

**“AÑO DE LA UNIVERSALIZACIÓN DE LA SALUD”.**



**UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE  
AREQUIPA**

**ESCUELA PROFESIONAL DE CIENCIA DE LA  
COMPUTACIÓN  
COMPUTACIÓN PARALELA Y DISTRIBUIDA**

---

## **Trabajo 04**

---

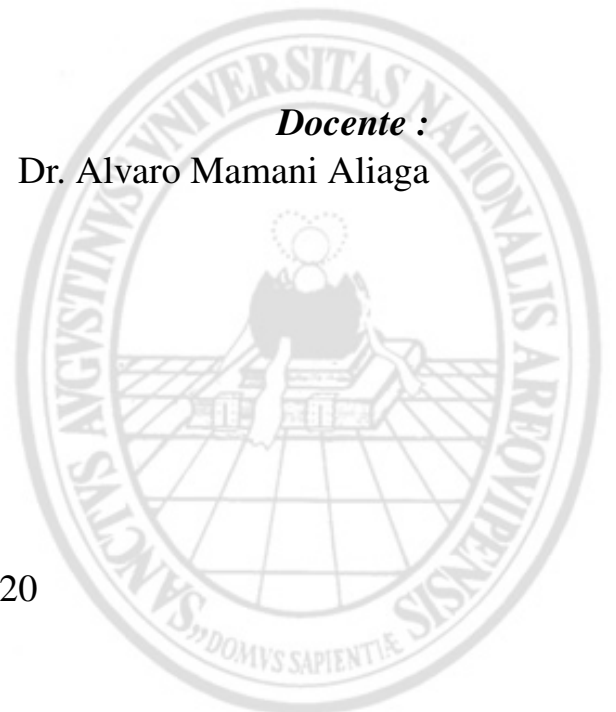
***Alumnos:***

Miguel Alexander, Herrera Cooper

***Docente :***

Dr. Alvaro Mamani Aliaga

2 de octubre de 2020



## Índice

<b>1. Ping Pong Algorithm</b>	<b>2</b>
<b>2. Regla Trapezoidal</b>	<b>3</b>
<b>3. Vector - Matriz</b>	<b>8</b>
<b>4. Parallel Odd-Even Transposition Sort</b>	<b>9</b>
<b>5. Conlusion</b>	<b>15</b>
<b>6. Repositorio</b>	<b>15</b>

# 1. Ping Pong Algorithm

## Resolución

---

Los procesos usan MPI\_Send y MPI\_Recv para lanzarse continuamente mensajes entre hasta 10 veces. El programa hace uso de un contador y determina el rango del otro proceso con la operación módulo. Mientras el contador incrementa, los procesos se turnan para ser emisor y receptor.

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 int main(int argc, char** argv) {
7     const int n_operaciones = 10;
8
9
10    MPI_Init(NULL, NULL);
11    // Find out rank, size
12    int rank_actual;
13    MPI_Comm_rank(MPI_COMM_WORLD, &rank_actual);
14    int world_size;
15    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16
17    int pp_contador = 0;
18    int rank = (rank_actual + 1) % 2;
19    while (pp_contador < n_operaciones) {
20        if (rank_actual == pp_contador % 2) {
21            pp_contador++;
22            MPI_Send(&pp_contador, 1, MPI_INT, rank, 0, MPI_COMM_WORLD);
23            sleep(1); //0.5 seg
24            printf("Proceso %d envio e incremento pp_contador %d al proceso %d\n",
25                rank_actual, pp_contador, rank);
26            sleep(1); //0.5 seg
27        } else {
28            MPI_Recv(&pp_contador, 1, MPI_INT, rank, 0, MPI_COMM_WORLD,
29                MPI_STATUS_IGNORE);
30            sleep(1); //0.5 seg
31            printf("Proceso %d recibio pp_contador %d de proceso %d\n",
32                rank_actual, pp_contador, rank);
33            sleep(1); //0.5 seg
34        }
35    }
36    MPI_Finalize();
37 }
```

```

Proceso 0 envio e incremento pp_contador 1 al proceso 1
Proceso 1 recibio pp_contador 1 de proceso 0
Proceso 1 envio e incremento pp_contador 2 al proceso 0
Proceso 0 recibio pp_contador 2 de proceso 1
Proceso 0 envio e incremento pp_contador 3 al proceso 1
Proceso 1 recibio pp_contador 3 de proceso 0
Proceso 1 envio e incremento pp_contador 4 al proceso 0
Proceso 0 recibio pp_contador 4 de proceso 1
Proceso 0 envio e incremento pp_contador 5 al proceso 1
Proceso 1 recibio pp_contador 5 de proceso 0
Proceso 0 recibio pp_contador 6 de proceso 1
Proceso 1 envio e incremento pp_contador 6 al proceso 0
Proceso 0 envio e incremento pp_contador 7 al proceso 1
Proceso 1 recibio pp_contador 7 de proceso 0
Proceso 0 recibio pp_contador 8 de proceso 1
Proceso 1 envio e incremento pp_contador 8 al proceso 0
Proceso 0 envio e incremento pp_contador 9 al proceso 1
Proceso 1 recibio pp_contador 9 de proceso 0
Proceso 0 recibio pp_contador 10 de proceso 1
Proceso 1 envio e incremento pp_contador 10 al proceso 0

```

Figura 1: Ejecucion del Ping Pong

## 2. Regla Trapezoidal

En este problema no se requiere comunicación entre tareas, por eso es apropiado utilizar la descomposición funcional del problema. Con este método se busca estimar el área bajo la curva para una función positiva.

La regla del Trapecio establece que si hay “n” trapecios dentro de una región, el área de todos los trapecios más el factor de error en la aproximación corresponde al área bajo la curva descrita por la función en el plano y se puede calcular con la Ecuación.

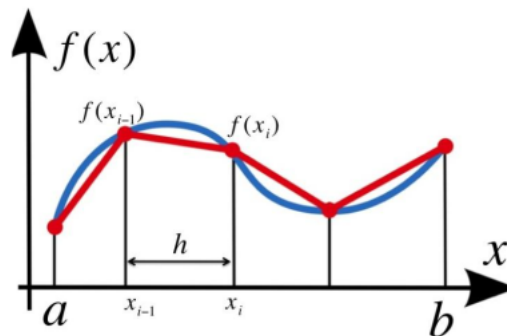
La función  $f(x) = x^2$  se utilizó para evaluar el método del trapecio.

Ecuación:

$$\frac{1}{2} h [f(x_{i-1}) + f(x_i)]$$

Donde h corresponde a la base del trapecio, del vértice izquierdo del trapecio y  $f(x_{i-1})$  corresponde a la longitud  $f(x_i)$  corresponde a la longitud del vértice derecho del trapecio tal como lo muestra la . La base “h” de cada trapecio es la misma dado que facilita las operaciones de distribución y recolección de información del problema entre todos los procesadores.

También existen muchas estrategias para lograr paralelizar un programa en un nivel aceptable. El método que se presenta consiste en la división del dominio en pequeños grupos de tareas con el objetivo de solucionar las operaciones de cada tarea en diferentes procesadores para obtener la solución en menos tiempo.



Las operaciones que se ejecutan dentro de cada procesador son en esencia las mismas, la única diferencia radica en que las cadenas de instrucciones que se procesan son diferentes. El problema se inicializa con un intervalo de integración  $[a, b]$  en donde se almacenan “n” trapecios. La aplicación paralela diseñada por el programador debe estar en capacidad de dividir los elementos que están almacenados en el intervalo  $[a, b]$  por todos los procesadores disponibles para ejecutar el programa.

El funcionamiento apropiado de las librerías y funciones de MPI dependen del programador, dado que en la fase de conceptualización y diseño del algoritmo debe tener en cuenta aspectos que influyen de forma directa con el apropiado funcionamiento del programa. Los aspectos principales que guían al programador hacia una apropiada implementación de estructuras paralelas tienen que ver con el número de procesadores que se esperan utilizar para la ejecución del programa. MPI está enfocado a computadores con alto nivel de rendimiento, así que es común enfocar aplicaciones hacia el uso de cientos de procesadores (esto depende del tamaño de las instrucciones y del programa como tal).

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <mpi.h>
5
6 const double a = 0;
7 const double b = 10;
8
9 /* Declaraciones de funciones */
10 void Get_input(int argc, char* argv[], int my_rank, double* n_p);
11 double Trap(double left_endpt, double right_endpt, int trap_count, double
    base_len);
12 double f(double x);
13
14 int main(int argc, char** argv) {
15     int my_rank, comm_sz, local_n;
16     double n, h, local_a, local_b;
17     double local_int, total_int;
18     double start, finish, loc_elapsed, elapsed;
19
20     MPI_Init(NULL, NULL);
21     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
22     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
23
24     /*Imprime en consola el nucleo que ejecuto el proceso*/
25     printf("soy el core nro. %d de %d\n", my_rank, comm_sz);
26

```

```

27 Get_input(argc, argv, my_rank, &n); /*Leer la entrada del usuario*/
28
29 /*Nota: h y local_n son iguales para todos los procesos */
30 h = (b - a) / n; /* longitud de cada trapezio */
31 local_n = n / comm_sz; /* cantidad de trapezios por proceso */
32
33 /* Duraci n del intervalo de integraci n de cada proceso = local_n * h.
   */
34 local_a = a + my_rank * local_n * h;
35 local_b = local_a + local_n * h;
36
37 MPI_Barrier(MPI_COMM_WORLD);
38 start = MPI_Wtime();
39
40 /* Calcular la integral local de cada proceso utilizando puntos finales
   locales*/
41 local_int = Trap(local_a, local_b, local_n, h);
42 finish = MPI_Wtime();
43 loc_elapsed = finish - start;
44 MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0,
   MPI_COMM_WORLD);
45
46 /* Suma las integrales calculadas por cada proceso */
47 MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
   MPI_COMM_WORLD);
48
49
50 if (my_rank == 0) {
51     printf("Con n = %.0f trapezoides, el valor de la integral entre %.0f a
   %.0f = %f \n", n, a, b, total_int);
52     printf("Tiempo transcurrido = %f milisegundos \n", elapsed * 1000);
53 }
54
55 /* Cerrar MPI */
56 MPI_Finalize();
57
58 return 0;
59 } /* main */
60
61
62 /*-----
63 * Funci n: Get_input
64 * Prop sito : obtener la entrada del usuario : el n mero de trapezios
65 **
66 * Args de entrada :
67 * 1. my_rank: rango de proceso en MPI_COMM_WORLD
68 * 2. comm_sz : n mero de procesos en MPI_COMM_WORLD
69 **
70 * Args de salida :
71 * 1. n_p: puntero al n mero de trapezios
72 */
73 void Get_input(int argc, char* argv[], int my_rank, double* n_p) {
74     if (my_rank == 0) {
75         if (argc != 2) {
76             fprintf(stderr, "uso: mpirun -np <N> %s <numero de trapezoides> \n",
   argv[0]);
77             fflush(stderr);

```

```

78     *n_p = -1;
79     }
80     else {
81         *n_p = atoi(argv[1]);
82     }
83     }
84     // Transmite el valor de n a cada proceso
85     MPI_Bcast(n_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
86
87     // n negativo termina el programa
88     if (*n_p <= 0) {
89         MPI_Finalize();
90         exit(-1);
91     }
92 } /* Get_input */
93
94 /*-----
95 * Funci n: Trap
96 * Propósito : funci n en serie para estimar una integral definida usando
97   la regla trapezoidal
98 **
99 * Args de entrada :
100 * -left_endpt
101 * -right_endpt
102 * -trap_count
103 * -base_len
104 **
105 * Valor de retorno : estimaci n de la regla trapezoidal de la integral de
106 * left_endpt a right_endpt usando trap_count trapecios
107 */
108 double Trap(double left_endpt, double right_endpt, int trap_count, double
109   base_len) {
110     double estimate, x;
111     int i;
112
113     estimate = (f(left_endpt) + f(right_endpt)) / 2.0;
114     for (i = 1; i <= trap_count - 1; i++) {
115         x = left_endpt + i * base_len;
116         estimate += f(x);
117     }
118     estimate = estimate * base_len;
119
120     return estimate;
121 } /* Trap */
122
123 /*-----
124 * Funci n: f
125 * Propósito : Calcular el valor de la funci n a integrar
126 * Args de entrada : x
127 */
128 double f(double x) {
129     double x1;
130     double x2;
131     x1 = ((x-4.0) * (x-4.0) * (x-4.0));
132     x2 = 2.0*x;
133     return ((0.2*x1)-x2)+12.0;;

```

133 } / \* f \* /



### 3. Vector - Matriz

#### Resolución

```

1 int main(void) {
2     float t0,t1, tiempo;
3     double* A = NULL;
4     double* x = NULL;
5     double* y = NULL;
6     int m=512;
7     for (int i = 0; i < 5; ++i) {
8         m = m*2;
9         A = malloc(m*m*sizeof(double));
10        x = malloc(m*sizeof(double));
11        y = malloc(m*sizeof(double));
12        if (A == NULL || x == NULL || y == NULL) {
13            fprintf(stderr, "Can't allocate storage\n");
14            exit(-1);
15        }
16        Read_matrix("A", A, m, m);
17        Read_vector("x", x, m);
18        t0 = clock();
19        Mat_vect_mult(A, x, y, m, m);
20        t1 = clock();
21        tiempo = ((t1-t0)/CLOCKS_PER_SEC);
22        printf("%f\n",tiempo);
23        free(A);free(x);free(y);
24    }
25    return 0;
26 }

```

```

0.003869
0.003874
0.013609
0.013634
0.055023
0.055064
0.219861
0.219652

```

## 4. Parallel Odd-Even Transposition Sort

### Resolución

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <mpi.h>
5 #include <time.h>      /* time */
6 // const int RMAX = 1000000000;
7 const int RMAX = 100;
8
9 /* Local functions */
10 void Usage(char* program);
11 void Print_list(int local_A[], int local_n, int rank);
12 void Merge_split_low(int local_A[], int temp_B[], int temp_C[],
13     int local_n);
14 void Merge_split_high(int local_A[], int temp_B[], int temp_C[],
15     int local_n);
16 void Generate_list(int local_A[], int local_n, int my_rank);
17 int Compare(const void* a_p, const void* b_p);
18
19 /* Functions involving communication */
20 void Get_args(int argc, char* argv[], int* global_n_p, int* local_n_p,
21     char* gi_p, int my_rank, int p, MPI_Comm comm);
22 void Sort(int local_A[], int local_n, int my_rank,
23     int p, MPI_Comm comm);
24 void Odd_even_iter(int local_A[], int temp_B[], int temp_C[],
25     int local_n, int phase, int even_partner, int odd_partner,
26     int my_rank, int p, MPI_Comm comm);
27 void Print_local_lists(int local_A[], int local_n,
28     int my_rank, int p, MPI_Comm comm);
29 void Print_global_list(int local_A[], int local_n, int my_rank,
30     int p, MPI_Comm comm);
31
32
33 int main(int argc, char* argv[]) {
34     int my_rank, p;
35     char g_i;
36     int* local_A;
37     int global_n;
38     int local_n;
39     MPI_Comm comm;
40     double start, finish;
41
42     MPI_Init(&argc, &argv);
43     comm = MPI_COMM_WORLD;
44     MPI_Comm_size(comm, &p);
45     MPI_Comm_rank(comm, &my_rank);
46
47     Get_args(argc, argv, &global_n, &local_n, &g_i, my_rank, p, comm);
48     local_A = (int*)malloc(local_n * sizeof(int));
49     if (g_i == 'g') {
50         Generate_list(local_A, local_n, my_rank);

```

```

51     }
52     else {
53         //Read_list(local_A, local_n, my_rank, p, comm);
54     }
55 #   ifdef DEBUG
56     Print_local_lists(local_A, local_n, my_rank, p, comm);
57 #   endif
58
59     start = MPI_Wtime();
60     Sort(local_A, local_n, my_rank, p, comm);
61     finish = MPI_Wtime();
62     if (my_rank == 0)
63         printf("Elapsed time = %e seconds\n", finish - start);
64
65 #   ifdef DEBUG
66     Print_local_lists(local_A, local_n, my_rank, p, comm);
67     fflush(stdout);
68 #   endif
69
70     Print_global_list(local_A, local_n, my_rank, p, comm);
71
72     free(local_A);
73
74     MPI_Finalize();
75
76     return 0;
77 } /* main */
78
79 void Generate_list(int local_A[], int local_n, int my_rank) {
80     int i;
81
82     srand(my_rank + 1);
83     for (i = 0; i < local_n; i++)
84         local_A[i] = rand() % RMAX;
85
86 } /* Generate_list */
87
88
89 void Usage(char* program) {
90     fprintf(stderr, "usage: mpirun -np <p> %s <g|i> <global_n>\n",
91             program);
92     fprintf(stderr, "    - p: the number of processes \n");
93     fprintf(stderr, "    - g: generate random, distributed list\n");
94     fprintf(stderr, "    - i: user will input list on process 0\n");
95     fprintf(stderr, "    - global_n: number of elements in global list");
96     fprintf(stderr, " (must be evenly divisible by p)\n");
97     fflush(stderr);
98 } /* Usage */
99
100
101 /*-----
102 * Function:    Get_args
103 * Purpose:     Get and check command line arguments
104 * Input args:  argc, argv, my_rank, p, comm
105 * Output args: global_n_p, local_n_p, gi_p
106 */
107 void Get_args(int argc, char* argv[], int* global_n_p, int* local_n_p,

```

```

108 char* gi_p, int my_rank, int p, MPI_Comm comm) {
109
110 if (my_rank == 0) {
111     if (argc != 3) {
112         Usage(argv[0]);
113         *global_n_p = -1; /* Bad args, quit */
114     }
115     else {
116         *gi_p = argv[1][0];
117         if (*gi_p != 'g' && *gi_p != 'i') {
118             Usage(argv[0]);
119             *global_n_p = -1; /* Bad args, quit */
120         }
121         else {
122             *global_n_p = strtol(argv[2], NULL, 10);
123             if (*global_n_p % p != 0) {
124                 Usage(argv[0]);
125                 *global_n_p = -1;
126             }
127         }
128     }
129 } /* my_rank == 0 */
130
131 MPI_Bcast(gi_p, 1, MPI_CHAR, 0, comm);
132 MPI_Bcast(global_n_p, 1, MPI_INT, 0, comm);
133
134 if (*global_n_p <= 0) {
135     MPI_Finalize();
136     exit(-1);
137 }
138
139 *local_n_p = *global_n_p / p;
140
141 } /* Get_args */
142
143
144
145 void Print_global_list(int local_A[], int local_n, int my_rank, int p,
146 MPI_Comm comm) {
147     int* A = NULL;
148     int i, n;
149
150     if (my_rank == 0) {
151         n = p * local_n;
152         A = (int*)malloc(n * sizeof(int));
153         MPI_Gather(local_A, local_n, MPI_INT, A, local_n, MPI_INT, 0,
154 comm);
155         printf("Global list:\n");
156         for (i = 0; i < n; i++)
157             printf("%d ", A[i]);
158         printf("\n\n");
159         free(A);
160     }
161     else {
162         MPI_Gather(local_A, local_n, MPI_INT, A, local_n, MPI_INT, 0,
163 comm);
164     }

```

```

165
166 } /* Print_global_list */
167
168
169 int Compare(const void* a_p, const void* b_p) {
170     int a = *((int*)a_p);
171     int b = *((int*)b_p);
172
173     if (a < b)
174         return -1;
175     else if (a == b)
176         return 0;
177     else /* a > b */
178         return 1;
179 } /* Compare */
180
181
182 void Sort(int local_A[], int local_n, int my_rank,
183     int p, MPI_Comm comm) {
184     int phase;
185     int* temp_B, * temp_C;
186     int even_partner; /* phase is even or left-looking */
187     int odd_partner; /* phase is odd or right-looking */
188
189     /* Temporary storage used in merge-split */
190     temp_B = (int*)malloc(local_n * sizeof(int));
191     temp_C = (int*)malloc(local_n * sizeof(int));
192
193     /* Find partners: negative rank => do nothing during phase */
194     if (my_rank % 2 != 0) {
195         even_partner = my_rank - 1;
196         odd_partner = my_rank + 1;
197         if (odd_partner == p) odd_partner = -1; // Idle during odd phase
198     }
199     else {
200         even_partner = my_rank + 1;
201         if (even_partner == p) even_partner = -1; // Idle during even
202         phase
203         odd_partner = my_rank - 1;
204     }
205
206     /* Sort local list using built-in quick sort */
207     qsort(local_A, local_n, sizeof(int), Compare);
208
209     for (phase = 0; phase < p; phase++)
210         Odd_even_iter(local_A, temp_B, temp_C, local_n, phase,
211             even_partner, odd_partner, my_rank, p, comm);
212
213     free(temp_B);
214     free(temp_C);
215 } /* Sort */
216
217 void Odd_even_iter(int local_A[], int temp_B[], int temp_C[],
218     int local_n, int phase, int even_partner, int odd_partner,
219     int my_rank, int p, MPI_Comm comm) {
220     MPI_Status status;

```

```

221
222     if (phase % 2 == 0) { /* Even phase, odd process <-> rank-1 */
223         if (even_partner >= 0) {
224             MPI_Sendrecv(local_A, local_n, MPI_INT, even_partner, 0,
225                 temp_B, local_n, MPI_INT, even_partner, 0, comm,
226                 &status);
227             if (my_rank % 2 != 0)
228                 Merge_split_high(local_A, temp_B, temp_C, local_n);
229             else
230                 Merge_split_low(local_A, temp_B, temp_C, local_n);
231         }
232     }
233     else { /* Odd phase, odd process <-> rank+1 */
234         if (odd_partner >= 0) {
235             MPI_Sendrecv(local_A, local_n, MPI_INT, odd_partner, 0,
236                 temp_B, local_n, MPI_INT, odd_partner, 0, comm,
237                 &status);
238             if (my_rank % 2 != 0)
239                 Merge_split_low(local_A, temp_B, temp_C, local_n);
240             else
241                 Merge_split_high(local_A, temp_B, temp_C, local_n);
242         }
243     }
244 } /* Odd_even_iter */
245
246
247
248 void Merge_split_low(int local_A[], int temp_B[], int temp_C[],
249     int local_n) {
250     int ai, bi, ci;
251
252     ai = 0;
253     bi = 0;
254     ci = 0;
255     while (ci < local_n) {
256         if (local_A[ai] <= temp_B[bi]) {
257             temp_C[ci] = local_A[ai];
258             ci++; ai++;
259         }
260         else {
261             temp_C[ci] = temp_B[bi];
262             ci++; bi++;
263         }
264     }
265
266     memcpy(local_A, temp_C, local_n * sizeof(int));
267 } /* Merge_split_low */
268
269
270 void Merge_split_high(int local_A[], int temp_B[], int temp_C[],
271     int local_n) {
272     int ai, bi, ci;
273
274     ai = local_n - 1;
275     bi = local_n - 1;
276     ci = local_n - 1;
277     while (ci >= 0) {

```

```

278     if (local_A[ai] >= temp_B[bi]) {
279         temp_C[ci] = local_A[ai];
280         ci--; ai--;
281     }
282     else {
283         temp_C[ci] = temp_B[bi];
284         ci--; bi--;
285     }
286 }
287
288 memcpy(local_A, temp_C, local_n * sizeof(int));
289 } /* Merge_split_low */
290
291
292 void Print_list(int local_A[], int local_n, int rank) {
293     int i;
294     printf("%d: ", rank);
295     for (i = 0; i < local_n; i++)
296         printf("%d ", local_A[i]);
297     printf("\n");
298 } /* Print_list */
299
300
301 void Print_local_lists(int local_A[], int local_n,
302     int my_rank, int p, MPI_Comm comm) {
303     int* A;
304     int q;
305     MPI_Status status;
306
307     if (my_rank == 0) {
308         A = (int*)malloc(local_n * sizeof(int));
309         Print_list(local_A, local_n, my_rank);
310         for (q = 1; q < p; q++) {
311             MPI_Recv(A, local_n, MPI_INT, q, 0, comm, &status);
312             Print_list(A, local_n, q);
313         }
314         free(A);
315     }
316     else {
317         MPI_Send(local_A, local_n, MPI_INT, 0, 0, comm);
318     }
319 } /* Print_local_lists */

```

La comunicación dentro de los mismos nodos funciona mucho mejor que entre nodos. En comparación con dos implementaciones, los tiempos de intercambio entre procesos serán la variable más significativa que afectará la eficiencia del tiempo.

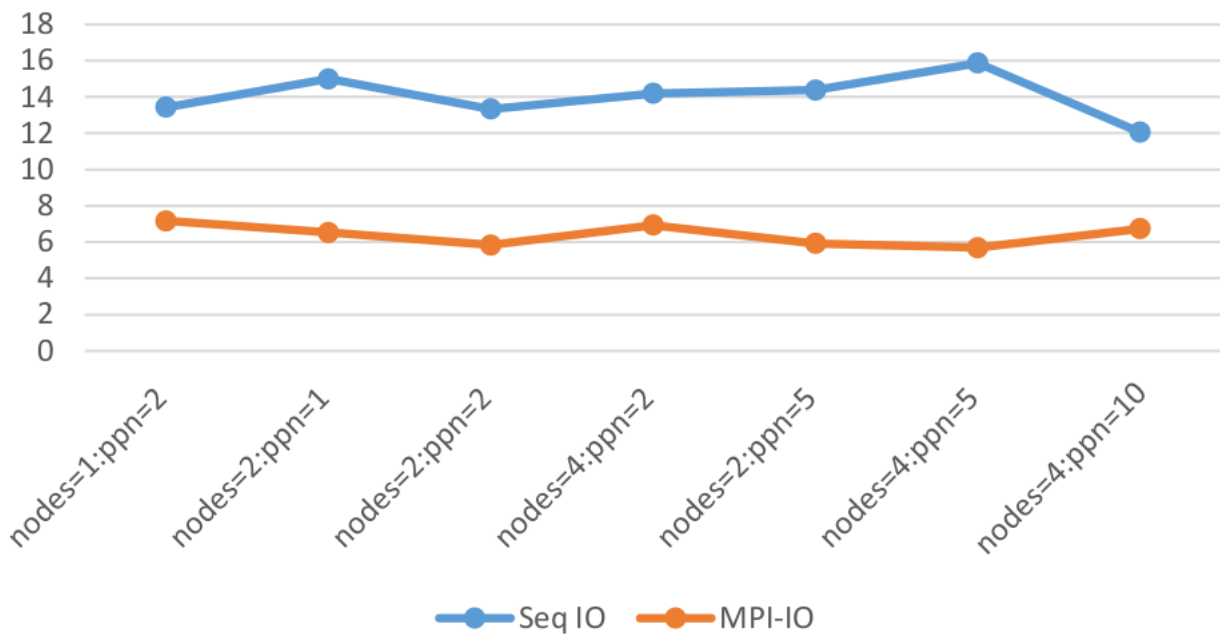


Figura 2: Rendimiento para diferentes I/O - Data = 100000

## 5. Conclusion

El rendimiento entre secuencial y MPI-I/O, no es lo suficientemente claro como para afirmar cuál es mucho mejor. Tal vez el sistema de las máquinas de prueba no tenga un sistema de archivos distribuido bien construido, por lo que la API MPI-IO no tiene una aceleración obvia.

En este proyecto, he aprendido que el programa es una parte esencial, pero el experimento es otra parte vital en la que puede demostrar plenamente que el trabajo que realiza es lo suficientemente bueno en computación paralela.

## 6. Repositorio

En el siguiente enlace se puede ver el código fuente del trabajo.