A PARALLEL IMPLEMENTATION OF A FEM SOLVER IN SCILAB

Author: *Massimiliano Margonari*

Keywords. Scilab; Open source software; Parallel computing; Mesh partitioning, Heat transfer equation.

Abstract: In this paper we describe a parallel application in Scilab to solve 3D stationary heat transfer problems using the finite element method. Two benchmarks have been done: we compare the results with reference solutions and we report the speedup to measure the effectiveness of the approach.

Contacts m.margonari@openeering.com

1. Introduction

Nowadays many simulation software have the possibility to take advantage of multi-processors/cores computers in the attempt to reduce the solution time of a given task. This not only reduces the annoying delays typical in the past, but allows the user to evaluate larger problems and to do more detailed analyses and to analyze a greater number of scenarios.
Engineers and scientists who are involved in simulation activities are generally familiar with the terms "*high performance computing"* (HPC). These terms have been coined to indicate the ability to use a powerful machine to efficiently solve hard computational problems.
One of the most important keywords related to the HPC is certainly *parallelism*. The total execution time can be reduced if the original problem can be divided into a given number of subtasks. Total time is reduced because these subtasks can be tackled concurrently, that means in parallel, by a certain number of machines or cores.
To completely take advantage of this strategy three conditions have to be satisfied: the first one is that the problem we want to solve has to exhibit a parallel nature or, in other words, it should be possible to reformulate it in smaller independent problems, which can be solved simultaneously. After, these reduced solutions can be opportunely combined to give the solution of the original problem.
Secondly, the software has to be organized and written to exploit this parallel nature. So typically, the *serial version* of the code has to be modified where necessary to this aim.
Finally, we need the right hardware to support this strategy.
If one of these three conditions is not fulfilled, the benefits of parallelization could be poor or even completely absent. It is worth mentioning that not all the problems arising from engineering can be solved effectively with a *parallel approach,* being their associated numerical solution procedure intrinsically *serial*.
One parameter which is usually reported in the technical literature to judge the goodness of a parallel implementation of an algorithm or a procedure is the so-called *speedup*. The speedup is simply defined as the ratio between the execution time on a single core machine and the execution time on a multicore. Being *p* the number of cores used in the computation this ratio is defined as $S = T_1/T_p$. Ideally, we would like to have a speedup not lower than the number of cores: unfortunately this does not happen mainly, but not only, because some serial operations have to be performed during the solution. In this context it is interesting to mention the *Amdahl law* which bounds the theoretical speedup that can be obtained, given the percentage of serial operations ($f \in [0,1]$) that has to be globally performed during the run. It can be written as:

$$S \leq \frac{1}{f + (1-f)/p}$$

It can be easily understood that the speedup *S* is strongly (and badly) influenced by *f* rather than by *p*. If we imagine to have an ideal computer with infinite cores ($p = \infty$) and implement an algorithm whit just the 5% of operations that have to be performed serially ($f = 0.05$) we get a speedup of 20 as a maximum. This clearly means that it is worth to invest in algorithms rather than in brutal force… Someone in the past has moved criticism to this law, saying that it is too pessimistic and unable to correctly estimate the real theoretical speedup: in any case, we think that the most important lesson to learn is that a good algorithm is much more important that a good machine.

Figure 1: The personal computer we all would like to have… (NASA Advanced Supercomputing (NAS) Facility at the Ames Research Facility)

As said before, many commercial software propose since many years the possibility to run parallel solutions. With a simple internet search it is quite easy to find some benchmarks which advertize the high performances and high speedup obtained using different architectures and solving different problems. All these noticeable results are usually the result of a very hard work of code implementation.
Probably the most used communication protocols to implement parallel programs, through opportunely provided libraries, are the MPI (Message Passing Interface), the PVM (Parallel Virtual Machine) and the openMP (open Message Passing): there certainly are other protocols and also variants of the aforementioned ones, such an the MPICH2 or HPMPI, which gained the attention of the programmers for some of their features.
As the reader has probably seen, in all the acronyms listed above there is a letter “P”. With a bit of irony we could say that it always stands for “problems”, in view of the difficulties that a programmer has to tackle when trying to implement a parallel program using such libraries. Actually, the use of these libraries is often and only a matter for expert programmers and they cannot be easily accessed by engineers or scientists who want to easily cut the solution time of their applications.
In this paper we would like to show that a naïve but effective parallel application can be implemented without a great programming effort and without using any of the above mentioned protocols. We used the Scilab platform (see [1]) because it is free and it provides a very easy and fast way to implement applications. The main objective of this work is actually to show that it is possible to implement a parallel application and solve large problems efficiently (e.g.: with a good speedup) in a simple way rather than to propose a super-fast application.
To this aim, we choose the stationary heat transfer equation written for a three dimensional domain together with appropriate boundary conditions. A standard Galerkin finite element (see [4]) procedure is then adopted and implemented in Scilab in such a way to allow a parallel execution.
This represents a sort of elementary “brick” for us: more complex problems involving partial differential equations can be solved starting from here, adding new features whenever necessary.

2. The stationary heat transfer equation

As mentioned above, we decided to consider the stationary and linear heat transfer problem for a three-dimensional domain Ω. Usually it is written as:

$$-\nabla \cdot (\mathrm{k}\nabla T) = f \qquad in\ \Omega \tag{1}$$

together with Dirichlet, Neumann and Robin boundary conditions, which can be expressed as:

$$\begin{aligned} T &= \bar{T} \qquad on\ \Gamma_{\mathrm{T}} \\ -k\frac{\partial T}{\partial \hat{n}} &= \bar{Q} \qquad on\ \Gamma_{\mathrm{Q}} \\ -k\frac{\partial T}{\partial \hat{\boldsymbol{n}}} &= h(T - \bar{T}_E) \qquad on\ \Gamma_{\mathrm{C}} \end{aligned} \tag{2}$$

The conductivity k is considered as constant, while f represents an internal heat source. On some portions of the domain boundary we can have imposed temperatures $\bar{T}$, given fluxes $\bar{Q}$ and also convections with an environment characterized by a temperature $\overline{T_E}$ and a convection coefficient h.
The discretized version of the Galerkin formulation for the above reported equations leads to a system of linear equations which can be shortly written as:

$$[K]\{T\} = \{F\} \tag{3}$$

The coefficient matrix $[K]$ is symmetric, positive definite and sparse. This means that a great amount of its terms are identically zero. The vector $\{T\}$ and $\{F\}$ collect the unknown nodal temperatures and nodal equivalent loads.
If large problems have to be solved, it immediately appears that an effective strategy to store the matrix terms is needed. In our case we decided to store in memory the non-zero terms row-by-row in a unique vector opportunely allocated, together with their column positions: in this way we also access efficiently the terms. We decided to not take advantage of the symmetry of the matrix (actually, only the upper or lower part could be stored, with a half of required storage) to simplify a little the implementation. Moreover, this allows us to potentially use the same pieces of code without any change, for the solution of problems which lead to a not-symmetric coefficient matrix.
The matrix coefficients, as well as the known vector, can be computed in a standard way, performing the integration of known quantities over the finite elements in the mesh. Without any loss of generality, we decided to only use ten-noded tetrahedral elements with quadratic shape functions (see [4] for more details on finite elements).
The solution of the resulting system is performed through the preconditioned conjugate gradient (PCG) (see [5] for details). In Figure 2 a pseudo-code of a classical PCG scheme is reported: the reader should observe that the solution process firstly requires to compute the product between the preconditioner and a given vector (*) and secondly the product between the system matrix and another known vector (**). This means that the coefficient matrix (and also the preconditioner) is not explicitly required, as it is when using direct solvers, but it could be not directly computed and stored.
This is a key feature of all the iterative solvers and we certainly can take advantage of it, when developing a parallel code.

```
Let ${x_0}$ the initial guess

$\{r\} = \{b\} - [A]\{x_0\}$
$i = 0$

While not converged do

    $i = i + 1$

    $\{z\} = [M]^{-1}\{r\}$                                (*)

    If $i > 1$ then
    $\{p\} = \{z\} + \frac{\rho}{\rho_1}\{p\}$
    else
    $\{p\} = \{z\}$
    end

    $\{q\} = [K]\{p\}$                                     (**)

    $\alpha = \frac{\rho}{\{p\} \cdot \{q\}}$

    $\{x\} = \{x\} + \alpha\{p\}$

    $\{r\} = \{r\} - \alpha\{q\}$

    $\rho_1 = \rho$

    $err = \|\{r\}\| / \|\{b\}\|$

    If $err < tol$ converged
end
```

Figure 2: The pseudo-code for a classical preconditioned conjugate gradient solver. It can be noted that during the iterative solution it is required to compute two matrix-vector products involving the preconditioner M (*) and the coefficient matrix K (**).

The basic idea is to partition the mesh in such a way that, more or less, the same number of elements are assigned to each core (process) involved in the solution, to have a well balanced job and therefore to fully exploit the potentiality of the machine. In this way each core fills a portion of the matrix and it will be able to compute some terms resulting from the matrix-vector product, when required. It is quite clear that some coefficient matrix rows will be slit on two or more processes, being some nodes shared by elements on different cores.

The number of *rows overlap* coming out from this fact strongly depends on the way we partition (* and **) the mesh. The ideal partition produces the minimum overlap, leading to the lesser number of non-zero terms that each process has to compute and store.

In other words, the efficiency of the solution process can depends on how we partition the mesh. To solve this problem, which really is a hard problem to solve, we decided to use the *partition functionality* of gmsh (see [2]) which allows the user to partition a mesh using a well-known library, the METIS (see [3]), which has been explicitly written to solve this kind of problems. The resulting mesh partition is certainly close to the best one and our

solver will use it when spreading the elements to the parallel processes.
An example of mesh partition performed with METIS is plot in Figure 4, where a car model mesh is considered: the elements have been drawn with different colors according to their partition. This kind of partition is obviously suitable when the problem is run on a four cores machine.
As a result, we can imagine that the coefficient matrix is split row-wise and each portion filled by a different process running concurrently with the others: then, the matrix-vector products required by the PCG can be again computed in parallel by different processes. The same approach can be obviously extended to the preconditioner and to the post-processing of element results.
For sake of simplicity we decided to use a Jacobi preconditioner: this means that the matrix $[M]$ in Figure 2 is just the main diagonal of the coefficient matrix. This choice allows us to trivially implement a parallel version of the preconditioner but it certainly produces poor results in terms of convergence rate. The number of iterations required to converge is usually quite high and it could be reduced adopting a more effective strategy. For this reason the solver will be hereafter addressed to as JCG and no more as PCG.

3. A brief description of the solver structure

In this section we would like to briefly describe the structure of our software and highlight some key points. The Scilab 5.2.2 platform has been used to develop our FEM solver: we only used the tools available in the standard distribution (i.e.: avoiding external libraries) to facilitate the portability of the resulting application and eventually to allow a fast translation to a compiled language.
A *master* process governs the run. It firstly reads the mesh partition, organizes data and then starts a certain number of *slave* parallel processes according to the user request. At this point, the parallel processes read the mesh file and load the information needed to fill their own portion of the coefficient matrix and known vector.
Once the slave processes have finished their work the master starts the JCG solver: when a matrix-vector product has to be computed, the master process asks to the slave processes to compute their contributions which will be appropriately summed together by the master.
When the JCG reaches the required tolerance the post-processing phase (e.g.: the computation of fluxes) is performed in parallel by the slave processes. The solution ends with the writing of results in a text file.
It appears that a *communication protocol* is mandatory to manage the run. We decided to use binary files to broadcast and receive information from the master to the slave processes and conversely.
The slave processes are able to wait for the binary files and consequently read them: once the task (e.g.: the matrix-vector product) has been performed, they write the result in another binary file which will be read by the master process.
This way of managing communication is very simple but certainly not the best from an efficiency point of view: writing and reading files, even if binary files, could take a not-negligible time. Moreover, the speedup is certainly bad influenced by this approach.
All the models proposed in the followings have been solved on a Linux 64 bit machine equipped with 8 cores and 16 Gb of shared memory. It has to be said that our solver does not necessarily require so powerful machines to run: the code has been actually written and run on a common Windows 32 bit dualcore notepad.

4. A first benchmark: the Mach 4 model

A first benchmark is proposed to test our solver: we downloaded from the internet a funny CAD model of the Mach 4 car (see the Japanese anime *Mach Go Go Go*), produced a mesh of it and defined a heat transfer problem including all kind of boundary conditions.
Even if the problem is not a real-case application, this is useful for our goal. The aim is to have a sufficiently large and non trivial model to solve on a multicore machine, compare the results with those obtained with a commercial software and measure the speedup factor.
In Table 1 some data pertaining to the mesh has been reported. The same mesh has been solved with our solver and with Ansys Workbench, for comparison purposes.
In Table 2 the time needed to complete the analysis (Analysis time), to compute the system matrix and vector terms (System fill-in time) and the time needed to solve the system with the JCG are reported together with their speedup. The termination accuracy has been always set to 10^{-6}: with this set up the JCG performs 1202 iterations to converge. It immediately appears that the global speedup is strongly influenced by the JCG solution phase, which does not scale as well as the fill-in phase. This is certainly due to the fact that, during the JCG phase, the parallel processes have to communicate much more than during other phases. A guess-solution vector has actually to be written at each iteration and the result of the matrix vector product has to be written back to the master process by the parallel runs. The adopted communication protocol, which is extremely simple and easy to implement, shows here all its limits. However, we would like to underline that the obtained speedup are more than satisfactory.
In Figure 5 the temperature field computed by Ansys Workbench (top) and the same quantity obtained with our solver (bottom) working with the same mesh are plotted. No appreciable difference is present: the same can be substantially said looking the fluxes, taking into account the fact that a very simple, and probably not the best, *flux recovery procedure* has been implemented in our solver. The longitudinal flux computed with the two solvers is reported in Figure 6.

n° of nodes	n° of tetrahedral elements	n° of unknowns	n° of nodal imposed temperatures
511758	317767	509381	2377

Table 1: Some data pertaining to the Mach 4 model are proposed in this table.

n° of cores	Analysis time [s]	System fill-in time [s]	JCG time [s]	Analysis speedup	System fill-in speedup	JCG speedup
1	6960	478	5959	1.00	1.00	1.00
2	4063	230	3526	1.71	2.08	1.69
3	2921	153	2523	2.38	3.12	2.36
4	2411	122	2079	2.89	3.91	2.87
5	2120	91	1833	3.28	5.23	3.25
6	1961	79	1699	3.55	6.08	3.51
7	1922	68	1677	3.62	7.03	3.55
8	2093	59	1852	3.33	8.17	3.22

Table 2: Mach 4 benchmark. The table collects the times needed to solve the model, to perform the system fill-in and to solve the system through the JCG. The speedups are also reported in the right part of the table.

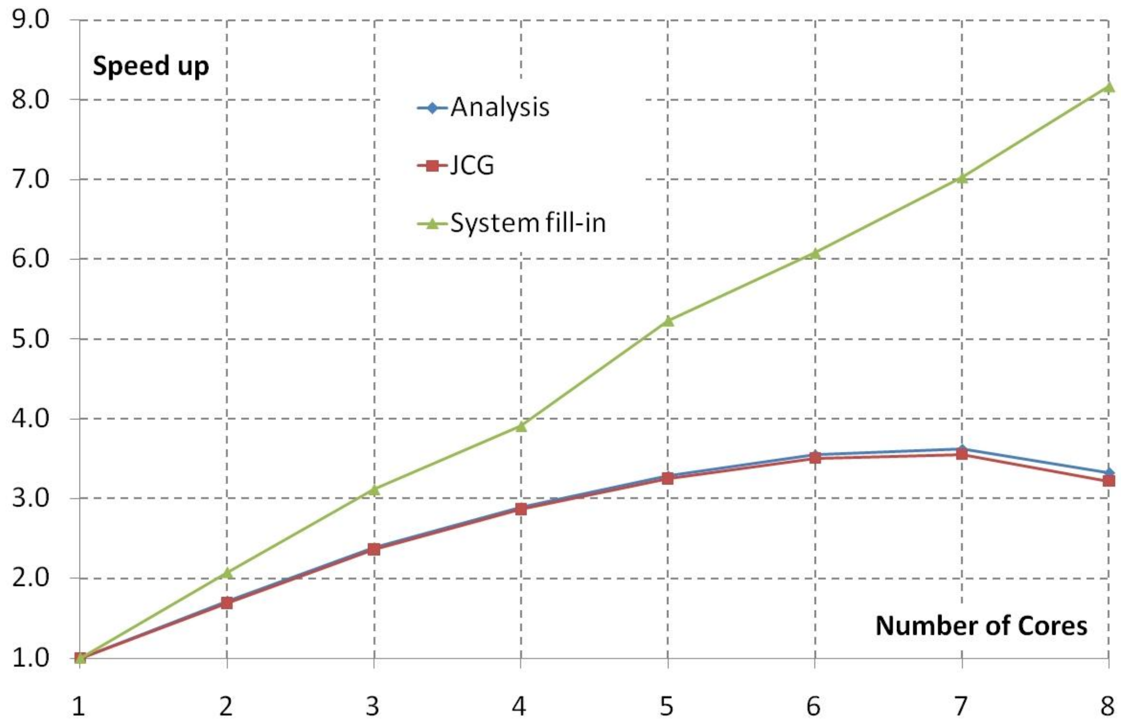


Figure 3: The speedup values collected in Table 2 have been plotted here against the number of cores.

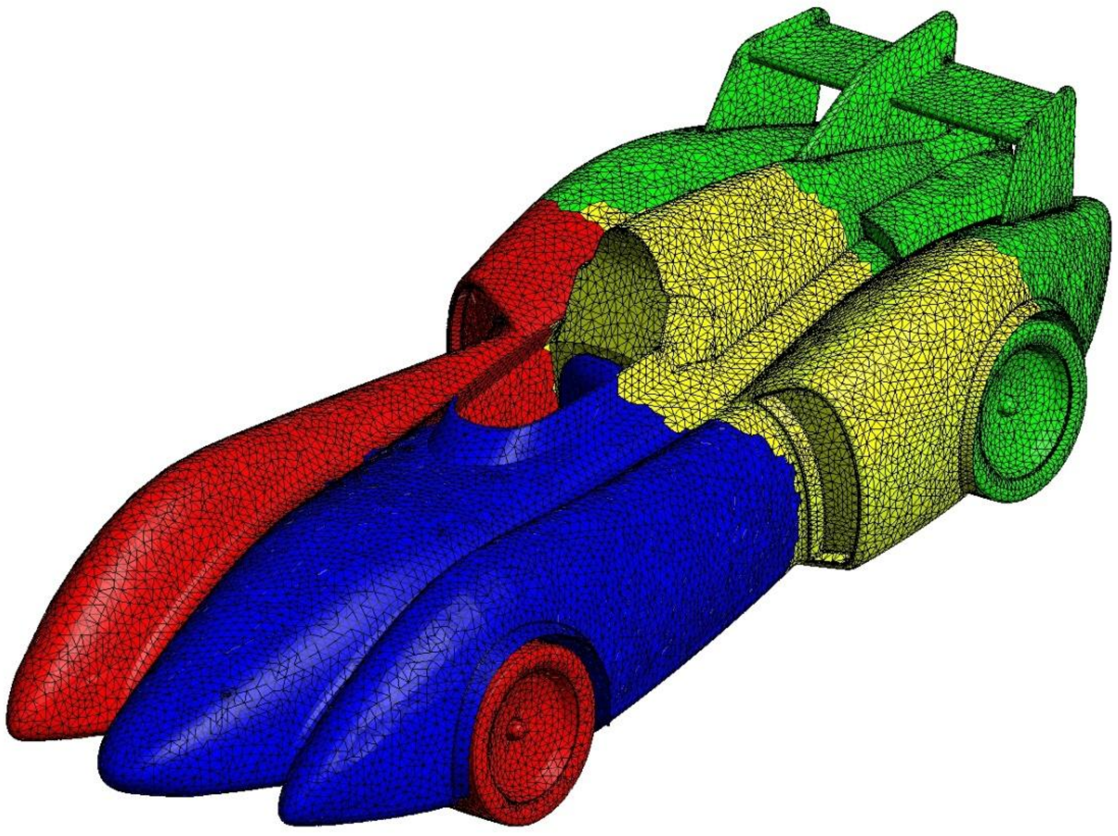


Figure 4: The Mach 4 mesh has been divided in 4 partitions (see colors) using the METIS library available in gmsh. This mesh partition is obviously suitable for a 4 cores run.

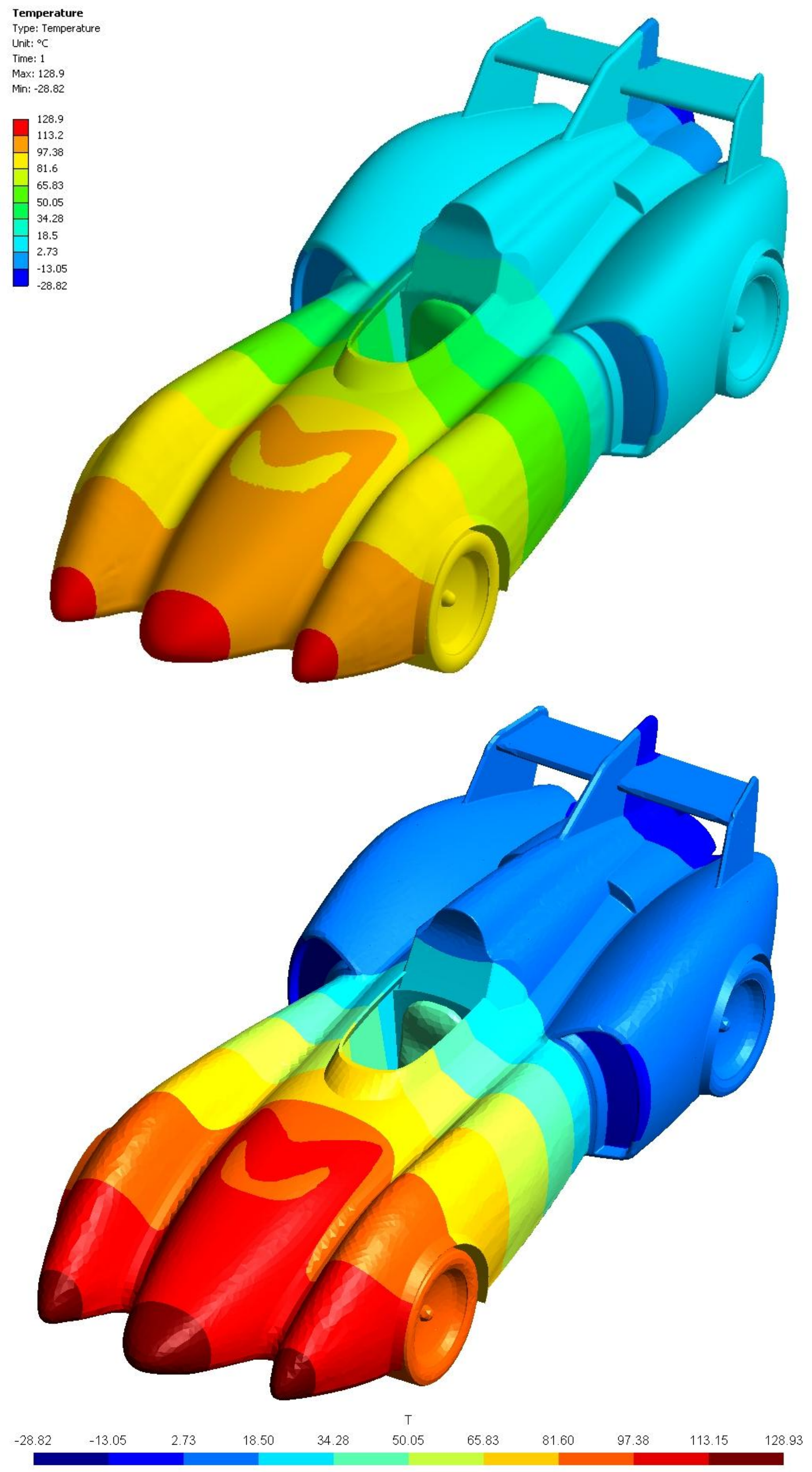
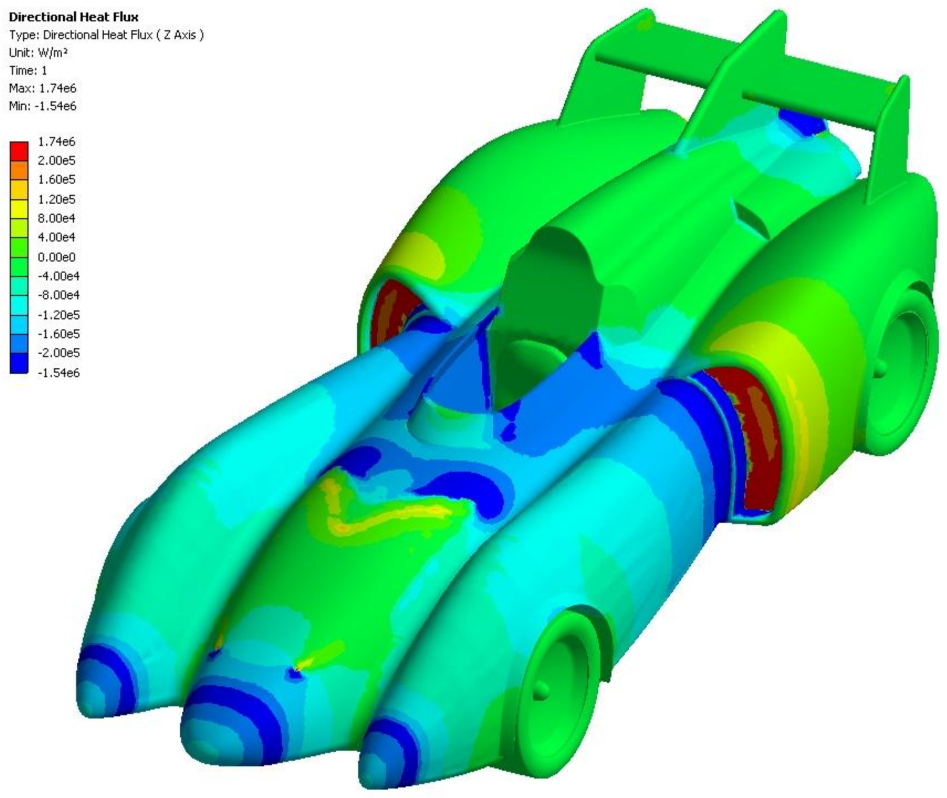


Figure 5: Mach 4 model: the temperature field computed with Ansys Workbench (top) and the same quantity computed with our solver (bottom). No appreciable differences are present. Note that different colormaps have been used.



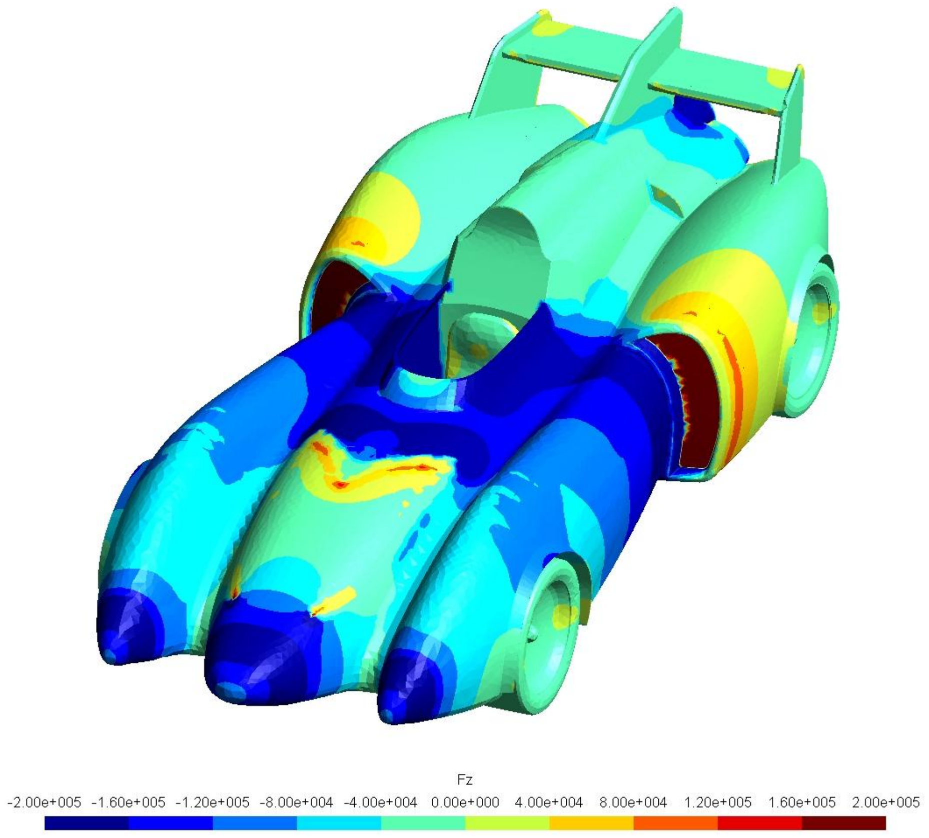


Figure 6: Mach 4 model: the nodal flux in the longitudinal direction computed with Ansys Workbench (top) and the same quantity computed with our solver (bottom). Differences in results are probably due to the naïve flux recovery strategy implemented in our solver. Note that different colormaps have been used.

5. A second benchmark: the motorbike engine model

The second benchmark involves the model of a motorbike engine (also in this case the CAD file has been downloaded from the internet) and the same steps already performed for the Mach 4 model have been repeated. The model is larger than before (see Table 3) and it can be seen in Figure 8, where the grid is plotted. However, it has to be mentioned that conceptually the two benchmarks have no differences; the main concern was also in this case to have a model with a non-trivial geometry and boundary conditions.
The final termination accuracy for the JCG has been set to 10^{-6} reaching convergence after 1380 iterations.
The Table 4 is analogous to Table 2: the time needed to complete different phases of the job and the analysis time are reported, as obtained for runs performed with increasing number of parallel processes involved.
Also in this case, the trend in the reduction of time with the increase of number of cores seems to follow the previous law (see Figure 7). The run with 8 parallel processes does not perform well because the machine has only 8 cores and we start up 9 processes (1 master and 8 slaves): this certainly wastes the performance.
In Figure 9 a comparison between the temperature field computed with Ansys Workbench (top) and our solver (bottom) is proposed. Also in this occasion no significant differences are presents.

n° of nodes	n° of tetrahedral elements	n° of unknowns	n° of nodal imposed temperatures
2172889	1320374	2136794	36095

Table 3: Some data pertaining to the motorbike engine model.

n° of cores	Analysis time [s]	System fill-in time [s]	JCG time [s]	Analysis speedup	System fill-in speedup	JCG speedup
1	33242	2241.0	28698	1.00	1.00	1.00
2	20087	1116.8	17928	1.65	1.60	2.01
3	14679	744.5	12863	2.26	2.23	3.01
4	11444	545.6	9973	2.90	2.88	4.11
5	9844	440.9	8549	3.38	3.36	5.08
6	8694	369.6	7524	3.82	3.81	6.06
7	7889	319.7	6813	4.21	4.21	7.01
8	8832	275.7	7769	3.76	3.69	8.13

Table 4: Motorbike engine benchmark. The table collects the times needed to solve the model (Analysis time), to perform the system fill-in (System fill-in) and to solve the system through the JCG, together with their speedup.

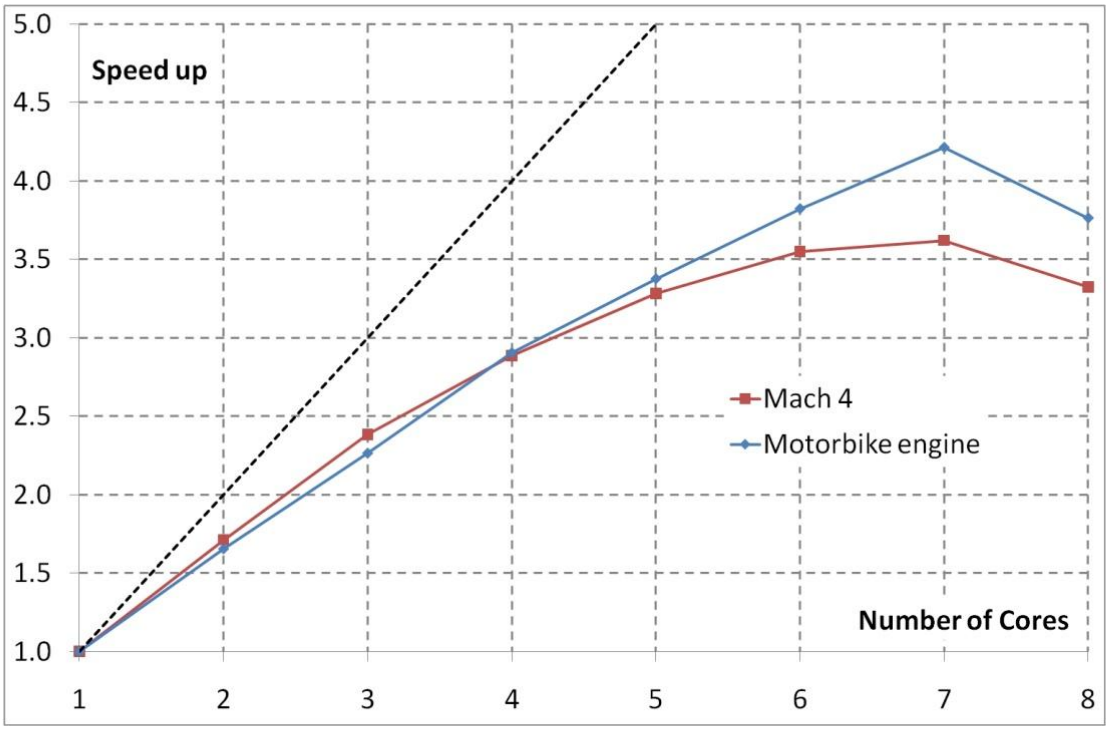


Figure 7: A comparison between the speedup obtained with the two benchmarks. The ideal speedup (the main diagonal) has been highlighted with a black dashed line. In both cases it can be seeing that the speedup follow the same roughly linear trend, reaching a value between 3.5 and 4 when using 6 cores. The performance drastically deteriorates when involving more than 6 cores probably because the machine where runs were performed has only 8 cores.

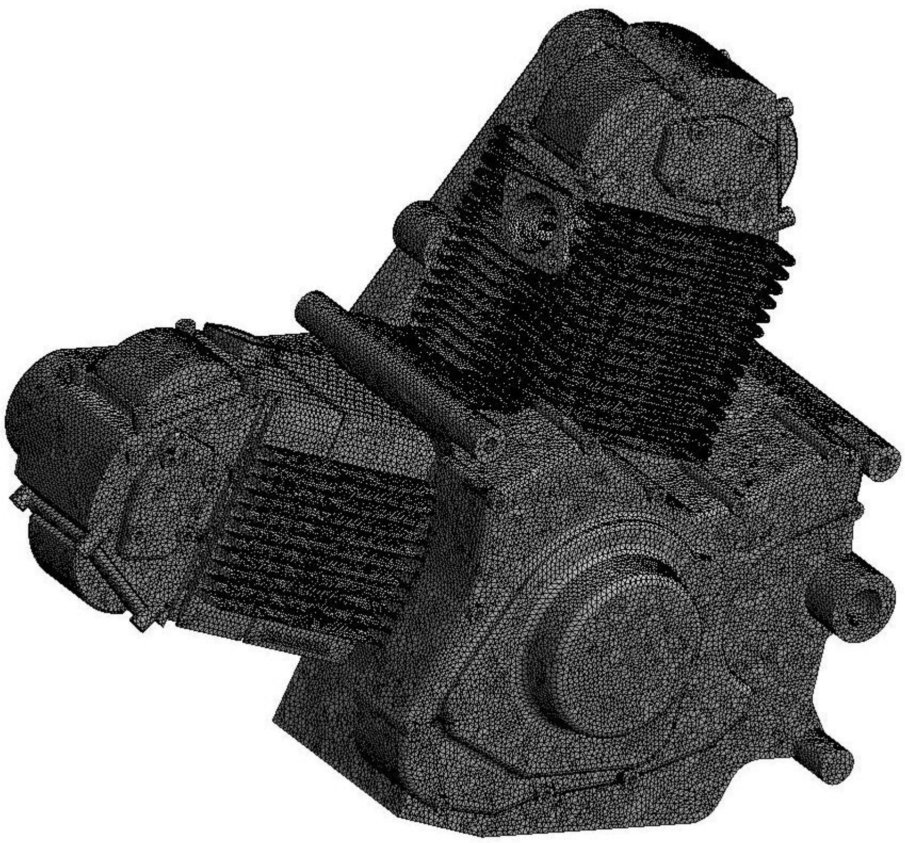
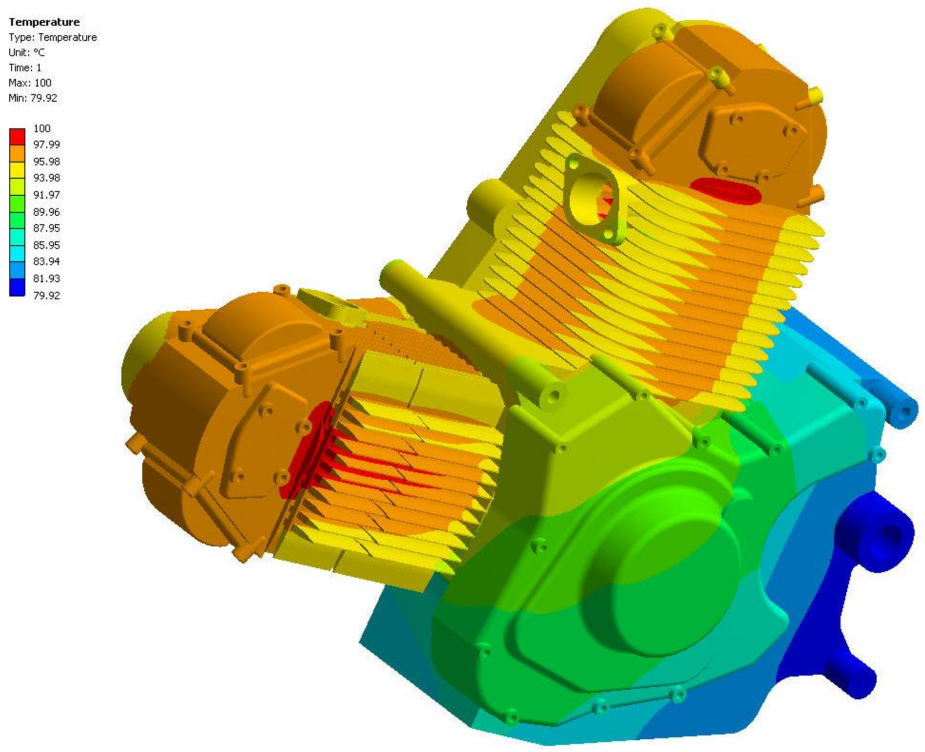


Figure 8: The motorbike engine mesh used for this second benchmark.



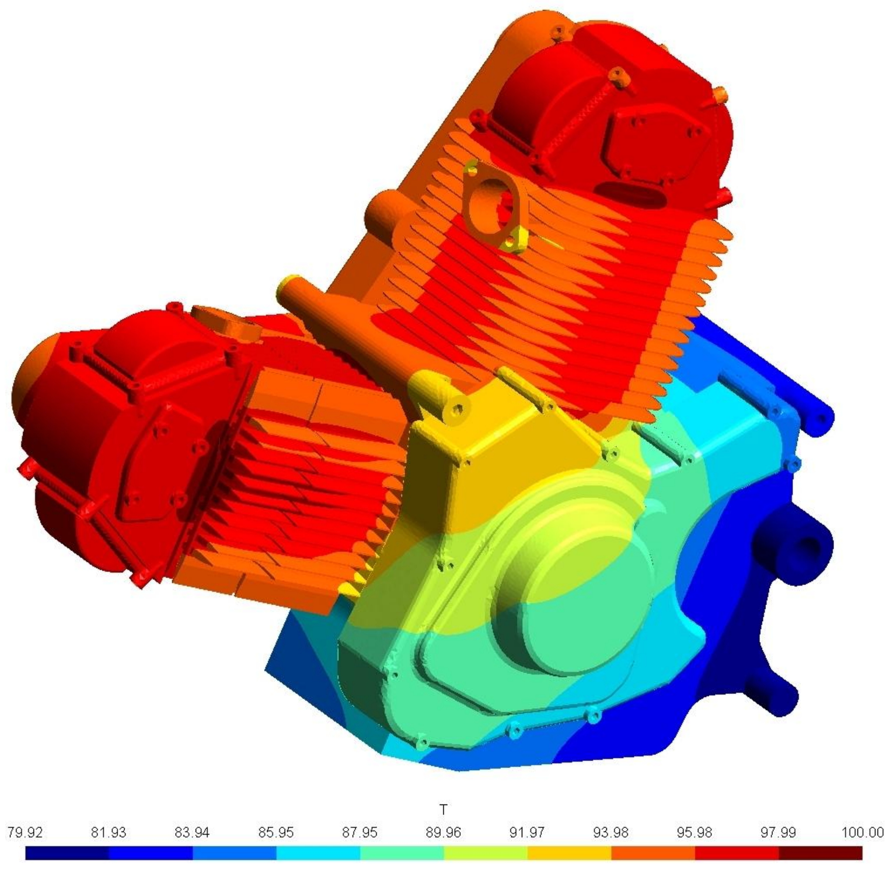


Figure 9: The temperature field computed by Ansys Workbench (top) and by our solver (bottom). Also in this case the two solvers lead to the same results, as it can be seen looking the plots: pay attention that different colormaps have been adopted.

6. Conclusions

In this work it has been shown how it is possible to use Scilab to write a parallel and portable application with a reasonable programming effort, without involving *hard* message passing protocols. The three dimensional heat transfer equation has been solved through a finite element code which take advantage of the parallel nature of the adopted algorithm: this can be seen as a sort of "elementary brick" to develop more complicated problems. The code could be rewritten with a compiled language to improve the run-time performance: also the message passing technique could be reorganized to allow a faster communication between the concurrent processes, also involving different machines connected through a net.
Stefano Bridi is gratefully acknowledged for his precious help.

7. References

[1] *http://www.scilab.org/* to have more information on Scilab.
[2] The Gmsh can be freely downloaded from: *http://www.geuz.org/gmsh/*
[3] *http://glaros.dtc.umn.edu/gkhome/views/metis* to have more details on the METIS library.
[4] O. C. Zienkiewicz, R. L. Taylor, (2000), *The Finite Element Method, volume 1: the basis*. Butterworth Heimemann.
[5] Y. Saad, (2003), *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM.