OpenMP Programming Assignment:

Laplace 2D Report

Sofia Almirante Castillo (1527718)

Pau Reig Llunell (1527816)

December 2022

Study of steps in parallelization

The aim of the study is to parallelize the code provided with *OpenMP* and understand the impact it has on the overall performance. Our focus is going to be to analyze and compare the execution times of the program with each other. We are doing this by studying the impact it has depending on the part of the code that is parallelized, the size of the matrix, the maximum number of iterations and the number of threads used for the parallelization.

In the first instance, we decided to study the differences of performance time depending on the part of the code parallelized. In the code lines below we can see the parts of the program that has been modified for the parallelization of the different parts of the code. The executing time while introducing the different strategies are both in Table 1 and Fig. 1.

Table 1: Execution times obtained for the different parallelizing strategies. The number of threads is by default (8), the size of the matrix of size nxn is n=4096 and the maximum iterations is 100.

Constants	**Method**	**Execution time (s)**
num_threads: -	-	49.88
n = 4096	error (reduction)	41.13
max_iter = 100	laplace_step (for)	31.9
	laplace_step (collapse)	32.3
	copy (for)	47.07
	copy (collapse)	45.92
	error + step	22.16
	error + step + copy	18.47
	error + step + copy + CI	18.18

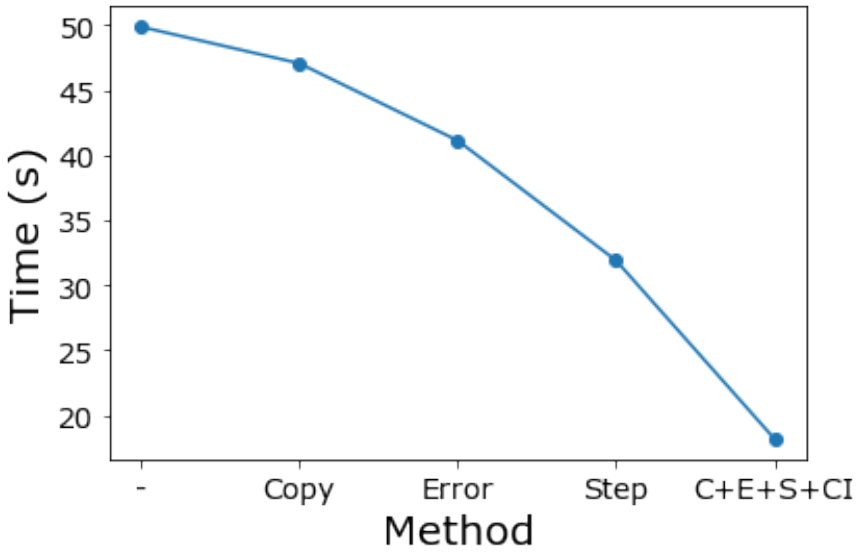

Figure 1: Execution time in seconds in function of the parallelized part of the code. The number of threads is by default (8), the size of the matrix of size nxn is n=4096 and the maximum iterations is 100.

The first strategy implemented was the parallelization in the error function, using the reduction method as shown in the code below. With this strategy, we observe a decrease of 17.5% in execution time compared to the original code.

```
#pragma omp parallel for private(x) shared(U) reduction(+:S)
  for (x = 1; x < X; x++)
  {
    S += U[x][T+1];
  }
```

Next, we worked on the parallelization of the function *laplace_step*, in which we expected to see the maximum improvement as it represents the main execution time of our program. Since we were going to parallelize a double *for* loop, we tried two different strategies. The first one was a *for* parallelization of the inside loop of the double *for* as shown in the first code below, while the other one was the *collapse* method of the double loop as shown in the second.

```
void laplace_step ( float *in, float *out, int n, int m )
{
  int i, j;
  for ( i=1; i < n-1; i++ )
  #pragma omp parallel for private(j) shared(in, out, n, m)
    for ( j=1; j < m-1; j++ )
      out[i*m+j]= stencil(in[i*m+j+1], in[i*m+j-1], in[(i-1)*m+j], in[(i+1)*m+j]);
}
```

```
void laplace_step ( float *in, float *out, int n, int m )
{
```

```
  int i, j;
  #pragma omp parallel for private(i,j) shared(in, out) collapse(2)
  for ( i=1; i < n-1; i++ )
    for ( j=1; j < m-1; j++ )
      out[i*m+j]= stencil(in[i*m+j+1], in[i*m+j-1], in[(i-1)*m+j], in[(i+1)*m+j]);
}
```

Applying the first strategy, we observed an improvement of 36.0% in execution time, while we observed a 35.2% using the second strategy. With the *for* method, we distribute the different columns of the same row in the different threads in order, while the *collapse* method decides itself what positions i and j of the matrix to compute. Because of the lack of improvement in execution time shown using *collapse*, we can deduce the way we distributed the positions of the matrix with the *for* strategy was already the most effective one, due to the fact that it is the most favorable while accessing the positions because of the way cache memory functions.

After that, we studied the impact of the parallelization in the copy of the matrix with both the *for* and *collapse* method, and we observed an improvement of execution time of 5.6% and 7.9% respectively.

```
void laplace_copy ( float *in, float *out, int n, int m )
{
  int i, j;
  #pragma omp parallel for private(i,j) shared(in, out) collapse(2)
  for ( i=1; i < n-1; i++ )
  num_threads(threads)
    for ( j=1; j < m-1; j++ )
      out[i*m+j]= in[i*m+j];
}
```

```
void laplace_copy ( float *in, float *out, int n, int m )
{
  int i, j;
  #pragma omp parallel for private(i,j) shared(in, out) collapse(2)
  for ( i=1; i < n-1; i++ )
  num_threads(threads)
    for ( j=1; j < m-1; j++ )
      out[i*m+j]= in[i*m+j];
}
```

Finally, we also parallelized the initialization of the matrix as shown in the code below.

```
void laplace_init ( float *in, int n, int m )
{
  int i, j;
  const float pi  = 2.0f * asinf(1.0f);
  memset(in, 0, n*m*sizeof(float));
  #pragma omp parallel shared(in, m, n)
  {
  #pragma omp for private(j)
```

```
  for (j=0; j<m; j++)  in[    j    ] = 0.f;
  #pragma omp for private(j)
  for (j=0; j<m; j++)  in[(n-1)*m+j] = 0.f;
  #pragma omp for private(i)
  for (i=0; i<n; i++)  in[   i*m   ] = sinf(pi*i / (n-1));
  #pragma omp for private(i)
  for (i=0; i<n; i++)  in[ i*m+m-1 ] = sinf(pi*i / (n-1))*expf(-pi);
  }
}
```

Applying all the parallel strategies at the same time, we can observe that we have reduced the execution time by an order of magnitude, which is a great improvement.

Study of maximum number of iterations

Once we applied all the strategies of parallelization mentioned before, we studied the impact of the maximum number of iterations. The results are shown in Table 2 and represented in Fig. 2.

Table 2: Execution times obtained for the different maximum of iterations of the parallelized code. The number of threads is by default (8) and the size of the matrix of size nxn is n=4096.

Constants	**iteracions**	**execution time (s)**
num_threads: -	100	5.5
n = 2048	200	10.85
	400	22.15
	1000	64.87

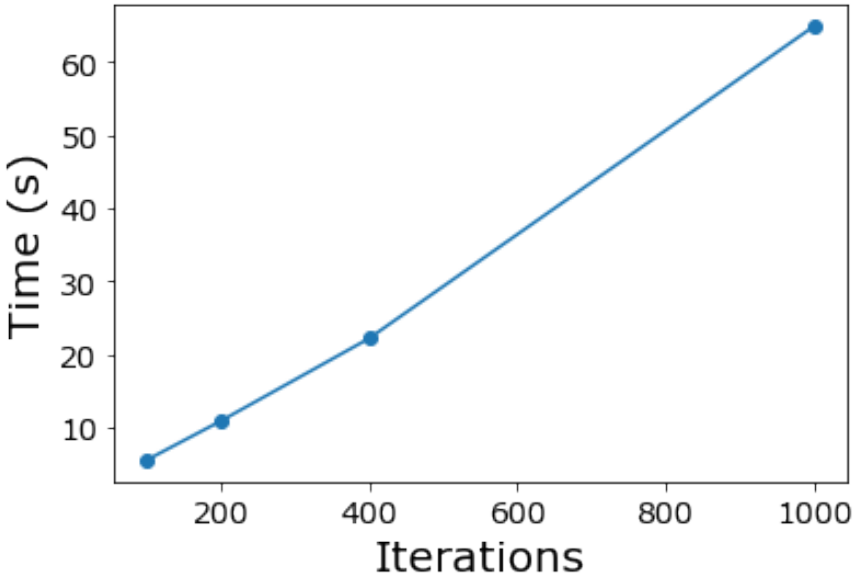

Figure 2: Execution time in seconds in function of the maximum iterations of the parallelized program. The number of threads is by default (8) and the size of the matrix of size nxn is n=4096.

We observe a linearity between the execution time and the maximum number of iterations provided to the program, which is compatible with what we expected since the *laplace_step* function represents the main amount of time inverted in executing the program, and this time is directly related to the number of times the loop of this function has to be executed (which directly depends on the number of iterations done by the program).

Study of size of the matrix

We reapplied the same strategy for focusing on the impact of the size of the matrix. The results are shown in Table 3 and Fig. 3.

Table 3: Execution times obtained for the different sizes of the matrix in the parallelized code. The number of threads is by default (8) and the size of the maximum number of iterations is 100.

Constants	**n**	**Execution time (s)**
num_threads: -	256	0.31
max_iter = 100	512	0.66
	1024	1.72
	2048	5.5
	4096	18.18
	8192	69.8

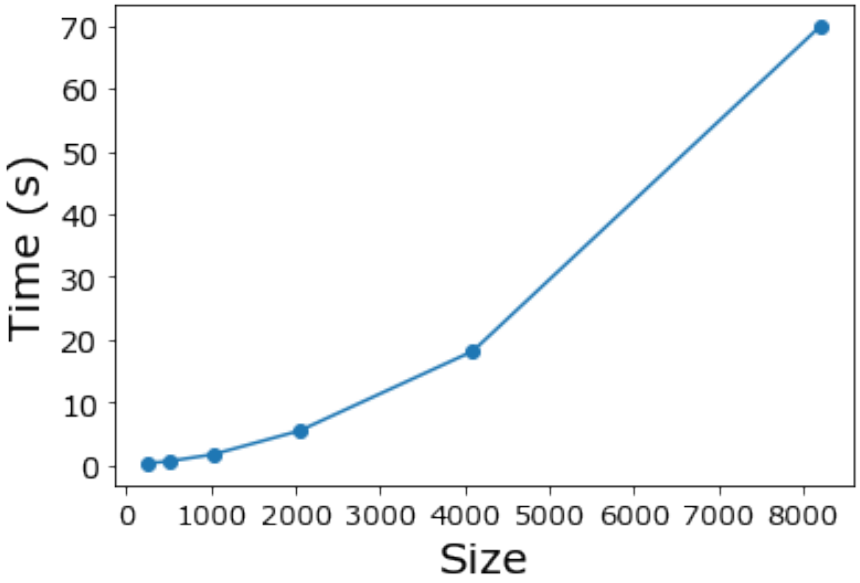

Figure 3: Execution time in seconds in function of the size in the matrix of the parallelized program. The number of threads is by default (8) and the maximum number of iterations is 100.

We now observe a potential growth in the execution time in relation to the matrix size.

Study of the number of threads

To finish the study, we studied the impact of the number of threads passed to the main function of the program (*laplace_step*) during the parallelization, as shown in the code below. The results are shown in Table 4 and Fig. 4.

```
    void laplace_step ( float *in, float *out, int n, int m )
{
  int i, j;
  for ( i=1; i < n-1; i++ )
  #pragma omp parallel for private(j) shared(in, out, n, m) num_threads(threads)
    for ( j=1; j < m-1; j++ )
      out[i*m+j]= stencil(in[i*m+j+1], in[i*m+j-1], in[(i-1)*m+j], in[(i+1)*m+j]);
}
```

Table 4: Execution time in seconds in function of the number of threads. The size of the matrix of size nxn is n=4096 and the maximum number of iterations is 100.

Constants	**num_threads**	**execution time (s)**
n = 4096	4	26.28
max_iter = 100	8	18.35
	16	121.13
	32	73.11
	9	185.70

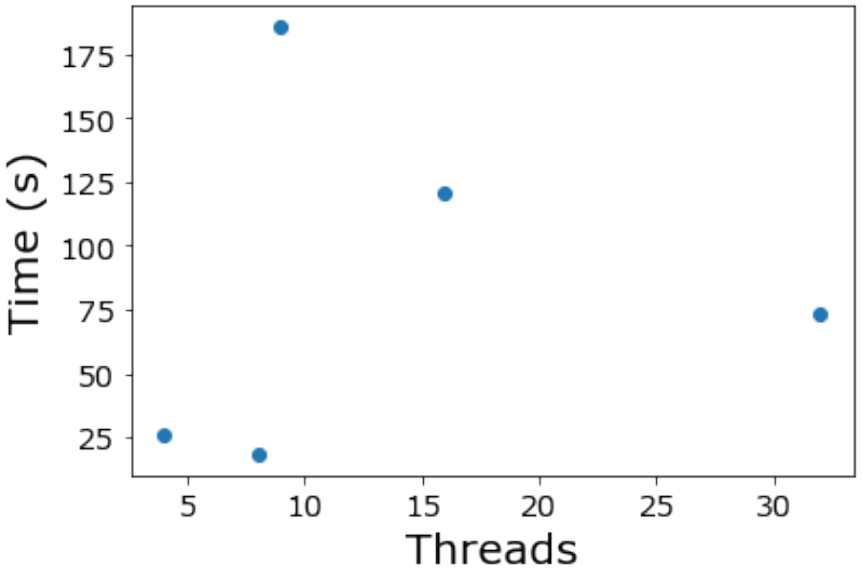

Figure 4: Execution time in seconds in function of the number of threads. The size of the matrix of size nxn is n=4096 and the maximum number of iterations is 100.

We observe that the number of threads that minimize the execution time is 8 (which coincide with the execution time when not specifying the number of threads). This is the expected result, since it is the same number of units the computer has, so it is the most efficient. We observe that for bigger number the efficiency decays, behavior that can be understood as the machine is wasting more time sending 2 small tasks to a lot of cores instead of 1 bigger task to less units.