Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXVIII, págs. 205-215 (artículo completo)
Cristian García Bauza, Pablo Lotito, Lisandro Parente, Marcelo Vénere (Eds.)
Tandil, Argentina, 3-6 Noviembre 2009

PARALLEL IMPLEMENTATION OF A FEM CODE BY USING MPI/PETSC AND OPENMP HYBRID PROGRAMMING TECHNIQUES

Hugo G. Castro[a,b], Rodrigo R. Paz[a], Mario A. Storti[a], Victorio E. Sonzogni[a] and Lisandro Dalcín[a]

[a]*Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC), INTEC-CONICET-UNL, Guemes 3450, (S300GLN) Santa Fe, Argentina, http://www.cimec.org.ar*

[b]*Grupo de Investigación en Mecánica de Fluidos, Universidad Tecnológica Nacional, Facultad Regional Resistencia, Chaco, Argentina, hugoguillermo_castro@yahoo.com.ar*

Keywords: MPI, OpenMP, PETSc, SMP, hybrid programming.

Abstract. The so called "hybrid parallelism paradigm", that combines programming techniques for architectures with distributed and shared memories using MPI ('Message Passing Interface') and OpenMP ('Open Multi-Processing') standards, is currently adopted to exploit the growing use of multicore computers in PC's clusters, thus improving the efficiency of codes in such architectures (several multicore nodes or clustered symmetric multiprocessors (SMP) connected by a fast net to do exhaustive computations).

Meanwhile, for large-scale applications projects there is PETSc, a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations, implementing MPI standard and widely used in parallel finite element codes development.

In this paper a parallel hybrid finite element code is developed and its performance evaluated, using MPI/PETSc for communication between nodes and OpenMP for parallelism within an SMP node. The cluster in which the code was run was the CIMEC's 'Coyote' cluster, which consists of 8 nodes with 8 cores each, connected through a Fast Ethernet net of 1Gb/sec.

1 INTRODUCTION

Many engineering and scientific problems related to Computational Fluid Dynamics (CFD) require intense numerical computation. A lot of research has been carried out using methodologies for obtaining solutions to large-scale problems in realistic time, (Behara and Mittal, 2009).

As commodity off-the-shelf symmetric multi-processors (SMPs) and high-speed network devices are widely deployed, SMP clusters have become an attractive platform for high-performance computing. To exploit such computing systems the tendency is to use the so called *hybrid parallelism paradigm* that combines programming techniques for architectures with distributed and shared memory, often using MPI and OpenMP standards, (Jost and Jin, 2003).The hybrid MPI/OpenMP approach is based on using MPI for coarse grained parallelism and OpenMP for fine grained loop level parallelism. The MPI programming paradigm assumes a private address space for each process. Data is transferred by explicit message exchange via calls to the MPI library. This model was originally designed for distributed memory architectures but is also suitable for shared memory systems.

Whilst message passing is necessary to communicate between nodes, it is not an efficient technique within a SMP node. In theory, a shared memory model such as OpenMP should offer a more efficient parallelization strategy within an SMP node. Hence a combination of shared memory and message passing parallelism paradigms within the same application (mixed mode programming) may provide a more efficient parallelization strategy than pure MPI.

This paper has been focused on mixed MPI/PETSc and OpenMP implementation on a finite element code for scalar PDE's and discusses the benefits of developing mixed mode MPI/OpenMP applications on the CIMEC's 'Coyote' beowulf cluster.

Section 2 provides a comparision of the different characteristics of the OpenMP and MPI paradigms. Section 3 discusses the implementation of mixed mode applications and Section 4 describes a number of situations where mixed mode programming is potentially beneficial. In Section 5 we describe the implementation of a mixed mode application and compare and contrast the performance of the code with pure MPI and OpenMP versions.

The results demonstrate that this style of programming may increase the code performance, taking account of some rule-of-thumb depending on the application on hand.

2 PROGRAMMING MODEL CHARACTERISTICS

2.1 MPI

The message passing programming model is a distributed memory model with explicit control paralelism. MPI, (MPI, 2009), is portable to both distributed and shared memory architecture. The explicit parallelism often provides a better performance and a number of optimized collective communication routines are available for optimal efficiency.

However MPI suffers from a few deficiencies. Debugging of applications can be time consuming and communications can create a large overhead and the code granularity often has to be large to minimise the latency. Finally, global operations can be very expensive.

2.2 PETSc

The Portable Extensible Toolkit for Scientific Computation (PETSc), (Balay et al., 2009), is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the MPI standard for all message-passing communication. The original purpose of PETSc was to enable its users to easily experiment between many different models, solving methods and discretizations and to eliminate the MPI from MPI programming. It is a freely available research code usable from C/C++, Fortran 77/90 and Python. PETSc has run problems with over 500 million unknowns, also it has run efficiently on $6,000$ processors. It has also been used in applications running at over 2 Teraflops. An overview of some of the components of PETSc can be seen in Fig. (1). An important feature of the package is the posibility to start at a high level and work your way down in level of abstraction.

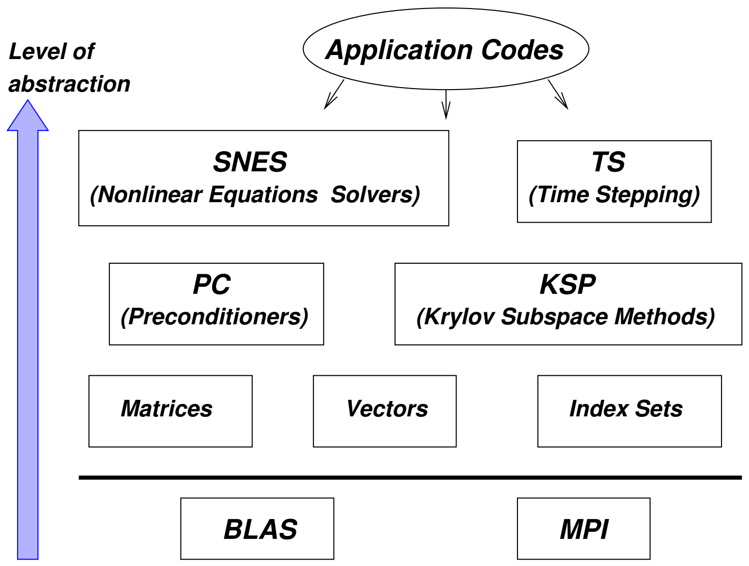

Figure 1: Hierarchical structure of the PETSc library

PETSc employs the distributed memory model. Each process has its own address space in memory and when needed, information is passed to other processes via MPI. To be more specific in e.g. a finite element problem, each process will own a contiguous subset of rows of the system matrix and will primarily work on this subset, sending/receiving information from/to other processes only when needed. This programming model makes PETSc specially suitable for clusters.

Unlike OpenMP, PETSc is not easily integrated in existing codes. PETSc is a set of library interfaces, with its own matrix and vector structures, assembly and solving methods which are organized in an object oriented way. The user declares the system matrices and vectors, and then uses PETSc to assemble, precondition and solve the problem by invoking calls to PETSc methods. Through the whole process, there is ample possibility for the user to customize and control the solution process. It is also possible to supply most options at the command line, which makes experimenting between different methods very easy.

2.3 OpenMP

OpenMP stands for Open specifications for Multi Processing. It is a collaborative work between interested parties from the hardware and software industry, government and academia. Essentially, OpenMP is an API (Application Program Interface) for writing multithreaded applications. It allows parallelise existing code, giving to programmer full control over the parallelization. It is comprised of three primary API components: compiler directives, runtime library options and environment variables.

OpenMP is mainly designed for symmetric multiprocessing (SMP) architectures i.e. systems with multiple CPUs on a shared memory. OpenMP is available for Fortran (77, 90 and 95) and C/C++. It has been implemented on many platforms including most Unix platforms and Windows. A full description of the OpenMP specification can be found in OpenMP (2009).

OpenMP uses a shared memory process consisting of multiple threads. The fork-join model execution is utilized, see Fig. (2).

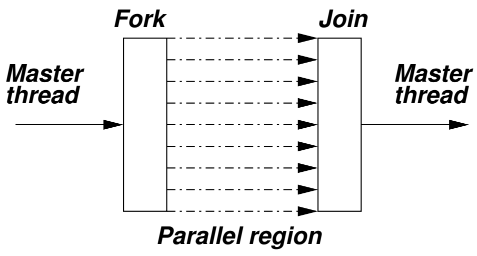

Figure 2: The master thread creates a team of parallel threads.

The OpenMP program is initially executed as a single process by the master thread. When entering a parallel region, the master thread creates a team of parallel threads, this is called fork. The statements in the program that are enclosed by the parallel region are then executed in parallel among the threads in the team. When all threads have finished their work in the parallel region, they synchronize and terminate, leaving only the master thread, this is called join. In the parallel region, OpenMPs memory model dictates that all variables except loop variables by default are shared by the different threads. One can alter this by specifying variables as private or shared before entering the parallel region. The model also supports nested parallelism i.e. it is possible to have parallel regions within other parallel regions.

OpenMP is used by inserting special compiler directives or runtime library routine calls into the existing code. The compiler directives are inserted as special comments (beginning with `#pragma omp`).

2.4 Mixed mode programming

By utilising a mixed mode programming model we should be able to take advantage of the benefits of both models. For example a mixed mode program may allow the data placement policies of MPI to be utilised with the finer grain parallelism of OpenMP. The majority of mixed mode applications involve a hierarchical model; MPI parallelisation ocurring at the top level, and OpenMP parallelisation ocurring below. Fig. (3) from the work of Smith and Bull (2001) shows a 2D grid which has been divided geometrically between four MPI processes.

These subarrays have been then further devided between three OpenMP threads. This model closely maps to the architecture of an SMP cluster, the MPI parallelisation ocurring between the SMP nodes and the OpenMP parallelisation within the nodes.

3 IMPLEMENTING A MIXED MODE APPLICATION

Although a large number of MPI implementations are thread safe, this cannot be guaranteed. To ensure the code is portable, all MPI calls should be made within thread sequential regions of the code. This often creates little problem as generally codes involve OpenMP parallelisation ocurring beneath the MPI parallelisation and hence the majority of MPI calls occur outside the OpenMP parallel regions. When MPI calls occur within an OpenMP parallel region the calls should be placed inside a `CRITICAL`, `MASTER` or `SINGLE` region, depending on the nature

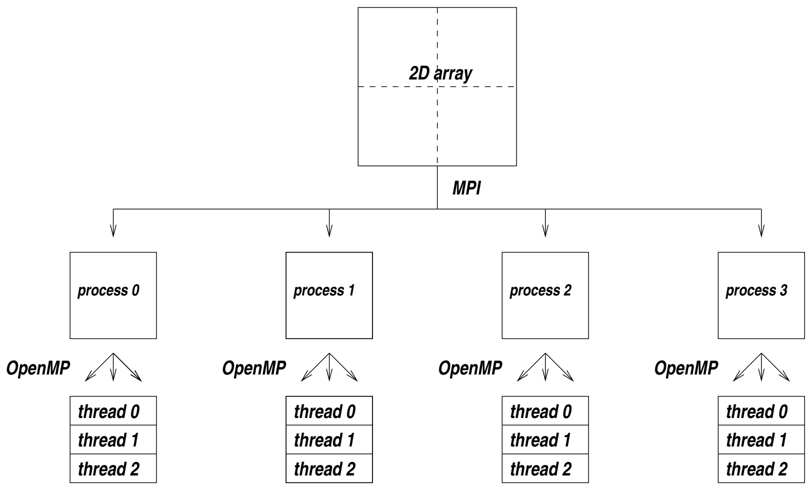

Figure 3: Schematic representation of hierarchical mixed mode programming model for a 2D array.

of the code, (Smith and Bull, 2001). When writing a mixed mode application it is important to consider how each paradigm carries out parallelisation, and whether combining the two mechanisms provides an optimal parallelisation strategy.

3.1 Benefits of mixed mode programming

According to Smith and Bull (2001), the situations where a mixed mode code may be more efficient than a corresponding MPI implementation are the following:

3.1.1 Codes which scales poorly with MPI

One of the largest areas of potential benefit from mixed mode programming is with codes which scale poorly with increasing MPI processes. If, for example, the corresponding OpenMP version scales well then an improvement in performance may be expected for a mixed mode code. If, however, the equivalent OpenMP implementation scales poorly, it is important to consider the reasons behind the poor scaling and whether these reasons are different for the OpenMP and MPI implementations. If both versions scale poorly for diferent reasons, for example the MPI implementation involves too much load imbalance and the OpenMP version suffers from cache misses due to data placement problems, then a mixed version may allow the code to scale to a larger number of processors before either of these problems become apparent. If however both the MPI and OpenMP codes scale poorly for the same reason, developing a mixed mode version of the algorithms may be of little use.

3.1.2 Load balance problems

One of the most common reasons for an MPI code to scale poorly is load imbalance. For example irregular applications such adaptive mesh refinement codes suffer from load balance problems when parallelised using MPI. By developing a mixed mode code for a clustered SMP system, MPI need only be used for communication between nodes, creating a coarser grained problem. The OpenMP implementation does not suffer from load imbalance and hence the performance of the code would be improved.

3.1.3 Fine grain parallelism problems

OpenMP generally gives better performance on fine grain problems, where an MPI application may become communication dominated. Hence when an application requires good scaling with a fine grain level of parallelism a mixed mode program may be more efficient. In theory, the performance of a pure MPI implementation should be poorer than a pure OpenMP implementation for these fine granularity situations. A mixed mode code could provide better performance, as load balance would only be an issue between SMPs, which may be achieved with coarser granularity.

3.1.4 Bandwidth and latency limited problems

The bandwidth and latency between SMP nodes can influence the performance of some codes substantially. Developing a mixed MPI/OpenMP version of a code previously written in MPI will typically reduce the number of inter-node messages, but increase the size of these messages.

On a simple interconnect, which only allows one message at a time to be sent/received by a node, the message bandwidth will be unaffected, since the same total amount of data is being transferred. The total latency, however, will decrease as there are fewer messages, resulting in a potential increase in performance. On a more sophisticated interconnect, which allows concurrent sending/receiving of messages, the smaller number of larger messages may have a detrimental effect, as the total network bandwidth cannot be exploited as efficiently.

4 HYBRID MPI/OPENMP PARALLELIZATION

The code used to implement the hybrid parallelism is a C++ code for the solution of scalar partial differential equations (advective-diffusive systems) by means of the stabilized finite element method. Initially the parallel code was implemented in C++ with MPI as message passing interface, having five main regions: *(I)* Problem Partition region: each node stores all the elements and associated degrees-of-freedom and creates parallel matrix and vectors (`MatCreateMPIAIJ`, `VecCreateMPI`), *(II)* FEM Matrix Computations: perform a loop over all elements in the mesh and computes each element matrices, *(III)* Matrix/Vector Values Setting: vector and matrix element contributions are stored in PETSc Matrices and Vectors classes (`VecSetValue`, `MatSetValue`), *(IV)* Matrix/Vector Scatters: parallel assemble all contributions in the global vector/matrix and *(V)* Linear System Solver: using an iterative Krylov Space-based linear solver.

The equipment used was the CIMEC's 'Coyote' which is a Beowulf kind of cluster, consisting of a server and 8 nodes: 6 nodes of 2 quad-core Intel Xeon E5335 2.5 GHz CPU with 8 Gb RAM per node and 2 nodes of 2 quad-core Intel Xeon E5335 2.0 GHz CPU with 8 Gb RAM per node (the server has 12 Gb RAM), interconnected via a Gigabit Ethernet network.

4.1 Test Case: Convection-diffusion skew to the mesh

The linear scalar advection-diffusion equation to be solved is,

$$\begin{aligned} a \cdot \nabla u - \nabla \cdot (\nu \nabla u) = s \quad & \in \Omega \\ u = u_D \quad & \text{on } \Gamma_D \end{aligned} \tag{1}$$

where u is the unknown scalar, $a(x)$ is the convection velocity vector (also known as advection velocity), $\nu > 0$ is the diffusivity (or thermal conductivity) of the medium and $s(x)$ is a volumetric source term. The variational formulation of the problem (1) is written as follows: find $u^h \in S^h$ such that

$$\begin{aligned}&\int_\Omega \nabla\omega^h \cdot (\nu\,\nabla u^h)\,d\Omega + \int_\Omega \omega^h\,(a\cdot\nabla u^h)\,d\Omega + \\ &\sum_{e=1}^{n_{el}} \int_{\Omega^e} (a\cdot\nabla\omega^h)\,\tau^{supg}\,[a\cdot\nabla u^h - \nabla\cdot(\nu\,\nabla u^h) - s]\,d\Omega = \int_\Omega \omega^h\,s\,d\Omega, \qquad \forall\omega^h \in \mathcal{V}^h\end{aligned} \tag{2}$$

where

$$\begin{aligned}\mathcal{S}^h &= \{u^h | u^h \in \mathcal{H}^1(\Omega),\ u^h|_{\Omega^e} \in \mathcal{P}_m(\Omega^e)\ \forall e \text{ and } u^h = u_D \text{ on } \Gamma_D\}\\ \mathcal{V}^h &= \{\omega^h | \omega^h \in \mathcal{H}^1(\Omega),\ \omega^h|_{\Omega^e} \in \mathcal{P}_m(\Omega^e)\ \forall e \text{ and } \omega^h = 0 \text{ on } \Gamma_D\}\end{aligned} \tag{3}$$

where $\mathcal{P}_m$ is the finite element interpolating space and for the sake of simplicity only Dirichlet boundary conditions are considered. According to Tezduyar and Osawa (2000) the stabilization parameter τ^{supg} can be defined as

$$\tau^{\text{supg}} = \frac{h_{\text{supg}}}{2||a||} \tag{4}$$

where

$$h_{\text{supg}} = 2\left(\sum_{i=1}^{n_{en}} |\mathbf{s}\cdot\nabla\omega_i|\right)^{-1}, \tag{5}$$

where $\mathbf{s}$ is a unit vector pointing to the streamline direction. The problem statement is depicted in Fig. (4) where the computational domain is a unit square, i.e., $\overline{\Omega} = [0,1] \times [0,1]$. This 2D test case has been widely used to illustrate the effectiveness of stabilized finite element methods in the modeling of convection-dominated flows, (Donea and Huerta, 2003).

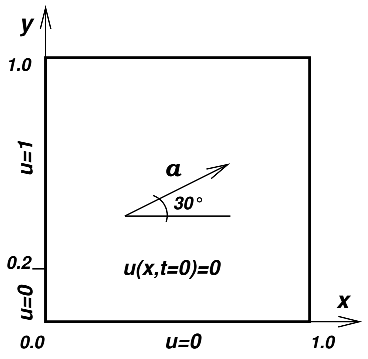

Figure 4: Convection of discontinuous inlet data skew to the mesh: problem statement.

The flow is unidirectional and constant, $||a|| = 1$, but the convective velocity is skew to the mesh with an angle of $30°$. The diffusivity coefficient is taken to 10^{-4}, thus obtaining an advective-dominated system. The inlet boundary data are continuous and at the oulet an homogeneous natural boundary condition was considered. The result for this case is displayed in Fig. (5). In order to compute the performance of the proposed algorithm a mesh of 4.5 millions of linear triangles was used.

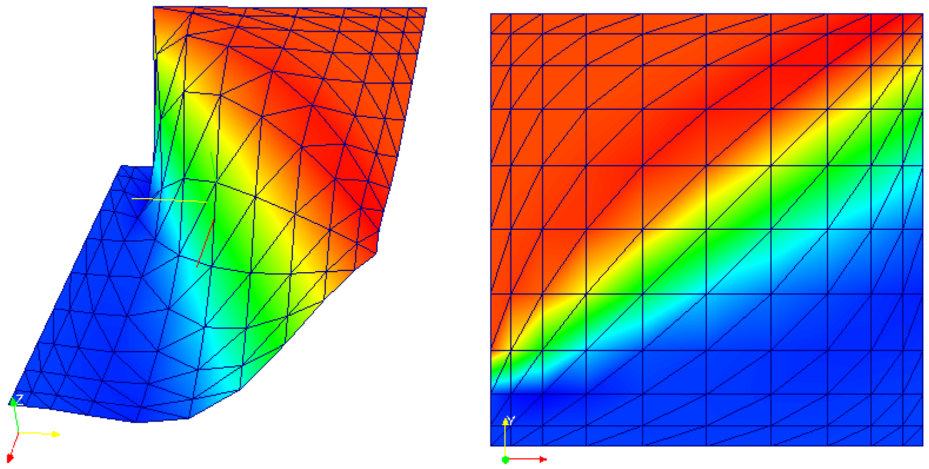

Figure 5: SUPG solution for the 2D convection-diffusion problem with downwind natural conditions.

5 RESULTS

The results for the consumed cpu time when sweep the number of SMP nodes (n_p) and number of cores (n_c) are shown in Figure (6) for the residual matrix calculation and in Figure (7) for the `MatMult` iterative solver operation (the kernel of Krylov Space-based iterative solver like the Stabilized Bi-Conjugate Gradient (BiCG-Stab) used in this work).

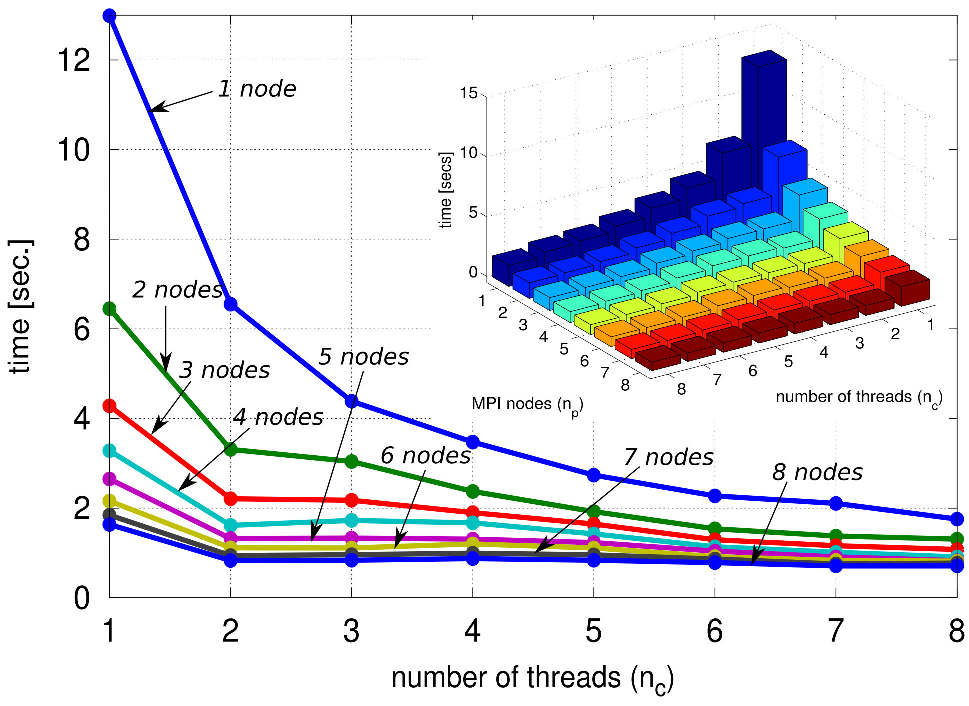

Figure 6: Elapsed time for the residual matrix computation.

The performance analysis of a parallel algorithm which is implemented on a given computer system is based on the execution time. Let T^* be the execution time to solved the problem on 1 node and 1 thread and T_p denote the execution time of the parallel algorithm when there are P processors in the system. In this work 'processors' are referred to the SMP nodes as much as to

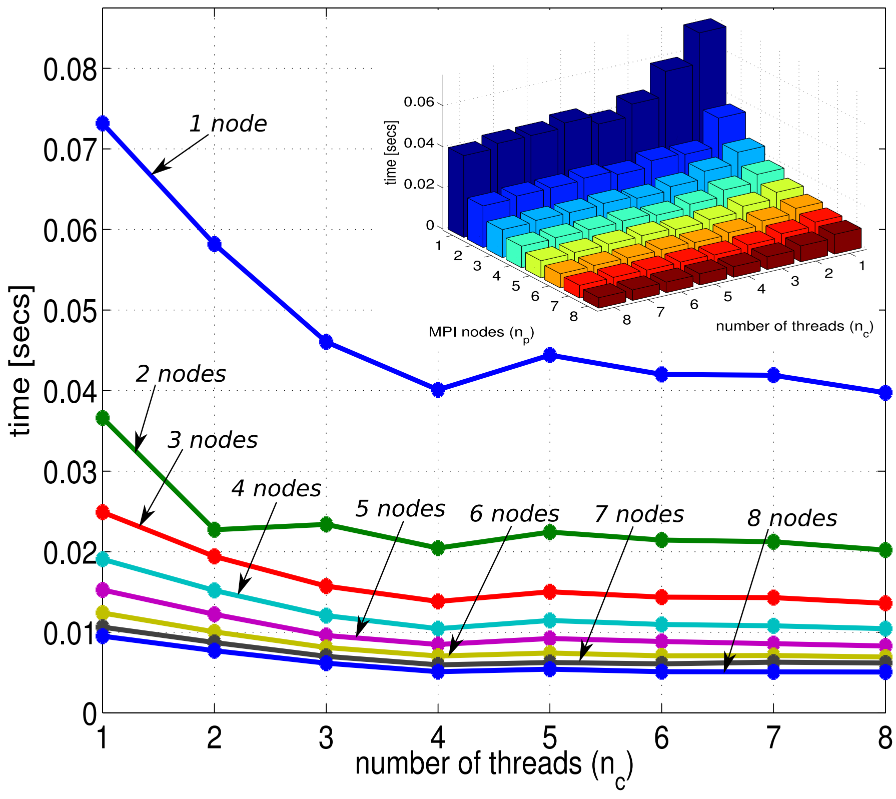

Figure 7: Elapsed time for the MatMult solver operation.

the cores within nodes. Then the speedup (S) and efficiency (E) are defined as

$$S = \frac{T^*}{T_p} \tag{6}$$

$$E = \frac{S}{P} = \frac{T^*}{T_p\, P} \tag{7}$$

respectively. The speedup obtained in the residual matrix calculation is illustrated in Figure (8) and for the `MatMult` iterative solver operation in Figure (9). Both figures shows at the left the speedup considering some node and then sweeping the number of cores available within each node. Reciprocally, at the right of both figures it can be seen the speedup for some number of cores and changing the numbers of nodes.

6 CONCLUSIONS

A stabilized finite element formulation of the linear scalar advection-diffusion equation has been implemented and tested on a cluster of SMP nodes. As shown in Figure (8) (*left*) a close-linear scalability of the FEM matrix computation task is achieved for the number of SMP nodes and threads available, thus improving the cpu consuming time on this typical finite element application.

Taken into account the linear system solve stage similar appreciations can be made. However, a 25%-50% of efficiency is achieved when computing in the range of 4-8 threads.

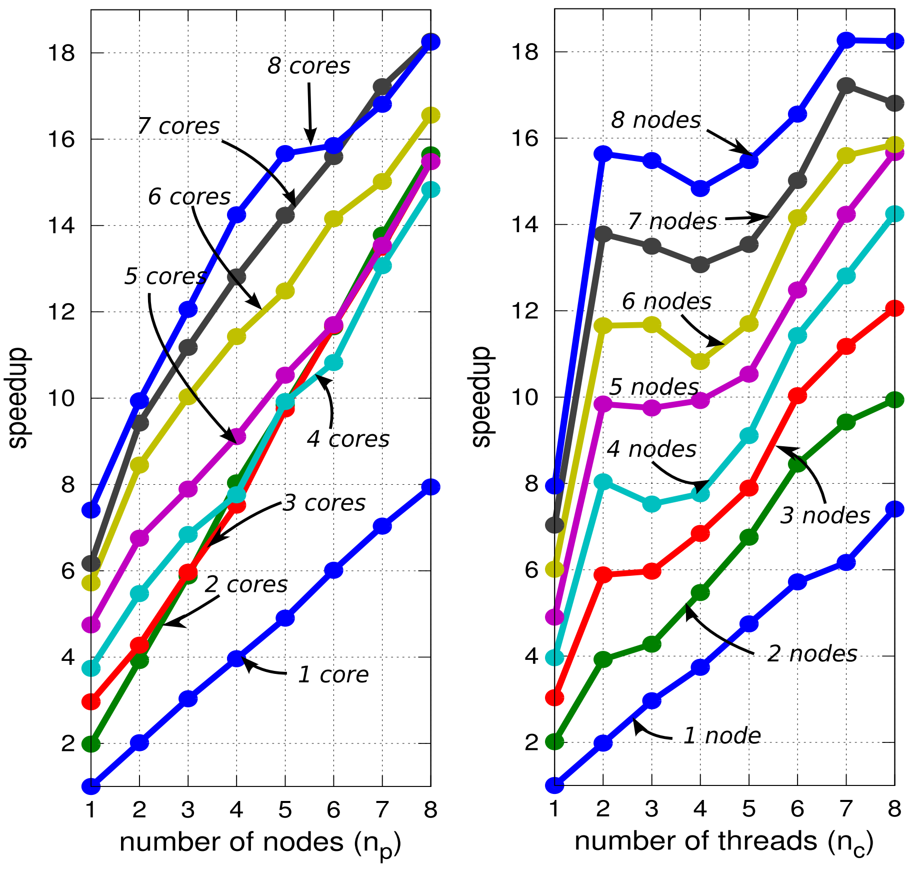

Figure 8: Speedup for the residual matrix computation.

7 ACKNOWLEDGMENTS

This work has received financial support from *Consejo Nacional de Investigaciones Científicas y Técnicas* (CONICET, Argentina, grant PIP 5271/05), *Universidad Nacional del Litoral* (UNL, Argentina, grants CAI+D 2009 65/334), *Agencia Nacional de Promoción Científica y Tecnológica* (ANPCyT, Argentina, grants PICT 01141/2007, PICT PME 209/2003, PICT-1506/2006, PICTO-23295/2004) and *Secretaría de Ciencia y Técnica de la Universidad Tecnológica Nacional, Facultad Regional Resistencia* (Chaco).

REFERENCES

Balay S., Buschelman K., Gropp W.D., Kaushik D., Knepley M.G., Curfman McInnes L., Smith B.F., and Zhang H. PETSc Web page. 2009. http://www.mcs.anl.gov/petsc.

Behara S. and Mittal S. Parallel finite element computation of incompresible flows. *Parallel Computing*, 35:195–212, 2009.

Donea J. and Huerta A. *Finite element methods for flow problems*. WILEY, 2003.

Jost G. and Jin H. Comparing the openmp, mpi, and hydrid programming paradigms on an smp cluster. *Fifth European Workshop on OpenMP*, 35:195–212, 2003.

MPI. MPI Web page. 2009. http://www.mpi-forum.org/.

OpenMP. OpenMP specification. 2009. http://www.openmp.org/mp-documents/spec30.pdf.

Smith L. and Bull M. Development of mixed mode mpi/openmp applications. *Scientific Programming*, 9:83–98, 2001.

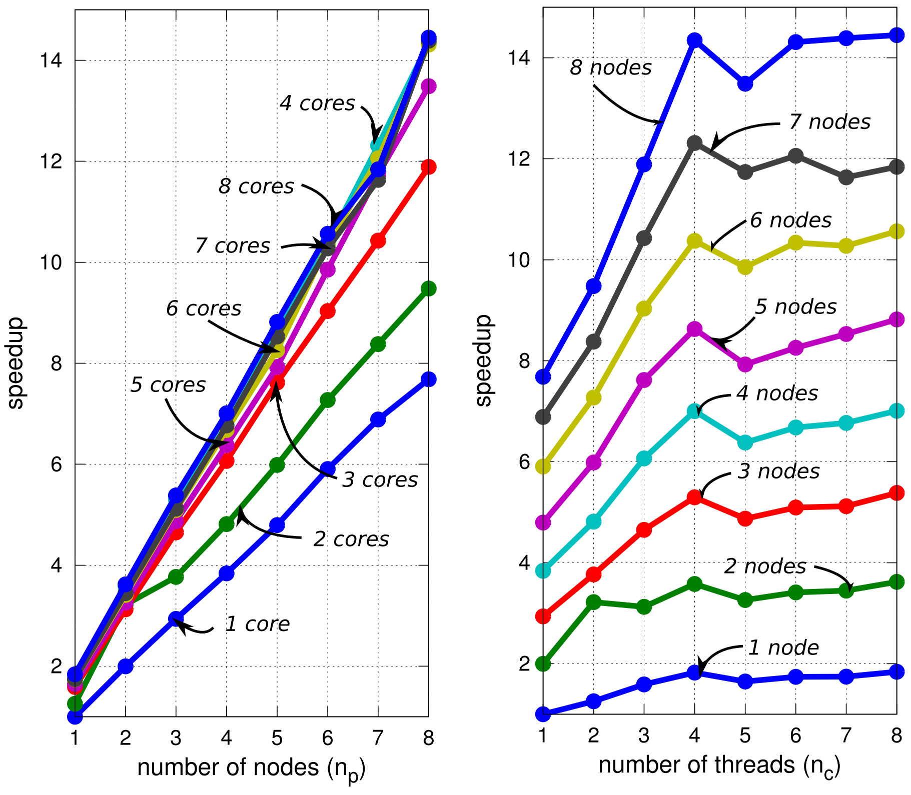

Figure 9: Speedup for the MatMult iterative solver operation.

Tezduyar T. and Osawa Y. Finite element stabilization parameters computed from element matrices and vectors. *Computer Methods in Applied Mechanics and Engineering*, 190:411–430, 2000.