Finite Elements Methods: a parallel implementation

Simone Colucci & Emanuele Fabbiani

February 15^{th}, 2016

Abstract

The Finite Elements algorithm is implemented through a C program and applied to an elliptic partial differential equation. A serial version is run on several mesh with growing size. The execution time is recorded and compared with theoretical prediction based on computational complexity. The results are coherent with the predictions. Using OpenMP directives, the code is made parallel and run on different machines. For every major function, the time gain is predicted using Ahmdal's law and then measured. The results show an overall improvement of the performance, even if lower than the theoretical limit. Other ways to speed up the executions are explored: an automatic optimisation of the machine code is tested. The optimised code outperforms both the serial and the parallel versions: an explanation is found looking to the improvement the compiler performs on the assembly.

Contents

1 Partial differential equations and finite element methods.

Partial differential equations (PDE) are differential equations where the unknown is a multivariable function. Far from being just complex mathematical problems, PDEs are very useful tools to model several different phenomena in Physics, Biology, Economics and Engineering. Maxwell's laws of electromagnetic fields and Navier-Stokes equations for fluids are just two examples of major applications of PDE theory. However, PDEs are usually impossible to solve in closed form: numerical methods are required to compute approximate solutions. The most used methods to solve PDEs numerically are known as Finite Element Methods. First introduced in the '40s FEMs utility was underestimated at the time because of the lack of computational power. Nowadays they enter the design process of planes, buildings and cars as a low-cost and effective alternative to real stress tests.

1.1 The Finite Element Method

To understand how the method works, we consider an elliptic homogeneous equation:

$$\begin{cases} -\Delta u\left(x,y\right) = f\left(x,y\right) & in\ \Omega \\ u = g\left(x,y\right) & on\ \partial\Omega \end{cases} \tag{1.1}$$

where $\Delta u\left(x,y\right) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}$ and $f : \mathbb{R}^2 \mapsto \mathbb{R}$ is a well-defined function in $\Omega \in \mathbb{R}^2$. FEMs are based on the weak formulation of the differential problem, so we have to introduce a test function v, which is supposed to be regular enough. We multiply the differential equation by v and we integrate over Ω:

$$\int_\Omega \left(-\Delta u\right) v = \int_\Omega fv \tag{1.2}$$

Then we apply the Gauss-Green formula:

$$\int_\Omega \Delta uv + \int_\Omega \nabla u \cdot \nabla v = \int_{\partial\Omega} v\frac{\partial u}{\partial n} \tag{1.3}$$

And we choose v such that $v|_{\partial\Omega} \equiv 0$. We got to the weak formulation:

$$\int_\Omega \nabla u \cdot \nabla v = \int_{\partial\Omega} v\frac{\partial u}{\partial n} + \int_\Omega fv \tag{1.4}$$

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega fv \tag{1.5}$$

This equation should hold for every v. Up to here, no approximation has been performed. Form now on, the Finite Elements approximation begins. We approximate the set Ω with the union of triangles $\mathcal{T}_h$.

$$\Omega \simeq \bigcup_{T\in\mathcal{T}_h} T \tag{1.6}$$

The equality holds only if Ω is a polygon. The length h of the longest edge of a triangle $T \in \mathcal{T}_h$ is a good index about how fine the mesh is. Then we approximate the function u with a new function u_h, which is supposed to be linear on every triangle T and continuous on Ω; moreover we want the test function v to present the same features. To underline the restriction, v will be called v_h. It is possible to show that the functional space u_h and v_h belong to has a dimension which is finite and equals the number n of vertexes of triangles in the internal of the set Ω. We define φ_i, $i = 1...n$ a set of functions which is a basis for the functional space v and u belongs to:

$$u_h\left(x,y\right) = \sum_{i=1}^{n} u_i\varphi_i\left(x,y\right) \tag{1.7}$$

The equation 1.5 is verified by every v if and only if it holds for all the elements φ_j in a basis of the functional space. We impose this condition and we get to the following linear system:

$$\sum_{i=1}^{n} u_i \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j = \int_\Omega f \cdot \varphi_j \quad \forall j = 1...n \tag{1.8}$$

We define $W = \{W_{i,j}\} = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j$, $u = \{u_i\}$ and $b = \{b_j\} \coloneqq \int_\Omega f \cdot \varphi_j$, so that we can write the system in matrix form:

$$Wu = b \tag{1.9}$$

We can exploit the linearity of the integral to ease the computation of W and b: we can compute the contribution on every triangle T and then sum them up. We choose the easiest set of basic functions φ, that is the $\varphi_j = 1$ on the vertex with index j and 0 on all the others. The computation of the local matrix W_T of the triangle T lead to a symmetric matrix:

$$W_T = \begin{bmatrix} \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_i & \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j & \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_k \\ \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j & \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_j & \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_k \\ \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_k & \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_k & \int_\Omega \nabla\varphi_k \cdot \nabla\varphi_k \end{bmatrix} = \frac{1}{4\,|T|} \begin{bmatrix} l_i \cdot l_i & l_i \cdot l_j & l_i \cdot l_k \\ l_j \cdot l_i & l_j \cdot l_j & l_j \cdot l_k \\ l_k \cdot l_i & l_k \cdot l_j & l_k \cdot l_k \end{bmatrix} \tag{1.10}$$

where $|T|$ is the area of the triangle and $l_i \cdot l_j$ is the scalar product between the edges described by the vectors l_i and l_j. We can compute the local right-hand side b_T, using some quadrature formula. We choose the barycentre formula, the easiest one:

$$b_T = \begin{bmatrix} \int_\Omega f \cdot \varphi_i \\ \int_\Omega f \cdot \varphi_j \\ \int_\Omega f \cdot \varphi_k \end{bmatrix} \simeq \frac{|T|}{3} \begin{bmatrix} f\,(x_B, y_B) \\ f\,(x_B, y_B) \\ f\,(x_B, y_B) \end{bmatrix} \tag{1.11}$$

dove

$$x_B = \frac{1}{3} \sum_{i=1}^{3} x_i \tag{1.12}$$

$$y_B = \frac{1}{3} \sum_{i=1}^{3} y_i \tag{1.13}$$

It can be proved that this formula is exact if and only if the function f is constant on the border of Ω.

1.2 Model problem

We take the following partial differential equation as a model:

$$\begin{cases} -\Delta u\,(x,y) = -4 & in\ \Omega = \left\{(x,y) \in \mathbb{R}^2 : x^2 + y^2 \leq 1\right\} \\ u = 1 & on\ \partial\Omega = \left\{(x,y) \in \mathbb{R}^2 : x^2 + y^2 = 1\right\} \end{cases} \tag{1.14}$$

We choose this very equation because we can deduce the true solution. If we try

$$\tilde{u}\,(x,y) = x^2 + y^2 \tag{1.15}$$

we get:

$$-\Delta\tilde{u}\,(x,y) = -\frac{\partial^2 \tilde{u}\,(x,y)}{\partial x^2} - \frac{\partial^2 \tilde{u}\,(x,y)}{\partial y^2} = -2 - 2 = -4 \tag{1.16}$$

As the solution to such a problem is proved to be unique, $\tilde{u}$ is the function we are looking for.

2 Serial implementation

We shall consider now the actual C-language implementation[1] of the algorithm that compute the approximate solution of the given problem with a finite elements approach. In order to focus on the interesting parts, we will disregard I/O operations: they are essential to the program execution but do not represent any case of study.

Roughly speaking, the code can be split into four sections:

1. Memory allocation and initialisation for both the stiffness matrix W and the right-hand side b.
2. Computation of the global stiffness matrix W and the global right-hand side b by computing and assembling up local ones.
3. Assignment of Dirichlet conditions.
4. Computation of the vector u solution of $Wu = b$.

2.1 Memory allocation and variable initialisation

The very first task the algorithm must handle is a proper dynamic memory allocation, to store both the stiffness matrix and the right-hand side. In order to improve memory usage and to make the access uniform, the matrix is stored as a linear 1D vector. Once allocated, all the elements of both the stiffness matrix and the right-hand side must be set to zero, as the assembling procedure (2.2) requires. Disregarding the initial part that simply allocates required memory space, we notice that the initialisation actually results in n store calls, n being the dimension of the vector. In our case, that leads us to $N^2 + N$ store calls, where N equals the number of vertices in the given mesh. The function implementing allocation and initialisation is shown in Algorithm 1.

Algorithm 1 Memory allocation and variable initialisation.

```
1 double* zeros(int dim) {
2   int i = 0;
3   double* vector = (double*) malloc(sizeof(double) * dim);
4   for (i = 0; i < dim; i++) {
5     vector[i] = 0;
6   }
7   return vector;
8 }
```

2.2 Assembling of the stiffness matrix and the right-hand side

The assembling operation is realised by initially calling, for each one of the N vertices in the mesh, two functions that compute local elements and then summing them back to the global ones. As every local contribute is summed to the existing values, the stiffness matrix and the vector should be initialised to zero. The map between triangles and local indices of triangles is stored in a preinitalized matrix. The code under consideration is shown Algorithm 2.

Since the computation of local elements uses only values at the three vertices of the local triangle, its complexity is independent from N, and can thus be ignored. As a result, we can state that the overall complexity of the assembling task is $O(N)$, i.e. linear with respect to the dimension of the mesh.

[1] The code is hosted in a Github repository: `https://github.com/donlelef/parallel-pde.git`.

Algorithm 2 Assembling of the stiffness matrix and the right-hand side.

```
for (tri = 0; tri < vertexSize; tri++) {
  for (i = 0; i < 3; i++) {
    globalVertex = (int) *(vertexNumbers + tri * 3 + i) - 1;
    vertices[i][0] = *(meshPoints + globalVertex * 2 + 0);
    vertices[i][1] = *(meshPoints + globalVertex * 2 + 1);
  }

  localstiffnessMatrix(vertices, localW);
  localVector(vertices, f, localB);

  for (i = 0; i < 3; i++) {
    globalVertex = (int) *(vertexNumbers + tri * 3 + i) - 1;
    b[globalVertex] += localB[i];
    for (j = 0; j < 3; j++) {
      globalVertex2 = (int) *(vertexNumbers + tri * 3 + j) - 1;
      *(w + globalVertex * meshSize + globalVertex2) += localW[i][j];
    }
  }
}
```

2.3 Dirichlet conditions assignment

Before starting the computation of the solution, we must assign particular values to the stiffness matrix and the right-hand side in order to make Dirichlet conditions hold on the boundary of the domain. We have to retrieve the vertices on boundary of the domain and assign particular values at the correspondent row in the stiffness matrix and the right-hand side vector. By looking at the implementation - code is shown in Algorithm 3 - we can notice that the number of operations is in the order of $N \cdot (1 + N_{boundary})$ where $N_{boundary}$ is the number of vertices on the boundary. Since this number grows almost linearly with N when N is big enough, the total amount of operations can be approximate with $N(1 + \alpha N) = N + \alpha N^2$, which is $O\left(N^2\right)$.

Algorithm 3 Dirichlet conditions assignment.

```
void assignDirichletCondition(int meshSize, double g, double* meshPoints,
    double* w, double* b) {
  int i, j;
  for (i = 0; i < meshSize; i++) {
    double x = *(meshPoints + 2 * i);
    double y = *(meshPoints + 2 * i + 1);
    if (fabs(x * x + y * y - 1) < TOL) {
      for (j = 0; j < meshSize; j++) {
        *(w + meshSize * i + j) = (i == j) ? 1 : 0;
      }
      b[i] = g;
    }
  }
}
```

2.4 Computation of the approximate solution

With both the matrix and the right-hand side properly assigned, it is possible to compute the FEM approximation of the solution u. As equation 1.8 points out, the procedure requires to solve a linear system with N equations in N unknowns. The system is solved by two functions that implement the Gaussian Elimination algorithm: at first, the stiffness matrix and the right-hand side are modified with several pivot operations till an upper-triangular structure is reached; then the solutions is computed backward - Algorithm 4.

The cost for computing the upper triangular system is in the order of $\frac{N^3}{3}$; the cost for solving the upper triangular system is N^2, so that the total asymptotically complexity is $O\left(\frac{N^3}{3} + N^2\right)$.

As we will see later, this is actually the bottleneck of the whole program.

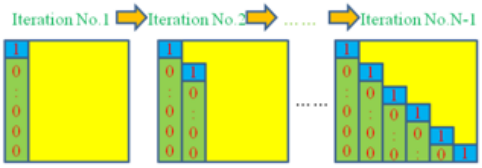


Figure 2.1: Gaussian elimination algorithm.

Algorithm 4 Computation of the approximate solution.

```
1  double* gaussianElimination(int n, double* matrix, double b[]) {
2    int i, j, k;
3    double aux, temp;
4    double* x = zeros(n);
5
6    /* Gaussian Elimination */
7    for (j = 0; j < (n - 1); j++) {
8      for (i = (j + 1); i < n; i++) {
9        aux = *(matrix + n * i + j) / *(matrix + n * j + j);
10       for (k = j; k < n; k++) {
11         *(matrix + n * i + k) -= (aux * (*(matrix + n * j + k)));
12       }
13       b[i] -= (aux * b[j]);
14     }
15   }
16
17   /* Back substitution */
18   x[n - 1] = b[n - 1] / *(matrix + n * (n - 1) + n - 1);
19   for (i = (n - 2); i >= 0; i--) {
20     temp = b[i];
21     for (j = (i + 1); j < n; j++) {
22       temp -= (*(matrix + n * i + j) * x[j]);
23     }
24     x[i] = temp / *(matrix + n * i + i);
25   }
26   return x;
27 }
```

3 Profiling the serial execution

After this brief study of the theoretical complexity of each kernel function, we shall now analyse the actual results obtained by running the code on a computer.

Several simulations were performed on an Intel i7-2630QM machine to study the computation time growth with respect to different mesh dimensions. All the interesting parameters related to the CPU under consideration are listed below.

Processor	RAM	Clock	Cores	Threads	L1 cache	L2 cache	L3 cache
i7-2630QM	4 GB	2 GHz	4	8	4x32 KB 8-way	4x256 KB 8-way	6 MB 12-way

Table 3.1: Intel i7-2630QM features.

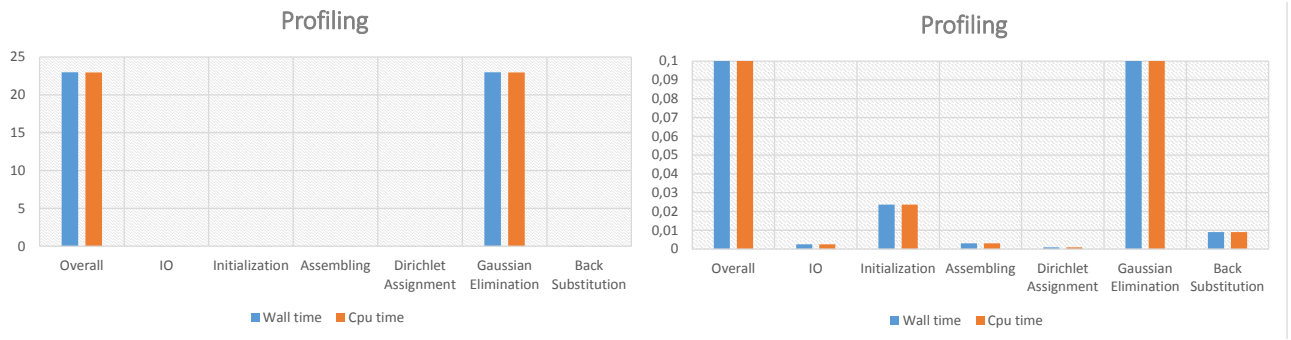
For each function, we computed both the wall clock time, which is the actual amount of time taken to perform the job, and the CPU time, which is the amount of time the function actually holds the CPU during its execution. Common profiling tools, like gprof, offer a resolution in the order of the tenths of milliseconds: looking at the values in Table 3.3, we consider such a precision too little. Moreover, they regularly sample the call stack, introducing more instruction and thus generating an overhead which may lead to untrustworthy results. Hence, we used native functions, in particular `omp_get_wtime()` of `omp.h` for wall clock time and `clock()` of `time.h` for CPU time. Both timers has resolutions in the order of microseconds.

The raw data we collected are shown in the following Table 3.3.

Mesh Size	Overall	IO	Initialisation	Assembling	Dirichlet	Gaussian Elimination
25	0.000080	0.000123	0.000014	0.000023	0.000003	0.000051
33	0.000151	0.000143	0.000032	0.000042	0.000004	0.000100
41	0.000232	0.000171	0.000036	0.000034	0.000005	0.000188
75	0.001117	0.000153	0.000033	0.000063	0.000010	0.001030
144	0.006962	0.000283	0.000109	0.000125	0.000020	0.006774
544	0.354619	0.000901	0.004083	0.000672	0.000139	0.353258
2173	22.964478	0.002509	0.023644	0.003044	0.000954	22.951455
3447	97.154916	0.004003	0.048192	0.004370	0.001921	97.125484
6084	513.562017	0.007300	0.150833	0.007964	0.004399	513.480326
13693	5563.750536	0.015359	0.754177	0.018652	0.014815	5563.298515

Table 3.3: Profiling of the serial execution time (seconds).

Since the simulations were run minimising the number of concurrent jobs other than the program under test, we notice that the wall clock time and CPU time do not actually differ significantly - see Figure 3.1. Hence, from now on we will consider just the CPU time measurement, indeed more significant.



(a) Real scale.

(b) Zoom.

Figure 3.1: CPU time and wall clock time (seconds).

To better understand the results, for each function we plot both the expected trend and the real one; moreover, to highlight asymptotic complexity, we also provide log-log scale plots. On each plot, an equation

showing the theoretical prediction has been added. It is the output of a least square regression performed on the data shown in Table 3.3 with the models described in Section 2.

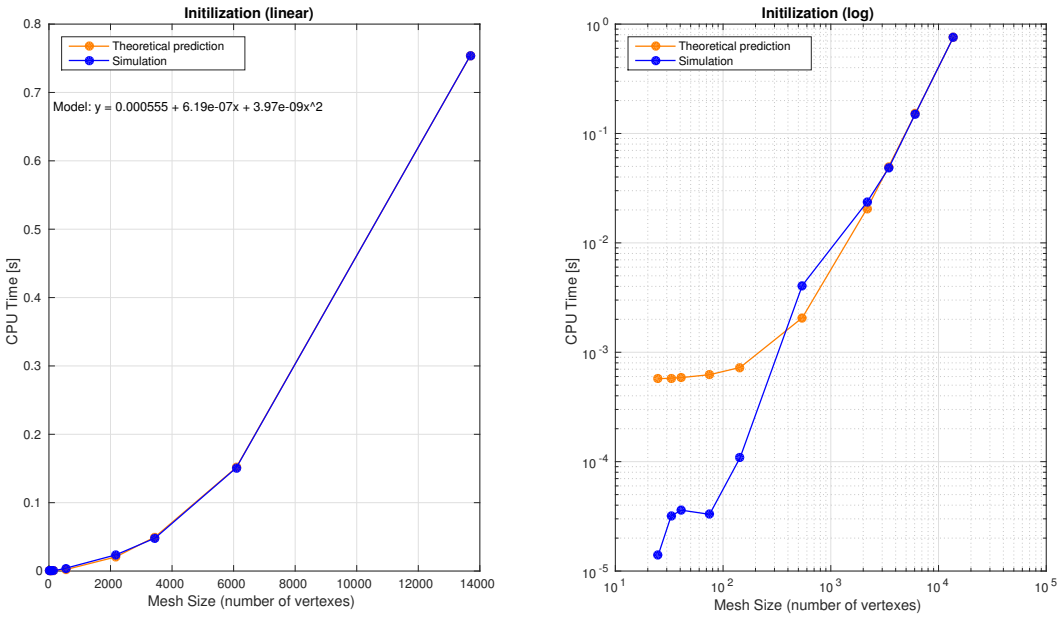


Figure 3.2: Initialisation.

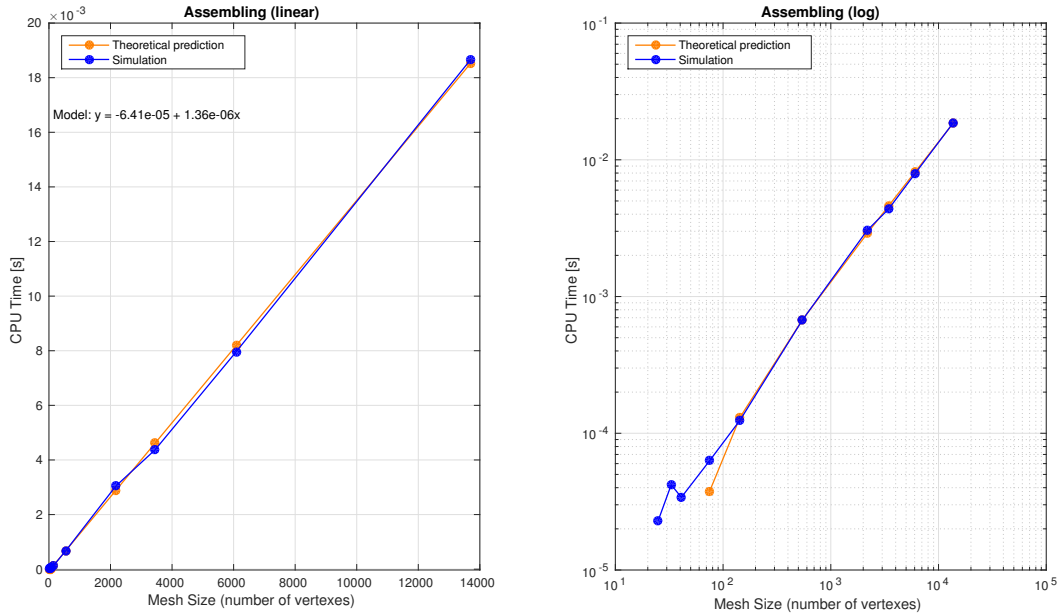


Figure 3.3: Assembling.

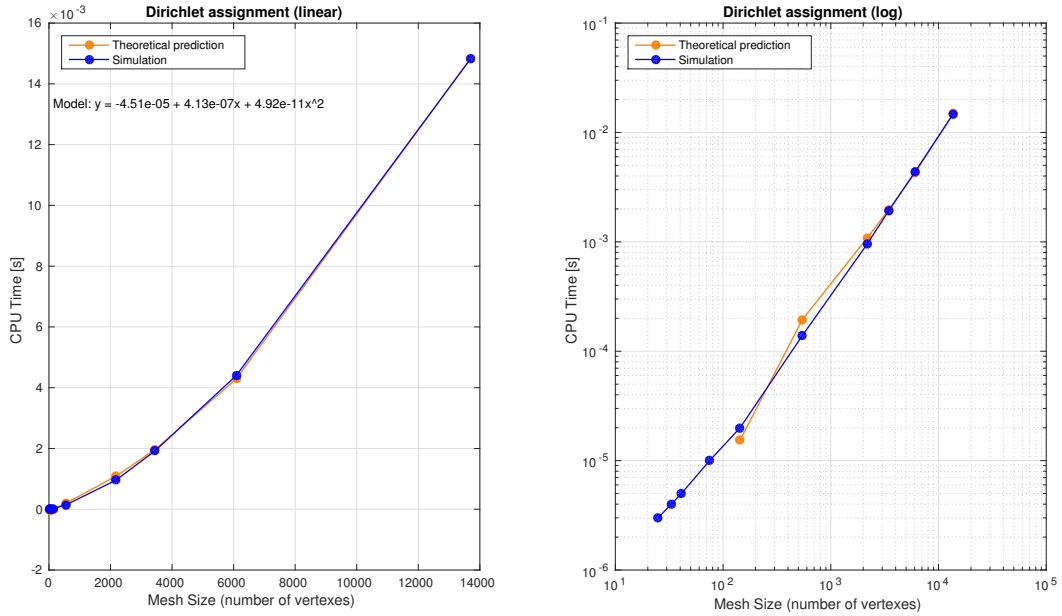


Figure 3.4: Dirichlet condition assignment.

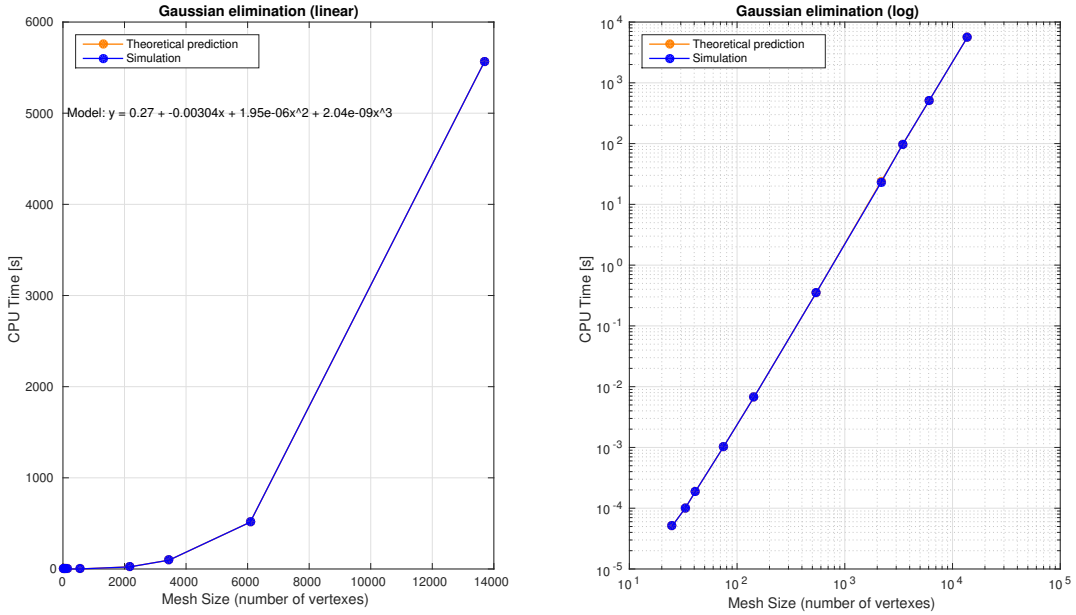


Figure 3.5: Gaussian elimination (theoretical prediction is almost perfectly overlapped to simulation)

4 Parallel implementation

The simulations run on the machine show how fast the computation time grows with respect to the mesh size. They also highlighted how the bottleneck of the whole program is the Gaussian Elimination algorithm, in agreement with what the theoretical study suggests.

Those results, however, are obtained by running the code in a serial fashion, thus wasting possible available parallelism offered by multicore machines. In other words, the application we considered till now is absolutely not scalable. A multicore machine, instead, lets us spread the job among cores, thus decreasing computation time with a factor theoretically equal to the number of available cores. Of course, this is never the case: because of data dependencies, only a fraction of the code can actually run in parallel. Moreover, when more than one thread is running, job distribution, resource access and synchronisation must be taken into account. The only way to do so is to add more instruction, thus slowing the execution. Parallelization is worth introducing only if the gain in computation speed is higher than the loss due to overhead.

As the code involves a lot of computations, an improvement in performance can be obtained: we shall see later in details how much and why.

The tools to build parallel code are provided by the Open-MP API interface, a set of compiler directives, library routines and environment variables that allow to build parallel applications on shared memory systems. We will not cover technical specifications for the API, since this is not the aim of this report. However, we will define and describe accurately every directive we actually use to develop the parallel code.

As we did for the serial ones, we will carry out the analysis function by function: for each one of them, we will study available parallelism and describe the implementation, along with the reasons beneath any choice we made.

4.1 Memory allocation and variable initialisation

As we already described, this function allocates the memory required given the dimension of the linear vector, and eventually sets to zero the value of each element. Disregarding allocation, we notice that the operations are completely independent one another, so the fraction $fraction_{parallel}$ of the code that can run in parallel is approximately 1. If t_{serial} is the computation time of the serial execution and N is the number of threads that run in parallel, one should have

$$t_{parallel} = (1 - fraction_{parallel}) \cdot t_{serial} + fraction_{parallel} \cdot \frac{t_{serial}}{N} \simeq \frac{t_{serial}}{N} \tag{4.1}$$

which represents an ideal speed-up equal to the number of running threads. The parallel implementation is shown in the

Algorithm 5 Memory allocation and variable initialisation.

```
1 double* zeros(int dim) {
2   int i = 0;
3   double* vector __attribute__ ((aligned (16)));
4   vector = (double*) malloc(sizeof(double) * dim);
5 #pragma omp parallel for private(i) schedule(static)
6   for (i = 0; i < dim; i++) {
7     vector[i] = 0;
8   }
9   return vector;
10 }
```

The whole procedure is realised by a loop that scans the vector and writes into indexed element. Hence, the parallelization is realised using the directive `#pragma omp paralell for`, that splits the iteration among threads and then let them execute. The clause `private(i)` specifies that the variable used as loop counter has to be private for each thread: we thus avoid the same element of the vector to be processed by more than one thread. We observed that every basic operation is a single `store`: the computational load is evenly distributed among all the threads. This is the reason why we decided to use the `schedule(static)` clause, which assign the same number of iterations to every thread. By default openMP gives every thread the total number of iteration divided by the number of threads, avoiding further distribution and thus keeping the granularity as low as possible and minimising the overhead.

The reason why an `__attribute__ ((aligned (16)))` has been added to the declaration of the vector will be explained in Section 6.

4.2 Assembling of the stiffness matrix and the right-hand side

The assembling procedure is made of several nested loops: each loop, but the outer one, contains a relatively small number of operations. Each iteration of the outer loop is independent from all the others, so it can be turned in a single parallel region. However, updates to the global stiffness matrix and the global right-hand side must be carefully handled. The pointers to the data structures are shared among all the running threads; but they must be accessed serially in order to prevent unpredictable results. If every memory access were negligible, the expected speedup would be close to the number of threads. However, this algorithm requires the access to four different data structure: the matrix of the global indices of vertices, the matrix of their coordinates, the stiffness matrix and the right-hand side vector. As the global index of the vertices of the triangle are almost randomly distributed, the other data structures are not accessed with a regular pattern, thus stressing memory hierarchy. Considering, for instance, the biggest mesh size M we used, we can compute the size W_b of the stiffness matrix in byte:

$$W_b = M \cdot M \cdot 8 = 13693^2 \cdot 8 \simeq 1,4\, GB \tag{4.2}$$

An irregular access pattern to such a matrix is likely to generate a lot of full misses in the cache hierarchy. Moreover, updates must be execute by one thread only. As the bottleneck of the algorithm is memory access, we expect the parallelization not to lead to a significant performance improvement: the overhead the management of threads introduces is likely to nullify the gain in computational time consumption. The code for the parallel version of the function is shown below.

Algorithm 6 Assembling of the stiffness matrix and the right-hand side

```
1 #pragma omp parallel for private(tri, i, j, globalVertex, globalVertex2,
    vertices, localW, localB) schedule(static)
2 for (tri = 0; tri < vertexSize; tri++) {
3   for (i = 0; i < 3; i++) {
4     globalVertex = (int) *(vertexNumbers + tri * 3 + i) - 1;
5     vertices[i][0] = *(meshPoints + globalVertex * 2 + 0);
6     vertices[i][1] = *(meshPoints + globalVertex * 2 + 1);
7   }
8
9   localstiffnessMatrix(vertices, localW);
10  localVector(vertices, f, localB);
11
12  for (i = 0; i < 3; i++) {
13    globalVertex = (int) *(vertexNumbers + tri * 3 + i) - 1;
14 #pragma omp critical
15    b[globalVertex] += localB[i];
16    for (j = 0; j < 3; j++) {
17      globalVertex2 = (int) *(vertexNumbers + tri * 3 + j) - 1;
18 #pragma omp critical
19      *(w + globalVertex * meshSize + globalVertex2) += localW[i][j];
20    }
21  }
22 }
```

The cycle is made parallel with a `#pragma omp paralell for` clause. All the variables which refers to the local element and the position where their contribution has to be added on the global stiffness matrix has to be private. The static scheduling is chosen because all the iterations are given the same workload: computing fixed-dimension local matrix and vector and summing them up to the global terms. However, as the theoretical introduction points out, we have to guarantee that one thread only is managing the global matrix at the same time: this is the reason why we have to add a `#pragma omp critical` clause to every instruction which reads or writes in the global matrix.

4.3 Dirichlet conditions assignment

The kernel of the function is a `for` loop which scans every row of the matrix and overwrites all the elements in some of them. As either one or no row is modified in every iteration, the are no dependencies between them. Theoretically, every iteration could run in parallel: in practice, the maximum number of threads the machine can execute in the same time is a physical limit. Theoretically, the maximum time gain could be achieved:

$$t_{parallel} = (1 - fraction_{parallel}) \cdot t_{serial} + fraction_{parallel} \cdot \frac{t_{serial}}{N} \simeq \frac{t_{serial}}{N} \tag{4.3}$$

The code below shows the whole cycle included in a `#pragma omp paralell` directive.

Algorithm 7 Dirichlet conditions assignment

```
1 void assignDirichletCondition(int meshSize, double g, double* meshPoints,
    double* w, double* b) {
2   int i, j;
3
4 #pragma omp parallel for private(i, j) schedule(dynamic)
5   for (i = 0; i < meshSize; i++) {
6     double x = *(meshPoints + 2 * i);
7     double y = *(meshPoints + 2 * i + 1);
8     if (fabs(x * x + y * y - 1) < TOL) {
9       for (j = 0; j < meshSize; j++) {
10        *(w + meshSize * i + j) = (i == j) ? 1 : 0;
11      }
12      b[i] = g;
13    }
14  }
15 }
```

The description of the algorithm suggest the load may be unfairly distributed among different threads. An iteration where the `if()` clause is true is much more demanding than one where the same condition is false. A true condition leads to n additional operation, where n is the number of vertices of the mesh: moreover, it requires to access a different matrix form the one which the condition is tested on. The cache penalty is likely to be paid. The `schedule(dynamic)` clause faces the problem by assigning a new bunch of iteration to the first thread which completes its job. The aim is keeping the maximum number of threads working until the whole cycle is over. The price to pay is the overhead of the repeated job assignment: however, it is much efficient than keeping one ore more threads inactive.

4.4 Computation of the approximate solution

As the amount of time required to perform gaussian elimination is way less than the one taken by the back substitution, we present only the parallel version of the former. As the serial profiling outlines, this function is the by far the most time-consuming of the whole program, so a dependency analysis is required to better understand which parts of the algorithm can be executed in parallel. The picture 4.1[2] shows which elements of the starting matrix $A \in \mathbb{R}^{n \times n}$ are updated. The kernel is composed by three loops: the first one scans the column of A: at its k-th iteration, it reads the elements in the k-th row and updates all the element $a_{i,j}$ such that $i > k$ and $j \geq k$. As the values in the k-th row have been updated until the $k-1$-th iteration, every iteration is dependent on the previous. Every kind of parallelization is thus forbidden. However, the update is performed via two more loops which scan the sub-matrix $\tilde{A} = \{a_{i,j}\}_{i,j>k}$ - in green in Figure 4.1. The update of a row of $\tilde{A}$ depends only on the elements of

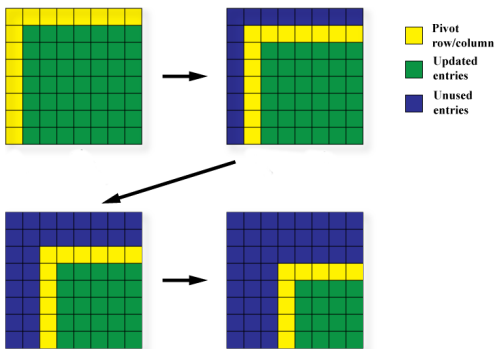


Figure 4.1: Scheme of Gaussian elimination algorithm.

[2] http://riebecca.blogspot.it/2008/12/supercomputing-course-openmp-syntax-and_15.html

that row and on the k-th row of A, so all the rows can be processed in parallel. As all the time-consuming operations are performed inside the parallel loop, we expect the parallel algorithm to last $\frac{1}{N}$ times the serial one, where N is the number of threads. However, a non negligible overhead is introduced by the thread management and synchronisation and the job assignments. All these operations have to be repeated for every iteration of the outer loops. The parallel version of the function is shown below.

Algorithm 8 Computation of the approximate solution

```
1 void gaussianElimination(int n, double* matrix, double b[]){
2   int j, i, k;
3   double aux;
4
5   for (j = 0; j < (n - 1); j++) {
6 #pragma omp parallel for private(aux, i, k) schedule(dynamic)
7     for (i = (j + 1); i < n; i++) {
8       aux = *(matrix + n * i + j) / *(matrix + n * j + j);
9       for (k = j; k < n; k++) {
10        *(matrix + n * i + k) -= (aux * (*(matrix + n * j + k)));
11      }
12      b[i] -= (aux * b[j]);
13    }
14  }
15 }
```

We decide to put the `#pragma omp parallel for` clause on the outer of the two parallelizable loops to avoid too many thread creation and deletion. Following the result of a previous study[3] we choose dynamic scheduling. However, Figure 4.1 shows that the workload changes for every iteration of the outer serial loop, while it remains constant over the iteration of the inner parallel loop. Thus, we believe a static scheduling could outperform a dynamic one, as it prevents the overhead for several job assignment among different threads. Data supporting our prediction are shown in Table 4.1.

Mesh Size	Dynamic	Static
2173	22.951455	19.608431
3447	97.125484	77.959359
6084	513.480326	487.208414

Table 4.1: Static and dynamic execution timing (seconds).

[3] S.F.McGinn & R.E.Shaw, Parallel Gaussian Elimination Using OpenMP and MPI, `http://www3.nd.edu/~zxu2/acms60212-40212-S12/Gaussian-openMP.pdf`

5 Profiling the parallel execution

As we did for the serial execution, we shall now proceed analysing the execution of the parallel code on computer machines. The profiling operations are made in the same way of the serial one, but this time simulations are run on different machines in order to study the speed-up of parallel code with respect to the number of available cores/threads. All interesting parameters related to the processors under consideration are listed below.

Processor	RAM	Clock	Cores	Threads	L1 I/D-cache	L2 cache	L3 cache
i7-2630QM	4 GB	2 GHz	4	8	4x32 KB 8-way	4x256 KB 8-way	6 MB 12-way
i5-4210U	16 GB	1.7 GHz	2	4	2x32 KB 8-way	2x256 KB 8-way	3 MB 12-way
i3-370M	4 GB	2.4 GHz	2	4	2x32 KB 2/4-way	2x256 KB 8-way	3 MB 12-way

Table 5.1: Processors features.

We will consider function by function the effect of the applied parallelization and compare it to the theoretical behaviour predicted before. For each kernel function we will show raw data, trends of computational time and computed speed-up, which is the ratio between serial code computational time and parallel one. Then we will comment the results and discuss whether or not they fit the theoretical prediction.

For each kernel function we will plot the trends of computational time for the different machines, along with the serial one.

5.1 Memory allocation and variable initialisation

As we expected, the overall CPU time the function requires to complete is significantly reduced by the parallel implementation. Apart form noise, the speedup is almost invariant with respect to the mesh size, as we may expect form such a regular pattern of operations. However, we expected a speedup very similar to the number of cores, but the simulation results show that the real gain is almost the half. We find two major causes of the difference: first of all, the number of thread is two times the cores, thanks to Intel hyper-threading. It means that two instruction flows are in execution on the same core: when the first is blocked, the second proceeds. However, this leads to a less significant improvement than two physical cores. On the other hand, a non negligible time is wasted in memory access: this part of the code can hardly be made parallel by the software. So, the real speedup is significant, but not as high as theoretical speculations foresaw.

Mesh Size	i3-370M Serial	i3-370M Parallel	i5-4210U Parallel	i7-2630QM Parallel
25	0.000029	0.005196	0.005953	0.010246
33	0.000022	0.000084	0.000059	0.000012
41	0.000023	0.001351	0.000065	0.000006
75	0.000077	0.005049	0.000087	0.000014
144	0.00017	0.000064	0.000115	0.000033
544	0.00181	0.001664	0.001129	0.000334
2173	0.024207	0.013216	0.010397	0.008686
3447	0.062386	0.032585	0.026124	0.01721
6084	0.187715	0.100648	0.115271	0.055533
13693	1.335043	0.510255	0.434716	0.278116

Table 5.2: Computational time for memory allocation and variable initialisation (seconds).

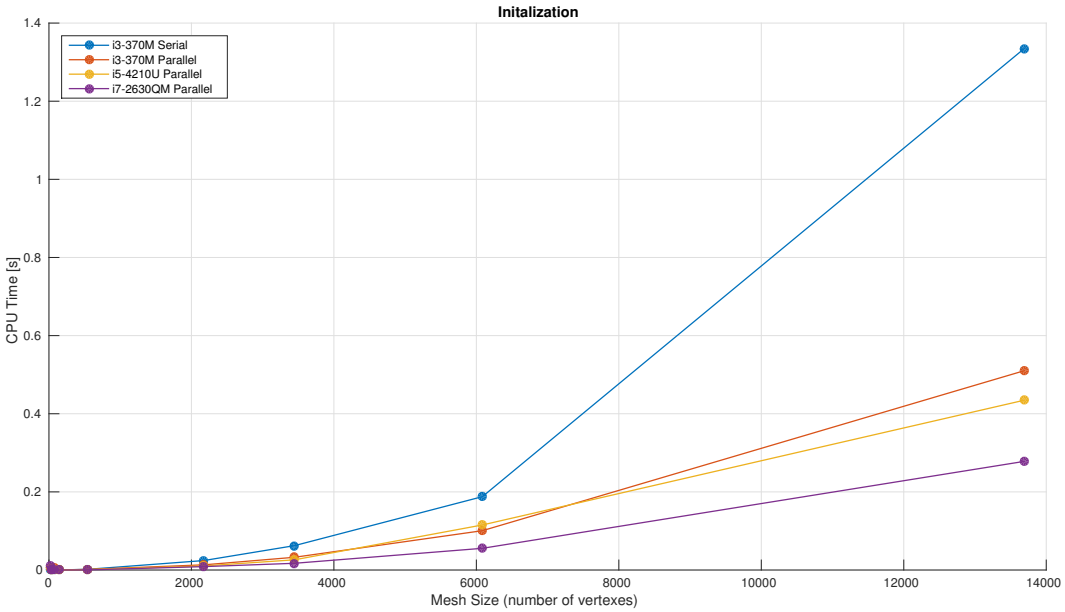


Figure 5.1: Memory allocation and initialisation time over mesh size.

Mesh Size	i3-370M	i5-4210U	i7-2630QM
2173	1.83	2.11	2.72
3447	1.91	2.15	2.80
6084	1.86	1.65	2.71
13693	2.61	2.06	2.71

Table 5.3: Speed-Up for memory allocation and variable initialisation

5.2 Assembling the stiffness matrix and the right-hand side

Being the piece of code which is most difficult to make parallel, as paragraph 4.2 points out, we expect little speedup. However, the parallel code is even slower than the serial one. We believe the critical regions to be the reason: because of the irregular access pattern to the global matrix, most of the time is spent in updating values in RAM memory. This operation cannot be made parallel, so we nullify the gain in computations speed by making the threads wait their turn to enter the critical regions. In order to prove the hypothesis, we run on the i3-370M machine a simulation where the critical region are wiped out. The results are shown in table 5.4.

Mesh Size	Serial	Parallel	Parallel without critical regions
2173	0.001667	0.006988	0.001227
3447	0.003327	0.011121	0.002010
6084	0.007782	0.020323	0.003701
13693	0.026694	0.053064	0.011567

Table 5.4: Execution time with and without critical regions (seconds).

Without the critical regions, the parallel code performs much better than the serial one, while the performance drops when the blocking memory management is imposed. However, wiping the critical regions out is not a feasible solution, as the output of the program which run without them is incoherent with the one of the other executions and with the real solution of the partial differential equation. We have to conclude that, in a real-life situation, such an algorithm is not worth parallelizing without previous modifications.

Mesh Size	i3-370M Serial	i3-370M Parallel	i5-4210U Parallel	i7-2630QM Parallel
25	0.00001	0.000075	0.000012	0.001842
33	0.000013	0.001905	0.000017	0.000116
41	0.000015	0.000101	0.000022	0.000155
75	0.000035	0.000929	0.000042	0.000299
144	0.000042	0.00037	0.000092	0.000602
544	0.000217	0.001505	0.000361	0.002575
2173	0.001667	0.006988	0.00207	0.01076
3447	0.003327	0.011121	0.003153	0.016062
6084	0.007782	0.020323	0.006971	0.028417
13693	0.026694	0.053064	0.021285	0.06478

Table 5.5: Computational time for assembling (seconds).

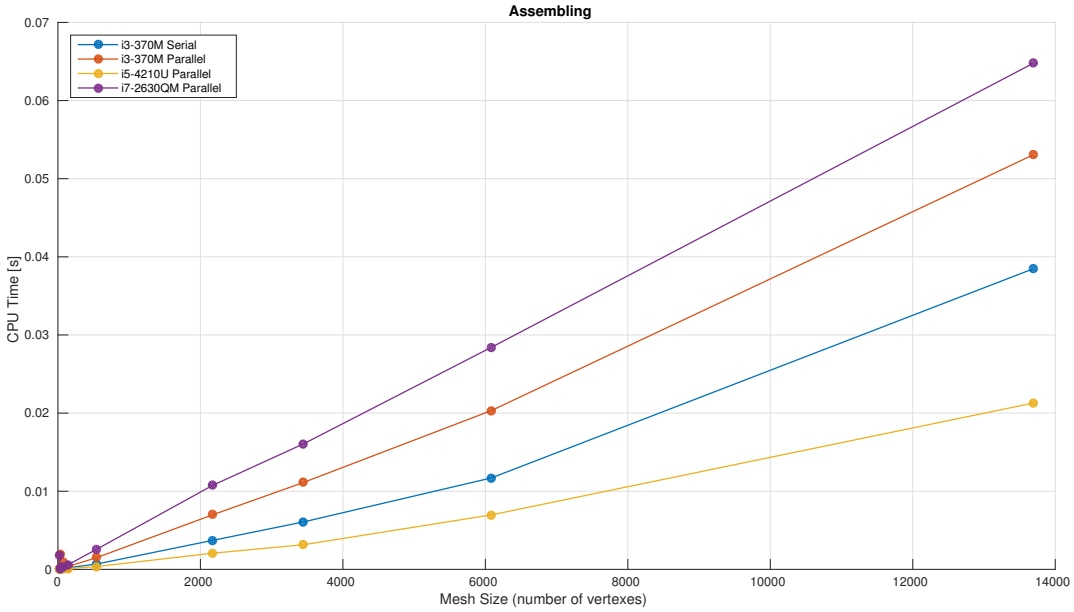


Figure 5.2: Assembling time over mesh size.

Mesh Size	i3-370M	i5-4210U	i7-2630QM
2173	0.23	1.03	0.28
3447	0.29	0.98	0.27
6084	0.38	1.01	0.28
13693	0.50	0.92	0.28

Table 5.6: Speed-Up for assembling

5.3 Dirichlet conditions assignment

Simulations show that the parallel version is faster than the serial one, as we expected from a priori study. By spreading the condition assignment among threads, we obtained a relevant speedup, although not as high as we predicted. Again, this is caused either by hyper-threading, which let two threads run on the same core

alternatively thus decreasing performance, either by overhead, in this case indeed relevant due to repeated assignment caused by `schedule(dynamic)`clause. As a result, we see a speedup which is approximately half the number of threads.

Mesh Size	i3-370M Serial	i3-370M Parallel	i5-4210U Parallel	i7-2630QM Parallel
25	0.00001	0.000007	0.000036	0.000006
33	0.000013	0.003206	0.000032	0.000005
41	0.000015	0.000005	0.000034	0.000005
75	0.000035	0.000011	0.000041	0.000006
144	0.000042	0.000017	0.000051	0.00001
544	0.000217	0.000105	0.000136	0.000045
2173	0.001667	0.000798	0.000538	0.000639
3447	0.003327	0.001593	0.00103	0.000587
6084	0.007782	0.00373	0.002519	0.001411
13693	0.026694	0.013636	0.009478	0.004924

Table 5.7: Computational time for Dirichlet conditions assignment (seconds).

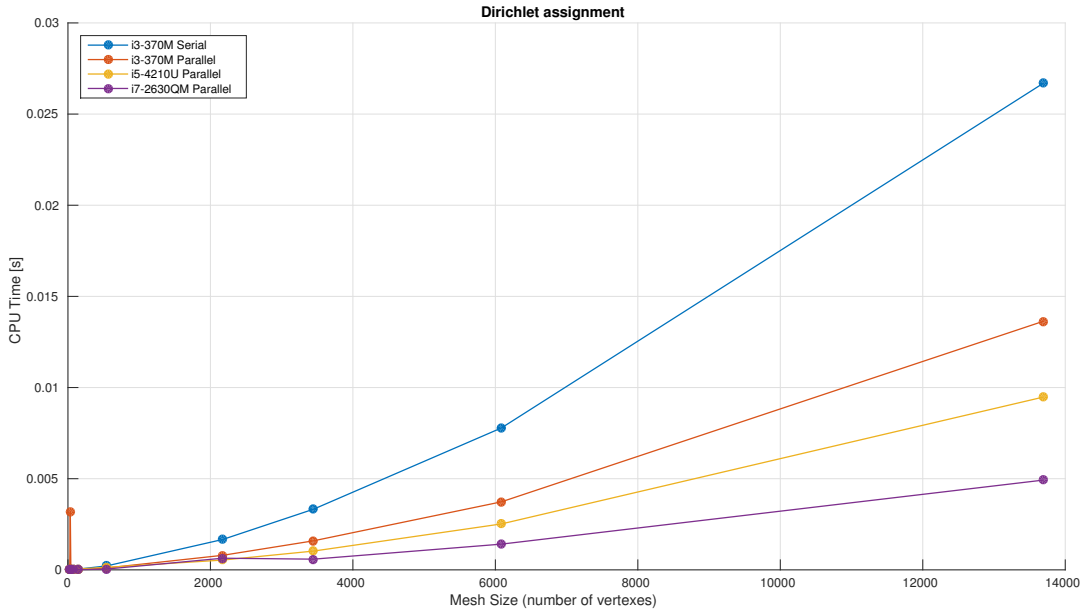


Figure 5.3: Dirichlet condition assignment time over mesh size.

Mesh Size	i3-370M	i5-4210U	i7-2630QM
2173	2.08	1.69	1.49
3447	2.08	1.77	3.27
6084	2.08	1.91	3.11
13693	1.95	1.53	3.00

Table 5.8: Speed-Up for Dirichlet conditions assignment

5.4 Computation of the approximate solution

The results below show how the parallel version behaved in simulations with respect to the serial one: a non negligible speedup is obtained, though not as high as one could expect. In addition to what has already been told, it is important to notice that the parallelization directive stays inside the outer loop, thus requiring more overhead and decreasing the speedup. This is also made worse by the dynamic job assignment. Resulting data and figures are shown below.

Mesh Size	i3-370M Serial	i3-370M Parallel	i5-4210U Parallel	i7-2630QM Parallel
25	0.000195	0.000112	0.000648	0.00006
33	0.000415	0.002478	0.000925	0.000096
41	0.000766	0.000223	0.001608	0.000135
75	0.002327	0.002265	0.006143	0.000739
144	0.015817	0.005991	0.010579	0.01123
544	0.630275	0.309024	0.209555	0.116902
2173	40.413426	19.283026	11.97183	6.636739
3447	161.381916	76.948164	47.403249	24.503626
6084	890.467981	427.581916	259.603595	141.665447
13693	10079.206121	6045.163729	2939.631746	2003.69768

Table 5.9: Computational time for solving the system (seconds).

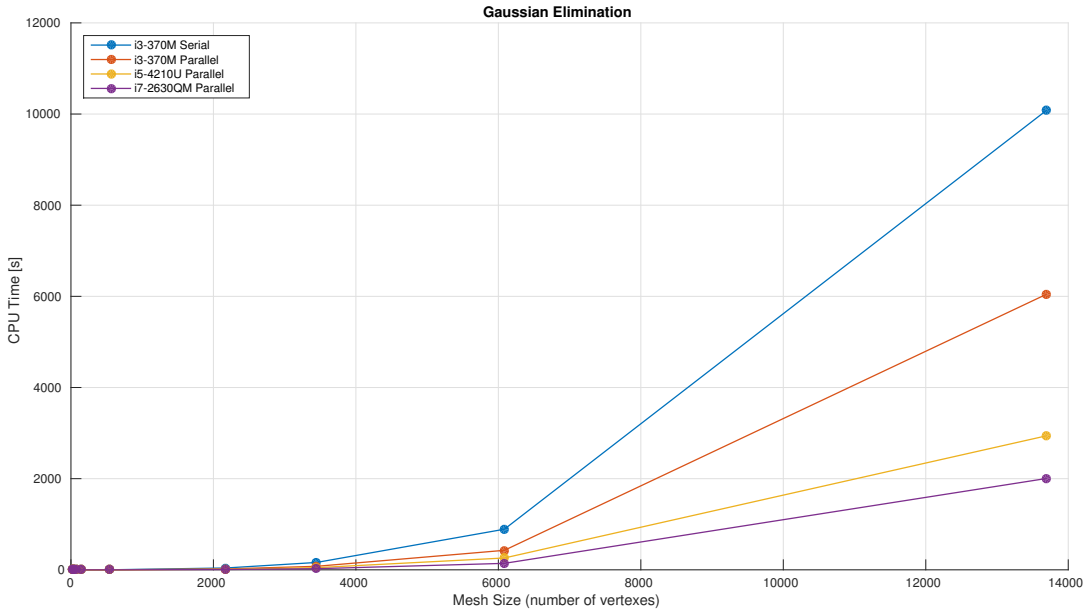


Figure 5.4: Gaussian elimination time over mesh size.

Mesh Size	i3-370M	i5-4210U	i7-2630QM
2173	2.09	1.31	3.45
3447	2.09	1.30	3.96
6084	2.08	1.31	3.62
13693	1.66	1.30	2.77

Table 5.10: Speed-Up for solving the system

6 Beyond instruction parallelism

As a final experiment, we try to let the gcc compiler optimise the machine code at its best. By default, the GNU compiler goes for compilation speed[4] and neglect assembly optimisation. However, when we consider the Finite Elements algorithm, compilation speed is a minor concern if compared to execution speed. Looking at the code, we con identify several possible ways to optimise the flow of machine instructions:

- All the functions we presented and analysed in the previous chapters involves lots of floating points operation: some of them are performed in the same way on consecutive elements of a vector. We could get a double throughput if we processed two elements at once via vectorised operations.
- The kernel of most function is made of nested for loops, each of them made of a small amount of instruction. At assembling level, it means a big waste in time for the control instruction (testing the counter, jumping, ...). A solution is loop unrolling, that is merging more than one small iteration into a single longer one. This procedure is limited by data dependency and by the actual number of available registers.
- The data the algorithm requires are stored in large data structure in ram memory. Even when cache hierarchy works at its best - we pointed out it is not always the case - every access to a resource which is outside the core is expensive. Optimising the memory access could lead to a big performance improvement.

A human optimisation of such a complex procedure would be difficult and may lead to several mistakes. However, gcc compiler provides options to turn on automatic assembly optimisation. We launched a compilation on the i7-2630QM machine with the following command: `gcc -Wall -O3 -msse2 -fopenmp main.c -o main`. The `-O3` flag make the compiler try all the improvements in the previous list: the `-msse2` options let the compiler use special 128-bit-long registers to perform vectorised operations. We profiled the execution and we shall present a comparison with the unoptimised code.

6.1 A bit of insight: profiling the optimised code

We provide a tabular and a graphical visualisation of the performance of the optimised code over the serial one we considered until now - Table 6.1 and Figure 6.1. As we may expect, a substantial improvement has been achieved in all the functions we considered. However, the most important gain concerns the Gaussian elimination algorithm. We try to understand why the improvement is so huge asking the compiler to log vectorizations and loop unrolling. Among the output, we get the following lines:

```
1 parallelMain.c:121: note: vectorized 1 loops in function.
2 parallelMain.c:130: note: LOOP VECTORIZED.
```

Line 130 is part of the function implementing gaussian elimination and contains the inner loop which scans one of the columns of the submatrix to be updated. Since this loop is repeated for all the columns of the stiffness matrix and for all the rows of the submatrix to update, a speedup concerning this fragment of code is critical for the overall performance. Even if a deep analysis of the assembly the compiler produces is beyond the aim of this report, we look for traces of the vectorization. In a piece of code which is supposed to be part of the translation of the gaussian elimination algorithm, we find the following lines:

```
1 movsd  xmm0,QWORD PTR [rdx+rax*1]
2 movhpd xmm1,QWORD PTR [rcx+rax*1+0x8]
3 movhpd xmm0,QWORD PTR [rax+rdx*1+0x8]
4 mulpd  xmm1,xmm3
5 subpd  xmm0,xmm1
```

In Intel Instruction Set Architecture (ISA), `xmm*` registers are 128-bit long and are used to perform vectorised operations. Without any modification of the code, the compiler exploited its knowledge of the program and the hardware of the machine to speed mathematical operations up. The only help we give it is the `__attribute__(aligned(16))` clause we add to vectors declaration to make them 16-byte aligned in memory and enhance the effectiveness of vectorised operations.

In the output of the optimised compilation, we find the trace of several loop unrollings:

```
1 parallelMain.c:75: note: Completely unroll loop 2 times
2 parallelMain.c:48: note: Completely unroll loop 3 times
3 parallelMain.c:214: note: Completely unroll loop 3 times
```

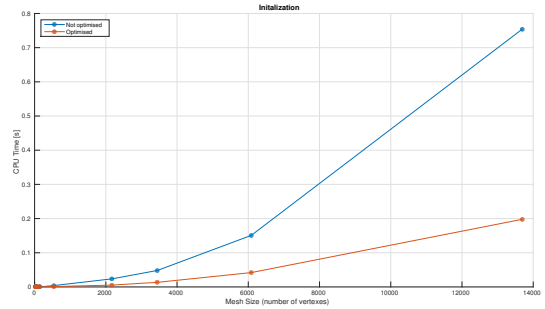
[4] GNU gcc official documentation, `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

The first and the second loop are part of the secondary functions which compute local stiffness matrix, while the latter is placed inside the assembling procedure.

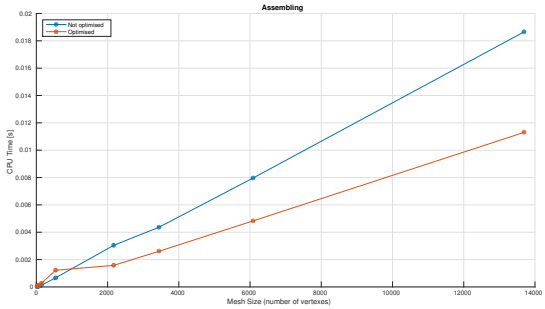
Altogether, the optimisation of the assembling let the program gain a greater speedup than the mere parallelization of sections of code.

Mesh Size	Initialisation	Assembling	Dirichlet assignment	Gaussian Elimination
2173	4.38	1.93	2.73	5.84
3447	3.58	1.67	2.84	6.19
6084	3.59	1.65	2.82	6.03
13693	3.81	1.65	2.90	5.80

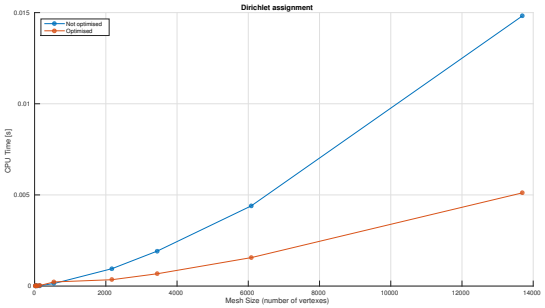
Table 6.1: Speedup of optimised code over non optimised serial.



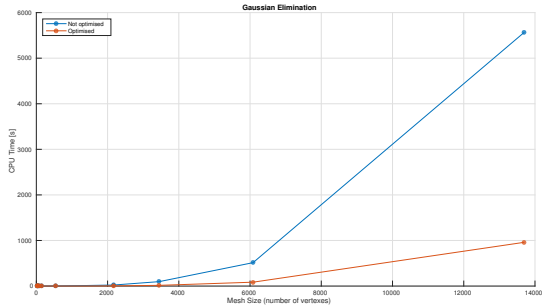
(a) Initialisation.



(b) Assembling.



(c) Dirichlet condition assignment.



(d) Gaussian elimination

Figure 6.1: Optimised and non optimised code execution time over mesh size.