})$. El primer algoritmo muestra una implementación básica de multiplicación de matrices que requiere tres bucles anidados, uno para las filas, otro para las columnas y otro para los productos de los elementos. En este caso, el número total de operaciones es proporcional a N^3, lo que indica una complejidad de $O(N^3)$. El segundo y tercer algoritmo también son variaciones del algoritmo clásico de multiplicación de matrices, pero con mejoras en la gestión de la memoria y en el orden de los bucles. Aunque estas mejoras pueden optimizar el rendimiento, la cantidad total de operaciones sigue siendo proporcional a N^3, por lo que la complejidad se mantiene en $O(N^3)$. En cambio, el cuarto algoritmo utiliza bloques de 2x2 para reducir el número total de iteraciones. Esto resulta en una complejidad reducida de $O(N^{3/2})$, lo que indica una mejora en la eficiencia del algoritmo en comparación con los tres primeros.

Luego se implemento el mismo proceso en los mismos tamaños de matriz y en cantidad de treads de 2,4,6,8,10,12 y 14 para la multiplicación de matrices con MPI, adicional se ejecuto en 2 nodos. Nuevamente se calcula la media del tiempo de ejecución en segundos para obtener los resultados finales.

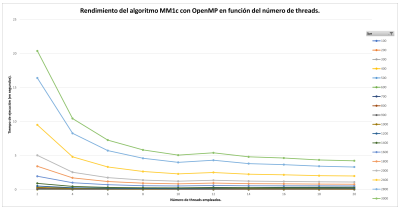
4 Resultados y análisis

En esta sección, se presentan los resultados de los experimentos para el desempeño de los cuatro programas. Los resultados probados se han dividido en dos partes; la implementación con OpenMP y la implementación con MPI.

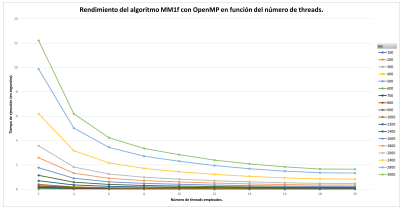
1. **Implementación con OpenMP:**

   Para fines de evaluación comparativa, se crearon 18 conjuntos de datos: aumentando el tamaño de la matriz de 100 a 1000 usando el paso 100 y aumentando el tamaño de la matriz de 1000 a 3000 usando el paso 200. Luego, estos conjuntos de datos se aplicaron a cada algoritmo. Para garantizar que el proceso de evaluación comparativa sea consistente, cada algoritmo se ejecutó 30 veces y se calculó el tiempo promedio de ejecución.

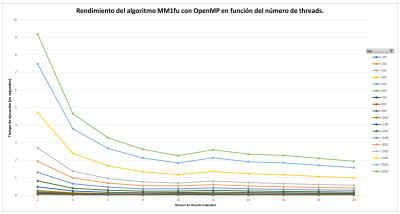
   La Fig. 1 proporciona información sobre el rendimiento de los cuatro algoritmos de multiplicación de matrices implementados con OpenMP en diferentes tamaños de matrices y número de threads. En general, se observa que todos los algoritmos mejoran significativamente su rendimiento al aumentar el número de threads empleados en la ejecución. Sin embargo, se puede observar que la mejora en el rendimiento se ralentiza a medida que se aumenta el número de threads empleados en la ejecución. En particular, se puede observar que para el algoritmo MM1c, la mejora en el rendimiento se ralentiza después de alcanzar alrededor de 10 a 12 hilos. Para MM1f y MM1fu, la mejora en el rendimiento se ralentiza después de alcanzar alrededor de 8 hilos. Para MM2f, la mejora en el rendimiento se ralentiza después de alcanzar alrededor de 6 a 8 hilos.



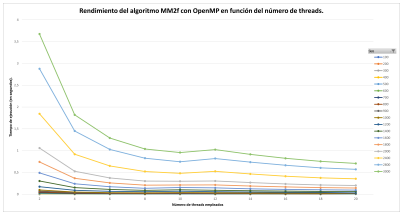
(a) MM1c



(b) MM1f



(c) MM1fu



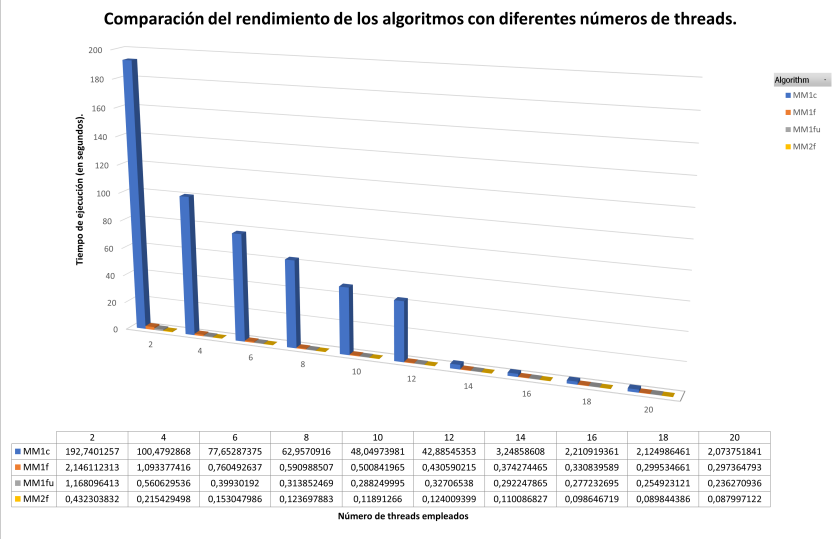
(d) MM2f

Fig. 1: OpenMP implementation

En la Fig. 2 también se puede observar que el desempeño generalmente mejora a medida que se aumenta el número de threads. Además se puede observar que MM2f es el código más eficiente en términos de rendimiento, ya que logra obtener los

tiempos de ejecución más cortos en comparación con los otros códigos, para todas las combinaciones de tamaño de matriz y número de threads que se han probado. MM1fu y MM1f también tienen un buen rendimiento, ligeramente inferior al de MM2f. Por otro lado, MM1c es el código con el rendimiento más bajo de los cuatro.

En cuanto a los códigos, MM2f y MM1fu utilizan una estrategia de particionamiento de matrices más eficiente que MM1f y MM1c, lo que les permite aprovechar mejor el paralelismo y mejorar su rendimiento. Además, los algoritmos en MM2f y MM1fu también están diseñados para minimizar la cantidad de operaciones de memoria necesarias, lo que reduce el tiempo de acceso a la memoria y mejora la eficiencia.



	2	4	6	8	10	12	14	16	18	20
MM1c	192,7401257	100,4792868	77,65287375	62,9570916	48,04973981	42,88545353	3,24858608	2,210919361	2,124986461	2,073751841
MM1f	2,146112313	1,093377416	0,760492637	0,590988507	0,500841965	0,430590215	0,374274465	0,330839589	0,299534661	0,297364793
MM1fu	1,168096413	0,560629536	0,39930192	0,313852469	0,288249995	0,32706538	0,292247865	0,277232695	0,254923121	0,236270936
MM2f	0,432303832	0,215429498	0,153047986	0,123697883	0,11891266	0,124009399	0,110086827	0,098646719	0,089844386	0,087997122

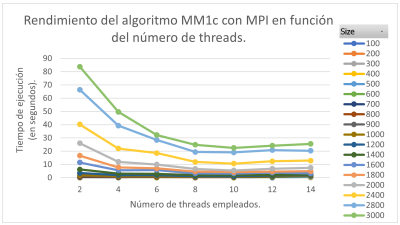
Fig. 2: Rendimiento de los algoritmos para diferentes números de threads.

2. **Implementación con MPI:**

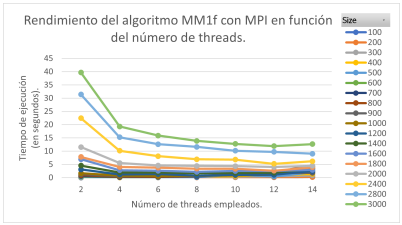
Del mismo modo que en la implementación MPI, para fines de evaluación comparativa, se crearon 18 conjuntos de datos: aumentando el tamaño de la matriz de 100 a 1000 usando el paso 100 y aumentando el tamaño de la matriz de 1000 a 2000 usando el paso 200. Luego, estos conjuntos de datos se aplicaron a cada algoritmo. Para garantizar que el proceso de evaluación comparativa sea consistente, cada algoritmo desarrollado se volvió a ejecutar 30 veces y se calculó el tiempo promedio de ejecución.

La Fig. 3 proporciona información sobre el rendimiento de los cuatro algoritmos de multiplicación de matrices implementados con MPI en diferentes tamaños de

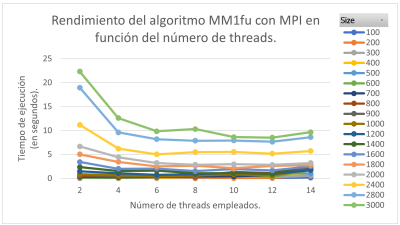
matrices y número de threads. Al igual que con la implementación OpenMP, se observa que todos los algoritmos mejoran significativamente su rendimiento al aumentar el número de threads empleados en la ejecución. Sin embargo, se puede observar que la mejora en el rendimiento se ralentiza a medida que se aumenta el número de threads empleados en la ejecución. En particular, se puede observar que para el algoritmo MM1c, la mejora en el rendimiento se ralentiza después de alcanzar alrededor de 8 a 10 hilos. Para MM1f y MM1fu, la mejora en el rendimiento se ralentiza después de alcanzar alrededor de 6 hilos. Para MM2f, la mejora en el rendimiento se ralentiza después de alrededor de 6 a 8 hilos, al igual que con la implementación OpenMP.



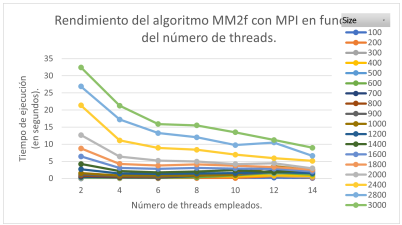
(a) MM1c



(b) MM1f



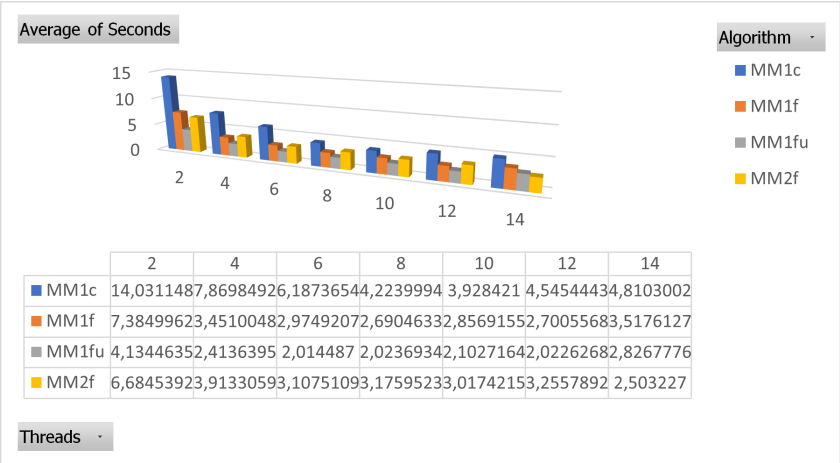
(c) MM1fu



(d) MM2f

Fig. 3: MPI implementation

En la Fig. 4 se puede notar que el rendimiento de los cuatro códigos mejora en términos generales a medida que se aumenta el número de threads. Sin embargo, es interesante destacar que la implementación MPI muestra una curva más homogénea en comparación con las implementaciones OpenMP, lo que indica que MPI es menos dependiente del número de hilos que se utilizan. Además, se puede observar que los códigos MM2f, MM1f y MM1fu logran obtener los tiempos de ejecución más cortos, lo que significa que son los códigos más eficientes en términos de rendimiento. Por otro lado, MM1c parece tener un rendimiento inferior en comparación con los otros códigos, lo que puede deberse a la forma en que se han optimizado los bucles de los códigos. En cuanto al aumento de los hilos, la gráfica sugiere que a medida que se aumentan los hilos, los cuatro códigos parecen tener un desempeño similar en términos de tiempo.



	2	4	6	8	10	12	14
MM1c	14,0311487	7,86984926	6,18736544	4,2239994	3,928421	4,54544434	4,8103002
MM1f	7,38499623	4,45100482	2,97492072	2,69046332	2,85691552	2,70055683	3,5176127
MM1fu	4,13446352	4,4136395	2,014487	2,02369342	2,10271642	2,02262682	2,8267776
MM2f	6,68453923	3,91330593	3,10751093	3,17595233	3,01742153	3,2557892	2,503227

Fig. 4: Rendimiento de los algoritmos para diferentes números de threads.

3. **OpenMP vs MPI:**

Ambas librerías de programación, MPI y OpenMP, son utilizadas para implementar paralelismo en sistemas con múltiples núcleos, permitiendo la ejecución concurrente de múltiples tareas. Sin embargo, MPI es una biblioteca de paso de mensajes, mientras que OpenMP es una biblioteca compartida de memoria.

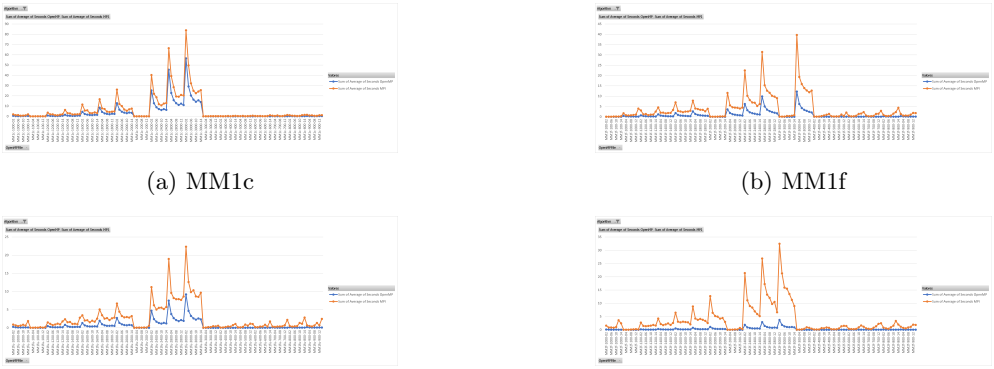
Según los resultados, ambas bibliotecas de programación mejoran el rendimiento de los algoritmos a medida que se aumenta el número de hilos o threads empleados en la ejecución. Sin embargo, la mejora del rendimiento se ralentiza a medida que se aumenta el número de hilos en la ejecución. En particular, MM1c es el algoritmo con el rendimiento más bajo de los cuatro en ambas implementaciones. Para OpenMP, la mejora en el rendimiento de MM1c se ralentiza después de alcanzar alrededor de 10 a 12 hilos, mientras que para MPI, la mejora se ralentiza después de alcanzar alrededor de 8 a 10 hilos.

MM2f es el algoritmo más eficiente en términos de rendimiento en ambas implementaciones, ya que logra obtener los tiempos de ejecución más cortos en comparación con los otros códigos, para todas las combinaciones de tamaño de matriz y número de hilos que se han probado. MM1fu y MM1f también tienen un buen rendimiento, aunque ligeramente inferior al de MM2f.

En cuanto a la implementación, MM2f y MM1fu utilizan una estrategia de particionamiento de matrices más eficiente que MM1f y MM1c, lo que les permite aprovechar mejor el paralelismo y mejorar su rendimiento. Además, los algoritmos

en MM2f y MM1fu también están diseñados para minimizar la cantidad de operaciones de memoria necesarias, lo que reduce el tiempo de acceso a la memoria y mejora la eficiencia.

En cuanto a las diferencias entre MPI y OpenMP, la Fig. 5 sugiere que MPI es menos dependiente del número de hilos que se utilizan, lo que indica que puede ser una mejor opción para sistemas con un número variable de núcleos o procesadores. Sin embargo, se necesitan más pruebas para llegar a conclusiones más sólidas.



(a) MM1c (b) MM1f

(c) MM1fu (d) MM2f

Fig. 5: OpenMP vs MPI

En términos de rendimiento, OpenMP es más eficiente que MPI en sistemas de memoria compartida ya que la comunicación entre los hilos se realiza a través de la memoria compartida, lo que reduce el tiempo de comunicación. En sistemas distribuidos, la comunicación a través de MPI puede ser más costosa en términos de tiempo debido a la latencia de la red.

En resumen, MPI y OpenMP son paradigmas de programación paralela utilizados para aplicaciones específicas. MPI se utiliza para sistemas distribuidos, mientras que OpenMP se utiliza para sistemas de memoria compartida. MPI es altamente escalable, pero menos eficiente en términos de rendimiento en sistemas de memoria compartida, mientras que OpenMP es más eficiente en sistemas de memoria compartida pero menos escalable que MPI.

5 Conclusión

En conclusión, los resultados presentados muestran que el rendimiento de los algoritmos de multiplicación de matrices mejora significativamente al aumentar el número de hilos utilizados en la ejecución, tanto para la implementación OpenMP como para la implementación MPI. Sin embargo, se puede observar que la mejora en el rendimiento

se ralentiza a medida que se aumenta el número de hilos, y cada algoritmo presenta un límite en cuanto al número de hilos que pueden utilizarse antes de que la mejora en el rendimiento disminuya significativamente.

En cuanto a los algoritmos en sí, se puede observar que MM2f es el código más eficiente en términos de rendimiento, tanto para la implementación OpenMP como para la implementación MPI. MM1fu y MM1f también tienen un buen rendimiento, aunque ligeramente inferior al de MM2f. Por otro lado, MM1c es el código con el rendimiento más bajo de los cuatro.

En cuanto a la implementación, se puede observar que MM2f y MM1fu utilizan una estrategia de particionamiento de matrices más eficiente que MM1f y MM1c, lo que les permite aprovechar mejor el paralelismo y mejorar su rendimiento. Además, los algoritmos en MM2f y MM1fu también están diseñados para minimizar la cantidad de operaciones de memoria necesarias, lo que reduce el tiempo de acceso a la memoria y mejora la eficiencia.

La elección del mejor modelo a la hora de desarrollar una aplicación en paralelo depende de múltiples factores. La implementación de MPI requiere más esfuerzo y experiencia para implementar de manera efectiva. Una de las ventajas del modelo MPI sobre otros cuatro es que MPI se puede utilizar tanto en sistemas de memoria compartida como en sistemas de memoria distribuida.

En general, se puede concluir que la implementación de algoritmos de multiplicación de matrices utilizando técnicas de paralelización es una estrategia efectiva para mejorar el rendimiento en términos de tiempo de ejecución. Además, se puede observar que el uso de técnicas de particionamiento de matrices eficientes y la minimización de operaciones de memoria pueden tener un impacto significativo en el rendimiento de los algoritmos.

References

[1] Al-Mulhem, M.S., Aidhamin, A., Al-Shaikh, R.: On benchmarking the matrix multiplication algorithm using OpenMP, MPI and CUDA programming languages. ResearchGate (2013)

[2] Tim.Lewis: cOMPunity - The Independent Community for OpenMP (2019). https://www.openmp.org/compunity/

[3] Nayfeh, B.A., Olukotun, K.: A single-chip multiprocessor. IEEE Computer **30**(9), 79–85 (1997) https://doi.org/10.1109/2.612253

[4] Jerraya, A.A., Tenhunen, H., Wolf, W.: Guest Editors' Introduction: Multiprocessor Systems-on-Chips. IEEE Computer **38**(7), 36–40 (2005) https://doi.org/10.1109/mc.2005.231

[5] Gorder, P.F.: Multicore Processors for Science and Engineering. Computing in Science and Engineering **9**(2), 3–7 (2007) https://doi.org/10.1109/mcse.2007.35

[6] Academy, S.: CS201: Words in Computer Architecture — Saylor Academy. https://learn.saylor.org/mod/page/view.php?id=18960

[7] Gorder, P.F.: Multicore Processors for Science and Engineering. Computing in Science and Engineering **9**(2), 3–7 (2007) https://doi.org/10.1109/mcse.2007.35

[8] Quinn, M.A.: Parallel Programming in C with MPI and OpenMP, (2003). http://www.inf.puc-rio.br/noemi/cd-06/cd3.pdf

[9] Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics (2003). https://ieeexplore.ieee.org/document/1592961

[10] Chandrashekar, B.N., Shastry, K.A., Manjunath, B.A., Geetha, V.: Performance Model of HPC Application On CPU-GPU Platform*. (2022). https://doi.org/10.1109/mysurucon55714.2022.9972737 . https://doi.org/10.1109/mysurucon55714.2022.9972737

[11] Syberfeldt, A.: A Comparative Evaluation of the GPU vs The CPU for Parallelization of Evolutionary Algorithms Through Multiple Independent Runs (2021). https://ssrn.com/abstract=3937048

[12] Pacheco, P., Malensek, M.: An Introduction to Parallel Programming. Morgan Kaufmann, ??? (2021)

[13] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., McDonald, J.: Parallel Programming in OpenMP. Morgan Kaufmann, ??? (2001)

[14] Fast Multidimensional Matrix Multiplication on CPU from Scratch (2022). https://siboehm.com/articles/22/Fast-MMM-on-CPU