Blucher Mechanical Engineering Proceedings
May 2014, vol. 1 , num. 1
www.proceedings.blucher.com.br/evento/10wccm

10th World Congress on Computational Mechanics
8-13 July 2012 • São Paulo • Brazil

CONJUGATE GRADIENT METHOD FOR SOLVING LARGE SPARSE LINEAR SYSTEMS ON MULTI-CORE PROCESSORS

J. B. Aparecido[1] , N. Z. de Souza[1], J. B. Campos-Silva[1]

[1]Department of Mechanical Engineering, College of Engineering of the University of São Paulo State, at Ilha Solteira Campus (jbaparecido@dem.feis.unesp.br)

Abstract. *In the mathematical modelling of the fluid flow and heat transfer processes it is frequent to find systems of second order linear or non linear partial differential equations. When solving such systems of partial differential equations through the use of numerical methods such as finite elements or finite differences it is necessary to do the discretization process that transforms the original systems of equations, defined over a continuum domain, into a linear or non linear algebraic system, defined over a discrete domain. Due to the characteristics of discretization methods for the partial differential equations domain as well for the equations themselves, generally the algebraic system that appears has the coefficient matrix with a very high sparsity. In this work we present the implementation in parallel processing of routines capable to solve large linear sparse systems with positive definite coefficient matrix, exploiting and preserving the initial sparsity. It is analyzed the use of the conjugate gradient method in the solution of large sparse linear systems running on multi-core processors.*

Keywords: *Iterative solution, Sparse systems, Conjugate gradient, Parallel processing, Multi-core*

1. INTRODUCTION

In the solution of fluid flows and heat transfer problems by numerical methods such as finite element method (FEM) and/or finite difference method (FDM), systems of partial differential equations are transformed to large and highly sparse systems of algebraic equations. More than ninety nine percent of the elements in the matrices are nulls, so some structured data is almost mandatory to prevent storage of zeros and to reduce cost of numerical solutions of the linear systems resulting.

There are several iterative methods that can be used to solve linear systems, for example, Gauss-Seidel, Jacobi, SOR, SSOR and conjugate gradient methods. In a previous work Campos-Silva & Aparecido (2003) presented results of data structure in the context of the Gauss-Seidel and SOR methods. In another previous work Aparecido et al. (2011) presented data structure and algorithms to solve large sparse linear systems using conjugate gradient and preconditioned conjugate gradient methods. In both works were used serial processing. In the present work, it is analyzed the use of the conjugate gradient method (CG) to solve large and sparse linear systems but running in parallel under the paradigm of shared memory. Particularly, to achieve that we used OpenMP – Open Multi-Processing that is an application programming interface that supports shared memory multiprocessing (Chapman et al., 2008).

A set of linear systems similar to those that originate in three dimensional applications of FEM and/or FDM are solved by a CG algorithm running under OpenMP and the results of performance, speedup and efficiency of the method are presented and discussed.

2. SOLUTION OF LINEAR SYSTEMS AND METHODS FOR MINIMIZATION

This paper is an extension to shared memory parallel processing from our previous paper on this subject (Aparecido et al., 2011). Here we shortened a little bit some mathematical details that can be found there.

Consider the linear system

$$\mathbf{A}\mathbf{x}^* = \mathbf{b} \tag{1}$$

where $\mathbf{A} \in \Re^{n \times n}$ is the coefficient matrix and $\mathbf{b} \in \Re^{n}$ the independent vector. Both supposedly known. $\mathbf{x}^*$ is the vector of unknowns or solution vector to be determined.

The above equation can be restructured as

$$\mathbf{r(x)} \equiv \mathbf{b} - \mathbf{A}\mathbf{x} \tag{2}$$

in which the solution vector $\mathbf{x}^*$ was replaced by a generic vector $\mathbf{x}$, so there will be a residual vector $\mathbf{r(x)}$. Of course, when $\mathbf{x} = \mathbf{x}^*$ the residue will be zero, $\mathbf{r(x)} = \mathbf{0}$. Applying the rules of linearity, it is clear that the residue $\mathbf{r(x)}$ is linear with $\mathbf{x}$, and thus infinitely differentiable, so $\mathbf{r}(\mathbf{x}) \in C^{\propto}(\mathcal{R}^n)$.

Additionally, we can define a quadratic functional as follows

$$\mathrm{F}(\mathbf{x}) \equiv \tfrac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x} - \mathbf{x}^{\mathrm{T}}\mathbf{b}\,, \tag{3}$$

whose gradient and Hessian are

$$\mathbf{g(x)} \equiv \frac{\partial \mathrm{F}(\mathbf{x})}{\partial \mathbf{x}} = \mathbf{A}\mathbf{x} - \mathbf{b} = -\mathbf{r(x)}\,, \tag{4}$$

$$\mathbf{G}(\mathbf{x}) \equiv \frac{\partial \mathbf{g(x)}}{\partial \mathbf{x}} = \mathbf{A}\,. \tag{5}$$

Having obtained the gradient and Hessian of the function F(**x**), one can use some method of minimization to obtain the position $\mathbf{x}^*$ and the value of F($\mathbf{x}^*$) at the minimum point. Note that at the minimum point $\mathbf{g(x) = 0}$, $\mathbf{r(x) = 0}$ and $\mathbf{Ax^* - b = 0}$. Therefore, calculation of the minimum point of the functional **F(x)**, Eq. (3), corresponds to the solution of the linear system, Eq. (1).

The Hessian matrix (**G**) in the case is the same matrix of coefficients, **A**, and it must be positive definite, to attend the second necessary condition for the occurrence of a strong minimum in a given position, and should also be symmetrical, since

$$\mathrm{G}_{\mathrm{ij}}(\mathbf{x}) = \frac{\partial^2 \mathrm{F}(\mathbf{x})}{\partial \mathrm{x}_{\mathrm{i}} \partial \mathrm{x}_{\mathrm{j}}} = \frac{\partial^2 \mathrm{F}(\mathbf{x})}{\partial \mathrm{x}_{\mathrm{j}} \partial \mathrm{x}_{\mathrm{i}}} = \mathrm{G}_{\mathrm{ji}}(\mathbf{x})\,. \tag{6}$$

3. METHOD OF CONJUGATE GRADIENTS (CG)

A classical method of minimization that is most used in the solution of large sparse linear systems is the Conjugate Gradient Method. Different definitions of the functional F(**x**),

Eq. (3), will lead to different variants of the method. (Barret et al., 1994). In Aparecido et al. (2011) we give some details about those methods and its mathematical foundations. There are vast literature about conjugate gradient methods and solution of large sparse linear systems by using conjugate gradient methods, we can cite Golub and van Loan(1996), Golub and Meurant(1983), and Faber and Manteufel(1984).

Doing so we obtain an algorithm, computationally efficient for the serial processing implementation of the Conjugate Gradient Method.

Algorithm 1 – Method of Conjugate Gradients (CG)
$\mathbf{x_o}$ = initial vector
ε = stop parameter (positive and sufficiently small)
$\mathbf{r_o = b - Ax_o}$
$\rho_o = \mathbf{r}_o^T \mathbf{r}_o$
k = 0
while $\|\mathbf{r_k}\| \geq \varepsilon\|$
    k = k + 1
    if k = 1
        $\mathbf{p_1 = r_o}$
    else
        $\beta_k = \frac{\rho_{k-1}}{\rho_{k-2}}$
        $\mathbf{p}_k = \mathbf{r}_{k-1} + \beta_k \mathbf{p}_{k-1}$
    end if
    $\mathbf{w}_k = \mathbf{A}\mathbf{p}_k$
    $\alpha_k = \frac{\rho_{k-1}}{\mathbf{p}_k^T \mathbf{w}_k}$
    $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_k \mathbf{p_k}$
    $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{w}_k$
    $\rho_k = \mathbf{r}_k^T \mathbf{r}_k$
end while

When Hestenes & Stiefel(1952) created the Conjugate Gradient Method, they presented formulation very similar to that shown above to the solution of linear systems. This methodology is suitable for solving large sparse linear systems, since do not performs computations “inside” the matrix **A** and thus avoids the phenomenon of fill-in common in direct methods. In the above algorithm the coefficient matrix **A** is used in only one matrix-vector product. Initially this method was designed as a direct method since, in exact arithmetic, the algorithm converges to the exact solution. However, in computer arithmetic, with round-off error, this finite termination in *n* steps is not guaranteed. On the other hand, for large linear systems a set of *n* iterations represent a high computational cost. Thus, for large linear systems, the conjugate gradient method is used with termination based on maximum number of iterations, usually much less than *n*, and in the value of the norm of the residue. The idea of considering the conjugate gradient method as iterative method was developed by Reid (1971). The use of iterative conjugate gradient method is useful; however the rate of convergence is critical to its success (van der Sluis and van der Vorst, 1992).

4. METHOD OF CONJUGATE GRADIENTS (CG) FOR PARALLEL PROCESSING WITH SHARED MEMORY

In this work we parallelize the Algorithm 1 that becomes Algorithm 2 aimed to run using OpenMP. In this case, till now, we parallelized just the most computer intensive part of the algorithm that is de product matrix-vector $\mathbf{Ap_k}$. In the near future we intend to parallelize also the scalar products, such $\mathbf{p_k^T w_k}$, as well linear combinations of vectors, such as $\mathbf{p}_k = \mathbf{r}_{k-1} + \beta_k \mathbf{p}_{k-1}$.

Algorithm 2 – Method of Conjugate Gradients (CG) – Parallel OpenMP
$\mathbf{x_o}$ = initial vector
ε = stop parameter (positive and sufficiently small)
$\mathbf{r_o} = \mathbf{b} - \mathbf{A x_o}$
$\rho_o = \mathbf{r}_o^T \mathbf{r}_o$
k = 0
while $\|\mathbf{r_k}\| \geq \varepsilon\|$
    k = k + 1
    if k = 1
        $\mathbf{p_1} = \mathbf{r_o}$
    else
        $\beta_k = \frac{\rho_{k-1}}{\rho_{k-2}}$
        $\mathbf{p}_k = \mathbf{r}_{k-1} + \beta_k \mathbf{p}_{k-1}$
    end if
    !Here starts parallel section
    !$OMP PARALLEL DO SHARED(*list-of-shared*) PRIVATE(*list-of-private*) SCHEDULE(STATIC) &
    !$OMP NUM_THREADS(intNumThreads) DEFAULT(NONE)
    $\mathbf{w}_k = \mathbf{A p}_k$
    !$OMP END PARALLEL DO
    !Here finishes parallel section
    $\alpha_k = \frac{\rho_{k-1}}{\mathbf{p_k^T w_k}}$
    $\mathbf{x_k} = \mathbf{x_{k-1}} + \alpha_k \mathbf{p_k}$
    $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{w}_k$
    $\rho_k = \mathbf{r}_k^T \mathbf{r}_k$
end while

For the parallelization of the Algorithm 1 we introduced the following logical line of pseudo code to start parallelization:

```
!$OMP PARALLEL DO SHARED(list-of-shared) PRIVATE(list-of-private) SCHEDULE(STATIC)  &
!$OMP NUM_THREADS(intNumThreads) DEFAULT(NONE)
```

Note that OpenMP is no verbose allowing with few lines of code to produce big computational effects. OpenMP have some constructs that are used do parallelize code (Chapman et al., 2008). OpenMP has a webpage http://openmp.org were information can be obtained.

In the pseudo code above the main construct are:

- PARRALLEL – starts the parallel section creating some threads;
- DO – starts the parallelization of the outermost loop inside the parallel construct;
- SHARED(*list-of-shared*) – declares which computational entities (the *list-of-shared*)

are shared among all threads;

- PRIVATE(*list-of-private*) – declares which computational entities (the *list-of-private*) are not shared among all threads having its own memory allocation for each thread;
- SCHEDULE(STATIC) – indicates how the loop workload will be distributed among all threads. The clause STATIC means that the workload will be distributed equally;
- NUM_THREADS(intNumThreads) – declares how many working threads are intended to be used. The aimed quantity of threads is equal to intNumThreads and must be less or equal to the maximum quantity of available threads for a given processor.
- DEFAULT(NONE) – Means that no one default are allowed in that construct.

To end the loop work and the parallel section we introduced the following line o pseudo code:

```
!$OMP END PARALLEL DO
```

Naturally, the code that goes inside the construct PARALLEL must be in according with the ideas of parallel processing with shared memory and OpenMP standards.

5. DEFINITION OF AN EXAMPLE CASE

Here we present the definition of linear systems to be used to generate numerical results, using routines developed in the project to which this work is associated.

Be the linear system **Ax = b**, where

$$\mathbf{A} \equiv [A_{i,j}], \mathbf{b} \equiv [b_i] \text{ where } A_{i,j} \equiv \begin{cases} -1, & if \quad i = j + int(n^{2/3}) \\ -1, & if \quad i = j + int(n^{1/3}) \\ -1, & if \quad i = j + 1 \\ +6 & \\ -1, & if \quad j = i + 1 \\ -1, & if \quad j = i + int(n^{1/3}) \\ -1, & if \quad j = i + int(n^{2/3}) \end{cases} \quad \text{and } b_i = 1/i; \quad i, j = 1,2,...,n. \quad (7)$$

This positive definite and symmetric matrix was created in an abstract way, but inspired by heptadiagonal matrices that appear in the discretization, using a variety of methods, of the steady or unsteady Poisson partial differential equation, when defined over three-dimensional domains. The methodology developed in this project is generic and can absorb various settings of matrices. The solution of similar type of linear systems, using some stationary methods and data structures like CRS - Compressed Row Storage was treated in Campos-Silva & Aparecido (2003). In Aparecido et al. (2011) we defined a new, till we know, data structure so called SCRS – Symmetric Compressed Row Storage aimed on saving some memory storage by exploiting the matrix symmetry. Also in this work we used SCRS.

6. RESULTS AND DISCUSION

Results presented in this paper were run in a desktop personal computer with a quad-core Intel i7 processor providing 8 threads. This type of computer uses three kinds of

caches: L_1, L_2 and L_3. So, the flow of data and instructions from RAM to cores, back and forth, is complex and if the algorithm does not exploit adequately the characteristics of processor then computations would be inefficient.

We solved the proposed sparse linear systems from 100 thousands unknowns until 10 millions of unknowns. We developed three main codes: the first one running serially; the second running "parallel OpenMP" but with just one thread; and a third one running in parallel, OpenMP, with from 2 to 8 threads.

The parallel code running with 2 to 8 threads were developed using Algorithm 2 shown previously in this paper.

The memory footprint for Algorithms 1 and 2 were taken to be approximately equal to (60*n*+12*nnz*) bytes. Where *n* is the order of the matrix **A** and *nnz* is the number of non zero elements in matrix **A**. It is important to remember that we used the SCRS data structure and so the *nnz* is about half of "*nnz*" that would be if using CRS data structure. Results for memory footprint are shown in Figure 1. One can see that memory footprint is linear with matrix **A** order. This characteristic can be used to define a sparse matrix: that its memory footprint be linear with its order. A full matrix memory footprint is quadratic with its order and even triangular matrices have memory footprint quadratic with its order.

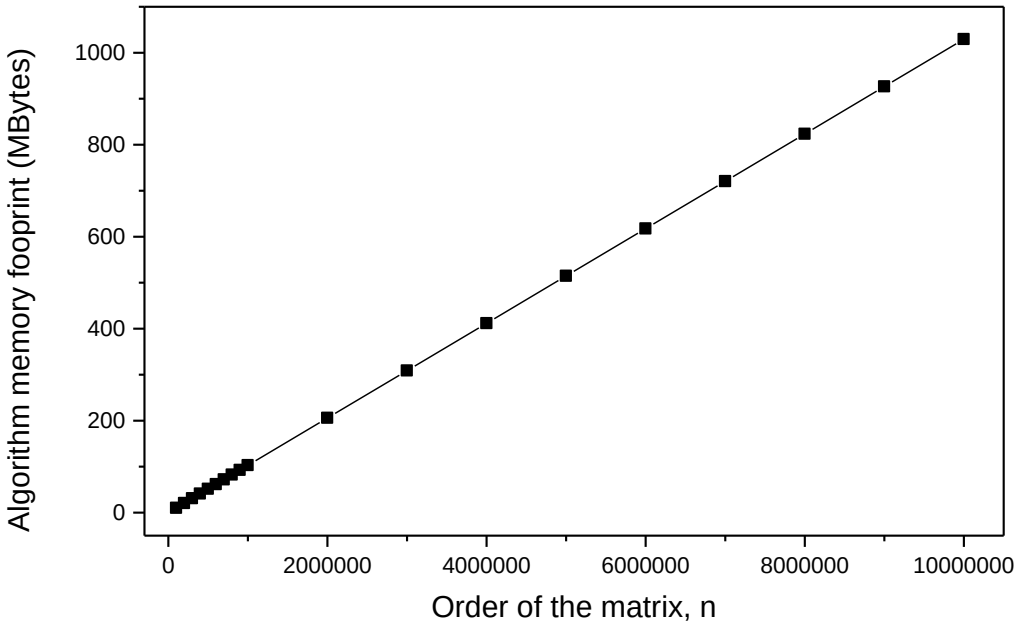

Figure 1 – Memory footprint for running the Conjugate Gradient Algorithms 1 and 2.

In Figure 2 we show the number of iterations to achieve convergence under the stopping criteria $\|\mathbf{r}\| \leq \varepsilon = 10^{-14}$. Generally, as expected when the order of the matrix **A** grows the number of iterations to achieve convergence also increases. But, sometimes when the order of the matrix grows the number of iterations to reach convergence decreases. We believe that this behavior is due to bigger or lesser ease to the data and instructions to flow from RAM to cores, back and forth, passing through caches. Depending on sizes of each one the work will be done better or not (Wiggers et al., 2007).

It´s notable that for 10 millions of unknowns the algorithm spent just about 1500 iterations to converge.

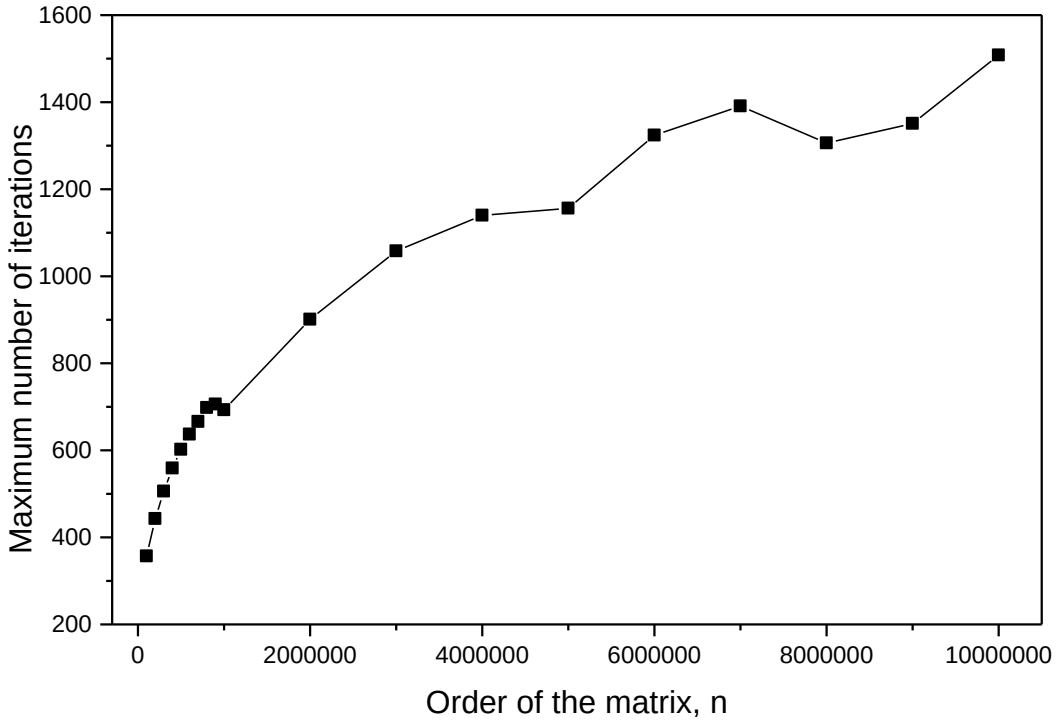

Figure 2 – Number of iterations to achieve converge under the criteria $\|\mathbf{r}\| \leq \varepsilon = 10^{-14}$

The results shown in Figures 1 and 2 for memory footprint and the number of iterations to reach convergence under the stopping criteria are the same for running serial or parallel, because in our parallel algorithm we just changed the way the product matrix-vector is done. More precisely, the footprint is nearly the same and the number of iterations to convergence is the same.

In Figure 3 we show the wall time (clock time) and the CPU time to run serially Algorithm 1. One can observe in Figure 3 that the results are practically equal. Actually, the wall time is slightly bigger than CPU time. This near equality is possible just because we avoided in Algorithms to do calls to the computer operating system that would increase the wall time. So, we recommend when measuring parameters in parallel processing to avoid to do calls to the system, such as printing, pausing, etc. As the size of the linear system grows the wall time and CPU also grows almost linearly.

Figure 4 shows the performance of the processor running Algorithm 1 serially. The numbers of flops per time were taken as $[(9n+4nnz)iter+4nnz+2n]$/wall-time, where *iter* is the number of iterations to reach convergence under the stopping criteria. For 100 thousands unknowns the performance obtained were about 1460MFlop/s, then when increasing the size of the linear system the performance decreases until about 1140 MFlop/s for 900 thousands unknowns. From 1 million to 10 millions of unknowns the performance stays oscillating around 1180 MFlop/s.

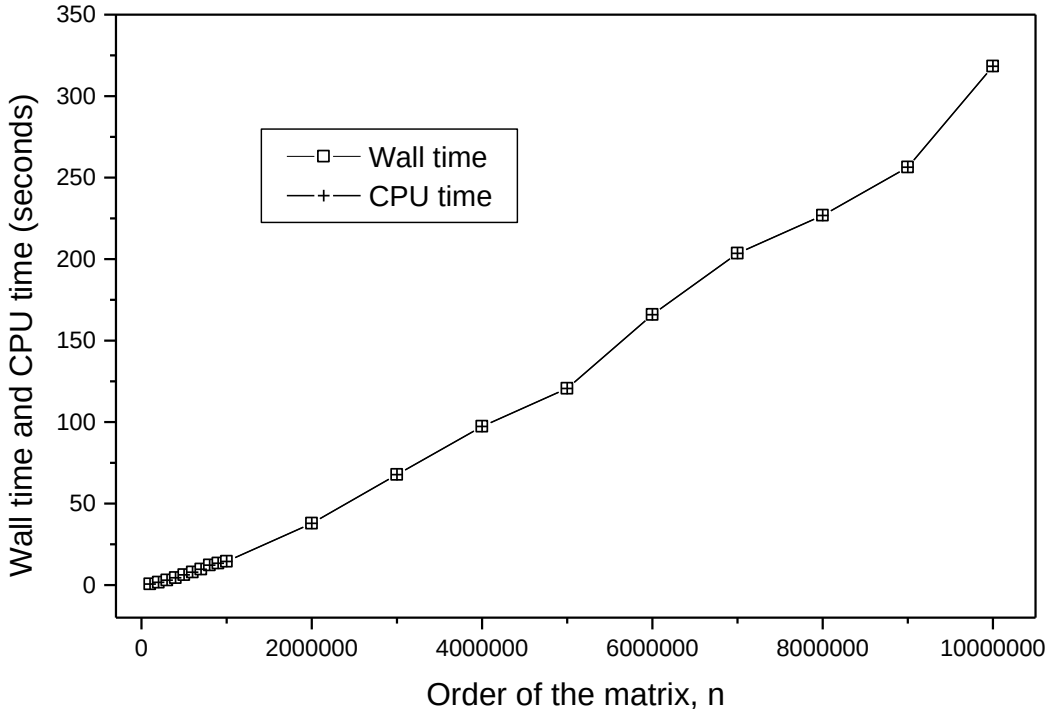

Figure 3 – Wall time and CPU time spent until the Algorithm 1 reached converge.

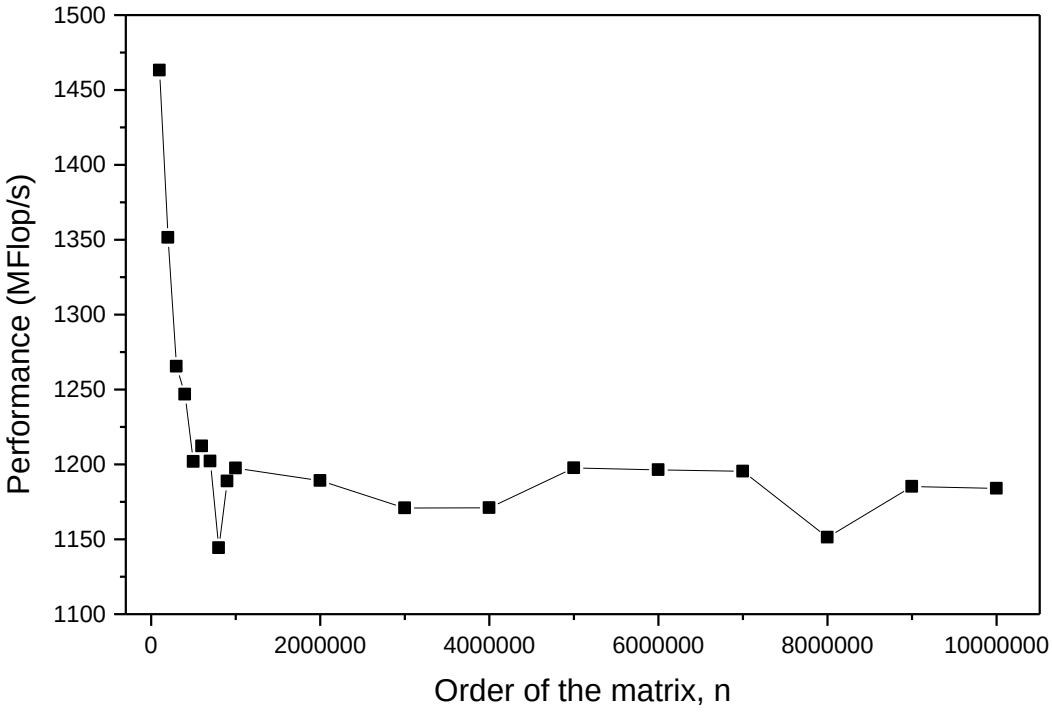

Figure 4 – Performance of the processor running serially Algorithm 1.

The overhead caused by OpenMP can be analyzed in Figures 5 and 6. The wall time for running Algorithm 2 in parallel OpenMP with just 1 thread is always bigger than the wall

time for running Algorithm 1 serially. This is because the useful work is the same in both case and there is an extra effort when running OpenMP constructs (Chapman et al., 2008). Note that for 10 millions of unknowns the time spent to reach convergence is about 330 seconds and for 100 thousands of unknowns the time spent is just a fraction of second.

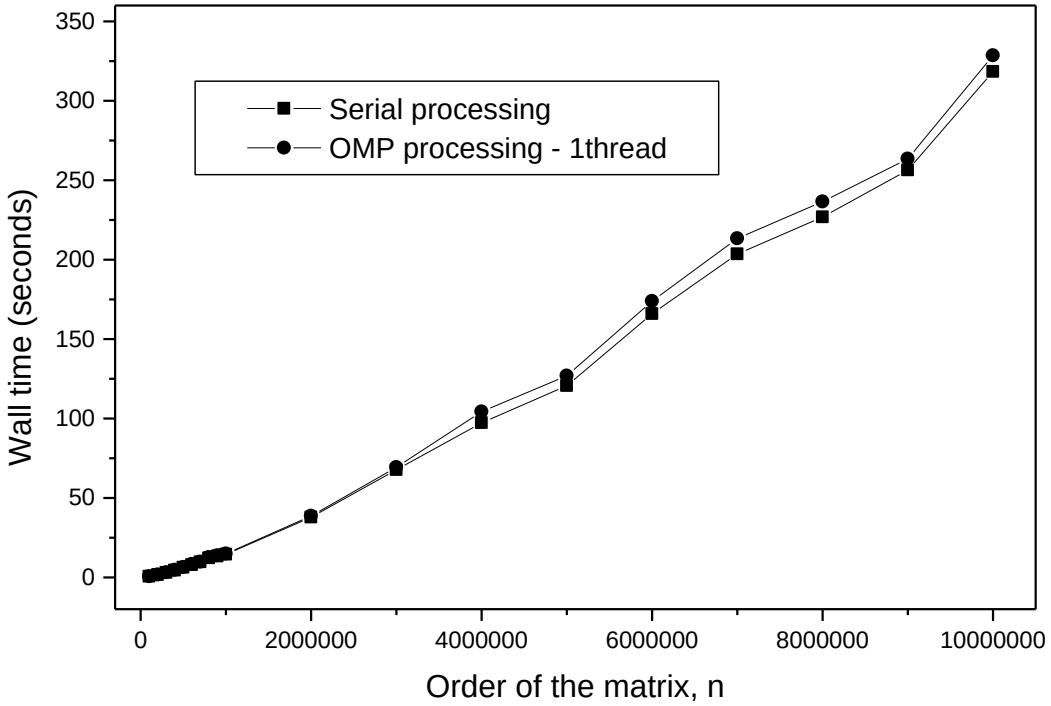

Figure 5 – Wall time for running Algorithm 1 serial and Algorithm 2 in "parallel" for 1 thread.

As the wall time spent to run a given case is different for running serial and running parallel with 1 thread, so the performance also will be different in both cases. In Figure 6 we show the overhead of OpenMP in terms of performance. One can see that for all cases ran the performance for parallel, 1 thread, is worst than when running serial. This is due to the fact that when running parallel, 1 thread, part of the computer processing capacity will be spent for running OpenMP constructs without producing any useful work. In average the overhead caused by OpenMP in terms of performance is lowering its value about 4.5% for the cases tested.

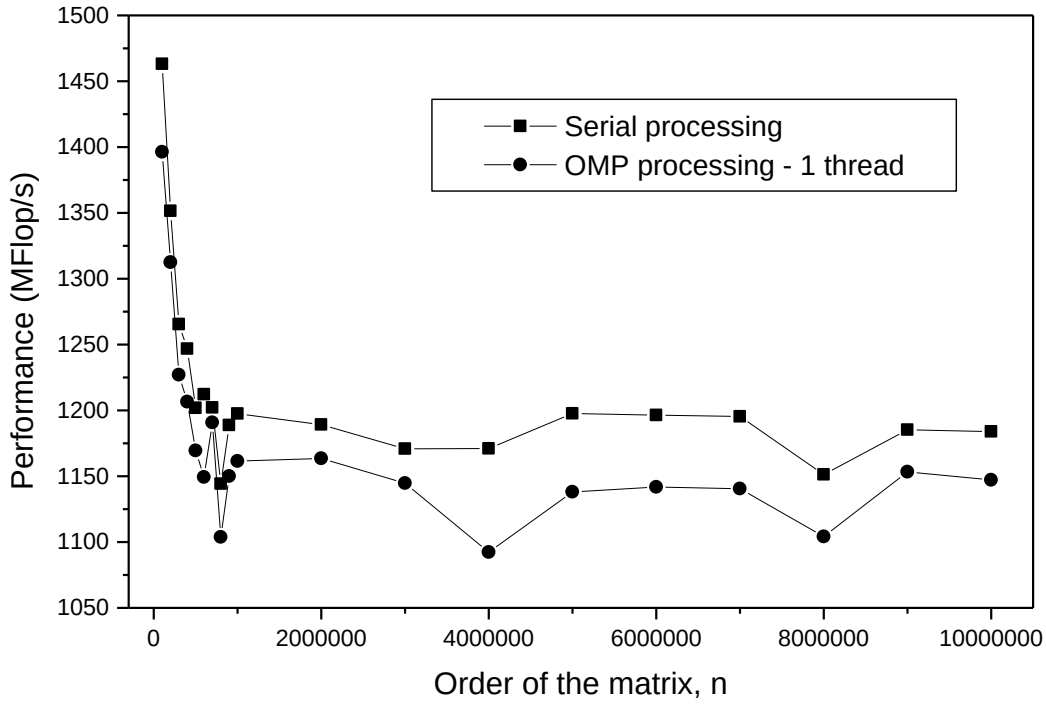

Figure 6 – Performance for Algorithm 1 running serial and Algorithm 2 running "parallel" with 1 thread.

From Figure 7 to 9 we present results for performance (MFlop/s), speedup and efficiency as function of the number of threads used that varied from 2 to 8 the maximum available in this kind of quad-core processor.

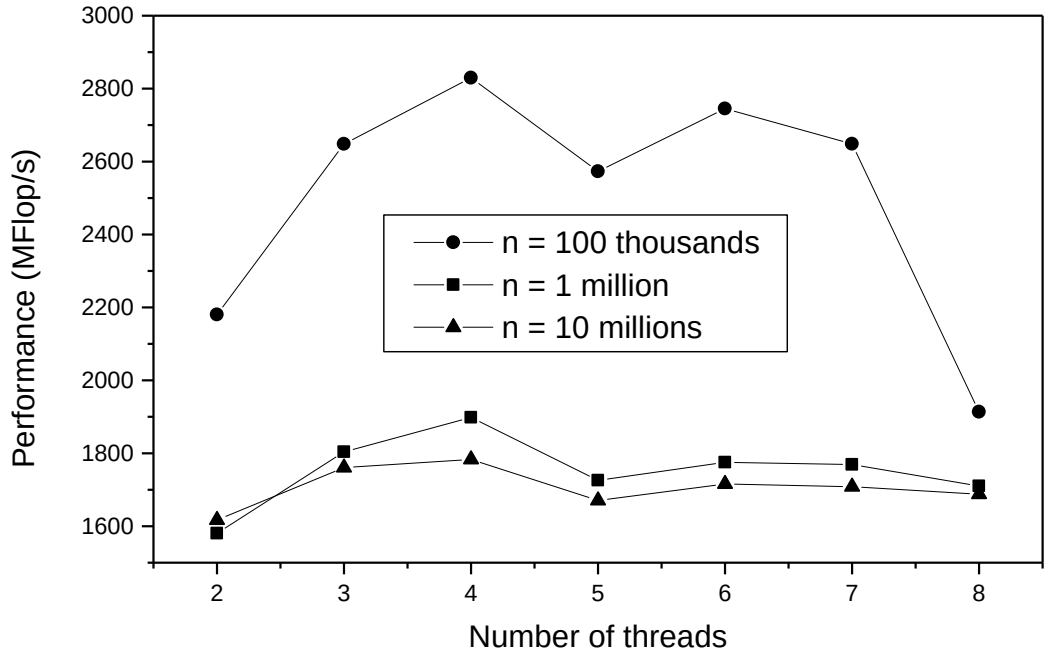

Figure 7 – Performance running in parallel for 2 to 8 threads, parameterized by the order of linear systems.

Results were parameterized using the number of unknowns in the linear system. Characteristic values taken were: 100 thousands, 1 million and 10 millions of unknowns.

In Figure 7 is shown results for performance as function of the number of threads used. For a given size of the problem being solved the maximum performance obtained is for 4 threads. Such quantity of threads is equal to the number of cores. Probably, that happens because each core runs better 1 thread than 2 threads simultaneously, under the same process. Similarly to what happened in the case of serial processing for the parallel processing also the best performance was achieved for problems with 100 thousands unknowns. For 1 million and 10 millions of unknowns performance was quite similar. Also note that for serial processing the performance peak value was about 1460 MFlop/s, Figure 4, and for parallel processing the performance peak was about 2800 MFlop/s using 4 threads, Figure 7.

Results for speedup can be seen in Figure 8. Qualitatively, results for performance are very similar to those ones for speedup. In this later case also the best speedup is achieved using 4 threads. Speedup peak is about 1.9 for the cases with 100 thousands unknowns. For the cases of 1 million and 10 millions of unknowns the speedup remains in the range from 1.3 to 1.6.

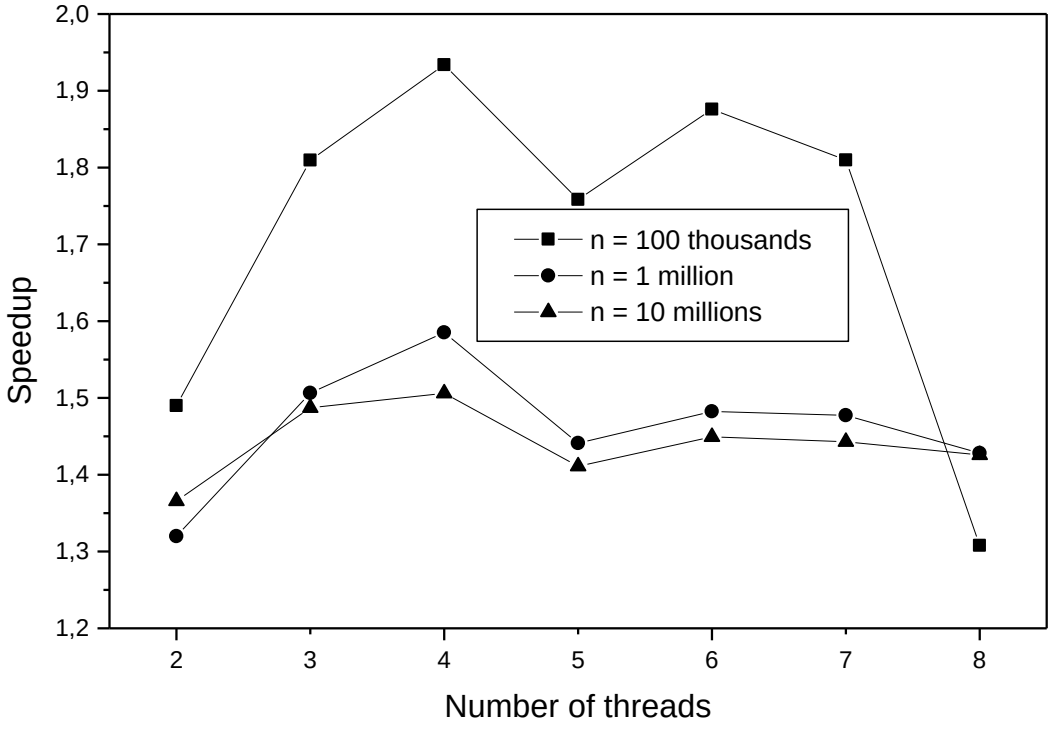

Figure 8 – Speedup when running in parallel for 2 to 8 threads, parameterized by the order of linear systems

Can be seen in Figure 9 that the best efficiency, for all cases tested, happens when using 2 threads and it ranges from 0.65 to 0.75. After that, when the quantity of threads used grows, the efficiency drops until reaching its minimum value, about 0.18, for 8 threads. The efficiency running the case for 100 thousands unknowns was about 10% bigger than those ones for 1

million and 10 millions of unknowns. Just when running with 8 threads that efficiencies are almost equal for all sizes of linear systems.

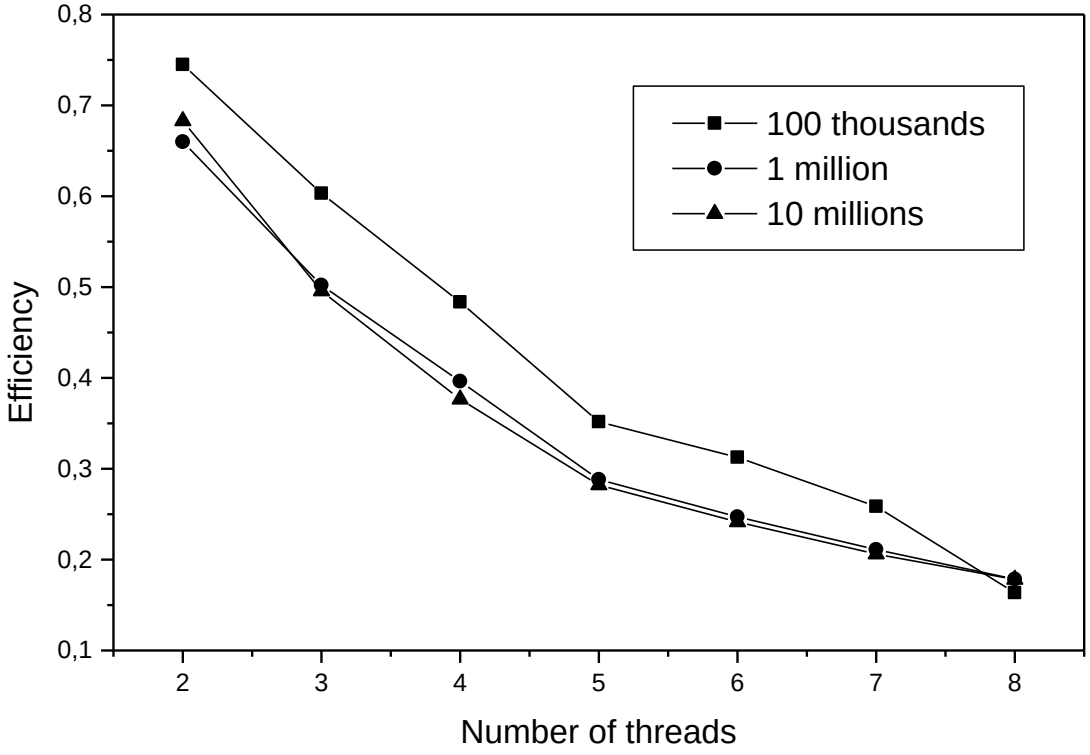

Figure 9 - Efficiency running in parallel for 2 to 8 threads, parameterized by the order of linear systems

When using 5 to 8 threads neither the performance nor the efficiency are very good, then we decided to show a bit more details of results for 2 to 4 threads. Also was shown in Figures 7 to 9 that the results for 1 and 10 millions of unknowns are very close. Thus we do some detailing in the results for the number of unknowns from 200 to 800 thousands. Results for performance, speedup and efficiency are shown in Figures 10 to 12, respectively.

In Figure 10 we can see that the best performance is reached when using 4 threads and the worst performance is obtained for 2 threads. For a given number of threads the performance peak happens for 200 thousands unknowns, after that as the number of unknowns grows performance goes down reaching its minimum for 800 thousands unknowns.

Similarly, to Figure 10 the best results obtained for speedup, Figure 11, were obtained using 4 threads, and differently from performance the speedup shows a maximum value for 200 thousands unknowns and afterwards starts to decrease but reaches a minimum for 600 thousands unknowns and then starts to grow as the number of unknowns increase. Generally, this non linear behavior as function of the problem size, or footprint, is caused by data and instructions flow, back and forth, from RAM to cores, passing through caches.

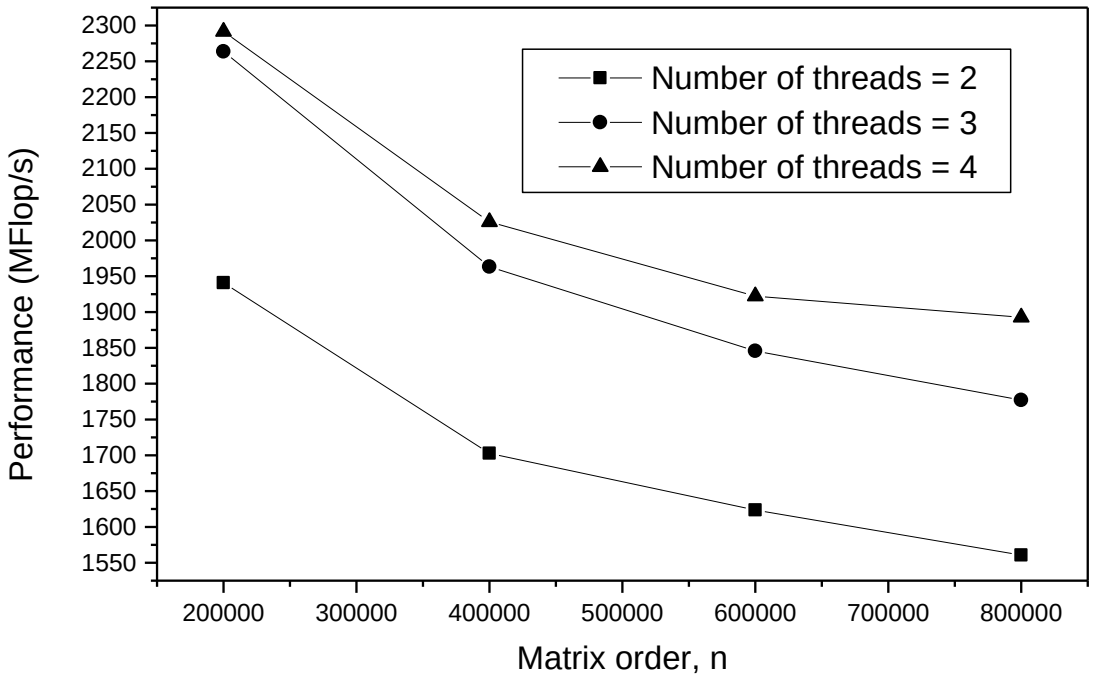

Figure 10 – Performance as function of matrix order and parameterized by the number of threads.

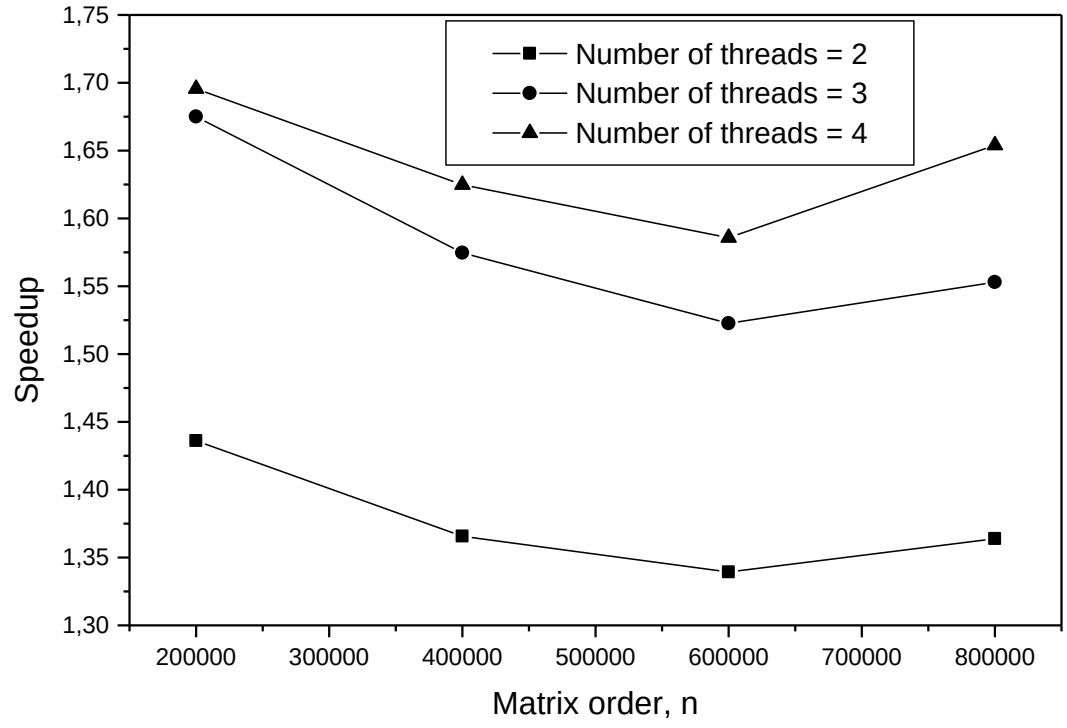

Figure 11 – Speedup as function of matrix order and parameterized by the number of threads.

The best efficiency obtained in the parallel multithreading is achieved using 2 threads as seen in Figure 12. Best performance and speedup happen for 4 threads but efficiency happens for 2 threads. This probably occurs because just part of the algorithm was parallelized for the

results presented in this work. So, remains some room for more parallelization of Algorithm 2, mainly in the scalar products and in linear combinations among vectors. We will tackle that in future works.

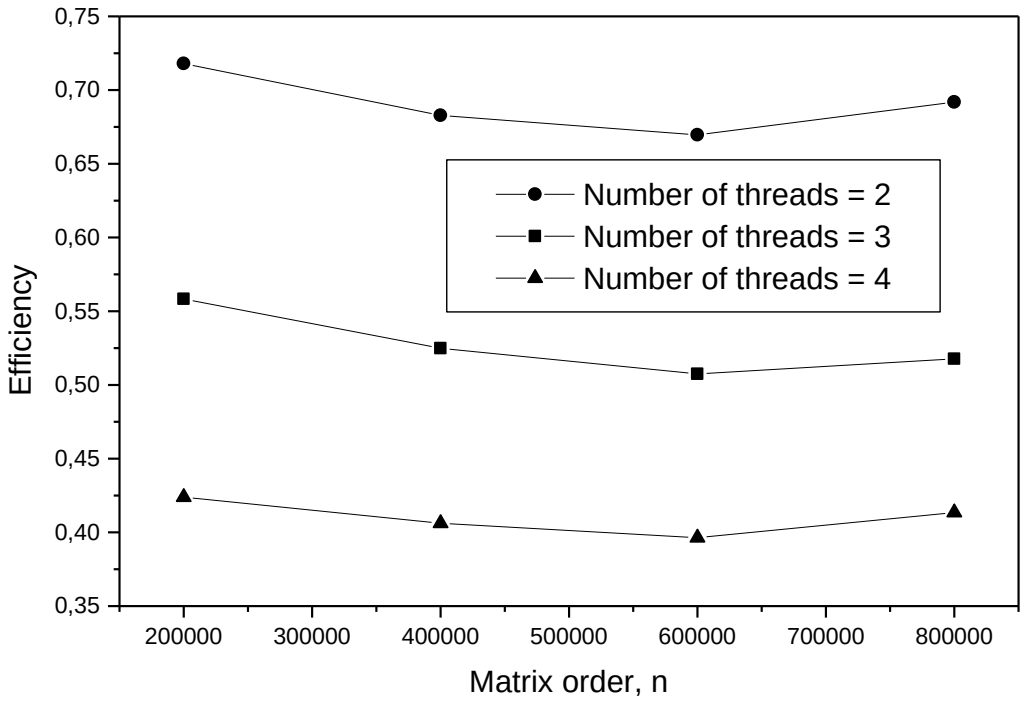

Figure 12 – Efficiency as function of matrix order and parameterized by the number of threads.

In Figure 13 are shown the results obtained using Algorithms 1 and 2. Note that the norm of the residual goes down until reaching the stopping criteria that was $\|\mathbf{r}\| \leq \varepsilon = 10^{-14}$. Between results obtained by serial and parallel processing the results for the unknowns have just tiny numerical differences. The numbers of iterations to reach converge were the same in both cases, as expected. For 10 millions of unknowns were necessary about 1500 iterations to reach convergence. We speculate that in the same computer one can reach until about 100 millions of unknowns, running in about one hour.

7. CONCLUSION

From the results and reasonings displayed before we can conclude that:

- OpenMP is a non verbose application programming interface that allows with few lines of code to produce code parallelization. Naturally, the code that goes inside an OpenMP construct must be in according with parallel processing theory and with OpenMP standards;
- Parallelized Conjugate Gradient Algorithm presented here worked well and was able to solve sparse linear systems with up to 10 millions of unknowns in about 5 minutes;
- The best performance and speedup were obtained using 4 threads;
- Probably, Algorithm 2 could be improved through parallelization of scalar products

and linear combinations of vectors;

- Algorithm 2 and/or the i7 processor does not scales so well and thus efficiency goes down quickly as the number of threads increases. Here is necessary further research to improve efficiency.

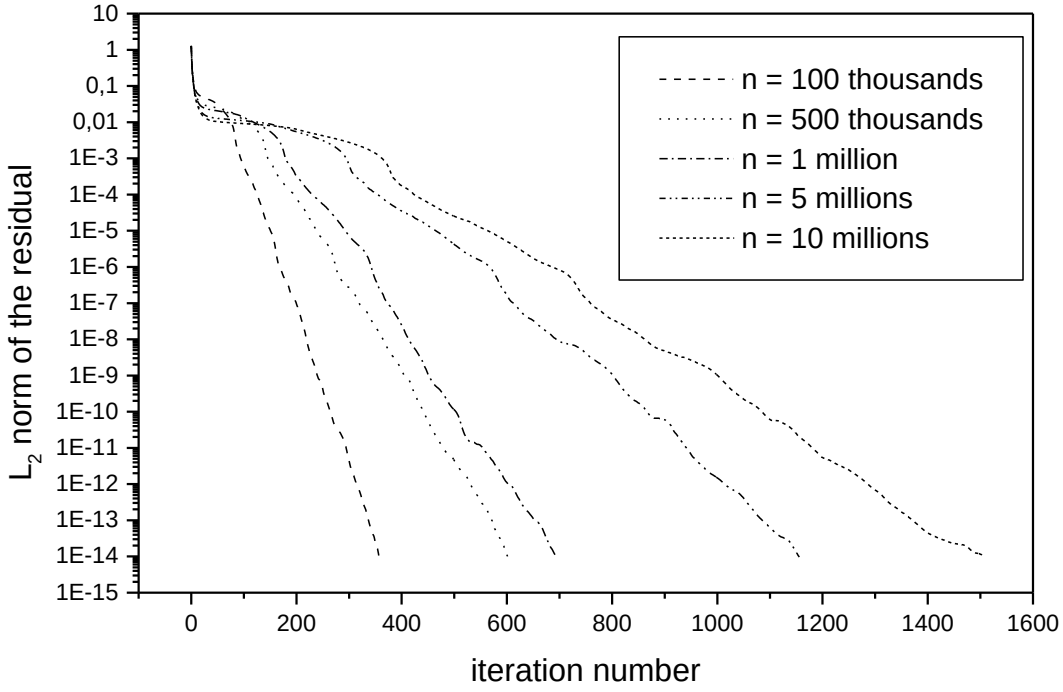

Figure 13 – L_2 norm of the residual as function of the number of iterations.

Acknowledgements

The authors would like to thank **Fundação de Amparo a Pesquisa do Estado de São Paulo - FAPESP** and the **Fundação para o Desenvolvimento da UNESP – FUNDUNESP**, both for supporting our projects in the last two decades.

8. REFERENCES

[1] J. B. Aparecido, N. Z. de Souza and J. B. Campos-Silva, "Data Structure and the Pre-Conditioned Conjugate Gradient Method for Solving Large Sparse Linear Systems with Symmetric Positive Definite Matrix". *Proceedings of CILAMCE-2011*, Ouro Preto, 2011

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eikhout, R. Pozo, C. Romine and H. Van Der Vorst, "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", *SIAM*, Philadelphia, PA, 1994.

[3] J. B. Campos-Silva and J. B. Aparecido, "Data structure and stationary iterative solution methods for large sparse linear systems". *Proceedings of the 17th International Congress*

of Mechanical Engineering – COBEM 2003, Paper 0036, November 10-14, São Paulo, SP, 2003.

[4] B. Chapman, G. Jost and R. van der Pas, "Using OpenMP", The MIT Press, Cambridge, 2008.

[5] V. Faber and T. Manteuffel, "Necessary and Sufficient Conditions for the Existence of a Conjugate Gradient Method", *SIAM J. Numer. Anal.*, 21, pp. 315-339, 1984.

[6] G. H. Golub and G. Meurant, "Résolution Numérique des Grandes Systèmes Linéaires", *Collection de la Direction des Etudes et Recherches de l´Electricité de France*, vol. 49, Eyolles, Paris, 1983.

[7] G. H. Golub and van Loan, C. F., "Matrix Computations", Third Edition, The John Hopkins University Press, 1996.

[8] M. R. Hestenes and E. Stiefel, "Methods of conjugates gradients for solving linear systems", *J. Res. Nat. Bur. Standards*, 49, pp. 409-436, 1952.

[9] J. Reid, "On the method of conjugate gradients for the solution of large sparse systems of linear equations", in *Large Sparse Sets of Linear Equations*, Ed. J. Reid, Academic Press, London, pp. 231-254, 1971.

[10] A. Van Der Sluis and H. Van Der Vorst, "The Rate of Convergence fo Conjugate Gradients", *Numer. Math.*, 48, pp. 543-560, 1992.

[11] W.A. Wiggers, V. Bakker, A.B.J. Kokkeler and G.J.M. Smit, "Implementing the conjugate gradient algorithm on multi-core systems", *In: International Symposium on System-on-Chip*, SoC, Tampere, Finland, 2007.