PARALLEL SOLUTION ON A PC CLUSTER AND BEHAVIOR ANALYSIS OF BI AND THREE-DIMENSIONAL STRUCTURAL PROBLEMS UTILIZING THE FINITE ELEMENT METHOD

Flávia Romano Villa Verde
Departament of Mechanical Engineering, University of Brasilia Campus Universitário Darcy Ribeiro CEP 70910-900 Brasília - DF
flaviarvv@gmail.com

Gerson Henrique Pfitscher
Departament of Computer Science, University of Brasilia Campus Universitário Darcy Ribeiro CEP 70910-900 Brasília - DF
gerson@unb.br

Dianne Magalhães Viana
Departament of Mechanical Engineering, University of Brasilia Campus Universitário Darcy Ribeiro CEP 70910-900 Brasília - DF
diannemv@unb.br

Abstract. *Large and complex engineering problems often need too much computing time and storage to run on single processor computers. Even if they can be solved, more powerful computing capabilities are required to obtain more accurate and reliable results within reasonable time. Parallel processing, the method of having many small tasks to solve one large problem, successfully fulfill such requirements for high-performance computing. We evaluate and compare the performance of two communication libraries, mpich-1.2.5 and pvm3.4.4, on a Linux PCs cluster connected by a gigabit Ethernet network, solving two structural problems applied to linear elasticity, utilizing the Finite Element Method for modeling and the Conjugate Gradient Method for the solution of the system equations. This appears to be a viable low cost option, since we utilize already consolidated technologies, with open architecture and public domain softwares. In this work, the mesh generation was done using the program GiD as a preprocessor. For mesh partition, the software package METIS was employed, and the visualization of the partitions was aided by the PMVis program. This article summarizes some numerical simulation results, and the behavior analysis for the two codes developed changing both the mesh granularity and the number of processors.*

Keywords*: Cluster computing, Beowulf clusters, communication libraries, domain decomposition, finite element method, conjugate gradient method*

1. Introduction

Researchers always need higher processing speed and more memory in order to investigate increasingly complex problems. Distributed parallel computing can be very cost effective when commodity workstations and PCs are used as the computing platforms. A cluster is a set of interconnected machines, used as a single computational resource. Essentially, any group of systems networked together and dedicated to a single purpose can be called a cluster. The difference between a cluster and a group of networked workstations is that the machines in a cluster function as a unit. Individual programs run either on the whole cluster or on some subset of the cluster machines. A Beowulf Cluster is composed of commodity off the shelf (COTS) PC machines all running the Linux operating system. The individual computers that make up a cluster are called nodes. Because the Beowulf cluster has multiple processors (CPU's), usually 1 or 2 per node, it has to be programmed in parallel.

PC clusters now offer solution for affordable supercomputing. The role of traditional supercomputers (like CRAY) was once taken over by workstations. Now the huge PC market allowed unprecedented dynamics of PC hardware evolution. Good PC offers far the best price/performance ratio unavailable with any workstation. Today's networking technology made PC clusters feasible computing tool and allowed them to beat other technologies also in absolute numbers (Ong and Farrel, 2000). Due to the increase in network hardware speed and the availability of low cost high performance workstations, cluster computing has become increasingly popular. Many research institutes, universities, and industrial sites around the world have started to purchase/build low cost clusters, such as Linux Beowulf-class clusters, for their parallel processing needs at a fraction of the price of mainframes or supercomputers (Kim and Kim, 1997), (Ong and Farrel, 2000). In November 2004, more than 58% of the 500 most powerful computer systems installed in the world were clusters. In November 2003 they were more than 41% (TOP500, 2004).

Large and complex engineering problems often need too much computing time and storage to run on ordinary single processor computers. Even if they can be solved, more powerful computing capability is required to obtain more accurate and reliable results within reasonable time. Parallel processing, the method of having many small tasks solve one large problem, successfully fulfilled such requirements for high-performance computing. Various types of parallel hardware architectures have been developed and parallel algorithms adapted to these hardware architectures have been suggested (Kumar, Grama, Gupta and Karypis, 2003), (Dongarra *et al*., 2003), (Hughes and Hughes, 2004).

The development of new parallel applications and the parallelization of existing sequential or serial applications, to fully exploit the power of a distributed system, such as a cluster, is a complex task. To make effective use of the parallel nature of a Beowulf cluster, we need to run either: a parallel application or a serial application that must be run repeatedly on different data (a parametric study). While there are some parallel applications available that some researchers can use, most software doesn't come already parallelized, and we usually have to write our own.

Also, most of the software programs and libraries for use in the clustered environment are distributed in source form and must be built on each system. Some problems are inherently parallel and others are not. There are several methods that can be used to write programs that can make use of multiple processors. These range from tools that analyze a program to detect parallelism to parallel programming using message passing libraries.

When a program is run in parallel, the program or code units run asynchronously on the nodes. That is, each portion of the code runs independently of the other program units which are running simultaneously on the other nodes. The program units can communicate and share data either by shared memory or by message passing.

In shared memory, where all the program units running on the nodes of an SMP (Symmetric Multi Processing) machine access data from a single central memory and, at any moment, the data can be accessed and eventually changed by any processor node. Every interaction between processor nodes is performed through the shared memory. The message passing is typical of a program run on a set of computing systems, each of them owning private memory and linked by means of a communication network. Any interaction between processes is achieved through an explicit exchange of messages.

A message is a string of data or a null string sent or received by a task. The tasks pass data between their application buffers in the form of messages to share intermediary information and aggregate the final results. Both the request for data from another node and the fetched data are referred to as messages. The Message Passing Interface, MPI, is a standard API (Application Programming Interface) that can be used to create parallel applications. MPI is designed primarily to support the SPMD, single program multiple data, model. As MPI is a standard communication library, programs written with MPI are highly portable. The PVM (Parallel Virtual Machine) is a subroutine library callable from C and Fortran programs, plus system support processes, for distributed memory parallelism. PVM's goal is to allow the user to create such a parallel virtual machine from any heterogeneous collection of machines and networks (Gropp, Lusk and Skjellum, 1999), (Geist *et al*., 1994).

This article presents some numerical simulation results of a parallel version, proposed by Jimack (Jimack and Touheed, 2000) of the Conjugate Gradient Method (CGM), used for the solution of the equilibrium equations of an elasticity problem for bi- and tri-dimensional linear systems, adapted to the distributed parallel environment of PC clusters. The domain partition algorithm developed by Karypis and Kumar (Karypis and Kumar, 1998) as been employed with METIS for partition the finite element meshes, and the visualization of the partitions was aided by the PMVis program. We present comparative measures of executions time for two tested parallel codes developed with MPI and PVM message passing libraries, changing both the mesh granularity and the number of processors.

2. Finite element method

The solution of static problems in solid mechanics requires the equilibrium of forces and moments at all points within the structure, the compatibility of strains in order to preserve the material continuities, the establishment of constitutive equations relating stress and strain, and the imposition of boundary conditions for the particular problem.

The discretization process of a solid continuous structure using the Finite Element Method (FEM) in problems related to linear elasticity leads to a system of algebraic equations of the form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{1}$$

where **A,** the stiffness matrix, is a positive definite symmetric matrix, **x** is the nodal displacement vector, and **b** is the force vector independent terms.

There are several algorithms to solve the system represented by the Eq. (1). Iterative methods entail less memory than direct methods even if the number of floating point operations needed is not known previously. These methods are suitable for computers architectures that enable parallelism of different tasks on the solution procedure. The CGM is one of the mentioned methods which have obtained popularity in FEM solution for the reason that it is simple, efficient, and provides significant reductions in saving time and storing data (Tan and Bathe, 1991).

The linear analysis is not an example of finite element method application representing very high computing costs, but permits realize the proposed comparative analysis between the two message passing libraries using a more simple code, giving results that can be considered for more complex problems.

3. Parallel program

The parallel programs developed in this work were structured to execute on a cluster of PCs, following the message passing paradigm. For this, the libraries mpich-1.2.5 and pvm3.4.4 were used to perform communications between

tasks, with one task assigned to each processor. The workflow for the parallel code is shown in Fig. 1. The program GiD was used as a pre-processor unit, on a single processor, to create the finite element mesh data. These data structures are then kept on a shared site of the network, so each processor can fetch it and work with it. The code is organized in three principal parts: mesh partition or domain decomposition, matrix assembling and solution of equation system with the conjugate gradient method.

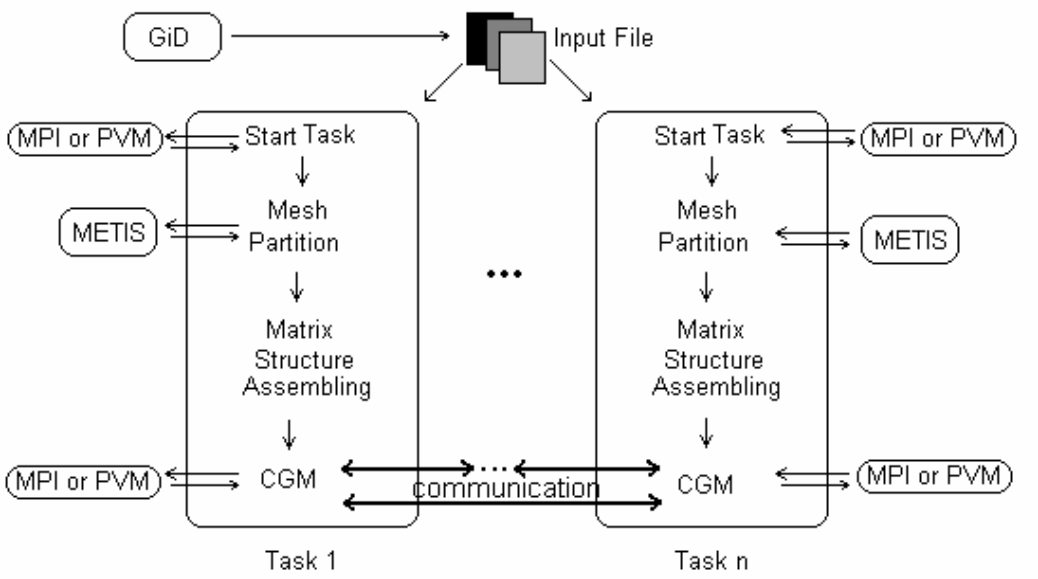


Figure 1. Parallel program workflow scheme: partition, assembling and CGM

3.1. Domain decomposition

In the development of parallel applications the partition stage intends to expose the opportunities for parallel execution. A good partition divides into small pieces both the computations associated with the problem and the data on which these computations operate. This divide the partition problem into two categories: the functional decomposition and the domain decomposition (Foster, 1995).

The former is the method of parallelization in which different but related parts of the program are run on different processor nodes to solve the problem. The size of the simultaneously executing parts of a parallel program is referred to as the granularity of the program. The granularity of a parallel system is dependent upon the number of nodes and their power.

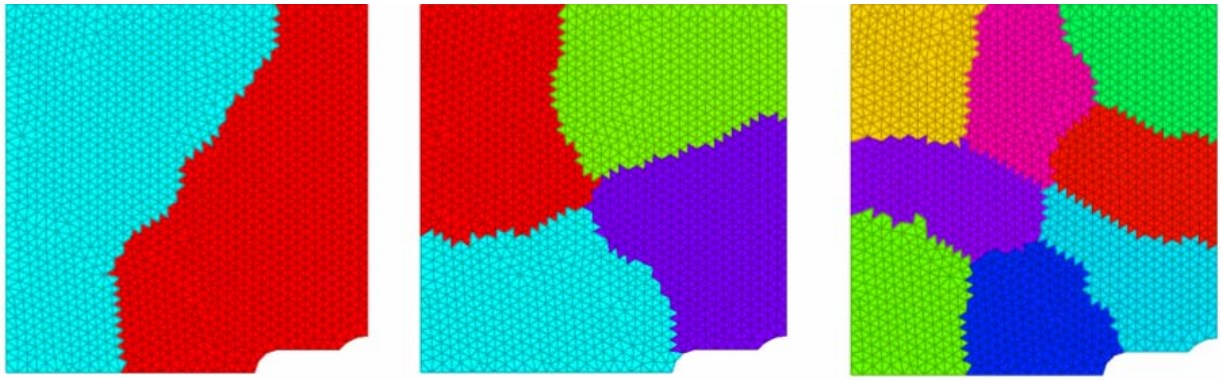
The second is a method of parallelizing a program which uses data (domain) decomposition. The same algorithm is run on more than one processor node and iterative tasks, like loop processing, are distributed in a controlled fashion among the different processor nodes. The results computed on the different processor nodes may be combined when all the nodes have completed. The functional parallelism, which is also called distributed paradigm, is less common in real applications than data decomposition, called parallel paradigm.

Programs which require the application of the same algorithm to many data points can be better parallelized with data decomposition. In our application, the mesh partition process splits the finite element mesh. This is done by the use of METIS program, version 4.0 (Karypis and Kumar, 1998). The domain decomposition is done in parallel, each task calling the function *METIS_PartMeshNodal*().

As the partitions present almost the same number of elements then each task as assigned approximately the same number of equations to compute. For the implemented programs (MPI and PVM) the number of partitions is equal to the number of tasks utilized, so each task is running on a single processor with approximately the same volume of data. In this part of the program there are no communication between tasks and at the end of the partition process each task has information about the size and which elements belongs to its own sub-domain.

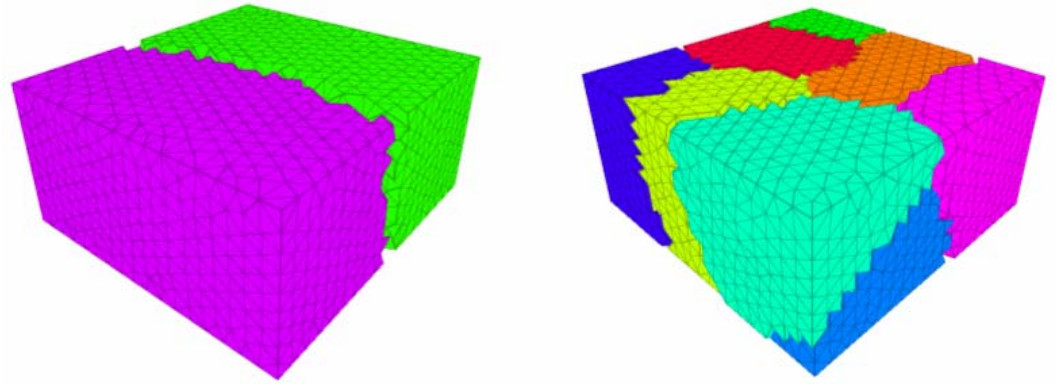
Figure 2 shows one bi-dimensional finite element mesh partitioned in (a) two, (b) four and (c) eight parts or sub-domains, and Fig. 3 shows a tri-dimensional mesh decomposed in (a) two and (b) eight sub-domains. The program used to generate the visualization was the PMVis (Partitioned Mesh Visualizer). During the partition process, the elements, the nodes, and the degrees of freedom (dof) are renamed in order to permit each task to visualize its sub-domain as a domain, without loosing the relation with the original domain.

Due to the mesh splitting between tasks there are interior and boundary dof. The interior dof belongs to a single sub-domain; on the other hand, the boundary dof belongs to two or more sub-domains. During the partition some auxiliary variables are created (vectors and matrices) and are responsible to keep the information necessary for matrix assembling and solution of CGM.



(a) (b) (c)

Figure 2. Bi-dimensional domain decomposition by METIS: (a) two, (b) four and (c) eight sub-domains



(a) (b)

Figure 3. Tri-dimensional domain decomposition by METIS: (a) two, and (b) eight sub-domains

3.2. Matrix assembling

Due to the domain decomposition, the stiffness matrix **A** must be restructured in order that the values of the products, needed by the CGM solution, do not have any change. Thus, the matrix **A** of each computing sub-domain is broken in four others $\mathbf{A_p}$, $\mathbf{A_s}$, $\mathbf{B_p}$ and $\mathbf{B_p}^T$, as shown in Fig. 4. $\mathbf{A_p}$ is a square matrix with size equals to the internal degrees of freedom of each sub-domain, where are stored the stiffness values referring to the internal degrees of freedom. In $\mathbf{A_s}$ are stored the values of the stiffness matrix relating to the degree of freedom on the partition boundary.

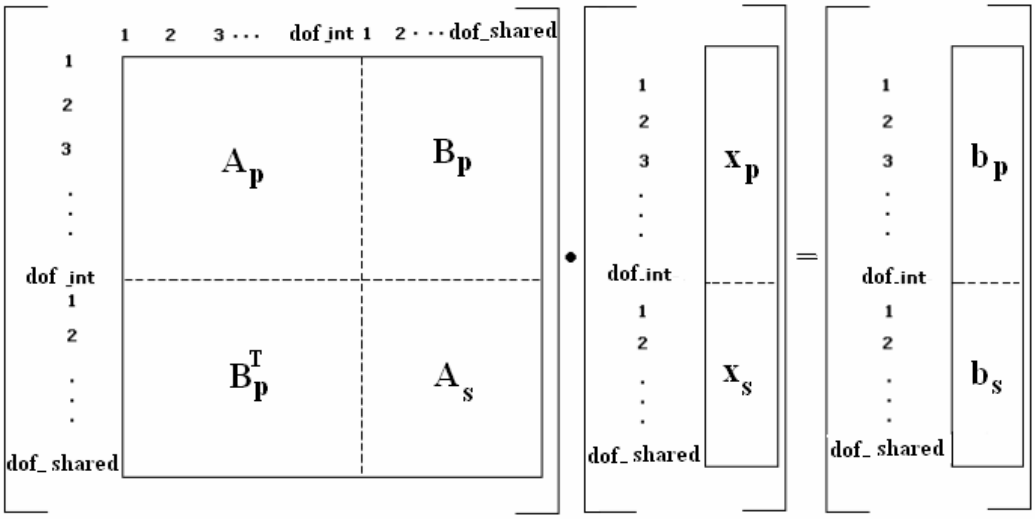


Figure 4. Matrix assembly for each sub-domain

The matrix $\mathbf{A_s}$ is also a square matrix and its size is defined by the number of degrees of freedom that are located on the boundary between partitions. The size of $\mathbf{B_p}$ is defined by the degrees of freedom located both on the partition boundary and on the partition's interior. This matrix is used to store the stiffness values that relate the internal dof with the partition boundary ones. $\mathbf{B_p}^{\mathbf{T}}$ is the transpose of $\mathbf{B_p}$. Due to the restructuring of the matrix **A**, the vectors **x** (displacements) and **b** (forces) are also reformulated. Each of them is broken in two parts. The vector **x** is decomposed in $\mathbf{x_p}$ and $\mathbf{x_s}$, where the first has the size equal to the internal dof and the second equals to the partition boundary ones. The same, for the vector **b** (Jimack and Touheed, 2000).

3.3. Parallel algorithm

The CGM is used to solve the equations of the sub-domains. This solution requires the communication between all the processors in order to work with the shared elements in each sub-domain boundary. The parallel algorithm for the CGM is shown in Fig. 5. In this version, the index ***i*** goes from **0** to the number of tasks minus one and the counter ***k*** points which iteration is been worked upon. From inside the CGM program, the algorithm calls the functions *InnerProduct*(), which does the inner product of vectors belonging to different tasks, and *Update*(), responsible to send the calculated values of each dof on the partition boundary to another task, which also shares it, refreshing these values, using the communication primitive functions *send*() and *receive*() (Jimack and Touheed, 2000).

For an arbitrary tolerance value ε

1. $\mathbf{x}_i^0 = \mathbf{0}$; $\mathbf{x}_{Si}^0 = \mathbf{0}$; $\beta^0 = 0$
2. Update($\mathbf{b}_{Si}$)
3. $\mathbf{r}_i^0 = \mathbf{b}_i$; $\mathbf{r}_{Si} = \mathbf{b}_{Si}$
4. $\mathbf{d}_i^0 = \mathbf{b}_i$; $\mathbf{d}_{Si} = \mathbf{b}_{Si}$
5. $\gamma^0 = \text{InnerProduct}(\mathbf{r}_i^0, \mathbf{r}_{Si}^0 \; ; \; \mathbf{r}_i^0, \mathbf{r}_{Si}^0)$

For $k = 0, 1, 2, ..$ repeat steps 6 to 17 until $\sqrt{\gamma^{k+1}} \leq \varepsilon$

6. $\mathbf{q}_i^k = \mathbf{A}_{Pi} \cdot \mathbf{d}_i^k + \mathbf{B}_{Pi} \cdot \mathbf{d}_{Si}^k$
7. $\mathbf{q}_{Si}^k = \mathbf{B}_{Pi}^{\mathrm{T}} \cdot \mathbf{d}_i^k + \mathbf{A}_{Si} \cdot \mathbf{d}_{Si}^k$
8. Update($\mathbf{q}_{Si}^k$)
9. $\tau^k = \text{InnerProduct}(\mathbf{d}_i^k, \mathbf{d}_{Si}^k \; ; \; \mathbf{q}_i^k, \mathbf{q}_{Si}^k)$
10. $\alpha^k = \gamma^k / \tau^k$
11. $\mathbf{x}_i^{k+1} = \mathbf{x}_i^k + \alpha^k \mathbf{d}_i^k$; $\mathbf{r}_i^{k+1} = \mathbf{r}_i^k - \alpha^k \mathbf{q}_i^k$
12. $\mathbf{x}_{Si}^{k+1} = \mathbf{x}_{Si}^k + \alpha^k \mathbf{d}_{Si}^k$; $\mathbf{r}_{Si}^{k+1} = \mathbf{r}_{Si}^k - \alpha^k \mathbf{q}_{Si}^k$
13. store γ^k
14. $\gamma^{k+1} = \text{InnerProduct}(\mathbf{r}_i^{k+1}, \mathbf{r}_{Si}^{k+1} \; ; \; \mathbf{r}_i^{k+1}, \mathbf{r}_{Si}^{k+1})$
15. If $\sqrt{\gamma^{k+1}} \leq \varepsilon$ STOP
16. $\beta^k = \gamma^{k+1} / \gamma^k$
17. $\mathrm{d}_i^{k+1} = \mathbf{r}_i^{k+1} + \beta^k \mathbf{d}_i^k$; $\mathrm{d}_{Si}^{k+1} = \mathbf{r}_{Si}^{k+1} + \beta^k \mathbf{d}_{Si}^k$

Figure 5**.** Algorithm for parallel solution of the CGM

4. Results

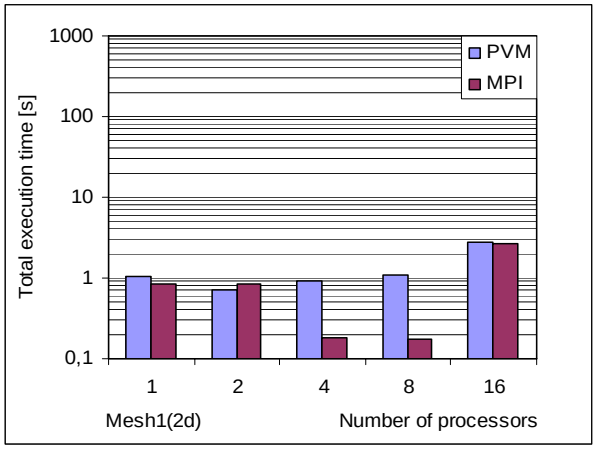
The runtime of a parallel program depends on two variables: the size of the problem and the number of processes. The performance of a parallel program is a complex and multifaceted issue. It must be considered, in addition to the execution time and scalability of the computational kernels, the mechanisms by which data are generated, stored, transmitted over networks, moved to and from disk, and passed between different stages of a computation. The metrics by which performance is measured can be as diverse as execution time, parallel efficiency, memory requirements, throughput, latency, input/output rates, network throughput, etc., (Foster, 1995). The execution time is the time that has elapsed from the moment when the first process to start actually begins execution of the program to the moment when the last process to complete execution executes its last statement (Pacheco, 1997).

We focus our work on execution time and parallel scalability because they are frequently among the more problematic aspects of parallel program design and because they are the most easily formalized in mathematical models. To evaluate the performance of the two communication libraries, *mpich-1.2.5* and *pvm3.4.4*, we used five bi-dimensional (2d) meshes composed by linear triangular elements, and five tri-dimensional (3d) meshes composed by linear tetrahedrons as detailed in Tab. 1. The parallel executions were made assigning only one task for each processor of a $11,000 COTS cluster composed of eight dual SMP AMD Athlon MP 1900+, 1.7 GHz, 1.0 GB RAM, 256 KB internal cache memory, 1.0 Gb/s Ethernet switch, with Red Hat 9.0 Linux operating system.

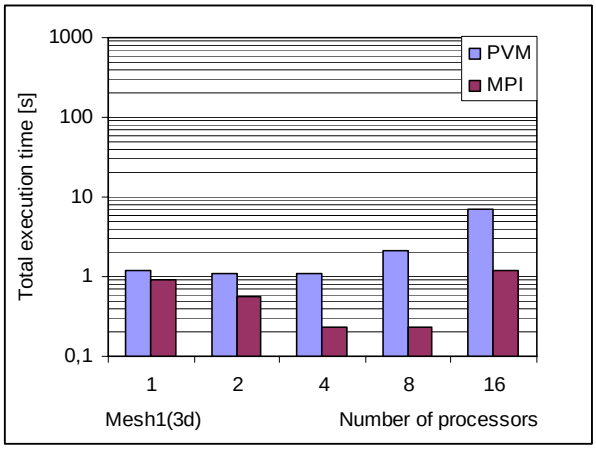
Table 1. Characteristics of the simulated finite element meshes.

Meshes	Number of nodes		Number of elements		Number of equations		Number of restrict dof	
	2d	3d	2d	3d	2d	3d	2d	3d
Mesh1	357	233	643	855	684	618	30	81
Mesh2	741	467	1,383	1,882	1,441	1,266	41	135
Mesh3	1,455	933	2,770	4,014	2,852	2,577	58	222
Mesh4	2,904	1,984	5,609	9,126	5,728	5,565	81	387
Mesh5	5,808	3,717	11,334	18,031	11,501	10,581	115	570

From Fig. 6 to Fig. 10 are presented the total executions time versus number of tasks or processors (one, two, four, eight and sixteen) for the ten meshes under test, for both message passing libraries. These values were obtained through the use of the *gettimeofday*() function, with the cluster dedicated to the execution of the parallel program.

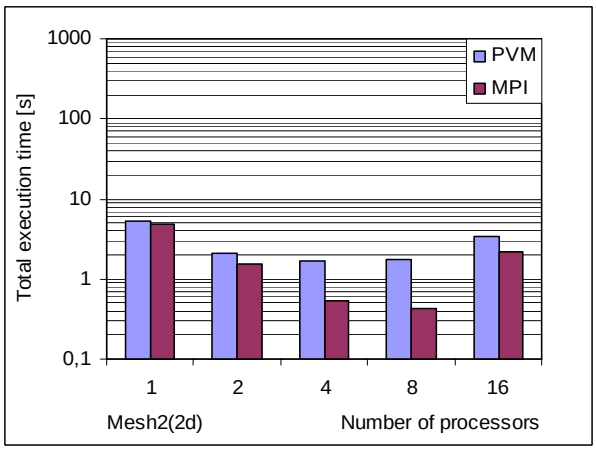


(a)

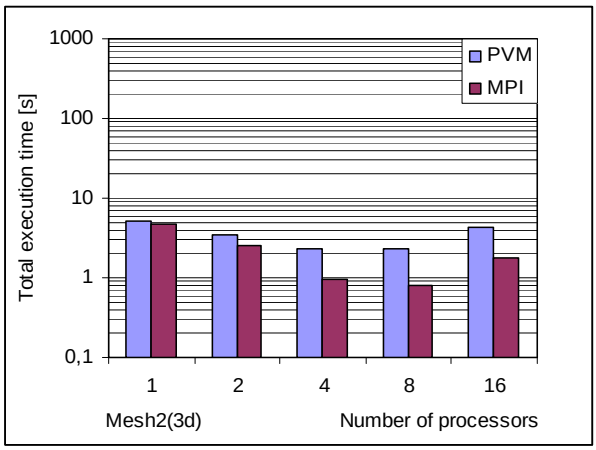


(b)

Figure 6. Total execution time versus number of tasks for Mesh1: (a) 2d and (b) 3d

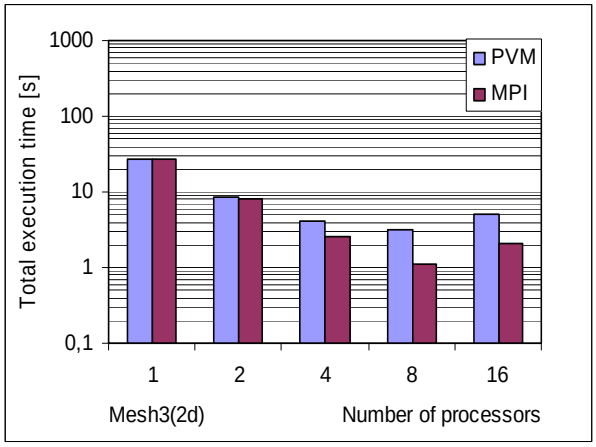


(a)

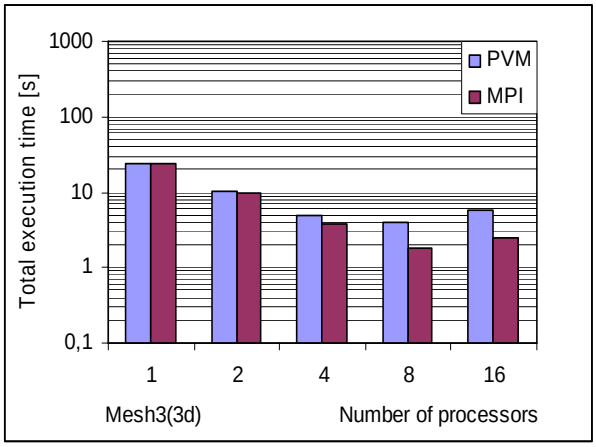


(b)

Figure 7. Total execution time versus number of tasks for Mesh2: (a) 2d and (b) 3d

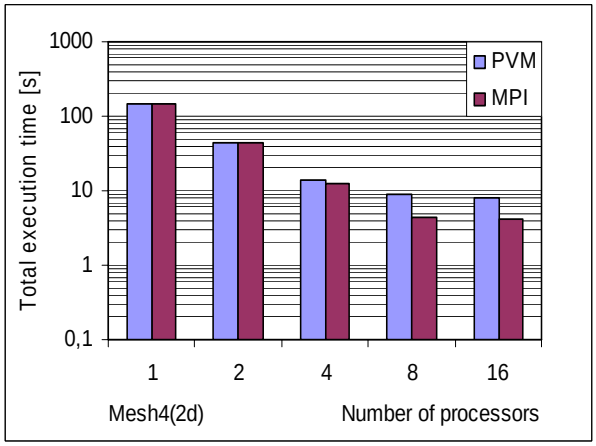


(a)

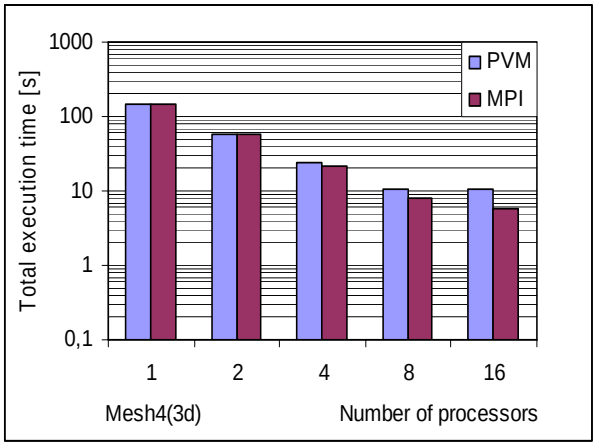


(b)

Figure 8. Total execution time versus number of tasks for Mesh3: (a) 2d and (b) 3d

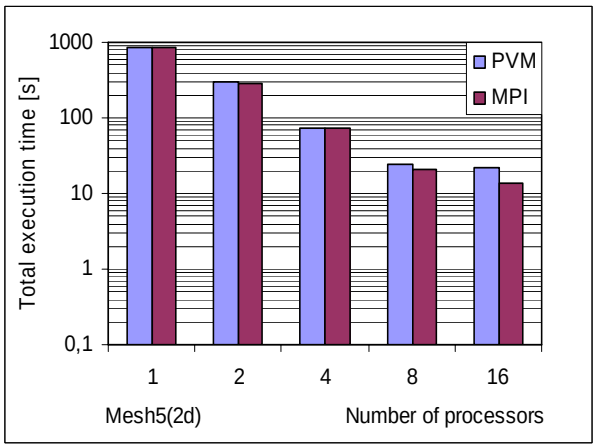


(a)

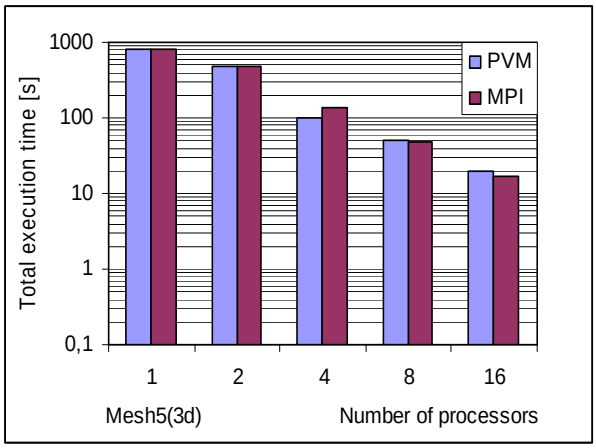


(b)

Figure 9. Total execution time versus number of tasks for Mesh4: (a) 2d and (b) 3d



(a)



(b)

Figure 10. Total execution time versus number of tasks for Mesh5: (a) 2d and (b) 3d

It can be observed in these figures that the smaller meshes, for which the costs of parallelization (communication) are proportionally more significant, compared to the execution cost itself, present degradation in performance when more processors are added to the solution of the problem. In these cases the MPI presents better results then PVM, which has associated the cost of the virtual machine.

This handicap tends to disappear when the size of the problem grows, and the two libraries passes to present a similar behavior. From the foregoing observations we can conclude that distributed systems perform better as the load increases, that is, distributed system response time is robust under heavy loading (El-Rewini and Lewis, 1997).

5. Conclusions

In this work were presented results of the performance of two simulation programs for a structural analysis, made parallel on a cluster of PCs, through the use of the *mpich-1.2.5* and *pvm3.4.4* message passing libraries of communication.

Some impressions in regards to the utilization of these libraries can briefly be presented. PVM is associated to a daemon program (*pvmd*) that configures a parallel virtual machine. This virtual machine is controlled by a shell, which allows adding or removing processors on the parallel machine, verifying the nodes being running, etc. These characteristics of the PVM permit to the users more transparency and control of the parallel virtual machine (Geist *et al*., 1994). On the other hand, the MPI has more facilities to perform executions than PVM, because it is not necessary to start the daemon previously, and also the fact that MPI presents some commands that can be utilized directly by the users for compilation (*mpicc*) and execution (*mpirun*) (Snir, M., *et al*., 1996), (Pacheco, 1997).

In this work, we could verify that the utilization of low cost COTS clusters of PCs (for less than $11,000) in the solution of engineering problems can be an interesting alternative, due to the low execution costs and high efficiency attained. For the solution of an elasticity problem, utilizing the finite elements method, we can also obtain excellent results on a PC network already existent without additional costs of hardware or software.

6. Acknowledgements

We would like to express our gratitude to the Institute of Exact Sciences, University of Brasilia, for the use of the Beowulf cluster, and to Capes for the grant support.

7. References

Dongarra,J., *et al*., 2003, “The Sourcebook of Parallel Computing”, ed. Morgan Kaufmann, San Francisco, USA, 832 p.

El-Rewini, H. and Lewis, T.G., 1997, “Distributed and Parallel Computing, Manning, Greenwich, CT, USA, 469 p.

Foster, I., 1995, “Designing and building parallel programs: concepts and tools for parallel software engeneering”, Addison-Wesley, New York, USA, 381 p.

Geist, A., *et al*., 1994, “PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing”, ed. MIT Press, Cambridge, USA, 176 p.

Gropp, W., Lusk, E. and Skjellum, A., 1999, “Using MPI: Portable Parallel Programming with the Message Passing Interface”, 2nd edn., ed. MIT Press, Cambridge, USA, 371 p.

Hughes, C. and Hughes, T., 2004, “Parallel and Distributed Programming Using C++”, ed. Addison-Wesley, New York, USA, 691 p.

Jimack, P.K., Touheed, N., 2000, “Developing Parallel Finite Element Software Using MPI”, In: Topping, B.H.V., Lamner, L. (eds.), HPC for Computational Mechanics, Saxe-Coburg, pp. 15–38.

Karypis, G., Kumar, V., 1998, “METIS* A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices”, Technical report, Univ. of Minnesota, Dep. of Comput. Science/Army HPC Res. Center, Minneapolis, USA.

Kim, S.J. and Kim, J.H., 1997, “Large-scale structural analysis using domain decomposition method on distributed parallel computing environment”, Proceedings of High Performance Computing on the Information Superhighway (HPC Asia'97), Seoul, Korea, pp. 573–578.

Kumar, V., Grama, A., Gupta A and Karypis, G., 2003, “An Introduction to Parallel Computing: Design and Analysis of Algorithms”. 2nd edn., ed. Addison-Weslley, New York, USA, 597 p.

Ong, H. and Farrel, P.A., 2000, “Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network”, Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, USA.

Pacheco, P.S., 1997, “Parallel programming with MPI”, Morgan Kaufmann, San Francisco, USA, 418 p.

Snir, M., *et al*., 1996, “MPI: the complete reference”, MIT Press, Cambridge, USA, 350p.

Tan, L.H. and Bathe, K.J., (1991), “Studies of finite element procedures – The CG Method and GMRES in ADINA and ADINA-F”, Computer & Structures, 40 (2), pp. 441–449.

TOP500, 2004, “SUPERCOMPUTER SITES”, http://www.top500.org/.

8. Responsibility notice

The authors are the only responsible for the printed material included in this paper.