Júlia López and Tània Pazos
u214964, u185115

Lab 2: MPI

Communication basics

1. Complete the Send/Receive blocks in the test_synchronous function and the correct computation of the ranks to whom send/receive.

As the statement indicates, each processor must send data -its rank- to the next processor in a circular manner, and receive data from the previous processor. To achieve this circular pattern, the send core and receive core -ranks of the processor to send data to and the processor to receive data from respectively- are computed as follows.

```
int send_core = (rank+1)%size;
int recv_core = (rank + size -1)%size;
```

The formula (rank+1)%size ensures that the rank to send data to is 0 for the last process, which has rank size-1.
Similarly, the expression (rank+size-1)%size ensures that the rank to receive data from is size-1 for the first process, which has rank 0.

Regarding the send and receive operations, we use the functions MPI_Send and MPI_Recv. Since both operations are blocking, the sender will wait until the receiver acknowledges the receipt of the message to continue.

```
        MPI_Send(&sbuf[i], 1, MPI_INT, send_core, 0, MPI_COMM_WORLD);
        MPI_Recv(&rbuf[i], 1, MPI_INT, recv_core, 0, MPI_COMM_WORLD, &status);
```

MPI_Send sends the data in sbuf[i], a unique integer representing its rank, to the next processor send_core with a tag of 0. MPI_Recv receives data from the previous processor recv_core and stores it into rbuf[i] with the same tag 0.
After receiving the data, each processor verifies if the received integer is indeed the rank of the sender using the function assert.

```
        assert(rbuf[i] == recv_core);
```

2. Use collective communication routines to compute the statistics required for the computing time

To compute the average, maximum and minimum time spent by each processor to perform the send-receive operations we first use the collective communication routine MPI_Allreduce, which adds the individual running times of each process. This sum is divided

by the number of processes (size) to compute the average time. We perform the same routine MPI_Allreduce to calculate the maximum and minimum running time.

```
MPI_Allreduce(&run_time, &average_time, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   average_time /= size;
   MPI_Allreduce(&run_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
   MPI_Allreduce(&run_time, &min_time, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
```

It is worth saying that only the root processor prints the statistics.

```
if (rank == 0) {
      // Print statistics
}
```

3. Complete the Send/Receive blocks in the test asynchronous function. Analyze and comment on the differences you observe between this and the first question.

In synchronous communication, MPI_Send and MPI_Recv are blocking functions, meaning they do not return control to the calling process until the operation has finished. However, in asynchronous communication, we use non-blocking variants of these functions (MPI_Isend and MPI_Irecv) that return immediately without waiting for the operation to finish. As a consequence, the calling process needs to keep track of the progress and completion of the operations performed, and this is achieved with requests. A request object MPI_Request stores information about the ongoing communication like the status of the operation. Hence, arrays of MPI_Request objects (send _requests and recv_requests) are created to store the requests for sending and receiving data. To store specific information of each receiving operation such as the source rank, tag, error code and count an array of MPI_Status objects is also declared.

```
MPI_Request send_requests[NUM_TEST], recv_requests[NUM_TEST];
MPI_Status recv_statuses[NUM_TEST];
```

The rank of the process to send data to and to receive data from is computed exactly the same way as before, but now -as stated above- non-blocking functions are used: MPI_Isend and MPI_Irecv. Both functions simply initiate the sending and the receiving of data respectively, but do not wait for their completion.

```
MPI_Isend(sbuf[i], 1, MPI_INT, send_core, 0, MPI_COMM_WORLD, &send_requests[i]);
MPI_Irecv(&buf[i], 1, MPI_INT, recv_core, 0, MPI_COMM_WORLD, &recv_requests[i]);
```

Before proceeding, we use the routine MPI_Waitall to ensure that all data has been sent and received successfully.

```
MPI_Waitall(NUM_TEST, send_requests, MPI_STATUSES_IGNORE);
MPI_Waitall(NUM_TEST, recv_requests, recv_statuses);
```

Note that this routine was not necessary in synchronous communication, since the blocking functions MPI_Send and MPI_Recv already implicitly ensured synchronization.

The computation and printing of the average, maximum and minimum running time is done exactly in the same way as in synchronous communication.

Figure 1 shows the results obtained after executing mpi_comms.c with 4 processes.

```
Synchronous send/receive test with 4 processes and 50 repetitions.
    average: 12.52 s
    min:     12.05 s
    max:     13.00 s
Asynchronous send/receive test with 4 processes and 50 repetitions.
    average: 1.01 s
    min:     1.00 s
    max:     1.01 s
```

Figure 1: Average, minimum and maximum running time of synchronous and asynchronous communication with 4 processes.

In a nutshell, it is clear that there is a significant difference in running time between synchronous and asynchronous communication. On the one hand, in synchronous communication each process must wait for the send and receive operations to finish before proceeding to the next step. As a result, communication times increase. On the other hand, asynchronous communication allows for processes to overlap communication with computation. Hence, processes do not have to block while waiting for communication to finish, and thus running times are notably lower.

Numerical integration: definite

1. Analyze the sequential version and explain how it can be parallelized.

In order to parallelize the sequential code, the workload -approximation of the definite integral using the midpoint algorithm- will be distributed to different processes.
In other words, the number of subintervals N -representing the first argument- will be divided among all processes so that each process will be responsible for the computation of a portion of the N subintervals. Each process will compute the local sum -area under $f(x)$- in its assigned intervals implementing the midpoint algorithm.
Once all processes have finished computed the local sums, we will use the MPI_SUM reduction operation to combine all local sums and retrieve the final value from the root process. The root process -with rank 0- will be the only process to compute the error and print the result, error and running time.

2. Create a parallel version of the program in a file called def integral par.c. Notice that you need to respect both the input and the output of the program.

To build the parallel version, the following key parts of the sequential version were added or modified.
Of course, the MPI environment is initialized using the appropriate function.

```
MPI_Init(&argc, &argv);
```

Then, the rank of the current process and the total number of processes inside our communicator MPI_COMM_WORLD is obtained using MPI_Comm_rank and MPI_Comm_size.

```
int rank, nproc;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

In order to divide the workload, the iterations of the loop - that go from 1 to N- that computes the area of the subrectangles are divided among the nproc processes. To achieve so, the first subrectangle assigned to a process is computed according to its rank. What is more, to ensure that each process handles a disjoint set of rectangles, the loop variable i is incremented by the number of processes.

```
for (i = rank+1; i <= N; i+= nproc) {
    double x_middle = (i - 0.5) * deltaX;
    local_result = local_result + function(x_middle) * deltaX;
}
```

To show that this strategy is correct, let us provide a concrete example with 10 intervals to compute the integral and 4 processes with ranks from 0, 1, 2 and 3. The starting interval for each process will be i=1 for process 0, i=2 for process 1, i=3 for process 2 and i=4 for process 3. After each iteration, variable i is incremented by the number of processes, hence the next interval for each process will be i=5 for process 0, i=6 for process 1, i=7 for process 2 and i=8 for process 3. This will continue until i exceeds the number of intervals, which is 10 for this specific example.

After the loop, each process will have the sum of the areas of its assigned intervals stored into the variable local_result. To sum all local results into the global variable total_result, we use the function MPI_Reduce.

```
  double total_result;
  MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Finally, only the root process computes the value of the exact integral, calculates the error and prints the results.

```
   if (rank == 0) {
       // Computations and printing results
   }
```

Before finishing, the code closes the MPI environment using MPI_Finalize.

3. Analyze how the error in the integration varies for large values of the computational domain L > 1000 and the need for a large number of points.

```
Sequential result with N=10 is -9868942.716880813241 (9521.248068201259, 9.88e+06) in 0.000018 seconds
Sequential result with N=100 is -1825204.130307016661 (9521.248068201259, 1.83e+06) in 0.000034 seconds
Sequential result with N=1000 is -49649.414232172647 (9521.248068201259, 5.92e+04) in 0.000038 seconds
Parallel result with N=10 is -9868942.716880813241 (9521.248068201259, 9.88e+06) in 0.003711 seconds
Parallel result with N=100 is -1825204.130307016894 (9521.248068201259, 1.83e+06) in 0.005445 seconds
Parallel result with N=1000 is -49649.414232172625 (9521.248068201259, 5.92e+04) in 0.000291 seconds
```

Figure 2: Results for N=10, N=100 and N=1000 with a computational domain L=10000.

```
Sequential result with N=10 is -505624958.718242883682 (99936.116492619214, 5.06e+08) in 0.000035 seconds
Sequential result with N=100 is -106857503.029289782047 (99936.116492619214, 1.07e+08) in 0.000021 seconds
Sequential result with N=1000 is -19043270.622324097902 (99936.116492619214, 1.91e+07) in 0.000040 seconds
Parallel result with N=10 is -505624958.718243002892 (99936.116492619214, 5.06e+08) in 0.001187 seconds
Parallel result with N=100 is -106857503.029289796948 (99936.116492619214, 1.07e+08) in 0.024270 seconds
Parallel result with N=1000 is -19043270.622324075550 (99936.116492619214, 1.91e+07) in 0.000216 seconds
```

Figure 3: Results for N=10, N=100 and N=1000 with a computational domain L=1000000.

First and foremost, it should be noted that both sequential and parallel implementations provide a value of the integral that deviates from the exact value.

Secondly, the more points we divide the x domain into, i.e. the larger N is, the more accurate the approximation of the integral. Indeed, for a computational domain of L = 10000, the error decreases from 9.88x10^6 (N=10) to 5.92x10^4 (N=1000).
The effect of the computational domain L on the error is also worth being mentioned. For larger values of the computational domain, the error increases noticeably. For instance, for L=10000, the error with 1000 points is 5.92x10^4, whereas for L=1000000, the error with 1000 points is 1.91x10^7. This suggests that the value of the computational domain has a greater impact on the accuracy of the integration than the number of points used in the discretization of the domain.
Last but not least, the parallel implementation accounts for longer execution times than the sequential implementation. Nevertheless, the difference between the runtime of the sequential version and the parallel version is smaller for larger values of N. These longer execution times of the parallel code can be explained by the overhead associated with parallelization.

4. Analyze the strong scaling for $N=10^9$ and L=1000.

```
Sequential result with N=1000000000 is -561.552196750522 (-561.552196750171, 3.51e-10) in 18.135656 seconds
Parallel result with N=1000000000 is -561.552196750191 (-561.552196750171, 2.02e-11) in 7.334645 seconds
```

Figure 4: Strong scaling results for $N=10^9$ and L=1000.

Opposite to the previous comment, for N large enough the parallel execution with MPI accounts for a remarkably smaller execution time compared to the sequential execution. Indeed the computation time decreases from approximately 18.14 seconds in the sequential implementation to 7.33 seconds in the parallel version.
Furthermore, the error for both sequential and parallel implementations is now minimal, with errors on the order of 10^-10 for the sequential case and 10^-11 for the parallel case.
Finally, we can compute the speedup as

$$speedup = \frac{sequential\ time}{parallel\ time} = \frac{18.135656}{7.334645} \approx 2.47$$

which indicates good strong scaling behavior.

Numerical integration: indefinite

1. Analyze the sequential version and explain the strategy to parallelize it.

The provided sequential version computes the indefinite integral of a function f(x) using numerical integration with the midpoint rule. First of all we have two additional functions which are:

```
double function(double x){
   return x*sin(x);
}
double exact_integral(double x){
   return sin(x) - x*cos(x);
}
```

The first one is the function to be integrated whereas the second one computes the exact integral value in range [0, x] which will come in handy when we want to see the comparison.

Moving on to the main part of the code, this program takes two command line arguments: N and XMAX. N is in the number of intervals we will split the function in order to compute their integral values which will determine the number of iterations and XMAX represents the upper limit of integration range, that is, the integration domain. Given we have two command line arguments, this block of code checks if the number of arguments is correct:

```
   /// Check parameters
   if(argc != 3) {
       fprintf(stderr, "Wrong number of parameters\nUsage:\n\t%s N XMAX\n",
argv[0]);
       return -1;
   }
```

Before computing the integral, the width of each integration interval deltaX and a dynamically allocated array to store the integral values at each interval are defined. The array has size N+1 to store the integral value at the upper limit of integration.

```
double deltaX = X_MAX/(double) N; // compute deltaX (width of each integration
interval)
double* integral = (double *) calloc(N+1, sizeof(double)); // array to store
integral values at each interval
```

To compute the integral the program performs a loop over N interactions computing each integral over the N intervals using the midpoint rule:

```
   for (i = 1; i <= N; i++){ //calculates integral at each interval
       double x = (i-0.5)*deltaX;
       integral[i] = integral[i-1] + deltaX*function(x); //store it int array
```

```
    }
```

Here, each entry of the dynamically allocated array integral takes the value of the previous entry plus the current interval's contribution, deltaX * function(x). This creates a dependency between intervals, which will need to be broken for parallelization. Before starting this computation we initialize a timer which when the integral is completed we will end.

Finally, it defines the definite result as the value stored in integral[N] and computes the error to see the difference. It also computes the running time of the computation of the integral with the timer we previously initialized. The code ends with the output of two files “indef_integral_seq.data” and “indef_integral_seq.info” and with the deallocation of space of the integral array.

The strategy for parallelizing the code using MPI involves distributing the computation of the integral across multiple processes. Each process will compute a part of the integral, and then the results will be combined using MPI reduction. In order to do this, the key point is the loop iterations which will have to be distributed among the processes. Each process will compute the sum of integrals of their intervals independently and will all be combined later on in a single sum using the MPI_Reduce routine.

2. Create a parallel version of the program in a file called indef_integral_par.c. Notice that you need to respect both the input and the output of the program. Include plots generated with the python program for different input parameters.

The parallel code we created is based on the provided sequential code, now we will describe the main differences between them and a more detailed explanation on our parallelization approach.

First of all in the beginning of the main function we initialize an MPI environment. We get the rank and the size of the processes of the communicator MPI_COMM_WORLD which encapsulates all of the processes in the current program.

```
MPI_Init(&argc, &argv); //we initialize MPI execution environment

//we get the rank of the current process and the total number of processes
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Again we check the correct command line arguments which have not changed however, if they are not correct we exit the MPI environment with MPI_FInalize().

To perform the integral computation as previously mentioned, we had to design the loop so that each independent process computes the integrals over some intervals without taking into account the previous interval. Our approach was the following:

```
double local_integral = 0.0; // local_integral is a local variable allocated by
each of the MPI processes
int i;
for (i = rank + 1; i <= N; i+= size) {
    double x = (i - 0.5) * deltaX;
    local_integral += deltaX * function(x);
}

double global_integral = 0.0; //global_integral will be the total real value of the
resulting integral
MPI_Reduce(&local_integral, &global_integral, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Here we can see each process will have a local_integral variable and perform the integral computation on some intervals defined by their rank, number of total processes and number of intervals. That is, if N = 10 and size = 4, then the process of rank 0 will compute iterations i=1, i=5, and i=9.

Then, we defined a global_integral value to store the sum of all the local_integral variables of the processes. To do so, we use MPI_Reduce which sums all of the local variables into global_integral in a single destination process which for simplicity is the process with rank 0.

Finally we will output the same two files (“indef_integral_par.dat” and “indef_integral_par.info”) but only one of the processes needs to do it hence we implemented that only the root process (rank 0 process) executes that block of code.

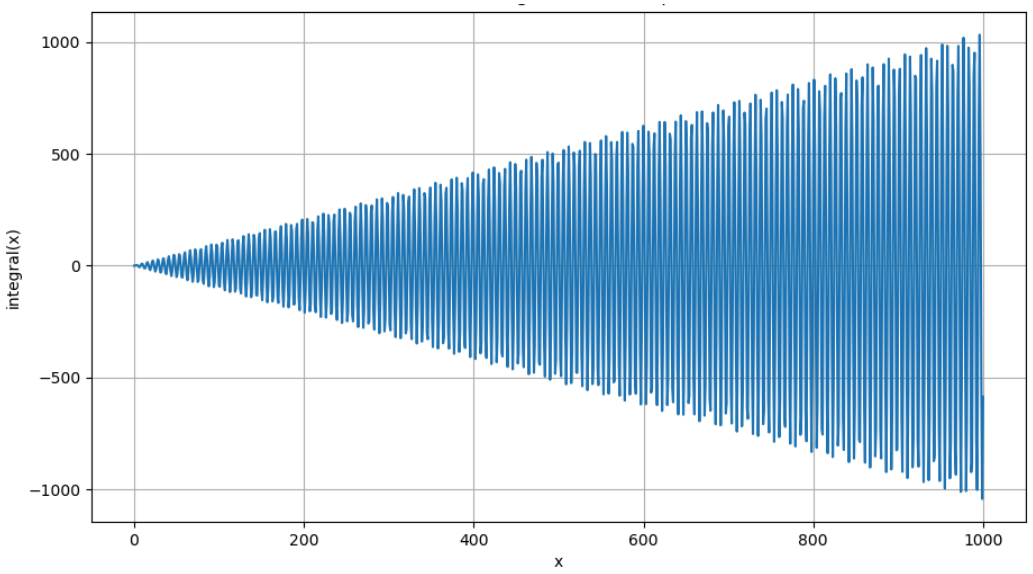

Figure 5: Plot of the indefinite integral for N=1000 and L=1000.

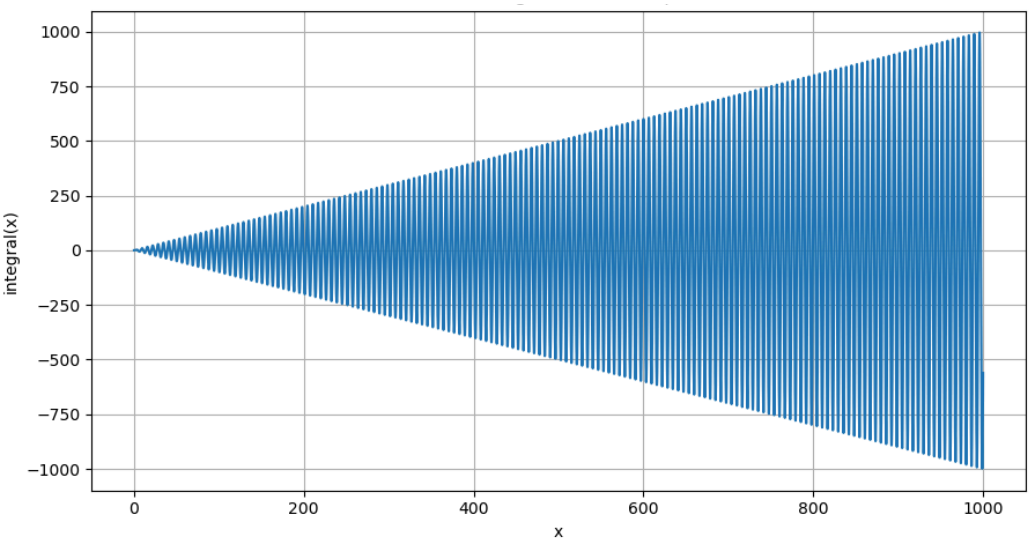

Figure 5: Plot of the indefinite integral for N=10^8 and L=1000.

As illustrated in Figure 5 and 6, the value of L determines the computational domain of the integral, whereas the value of N represents the number of subdomains that we divide the domain into.

3. Analyze the strong scaling for N=10^8 and L=1000.

```
Result with N=100000000 is -561.552196752452 (-561.552196750171, 2.28e-09) in 1.747975 seconds
Result with N=100000000 is -561.552196752408 (-561.552196750171, 2.24e-09) in 0.919575 seconds
Result with N=100000000 is -561.552196752464 (-561.552196750171, 2.29e-09) in 0.456412 seconds
Result with N=100000000 is -561.552196752462 (-561.552196750171, 2.29e-09) in 0.257895 seconds
Result with N=100000000 is -561.552196752435 (-561.552196750171, 2.26e-09) in 0.186945 seconds
Result with N=100000000 is -561.552196752440 (-561.552196750171, 2.27e-09) in 0.117492 seconds
```

Figure 7: Strong scaling results for 1, 2, 4, 8, 16 and 32 number of tasks respectively.

As Figure 7 shows, while increasing the number of tasks does not significantly change the value of the integral nor the computation error, it does have a strong effect on the execution time. Indeed, as the number of tasks increases from 1 to 32, the error reduces from approximately 1.7 to 0.12.

4. What differences do you observe with the definite integral exercise?

On the one hand, although both exercises compute integrals applying MPI parallelization, the computation of an indefinite integral is more complex since it depends on running sums. What is more, the accuracy of the computation must be maintained among all the processes involved.
On the other hand, the computation of a definite integral involves a simpler MPI parallelization, since the sum of the function values is made over a fixed interval.
Apart from differences in the parallelization strategy and implementation complexity, the two exercises also defer on how to manage the error. Exercise 2 focuses on minimizing the integration error through increasing N (dividing the computational domain into more subdomains). However, exercise 3 must manage error propagation since there are dependencies between subintervals.

Hybrid programming: integration of a 2D surface

1. Start with the MPI parallelization. Explain what is the strategy to parallelize, especially how the data is going to be shared among ranks.

The main strategy for MPI parallelization we used is that each process handles a subset of the grid rows and computes the partial integral for the assigned rows. We firstly initialize the MPI environment:

```
/// TODO Init MPI
int rank, size;
int provided;

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

Next as in previous programs, we check the correct number of command-line arguments which are the filename of the input file, and Nx and Ny which are the number of grid points in each dimension x and y respectively.

```
/// Check parameters
if(argc != 4) {
    fprintf(stderr, "Wrong number of parameters\nUsage:\n\t%s filename Nx Ny\n",
argv[0]);
    MPI_Finalize();
    return -1;
}
char* filename = argv[1];
int Nx = (int)strtol(argv[2], NULL, 10);
int Ny = (int)strtol(argv[3], NULL, 10);

const double deltaX = 2.0*X_MAX/(double) Nx;
const double deltaY = 2.0*Y_MAX/(double) Ny;
```

Moving on to the crucial part of the MPI parallelization, the computation of the rank ranges. First and foremost, we must compute the number of rows per process. This is achieved by dividing the total number of rows in the grid Nx over the total number of processes (size). We also store the remainder if the number of rows cannot be evenly distributed among the processes.

```
int rows_per_rank = Nx / size; //how many rows each process will handle
int remainder = Nx %size; //distribute if any leftover row
```

To handle a possible non-zero remainder, we distribute the leftover rows among the processes with rank strictly smaller than the remainder.

```
int extra_rows;
  if(rank < remainder) extra_rows = rank;
  else extra_rows = remainder;
```

Once we have distributed the rows among the processes, we compute the start and ending row index.

```
int start_row = rank * rows_per_rank + extra_rows;
int end_row = start_row + rows_per_rank - 1;
if(rank < remainder) end_row ++;
```

Then, each process reads the assigned portion of data with the MPI routines MPI_File_open, MPI_File_read_at_all and MPI_File_close.

```
int num_rows = end_row - start_row + 1;
double* data = (double*)malloc(num_rows * Ny * sizeof(double));

MPI_File fh;
MPI_Offset offset = sizeof(double) * start_row * Ny;
MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_read_at_all(fh, offset, data, num_rows * Ny, MPI_DOUBLE,
MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

After the parallelization of the y dimension using OpenMP, we will combine all partial results of each process. This is explained in the following point.

2. Complete the parallelization using OpenMP to parallelize the other dimension. Explain how you have proceeded.

To parallelize dimension y, we will declare the variable global_sum to hold the final result of the integral computed by different threads.

```
double global_sum;
```

Then, we initialize a parallel region to be executed by the threads and declare the local variable local_result to accumulate the partial results for each thread.
We use the #pragma omp parallel for directive to automatically distribute the iterations of the for loop. In the outer loop, we iterate over the assigned rows for the current process, whereas in the inner loop we iterate over the columns of the grid (Ny). We accumulate the integral value in local_sum. The clause reduction(+:local_sum) is added to force that all

private copies of local_sum are added at the end of the parallel region. Note that even though local_sum is declared before the parallel region -and hence it should be shared-, the reduction clause makes a private copy for each thread.

```
double local_sum = 0.0;
   #pragma omp parallel for reduction(+:local_sum)
   for (int j = 0; j < end_row - start_row; j++) {
       for (int i = 0; i < Nx; i++) {
           local_sum += data[j * Nx + i] * deltaX * deltaY;
       }
   }
```

To combine the results of all processes into a single variable global_result in the root process, we use the MPI_Reduce routine with MPI_SUM operation.

```
MPI_Reduce(&local_sum, &global_result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Lastly, the root process prints the results and we free the dynamically allocated memory of the array data and finalize the MPI environment.

```
if(rank == 0)
{
    printf("Result with N=%d M=%d size: %d threads: %d is %.12lf in %lf seconds\n",
Nx, Ny,size, omp_get_max_threads(), result, run_time);
}

free(data);
MPI_Finalize();
return 0;
```

3. Analyze and comment on the scaling for different values of MPI tasks and with/without OpenMP.

	1 task	**2 tasks**	**4 tasks**
1 thread	-0.284663673973 in 0.000036 seconds	-0.289355383786 in 0.000058 seconds	-0.276743334867 in 0.000044 seconds
2 threads	-0.284663673973 in 0.000093 seconds	-0.289355383786 in 0.000159 seconds	-0.276743334867 in 0.005967 seconds
4 threads	-0.284663673973 in 0.000180 seconds	-0.289355383786 in 0.000196 seconds	-0.276743334867 in 0.019123 seconds

Table 1: Results with a 100x100 grid.

	1 task	2 tasks	4 tasks
1 thread	-0.287305879493 in 0.739106 seconds	-0.287329385898 in 0.294802 seconds	-0.287266263784 in 0.251624 seconds
2 threads	-0.287305879493 in 0.428338 seconds	-0.287329385898 in 0.183439 seconds	-0.287266263784 in 0.459564 seconds
4 threads	-0.287305879492 in 0.404252 seconds	-0.287329385898 in 0.187962 seconds	-0.287266263784 in 0.443766 seconds

Table 2: Results with a 20000x20000 grid.

Firstly, it should be pointed out that the value of the integral is slightly modified by changing the number of tasks for a fixed number of threads. Nevertheless, the integral values are consistent for different numbers of tasks and threads.

Regarding the execution time, it is clear that it is significantly reduced as the number of tasks increases. Increasing the number of threads for a fixed number of tasks slightly reduces the execution time, but this is only seen for a large grid size. Indeed, for a small grid size (100x100) an increase on the number of threads produces an increase on the execution time, which is likely due to thread overhead.

In a nutshell, the overhead that OpenMP produces for small grids suggests that the benefits of a multithreading environment are not worth the computational costs.