Universidad del Valle de Guatemala
Facultad de Ingeniería
Departamento de Ciencias de la Computación
CC3069 Computación Paralela y Distribuida
Catedrático: Miguel Novella
Ciclo 1 de 2023

Laboratorio 3

Bryann Alfaro 19372
Brandon Hernández 19376

Guatemala, Ciudad de Guatemala 19 de abril de 2023

(10 pts) Explique por qué y cómo usamos comunicación grupal en las siguientes funciones de mpi_vector_add.c:

i. Check_for_error()

En este método se usa comunicación grupal, ya que debido a su propósito, debe verificar en algún proceso si existe algún error y esto detiene toda la ejecución. Por lo tanto, debe avisar a los demás procesos sobre esta detención en el flujo.
La forma en que se usa comunicación grupal es por medio de la llamada a MPI_Allreduce en donde con la variable ok se define si el flujo se detiene (ok == 0) o no.

ii. Read_n()

En este método usamos comunicación grupal, ya que los procesos necesitan saber el orden en que ingresa el vector desde el proceso 0 para poder realizar los cálculos correspondientes.
La forma en que esto se lleva a cabo es, luego de recibir el orden en el proceso 0 , se realiza una llamada a MPI_Bcast en donde a los demás procesos en el comunicador se les manda ese orden obtenido.

iii. Read_data()

En este método se utiliza la comunicación grupal, ya que los procesos necesitan la información a la que le aplicarán las operaciones, por lo que el proceso con rank 0 envía los vectores correspondientes.
La forma en la que se lleva a cabo, es por medio de la función MPI_Scatter, la cual envía al resto de procesos el vector que deberán operar en partes iguales y de forma ordenada.

iv. Print_vector()

En este método se utiliza la comunicación grupal, para poder mostrar en pantalla el vector con el que cuenta cada uno de los procesos, esta función es más conveniente cuando los vectores ya fueron modificados.
La forma en que se lleva a cabo es que el proceso 0 obtiene todos los vectores del resto de procesos y los concatena con base a su rank, para luego mostrar en pantalla los valores que contiene el vector.

(15 pts) Descargue y modifique el programa vector_add.c para crear dos vectores de al menos 100,000 elementos generados de forma aleatoria. Haga lo mismo con mpi_vector_add.c. Imprima únicamente los primeros y últimos 10 elementos de cada vector (y el resultado) para validar. Incluya captura de pantalla.

vector_add.c

```c
void Generate_Vector_Random(
      double  a[]         /* out */,
      int     n           /* in  */,
      char    vec_name[]  /* in  */) {
   int i;
   printf("\nCreating vector %s ..\n", vec_name);
   for (i = 0; i < n; i++)
      a[i] = rand() % 200;

}  /* Generate_Vector_Random */
```

```c
void Print_vector(
      double  b[]     /* in */,
      int     n       /* in */,
      char    title[] /* in */) {
   int i;
   printf("%s\n", title);
   for (i = 0; i < 10; i++)
      printf("%f ", b[i]);
   //the last 10 elements
   printf("\nLast 10 elements\n");
   for (i = n-10; i < n; i++)
      printf("%f ", b[i]);
   printf("\n");
}  /* Print_vector */
```

```
PS C:\Users\Bryann\Desktop\9no semestre\paralela\CPD_Lab03> .\ve
ctor_add2.exe
What's the order of the vectors?
100000

Creating vector x ..

The first vector is
62.000000 161.000000 151.000000 180.000000 98.000000 189.000000
149.000000 183.000000 66.000000 107.000000
Last 10 elements
19.000000 105.000000 6.000000 107.000000 83.000000 57.000000 13.
000000 1.000000 105.000000 158.000000

Creating vector y ..

The second vector is
165.000000 140.000000 196.000000 120.000000 176.000000 83.000000
 57.000000 6.000000 31.000000 81.000000
Last 10 elements
127.000000 40.000000 1.000000 23.000000 18.000000 30.000000 157.
000000 136.000000 83.000000 139.000000

The sum is
227.000000 301.000000 347.000000 300.000000 274.000000 272.00000
0 206.000000 189.000000 97.000000 188.000000
Last 10 elements
146.000000 145.000000 7.000000 130.000000 101.000000 87.000000 1
70.000000 137.000000 188.000000 297.000000
PS C:\Users\Bryann\Desktop\9no semestre\paralela\CPD_Lab03>
```

mpi_vector_add.c

```c
void Generate_Vector_Random(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;
   int local_ok = 1;
   char* fname = "Generate_Vector_Random";

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      if (a == NULL) local_ok = 0;
      Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
      //printf("Enter the vector %s\n", vec_name);
      //fill vec with indez

      for (i = 0; i < n; i++)

         a[i] = rand() % 100;

      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
         comm);
      free(a);
   } else {
      Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
         comm);
   }
}  /* Generate_Vector_Random */
```

```c
void Print_vector(
      double    local_b[]  /* in */,
      int       local_n    /* in */,
      int       n          /* in */,
      char      title[]    /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {

   double* b = NULL;
   int i;
   int local_ok = 1;
   char* fname = "Print_vector";

   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      if (b == NULL) local_ok = 0;
      Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < 10; i++)
         printf("%f ", b[i]);
      printf("\n");
      printf("Last 10 elements :\n");
      for (i = n-10; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0,
         comm);
   }
}  /* Print_vector */
```

```
PS C:\Users\Bryann\Desktop\9no semestre\paralela\CPD_Lab03> mpiexec -n 4  .\mpi_vector_add2.exe
x is
49.000000 0.000000 88.000000 45.000000 74.000000 90.000000 24.000000 96.000000 5.000000 50.000000
Last 10 elements :
49.000000 56.000000 2.000000 47.000000 47.000000 61.000000 93.000000 9.000000 92.000000 67.000000
y is
31.000000 98.000000 68.000000 28.000000 92.000000 50.000000 36.000000 22.000000 75.000000 6.000000
Last 10 elements :
31.000000 40.000000 80.000000 70.000000 63.000000 15.000000 73.000000 73.000000 33.000000 53.000000
The sum is
80.000000 98.000000 156.000000 73.000000 166.000000 140.000000 60.000000 118.000000 80.000000 56.000000
Last 10 elements :
80.000000 96.000000 82.000000 117.000000 110.000000 76.000000 166.000000 82.000000 125.000000 120.000000

Took 12.845800 ms to run
PS C:\Users\Bryann\Desktop\9no semestre\paralela\CPD_Lab03>
```

(5 pts) Mida los tiempos de ambos programas y calcule el speedup logrado con la versión paralela. Realice al menos 10 mediciones de tiempo para cada programa y obtenga el promedio del tiempo de cada uno. Cada medición debe estar en el orden de los ~5 segundos para asegurar valores estables (utilice una cantidad de elementos adecuada para que a su máquina le tome por lo menos ~5 cada corrida). Utilice estos promedios para el cálculo del speedup. Incluya capturas de pantalla.

Luego de experimentar con el largo de los vectores, se concluyó que el mejor valor para probar es 100 000 000, debido a que en la máquina que se estarán haciendo las pruebas se tarda ~5 segundos y al agregar más esta se traba y no da el resultado.

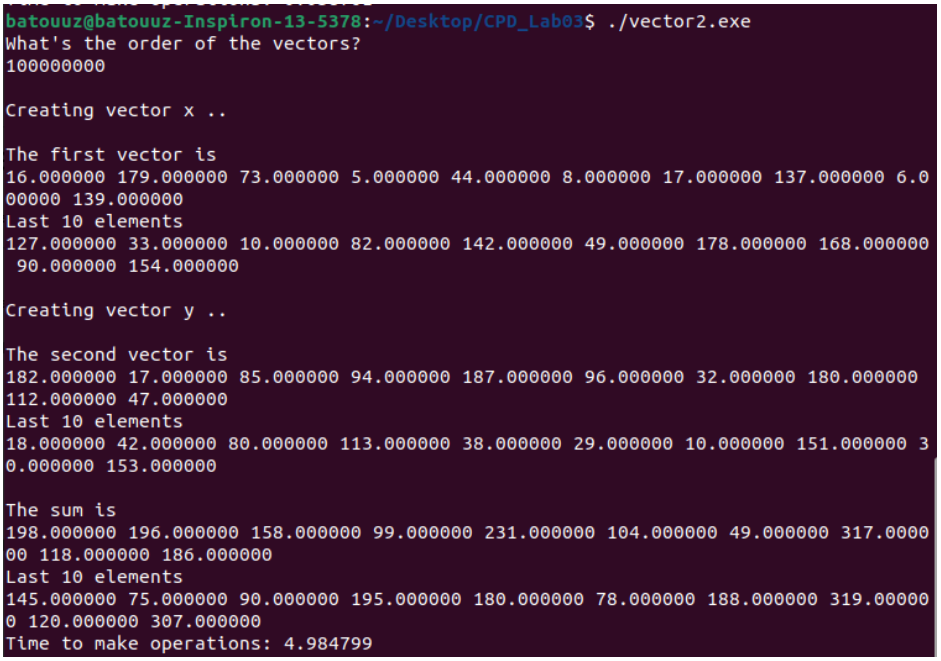


Tabla 1: Corridas con prints

# Corrida	Secuencial (s)	Paralelo (s)
1	5.107437	8.763100
2	5.085879	9.775500
3	5.104335	8.780503
4	5.313347	9.059645
5	5.023571	10.058553
6	5.044823	8.519407
7	5.448767	8.767769
8	5.038069	8.566072
9	5.045688	8.811874
10	5.013444	9.390607
Promedio	**5.122536**	**9.049303**

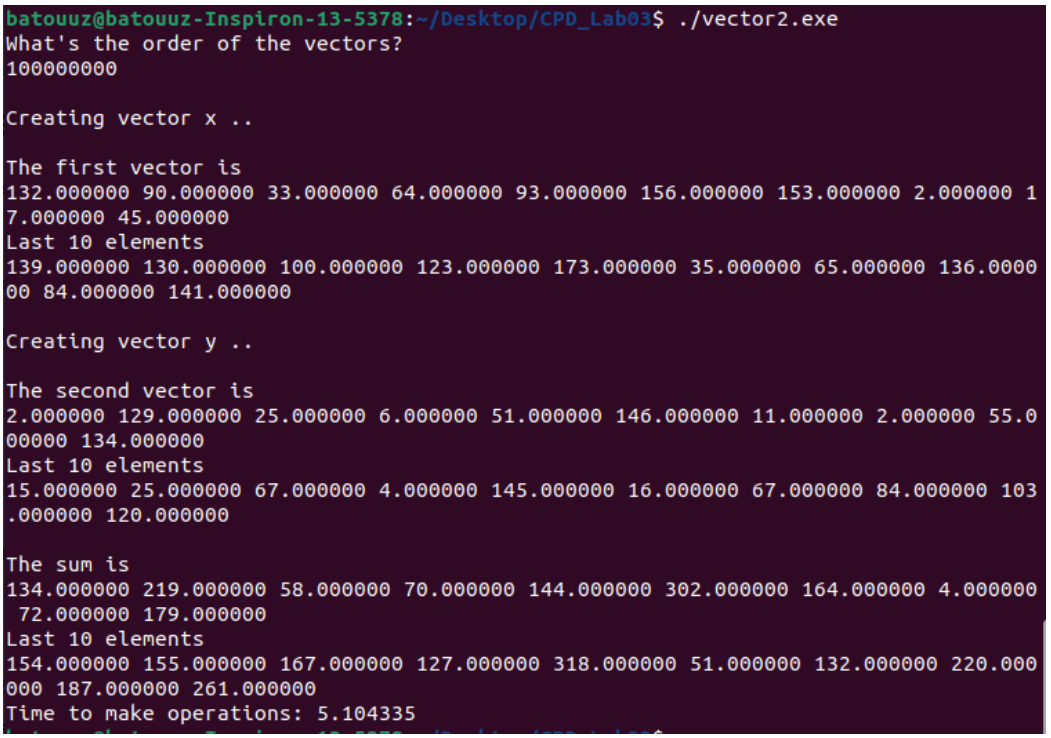
Con base a estos resultados se puede calcular el speedup y la eficiencia:

speedup = t_s / t_p = 5.122536 / 9.049303 = 0.566069

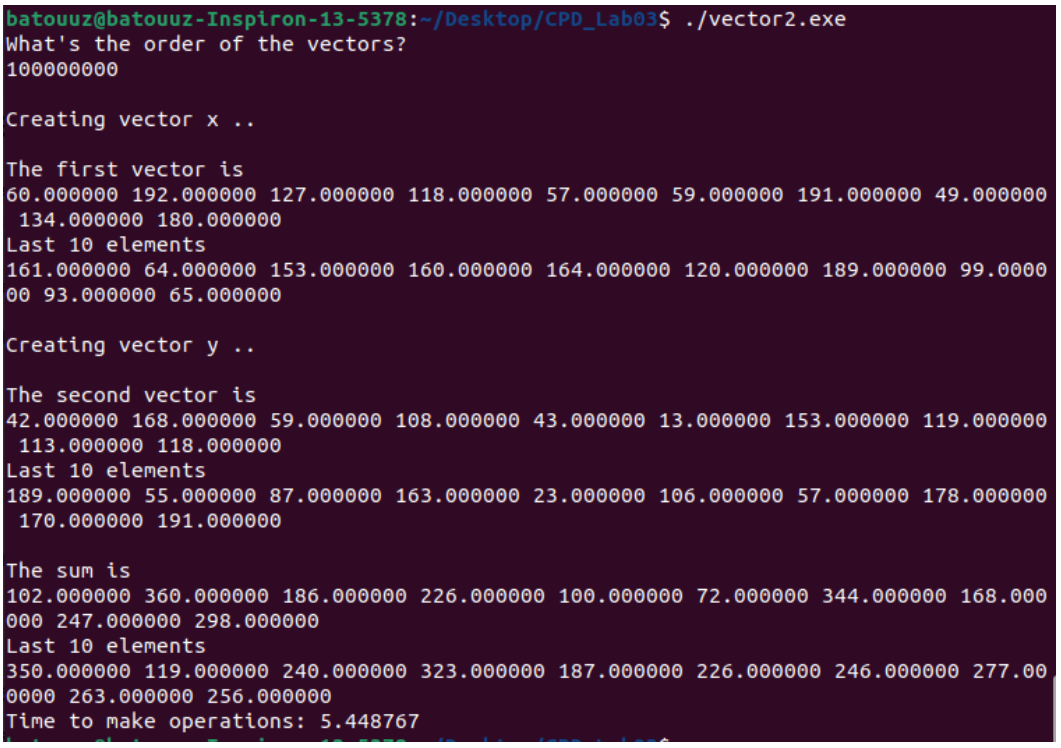
eficiencia = speedup / cores = 0.566069 / 4 = 0.141517

Algunas capturas de las corridas:

- Secuencial



```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ ./vector2.exe
What's the order of the vectors?
100000000

Creating vector x ..

The first vector is
132.000000 90.000000 33.000000 64.000000 93.000000 156.000000 153.000000 2.000000 1
7.000000 45.000000
Last 10 elements
139.000000 130.000000 100.000000 123.000000 173.000000 35.000000 65.000000 136.0000
00 84.000000 141.000000

Creating vector y ..

The second vector is
2.000000 129.000000 25.000000 6.000000 51.000000 146.000000 11.000000 2.000000 55.0
00000 134.000000
Last 10 elements
15.000000 25.000000 67.000000 4.000000 145.000000 16.000000 67.000000 84.000000 103
.000000 120.000000

The sum is
134.000000 219.000000 58.000000 70.000000 144.000000 302.000000 164.000000 4.000000
 72.000000 179.000000
Last 10 elements
154.000000 155.000000 167.000000 127.000000 318.000000 51.000000 132.000000 220.000
000 187.000000 261.000000
Time to make operations: 5.104335
```



```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ ./vector2.exe
What's the order of the vectors?
100000000

Creating vector x ..

The first vector is
60.000000 192.000000 127.000000 118.000000 57.000000 59.000000 191.000000 49.000000
 134.000000 180.000000
Last 10 elements
161.000000 64.000000 153.000000 160.000000 164.000000 120.000000 189.000000 99.0000
00 93.000000 65.000000

Creating vector y ..

The second vector is
42.000000 168.000000 59.000000 108.000000 43.000000 13.000000 153.000000 119.000000
 113.000000 118.000000
Last 10 elements
189.000000 55.000000 87.000000 163.000000 23.000000 106.000000 57.000000 178.000000
 170.000000 191.000000

The sum is
102.000000 360.000000 186.000000 226.000000 100.000000 72.000000 344.000000 168.000
000 247.000000 298.000000
Last 10 elements
350.000000 119.000000 240.000000 323.000000 187.000000 226.000000 246.000000 277.00
0000 263.000000 256.000000
Time to make operations: 5.448767
```

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ ./vector2.exe
What's the order of the vectors?
100000000

Creating vector x ..

The first vector is
47.000000 120.000000 10.000000 167.000000 32.000000 144.000000 87.000000 151.000000
 188.000000 92.000000
Last 10 elements
94.000000 121.000000 125.000000 140.000000 142.000000 192.000000 43.000000 89.00000
0 8.000000 17.000000

Creating vector y ..

The second vector is
161.000000 29.000000 196.000000 85.000000 93.000000 2.000000 64.000000 163.000000 6
8.000000 72.000000
Last 10 elements
112.000000 149.000000 197.000000 102.000000 167.000000 69.000000 109.000000 168.000
000 176.000000 162.000000

The sum is
208.000000 149.000000 206.000000 252.000000 125.000000 146.000000 151.000000 314.00
0000 256.000000 164.000000
Last 10 elements
206.000000 270.000000 322.000000 242.000000 309.000000 261.000000 152.000000 257.00
0000 184.000000 179.000000
Time to make operations: 5.013444
```

- Paralelo

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --
oversubscribe -np 4 ./mpi_add.exe
x is
9.000000 50.000000 25.000000 20.000000 16.000000 66.000000 11.000000 98.000000 6.00
0000 90.000000
Last 10 elements :
8.000000 48.000000 93.000000 45.000000 93.000000 38.000000 55.000000 5.000000 9.000
000 58.000000
y is
49.000000 13.000000 77.000000 56.000000 1.000000 51.000000 41.000000 75.000000 75.0
00000 45.000000
Last 10 elements :
81.000000 4.000000 54.000000 57.000000 99.000000 81.000000 94.000000 40.000000 21.0
00000 3.000000
The sum is
58.000000 63.000000 102.000000 76.000000 17.000000 117.000000 52.000000 173.000000
81.000000 135.000000
Last 10 elements :
89.000000 52.000000 147.000000 102.000000 192.000000 119.000000 149.000000 45.00000
0 30.000000 61.000000

Took 8.763100 s to run
```

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --
oversubscribe -np 4 ./mpi_add.exe
x is
83.000000 11.000000 36.000000 74.000000 28.000000 59.000000 54.000000 4.000000 76.0
00000 5.000000
Last 10 elements :
19.000000 84.000000 76.000000 27.000000 88.000000 54.000000 42.000000 42.000000 0.0
00000 97.000000
y is
94.000000 77.000000 53.000000 72.000000 76.000000 4.000000 20.000000 55.000000 32.0
00000 18.000000
Last 10 elements :
45.000000 53.000000 67.000000 10.000000 79.000000 64.000000 5.000000 18.000000 33.0
00000 78.000000
The sum is
177.000000 88.000000 89.000000 146.000000 104.000000 63.000000 74.000000 59.000000
108.000000 23.000000
Last 10 elements :
64.000000 137.000000 143.000000 37.000000 167.000000 118.000000 47.000000 60.000000
 33.000000 175.000000

Took 8.767769 s to run
```

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --
oversubscribe -np 4 ./mpi_add.exe
x is
78.000000 75.000000 7.000000 94.000000 73.000000 95.000000 94.000000 58.000000 19.0
00000 82.000000
Last 10 elements :
95.000000 62.000000 5.000000 59.000000 26.000000 90.000000 34.000000 48.000000 59.0
00000 14.000000
y is
52.000000 8.000000 93.000000 84.000000 89.000000 31.000000 19.000000 37.000000 82.0
00000 50.000000
Last 10 elements :
1.000000 33.000000 22.000000 73.000000 73.000000 73.000000 84.000000 80.000000 9.00
0000 53.000000
The sum is
130.000000 83.000000 100.000000 178.000000 162.000000 126.000000 113.000000 95.0000
00 101.000000 132.000000
Last 10 elements :
96.000000 95.000000 27.000000 132.000000 99.000000 163.000000 118.000000 128.000000
 68.000000 67.000000

Took 9.390607 s to run
```

Tabla 2: Corridas sin prints

# Corrida	Secuencial (s)	Paralelo (s)
1	5.187674	7.484320
2	5.470068	7.392342
3	5.368541	8.297977
4	5.199810	8.038713
5	5.052345	7.976801
6	5.170718	8.149629
7	5.111353	8.176504
8	5.051015	8.121117
9	5.170225	7.405854
10	5.384953	8.127536
Promedio	**5.216670**	**7.917079**

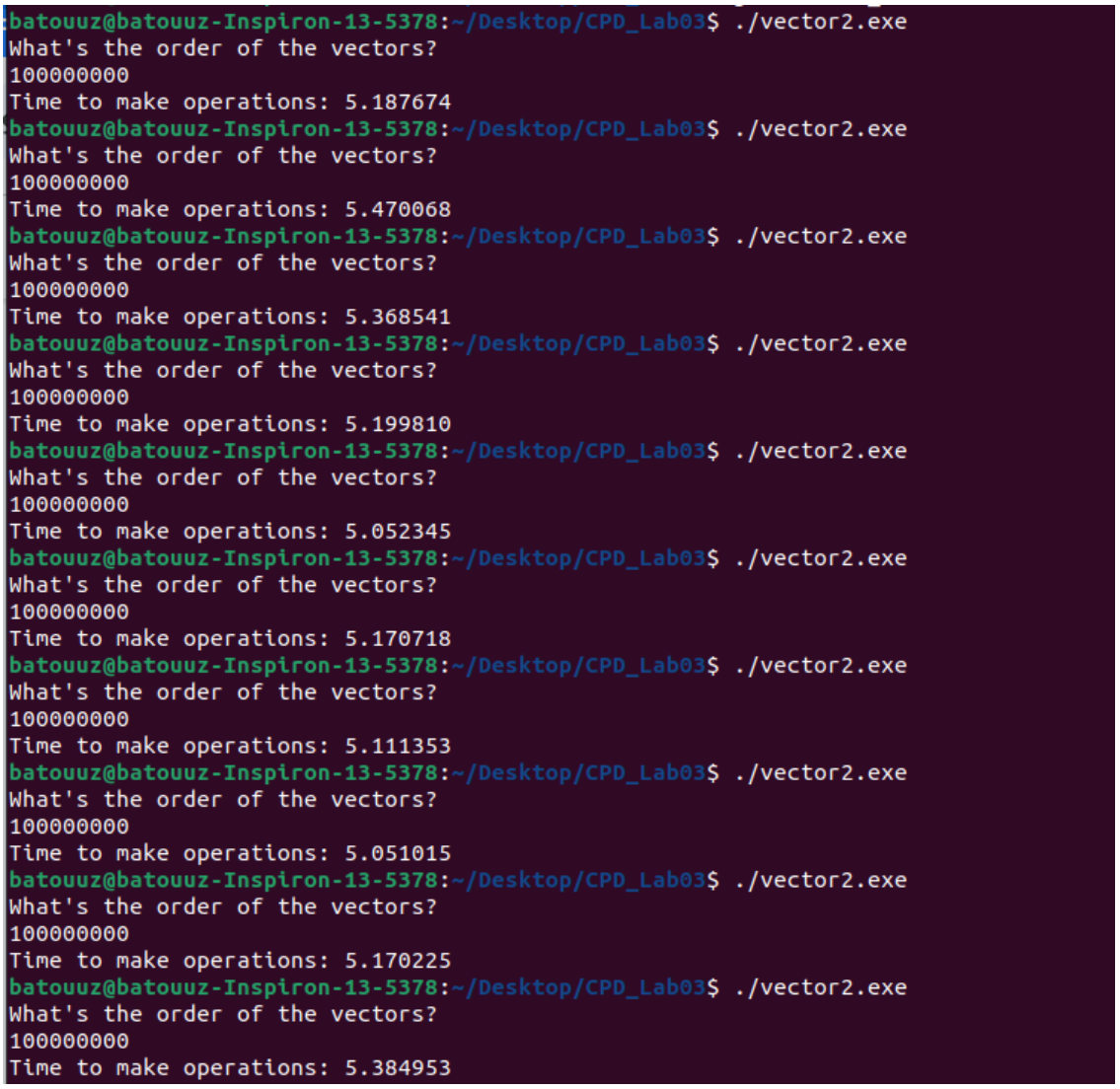
Con base a estos resultados se puede calcular el speedup y la eficiencia:

speedup = t_s / t_p = 5.216670 / 7.917079 = 0.6589134705

eficiencia = speedup / cores = 0.6589134705 / 4 = 0.1647283676

Algunas capturas de las corridas:

- Secuencial



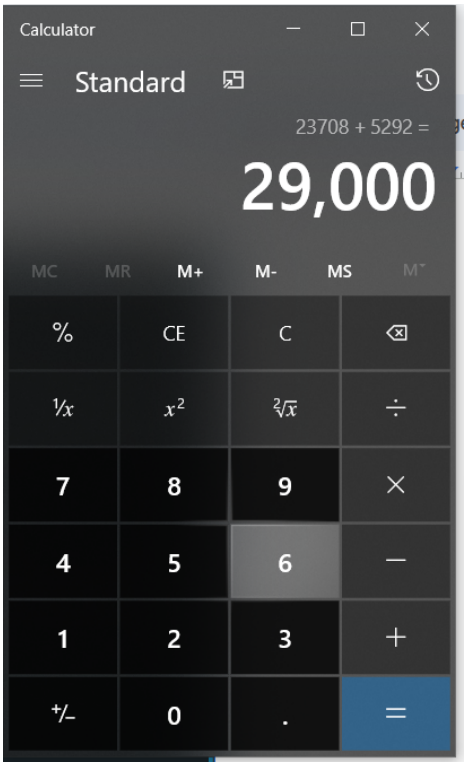
- Paralelo

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 7.484320 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 7.392342 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 8.297977 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 8.038713 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 7.976801 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 8.149629 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 8.176504 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 8.121117 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 7.405854 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus --oversubscribe -np 4 ./mpi_add.exe

Took 8.127536 s to run
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$
```

(55 pts) Modifique el programa mpi_vector_add.c para que calcule de dos vectores 1) el producto punto 2) el producto de un escalar por cada vector (el mismo escalar para ambos). Verifique el correcto funcionamiento de su programa (para ello puede probar con pocos elementos para validar). Incluya captura de pantalla.

Producto Punto

```
PS C:\Users\Bryann\Desktop\9no semestre\paralela\CPD_Lab03> mpiexec -n 5  .\mpi_vector_addOps.exe
x is
38.000000 76.000000 97.000000 52.000000 45.000000 0.000000 43.000000 89.000000 75.000000 84.000000
Last 10 elements :
38.000000 76.000000 97.000000 52.000000 45.000000 0.000000 43.000000 89.000000 75.000000 84.000000
y is
41.000000 50.000000 34.000000 61.000000 10.000000 76.000000 32.000000 86.000000 32.000000 63.000000
Last 10 elements :
41.000000 50.000000 34.000000 61.000000 10.000000 76.000000 32.000000 86.000000 32.000000 63.000000
The sum is
79.000000 126.000000 131.000000 113.000000 55.000000 76.000000 75.000000 175.000000 107.000000 147.000000
Last 10 elements :
79.000000 126.000000 131.000000 113.000000 55.000000 76.000000 75.000000 175.000000 107.000000 147.000000

Took 8.109200 ms to run
Dot product is 29000.000000
PS C:\Users\Bryann\Desktop\9no semestre\paralela\CPD_Lab03>
```



Producto Escalar

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus
 --oversubscribe -np 5 ./mpi_vope.exe
x = [26.000000 5.000000 57.000000 28.000000 91.000000 70.000000 61.000000 24.000
000 30.000000 79.000000 ... 26.000000 5.000000 57.000000 28.000000 91.000000 70.
000000 61.000000 24.000000 30.000000 79.000000 ]

y = [1.000000 26.000000 48.000000 15.000000 41.000000 40.000000 96.000000 57.000
000 97.000000 21.000000 ... 1.000000 26.000000 48.000000 15.000000 41.000000 40.
000000 96.000000 57.000000 97.000000 21.000000 ]


Constant that will multiply the vector:
2
x = [52.000000 10.000000 114.000000 56.000000 182.000000 140.000000 122.000000 4
8.000000 60.000000 158.000000 ... 52.000000 10.000000 114.000000 56.000000 182.0
00000 140.000000 122.000000 48.000000 60.000000 158.000000 ]

y = [2.000000 52.000000 96.000000 30.000000 82.000000 80.000000 192.000000 114.0
00000 194.000000 42.000000 ... 2.000000 52.000000 96.000000 30.000000 82.000000
80.000000 192.000000 114.000000 194.000000 42.000000 ]


Took 0.000225 ms to run
```

Como se puede observar en la imagen, se multiplicaron los vectores con la constante 2, dando como resultado dos vectores con sus valores siendo el doble que antes.

```
batouuz@batouuz-Inspiron-13-5378:~/Desktop/CPD_Lab03$ mpirun --use-hwthread-cpus
 --oversubscribe -np 5 ./mpi_vope.exe
x = [45.000000 42.000000 61.000000 5.000000 18.000000 26.000000 51.000000 55.000
000 58.000000 2.000000 ... 45.000000 42.000000 61.000000 5.000000 18.000000 26.0
00000 51.000000 55.000000 58.000000 2.000000 ]

y = [46.000000 65.000000 51.000000 54.000000 1.000000 30.000000 69.000000 32.000
000 94.000000 69.000000 ... 46.000000 65.000000 51.000000 54.000000 1.000000 30.
000000 69.000000 32.000000 94.000000 69.000000 ]


Constant that will multiply the vector:
4
x = [180.000000 168.000000 244.000000 20.000000 72.000000 104.000000 204.000000
220.000000 232.000000 8.000000 ... 180.000000 168.000000 244.000000 20.000000 72
.000000 104.000000 204.000000 220.000000 232.000000 8.000000 ]

y = [184.000000 260.000000 204.000000 216.000000 4.000000 120.000000 276.000000
128.000000 376.000000 276.000000 ... 184.000000 260.000000 204.000000 216.000000
 4.000000 120.000000 276.000000 128.000000 376.000000 276.000000 ]


Took 0.000657 ms to run
```

Lo mismo se puede decir de esta captura en donde constante = 4.

(15 pts) Finalmente, escriba una reflexión del laboratorio realizado en donde hable de las técnicas aplicadas, lo que se aprendió y pudo repasar, elementos que le llamaron la atención, ediciones/mejoras que considera que son posibles y cualquier otra cosa relevante que tengan en mente. (No hay mínimo de palabras/párrafos, pero si desarrollan poco o de forma superficial seguramente tendrán nota baja en este inciso)

Durante el desarrollo de este laboratorio se pudo aplicar técnicas de comunicación global, por ejemplo, al momento de realizar la sumatoria de los resultados obtenidos por los procesos en el producto punto por medio de la llamada a la rutina MPI_Reduce.

Se pudo aprender además, el uso que tienen otras directivas de comunicación global en los métodos que se encontraban dentro de los programas y de esta manera comprender mejor el uso que se le puede dar a estas rutinas. Matemáticamente, se pudo repasar nuevamente conceptos de manejo de vectores y la forma en que se puede operar con los mismos, además de poder ver el potencial de paralelización dentro de los cálculos.

Algo que llamó la atención a gran medida, fueron los prints. Debido a que al estar trabajando en el punto C, se encontró con el problema que el programa paralelo se tomaba ~9 segundos para poder terminar y al momento de quitar los prints se redujo el tiempo en 1 segundos y empezó a dar valores de ~8 segundos. Mientras que el secuencial se tardaba ~5 segundos en todo, incluso sin los prints. Esto se puede explicar con que hay menos movimiento de información entre los procesos, que al ser una gran cantidad de información por cada proceso llega a tardar porque tiene que ir en orden, se tiene que guardar en memoria y luego se tiene que mostrar en pantalla.

Al observar el laboratorio y sus ejercicios, se puede afirmar que la computación distribuida cuenta con una mayor eficiencia en el ejercicio de producto punto que en el de producto escalar. Debido a que en este se envía menos información entre procesos. Ya que al final solo hay que guardar N valores que son el número de procesos mientras que en el producto escalar se tienen que guardar (N x n) en donde n es el tamaño del array.