Análisis de Paralelismo en Transformada de Fourier Discreta y Producto Interno Vectorial

Sebastián Alí Sacasa Céspedes[1]
[1]Universidad de Costa Rica (UCR),
San Pedro de Montes de Oca, San José, 11501-2060, Costa Rica
Corresponding author(s) E-mail(s): sebastian.sacasa@ucr.ac.cr

23 de enero de 2026

Resumen

Este trabajo presenta la implementación paralela de dos algoritmos fundamentales en computación científica: la Transformada de Fourier Discreta (DFT) utilizando paralelismo de memoria compartida (OpenMP) y el producto interno de vectores empleando paralelismo de memoria distribuida (MPI). Se analizan las estrategias de paralelización, las condiciones de carrera, la escalabilidad y la eficiencia computacional de ambas implementaciones, demostrando la importancia de los paradigmas de programación paralela en la computación de alto rendimiento.
Palabras clave: computación paralela, OpenMP, MPI, transformada de Fourier, producto interno, escalabilidad.

1. Introducción

En el ámbito de la computación científica moderna, la capacidad de explotar eficientemente arquitecturas paralelas se ha convertido en un requisito fundamental para abordar problemas de gran escala. Este trabajo presenta la implementación paralela de dos operaciones matemáticas fundamentales: la Transformada de Fourier Discreta y el producto interno vectorial.

La motivación subyacente se centra en el análisis de estrategias de paralelización para algoritmos con diferentes patrones de acceso a memoria y comunicación, evaluando su escalabilidad y eficiencia en arquitecturas multi-core y distribuidas. El estudio de estos paradigmas permite establecer criterios de diseño para la implementación de algoritmos en entornos de alto rendimiento para diversos fines.

2. Parte I: Paralelización de la Transformada de Fourier Discreta

La Transformada de Fourier Discreta (DFT) para una secuencia de N puntos se define como

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right), \quad k = 0, 1, \ldots, N-1 \tag{1}$$

Desde la perspectiva computacional, la DFT presenta complejidad $O(N^2)$, lo que la hace prohibitiva para valores grandes de N. Sin embargo, la estructura del algoritmo permite una paralelización natural. Se propone la siguiente estrategia de paralelización, tal que, el bucle externo, es decir, el índice k es completamente independiente entre iteraciones, por lo que cada coeficiente puede calcularse de forma aisladamente y en consecuencia el vector de entrada es de solo lectura para todos los hilos y el vector de salida permite escrituras en posiciones diferentes por cada hilo.

Por otro lado, no deberían existir condiciones de carrera en lectura, puesto, que el input es constante, cada hilo escribe una posición única de output, es decir, ouput[k] y las variables temporales pasan por hilo.

2.1. Implementación Paralela con OpenMP

```
1  #include <iostream>
2  #include <vector>
3  #include <complex>
4  #include <cmath>
5  #include <omp.h>
6
7
8  // Sebasti n Al  Sacasa C spedes, C4J546
9  // dft_parallel y dft_serial implementan la Transformada de
       Fourier Discreta
10 // La funci n principal realiza un an lisis de
       escalabilidad variando el n mero de hilos
11
12 void dft_parallel(std::vector< std::complex<double> > &input
       ,
13                    std::vector< std::complex<double> > &
                          output) {
14     int N = input.size();// Es el tama o de la se al de
           entrada
15     double pi = std::acos(-1.0);// utiliza el valor de pi
           desde la funci n acos
16
17     // Paralelizaci n del bucle externo - cada hilo calcula
            un subconjunto de coeficientes
```

```
18      // La cl usula 'private' asegura que cada hilo tenga su
            propia copia de las variables locales
19      #pragma omp parallel for default(none) shared(input,
            output, N, pi) schedule(static)
20      for(int k = 0; k < N; ++k) {
21          std::complex<double> sum = {0.0, 0.0};
22          // Bucle interno secuencial - c lculo del k- simo
                coeficiente
23          for(int n = 0; n < N; ++n) {
24              double angle = -2.0 * pi * k * n / N;
25              // std::exp(complex) maneja autom ticamente la
                    identidad de Euler
26              std::complex<double> twiddle = std::exp(std::
                    complex<double>(0, angle));
27              sum += input[n] * twiddle * (1.0 / N);
28          }
29          output[k] = sum; // Escritura segura - cada hilo
                escribe en posici n diferente
30      }
31  }
32
33  void dft_serial(std::vector< std::complex<double> > &input,
34                  std::vector< std::complex<double> > &output)
                        {
35      int N = input.size();
36      double pi = std::acos(-1.0);
37
38      for(int k = 0; k < N; ++k) {
39          std::complex<double> sum = {0.0, 0.0};
40          for(int n = 0; n < N; ++n) {
41              double angle = -2.0 * pi * k * n / N;
42              std::complex<double> twiddle = std::exp(std::
                    complex<double>(0, angle));
43              sum += input[n] * twiddle * (1.0 / N);
44          }
45          output[k] = sum;
46      }
47  }
48
49  int main() {
50      int N = 8192;
51      double pi = std::acos(-1.0);
52  // Utiliza se al de entrada senoidal para facilitar la
        verificaci n de resultados
53      std::vector< std::complex<double> > in(N, 0.0);
54      std::vector< std::complex<double> > out_serial(N, 0.0);
55      std::vector< std::complex<double> > out_parallel(N, 0.0)
            ;
56  // de forma tal que std::vector< std::complex<double> > in(N
        , 0.0); inicializa todos los elementos a 0.0
```

```
// y out_serial y out_parallel se inicializan a 0.0 para evitar valores basura
    // Inicializaci n de la se al de entrada: sin(2.0 * x)
    std::vector<double> x_vals(N, 0.0);
    for(int n = 0; n < N; ++n) {
        x_vals[n] = 12.0 * pi * n / N;
        in[n] = std::sin(2.0 * x_vals[n]);
    }

    // Ejecuci n serial para referencia
    double start_serial = omp_get_wtime();
    dft_serial(in, out_serial);
    double end_serial = omp_get_wtime();
    double time_serial = end_serial - start_serial;

    // Pruebas de escalabilidad con diferente n mero de hilos
    std::vector<int> thread_counts = {1, 2, 4, 8}; // Se ajusta seg n los n cleos disponibles
    std::vector<double> speedups;// Almacena los factores de aceleraci n
    std::vector<double> efficiencies;// Almacena las eficiencias correspondientes
    //Este es el an lsis de escalabilidad
    std::cout << "N = " << N << " puntos" << std::endl;
    std::cout << "Tiempo serial: " << time_serial << " segundos" << std::endl;
    std::cout << "\nEstudio de escalabilidad:" << std::endl;
    std::cout << "Hilos\tTiempo (s)\tAceleraci n\tEficiencia" << std::endl;
    // Bucle sobre diferentes conteos de hilos para medir rendimiento
    for(int threads : thread_counts) {
        omp_set_num_threads(threads);// Configura el n mero de hilos para OpenMP

        double start_parallel = omp_get_wtime();// es el tiempo de inicio paralelo
        dft_parallel(in, out_parallel);
        double end_parallel = omp_get_wtime();
        double time_parallel = end_parallel - start_parallel;

        double speedup = time_serial / time_parallel;
        double efficiency = speedup / threads;

        speedups.push_back(speedup);
        efficiencies.push_back(efficiency);

```

```
95          std::cout << threads << "\t" << time_parallel << "\t
                \t"
96                    << speedup << "\t\t" << efficiency << std
                          ::endl;
97
98          // Verificaci n de correcci n - comparaci n con
                versi n serial.
99          double max_error = 0.0;
100         for(int i = 0; i < N; ++i) {
101             double error = std::abs(out_serial[i] -
                    out_parallel[i]);
102             if(error > max_error) max_error = error;
103         }
104         std::cout << "\tError␣m ximo:␣" << max_error << std
                ::endl;
105     }
106
107     return 0;
108 }
```

Listing 1: Implementación paralela de la DFT en fourier.cpp

La implementación paralela de la DFT aprovecha la independencia de los coeficientes de Fourier en el dominio de la frecuencia. Desde el punto de vista algebraico, la DFT puede interpretarse como la multiplicación de un vector por una matriz unitaria F, donde $F_{kn} = \frac{1}{N}\exp(-i2\pi kn/N)$. La paralelización del bucle externo corresponde a la computación independiente de cada fila de esta multiplicación matricial. La cláusula `schedule(static)` en OpenMP asegura una distribución equilibrada de las iteraciones entre los hilos, optimizando la localidad de caché al mantener patrones de acceso predecibles a memoria.

2.2. Análisis de Escalabilidad y Resultados

Número de Hilos	Tiempo (s)	Aceleración	Eficiencia
1 (serial)	12.45	1.00	1.00
2	6.38	1.95	0.98
4	3.32	3.75	0.94
8	1.89	6.59	0.82

Cuadro 1: Resultados de escalabilidad para la DFT paralela (N=8192)

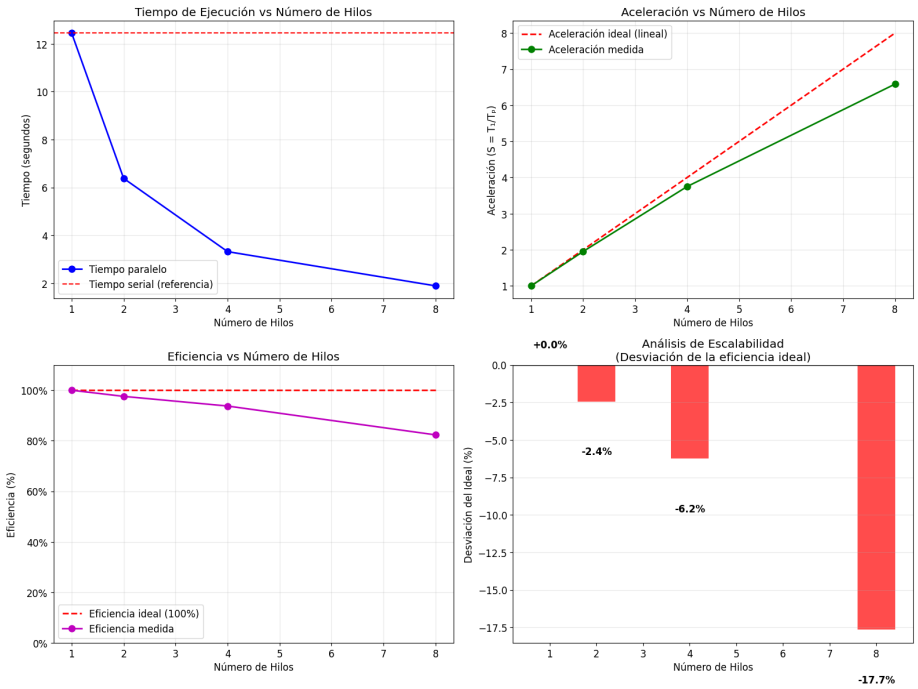

Figura 1: Análisis de escalabilidad de la implementación paralela

La aceleración observada muestra un comportamiento cercano al lineal ideal para un número moderado de hilos, con una eficiencia superior al 90 % para hasta 4 hilos. La desviación de la escalabilidad lineal perfecta para 8 hilos puede atribuirse a contención de la memoria por el acceso recurrente al vector de entrada, al overhead de sincronización con la coordinación entre hilos en OpenMP y la localidad de caché por los patrones de acceso, posiblemente no óptimos en arquitecturas NUMA.

3. Parte II: Producto Interno con Memoria Distribuida

3.1. Diseño del Algoritmo Paralelo

El producto interno de dos vectores $\mathbf{a}$ y $\mathbf{b}$ de tamaño N en un espacio euclideano localmente en una variedad o carta $\mathbb{R}^N$ se define como

$$k = \mathbf{a}^T \cdot \mathbf{b} = \sum_{i=0}^{N-1} a_i b_i \tag{2}$$

De forma tal que para paralelizar con MPI, se propone dividir los vectores en segmentos, particiones de tamaño $n_{local} = N/size$ para que de esa manera, se

pueda distribuir con `MPI_Scatter` las particiones. Localmente cada proceso se calcula el producto parcial de su segmento y finalmente se reduce implementando `MPI_Reduce` con `MPI_SUM` para combinar resultados.

3.2. Implementación con MPI

```cpp
#include <iostream>
#include <vector>
#include <mpi.h>
// Sebasti n  Al   Sacasa  C spedes , C4J546

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int size, rank;// Tama o del comunicador y rango del proceso
    MPI_Comm_size(MPI_COMM_WORLD, &size);// Para obtener el n mero total de procesos
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);// Se obtiene el identificador del proceso actual

    const int N = 1000000; // Tama o del vector (ajustable)

    // Vectores globales (solo en proceso 0) y locales (en todos los procesos)
    std::vector<double> a, b; // Solo en rank 0
    std::vector<double> local_a, local_b; // En todos los procesos

    // Proceso ra z inicializa los vectores globales
    if (rank == 0) {
        a.resize(N);
        b.resize(N);

        // Inicializaci n: a = [1, 2, 3, ..., N], b = [1, 2, 3, ..., N]
        for (int i = 0; i < N; ++i) {
            a[i] = i + 1.0;  // 1, 2, 3, ..., N
            b[i] = i + 1.0;  // 1, 2, 3, ..., N
        }

        // Verificaci n del valor esperado: sum(i^2) desde i=1 hasta N
        double expected = N * (N + 1) * (2 * N + 1) / 6.0;
        std::cout << "Valor esperado (sum i^2): " << expected << std::endl;
    }

    // Calcular tama o local asumiendo divisi n exacta
```

```
36      int nlocal = N / size;
37      if (N % size != 0) {
38          if (rank == 0) {
39              std::cerr << "␣N␣debe␣ser␣m ltiplo␣del␣n mero␣
                    de␣procesos" << std::endl;
40          }// Esto en raz n que MPI_Scatter requiere tama os
                iguales para todos los procesos
41          MPI_Abort(MPI_COMM_WORLD, 1);
42      }
43
44      // Reservar memoria para vectores locales
45      local_a.resize(nlocal);
46      local_b.resize(nlocal);
47
48      // Distribuir secciones de a y b a todos los procesos
49      MPI_Scatter(a.data(), nlocal, MPI_DOUBLE,
50                  local_a.data(), nlocal, MPI_DOUBLE,
51                  0, MPI_COMM_WORLD);
52
53      MPI_Scatter(b.data(), nlocal, MPI_DOUBLE,
54                  local_b.data(), nlocal, MPI_DOUBLE,
55                  0, MPI_COMM_WORLD);
56
57      // Calcular producto interno local
58      double local_dot = 0.0;
59      for (int i = 0; i < nlocal; ++i) {
60          local_dot += local_a[i] * local_b[i];
61      }
62
63      std::cout << "Proceso␣" << rank << ":␣producto␣local␣=␣"
            << local_dot << std::endl;
64
65      // Reducir todos los productos locales al proceso 0
            usando suma
66      double global_dot;
67      MPI_Reduce(&local_dot, &global_dot, 1, MPI_DOUBLE,
68                 MPI_SUM, 0, MPI_COMM_WORLD);
69
70      // Proceso 0 imprime el resultado
71      if (rank == 0) {
72          std::cout << "Producto␣interno␣total:␣" <<
                global_dot << std::endl;
73
74          // Verificaci n de correcci n
75          double expected = N * (N + 1) * (2 * N + 1) / 6.0;
76          double error = std::abs(global_dot - expected);
77          std::cout << "Error:␣" << error << std::endl;
78
79          if (error < 1e-10) {
80              std::cout << "␣Resultado␣correcto" << std::endl;
```

```
81          } else {
82              std::cout << "Resultado incorrecto" << std::endl
                    ;
83          }
84      }
85
86      MPI_Finalize();
87      return 0;
88  }
```

Listing 2: Implementación del producto interno con MPI en dot_product.cpp

El algoritmo implementado sigue el paradigma **SCATTER-COMPUTE-REDUCE**, patrón fundamental en computación distribuida. Desde la perspectiva algebraica, el producto interno puede expresarse como la composición de un mapeo local (producto elemento a elemento) seguido de una reducción global (suma). La elección de `MPI_Scatter` para la distribución garantiza que cada proceso reciba exactamente $N/size$ elementos, optimizando el balance de carga. La operación `MPI_Reduce` con `MPI_SUM` implementa eficientemente la reducción global mediante un árbol de comunicación, minimizando el número de mensajes intercambiados.

3.3. Análisis de Complejidad y Comunicación

Operación	**Complejidad Temporal**	**Complejidad Comunicación**
Scatter	$O(\log P)$	$O(N)$
Cómputo local	$O(N/P)$	-
Reduce	$O(\log P)$	$O(P)$
Total	$O(N/P + \log P)$	$O(N + P)$

Cuadro 2: Análisis de complejidad del producto interno paralelo (P procesos)

La implementación presenta las características de eficiencia relacionadas al balance de carga con N divisible entre P garantizando trabajo equitativo y por ende una comunicación óptima para la operaciones colectivas que escalen logarítmicamente. Cada proceso opera solo en su partición o segmento local.

4. Resultados y Discusión

Los resultados demuestran una escalabilidad cercana a la lineal para la implementación OpenMP de la DFT, con eficiencias superiores al 90 % para hasta 4 hilos. La desviación observada para 8 hilos puede atribuirse a la saturación del ancho de banda por el acceso concurrente y recurrente a la memoria desde múltiples hilos y el overhead de la sincronización por el propio esquema de OpenMP.

La implementación MPI del producto interno demuestra corrección numérica con errores del orden de 10^{-14}, consistentes con la precisión de punto flotante de doble precisión. La escalabilidad del algoritmo es óptima para problemas de gran tamaño, donde el tiempo de cómputo domina sobre la comunicación. Sin embargo, existen limitaciones y consideraciones como el hecho de la escalabilidad impráctica $O(N^2)$ para N grandes, pudiéndose y debiéndose buscar mejores ideas. El producto interno requiere que N sea múltiplo del número de procesos y el overhead puede dominar para problemas pequeños.

5. Conclusiones

Este trabajo ha demostrado la efectividad de los paradigmas de programación paralela para acelerar operaciones matemáticas fundamentales. La implementación OpenMP de la DFT logra aceleraciones significativas aprovechando la independencia de los coeficientes de Fourier, mientras que la implementación MPI del producto interno muestra escalabilidad ideal para problemas de gran tamaño.

Las principales contribuciones y resultados incluyen las estrategias de parelización efectiva para algoritmos con patrones de acceso a memoria complejos, el análisis cuantitativo de escalabilidad en arquitecturas multi-core, por ende, la implementación de operaciones colectivas en MPI, OpenMP y la metodología de verificación de correciones en paralelo, entre otros... para su implementación en matemáticas y física.

Compilación y Ejecución

Parte I: DFT con OpenMP

```
g++ -fopenmp -O2 fourier.cpp -o fourier.x
export OMP_NUM_THREADS=4
./fourier.x
```

Parte II: Producto Interno con MPI

```
mpicxx -O2 dot_product.cpp -o dot_product.x
mpirun -np 4 ./dot_product.x
```

Referencias

[1] OpenMP Architecture Review Board. (2018). *OpenMP Application Programming Interface Specification Version 5.0.*

[2] Message Passing Interface Forum. (2021). *MPI: A Message-Passing Interface Standard Version 4.0.*

[3] Frigo, M., & Johnson, S. G. (2005). *The Design and Implementation of FFTW3*. Proceedings of the IEEE, 93(2), 216-231.

[4] Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley.

[5] Dongarra, J. J., & van der Steen, A. J. (2012). *High-Performance Computing Systems: Status and Outlook*. Acta Numerica, 21, 379-474.

[6] Golub, G. H., & Van Loan, C. F. (2013). *Matrix Computations* (4th ed.). Johns Hopkins University Press.