Nombre: Randú Jean Franco Cerpa García

Problema 1:

Use MPI to implement the histogram program discussed in Section 2.7.1. Have process 0 read in the input data and distribute it among the processes. Also have process 0 print out the histogram.

El problema es construir el histograma de un conjunto de datos reales en [min,max) usando B bins, con el proceso 0 leyendo los datos y reportando el resultado.

Ejecución:

```
Histograma (5 bins) en rango [0.000000, 5.000000):
[0, 1): 6  |******|
[1, 2): 3  |***|
[2, 3): 2  |**|
[3, 4): 3  |***|
[4, 5) (incluye max): 6  |******|
```

Input:

```
20 5 0.0 5.0
1.3 2.9 0.4 0.3 1.3 4.4 1.7 0.4 3.2 0.3 4.9 2.4 3.1 4.4 3.9 0.4 4.2 4.5 4.9 0.9
```

20 → data_count: cuántos datos vienen después.

5 → bin_count: cuántos bins (intervalos iguales) quieres.

0.0 → min_meas (límite inferior del rango).

5.0 → max_meas (límite superior del rango).

```
33    if (rank == 0) {
34        FILE* f = fopen(argv[1], "r");
35        if (!f) { perror("fopen"); MPI_Abort(MPI_COMM_WORLD, 1); }
36        if (fscanf(f, "%lld %d %lf %lf", &data_count, &bin_count, &min_meas, &max_meas) != 4) {
37            fprintf(stderr, "Formato: data_count bin_count min_meas max_meas \\n <data...>\n");
38            MPI_Abort(MPI_COMM_WORLD, 1);
39        }
40        if (bin_count <= 0 || max_meas <= min_meas) {
41            fprintf(stderr, "Parametros invalidos.\n");
42            MPI_Abort(MPI_COMM_WORLD, 1);
43        }
44        data = (double*)malloc(sizeof(double)*data_count);
45        for (long long i = 0; i < data_count; i++) {
46            if (fscanf(f, "%lf", &data[i]) != 1) {
47                fprintf(stderr, "Faltan datos en el input.\n");
48                MPI_Abort(MPI_COMM_WORLD, 1);
49            }
50        }
51        fclose(f);
52    }
```

Se leen los datos del txt, y se validan estos datos para no tener datos erróneos.

```
66    // Scatterv de datos
67    int* sendcounts = (int*)malloc(sizeof(int)*comm_sz);
68    int* displs = (int*)malloc(sizeof(int)*comm_sz);
69    long long base = data_count / comm_sz, rem = data_count % comm_sz;
70    long long offset = 0;
71    for (int p = 0; p < comm_sz; p++) {
72        long long chunk = base + (p < rem ? 1 : 0);
73        sendcounts[p] = (int)chunk;
74        displs[p] = (int)offset;
75        offset += chunk;
76    }
77
78    double* local_data = (double*)malloc(sizeof(double)*sendcounts[rank]);
79    MPI_Scatterv(data, sendcounts, displs, MPI_DOUBLE,
80                 local_data, sendcounts[rank], MPI_DOUBLE,
81                 0, MPI_COMM_WORLD);
```

Se distribuyen la data del proceso principal a los demás procesos.

```
91      // Reducción a proceso 0
92      int* global_counts = NULL;
93      if (rank == 0) global_counts = (int*)calloc(bin_count, sizeof(int));
94      MPI_Reduce(local_counts, global_counts, bin_count, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
95
96      // Proceso 0 imprime histograma
97      if (rank == 0) {
98          printf("Histograma (%d bins) en rango [%.6f, %.6f):\n", bin_count, min_meas, max_meas);
99          for (int b = 0; b < bin_count; b++) {
100             double lo = (b == 0 ? min_meas : bin_maxes[b-1]);
101             double hi = bin_maxes[b];
102             printf("[%g, %g)%s: %d ", lo, hi, (b==bin_count-1 ? " (incluye max)" : ""), global_counts[b]);
103             int stars = global_counts[b] > 60 ? 60 : global_counts[b];
104             printf(" |");
105             for (int s = 0; s < stars; s++) putchar('*');
106             printf("|\n");
107         }
```

se hace un reduce para sumar los contadores de cada proceso todos al rank 0.

Problema 2:

3.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write an MPI program that uses a Monte Carlo method to estimate π. Process 0 should read in the total number of tosses and broadcast it to the other processes. Use `MPI_Reduce` to find the global sum of the local variable `number_in_circle`, and have process 0 print the result. You may want to use **long long int**s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

```
Aproximación de Pi con 1000000 lanzamientos: 3.140184000000
vboxuser@nodo1:~$ mpirun -np 3 --hostfile ~/hosts -wdir ~ ./mpi_pi 10000000
```

```
Aproximación de Pi con 10000000 lanzamientos: 3.141458000000
vboxuser@nodo1:~$
```

```
20        long long n_total = 0;
21        if (rank==0) n_total = atoll(argv[1]);
22        MPI_Bcast(&n_total,1,MPI_LONG_LONG,0,MPI_COMM_WORLD);
23
24        long long base = n_total/size, rem = n_total%size;
25        long long n_local = base + (rank<rem?1:0);
```

Se envía el dato a los procesos y se reparte lo más parejo posible.

```
29        long long hits = 0;
30        for(long long i=0;i<n_local;i++){
31            double x = urand(&seed), y = urand(&seed);
32            if (x*x + y*y <= 1.0) hits++;
33        }
```

Cuenta aciertos hits si caen dentro del cuarto de circulo x2+y2≤1.

3.3. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which `comm_sz` is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any `comm_sz`.

```
16    // 1) Si p no es potencia de dos, "apartar" los ranks extra hacia la potencia de dos inferior
17    int active = 1;
18    while ((active << 1) <= p) active <<= 1;  // mayor potencia de 2 <= p
19    // Fase de "compactación" hacia ranks [0..active-1]
20    if (rank >= active) {
21        int partner = rank - active;              // envía al partner "debajo"
22        MPI_Send(&local, 1, MPI_LONG_LONG, partner, 0, MPI_COMM_WORLD);
23        // Este rank ya no participa en el árbol:
24        MPI_Finalize();
25        return 0;
26    } else {
27        int partner = rank + active;
28        if (partner < p) {
29            long long recv_val = 0;
30            MPI_Recv(&recv_val, 1, MPI_LONG_LONG, partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
31            local += recv_val;
32        }
33    }
```

si no es potencia de 2 el total de procesos, se pliegan hacia debajo de tal forma

que queda compactado en una potencia de 2 en total, el emisor envía la suma y el receptor recibe esta suma, si el n de procesos es potencia de 2 se salta el paso de compactar y se realiza la misma suma.

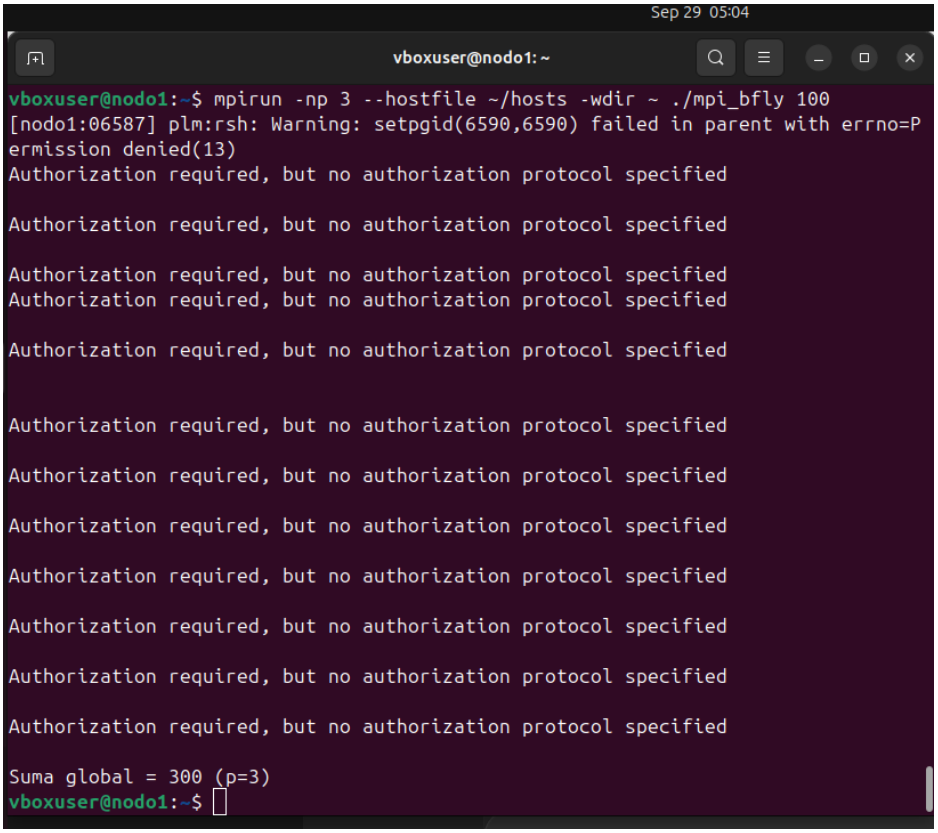
```
Suma global = 6 (p=3)
vboxuser@nodo1:~$
```

```
vboxuser@nodo1:~$ mpirun -np 3 --hostfile ~/hosts -wdir
```

Como solo trabajo con 3 procesos, el proceso, la potencia de 2 más cercana es 2, entonces el rank 2 se apila en rank 0 y rank 2 termina, rank 1 se suma a rank 0 y termina y rank 0 suma lo recibido y queda con la suma global, los datos se asignan así:

Rank+1, entonces el rank 0 tiene el valor de 1, rank 1 valor de 2 y rank 2 valor de 3.

3.4. Write an MPI program that computes a global sum using a butterfly. First write your program for the special case in which `comm_sz` is a power of two. Can you modify your program so that it will handle any number of processes?



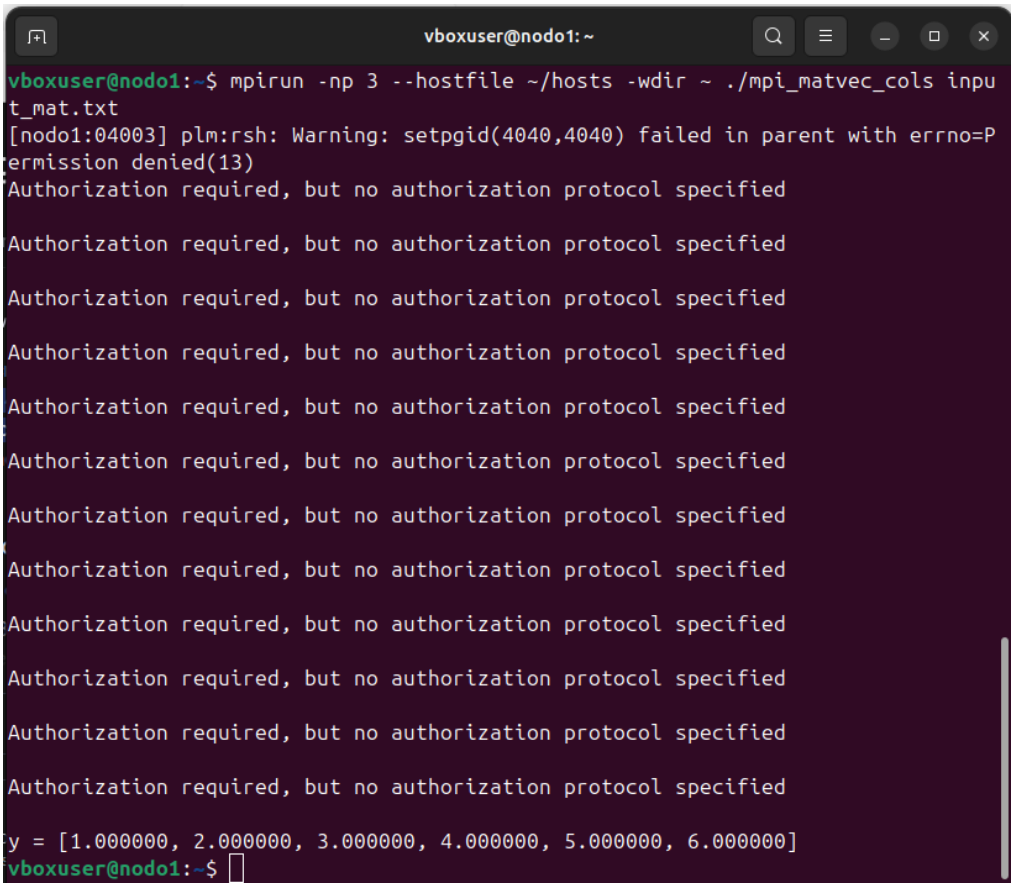
```
21      if (rank >= active) {
22          // Enviar mi contribución al "gemelo" rank-active y salir
23          twin = rank - active;
24          MPI_Send(&local, 1, MPI_LONG_LONG, twin, 100, MPI_COMM_WORLD);
25          // Esperar el resultado final para tener también la suma global (estilo all-reduce)
26          MPI_Recv(&local, 1, MPI_LONG_LONG, twin, 200, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
27          // Imprimir solo si quieres ver que TODOS lo tienen; aquí deja que imprima rank 0
28          MPI_Finalize();
29          return 0;
30      } else {
31          // Si tengo un gemelo plegado, acumulo su contribución
32          twin = rank + active;
33          if (twin < p) {
34              long long add=0;
35              MPI_Recv(&add, 1, MPI_LONG_LONG, twin, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
36              local += add;
37          }
38      }
39
```

cuando el proceso actual es mayor o igual a la mayor potencia de 2, se manda su dato al gemelo que es donde se apila su dato, si rank es menor a la mayor potencia de 2 que sea menor o igual que p, recibe y suma, como las vms solo tiene 3 procesos, se trabaja con 2 y se tiene un gemelo para rank 0.

```
42      for (int d = 0; (1<<d) < active; d++){
43          int partner = rank ^ (1<<d);
44          long long recv_val = 0;
45          MPI_Sendrecv(&local, 1, MPI_LONG_LONG, partner, 300+d,
46                       &recv_val, 1, MPI_LONG_LONG, partner, 300+d,
47                       MPI_COMM_WORLD, MPI_STATUS_IGNORE);
48          local += recv_val;
49      }
```

En cada fase d cada proceso intercambia su suma local con su partner y los acumula, es el proceso mariposa como tal. Usar MPI_Sendrecv evita deadlocks (send-send) . Al final todos los procesos conocen la suma global pero en este código solo imprime rank 0.

3.5. Implement matrix-vector multiplication using a block-column distribution of the matrix. You can have process 0 read in the matrix and simply use a loop of sends to distribute it among the processes. Assume the matrix is square of order n and that n is evenly divisible by `comm_sz`. You may want to look at the MPI function `MPI_Reduce_scatter`.



Datos:

```
1    6
2    1 2 3 4 5 6
3    1 0 0 0 0 0
4    0 1 0 0 0 0
5    0 0 1 0 0 0
6    0 0 0 1 0 0
7    0 0 0 0 1 0
8    0 0 0 0 0 1
```

```c
if (rank==0){
    FILE* f = fopen(argv[1], "r");
    if (!f){ perror("fopen"); MPI_Abort(MPI_COMM_WORLD,1); }
    if (fscanf(f, "%d", &n) != 1){ fprintf(stderr,"Formato: n, luego n doubles de x, luego n*n de A\n"); MPI_Abort(MPI_COMM_WORLD,1); }
    if (n<=0){ fprintf(stderr,"n invalido\n"); MPI_Abort(MPI_COMM_WORLD,1); }
    x_full = (double*)malloc(sizeof(double)*n);
    for (int i=0;i<n;i++) if (fscanf(f, "%lf", &x_full[i])!=1){ fprintf(stderr,"x incompleto\n"); MPI_Abort(MPI_COMM_WORLD,1); }
    A_full = (double*)malloc(sizeof(double)*n*(size_t)n);
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
            if (fscanf(f, "%lf", &A_full[i*(size_t)n + j])!=1){ fprintf(stderr,"A incompleta\n"); MPI_Abort(MPI_COMM_WORLD,1); }
    fclose(f);
}
```

Lectura de los datos del input, el primer dato es el tamaño de la matriz y el vector, debe ser divisible por size, en la segunda fila es el valor del vector, y por ultimo los elementos de la matriz A, en este caso la matriz identidad.

```c
if (rank==0){
    // Para q=0, empacar a A_loc/x_loc; para q>0, empaquetar a tmp y enviar
    for (int q=0; q<p; q++){
        int c0 = q*ccols;
        if (q==0){
            for (int j=0;j<ccols;j++) x_loc[j] = x_full[c0+j];
            pack_block_cols(A_full, n, c0, ccols, A_loc);
        } else {
            double* tmpA = (double*)malloc(sizeof(double)*n*(size_t)ccols);
            double* tmpx = (double*)malloc(sizeof(double)*ccols);
            for (int j=0;j<ccols;j++) tmpx[j] = x_full[c0+j];
            pack_block_cols(A_full, n, c0, ccols, tmpA);
            MPI_Send(tmpx, ccols, MPI_DOUBLE, q, 10, MPI_COMM_WORLD);
            MPI_Send(tmpA, n*ccols, MPI_DOUBLE, q, 11, MPI_COMM_WORLD);
            free(tmpA); free(tmpx);
        }
    }
} else {
    MPI_Recv(x_loc, ccols,   MPI_DOUBLE, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(A_loc, n*ccols, MPI_DOUBLE, 0, 11, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Corta la matriz y el vector en bloques de columnas y manda a cada proceso, en este caso 2 columnas y filas por proceso.

```c
double* y_block = (double*)malloc(sizeof(double)*rrows);
int* recvcounts = (int*)malloc(sizeof(int)*p);
for (int q=0;q<p;q++) recvcounts[q] = rrows;
MPI_Reduce_scatter(y_loc, y_block, recvcounts, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Suma las contribuciones entre los procesos y reparte el resultado en bloques de filas

```
91      double* y_full = NULL;
92      if (rank==0) y_full = (double*)malloc(sizeof(double)*n);
93      MPI_Gather(y_block, rrows, MPI_DOUBLE, y_full, rrows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
94
95      if (rank==0){
96          printf("y = [");
97          for (int i=0;i<n;i++) printf("%s%.6f", (i?", ":""), y_full[i]);
98          printf("]\n");
99      }
```

imprime el vector completo en rank 0.

3.7. A **ping-pong** is a communication in which two messages are sent, first from process A to process B (ping) and then from process B back to process A (pong). Timing blocks of repeated ping-pongs is a common way to estimate the cost of sending messages. Time a ping-pong program using the C `clock` function on your system. How long does the code have to run before `clock` gives a nonzero run-time? How do the times you got with the `clock` function compare to times taken with `MPI_Wtime`?

```
vboxuser@nodo1:~$ mpirun -np 2 --host 192.168.56.101,192.168.56.102 -wdir ~ ./mp
i_pingpong 0 100000
Authorization required, but no authorization protocol specified

Authorization required, but no authorization protocol specified

Authorization required, but no authorization protocol specified

Authorization required, but no authorization protocol specified

Authorization required, but no authorization protocol specified

Authorization required, but no authorization protocol specified

pingpong: size=0 bytes  iters=100000
MPI_Wtime: wall=103.737060 s   RTT=1037.371us   one-way≈518.685us   BW≈0.0 MB/s
clock()  : cpu =98.469487 s   (ticks=98469487, CLOCKS_PER_SEC=1000000)
vboxuser@nodo1:~$
```

Rtt es el tiempo total/ iteraciones, one way es el tiempo rtt/2

```
19      if(p < 2){ if(rank==0) fprintf(stderr,"Se requieren 2 procesos.\n"); MPI_Abort(MPI_COMM_WORLD,1); }
20      if(rank >= 2){ MPI_Finalize(); return 0; }  // solo 0 y 1 participan
21
22      size_t msg_bytes = (argc>=2)? strtoull(argv[1],NULL,10) : 0ULL;     // por defecto: 0 bytes
23      long long iters  = (argc>=3)? atoll(argv[2]) : 100000LL;            // por defecto: 100000
24      int warmup = 1000;
25      if (warmup > iters/10) warmup = (int)(iters/10);
26
27      char* buf = NULL;
28      if (msg_bytes > 0){
29          buf = (char*)malloc(msg_bytes);
30          for (size_t i=0;i<msg_bytes;i++) buf[i] = (char)(i & 0xFF);
31      }
```

Se definen los procesos que participan para el ping pong, solo 2. Se define el tamaño del mensaje y el numero de iteraciones, se pueden escribir por consola, si el dato es mayor a 0 se rellena el buffer.

```
33    // Sincroniza y calienta
34    barrier2(MPI_COMM_WORLD);
35    for(int i=0;i<warmup;i++){
36        if(rank==0){ MPI_Send(buf, (int)msg_bytes, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
37                     MPI_Recv(buf, (int)msg_bytes, MPI_CHAR, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); }
38        else       { MPI_Recv(buf, (int)msg_bytes, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
39                     MPI_Send(buf, (int)msg_bytes, MPI_CHAR, 0, 0, MPI_COMM_WORLD); }
40    }
41    barrier2(MPI_COMM_WORLD);
```

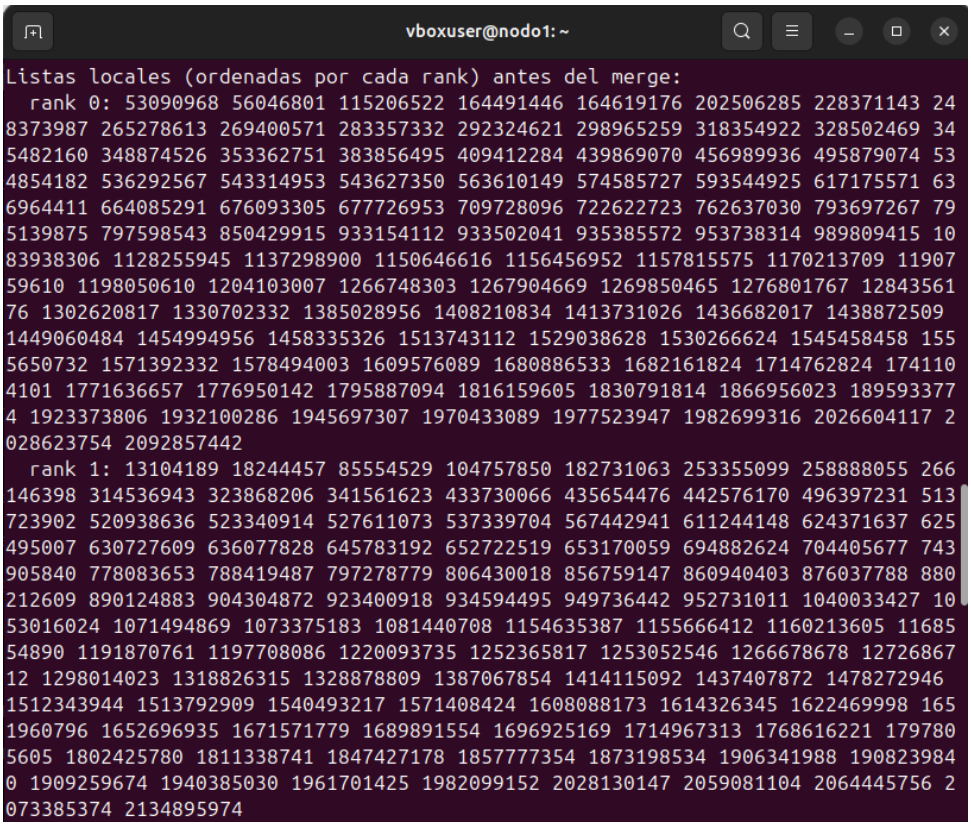
Calienta la ruta para evitar medir arranques “frios”, warmup = 1000.

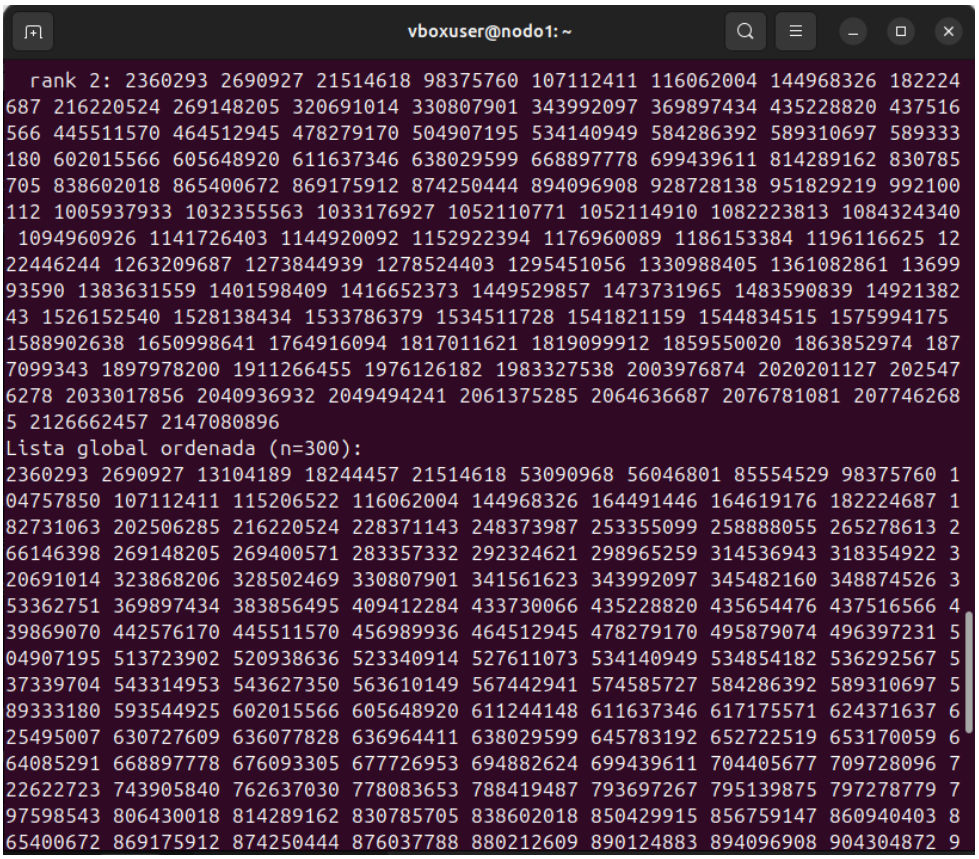
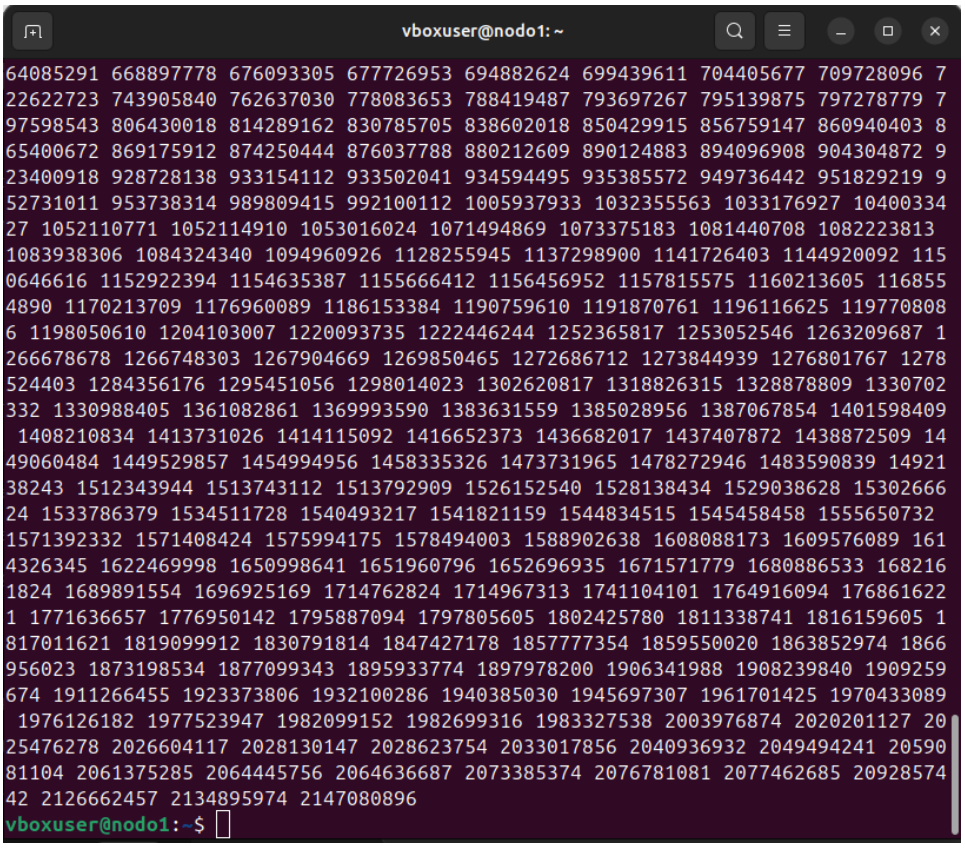
```
43    // Tiempo de pared
44    double t0 = MPI_Wtime();
45    // Tiempo de CPU
46    clock_t c0 = clock();
47
48    for(long long i=0;i<iters;i++){
49        if(rank==0){ MPI_Send(buf, (int)msg_bytes, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
50                     MPI_Recv(buf, (int)msg_bytes, MPI_CHAR, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); }
51        else       { MPI_Recv(buf, (int)msg_bytes, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
52                     MPI_Send(buf, (int)msg_bytes, MPI_CHAR, 0, 0, MPI_COMM_WORLD); }
53    }
54
55    clock_t c1 = clock();
56    double t1 = MPI_Wtime();
57
58    if(rank==0){
59        double wall = t1 - t0;                          // seg (tiempo de pared)
60        double cpu  = (double)(c1 - c0) / CLOCKS_PER_SEC; // seg (tiempo de CPU)
61        double rtt  = wall / (double)iters;             // round-trip time
62        double one_way_lat = rtt / 2.0;                 // latencia aprox. unidireccional
63        double bytes_xfer = (double)msg_bytes * 2.0 * (double)iters; // ida+vuelta
64        double bandwidth  = (bytes_xfer / wall) / 1e6;  // MB/s
65
66        printf("pingpong: size=%zu bytes  iters=%lld\n", msg_bytes, iters);
67        printf("MPI_Wtime: wall=%.6f s   RTT=%.3fus   one-way≈%.3fus   BW≈%.1f MB/s\n",
68               wall, 1e6*rtt, 1e6*one_way_lat, bandwidth);
69        printf("clock()  : cpu =%.6f s   (ticks=%ld, CLOCKS_PER_SEC=%ld)\n",
70               cpu, (long)(c1-c0), (long)CLOCKS_PER_SEC);
71    }
72
```

Se guardan los tiempos antes de hacer el ping pong, y luego de hacerlo, y se muestran los resultados.

Wall time es el tiempo completo que transcurre desde un punto a otro, incluye todo. Clock solo mide el tiempo que se pasó ejecutando instrucciones en la cpu.

3.8. Parallel merge sort starts with n/`comm_sz` keys assigned to each process. It ends with all the keys stored on process 0 in sorted order. To achieve this, it uses the same tree-structured communication that we used to implement a global sum. However, when a process receives another process' keys, it merges the new keys into its already sorted list of keys. Write a program that implements parallel mergesort. Process 0 should read in n and broadcast it to the other processes. Each process should use a random number generator to create a local list of `n/comm_sz` `ints`. Each process should then sort its local list, and process 0 should gather and print the local lists. Then the processes should use tree-structured communication to merge the global list onto process 0, which prints the result.




vboxuser@nodo1: ~
rank 2: 2360293 2690927 21514618 98375760 107112411 116062004 144968326 182224687 216220524 269148205 320691014 330807901 343992097 369897434 435228820 437516566 445511570 464512945 478279170 504907195 534140949 584286392 589310697 589333180 602015566 605648920 611637346 638029599 668897778 699439611 814289162 830785705 838602018 865400672 869175912 874250444 894096908 928728138 951829219 992100112 1005937933 1032355563 1033176927 1052110771 1052114910 1082223813 1084324340 1094960926 1141726403 1144920092 1152922394 1176960089 1186153384 1196116625 1222446244 1263209687 1273844939 1278524403 1295451056 1330988405 1361082861 1369993590 1383631559 1401598409 1416652373 1449529857 1473731965 1483590839 1492138243 1526152540 1528138434 1533786379 1534511728 1541821159 1544834515 1575994175 1588902638 1650998641 1764916094 1817011621 1819099912 1859550020 1863852974 1877099343 1897978200 1911266455 1976126182 1983327538 2003976874 2020201127 2025476278 2033017856 2040936932 2049494241 2061375285 2064636687 2076781081 2077462685 2126662457 2147080896
Lista global ordenada (n=300):
2360293 2690927 13104189 18244457 21514618 53090968 56046801 85554529 98375760 104757850 107112411 115206522 116062004 144968326 164491446 164619176 182224687 182731063 202506285 216220524 228371143 248373987 253355099 258888055 265278613 266146398 269148205 269400571 283357332 292324621 298965259 314536943 318354922 320691014 323868206 328502469 330807901 341561623 343992097 345482160 348874526 353362751 369897434 383856495 409412284 433730066 435228820 435654476 437516566 439869070 442576170 445511570 456989936 464512945 478279170 495879074 496397231 504907195 513723902 520938636 523340914 527611073 534140949 534854182 536292567 537339704 543314953 543627350 563610149 567442941 574585727 584286392 589310697 589333180 593544925 602015566 605648920 611244148 611637346 617175571 624371637 625495007 630727609 636077828 636964411 638029599 645783192 652722519 653170059 664085291 668897778 676093305 677726953 694882624 699439611 704405677 709728096 722622723 743905840 762637030 778083653 788419487 793697267 795139875 797278779 797598543 806430018 814289162 830785705 838602018 850429915 856759147 860940403 865400672 869175912 874250444 876037788 880212609 890124883 894096908 904304872 9


```
5     static int cmp_int(const void* a, const void* b){
6     int x = *(const int*)a, y = *(const int*)b;
7     return (x>y) - (x<y);
8     }
9     static void merge_int(const int* A, int nA, const int* B, int nB, int* C){
10    int i=0,j=0,k=0;
11    while(i<nA && j<nB) C[k++] = (A[i]<=B[j]) ? A[i++] : B[j++];
12    while(i<nA) C[k++] = A[i++];
13    while(j<nB) C[k++] = B[j++];
14    }
```

El comparador de datos y la función lineal para 2 listas ya ordenadas.

```
21    if (rank==0 && argc<2){ fprintf(stderr,"Uso: %s <n_total>\n", argv[0]); MPI_Abort(MPI_COMM_WORLD,1); }
22
23    int n = 0;
24    if (rank==0) n = atoi(argv[1]);
25    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

con bcast difunde n en los demás procesos.

```
33    // Genera datos locales pseudoaleatorios, reproducibles por rank
34    unsigned int seed = 123456u ^ (rank*2654435761u) ^ (unsigned int)n;
35    int* local = (int*)malloc(sizeof(int)*local_n);
36    for(int i=0;i<local_n;i++){ seed = seed*1664525u + 1013904223u; local[i] = (int)(seed & 0x7fffffff); }
37
38    // Ordena localmente
39    qsort(local, local_n, sizeof(int), cmp_int);
```

Genera los datos en cada rank y ordena de manera local.

```
56    int curr_n = local_n;
57    for (int step=1; step<p; step<<=1){
58        if ((rank % (2*step)) == 0){
59            int partner = rank + step;
60            if (partner < p){
61                int recv_n=0;
62                MPI_Recv(&recv_n,1,MPI_INT,partner,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
63                int* buf = (int*)malloc(sizeof(int)*recv_n);
64                MPI_Recv(buf,recv_n,MPI_INT,partner,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
65
66                int* merged = (int*)malloc(sizeof(int)*(curr_n + recv_n));
67                merge_int(local, curr_n, buf, recv_n, merged);
68                free(local); free(buf);
69                local = merged; curr_n += recv_n;
70            }
71        }else if ((rank % (2*step)) == step){
72            int partner = rank - step;
73            MPI_Send(&curr_n,1,MPI_INT,partner,0,MPI_COMM_WORLD);
74            MPI_Send(local,curr_n,MPI_INT,partner,0,MPI_COMM_WORLD);
75            break; // este rank ya terminó
76        }
77    }
```

En cada nivel step = 1, 2, 4, … la reducción funciona así: los emisores (los que cumplen rank % (2*step) == step) envían primero su tamaño curr_n y luego su arreglo ordenado al partner, rank – step, y salen del bucle (no vuelven a participar); los receptores (los que cumplen rank % (2*step) == 0) comprueban si existe partner = rank + step, entonces reciben el tamaño y después los datos, fusionan su lista con la recibida en un nuevo buffer ordenado, liberan los buffers antiguos y continúan al siguiente nivel con la lista más grande. Tras completar todos los niveles, el rank 0 habrá recibido y fusionado todo y queda con la lista global ordenada de tamaño n

p=3

- step=1: parejas (0↔1), (2↔3 pero 3 no existe). → 1 envía a 0; 2 no hace nada aún.
- step=2: pareja (0↔2). → 2 envía a 0; 0 recibe y fusiona → 0 queda con todo.