PARALLEL BOUNDARY ELEMENTS USING OPENMP

Cunha, M.T.F., Coutinho, A.L.G.A., Telles, J.C.F.
Universidade Federal do Rio de Janeiro - COPPE / UFRJ
Caixa Postal 68.506, CEP 21.945-970, Rio de Janeiro, RJ, Brasil.
{mcunha@pec.coppe.ufrj.br, alvaro@coc.ufrj.br, telles@coc.ufrj.br}

Abstract—
This work presents our efforts towards single node optimization and parallelization of an existing Boundary Element Method (BEM) code using OpenMP. Basic techniques of High Performance Computing (HPC) are employed to enhance serial code performance and to identify code parts to be parallelized. Numerical experiments on a SGI Origin 2000 on large problems show the effectiveness of the proposed approach.

Keywords— **Boundary Elements, OpenMP**

I. INTRODUCTION

The Boundary Element Method is a numerical method to solve scientific and engineering problems where only the boundary needs to be discretized. This feature reduces drastically the number of grid points necessary to model a given problem. However, in constrast with finite differences, finite volume and finite elements, the other common discretization methods, the resulting BEM systems of equations involve dense matrices. In this work an existing BEM Fortran implementation, described in detail by Brebbia and Dominguez [BRE 92], is reviewed and rewritten to achieve high performance on cc-NUMA architectures. This effort is the initial step to develop a new generation of efficient BEM codes to be used in many important applications, from acoustics to frature mechanics. The program paradigm to be used is OpenMP, a standard and portable Application Programming Interface (API) for writing shared memory parallel programs [OMP 01]. However, we first conform the code to the ANSI F90 specification. The remainder of this work is as follows. In next section we briefly review the BEM theory. In the following Section we describe the selected application, giving guidelines for the optimization process. Section 3 covers the single node optimization aspects, while in Section 4 we present the parallel code and a performance analysis on a SGI Origin 2000. Finally the paper ends with a summary of our main conclusions.

II. OUTLINE OF BEM THEORY

The Boundary Element Method (BEM) is a technique for the numerical solution of partial differential equations with proper boundary and initial conditions. By using a weighted residual formulation, Green's third identity, Betti's reciprocal theorem or some other procedure, an equivalent integral equation can be obtained and then converted to a form that involves only surface integrals, i.e., over the boundary. The boundary is then divided into elements and the integrals over the boundary are simply the sum of the integration over each element, resulting in a dense and non-symmetric system of linear equations.

The discretization process involves selecting nodes along the boundary where the unknown values are considered. An interpolation function relates one or more nodes on the element to the potential and fluxes anywhere on the element. The simplest case places a node in the center of each element and defines an interpolation function that is a constant across the entire element. Once the boundary potentials have been obtained, the interior points can be computed directly from the basic integral equation describing the system. [BRE 84] [TEL 83]

A. Differential Equation

Potential problems are governed by Laplace's equation :

$$\nabla^2 = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} = 0 \qquad \text{in } \Omega \tag{1}$$

and boundary conditions :

$$u = \bar{u} \quad \text{on } \Gamma_1 \quad \text{and}$$
$$q = \bar{q} = \partial u / \partial \eta \quad \text{on } \Gamma_2 \tag{2}$$

where u is the potential function, $\bar{u}$ and $\bar{q}$ are prescribed values, $\Gamma = \Gamma_1 + \Gamma_2$ is the boundary of domain Ω and η is the unit outward normal to surface Γ.

B. Integral Equation

The integral equation for potential problems reads :

$$u(\xi) + \int_\Gamma u(x)\, q^*(\xi, x)\, d\Gamma(x) = \int_\Gamma q(x)\, u^*(\xi, x)\, d\Gamma(x) \tag{3}$$

The weighting function u^* satisfies Laplace's equation and represents the field generated by a concentrated unit charge acting at the source point ξ.

The function q^* is the outward normal derivative of u^* along the boundary Γ with respect to the field point x, i.e.:

$$q^*(\xi, x) = \partial u^*(\xi, x) / \partial \eta(x) \tag{4}$$

In order to obtain an integral equation involving only the variables on the boundary, one can take the limit of the integral equation as the point ξ tends to the boundary Γ. This limit must take into account the discontinuity of the second integral.

The resulting equation for potential problems is :

$$c(\xi)\ u(\xi) + \int_\Gamma u(x)\ q^*(\xi,x)\ d\Gamma(x) = \int_\Gamma q(x)\ u^*(\xi,x)\ d\Gamma(x) \quad (5)$$

in which the second integral is computed in the Cauchy principal value sense.

The coefficient c is a function of the internal angle the boundary Γ makes at point ξ.

C. Discretization

The integral statement for potential problems can be written as follows :

$$c_i\ u_i + \int_\Gamma u\ q^*\ d\Gamma = \int_\Gamma q\ u^*\ d\Gamma \quad (6)$$

Notice that the source point ξ was taken as the boundary node i on which the unit charge is applied, i.e., $u(\xi) = u_i$.

After the discretization of the boundary into N elements, the integral equation can be written :

$$c_i u_i + \sum_{j=1}^{N} \int_{\Gamma_j} u q^* d\Gamma = \sum_{j=1}^{N} \int_{\Gamma_j} q u^* d\Gamma \quad (7)$$

This equation represents, in discrete form, the relationship between the node i at which the unit charge is applied and all the j elements on the boundary, including $i=j$.

For the constant element case the boundary is always smooth as the node is at the center of the element, hence the coefficient c_i is equal to ½ and the values of u and q are taken out of the integrals :

$$\frac{1}{2}u_i + \sum_{j=1}^{N} \int_{\Gamma_j} q^* d\Gamma\, u_j = \sum_{j=1}^{N} \int_{\Gamma_j} u^* d\Gamma\, q_j \quad (8)$$

The equation can thus be written as follows :

$$\frac{1}{2}u_i + \sum_{j=1}^{N} \hat{H}_{ij} u_j = \sum_{j=1}^{N} G_{ij} q_j \quad (9)$$

and can be further rewritten as :

$$\sum_{j=1}^{N} H_{ij} u_j = \sum_{j=1}^{N} G_{ij} q_j \quad (10)$$

where

$$H_{ij} = \begin{cases} \hat{H}_{ij} \rightarrow i \neq j \\ \hat{H}_{ij} + \frac{1}{2} \rightarrow i = j \end{cases} \quad (11)$$

The equation can be written in matrix form as :

$$\mathbf{H\ U = G\ Q} \quad (12)$$

By applying the prescribed conditions, the equation can be reordered with all the unknowns on the left-hand side and a vector of known values on the right-hand side. This gives :

$$\mathbf{A\ Y = F} \quad (13)$$

where $\mathbf{Y}$ is the vector of unknowns u's and q's.

D. Internal Points

Once the system is solved, all the values on the boundary are known and the values of u and q at any interior point can be calculated using :

$$u_i = \int_\Gamma q\ u^*\ d\Gamma - \int_\Gamma u\ q^*\ d\Gamma \quad (14)$$

Internal fluxes can be derived from the potential equation, as follows :

$$q_i = \frac{\partial u}{\partial x} = \int_\Gamma q \frac{\partial u^*}{\partial x} d\Gamma - \int_\Gamma u \frac{\partial q^*}{\partial x} d\Gamma \quad (15)$$

The same discretization process can be applied for the integrals above.

E. Fundamental Solution

For a two dimensional isotropic domain, the fundamental solution is :

$$u^*(\xi,x) = \frac{1}{2\pi} \ln \frac{1}{r}$$

$$q^*(\xi,x) = \frac{1}{2\pi r} r_\eta \quad (16)$$

where r is the distance from the source point ξ to the field point x and $r_\eta = \partial r(\xi,x)/\partial \eta(x)$.

F. Numerical Implementation

For generality, the integrals are here calculated using Gauss quadrature for all elements, except for $i=j$, as follows :

$$\hat{H}_{ij} = \int_\Gamma q^* d\Gamma = \frac{l_j}{2} \sum_{k=1}^{K} q_k^* w_k$$

$$G_{ij} = \int_\Gamma u^* d\Gamma = \frac{l_j}{2} \sum_{k=1}^{K} u_k^* w_k \quad (17)$$

where l_j is the element length and w_k is the weight associated with the integration point k. The functions u^* and q^* are evaluated at the same point.

For the particular case of constant elements, integrals corresponding to the singular elements can be computed analytically :

$$\hat{H}_{ij} = \int_\Gamma q^* d\Gamma = \int_\Gamma \frac{\partial u^*}{\partial r} \frac{\partial r}{\partial \eta} d\Gamma \equiv 0$$

$$G_{ij} = \int_\Gamma u^* d\Gamma = \int_\Gamma \ln \frac{1}{r} d\Gamma = \frac{1}{\pi} r \left[\ln \frac{1}{r} + 1 \right] \quad (18)$$

The first integral is identically zero due to orthogonality between the normal and surface of the element.

III. The Application

The program reviewed here is a simple code for solving potential problems using constant elements. The main program defines some general variables, integer and real arrays, as shown :

```
program POCONBE
integer                          :: N,L
integer,parameter                :: NMAX=1000
integer,dimension(NMAX)          :: KODE
real                             :: d
real,dimension(NMAX+1)           :: X,Y
real,dimension(NMAX)             :: FI,DFI
real,dimension(NMAX,NMAX)        :: G,H
real,dimension(20)               :: CX,CY,POT,FLUX1,FLUX2
call INPUT
call GHMAT
call SLNPD(G,DFI,d,N,NMAX)
call INTER
call OUTPUT
end program POCONBE
```

In order to compute the coefficients of G and H matrices the GHMAT subroutine calls two additional subroutines, EXTIN and LOCIN. While EXTIN computes the off-diagonal coefficients of H and G, LOCIN calculates only the diagonal elements of G matrix. EXTIN subroutine is also called by INTER subroutine to compute potential and fluxes at internal points.

Profiler output shows execution times, cycles and instructions of the whole program, as well as of each function, separately :

```
2016306973 : Total number of instructions executed
1353289838 : Total computed cycles
     5.413 : Total computed execution time (secs.)
     0.672 : Average cycles / instruction
```

function	secs	excl.%	cycles	instructions	calls
SLNPD	2.854	52.7%	713411657	1461982206	1
EXTIN	1.631	30.1%	407840000	296960000	1004000
__logf	0.851	15.7%	212808208	215814816	4017000
GHMAT	0.065	1.2%	16263827	37971543	1

The results produced by program profiling shows that SLNPD routine takes the greatest portion of execution time and should be the first part of the code to be optimized. SLNPD is a standard routine, which solves a linear system of equation using pivoting. Results are stored in the same right hand size vector. In larger systems, the solver has a major importance and a good change in this part of the code brings the best results in the overall performance.

IV. Single Node Optimization

A. Exploiting Tuned Codes

The quickest and easiest way to improve performance of the program is to link it with tuned libraries. LAPACK, Linear Algebra Package, is a public domain library of subroutines for solving linear system of equations, linear least square problems, eigenvalue problems and singular value problems. It has been designed to be efficient on a wide range of modern high-performance computers. LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms, BLAS. Highly efficient machine-specific implementations of the BLAS are available for many high-performance computers. The BLAS enable LAPACK routines to achieve high performance with portable code. [AND 99]

Boundary Elements problems produce non-symmetric dense matrices and LAPACK's SGESV subroutine can be used to solve such systems of linear equations. Thus, the first modification in the code can be the substitution of SLNPD by SGESV.

B. Hand Tuning Techniques

The first part of the code to be hand-tuned is EXTIN, the subroutine that computes the off-diagonal coefficients of G and H matrices, using a 4-point Gauss integration formula. It also computes, using the same numerical integration formula, integrals and its derivatives required for the computation of potential and fluxes at internal points. The routine also performs a change to a dimensionless system of coordinates.

The main hand tuning techniques applied to the code in order to obtain better performance follows. [DOW 98] [FOS 94] [WAD 00] [SGI 98]

B.1 Arithmetic Optimizations

Strength Reduction :

Operations or expressions have time costs associated with them. Sometimes it is possible to replace a more expensive calculation with a cheaper one. *Strength reduction* reduces the computation costs of an operation while providing mathematically identical results. There are many opportunities for strength reductions. However, most compilers automatically perform these optimizations.

Data Type Conversions :

The type and precision of a data value determine the amount of storage required to contain the value and the way in which the value can be operated on. The selection of a type and precision to represent data can have significant effect on performance. Statements that contain runtime type conversions suffer a performance penalty each time the statement is executed. If the statement is located in a portion of the program where there is a lot of activity, the total penalty can be significant.

Common Subexpression Elimination :

Subexpressions are pieces of expressions. Some compiler may recognize repeated patterns in the code and replace all but one with a temporary variable. However, the ability of the compilers to recognize common subexpressions is limited, especially when there are multiple components, or their order is permuted.

B.2 Inlining

Inlining involves replacing a procedure reference with a copy of the code of the procedure.

Some of the benefits of inlining are :

- There is no procedure reference, so the overhead of referencing the procedure, such as saving and restoring registers are eliminated.
- When a procedure is referenced from a loop, certain optimizations are inhibited. When the reference is inlined, such optimizations are enabled.
- An inlined procedure may enable other optimization because relationships between the reference point and the referenced procedure are easier to determine.

One can inline procedures in the source code by hand or requesting that automatic inlining to be done by the compiler. The advantage of manual inlining is that the code is inlined regardless of the platform the program is compiled on.

There are two disadvantages of manual inlining :

- It can be very time-consuming and prone to error. The C++ language allows defining functions as inline within the source code, so this disadvantage does not apply to C++
- Inlining of a particular procedure reference may improve performance on some architectures while degrading performance on others.

Performance improvements resulting from inlining vary depending on the following :

- The size and number of the inlined procedure.
- The frequency with which inlined routines are called from a given location in source code.
- The number of different locations where each procedure is referenced.

B.3 Loop Optimizations

In nearly all high performance applications, loops are where the majority of the execution time is spent. Sometimes the compiler is clever enough to generate the faster versions of the loops and other times one has to do some rewriting of the loops to help the compiler. Some of the most productive loop optimizations are those that minimize cache misses. Other loop transformations include those that reduce the *loop overhead*, remove loads and stores from innermost loops and eliminate delays caused by data dependencies.

On some architetures, there is no overhead in the counter update or the conditional branch within an innermost loop. However, outer loops usually involve a delay in the counter update of a branch. Depending on the construction of the *loop nest*, one may have some flexibility in the ordering of the loops. At times, one can swap the outer and inner loops with great benefit.

Many optimizations performed on nested loops are meant to improve the memory access patterns. Often nested loops deal with multidimensional arrays. Computing in multidimensional arrays can lead to non-unit-stride memory access.

Stride minimization is important where the stride can be brought down to a value smaller than the number of elements of an array that can be stored in a single cache line. Stride minimization is also applicable to very large strides, where it fits in successive elements are in different *pages* of memory. Even if the stride cannot be brought down to a level where successive cache lines, one can improve performance by reducing the stride to the point where successive elements may be in the same page. This can reduce TLB misses and page faults.

Loop Interchange :

Loop interchange involves changing the nesting order in nested loops. This technique can improve performance in two ways :

- It can minimize the stride in which array elements are accessed on successive iterations of a loop.
- It can reduce loop overhead.

Branches Within Loops :

Numerical codes usually spend most of their time in loops and branches have great impact on its performance. Branches can force a strict order of a loop and prevent the compiler from optimizing the code effectively. Eliminating branches eliminates control dependency and allows the compiler to pipeline more arithmetic operations.

Sometimes, programmers place branches in loops to process events that could be handled outside or even ignored. One example is loop invariant conditionals, where results are the same regardless of what happens in each iteration, usually they can be processed outside of the loop.

For loop index dependent conditionals, the test is true for certain ranges of the loop index variables and changes with a predictable pattern. Thus, the if statement partitions the iterations into distinct sets : those for which it is true and those for which it is false. One can take advantage of the predictability of the test to restructure the loop into several loops, one for each different partition.

Loop Unrolling :

Loop unrolling involves replicating the functional components of a loop, while reducing the iteration count proportionately.

The major benefits of loop unrolling are :

- Data dependence delays can be reduced or eliminated
- Loads and stores may be eliminated in successive loop iterations
- Loop overhead may be reduced
- Larger basic blocks and more instructions between branches

For loops with very small iteration counts, it is possible to eliminate the loop entirely. This is the simplest form of loop unrolling. The advantage of unrolling an innermost loop is that there is no need to modify the layout of the array to accommodate the tuning. There may be an advantage to unrolling the outer loop to improve instructions scheduling and to reduce the number of loads and stores. Although this technique brings little performance improvement, it shows two internal loops as natural candidates for the parallel implementation of this code.

Array Elements in Loops

When making repeated use of an array element within a loop, one wants to be charged just once for loading it from memory. Compilers should recognize that each array element is being used a few times and that it only be load once, but not all can do it. Using scalar variables is an optimization that some compilers are not free to make.

Interestingly, while using scalar variables is useful for RISC and superscalar machines, it does not help code that runs on parallel hardware. A parallel compiler looks for opportunities to eliminate the scalars or, at least, to replace them with temporary vectors. If the code runs on a parallel machine from time to time, one must be careful about introducing scalar variables in loops. A dubious performance gain in one instance could be a real performance loss in another.

C. Experimental Results Summary

Applications are run on a SGI's Origin 2000 with 8 Gb memory and 16 R10000 250 MHz processors. The operational system is IRIX, version 6.5.10m and the compiler is MIPSpro, version 7.3.1.m. Profiler is SpeedShop. All codes were compiled with -n32 –mips4 -O3 compiling options. Ssrun was used with the –ideal option. [SGI 01] [SGI 02]

The effects of each technique in the overall performance of a small problem can be seen in Table 1. Once the overall performance is limited by the contribution of each procedure being tuned, the effect of each optimization technique is shown in Table 2.

Small systems can be greatly affected by cache size. Therefore, performance must be measured with a larger sample. With a larger set of data, the same application brings the results shown in Table III and the contribution of each procedure is presented in Table IV.

Tables show the percentage ratio of the execution time of the new version with respect to the execution time of the original version.

TABLE I

1000 ELEMENTS - PROGRAM PERFORMANCE

	Secs	%
Original version	5.413	
Substitution of SLNPD by SGESV	3.819	70.5%
Arithmetic optimizations	3.567	65.8%
Inlining EXTIN	3.454	63.8%
Loop interchange	2.137	39.4%
Conditional branch elimination	2.072	38.2%
Loop unrolling	2.063	38.1%

TABLE II

1000 ELEMENT S- SUBROUTINES PERFORMANCE

	Function	Secs	%
Original version	EXTIN	1.631	
Arithmetic optimizations	EXTIN	1.380	84.6%
Inlining EXTIN	GHMAT	1.324	81.1%
Loop interchange	GHMAT	0.443	27.1%
Conditional branch elimination	GHMAT	0.377	23.1%
Loop unrolling	GHMAT	0.368	22.5%

TABLE III

8000 ELEMENTS - PROGRAM PERFORMANCE

	Secs	%
Original version	626.625	
Substitution of SLNPD by SGESV	339.598	54.1%
Arithmetic optimizations	323.697	51.6%
Inlining EXTIN	317.289	50.6%
Loop interchange	232.467	37.0%
Conditional branch elimination	228.127	36.4%
Loop unrolling	227.554	36.1%

TABLE IV

8000 ELEMENTS - SUBROUTINES PERFORMANCE

	Function	Secs	%
Original version	EXTIN	103.771	
Arithmetic optimizations	EXTIN	87.870	84.6%
Inlining EXTIN	GHMAT	85.594	82.2%
Loop interchange	GHMAT	28.358	27.3%
Conditional branch elimination	GHMAT	24.004	23.1%
Loop unrolling	GHMAT	23.431	22.5%

V. PARALLEL PROCESSING

Because there is always a limit to the performance of a single CPU, computer programs have increased their performance by using multiple CPUs.

The Cray Scientific Library, SCSL, is an optimized library containing BLAS and LAPACK routines, many of them parallelized and linked into the program using a compiling option. Once the solver takes the most part of the execution time, a very simple and effective approach to implement a parallel version of the program is to use a parallel the solver, done with a simple compiler option.

The other parts of the code should be parallelized manually. We use OpenMP, which is a standard and portable Application Programming Interface (API) for writing shared memory parallel programs. [CHA 01]

In shared memory architectures all processors has direct and equal access to all the memory in the system. One advantage of the shared memory parallelization is that it can be done incrementally. That is, the user can identify the most time-consuming part of the code, usually loops, and parallelize just that. Another advantage of this parallelization style is that loops can be parallelized without being concerned about how the work is distributed across multiple processors and how data elements are accessed by each CPU.

OpenMP is a set of compiler directives that may be embedded within a program written with a standard programming language such as Fortran or C/C++. In Fortran these directives take form of source code comments identified by the $OMP prefix and are simply ignored by a non-OpenMP compiler. Thus, the same source code can be used to compile a serial or parallel version of the application. In addition to directives, OpenMP also includes a set of runtime library routines and environment variables that are used to examine and modify the execution parameters. Introduction of OpenMP directives defines parallel regions to be executed by multiple CPU's concurrently as shown :

```
subroutine GHMAT

integer :: i,j,k,kk
real :: ax,bx,ay,by,sl,eta1,eta2,ra,rd1,rd2,ch,  &
        xco1,yco1,xco2,yco2,xco3,yco3,xco4,yco4, &
        tmp1,tmp2,tmp3,tmp4,rdn1,rdn2,rdn3,rdn4

X(N+1) = X(1)
Y(N+1) = Y(1)
!$OMP PARALLEL DO SHARED(N,X,Y,XM,YM)
do i=1,N
  XM(i) = (X(i) + X(i+1)) * 0.5
  YM(i) = (Y(i) + Y(i+1)) * 0.5
enddo
!$OMP END PARALLEL DO

G=0.;H=0.
do j=1,N
  kk = j + 1
  ax = (X(kk) - X(j)) * 0.5
  bx = (X(kk) + X(j)) * 0.5
  ay = (Y(kk) - Y(j)) * 0.5
  by = (Y(kk) + Y(j)) * 0.5
  sl = sqrt(ax * ax + ay * ay)
  eta1 =  ay / sl
  eta2 = -ax / sl
  xco1 = ax * GI(1) + bx
  yco1 = ay * GI(1) + by
  ...
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP DO PRIVATE(i,ra,rd1,rd2,tmp1,tmp2,tmp3,
                  tmp4,rdn1,rdn2,rdn3,rdn4)
  do i=1,j-1
    ra   = sqrt((XM(i)-xco1)*(XM(i)-xco1) +
           (YM(i)-yco1)*(YM(i)-yco1))
    tmp1 = 1. / ra
    rd1  = (xco1 - XM(i)) * tmp1
    rd2  = (yco1 - YM(i)) * tmp1
    rdn1 = rd1 * eta1 + rd2 * eta2
    ...
    G(i,j) = (log(tmp1) * OME(1) + &
              log(tmp2) * OME(2) + &
              log(tmp3) * OME(3) + &
              log(tmp4) * OME(4)) * sl
    H(i,j) = -(rdn1 * OME(1) * tmp1 + &
               rdn2 * OME(2) * tmp2 + &
               rdn3 * OME(3) * tmp3 + &
               rdn4 * OME(4) * tmp4) * sl
  enddo
  !$OMP ENDDO
  !$OMP DO PRIVATE(i,ra,rd1,rd2,tmp1,tmp2,tmp3,
                  tmp4,rdn1,rdn2,rdn3,rdn4)
  do i=j+1,N
    ...
  enddo
  !$OMP ENDDO
  !$OMP END PARALLEL
  G(j,j) = 2. * sl * (1. - log(sl))
  H(j,j) = 3.1415926
enddo
!$OMP PARALEL DO PRIVATE(i,ch)
do j=1,N
  if (KODE(j) > 0) then
    do i=1,N
      ch     =  G(i,j)
      G(i,j) = -H(i,j)
      H(i,j) = -ch
    enddo
  endif
enddo
!$OMP END PARALLEL DO
call SGEMV('N',N,N,1.,H,NX,FI,1,0.,DFI,1)
end subroutine GHMAT
```

The execution times of the parallelized version of the program are shown in Table V.

TABLE V
PARALLEL PERFORMANCE

CPUs	User	System	Total	Speedup
1	848.508	3.628	14:16.68	-
2	858.356	3.578	7:13.60	1,98
4	896.089	3.464	3:47.54	3,77
8	945.865	3.763	2:01.71	7,04
16	1090.762	3.927	1:11.72	11,95

VI. Conclusion

Profiles show the solver as the main and the first part of the code to be tuned. Use of optimized libraries is the simplest and the best way to improve performance. The benefits are greater in larger systems. Hand tuning techniques also bring good results but loop optimization is the most effective technique use in the application.. Besides the performance gain, single node optimization allows to identify in an easy way the best parts of the code to be parallized. This parallelization is accomplished simply by the insertion of OpenMP directives.

Suggestions for future works are the use of a parallel solver from an existing library and the implementation of serial and parallel versions of iterative solvers. [BLA 97] [BAR 92]

Acknowledgements :

Authors thank to SIMEPAR – Sistema de Meteorologia do Paraná that kindly granted the use of its supercomputing facilities and provided support needed in this work.

References :

[AND 99] ANDERSON, E. et al. *LAPACK User's Guide*. 3rd ed. SIAM, 1999.

[BAR 92] BARRA, L.P.S.; COUTINHO, A.L.G.A.; TELLES, J.C.F., et al. *Iterative Solution of BEM Equations by GMRES Algorithm*. Computer and Structures. 44: (6) 1249-1253 SEP 3 1992.

[BLA 97] BLACKFORD, L. S. et al; *ScaLAPACK User's Guide*. SIAM, 1987.

[BRE 84] BREBBIA, C.A.; TELLES, J.C.F.; WROBEL, L.C.; *Boundary Elements Techniques : Theory and Applications in Engineering*. Springer-Verlag, 1984.

[BRE 92] BREBBIA, C. A.; DOMINGUEZ, J. *Boundary Elements : An Introductory Course*, 2nd ed. CMP - McGraw Hill, 1992.

[CHA 01] CHANDRA, R. et al.; *Parallel Programming in OpenMP*, Academic Press, 2001.

[DOW 98] DOWD, K.; SEVERANCE, C. *High Performance Computing*, 2nd ed. O' Reilly, 1998.

[FOS 94] FOSTER, I. *Designing and Building Parallel Programs*, Addison-Wesley, 1994.

[OMP 01] OpenMP. At (JUN01) : www.openmp.org.

[SGI 98] *Origin* 2000 *and Onyx2 Performance Tuning and Optimization Guide*. Silicon Graphics Inc., document 007-3430-002, 1998. http://techpubs.sgi.com/library.

[SGI 01] *SGI 64-Bit UNIX Operating System*. At (JUN01) : www.sgi.com/software/irix6.5.

[SGI 02] *SGI Developments Products*. At (JUN01) : www.sgi.com/developers/devtools/index.html.

[TEL 83] TELLES, J.C.F. *The Boundary Element Method Applied to Inelastic Problems*, Springer-Verlag, 1983.

[WAD 00] WADLEIGH, K. R.; Crawford, I. L. *Software Optimization* for *High Performance Computing*. Prentice-Hall, 2000.