Implementación y Pruebas: MPI y OpenMP

Edison Gregorio Paria Fernandez
Universidad Nacional de San Agustín
Arequipa, Perú
eparia@unsa.edu.pe

***Abstract*—Este trabajo presenta el análisis de dos ejemplos del libro "An Introduction to Parallel Programming" de Peter Pacheco [1]. En el primero, se explora la multiplicación matriz-vector utilizando MPI, donde los datos se distribuyen entre procesos para el cálculo. En el segundo, se analizan dos implementaciones de ordenamiento odd-even transposition sort con OpenMP, considerando distribución de datos y paralelismo. Se realiza una evaluación con diferentes tamaños de datos para analizar el rendimiento de las implementaciones en función de la cantidad de los datos.**

***Index Terms*—MPI, OpenMP, multiplicación matriz-vector, ordenamiento odd-even transposition sort.**

I. Introducción

En el ámbito de la programación paralela y distribuida, las tecnologías como MPI (Message Passing Interface) y OpenMP son fundamentales para aprovechar eficientemente la capacidad de cómputo de sistemas con múltiples núcleos o nodos. Este trabajo se centra en la implementación y evaluación de dos ejemplos del libro "An Introduction to Parallel Programming" de Peter Pacheco [1].

En el primer caso, se explorará una multiplicación matriz-vector utilizando MPI, donde se distribuyen los datos entre múltiples procesos para su cálculo. En el segundo caso, se estudiarán dos implementaciones de ordenamiento odd-even transposition sort utilizando OpenMP, abordando la distribución de datos y paralelismo en el cálculo.

II. MPI - Message Passing Interface

MPI [2] es una especificación estándar para la comunicación y coordinación entre procesos en sistemas distribuidos y paralelos. Está diseñado para permitir que múltiples procesos trabajen en conjunto para resolver un problema común, intercambiando mensajes entre sí. MPI es particularmente útil en sistemas donde los procesos se ejecutan en nodos separados y necesitan comunicarse y sincronizarse para llevar a cabo cálculos complejos. La programación con MPI generalmente involucra la partición de tareas y la distribución de datos entre los procesos para trabajar en paralelo.

La multiplicación matriz-vector con MPI es una técnica para distribuir una matriz y un vector entre múltiples procesos y realizar cálculos en paralelo. Se utiliza *MPI_Allgather* para distribuir el vector entre los procesos. Cada proceso calcula una porción local del resultado y luego se utiliza *MPI_Gather* para recopilar los resultados parciales y obtener el resultado global. Esto permite aprovechar el paralelismo y acelerar los cálculos en sistemas con múltiples nodos o núcleos.

Algorithm 1 Mat_vect_mult

```
1: procedure MAT_VECT_MULT(local_A[], local_x[], local_y[], local_m
2:     x ← allocate memory for an array of size n
3:     MPI_Allgather(local_x, local_n, MPI_DOUBLE, x, local_n
4:     for local_i ← 0 to local_m − 1 do
5:         local_y[local_i] ← 0.0
6:         for j ← 0 to n − 1 do
7:             local_y[local_i] ← local_y[local_i] +
   local_A[local_i · n + j] · x[j]
8:         end for
9:     end for
10:    free(x)
11: end procedure
```

En el algoritmo 1 se desarrolla la función de multiplicación de matriz-vector distribuida utilizando la biblioteca MPI. La comunicación entre procesos se realiza mediante *MPI_Allgather* para recolectar datos y asegurar que cada proceso tenga acceso a los mismos datos globales. Luego, se realiza la multiplicación de la matriz local por el vector global para generar el vector de resultados local. Se realizaron las pruebas con diferentes cantidades de datos (ver Tabla I),

TABLE I
Tiempo de procesamiento en la multiplicación matriz-vector

Tamaño	Cantidad de Procesos	Tiempo (ms)
1000	1	12.45
	2	7.32
	4	5.65
	8	4.89
10000	1	135.86
	2	83.72
	4	58.21
	8	49.47
100000	1	1432.98
	2	892.41
	4	623.79
	8	528.66
1000000	1	16054.73
	2	10027.19
	4	7009.54
	8	5983.32

La Tabla I proporciona información sobre el tiempo de procesamiento de la multiplicación matriz-vector para diferentes tamaños de datos (matrices) y diferentes cantidades de

procesos. Se observa que a mayor cantidad de datos mayor el tiempo de procesamiento. La disminución del tiempo de procesamiento al aumentar la cantidad de procesos evidencia el beneficio de la optimización paralela. La división del trabajo entre múltiples procesos permite que el cálculo se realice en paralelo y, por lo tanto, en menos tiempo en comparación con la ejecución secuencial.

III. OpenMP

OpenMP [3] es una API (Application Programming Interface) que permite la programación paralela y multihilo en sistemas con múltiples núcleos en una única máquina. Se basa en directivas de compilador y bibliotecas, lo que facilita la creación de hilos que pueden ejecutarse en paralelo para acelerar el rendimiento de un programa. OpenMP se centra en aprovechar los recursos de cómputo locales disponibles en una máquina [4], como múltiples núcleos de CPU, para dividir tareas en subprocesos y ejecutarlos en paralelo. A menudo se utiliza para optimizar aplicaciones donde los datos se pueden descomponer en partes más pequeñas y paralelas, como bucles y secciones de código.

El ordenamiento odd-even transposition sort es un algoritmo de ordenamiento que puede ser paralelizado con OpenMP. las dos implementaciones son: la primera con directivas *parallel* y *for*, y la segunda con directivas *parallel* y *sections*. Ambas implementaciones dividen el arreglo en partes iguales y realizan intercambios de elementos en pares consecutivos, creando un patrón de ordenamiento. Sin embargo, la segunda implementación ofrece mayor flexibilidad al permitir la ejecución de múltiples tareas dentro de una sección paralela. En la función Odd_Even_Sort (ver Algorithm 2), se utiliza la directiva OpenMP *parallel for* para paralelizar los bucles que realizan las comparaciones y los intercambios en fases impares y pares del algoritmo. Cada hilo en el equipo paralelo ejecutará un conjunto de iteraciones del bucle *for*.

En la función Odd_Even_Sort2 (ver Algorithm 3), se utiliza una combinación de directivas OpenMP *parallel* y *for* para controlar las iteraciones del bucle de fases y los bucles anidados. El bucle de fases (*while*) se ejecuta secuencialmente, mientras que los bucles anidados (*for*) dentro de cada fase se paralelizan. Esto puede ayudar a evitar condiciones de carrera al tener una mayor granularidad en la paralelización y limitar las interacciones entre hilos en cada fase.

La diferencia principal entre ”for” y ”parallel for” en este contexto es que ”parallel for” paraleliza las iteraciones del bucle en hilos separados, mientras que ”parallel” se utiliza para crear un equipo paralelo y ejecutar múltiples bucles anidados en paralelo.

Para probar el tiempo de procesamiento de los datos se realizaron varias pruebas, se esta considerando 4 procesadores ($thread_count = 4$) con diferentes tamaños de datos. (ver Tabla II) Los resultados obtenidos en la tabla II, se observa que con los primeros 1000 datos los tiempos de procesamiento son muy similares, pero con 10000 se puede observar una diferencia en el tiempo de procesamiento entre los dos algoritmos. Odd_Even_Sort2 parece ser ligeramente más eficiente que

Algorithm 2 Odd Even Sort

```
1:  procedure OddEvenSort(a, n, thread_count)
2:      tmp ← 0
3:      for phase ← 0 to n − 1 do
4:          if phase % 2 = 0 then
5:              pragma omp parallel for num_threads(thread_count) default(none) shared(a, n) private(tmp)
6:              for i ← 1 to n − 1 step 2 do
7:                  if a[i − 1] > a[i] then
8:                      tmp ← a[i − 1]
9:                      a[i − 1] ← a[i]
10:                     a[i] ← tmp
11:                 end if
12:             end for
13:         else
14:             pragma omp parallel for num_threads(thread_count) default(none) shared(a, n) private(tmp)
15:             for i ← 1 to n − 2 step 2 do
16:                 if a[i] > a[i + 1] then
17:                     tmp ← a[i + 1]
18:                     a[i + 1] ← a[i]
19:                     a[i] ← tmp
20:                 end if
21:             end for
22:         end if
23:     end for
24: end procedure
```

TABLE II
TIEMPO DE PROCESAMIENTO EN LA MULTIPLICACIÓN MATRIZ-VECTOR

Tamaño	Odd_Even_Sort(ms)	Odd_Even_Sort2 (ms)
1000	2	2
10000	226	205
100000	22958	22312

Odd_Even_Sort. Con datos más grandes (100000), la diferencia en los tiempos de procesamiento entre los algoritmos es más evidente. Odd_Even_Sort2 sigue siendo más rápido. Esto sugiere que la implementación Odd_Even_Sort2, que utiliza una estructura de paralelismo más eficiente, puede ser más beneficiosa cuando se enfrenta a tareas de mayor escala. Sin embargo, es importante tener en cuenta que el rendimiento real de estos algoritmos también depende de la arquitectura del hardware, el número de hilos disponibles y otros factores del sistema.

IV. conclusiones

La utilización de MPI y OpenMP para implementar algoritmos paralelos tiene el potencial de mejorar el rendimiento de las aplicaciones en sistemas de cómputo multicore y distribuido. Sin embargo, es fundamental realizar un análisis cuidadoso de la distribución de datos, el equilibrio de carga

Algorithm 3 Odd Even Sort 2

```
1: procedure OddEvenSort2(a, n, thread_count)
2:     tmp ← 0
3:     phase ← 0
4:     pragma omp parallel num_threads(thread_count) default(none) shared(a, n) private(i, tmp, phase)
5:     while phase < n do
6:         if phase % 2 = 0 then
7:             pragma omp for
8:             for i ← 1 to n − 1 step 2 do
9:                 if a[i − 1] > a[i] then
10:                    tmp ← a[i − 1]
11:                    a[i − 1] ← a[i]
12:                    a[i] ← tmp
13:                end if
14:            end for
15:        else
16:            pragma omp for
17:            for i ← 1 to n − 2 step 2 do
18:                if a[i] > a[i + 1] then
19:                    tmp ← a[i + 1]
20:                    a[i + 1] ← a[i]
21:                    a[i] ← tmp
22:                end if
23:            end for
24:        end if
25:        phase ← phase + 1
26:    end while
27: end procedure
```

y los costos de comunicación para lograr una verdadera aceleración en la ejecución de los programas.

El código fuente de este trabajo esta en: https://github.com/edisonparia/hpp.git en la carpeta trabajo02.

References

[1] P. Pacheco and M. Malensek, *An introduction to parallel programming*. Morgan Kaufmann, 2021.

[2] B. Barker, "Message passing interface (mpi)," in *Workshop: high performance computing on stampede*, vol. 262, Cornell University Publisher Houston, TX, USA, 2015.

[3] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[4] T. G. Mattson, Y. H. He, and A. E. Koniges, *The OpenMP common core: making OpenMP simple again*. MIT Press, 2019.