UNIVERSIDAD NACIONAL DE SAN AGUSTÍN AREQUIPA

FACULTAD DE INGENIERÍA DE PRODUCCIÓN DE SERVICIOS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Laboratorio 3

Curso:
Computación Paralela y Distribuída

Alumno:
Pérez Villasante, Jazmín Gabriela

Docente:
EDGAR SARMIENTO CALISAYA

Semestre:
VIII

Arequipa - Perú
2025

Laboratorio 3

EJERCICIO 3.1

```
#include <stdio.h>
#include <mpi.h>

#define TOTAL_DATA_POINTS 20
#define NUMBER_OF_BINS 5

int find_bin(float value, float global_min, float global_max, int bin_count, double bi

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int process_rank, total_processes;
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &total_processes);

    float all_data[] = {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};

    int data_per_process = TOTAL_DATA_POINTS / total_processes;
    float local_data[data_per_process];

    MPI_Scatter(all_data, data_per_process, MPI_FLOAT,
                local_data, data_per_process, MPI_FLOAT,
                0, MPI_COMM_WORLD);

    float local_minimum = local_data[0];
    float local_maximum = local_data[0];
    for (int i = 1; i < data_per_process; i++) {
        if (local_data[i] < local_minimum) local_minimum = local_data[i];
        if (local_data[i] > local_maximum) local_maximum = local_data[i];
    }

    float global_minimum, global_maximum;
    MPI_Allreduce(&local_minimum, &global_minimum, 1, MPI_FLOAT, MPI_MIN, MPI_COMM_WOR
    MPI_Allreduce(&local_maximum, &global_maximum, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WOR
```

```
    double width_per_bin = (global_maximum - global_minimum) / NUMBER_OF_BINS;

    int local_histogram[NUMBER_OF_BINS];
    for (int i = 0; i < NUMBER_OF_BINS; i++) {
        local_histogram[i] = 0;
    }

    for (int i = 0; i < data_per_process; i++) {
        int bin_index = find_bin(local_data[i], global_minimum, global_maximum, NUMBE
        local_histogram[bin_index]++;
    }

    int global_histogram[NUMBER_OF_BINS];
    MPI_Reduce(local_histogram, global_histogram, NUMBER_OF_BINS, MPI_INT, MPI_SUM, 0

    if (process_rank == 0) {
        printf("Histogram Results:\n");
        for (int i = 0; i < NUMBER_OF_BINS; i++) {
            float bin_lower = global_minimum + i * width_per_bin;
            float bin_upper = global_minimum + (i + 1) * width_per_bin;
            printf("Bin %d [%.2f-%.2f): %d values\n", i, bin_lower, bin_upper, global_
        }
    }

    MPI_Finalize();
    return 0;
}

int find_bin(float value, float global_min, float global_max, int bin_count, double bi
    int bin_index = (int)((value - global_min) / bin_width);

    if (bin_index == bin_count) {
        bin_index--;
    }

    return bin_index;
}
```

```
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test11
Histogram Results:
Bin 0 [0.30-1.22): 6 values
Bin 1 [1.22-2.14): 3 values
Bin 2 [2.14-3.06): 2 values
Bin 3 [3.06-3.98): 3 values
Bin 4 [3.98-4.90): 6 values
```

EJERCICIO 3.2

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <mpi.h>
#include <time.h>

int main(int argc, char* argv[]) {
    long long int tosses, local_tosses;
    long long int number_in_circle = 0, local_in_circle = 0;
    int rank, size;

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

    if (rank == 0) {
        printf("Ingrese el número total de lanzamientos: ");
        fflush(stdout);
        scanf("%lld", &tosses);
    }

    MPI_Bcast(&tosses, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);

    local_tosses = tosses / size;

    srand(time(NULL) + rank);

    for (long long int toss = 0; toss < local_tosses; toss++) {
        double x = (double)rand() / RAND_MAX * 2.0 - 1.0;
        double y = (double)rand() / RAND_MAX * 2.0 - 1.0;
        double dist_sq = x * x + y * y;
        if (dist_sq <= 1.0) local_in_circle++;
    }

    MPI_Reduce(&local_in_circle, &number_in_circle, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_

    if (rank == 0) {
        double pi_estimate = 4.0 * (double)number_in_circle / (double)tosses;
        printf("Estimación de pi = %lf\n", pi_estimate);
    }

    MPI_Finalize();
    return 0;
}
```

```
vagrant@master:~$ vim hosts

[1]+  Stopped                 vim hosts
vagrant@master:~$ vim hosts
vagrant@master:~$ vim hosts
vagrant@master:~$ mpirun -np 4 -hostfile hosts ./test2
Ingrese el número total de lanzamientos: 1000000
Estimación de pi = 3.142916
vagrant@master:~$ mpirun -np 4 -hostfile hosts ./test2
Ingrese el número total de lanzamientos: 100000000
Estimación de pi = 3.141584
vagrant@master:~$
```

EJERCICIO 3.3 parte 1

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, comm_sz;
    int local_val, partner, step;
    int global_sum;

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

    local_val = rank + 1;

    global_sum = local_val;

    for (step = 1; step < comm_sz; step *= 2) {
        if (rank % (2*step) == 0) {
            partner = rank + step;
            if (partner < comm_sz) {
                int temp;
                MPI_Recv(&temp, 1, MPI_INT, partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGN
                global_sum += temp;
            }
        } else {
            partner = rank - step;
            MPI_Send(&global_sum, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
            break;
        }
    }

    if (rank == 0) {
        printf("Suma global = %d\n", global_sum);
    }

    MPI_Finalize();
```

```
    return 0;
}
```

```
vagrant@master:~$ vim hosts
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test11
Histogram Results:
Bin 0 [0.30-1.22): 6 values
Bin 1 [1.22-2.14): 3 values
Bin 2 [2.14-3.06): 2 values
Bin 3 [3.06-3.98): 3 values
Bin 4 [3.98-4.90): 6 values
vagrant@master:~$ vim ej3.3.1.c
vagrant@master:~$ mpicc -o test31 ej3.3.1.c
vagrant@master:~$ scp test31 vagrant@192.168.20.11:~
test31                                        100%   17KB   2.4MB/s   00:00
vagrant@master:~$ scp test31 vagrant@192.168.20.12:~
test31                                        100%   17KB   1.7MB/s   00:00
vagrant@master:~$ scp test31 vagrant@192.168.20.13:~
test31                                        100%   17KB   2.3MB/s   00:00
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test31
Suma global = 10
```

EJERCICIO 3.3 parte 2

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    int rank, comm_sz;
    int local_val, partner, step;
    int global_sum;

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

    local_val = rank + 1;
    global_sum = local_val;

    for (step = 1; step < comm_sz; step *= 2) {
        if (rank % (2*step) == 0) {
            partner = rank + step;
            if (partner < comm_sz) {
                int temp;
                MPI_Recv(&temp, 1, MPI_INT, partner, 0, MPI_COMM_WORLD, MPI_STATUS_IGN
                global_sum += temp;
            }
        } else {
            partner = rank - step;
            MPI_Send(&global_sum, 1, MPI_INT, partner, 0, MPI_COMM_WORLD);
            break;
        }
    }
```

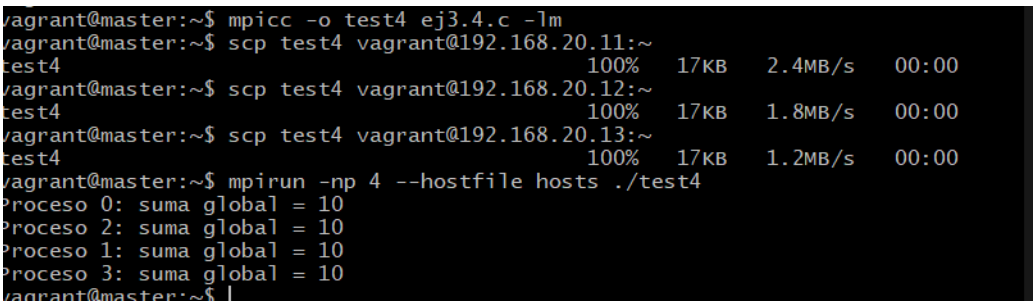
```
    if (rank == 0) {
        printf("Suma global = %d\n", global_sum);
    }

    MPI_Finalize();
    return 0;
}
```

```
vagrant@master:~$ vim ej3.3.2.c
vagrant@master:~$ mpicc -o test32 ej3.3.2.c
vagrant@master:~$ scp test32 vagrant@192.168.20.11:~
test32                                        100%   17KB   1.1MB/s   00:00
vagrant@master:~$ scp test32 vagrant@192.168.20.12:~
test32                                        100%   17KB   1.7MB/s   00:00
vagrant@master:~$ scp test32 vagrant@192.168.20.13:~
test32                                        100%   17KB   1.9MB/s   00:00
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test32
Suma global = 10
```

EJERCICIO 3.4

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

int main(int argc, char** argv) {
    int rank, comm_sz;
    int local_val, partner, step;
    int recv_val;

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

    local_val = rank + 1;

    int p = 1;
    while (p * 2 <= comm_sz) p *= 2;

    if (rank >= p) {
        MPI_Send(&local_val, 1, MPI_INT, rank - p, 0, MPI_COMM_WORLD);

        MPI_Finalize();
        return 0;
    }

    if (rank + p < comm_sz) {
        MPI_Recv(&recv_val, 1, MPI_INT, rank + p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
        local_val += recv_val;
```

```
    }

    int log_p = (int)log2(p);
    for (step = 0; step < log_p; step++) {
        int distance = 1 << step;
        partner = rank ^ distance;

        MPI_Sendrecv(&local_val, 1, MPI_INT, partner, 0,
                     &recv_val, 1, MPI_INT, partner, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        local_val += recv_val;
    }

    printf("Proceso %d: suma global = %d\n", rank, local_val);

    MPI_Finalize();
    return 0;
}
```



EJERCICIO 3.5

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

int main(int argc, char** argv) {
    int rank, size;
    int n = 4;
    double *A = NULL, *x = NULL;
    double *local_A, *local_x, *local_y, *y;
    int local_cols;

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

```
if (n % size != 0) {
    if (rank == 0) printf("n debe ser divisible por el numero de procesos\n");
    MPI_Finalize();
    return 0;
}

local_cols = n / size;
if (rank == 0) {
    A = malloc(n * n * sizeof(double));
    x = malloc(n * sizeof(double));
    y = malloc(n * sizeof(double));

    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i*n + j] = (i == j) ? 1.0 : 0.0;

    for (int i = 0; i < n; i++)
        x[i] = i + 1;
}

// Reservar espacio local
local_A = malloc(n * local_cols * sizeof(double));
local_x = malloc(local_cols * sizeof(double));
local_y = malloc(n * sizeof(double));

// Distribuir bloques de columnas de A
MPI_Scatter(A, n*local_cols, MPI_DOUBLE,
            local_A, n*local_cols, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

MPI_Scatter(x, local_cols, MPI_DOUBLE,
            local_x, local_cols, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

for (int i = 0; i < n; i++)
    local_y[i] = 0.0;

for (int i = 0; i < n; i++) {
    for (int j = 0; j < local_cols; j++) {
        local_y[i] += local_A[i*local_cols + j] * local_x[j];
    }
}

int *recvcounts = malloc(size * sizeof(int));
for (int i = 0; i < size; i++) recvcounts[i] = n / size;

double *local_result = malloc(n/size * sizeof(double));
```

```
    MPI_Reduce_scatter(local_y, local_result, recvcounts,
                       MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    MPI_Gather(local_result, n/size, MPI_DOUBLE,
               y, n/size, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Resultado y = [ ");
        for (int i = 0; i < n; i++) printf("%.1f ", y[i]);
        printf("]\n");
    }

    free(local_A); free(local_x); free(local_y);
    if (rank == 0) { free(A); free(x); free(y); }
    free(local_result); free(recvcounts);

    MPI_Finalize();
    return 0;
}
```

```
vagrant@master:~$ vim ej3.5.c
vagrant@master:~$ mpicc -o test5 ej3.5.c
vagrant@master:~$ scp test5 vagrant@192.168.20.11:~
test5                                         100%   17KB   1.7MB/s   00:00
vagrant@master:~$ scp test5 vagrant@192.168.20.12:~
test5                                         100%   17KB   1.4MB/s   00:00
vagrant@master:~$ scp test5 vagrant@192.168.20.13:~
test5                                         100%   17KB   1.7MB/s   00:00
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test5
Resultado y = [ 1.0 2.0 3.0 4.0 ]
vagrant@master:~$
```

EJERCICIO 3.6

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (argc < 2) {
        if (rank==0) fprintf(stderr, "Uso: mpirun -np P ./matvec_block_submatrix N\n"
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
```

```
int n = atoi(argv[1]);
int q = (int)round(sqrt((double)size));
if (q*q != size) {
    if (rank==0) fprintf(stderr, "El número de procesos debe ser un cuadrado perfe
    MPI_Abort(MPI_COMM_WORLD, 2);
}
if (n % q != 0) {
    if (rank==0) fprintf(stderr, "n debe ser divisible por sqrt(comm_sz)=%d\n", q)
    MPI_Abort(MPI_COMM_WORLD, 3);
}

int rows_per_block = n / q;
int r = rank / q;
int c = rank % q;

double *Ablock = malloc(rows_per_block * rows_per_block * sizeof(double));
double *xseg = malloc(rows_per_block * sizeof(double));
double *y_partial = calloc(rows_per_block, sizeof(double));
double *y_block = NULL;

if (rank == 0) {
    double *A = malloc(n * n * sizeof(double));
    double *x = malloc(n * sizeof(double));
    for (int i = 0; i < n; ++i) {
        x[i] = (double)(i+1);
        for (int j = 0; j < n; ++j) {
            A[i*n + j] = (double)(i*n + j + 1);
        }
    }

    for (int pr = 0; pr < q; ++pr) {
        for (int pc = 0; pc < q; ++pc) {
            int dest = pr * q + pc;
            double *tmp = malloc(rows_per_block * rows_per_block * sizeof(double))
            for (int i = 0; i < rows_per_block; ++i) {
                int global_i = pr * rows_per_block + i;
                for (int j = 0; j < rows_per_block; ++j) {
                    int global_j = pc * rows_per_block + j;
                    tmp[i*rows_per_block + j] = A[global_i * n + global_j];
                }
            }
            if (dest == 0) {
                for (int i=0;i<rows_per_block*rows_per_block;++i) Ablock[i]=tmp[i]
            } else {
                MPI_Send(tmp, rows_per_block*rows_per_block, MPI_DOUBLE, dest, 0,
            }
            free(tmp);
```

```
        }
    }

    for (int pc = 0; pc < q; ++pc) {
        int dest = pc * q + pc;
        double *x_tmp = &x[pc * rows_per_block];
        if (dest == 0) {
            for (int i=0;i<rows_per_block;++i) xseg[i] = x_tmp[i];
        } else {
            MPI_Send(x_tmp, rows_per_block, MPI_DOUBLE, dest, 1, MPI_COMM_WORLD);
        }
    }

    free(A);
    free(x);
} else {
    MPI_Recv(Ablock, rows_per_block*rows_per_block, MPI_DOUBLE, 0, 0, MPI_COMM_WOR
    if (r == c) {
        MPI_Recv(xseg, rows_per_block, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATU
    }
}

MPI_Comm col_comm, row_comm;
MPI_Comm_split(MPI_COMM_WORLD, c, r, &col_comm);
MPI_Comm_split(MPI_COMM_WORLD, r, c, &row_comm);

int col_root = c;
MPI_Bcast(xseg, rows_per_block, MPI_DOUBLE, col_root, col_comm);

for (int i = 0; i < rows_per_block; ++i) {
    double sum = 0.0;
    for (int j = 0; j < rows_per_block; ++j) {
        sum += Ablock[i*rows_per_block + j] * xseg[j];
    }
    y_partial[i] = sum;
}

int row_root = r;
if (r == c) {
    y_block = calloc(rows_per_block, sizeof(double));
}
MPI_Reduce(y_partial, y_block, rows_per_block, MPI_DOUBLE, MPI_SUM, row_root, row_

if (r == c) {
    int diag_rank = rank;
    if (diag_rank == 0) {
        double *y = malloc(n * sizeof(double));
        for (int i=0;i<rows_per_block;++i) y[i] = y_block[i];
```

```
            for (int pr = 1; pr < q; ++pr) {
                int src = pr * q + pr;
                MPI_Recv(&y[pr*rows_per_block], rows_per_block, MPI_DOUBLE, src, 2, MP
            }
            if (rank == 0) {
                printf("Resultado y = A*x (n=%d):\n", n);
                for (int i = 0; i < n; ++i) printf("%g ", y[i]);
                printf("\n");
            }
            free(y);
        } else {
            MPI_Send(y_block, rows_per_block, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
        }
    }

    free(Ablock);
    free(xseg);
    free(y_partial);
    if (y_block) free(y_block);
    MPI_Comm_free(&col_comm);
    MPI_Comm_free(&row_comm);
    MPI_Finalize();
    return 0;
}
```

```
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test6 6
Resultado y = A*x (n=6):
91 217 343 469 595 721
vagrant@master:~$
```

EJERCICIO 3.7

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

#define NITER 10000
int main(int argc, char** argv) {
    int rank, size;
    int msg = 0;
    MPI_Status status;

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

```
    if (size != 2) {
        if (rank == 0) {
            printf("Este programa necesita exactamente 2 procesos\n");
        }
        MPI_Finalize();
        return 0;
    }

    clock_t start_c, end_c;
    if (rank == 0) {
        start_c = clock();
        for (int i = 0; i < NITER; i++) {
            msg = i;
            MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            MPI_Recv(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        }
        end_c = clock();
        double tiempo_clock = ((double)(end_c - start_c)) / CLOCKS_PER_SEC;
        printf("Tiempo con clock(): %f segundos\n", tiempo_clock);
    } else {
        for (int i = 0; i < NITER; i++) {
            MPI_Recv(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0) {
        double start_mpi = MPI_Wtime();
        for (int i = 0; i < NITER; i++) {
            msg = i;
            MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            MPI_Recv(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        }
        double end_mpi = MPI_Wtime();
        printf("Tiempo con MPI_Wtime(): %f segundos\n", end_mpi - start_mpi);
    } else {
        for (int i = 0; i < NITER; i++) {
            MPI_Recv(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            MPI_Send(&msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    }

    MPI_Finalize();
    return 0;
}
```

```
vagrant@master:~$ vim hosts
vagrant@master:~$ mpirun -np 2 --hostfile hosts ./test7
Tiempo con clock(): 13.913131 segundos
Tiempo con MPI_Wtime(): 14.333295 segundos
vagrant@master:~$
```

EJERCICIO 3.8

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

void merge(int* A, int sizeA, int* B, int sizeB, int* C) {
    int i=0, j=0, k=0;
    while (i < sizeA && j < sizeB) {
        if (A[i] <= B[j]) C[k++] = A[i++];
        else C[k++] = B[j++];
    }
    while (i < sizeA) C[k++] = A[i++];
    while (j < sizeB) C[k++] = B[j++];
}

int cmpfunc(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int main(int argc, char** argv) {
    int rank, size, n;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        printf("Ingrese n (tamaño total, divisible por %d): ", size);
        fflush(stdout);
        scanf("%d", &n);
    }

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int local_n = n / size;
    int* local = (int*) malloc(local_n * sizeof(int));

    srand(rank + 1);
    for (int i = 0; i < local_n; i++) {
        local[i] = rand() % 100;
    }

    qsort(local, local_n, sizeof(int), cmpfunc);
```

```
if (rank == 0) {
    int* temp = (int*) malloc(n * sizeof(int));
    for (int i = 0; i < local_n; i++)
        temp[i] = local[i];

    for (int p = 1; p < size; p++) {
        MPI_Recv(temp + p*local_n, local_n, MPI_INT, p, 0, MPI_COMM_WORLD, MPI_STA
    }

    printf("Listas locales ordenadas:\n");
    for (int i = 0; i < n; i++) printf("%d ", temp[i]);
    printf("\n\n");
    free(temp);
} else {
    MPI_Send(local, local_n, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

int step = 1;
int* merged = local;
int merged_size = local_n;

while (step < size) {
    if (rank % (2*step) == 0) {
        if (rank + step < size) {
            int recv_size = merged_size;
            int* recv_data = (int*) malloc(recv_size * sizeof(int));

            MPI_Recv(&recv_size, 1, MPI_INT, rank+step, 0, MPI_COMM_WORLD, MPI_STA
            recv_data = (int*) malloc(recv_size * sizeof(int));
            MPI_Recv(recv_data, recv_size, MPI_INT, rank+step, 0, MPI_COMM_WORLD,

            int* new_merge = (int*) malloc((merged_size+recv_size)*sizeof(int));
            merge(merged, merged_size, recv_data, recv_size, new_merge);

            if (merged != local) free(merged);
            free(recv_data);
            merged = new_merge;
            merged_size += recv_size;
        }
    } else {
        int dest = rank - step;
        MPI_Send(&merged_size, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
        MPI_Send(merged, merged_size, MPI_INT, dest, 0, MPI_COMM_WORLD);
        break;
    }
    step *= 2;
}
```

```
    if (rank == 0) {
        printf("Lista global ordenada:\n");
        for (int i = 0; i < merged_size; i++) printf("%d ", merged[i]);
        printf("\n");
    }

    if (merged != local) free(merged);
    else free(local);

    MPI_Finalize();
    return 0;
}
```

```
vagrant@master:~$ scp test8 vagrant@192.168.20.11:~
test8                                         100%   17KB   1.9MB/s   00:00
vagrant@master:~$ scp test8 vagrant@192.168.20.12:~
test8                                         100%   17KB   1.7MB/s   00:00
vagrant@master:~$ scp test8 vagrant@192.168.20.13:~
test8                                         100%   17KB   1.7MB/s   00:00
vagrant@master:~$ mpirun -np 4 --hostfile hosts ./test8
Ingrese n (tamaño total, divisible por 4): 20
Listas locales ordenadas:
15 77 83 86 93 19 61 75 88 90 25 40 46 68 85 1 26 63 74 83

Lista global ordenada:
1 15 19 25 26 40 46 61 63 68 74 75 77 83 83 85 86 88 90 93
```

EJERCICIO 3.9

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

int main(int argc, char** argv) {
    int rank, size;
    int n = 16;
    double start, end;

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

    if (n % size != 0) {
        if (rank == 0) printf("El tamaño del vector debe ser divisible por el número d
        MPI_Finalize();
        return 0;
    }

    int local_n = n / size;
    int* block_data = (int*) malloc(local_n * sizeof(int));
```

```
for (int i = 0; i < local_n; i++) {
    block_data[i] = rank * local_n + i;
}

int* cyclic_data = (int*) malloc(local_n * sizeof(int));
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

for (int i = 0; i < local_n; i++) {
    int global_index = rank * local_n + i;
    int dest = global_index % size;
    int pos  = global_index / size;
    if (dest == rank) {
        cyclic_data[pos] = block_data[i];
    } else {
        MPI_Send(&block_data[i], 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
    }
}
for (int i = 0; i < local_n; i++) {
    int global_index = i * size + rank;
    if (global_index < n) {
        MPI_Recv(&cyclic_data[i], 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, M
    }
}

MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();
if (rank == 0) printf("Tiempo BLOCK -> CYCLIC = %f segundos\n", end - start);

int* block_data2 = (int*) malloc(local_n * sizeof(int));
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

for (int i = 0; i < local_n; i++) {
    int global_index = i * size + rank;
    int dest = global_index / local_n; int pos  = global_index % local_n;
    if (dest == rank) {
        block_data2[pos] = cyclic_data[i];
    } else {
        MPI_Send(&cyclic_data[i], 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
    }
}
for (int i = 0; i < local_n; i++) {
    int global_index = rank * local_n + i;
    MPI_Recv(&block_data2[i], 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_S
}
```

```
    MPI_Barrier(MPI_COMM_WORLD);
    end = MPI_Wtime();
    if (rank == 0) printf("Tiempo CYCLIC -> BLOCK = %f segundos\n", end - start);


    /*
    for (int p = 0; p < size; p++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (rank == p) {
            printf("Rank %d final data: ", rank);
            for (int i = 0; i < local_n; i++) printf("%d ", block_data2[i]);
            printf("\n");
        }
    }
    */

    free(block_data);
    free(cyclic_data);
    free(block_data2);

    MPI_Finalize();
    return 0;
}
```

1. Vagrantfile

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/focal64"

  # Configuración global reducida para máquinas con pocos recursos
  config.vm.provider "virtualbox" do |vb|
    vb.memory = 500   # 500 MB de RAM
    vb.cpus = 1       # 1 CPU
  end

  # Script común: instala MPI
  $install_mpi = <<-SHELL
    sudo apt-get update -y
    sudo apt-get install -y build-essential openmpi-bin openmpi-common libopenmpi-dev
  SHELL

  # Script para master: genera clave y la comparte
  $master_setup = <<-SHELL
    # Generar clave si no existe
    if [ ! -f /home/vagrant/.ssh/id_rsa ]; then
      ssh-keygen -t rsa -b 2048 -f /home/vagrant/.ssh/id_rsa -q -N ""
      chown vagrant:vagrant /home/vagrant/.ssh/id_rsa*
```

```
    fi
    # Guardar clave pública en carpeta compartida
    cp /home/vagrant/.ssh/id_rsa.pub /vagrant/master_key.pub
  SHELL

  # Script para workers: agregar clave del master
  $worker_setup = <<-SHELL
    if [ -f /vagrant/master_key.pub ]; then
      mkdir -p /home/vagrant/.ssh
      cat /vagrant/master_key.pub >> /home/vagrant/.ssh/authorized_keys
      chown -R vagrant:vagrant /home/vagrant/.ssh
      chmod 600 /home/vagrant/.ssh/authorized_keys
    fi
  SHELL

  # MASTER
  config.vm.define "master" do |master|
    master.vm.hostname = "master"
    master.vm.network "private_network", ip: "192.168.20.10"
    master.vm.provision "shell", inline: $install_mpi
    master.vm.provision "shell", inline: $master_setup, run: "always"
  end

  # CPU 1
  config.vm.define "cpu1" do |cpu|
    cpu.vm.hostname = "cpu1"
    cpu.vm.network "private_network", ip: "192.168.20.11"
    cpu.vm.provision "shell", inline: $install_mpi
    cpu.vm.provision "shell", inline: $worker_setup, run: "always"
  end

  # CPU 2
  config.vm.define "cpu2" do |cpu|
    cpu.vm.hostname = "cpu2"
    cpu.vm.network "private_network", ip: "192.168.20.12"
    cpu.vm.provision "shell", inline: $install_mpi
    cpu.vm.provision "shell", inline: $worker_setup, run: "always"
  end

  # CPU 3
  config.vm.define "cpu3" do |cpu|
    cpu.vm.hostname = "cpu3"
    cpu.vm.network "private_network", ip: "192.168.20.13"
    cpu.vm.provision "shell", inline: $install_mpi
    cpu.vm.provision "shell", inline: $worker_setup, run: "always"
  end
end
```