Júlia López and Tània Pazos
u214964, u185115

Lab 1: OpenMP

Numerical integration

1. Test different values of N and check its effect on the results. Compare them with the real value (up to 11 significant digits). Explain what you observe.

N	Pi	Time to compute Pi (s)
10	3.142425985001098	0.000002
1000	3.141592736923131	0.000012
10000	3.141592654423136	0.000124
10000000	3.141592653590436	0.099717

Table 1: Values of pi and computation time in seconds after compiling the program pi_seq.c.

As Table 1 shows, the greater N is, the closer the value of pi obtained after compiling the program to the real value of pi. However, it should be noted that the time spent to compute this value significantly increases as we obtain more accurate approximations.

2. Parallelize the code using ONLY the #pragma omp parallel directive. Name this code pi par.c. You can modify or add anything you need to the code. Make sure the results are the same for different numbers of OpenMP threads. Explain how you have parallelized it.

Figure 1 illustrates that using only the #pragma omp parallel directive does not allow for a correct approximation of the value of pi due to a race condition: different threads are updating the same shared variable sum.

```
[U214964@ohpc 1_pi]$ cat pi_573339.out
gcc pi_par.c -o pi_par -fopenmp

Pi with 10 steps is 6.284851970002195 in 0.000042 seconds

Pi with 1000 steps is 5.657506465945831 in 0.000076 seconds

Pi with 10000 steps is 6.021416356390624 in 0.000422 seconds

Pi with 10000000 steps is 5.955367412594454 in 0.706503 seconds
```

Figure 1: Compilation results of pi_par.c without race conditions management.

To parallelize the code without race conditions, the code implements the following key points.

The workload, i.e., the total number of rectangles, is distributed into several threads so that each thread is assigned a specific range of rectangles. Hence, the range [first, last) determines the rectangles that will be assigned to a thread. In the case of the last thread, the code adjusts the last index to N. The range is computed as follows.

```
        int id = omp_get_thread_num();
        int nthreads = omp_get_num_threads();

        int first = N/nthreads*id;
        int last = N/nthreads*(id+1);

        if(id == nthreads-1) last = N;
```

Then, each thread computes the area of each of the rectangles in its range and adds this contribution to the variable sum, which is private for each thread. This ensures that each thread computes its contribution independently and thus avoids corrupted results.

```
        for(int i = first; i<last; i++){ //iterate through all N rectangles
            x = (i+0.5)*dx; //midpoint of the current interval
            double height = 4.0 / (1.0 + x*x); //computes the height of the
rectangle
            sum += height*dx;
        }
```

Each thread's sum is stored in a different position of the array sum_threads to avoid race conditions.

```
        sum_threads[id] = sum;
```

Finally, the contributions of all threads stored in sum_threads are added into the variable pi, which is exactly the approximation of the value of pi.

```
for (int i = 0; i < omp_get_max_threads(); i++) {
        pi += sum_threads[i];
    }
```

Regarding the main function, it simply checks that the correct number of arguments are received -in this case, the number of rectangles N- and calculates the time needed for the computation of pi.

3. Parallelize the code using tasks. Name this code pi task.c. Use a divide-and-conquer strategy. That is, create tasks that split the domain recursively. Explain how you have done it. To avoid stack overflow due to excessive task creation, you will have to modify the code so that the tasks compute at least 1024 steps. Therefore, this code must accept another argument, M, which is the minimum number of steps in a task.

To parallelize this code using tasks we have implemented two functions: our original computePi function and a new tasksPi function, and the main function.

The computePi function calculates the partial sum of the integral approximation for pi within a specific range of rectangles. First of all, concerning the previous implementations of this code, we have added more parameters: start, end, N, and pi.

```
void computePi(int start, int end, int N, double* pi)
```

Start and end define the first and last rectangle each task is going to consider for its computation, N will reference the total number of rectangles, and pi is a pointer to the global variable pi which we will use to store the sum of all the partial sums. Differing from the sequential and parallel implementations, we decided to store pi in a global variable for convenience; hence we changed the type of the function to void. Since now pi is a global variable, we used a #pragma omp atomic to avoid race conditions:

```
#pragma omp atomic
        *pi += sum;
```

Then, we implemented tasksPi function which implements the divide and conquer approach using OpenMP tasks to recursively compute pi.

```
void tasksPi(int start, int end, int M, int N, double* pi){
```

As inputs we have again the start and end indexes, the N and the pointer to the global variable pi but here we add a new parameter M which is a threshold value that determines when the difference end-start is small enough to compute the sum, that is, to call the computePi function.

```
if (end - start <= M)
   { // if the number of steps is smaller than M we compute Pi
       computePi(start, end, N, pi);
   }
```

If that is not the case, we will split the range into two halves (start to middle and middle to end) and recursively launch an OpenMP task for each half calling tasksPi. Lastly, we use the taskwait directive to ensure that all child tasks are completed before proceeding.

```
else
   { //if it is not, we divide into smaller steps
       int middle = (start + end) / 2;
       double left_sum, right_sum;
#pragma omp task  // left tasks
       {
           tasksPi(start, middle, M, N, pi);
       }
#pragma omp task // right tasks
       {
           tasksPi(middle, end, M, N, pi);
       }
#pragma omp taskwait // wait for all the tasks to finish
   }
```

For the main function, we continue with the same structure as before. However, now we have added another command line input: M so we have added the appropriate adaptations to ensure it is correctly implemented.

```
if(argc != 3){
       printf("Incorrect number of arguments\n");
```

```
        return 1;
    }

    int N = atoi(argv[1]); //We convert to string the input argument N
    int M = atoi(argv[2]); //same for argument M
```

To finish, we have added the allocation of memory and initialization for the global variable pi and directly starting a parallel region with the directive #pragma omp parallel followed by #pragma omp single to ensure that only one task calls the tasksPi function.

```
double* pi = (double*)malloc(sizeof(double));
*pi = 0.0;
#pragma omp parallel // to parallelize the computation of pi in the tasksPi function
    {
        #pragma omp single // to make sure only one thread calls the function
        {
            tasksPi(0, N, M, N, pi);
        }
    }
```

4. Compare the three versions with N = 1024^3 = 1073741824. For the case of pi task.c, use the M = 1024^2. Explain the results.

```
Pi with 1073741824 steps is 3.141592653589982 in 8.480123 seconds

Pi with 1073741824 steps is 3.141592653589982 in 8.404066 seconds

Pi with 1073741824 steps is 3.141592653589797 in 8.401711 seconds
```

Figure 2: Computation time of pi using a sequential, a multi-threaded and a task-based schema.

As we can see in Figure 2, the execution time for the computation of pi parallelized with tasks is the fastest, followed by the parallelization with threads and the sequential implementation.
It is worth noting that the difference between the two parallelized codes is very small in comparison with the difference between those two and the sequential execution. Indeed, for a large N, the effect of parallelization is noticeable.

Sorting

1. Explain the strategies you have followed in order to parallelize the algorithms. Which dependencies between iterations did you find?

The implementation in the sort_openmp function distributes among _NUM_THREADS threads the task of sorting -via Insertion Sort- the array of N elements.

Firstly, we calculate the chunk size according to the total number of elements in the array and the number of threads. To handle the case where the division between the total number of elements and the number of threads is not exact, a remainder is computed.

```
   int chunk_size = n/_NUM_THREADS;
   int remainder = n % _NUM_THREADS;
```

The array tmp_array is created to store the sorted chunks computed by each thread.
Then, we enter the parallel region using the #pragma omp parallel directive, which will be executed by _NUM_THREADS threads. Inside, the chunk -start and end indices of the array that will have to be sorted- of each thread is computed according to its thread ID and chunk size. In the case of the last thread, the remainder is added to its chunk.

```
   #pragma omp parallel num_threads(_NUM_THREADS)
   {
       int thread_id = omp_get_thread_num();
       // compute the start and end of the chunk of each thread
       int start = thread_id * chunk_size;
       int end = start + chunk_size; // end will be last index for that thread + 1

       if(thread_id == _NUM_THREADS-1){ // chunk of the last thread will be added
the remainder
           end += remainder;
       }
```

Each thread performs Insertion Sort of its assigned chunk and stored the final sorted chunk in tmp_array. Note that since each thread has non-overlapping change ranges, no race conditions appear.

```
// each thread reads chunk in array and sorts in tmp_array
       for (int i=start; i<end; i++) {
           int tmp = array[i];
```

```
            int j;
            // perform insertion sort in the chunk of array
            for(j=i-1; j >= start && tmp < array[j]; j--) {
                array[j+1] = array[j];
            }
            array[j+1] = tmp;
        }

        // each thread saves chunk into tmp_array
        for(int i=start; i<end;i++) {
            tmp_array[i] = array[i];
        }
    }
```

Lastly, the code enters a sequential phase to merge the sorted chunks from tmp_array into the final array. The array head_i has _NUMN_THREADS_ elements indicating the current index of each thread's chunk in tmp_array.

```
int head_i[_NUM_THREADS];
    for(int i = 0; i<_NUM_THREADS; i++){
        head_i[i] = i*chunk_size;
    }
```

The code iterates over the head_i, identifies the minimum value among the heads and to which thread it belongs to, copies the minimum value to the final array and increments the head index of the "owner" thread.

```
    int i = 0; // starting index to fill final array
    while (i<n) {
        int min_val = INT64_MAX; // store minimum value among heads of each thread
        int min_thread = -1; // thread with minimum value

        // find the minimum value among heads
        for(int j=0; j<_NUM_THREADS; j++){ // iterate through each head
            // check if head in correct thread chunk range
            if(head_i[j] < (j+1)*chunk_size && tmp_array[head_i[j]] < min_val) {
                min_val = tmp_array[head_i[j]];
                min_thread = j;
            }
        }
        // save minimum value in array and move the head that contained it
        array[i] = min_val;
        i++;
```

```
        head_i[min_thread]++;
    }
```

2. Plot the speedup for an array of 200.000 elements and 1, 2, 4, 8, and 16 cores for a strong scaling test. Plot the ideal speedup in the figures and use a logarithmic scale to print the results. Discuss the results and the gain.

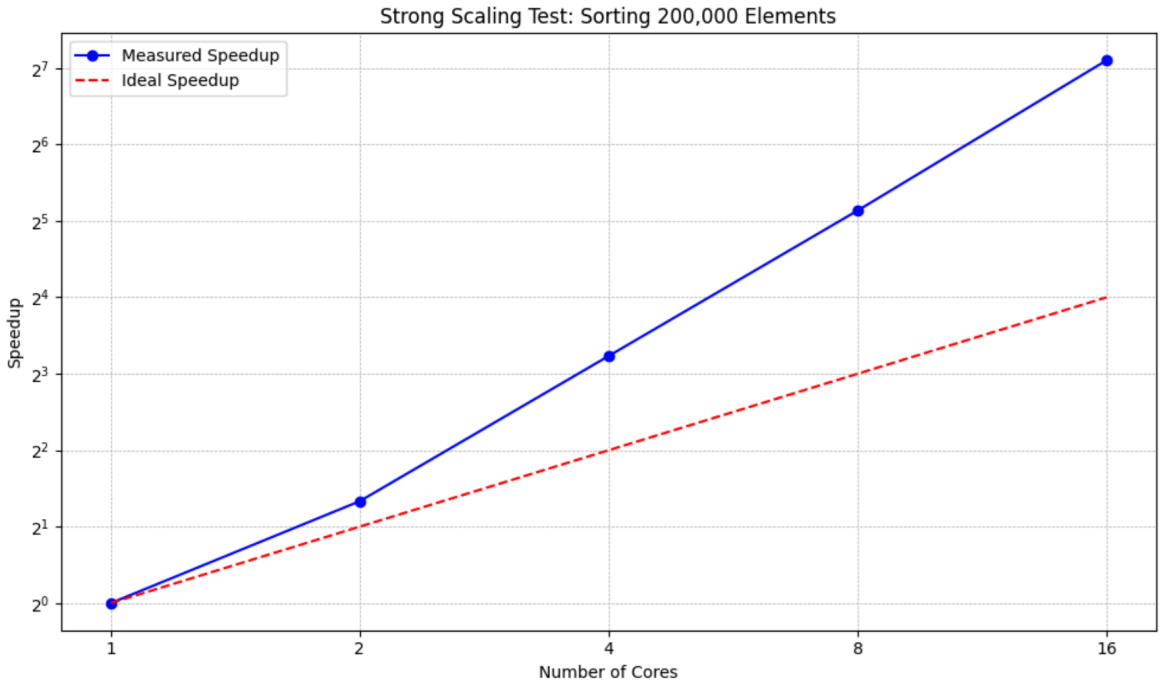

Figure 3: Speedup for an array of 200000 elements and 1, 2, 4, 8, and 16 cores in logarithmic scale.

Figure 3 illustrates the measured speedup in blue, which is the speedup of our computed results, and the ideal speedup in red, which is the theoretical maximum speedup that could be achieved if the algorithm worked perfectly.

The ideal speedup follows a linear trend and it can be seen that the measured speedup scales better than the ideal one. This indicates that our parallelization of the code is highly efficient and that it has succeeded in the maximization of resources. Indeed, the distribution of the initial array into smaller chunks sorted by each thread allows storing the smaller sorted arrays in the cache, which is faster to access than the main memory.

Prime numbers

1. Run the code with 1, 2, and 4 threads. Which scheduling technique scales the best?

```
Number of processors available = 2
Number of threads =              1

                      Default        Static       Dynamic
       N     Pi(N)         Time          Time          Time

       1         0     0.000008      0.000001      0.000002
       2         1     0.000001      0.000001      0.000001
       4         2     0.000001      0.000001      0.000001
       8         4     0.000001      0.000001      0.000001
      16         6     0.000001      0.000001      0.000001
      32        11     0.000002      0.000002      0.000002
      64        18     0.000003      0.000003      0.000004
     128        31     0.000009      0.000009      0.000010
     256        54     0.000026      0.000029      0.000029
     512        97     0.000098      0.000099      0.000115
    1024       172     0.000361      0.000347      0.000329
    2048       309     0.001220      0.001203      0.001117
    4096       564     0.004469      0.004488      0.004169
    8192      1028     0.016175      0.016051      0.014708
   16384      1900     0.059767      0.057372      0.051438
   32768      3512     0.211285      0.205519      0.189563
   65536      6542     0.784901      0.752781      0.708594
  131072     12251     2.954498      2.781224      2.665339
```

Figure 4: Compilation results of primes.c with 1 thread and chunk size = 20.

```
Number of processors available = 2
Number of threads =              2

                      Default        Static       Dynamic
       N     Pi(N)         Time          Time          Time

       1         0     0.000044      0.000002      0.000002
       2         1     0.000001      0.000001      0.000001
       4         2     0.000001      0.000001      0.000001
       8         4     0.000001      0.000001      0.000001
      16         6     0.000001      0.000001      0.000001
      32        11     0.000002      0.000001      0.000002
      64        18     0.000003      0.000005      0.000004
     128        31     0.000009      0.000013      0.000008
     256        54     0.000031      0.000042      0.000024
     512        97     0.000110      0.000152      0.000077
    1024       172     0.000379      0.000539      0.000251
    2048       309     0.001352      0.001840      0.000878
    4096       564     0.004831      0.005301      0.003235
    8192      1028     0.015257      0.016721      0.011732
   16384      1900     0.054641      0.059231      0.041881
   32768      3512     0.184453      0.198762      0.146818
   65536      6542     0.680657      0.727043      0.551887
  131072     12251     2.569860      2.672520      2.060829
```

Figure 5: Compilation results of primes.c with 2 threads and chunk size = 20.

```
Number of processors available = 4
Number of threads =              4

                       Default        Static       Dynamic
       N     Pi(N)          Time          Time          Time

       1         0      0.000117      0.000002      0.000003
       2         1      0.000002      0.000002      0.000002
       4         2      0.000002      0.000002      0.000002
       8         4      0.000002      0.000002      0.000002
      16         6      0.000002      0.000002      0.000002
      32        11      0.000002      0.000002      0.000003
      64        18      0.000003      0.000003      0.000003
     128        31      0.000006      0.000007      0.000007
     256        54      0.000016      0.000024      0.000015
     512        97      0.000062      0.000080      0.000042
    1024       172      0.000224      0.000266      0.000127
    2048       309      0.000750      0.000956      0.000428
    4096       564      0.002868      0.003590      0.001546
    8192      1028      0.009175      0.011854      0.005596
   16384      1900      0.031337      0.043146      0.020707
   32768      3512      0.114425      0.131423      0.076393
   65536      6542      0.438333      0.402303      0.287816
  131072     12251      1.377184      1.402442      1.075638
```

Figure 6: Compilation results of primes.c with 4 threads and chunk size = 20.

Figure 4, Figure 5 and Figure 6 show that dynamic scheduling is the scheduling that scales the best. The difference in the execution time using the default scheduler and the static scheduler is minimal. In addition, it can be observed that as the number of threads increases, the general execution time decreases.

2. Why? Discuss the results.

First of all, we can see that by increasing the number of threads we obtain a more efficient compilation time, since the workload is distributed.
Nevertheless, the factor that has the biggest influence on the execution time is the type of scheduling. On the one hand, the static scheduler divides the iterations statically at compile-time, i.e., each thread is assigned a fixed number of iterations from the start. This explains why the static scheduler may result in load imbalance when there are iterations that take longer to process than others: the threads that finish before must wait for the other threads that are still processing, without being able to “help”. On the other hand, the dynamic scheduler performs the assignment of iterations to threads at runtime, i.e., each thread is assigned a small chunk of iterations, and when the thread finishes processing, it requests for a new chunk. This is why the dynamic scheduler provides better load balancing -and hence often smaller execution times- since threads are assigned new iterations once they complete the previously assigned ones. It should be said that the default scheduler usually behaves like the static scheduler, with similar execution times.
In a nutshell, the dynamic scheduler tends to be faster than the static scheduler thanks to its flexibility with varying execution times iterations, especially when the chunk size is large.