PARALLELIZATION OF THE FINITE ELEMENT METHOD FOR HYBRID PARALLEL ARCHITECTURES

Henrique Conde Carvalho de Andrade

Ana Beatriz de Carvalho Gonzaga e Silva

Vicente Helano Feitosa Batista

Fernando Luiz Bastos Ribeiro

henrique@coc.ufrj.br

anabeatrizgonzaga@coc.ufrj.br

vicentehelano@cariri.ufc.br

fernando@coc.ufrj.br

Civil Engineering Program/Federal University of Rio de Janeiro

Cento de Tecnologia, Ilha do Fundão, CEP 21945-970, Rio de Janeiro, Brazil

Abstract.*In this work we present a parallel implementation of the finite element method designed for hybrid parallel machines, which is the case of PC clusters with multicore processors connected via local network. Compressed data structures are used to store the coefficient matrices and obtain iterative solutions in a subdomain-by-subdomain approach. The MPI standard is used for distributed memory interprocess communication, while multithreaded code is programmed in OpenMP. The MPI standard is also used for shared-memory, with intranode communication being performed simulating a virtual distributed-memory system in a multicore hardware. The efficiency of the proposed method is tested on three-dimensional elasticity and heat transfer problems, run on two different machines.*

Keywords:*finite element method, compressed data structures, hybrid parallelism, parallel computing*

1 INTRODUCTION

The evolution of computers technology in recent years have increased the capacity of scientists, engineers and mathematicians to solve complex problems involving the solution of large systems, that must be solved thousands of times. Some examples of these problems are nonlinear and time dependent analyses, which is the case of many engineering applications. To take advantage of this technological improvement, the employed must be optimized to obtain a solution in a reasonable time.

The finite element method is a numerical technique widely used to solve engineering problems in different fields. A typical finite element method algorithm is presented in Algorithm 1. One of the major issues concerning optimization techniques is the choice of efficient data structures used to store the matrix coefficients. In addition, there are several levels of parallelization that can be used to speed up the code, such as on-chip optimizations, distributed, shared and GPU parallelization.

The two most commonly used data structures in finite element iterative solutions are element-based and compressed data structures (Ribeiro & Coutinho, 2005). Compressed data structures may be implemented under the form of the well-known CSR format or as an edge-based scheme, both referring to the graph of the mesh. In this work we used a modified version of the original CSR, namely the CSRC(Compressed Sparse Row/Column), to store the coefficient matrix. A detailed discussion on this data structure is found in (F.L.B. Ribeiro & I.A. Ferreira, 2007), where a finite element subdomain-by-subdomain parallel implementation is presented. Differently from domain decomposition methods, in a subdomain-by-subdomain parallelization strategy the serial and parallel codes are exactly the same, the parallelization does not affect convergence and a better scalability may be expected.

Considering multithreaded platforms, the bottleneck is the access of large amount of shared memory. This issue reflects on a more difficult parallelization of the solver in comparison with the matrix assembly, due to matrix-vector operations. Matrix assembly can be straightforward parallelized by using a coloring mesh algorithm (G. Mahinthakumar & F. Saied, 2002,Hughes, Shakib, & Johan, 1989). Techniques to improve matrix-vector operations based on indices compression were proposed by (Kourtis, Goumas, & Koziris, 2008). Liu, Zhang, Sun, & Qiu (2009) tested some methods to balance thread work load using CSR. Krotkiewski & Dabrowski (2010) tested several versions of CSR for symmetric matrices.

In this paper we present a distributed/shared memory implementation of the finite element method using the MPI standard for interprocess communication and OpenMP directives for threaded code. The distributed memory counterpart of the present implementation is an improvement of the version found in (F.L.B. Ribeiro & I.A. Ferreira, 2007). In the present formulation, a communication map minimizes overall communications.

Algorithm 1- Main steps of a general finite element code.

Main steps of a general finite element code.

```
1.  $set\ \ t_0 = 0$
2.  $for\ \ i = 0,1,....\ do:$
3.  $t_{i+1} = t_i + \Delta t$
4.  $compute\ external\ \ forces\ \ f_{i+1}\ at\ time\ \ t_{i+1}$
5.  $set\ \ u_{i+1}^0 = u_i + (1-\alpha)\Delta t\ \dot{u}_i, \quad \dot{u}_{i+1}^0 = 0$
6.  $for\ \ j = 0,1,....\ until\ convergence\ do:$
7.        $r^j = f_{i+1} - M\,\dot{u}_{i+1}^j - K\,u_{i+1}^j,\ \ A = M + \alpha\Delta t K$
8.        $\Delta\dot{u}_{i+1}^{j+1} = A^{-1} r^j$
9.        $\dot{u}_{i+1}^{j+1} = \dot{u}_{i+1 i}^{j} + \Delta\dot{u}_{i+1}^{j+1}$
10.       $u_{i+1}^{j+1} = u_{i+1}^{j} + \alpha\,\Delta t\,\Delta\dot{u}_{i+1}^{j+1}$
11. $enddo$
12. $enddo$
```

2 PARALLEL IMPLEMENTATION

2.1 Distributed memory

Concerning distributed memory systems, the main task of a parallel implementation is to achieve a well-balanced execution of the code minimizing redundancy and interprocess communication. In the SBS approach used in this work, non-overlapping partitions are employed together with the CSRC format (F.L.B. Ribeiro & I.A. Ferreira, 2007) and (G. O. Ainsworth Jr., F. L. B. Ribeiro, & C. Magluta, 2011). Denoting the original mesh by *M* and being *V* its set of nodes, after partitioning, each mesh partition M^i will have a set of nodes $V^i = \{V^{i,1}, V^{i,2}, V^{i,3}\}$. The subset $V^{i,1}$ contains nodes belonging to the interior or to the external boundary of the partition, subset $V^{i,2}$ corresponds to nodes belonging to the partition but that are placed at a common boundary between two or more partitions, and subset $V^{i,3}$is the set of internal boundary nodes that belong to another partition. Note that with these definitions and considering *n* partitions, the set *V* of nodes of the original mesh is given by

$$V = \bigcup_{i=0}^{n-1} \left\{ V^{i,1}, V^{i,2} \right\} \tag{1}$$

The following system of equations is assigned to each mesh partitionM^i:

$$A^i x^i = b^i \tag{2}$$

where,

$$A^i = \begin{bmatrix} A^i_{11} & A^i_{12} & A^i_{13} \\ A^i_{21} & A^i_{22} & A^i_{23} \\ A^i_{31} & A^i_{32} & A^i_{33} \end{bmatrix}; \quad b^i = \begin{bmatrix} b^i_1 \\ b^i_2 \\ b^i_3 \end{bmatrix}; x^i = \begin{bmatrix} x^i_1 \\ x^i_2 \\ x^i_3 \end{bmatrix} \tag{3}$$

Coefficients in A^i_{11}, A^i_{12}, A^i_{13}, A^i_{21} and A^i_{31} can be globally computed, while the remaining sub matrices contain partial contributions from M^i. Similarly, the right-hand-side term b^i_1 is global and b^i_2 and b^i_3 are partially computed.

Interprocess communication is performed whenever needed to complete global tasks. The two main tasks performed by Algorithm 1 are the element calculations and the assembly of the global arrays in step 7 and the iterative solution of a linear system in step 8. In the reminder of the algorithm there are only operations of the type:

$$a = u \, . \, p \qquad (dot \;\; product) \tag{4}$$

$$u = u + \alpha p \quad (vector \;\; update) \tag{5}$$

In addition to operations (4) and (5), the PCG solver involves *matvec* products:

$$p = Au \qquad (matvec) \tag{6}$$

A typical *matvec* operation $p^i = A^i u^i$ takes the form

$$\begin{bmatrix} p^i_1 \\ p^i_2 \\ p^i_3 \end{bmatrix} = \begin{bmatrix} A^i_{11} & A^i_{12} & A^i_{13} \\ A^i_{21} & A^i_{22} & A^i_{23} \\ A^i_{31} & A^i_{32} & A^i_{33} \end{bmatrix} \begin{bmatrix} u^i_1 \\ u^i_2 \\ u^i_3 \end{bmatrix} \tag{7}$$

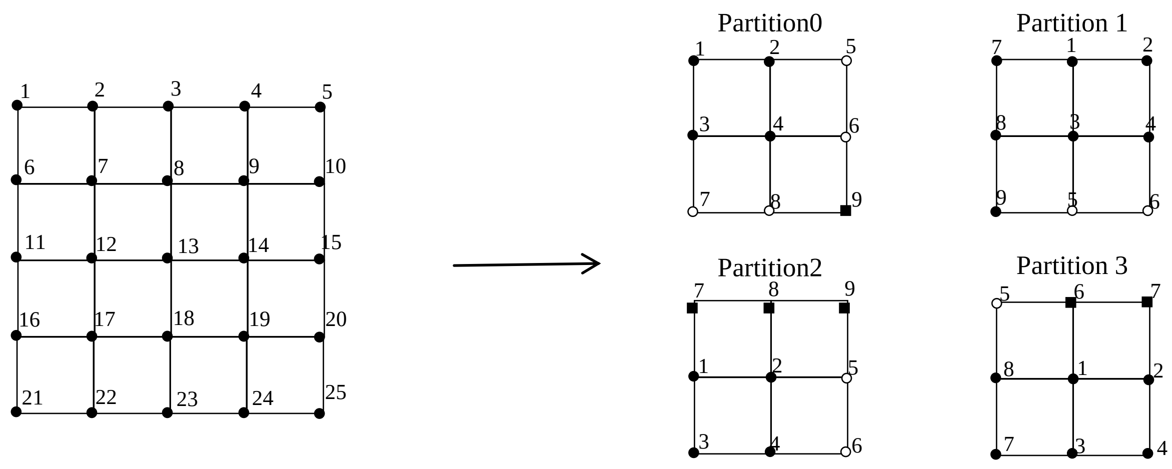
Algorithms to compute these products are described in (F.L.B. Ribeiro & I.A. Ferreira, 2007). Before the operation begins the vector u^i must be global and the resulting terms in p^i_2 and p^i_3 are only partial. This result will be the argument for the subsequent *matvec* operation, so it follows that these coefficients must be communicated and assembled in such a way that all partitions have their own assembled copies of p^i_2 and p^i_3. Therefore, vector update operations must be carried out over the whole set of equations:

$$\begin{bmatrix} u_1^i \\ u_2^i \\ u_3^i \end{bmatrix} = \begin{bmatrix} u_1^i \\ u_2^i \\ u_3^i \end{bmatrix} + \alpha \begin{bmatrix} p_1^i \\ p_2^i \\ p_3^i \end{bmatrix} \tag{8}$$

and scalar products are performed according to

$$a^i = u_1^i . p_1^i + u_2^i . p_2^i \tag{9}$$

Communication for dot products can be straightforward implemented using the collective routine *MPI_Allreduce* to gather and sum the partial contributions in each node. Vector updates do not require communication, as all vectors involved are global at this stage. The weight of communication relies operations. After each *matvec* operation, each partition sends p_2^i and p_3^i to the neighbouring partitions, where they must be received. For this we assign to each partition a map containing all the neighbouring nodes listed by each neighbouring partition, as described inFigure 1. From these maps, we build local send/receive buffers to perform the necessary communication using the routines *MPI_Isend* and *MPI_Irecv*.



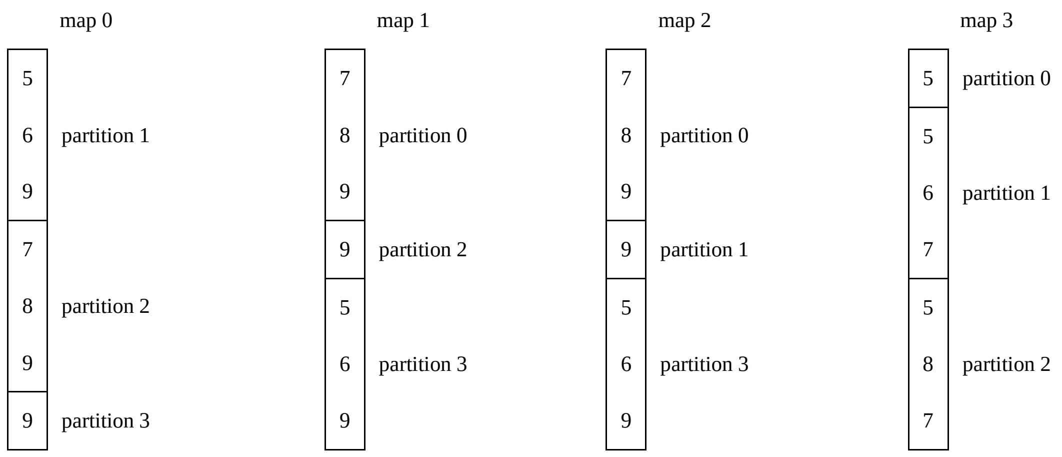


Figure 1– Example of map partition and communication vectors (map 0, 1,2 and 3).

2.2 Shared memory

In this work, shared memory programming is implemented with the use of OpenMP standard. As in the case of distributed memory, shared memory programming must focus on the two main steps of Algorithm 1, i.e., the loop on elements and the solver. The loop on elements is parallelized with local arrays being kept as private and global arrays are treated as shared variables. In this phase, the mesh must be colored grouping non-connected elements, in order to avoid memory concurrency conflicts. Each color defines an independent set of elements whose coefficients can be added to the global matrix concurrently. In the PCG solver, a single parallel region is used. Vector updates are parallelized in the standard way, with the directive *omp do*. Dot products are parallelized with the *omp do reduction* directive. For matvec operations the pure CSR format can be straightforward parallelized using the *omp do* directive. However, the CSRC format is much more efficient for symmetric matrices but it is more difficult to parallelize as it may lead to the occurrence of race condition. To resolve this issue we used a buffer that stores all partial results by thread. At the end of the algorithm, results are gathered from this buffer into vector p (Eq. 6). This procedure is described in Algorithm 2.

Algorithm 2 - MatvecCSRC(symmetric OpenMP version).

MatvecCSRC(symmetric OpenMP version).

Buffer initialization:

```
1. !$         thread_id = omp_get_thread_num() + 1
2 .  Do i = 1,  num_threads
3 .           inc = (i-1)*neq
4 .!$ omp do
5.            Do j = thread_heigth(i) + inc, thread_begin(i) + inc – 1
6.                     thread_y(j) = 0.0
7.            EndDo
8.!$omp end do
9.   EndDo
```

Matvec in CSRC:

```
10.  inc = (thread_id – 1 )*neq
11.!$omp barrier
12.  Do i=thread_begin(thread_id),  thread_end(thread_id)
13.           y(i) = 0.0
14.           xi   = x(i)
15.           t    = ad(i)*xi
16 .          Do k =ia(i), ia(i+1) -1
17.                    jak  = ja(k)
18.                    s    = al(k)
19.                    t    = t + s*x(jak)
20.                    jak = jak + inc
21.                    thread_y(jak) = thread_y(jak) + s*xi
22.           EndDo
23. .!$omp barrier
```

Gathered from the buffer to vector p:

```
24.  Do i = 1, num_threads
25.           inc = (i-1)*neq
26. .!$ omp do
27.           Do j = thread_heigth(i),  thread_end(i)
28.                    y(i) = y(i) + thread_y(j+inc)
29.           EndDo
30.!$ omp end do
31.  EndDo
```

3 NUMERICAL EXAMPLES

This section presents the results obtained in two computational platforms. Platform 1 consists of a 10 nodes cluster, each node with two Xeon E5620 processors and 16 GB of RAM. Communication is performed via LAN provided by a 2x20 Gb/s infiniBand system.

Platform 2 is comprised by a single node, consisting of two Xeon E5-5420 processors with 64 GB of RAM. The characteristics of the two platforms can be seen in Table 1. In platform 2, the OS native package *numactl* controls NUMA policy for shared memory. For this package, the parameter *interleave* is set to *interleave=all*.

Table 1–Hardware platforms characteristics

	Platform 1	*Platform 2*
processor	E5620	E5-2640
codename	Nehalem EP/Beckton	Sandy Bridge-EP
number of nodes	10	1
Socket per node	2	2
cores/threads	4/8	6/12
cache size (L1)	4x32KB	4x32KB
cache size (L2)	4x256KB	4x256KB
cache size (L3)	12 MB	15 MB
clock	2.4 Ghz	2.5 Ghz
QPI	5.86 GT/s	7.2 GT/s
Memory	16 G RAM (1.067 Mhz)	64 G RAM (1.333 Mhz)
Motherboard	S5500BC	S2600CO
LAN	InfiniBand-Qlogic (QLE7240)	-

Table 2- Software platforms characteristics

	Platform 1	*Platform 2*
OS	CentOS–5.4	CentOS–6.4
Compiler	ifort 12.0.4	ifort 12.0.4
Flags	-O3 –static-intel -ipo	-O3 –static-intel -ipo
MPI packege	MVAPICH2-1.8.1	MVAPICH2-1.8.1

3.1 Three-dimensional elasticity problem

This example shows the analysis of a solid structure in the form a unit cube, with all displacements restricted at the base (plane z = 0) and subjected to a uniform vertical load (q_z = -0.16) distributed at the top surface (plane z = 1). The elasticity modulus E is equal to 1 and ν = 0.3. Three meshes were generated, according to Table 3. The results in Figure 2-Figure 5 were obtained in platform 1. Figure 1shows for the three meshes the relation (internal boundary equations / total number) of equations, which gives an idea of the amount of communication to be performed. Figure 3shows the speedups for the MPI implementation using one core per node which is, strictly speaking, a pure case of distributed memory architecture. Figure 4- Figure 6show the speedups for distributed/shared memory using MPI for both architectures, for meshes 1, 2 and 3, respectively. Figure 7-Figure 9 - Three-dimensional elasticity problem for mesh 3 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x MPI in shared memory show the speedups for shared memory using platform 2.

Results show that the MPI implementation considering a pure distributed memory system achieved speedup values near the ideal curve for all meshes (Figure 3a). Higher speedup values in this stage are a consequence of the parallelization technique used, that avoid the need of communication. At the solver phase, values are below the ideal curve (Figure 3b), but can still be considered as good results, with speedups of 7.91, 8.14 and 8.98 respectively for meshes 1, 2 and 3, for 10 cores. The small difference between the values for each mesh shows the good efficiency of the designed communication scheme(Figure 1).

The analysis of the results obtained in platform 1 using the MPI implementation for a hybrid distributed/shared memory system shows that at the element phase the behavior is the same as the one observed for pure distributed memory systems (Figure 4a, Figure 5a and Figure 6a). Considering the solver stage, the speedup values decreases the number of cores per node increases (Figure 4b, Figure 5b and Figure 6b). This difference between the two phases occurs because at the solver phase memory access is a major issue.

Comparison between MPI and OpenMP in platform 2 present close speedups for 6 or less cores at the element stage. Above 6 cores, the MPI appeared to be more effective than the OpenMP implementation (Figure 7a, Figure 8a and Figure 9a). At the solver phase, the OpenMP was slightly superior for 6 cores in all meshes and for 8 cores for meshes 2 and 3. For 12 cores, the MPI programming was more efficient with speedups of 7.18, 7.24 e 7.35 against 4.56, 5.58 and 5.67 for OpenMP (Figure 7b, Figure 8b and Figure 9b).

Table 3 - Mesh characteristics for the 3D elasticity problem.

	hexahedrons	Nodes	Equations
mesh 1	64,000	68,921	201,720
mesh 2	512,000	531,441	1,574,640
mesh 3	4,096,000	4,173,281	12,442,080

Table 4 – maximum speedup in MPI Platform 1 for mesh 1

Mesh 1			
	Speedup	Number of cores per node	Number of cores
Element	79.67	8	80
Solver	30.25	8	48
Total	37.13	8	48

Table 5 - maximum speedup in MPI Platform 1 for mesh 2

Mesh 2			
	Speedup	Number of cores per node	Number of cores
Element	80.10	8	80
Solver	26.94	6	60
Total	33.98	8	80

Table 6 - maximum speedup in MPI Platform 1 for mesh 3

Mesh 3			
	Speedup	Number of cores per node	Number of cores
Element	71.16	8	80
Solver	29.53	8	80
Total	33.46	8	80

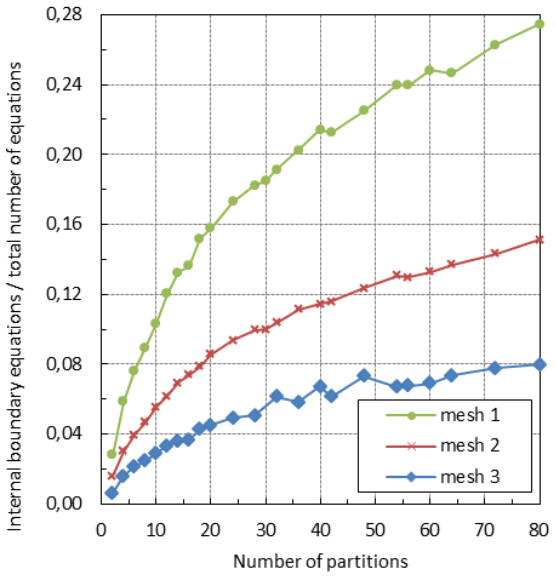


Figure 2- Relation internal boundary equations / total number of equations for the meshes 1, 2 and 3.

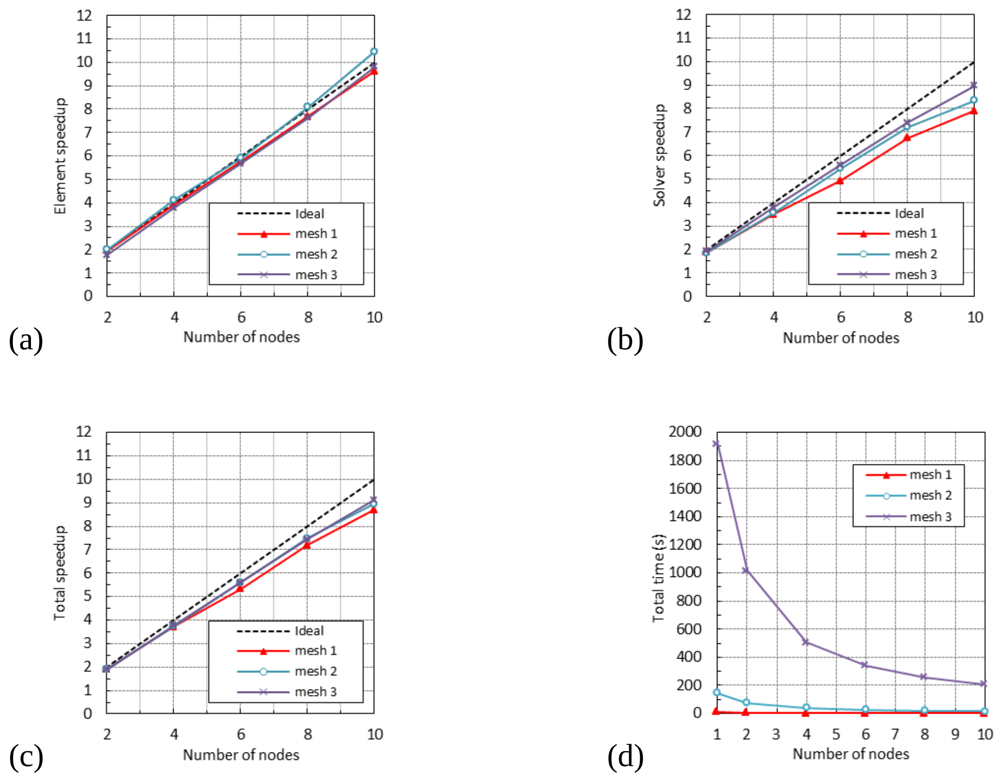
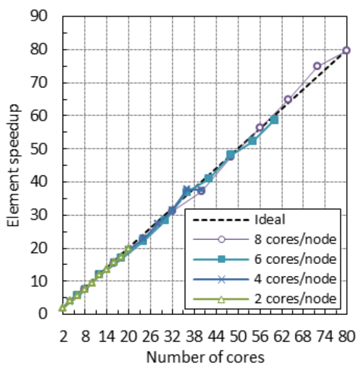
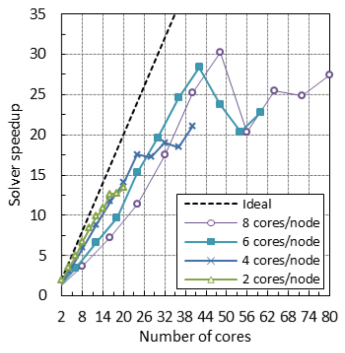


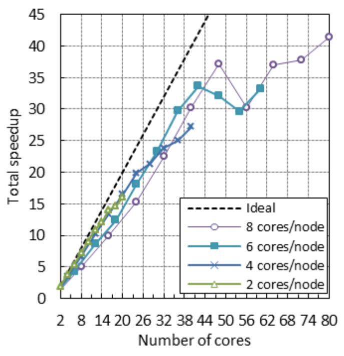
Figure 3 - Three-dimensional elasticity problems (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed memory.



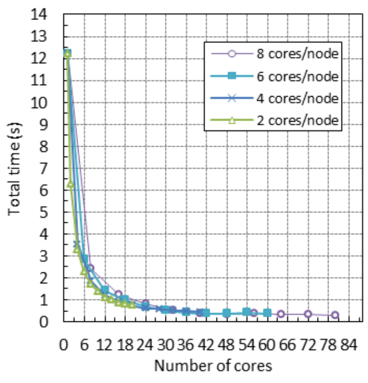
(a)



(b)

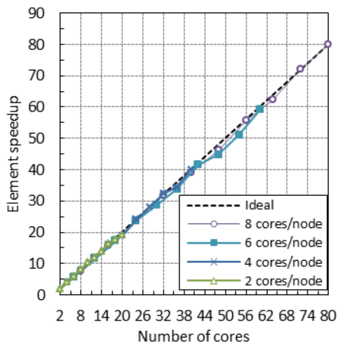


(c)

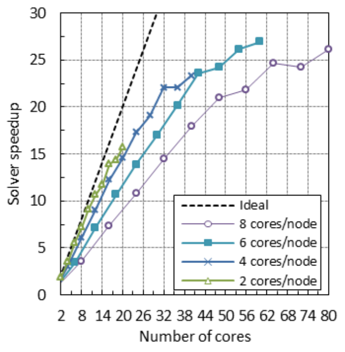


(d)

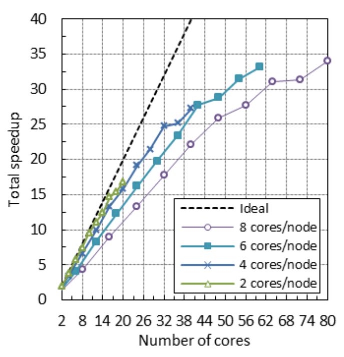
Figure 4 - Three-dimensional elasticity problem for mesh 1 (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed/shared memory.



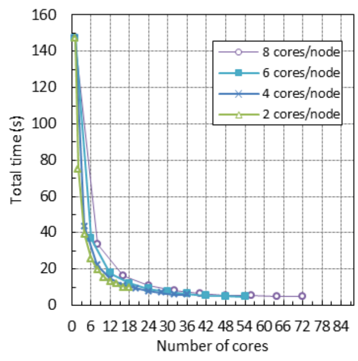
(a)



(b)

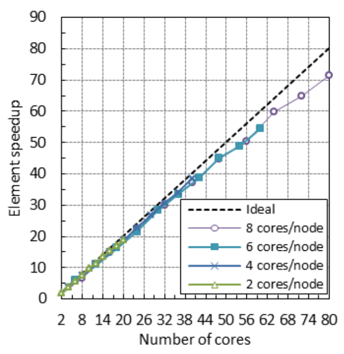


(c)

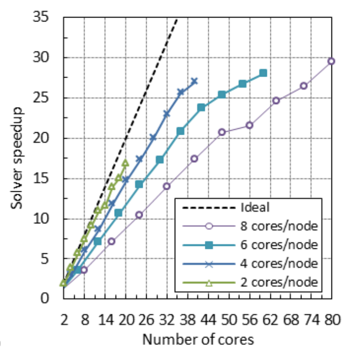


(d)

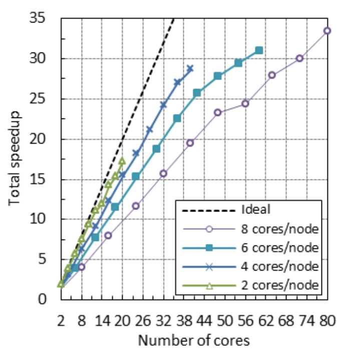
Figure 5 - Three-dimensional elasticity problem for mesh 2 (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed/shared memory.



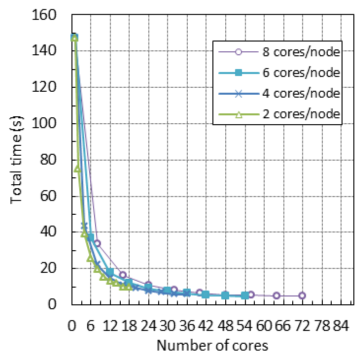
(a)



(b)

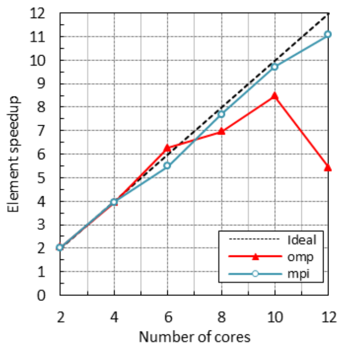


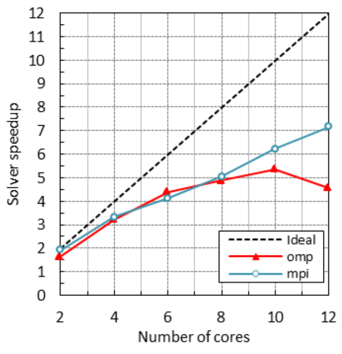
(c)



(d)

Figure 6 Three-dimensional elasticity problem for mesh 3 (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed/shared memory.


a)


b)

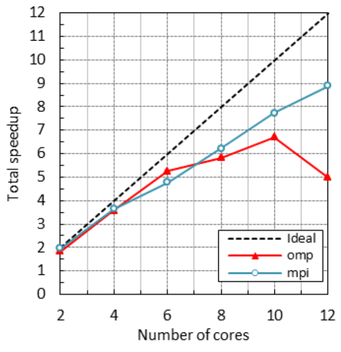
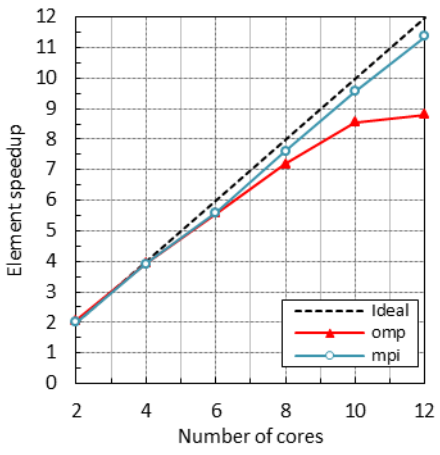
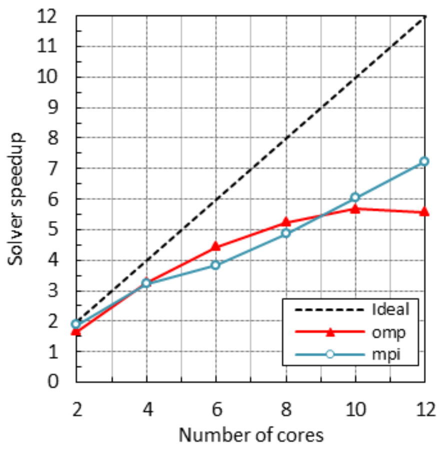
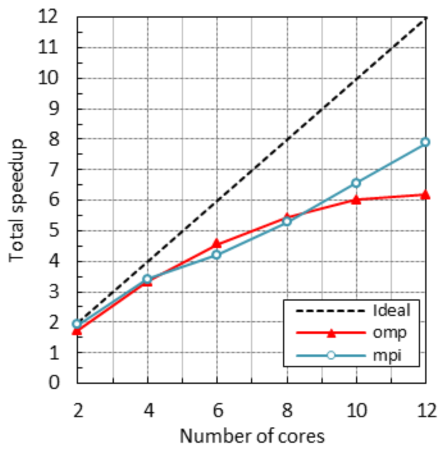

c)

Figure 7- Three-dimensional elasticity problem for mesh 1 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x MPI in shared memory

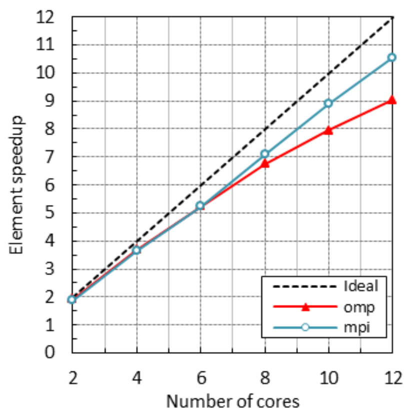

a)


b)

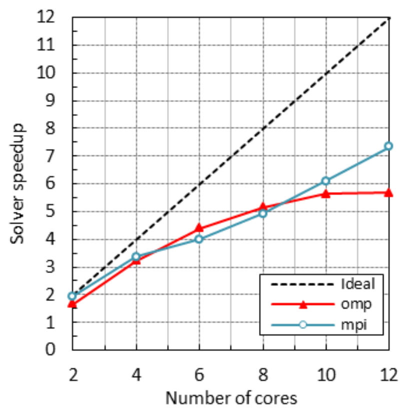


c)

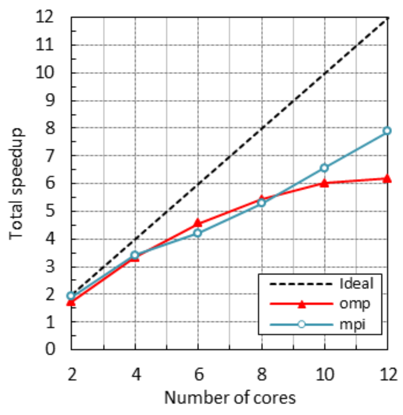
Figure 8 - Three-dimensional elasticity problem for mesh 2 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x MPI in sharedmemory



a)



b)



c)

Figure 9 - Three-dimensional elasticity problem for mesh 3 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x MPI in shared memory

3.2 Three-Dimensional diffusion problem

The second test case is a three-dimensional diffusion problem for a cube of unit length. Point sources (Q = ± 50) are applied at two opposite corners forming an internal diagonal and the solution ϕ is set to zero at all other remaining corners. The diffusion coefficient *k* is equal to 100.The characteristics of the three meshes are shown in Table 7. Three meshes were generated, according to Table 1. The results in Figures 11-14 were obtained in platform 1. Figure 10shows for the three meshes the relation (internal boundary) / (total number of equations), which gives an idea of the amount of communication to be performed. Figure 11shows the speedups for the MPI implementation using one core per node which is, strictly speaking, a pure case of distributed memory architecture. Figures 12, 13 and 14show the speedups for distributed/shared memory using MPI for both architectures, for meshes 1, 2 and 3, respectively. Figures 15, 16 and 17show the speedups for shared memory using platform 2.

Table 7- Mesh characteristics for the 3D diffusion problem.

	hexahedrons	nodes	equations
mesh 1	64,000	68,921	201,720
mesh 2	512,000	531,441	1,574,640
mesh 3	4,096,000	4,173,281	12,442,080

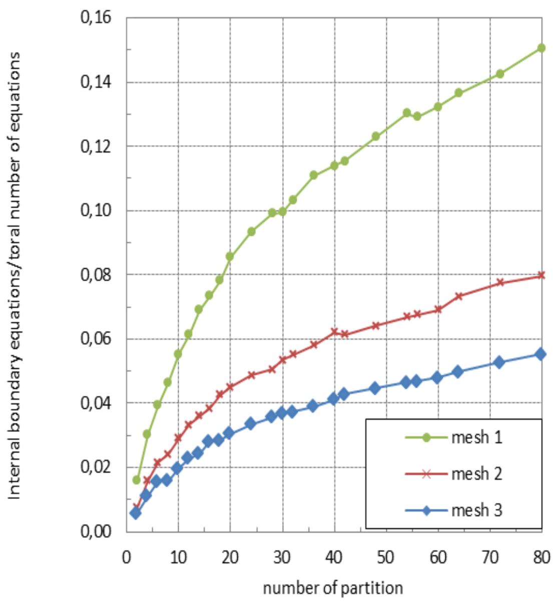


Figure 10 - Relation internal boundary equations / total number of equations for the meshes 1 , 2 and 3.

Figure 11a shows speedups above the ideal curve for all meshes for both element and solver phases. The difference between results at solver phase for elasticity and thermal examples can be explained by the smaller (internal boundary)/(total number of equations) rate for the thermal example, meaning lower communication costs (Figure 10).

Distributed/shared memory tests shows speedups slightly above the ideal independent of the number of cores per node used (Figure 12a, Figure 13a, Figure 14a). At the solver stage, the behavior is similar to the elasticity example (Figure 12b, Figure 13b, Figure 14b). Higher speedup values were obtained using more than one core per node(Table 8-Table 10).

Tests using both MPI and OpenMP implementations demonstrate a better result for the MPI in the element phase (Figure 15a, Figure 16a and Figure 17a) and for the solver phase using 12 cores per node, with speedups of 6.66, 6.81 and 6.62 against 3.98, 5.10 and 4.91 for the OpenMP (Figure 15b, Figure 16b and Figure 17b). The OpenMP has shown to be more efficient at the solver phase using 6 cores per node in all meshes and 8 cores per node in mesh 1.

Table 8 - maximum speedup in MPI Platform 1 for mesh 1

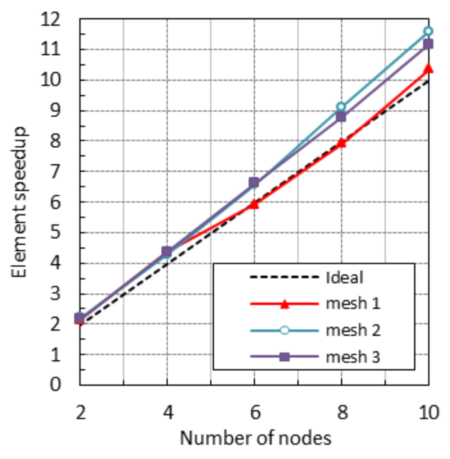
	Mesh 1		
	Speedup	Number of cores per node	Number of cores
Element	85.43	8	80
Solver	45.53	8	64
Total	50.68	8	64

Table 9 - maximum speedup in MPI Platform 1 for mesh 2

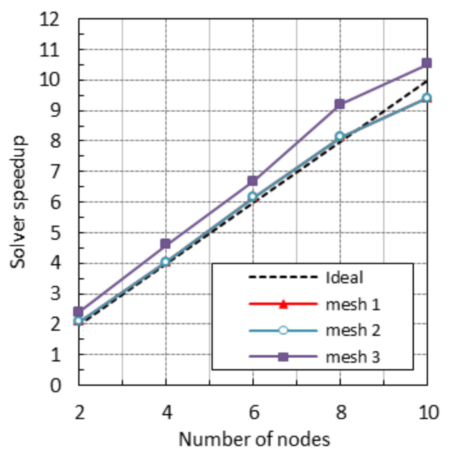
	Mesh 2		
	Speedup	Number of cores per node	Number of cores
Element	81.28	8	80
Solver	30.62	8	80
Total	33.29	8	80

Table 10 - maximum speedup in MPI Platform 1 for mesh 3

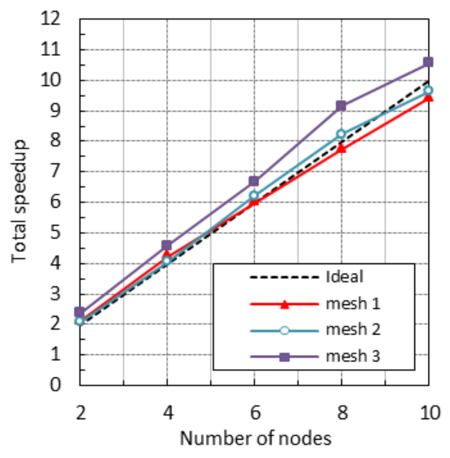
	Mesh 3		
	Speedup	Number of cores per node	Number of cores
Element	83.62	8	80
Solver	33.45	6	60
Total	34.60	6	60



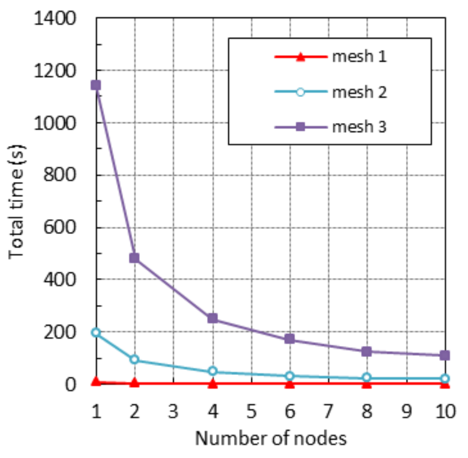
a)



b)

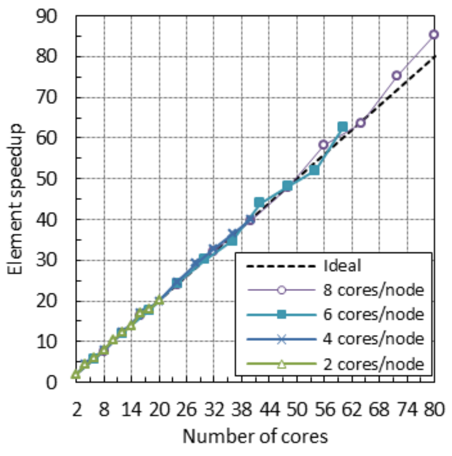


c)

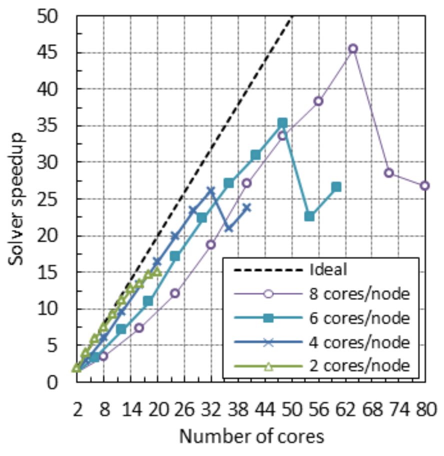


d)

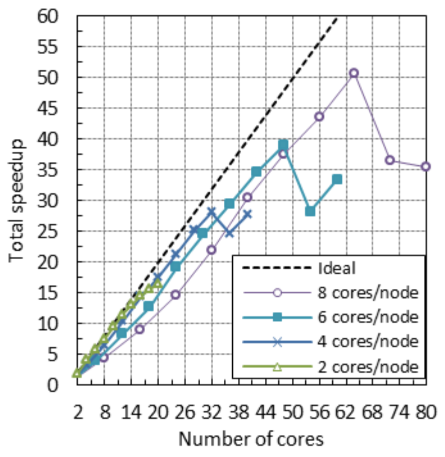
Figure 11 - Three-dimensional diffusion problems (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed memory.



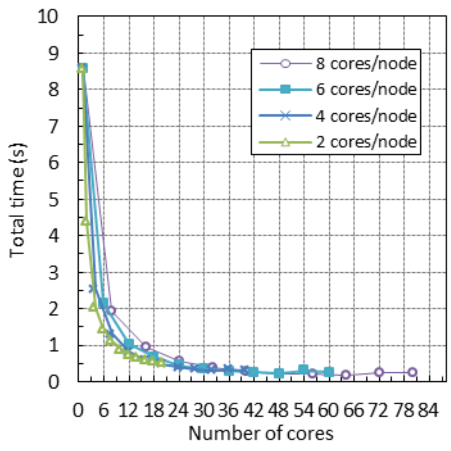
a)



b)

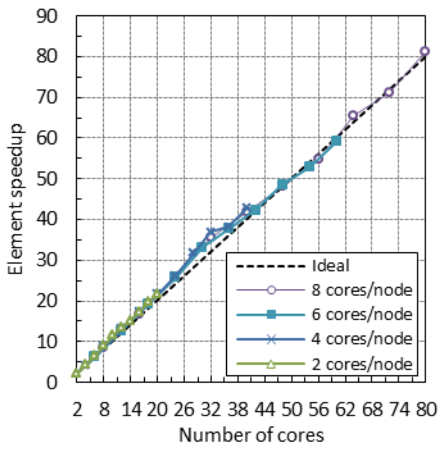


c)

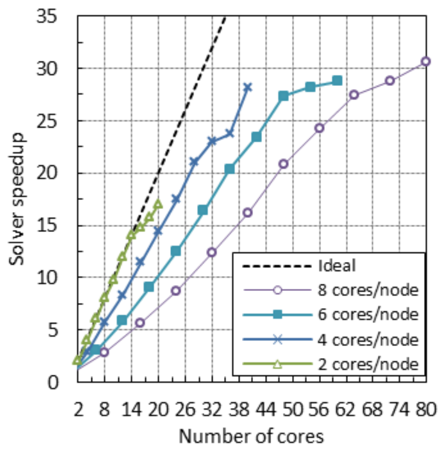


d)

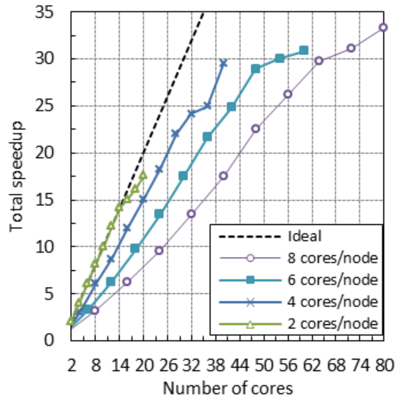
Figure 12 Three- dimensional diffusion problem for mesh 1 (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed/shared memory.



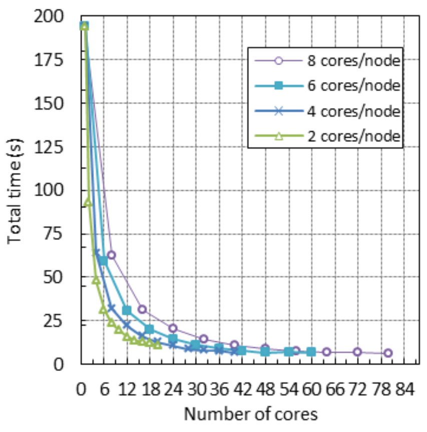
a)



b)

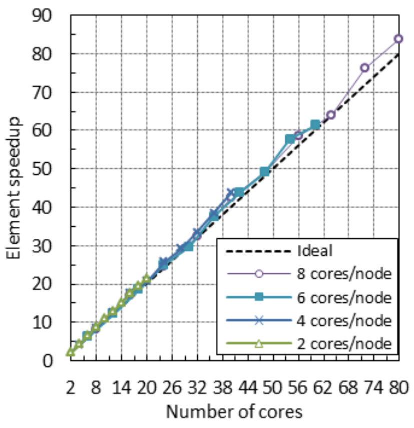


c)

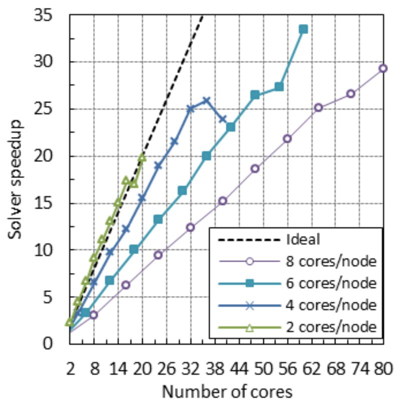


d)

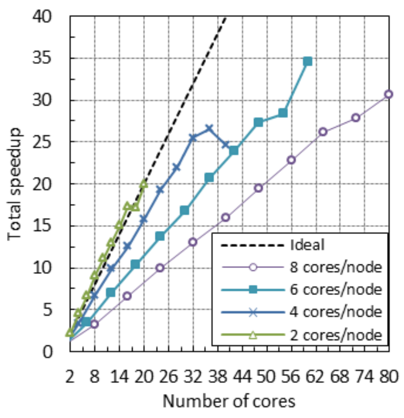
Figure 13 - Three- dimensional diffusion problem for mesh 2 (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed/shared memory.



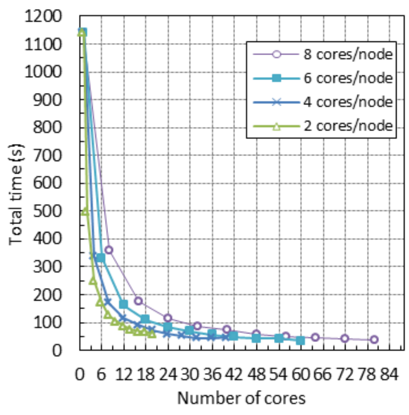
a)



b)

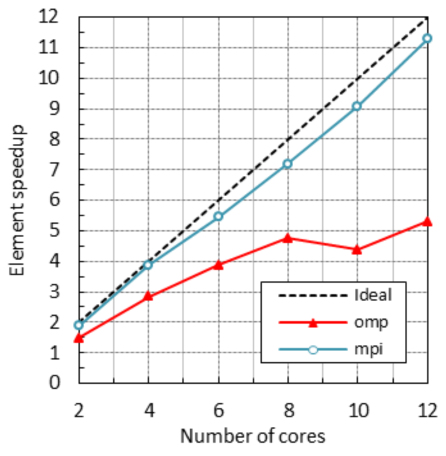


c)

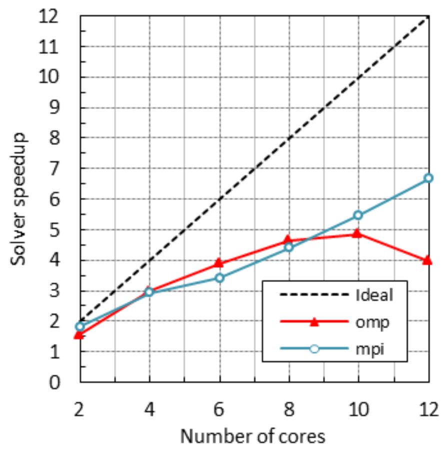


d)

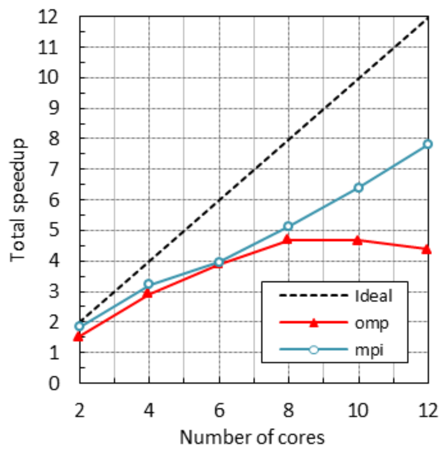
Figure 14- Three- dimensional diffusion problem for mesh 3 (Platform 1): a) speedups for elements calculations, b) solver speedups, c) total speedups and d) total times in distributed/shared memory.



a)

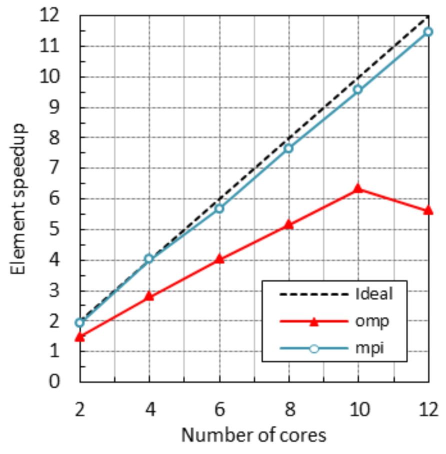


b)

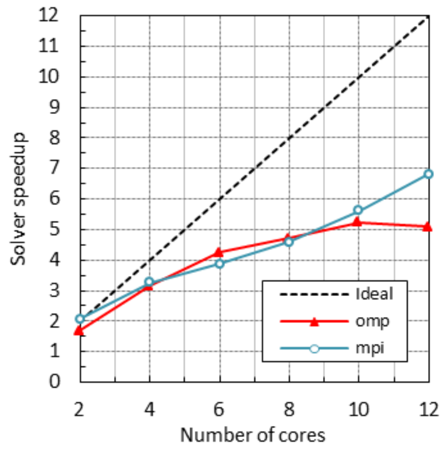


c)

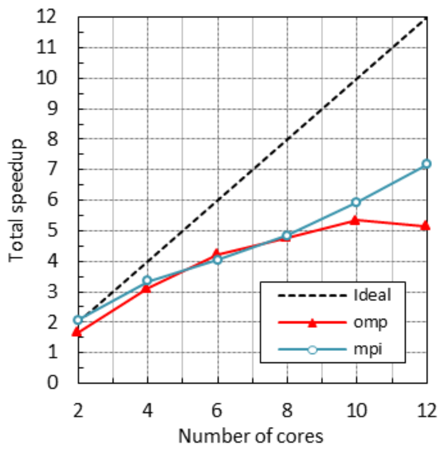
Figure 15 - Three-dimensional diffusion problem for mesh 1 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x Mpi in shared memory.



a)

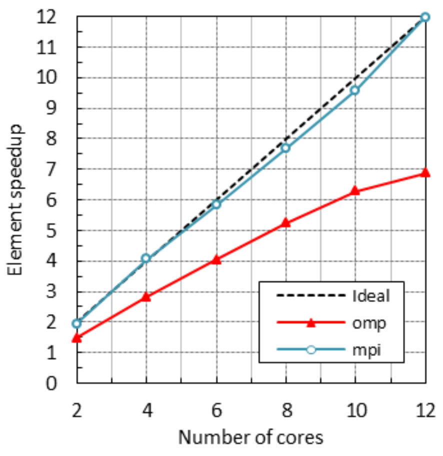


b)

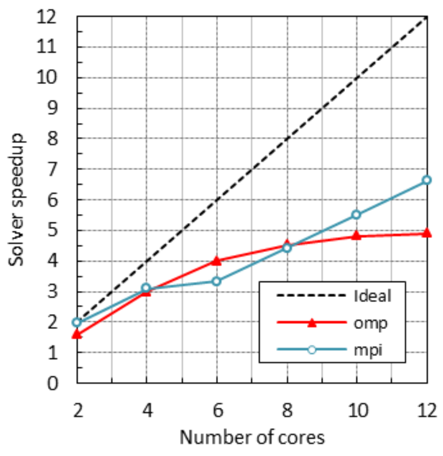


c)

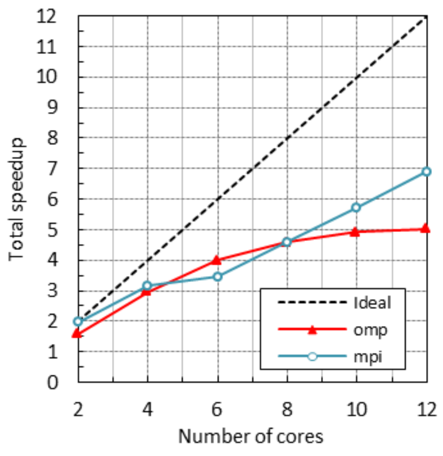
Figure 16 - Three-dimensional diffusion problem for mesh 2 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x Mpi in shared memory.



a)



b)



c)

Figure 17 - Three-dimensional diffusion problem for mesh 3 (Platform 2): a) speedups for elements calculations, b) solver speedups and c) total speedups for OpenMP x Mpi in shared memory.

4 CONCLUSIONS

In this paper we presented a parallel implementation of the finite element method designed for hybrid parallel machines. Three-dimensional elasticity and heat transfer problems were tested in two platforms: platform 1 was used to test the efficiency of the MPI implementation for distributed/shared memory, while platform 2 was used to compare the MPI against the OpenMP implementations in a shared memory environment. To measure the efficiency of the MPI implementation in a pure distributed memory system, tests were run with 1 core per node. As shown in the results, we may conclude that the present implementation has proved to be very efficient, with speedups near the ideal curve in both, element and solver phases, for all three meshes. Also in platform 1, we tested the same MPI implementation to obtain shared memory parallelization, using all cores available in each node. For the element phase, as it was expected, nearly ideal speedups were obtained, as there is no communication in this phase. For the solver, however, the efficiency ranged from 0.34 to 0.38 for 80 cores and total efficiency ranged from 0.42 to 0.51, which can still be considered as good results. In platform 2,the MPI and Open MPI implementations presented similar results for 6 cores or less. For more than 6 cores, clearly the MPI behaved better than the OpenMP, meaning that this type of implementation can be successfully used in distributed/shared memory platforms.

REFERENCE

F.L.B. Ribeiro, & I.A. Ferreira. (2007). Parallel implementation of the finite elment method using compressed data structures. *Computational Mechanics* , 31-48.

G. Mahinthakumar, & F. Saied. (2002). A Hybrid Mpi-OpenMP implementation of an implicit finire-element code on parallel architectures. *The International Journal of High Performance Computing Apllications* , 371-393.

G. O. Ainsworth Jr., F. L. B. Ribeiro, & C. Magluta. (2011). A parallel subdomain by implementation of the implicitly restarted Arnoldi/Lanczos method. *computational mechanics* , 563-577.

Hughes, T. J., Shakib, F., & Johan, Z. (1989). A multi-element group preconditioned gmres algorithm for nonsymmetric systems arinsing in finite element analysis. *Computer methods in applied mechanics and engineering* , 415-456.

Kourtis, K., Goumas, G., & Koziris, N. (2008). Improving the performance of Multithreaded Sparse Matrix-Vector Multiplication using Index and Value Compression. *International Conference on Parallel Processing*, (pp. 511-519).

Krotkiewski, M., & Dabrowski, M. (2010). Parallel symmetric sparse matrix-vector product on scalar multi-core CPUS. *Parallel Computing* , 181-198.

Liu, S., Zhang, Y., Sun, X., & Qiu, R. (2009). Performance Evaluation of Multithreaded Sparse Matrix-Vector Multiplication using OpenMP. *11th IEEE internacional Conference on High Performance Computing and Communications*, (pp. 659-665).

Ribeiro, F. L., & Coutinho, A. L. (2005). Comparison between element, edge and compressed storage schemes for iterative solutions in finite elemen analyses. *Int J Numer Methods Eng* , 569-588.