Desarrollo de Software en Arquitecturas Paralelas

DanielAsensiRoch: **48776120C**

Índice

1. Estrategia propuesta

La estrategia propuesta es distribuir los cálculos de forma cíclica entre los procesos, de modo que cada proceso evalúe un subconjunto de enteros. El número de enteros primos calculados por cada proceso se sumará posteriormente en el proceso root mediante la función *MPI Reduce()*. Además, se medirá el tiempo de cálculo de cada proceso y se enviará al proceso root para calcular el tiempo total de ejecución y el speed-up y eficiencia en comparación con la ejecución secuencial. El proceso root también realizará un cálculo secuencial para obtener el número de enteros primos menores que n y se mostrarán los resultados finales por pantalla.

2. Explicación del código

Se definen varias variables utilizadas en el programa, como n, *n_factor*, *n_min*, y *n_max*, que son valores para el rango de valores para los cuales se contará el número de primos.

primos y primos_parte son variables que contienen el número de primos para el rango de valores que se está considerando en la ejecución.

t0, t1, tiempo_secuencial y tiempo_parcial_total son variables que se utilizan para medir el tiempo de ejecución del programa.

rank y num_procs son variables que contienen el identificador del proceso y el número total de procesos en la ejecución paralela, respectivamente.

```
// valores para el rango de valores para los cuales se contar  el n mero de primos.
    int n, n_factor, n_min, n_max;
    // variables que contienen el n mero de primos para el rango de valores que se est
    considerando en la ejecuci n.
    int primos, primos_parte;
    // variables que se utilizan para medir el tiempo de ejecuci n del programa.
    double t0, t1, tiempo_secuencial, tiempo_parcial_total;
    // variables que contienen el identificador del proceso y el n mero total de procesos
    en la ejecuci n paralela, respectivamente.
    int rank, num_procs;
```

Se imprimen algunas líneas de texto para indicar el propósito del programa. Luego, se inicializa MPI y se obtiene el identificador del proceso y el número total de procesos. Se definen los valores para *n_min, n_max y n_factor* que se utilizarán para determinar el rango de valores que se considerarán para contar el número de primos.

```
printf("\n");
printf(" Programa PARALELO para contar el n mero de primos menores que un valor.\n");
printf("\n");

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

n_min = 500;
n_max = 50000000;
n_factor = 10;
```

La ejecución del programa comienza con la asignación de n al valor mínimo n_min. Luego, se inicia un bucle while que continuará hasta que n alcance el valor máximo n_max.

Dentro del bucle, se verifica si rank es igual a 0. Si es así, se mide el tiempo de ejecución secuencial del programa para contar el número de primos entre 2 y n. Esto se hace utilizando la función MPI_Wtime() que devuelve el tiempo actual en segundos. Luego, se realiza un bucle que itera a través de todos los números entre 2 y n, y para cada número se llama a la función esPrimo() que determina si es primo o no. Si es primo, se incrementa el contador primos. Una vez que se han contado todos los primos para este rango de valores de n, se mide el tiempo de ejecución total del proceso secuencial y se almacena en la variable tiempo_secuencial.

```
n = n_min;
while (n <= n_max)
{
```

```
    // Tiempo de ejecuci n secuencial
    if (rank == 0)
    {
        t0 = MPI_Wtime();
        primos = 0;
        for (int i = 2; i <= n; i++)
        {
            if (esPrimo(i) == 1)
                primos++;
        }
        t1 = MPI_Wtime();
        tiempo_secuencial = t1 - t
```

Luego, se utiliza MPI_Barrier() para sincronizar todos los procesos antes de iniciar la ejecución paralela. Esto asegura que todos los procesos comiencen al mismo tiempo. Después, se mide el tiempo de ejecución para contar el número de primos para el rango de valores de n asignado a cada proceso utilizando la función numero_primos_par(). La función recibe el valor de n, el identificador del proceso rank y el número total de procesos num_procs. Cada proceso cuenta los primos para un rango de valores de n que es igual a rank + num_procs * k, donde k es un número entero que se incrementa en cada iteración del bucle. Esto asegura que cada proceso examine diferentes valores de n.

Después de contar los primos para el rango de valores de n asignado a cada proceso, se utiliza la función MPI_Reduce() para sumar todos los valores de primos_parte y obtener el total de primos en todos los procesos. Este valor se almacena en primos_total.

Si rank es igual a 0, se realiza la impresión de los resultados del programa. Se mide el tiempo parcial de ejecución de cada proceso utilizando MPI_Wtime() y se almacena en el arreglo tiempos_parciales. Se utiliza la función MPI_Recv() para recibir los tiempos parciales de ejecución de cada proceso y almacenarlos en el arreglo. Luego, se mide el tiempo de ejecución total del programa paralelo y se almacena en la variable tiempo_parcial_total. Finalmente, se imprimen los resultados, incluyendo el número total de primos encontrados, el tiempo de ejecución secuencial, los tiempos parciales de cada proceso, el tiempo de ejecución total del programa paralelo, el speedup y la eficiencia.

Si rank no es igual a 0, se utiliza la función MPI_Send() para enviar el tiempo parcial de ejecución al proceso 0.

```
MPI_Barrier(MPI_COMM_WORLD);

// Tiempo de ejecuci n paralela
t0 = MPI_Wtime();
primos_parte = numero_primos_par(n, rank, num_procs);
double tiempo_parcial = MPI_Wtime() - t0;

int primos_total;
MPI_Reduce(&primos_parte, &primos_total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0)
{
    double tiempos_parciales[num_procs];
    tiempos_parciales[0] = tiempo_parcial;
    for (int i = 1; i < num_procs; i++)
    {
        MPI_Recv(&tiempos_parciales[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
    }

    t1 = MPI_Wtime();
    tiempo_parcial_total = t1 - t0;

    printf(" Primos menores que %10d: %10d.\n Tiempo secuencial: %5.2f segundos.\n", n,
    primos, tiempo_secuencial);
    printf("Tiempos parciales: ");
    for (int i = 1; i < num_procs; i++)
    {
        printf("%5.2f,", tiempos_parciales[i]);
    }
    printf("\n Tiempo paralelo total: %5.2f segundos. ", tiempo_parcial_total);
```

```
30      printf("\nSpeedup: %5.2f. Eficiencia: %5.2f\n", tiempo_secuencial / tiempo_parcial_total
        , (tiempo_secuencial / tiempo_parcial_total) / num_procs);
31      printf("\n---------------------------------------------------------\n");
32  }
33  else
34  {
35      MPI_Send(&tiempo_parcial, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
36  }
```

Al final del bucle, se multiplica el valor actual de n por n_factor para aumentar su valor en un factor constante. Esto asegura que el programa examine un rango de valores de n cada vez más grande en cada iteración del bucle.

```
1   int numero_primos_par(int n, int rank, int num_procs)
2   {
3       int primo, total = 0;
4       for (int i = 2 + rank; i <= n; i += num_procs)
5       {
6           if (esPrimo(i) == 1)
7               total++;
8       }
9       return total;
10  }
11
12  int esPrimo(int p)
13  {
14      for (int i = 2; i <= sqrt(p); i++)
15      {
16          if (p % i == 0)
17              return 0;
18      }
19      return 1;
20  }
```

La función esPrimo() determina si un número p es primo o no. Se realiza un bucle que itera desde 2 hasta la raíz cuadrada de p, y para cada valor de i se verifica si p es divisible por i. Si p es divisible por algún valor de i, entonces no es primo y la función devuelve 0. De lo contrario, la función devuelve 1 para indicar que p es primo.

La función numero_primos_par() cuenta el número de primos menores o iguales a n para el rango de valores de n asignado a cada proceso en la ejecución paralela. Se inicializa la variable total en 0 y se realiza un bucle que itera desde 2 + rank hasta n en incrementos de num_procs. Para cada valor de i, se llama a la función esPrimo() para determinar si es primo o no. Si es primo, se incrementa la variable total. Al final del bucle, la función devuelve el valor de total que indica el número total de primos para el rango de valores de n asignado a ese proceso en particular.

3. ¿Por qué los tiempos de algunos procesos son bastante inferiores a los del resto?

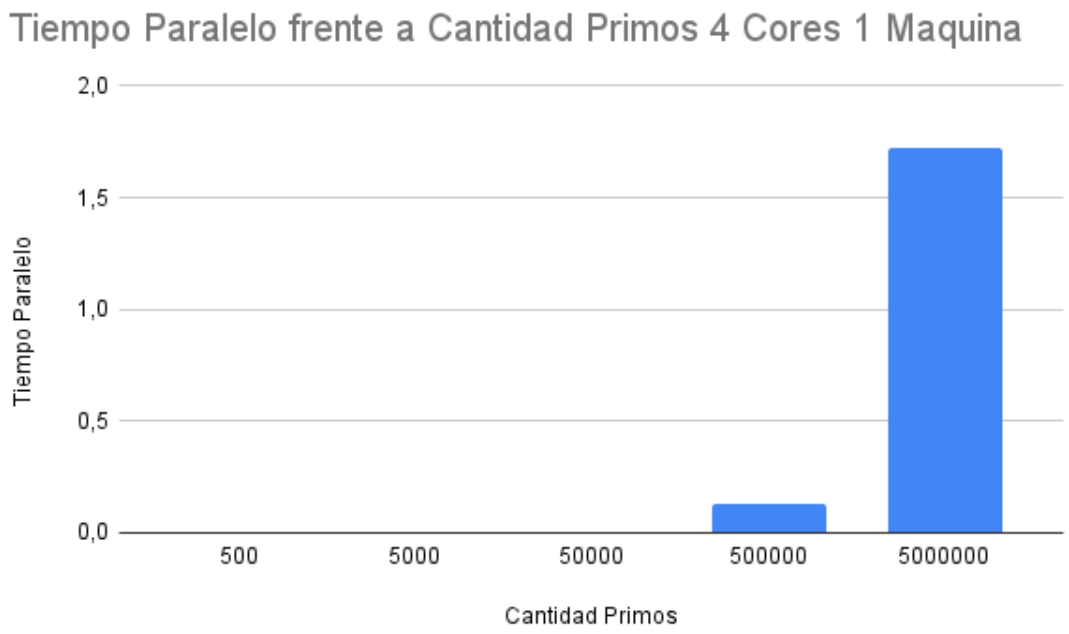
La razón por la que algunos procesos tardan mucho menos que otros al realizar el algoritmo de manera paralela puede ser debido a varios factores:

- El tamaño del subrango asignado a cada proceso: si el tamaño del subrango asignado a un proceso es muy pequeño, es posible que el tiempo que tarda en ejecutar la función "numero_primos_par"sea insignificante en comparación con el tiempo necesario para comunicar los resultados parciales a través de la red MPI. Por otro lado, si el tamaño del subrango es muy grande, puede llevar más tiempo para que el proceso complete el cómputo.
- El desequilibrio en la distribución de números primos en el rango de valores: algunos subrangos pueden tener más números primos que otros, lo que puede resultar en una carga desequilibrada entre los procesos. Por lo tanto, algunos procesos pueden terminar mucho antes que otros.

- La variabilidad en la velocidad de la CPU o en la carga del sistema: los procesos pueden ejecutarse en diferentes nodos o máquinas en un clúster, y puede haber una variabilidad en la velocidad de la CPU o en la carga del sistema en estos nodos, lo que puede afectar el tiempo que tardan los procesos en completar el cómputo.

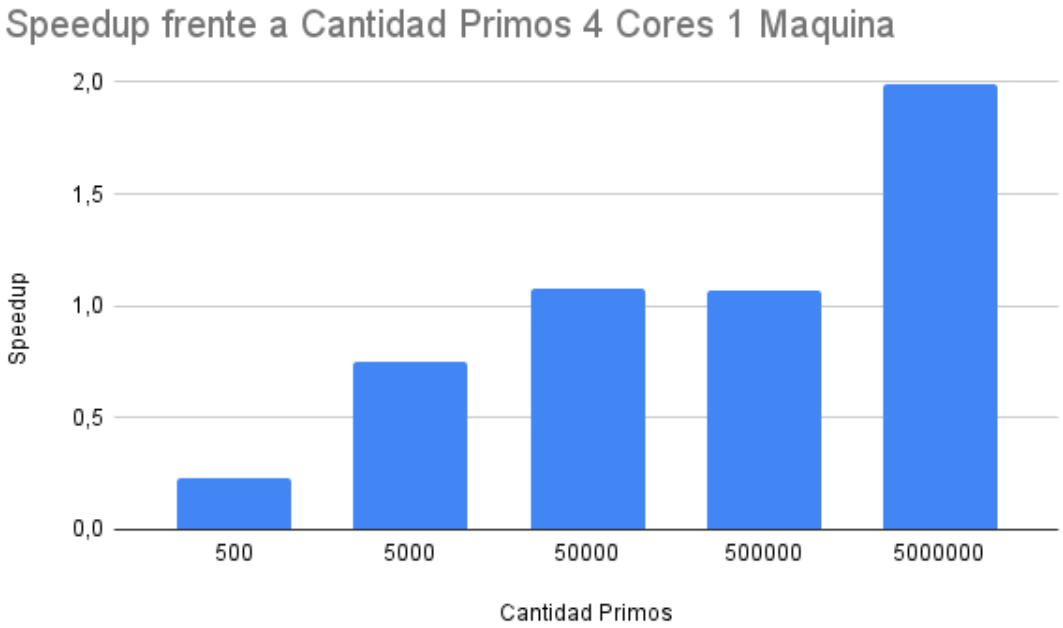
4. Gráficas comentadas

4.1. Gráficas 4 con 1 Maquina



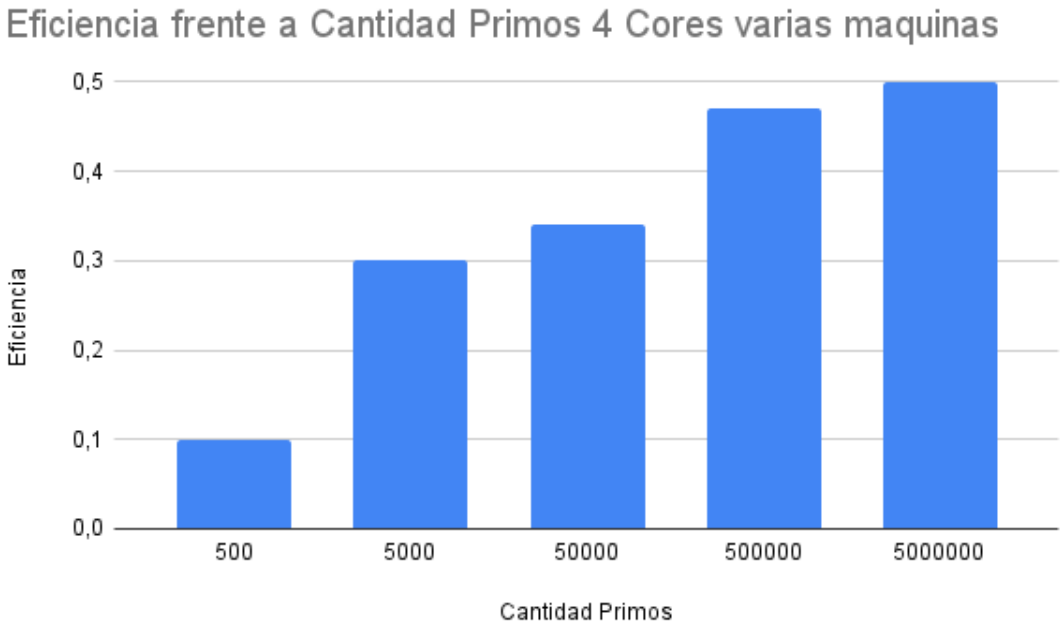
0.45

Figura 1: Tiempo Paralelo frente a Cantidad Primos 4 Cores 1 Maquina



0.45

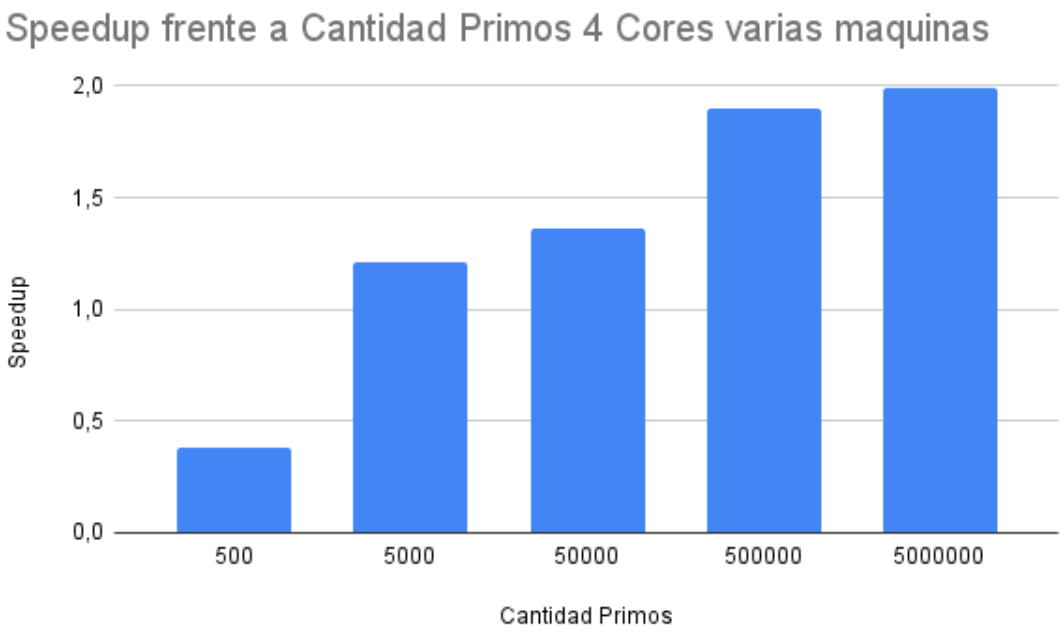
Figura 2: Speedup frente a Cantidad Primos 4 Cores 1 Maquina



0.45

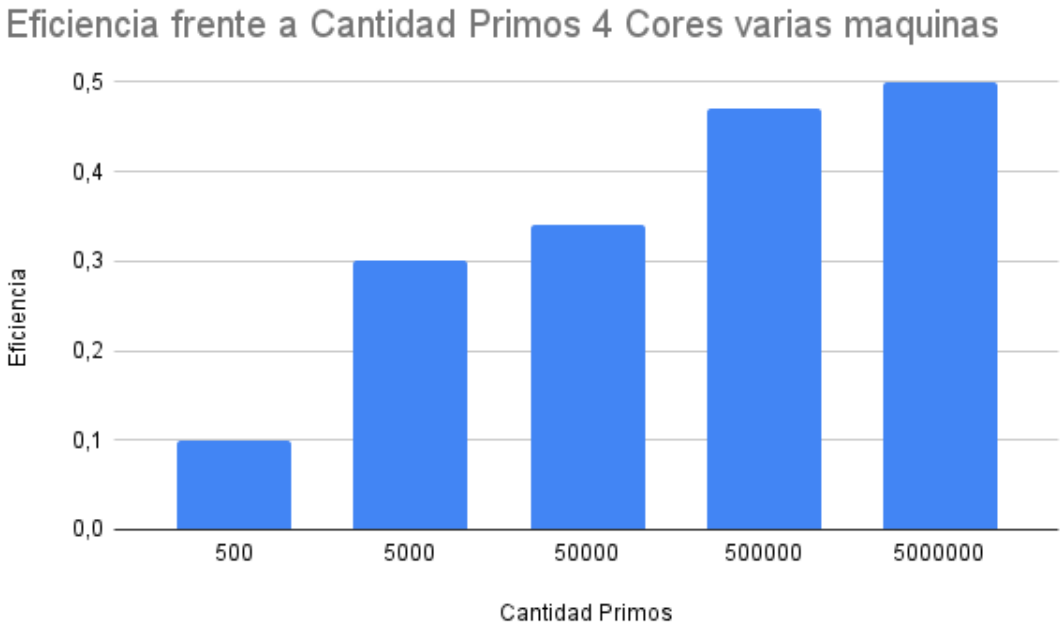
Figura 3: Eficiencia frente a Cantidad Primos 4 Cores varias maquinas

4.2. Gráficas 4 varias Maquinas



0.45

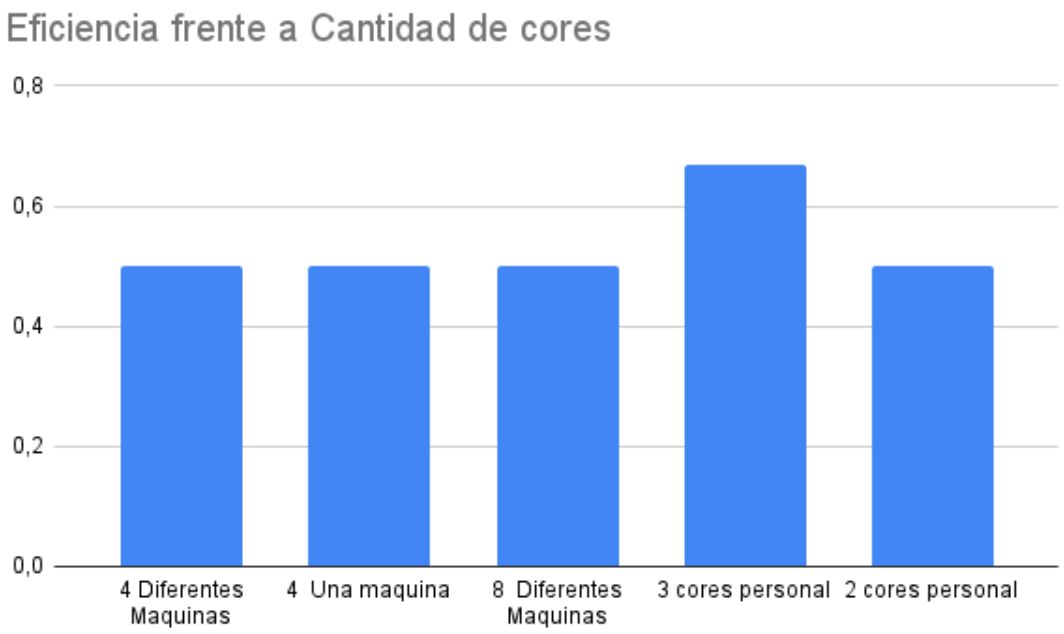
Figura 4: Speedup frente a Cantidad Primos 4 Cores varias maquinas



0.45

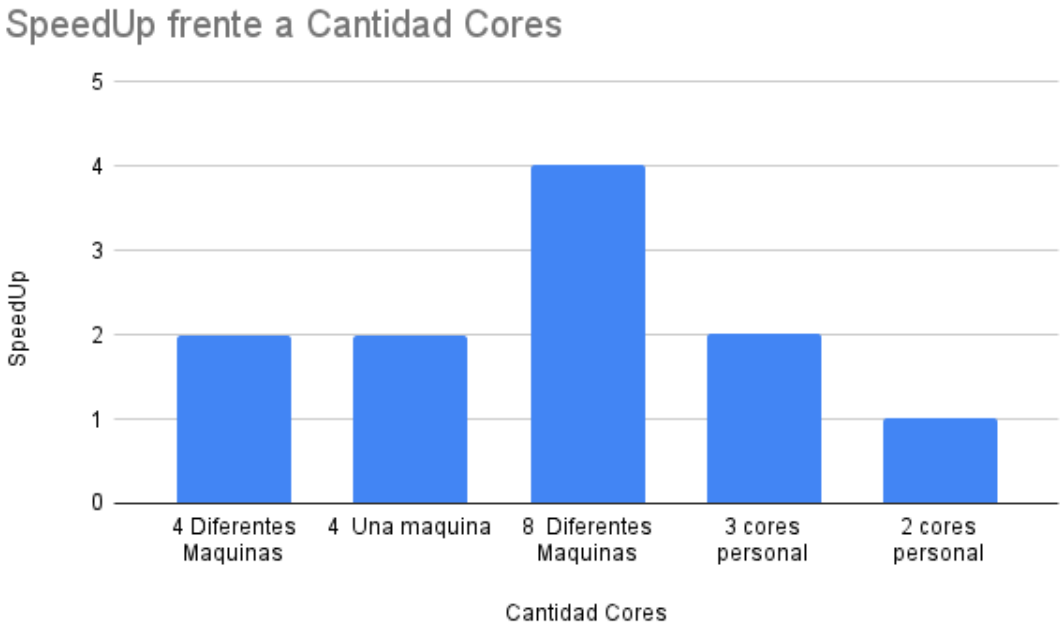
Figura 5: Eficiencia frente a Cantidad Primos 4 Cores varias maquinas

4.3. Comparación



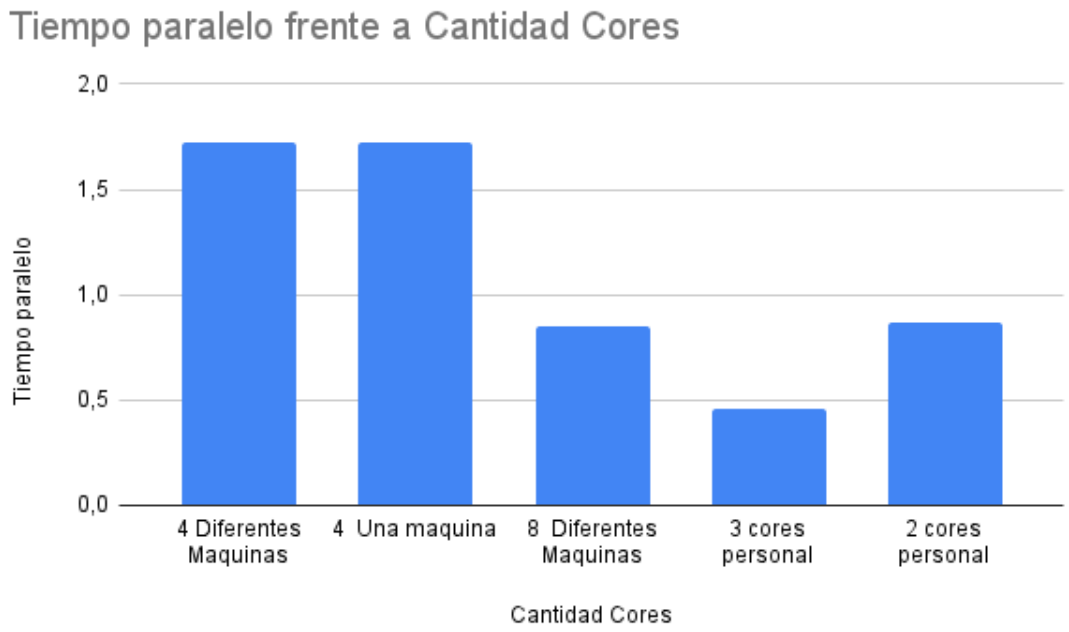
0.45

Figura 6: Eficiencia frente a Cantidad de cores



0.45

Figura 7: SpeedUp frente a Cantidad Cores



0.45

Figura 8: Tiempo paralelo frente a Cantidad Cores

5. ¿Por qué el speed-up y la eficiencia no se acercan, en especial en algunos casos, al speed-up y eficiencia perfectos?

El speed-up y la eficiencia son medidas importantes para evaluar el rendimiento de un programa paralelo en comparación con su versión secuencial. El speed-up mide la mejora en el tiempo de ejecución que se obtiene al utilizar múltiples procesos en comparación con el tiempo de ejecución de una sola tarea secuencial. La eficiencia mide cuánto de la capacidad de procesamiento de los procesadores se está utilizando.

El speed-up perfecto se logra cuando el tiempo de ejecución se reduce en la misma proporción que el número de procesos utilizados, lo que significa que el programa es perfectamente paralelizable. Por ejemplo, si el tiempo de ejecución de un programa secuencial es de 10 segundos y el tiempo de ejecución de un programa paralelo con 2 procesos es de 5 segundos, entonces el speed-up es de 2 (10/5) y la eficiencia es de 100

Sin embargo, en la práctica, la mayoría de los algoritmos no son perfectamente paralelizables y pueden tener limitaciones que disminuyen el speed-up y la eficiencia. Algunos de estos factores son la comunicación, la sobrecarga de memoria, los problemas de sincronización y el balance de carga.

Por ejemplo, la comunicación entre procesos puede ser costosa en términos de tiempo, especialmente cuando se manejan grandes cantidades de datos. Además, los cuellos de botella en la red o la limitación en el ancho de banda pueden disminuir el speed-up y la eficiencia. La sobrecarga de memoria puede aumentar el tiempo de ejecución, especialmente en sistemas con múltiples procesadores. Los problemas de sincronización pueden reducir la eficiencia si los procesos tienen que esperar a que otros procesos completen su trabajo antes de continuar. Finalmente, una distribución desigual de la carga de trabajo entre los procesos puede reducir el speed-up y la eficiencia.

5.1. Solución posible

Para mejorar el speed-up y la eficiencia de un programa paralelo, se pueden aplicar varias técnicas para abordar los factores que pueden limitar el rendimiento, como los siguientes:

- Reducción de la comunicación: Se pueden aplicar técnicas para minimizar la comunicación entre procesos, como el uso de comunicación colectiva en lugar de comunicación punto a punto o la reducción

del tamaño de los datos a transmitir.

- Optimización de la memoria: Se pueden aplicar técnicas para reducir la sobrecarga de la memoria, como el uso de técnicas de memoria compartida o distribuida.
- Mejora de la sincronización: Se pueden aplicar técnicas para mejorar la sincronización entre procesos, como el uso de técnicas de programación asincrónica o la implementación de mecanismos de comunicación de baja latencia.
- Balanceo de carga: Se pueden aplicar técnicas para equilibrar la carga de trabajo entre los procesos, como la división de la tarea en sub-tareas más pequeñas y la asignación de tareas de manera dinámica en función de la carga de trabajo actual.

Además, también se pueden utilizar herramientas de análisis de rendimiento para identificar cuellos de botella en el programa y optimizar su rendimiento

6. Código completo

```
    #include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

int numero_primos_par(int n, int rank, int num_procs);
int esPrimo(int p);

int main(int argc, char *argv[])
{
    // valores para el rango de valores para los cuales se contar  el n mero de primos.
    int n, n_factor, n_min, n_max;
    // variables que contienen el n mero de primos para el rango de valores que se est
    considerando en la ejecuci n.
    int primos, primos_parte;
    // variables que se utilizan para medir el tiempo de ejecuci n del programa.
    double t0, t1, tiempo_secuencial, tiempo_parcial_total;
    // variables que contienen el identificador del proceso y el n mero total de procesos
    en la ejecuci n paralela, respectivamente.
    int rank, num_procs;

    printf("\n");
    printf(" Programa PARALELO para contar el n mero de primos menores que un valor.\n");
    printf("\n");

    /**
     * inicializa MPI y se obtiene el identificador del proceso y el n mero total de
    procesos.
     */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    n_min = 500;
    n_max = 50000000;
    n_factor = 10;

    n = n_min;
    while (n <= n_max)
    {
        // Tiempo de ejecuci n secuencial
        if (rank == 0)
        {
            t0 = MPI_Wtime();
            primos = 0;
```

```
44              for (int i = 2; i <= n; i++)
45              {
46                  if (esPrimo(i) == 1)
47                      primos++;
48              }
49              t1 = MPI_Wtime();
50              tiempo_secuencial = t1 - t0;
51          }
52
53          /**
54           * Se utiliza la funci n "MPI_Barrier" para sincronizar todos los procesos antes de
     comenzar el tiempo de ejecuci n paralela.
55           */
56          MPI_Barrier(MPI_COMM_WORLD);
57
58          // Tiempo de ejecuci n paralela
59          t0 = MPI_Wtime();
60          primos_parte = numero_primos_par(n, rank, num_procs);
61          double tiempo_parcial = MPI_Wtime() - t0;
62
63          int primos_total;
64          MPI_Reduce(&primos_parte, &primos_total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
65
66          if (rank == 0)
67          {
68              double tiempos_parciales[num_procs];
69              tiempos_parciales[0] = tiempo_parcial;
70              for (int i = 1; i < num_procs; i++)
71              {
72                  MPI_Recv(&tiempos_parciales[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
73              }
74
75              t1 = MPI_Wtime();
76              tiempo_parcial_total = t1 - t0;
77
78              printf(" Primos menores que %10d: %10d.\n Tiempo secuencial: %5.2f segundos.\n",
     n, primos, tiempo_secuencial);
79              printf("Tiempos parciales: ");
80              for (int i = 0; i < num_procs; i++)
81              {
82                  printf(" %5.2f,", tiempos_parciales[i]);
83              }
84              printf("\n Tiempo paralelo total: %5.2f segundos. ", tiempo_parcial_total);
85              printf("\nSpeedup: %5.2f. Eficiencia: %5.2f\n", tiempo_secuencial /
    tiempo_parcial_total, (tiempo_secuencial / tiempo_parcial_total) / num_procs);
86              printf("\n-------------------------------------------------------\n");
87          }
88          else
89          {
90              MPI_Send(&tiempo_parcial, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
91          }
92          n = n * n_factor;
93      }
94      printf("\n");
95      MPI_Finalize();
96      return 0;
97  }
98
99  // Devuelve el numero de primos menores que n
100 int numero_primos_par(int n, int rank, int num_procs)
101 {
102     int primo, total = 0;
103     for (int i = 2 + rank; i <= n; i += num_procs)
104     {
105         if (esPrimo(i) == 1)
106             total++;
107     }
```

```
108     return total;
109 }
110
111 int esPrimo(int p)
112 {
113     for (int i = 2; i <= sqrt(p); i++)
114     {
115         if (p % i == 0)
116             return 0;
117     }
118     return 1;
119 }
```