Parallel Boundary Elements Using Lapack and ScaLapack

Manoel T. F. Cunha, José C. F. Telles and Alvaro L. G. A. Coutinho
COPPE - Universidade Federal do Rio de Janeiro - Brazil
mcunha@coc.ufrj.br, telles@coc.ufrj.br, alvaro@coc.ufrj.br

Abstract

The present work introduces the main steps towards the parallelization of existing Boundary Element Method (BEM) codes using available standard and portable libraries for writing parallel programs, such as LAPACK and ScaLAPACK. Here, a well-known BEM Fortran implementation is reviewed and rewritten to run on shared and distributed memory systems. This effort is the initial step to develop a new generation of parallel BEM codes to be used in many important engineering problems. Numerical experiments on a SGI Origin 2000 show the effectiveness of the proposed approach.

1. Introduction

Since there is always a limit to the performance of a single CPU, computer programs have evolved to increase their performance by using multiple CPUs. The last decade has seen a tremendous increase in the availability and affordability of parallel systems. Over the years, there has been a surge in both quantity and scalability of multiprocessor platforms. Parallel computing techniques can have an enormous impact on an application performance and many parallel libraries facilitate access to this improvement. Since such procedures are not widespread among BEM practitioners, the present authors decided to introduce these techniques into a well-documented teaching package BEM application program, described in detail by Brebbia and Dominguez [5]. The code is herein reviewed and rewritten to run on shared and distributed memory systems using LAPACK [1], ScaLAPACK [4] and OpenMP [7]. The implementation process provides guidelines to develop parallel BEM codes, applicable to many engineering problems.

The text is organized as follows: next section presents an outline of the BEM theory, the following section presents parallel memory architectures and parallel libraries. Section 4 describes the selected application and the parallel code while in Section 5 a performance analysis is presented. The paper ends with a summary of the main conclusions.

2. Outline of BEM theory

The Boundary Element Method (BEM) [14] [6] is a technique for the numerical solution of partial differential equations with initial boundary 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 reduced dense and non-symmetric system of linear equations.

The discretization process also 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 values have been obtained, the interior points can be computed directly from the basic integral equation in a post-processing routine.

2.1. Differential Equation

Potential problems are governed by Laplace’s equation

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

(1)

and boundary conditions:

$$ u = \bar{u} \quad \text{on } \Gamma_1 $$

and

$$ q = \bar{q} = \frac{\partial u}{\partial n} \quad \text{on } \Gamma_2 $$

(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 $\Omega$ and $n$ is the unit outward normal to surface $\Gamma$.

2.2. Integral Equation

The integral equation for potential problems reads:

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

(3)
The weighting function \( u^* \) satisfies Laplace's equation and represents the field generated by a concentrated unit charge acting at the source point \( \xi \).

The function \( q^* \) is the outward normal derivative of \( u^* \) along the boundary \( \Gamma \) with respect to the field point \( x \), i.e., \( q^* (\xi, x) = \partial u^* (\xi, x) / \partial n (x) \).

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 \( \xi \) tends to the boundary \( \Gamma \). This limit must take into account the discontinuity of the second integral.

The resulting equation for potential problems is:

\[
c(\xi)u(\xi) + \int_{\Gamma} u(x) q^*(\xi, x) \, d\Gamma = \int_{\Gamma} q(x) u^*(\xi, x) \, d\Gamma
\]

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 \( \Gamma \) makes at point \( \xi \).

2.3. Discretization

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

\[
c_i u_i + \sum_{j=1}^{N} \int_{\Gamma_j} u q^* \, d\Gamma = \sum_{j=1}^{N} \int_{\Gamma_j} q u^* \, d\Gamma
\]

(5)

Notice that the source point \( \xi \) 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:

\[
c_i u_i + \sum_{j=1}^{N} \int_{\Gamma_j} u q^* \, d\Gamma = \sum_{j=1}^{N} \int_{\Gamma_j} q u^* \, d\Gamma
\]

(6)

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 \( j = j \).

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 \( \frac{1}{2} \) and the values of \( u \) and \( q \) are taken out of the integrals:

\[
\frac{1}{2} u_i + \sum_{j=1}^{N} \int_{\Gamma_j} q^* u \, d\Gamma = \sum_{j=1}^{N} \int_{\Gamma_j} u^* q \, d\Gamma
\]

(7)

The equation can thus be written as follows:

\[
\frac{1}{2} u_i + \sum_{j=1}^{N} \hat{H}_{ij} u_j = \sum_{j=1}^{N} G_{ij} q_j
\]

(8)

and can be further rewritten as:

\[
\sum_{j=1}^{N} \hat{H}_{ij} u_j = \sum_{j=1}^{N} G_{ij} q_j
\]

(9)

where:

\[
H_{ij} = \begin{cases} 
\hat{H}_{ij} & i \neq j \\
\hat{H}_{ij} + \frac{1}{2} & i = j 
\end{cases}
\]

The equation can be written in matrix form as:

\[
H \mathbf{u} = \mathbf{G} \mathbf{q}
\]

(11)

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:

\[
A \mathbf{y} = \mathbf{f}
\]

(12)

where \( \mathbf{y} \) is the vector of unknowns \( u \)'s and \( q \)'s.

2.4. 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:

\[
u_i = \int_{\Gamma} q \ u^* \, d\Gamma - \int_{\Gamma} u \ q^* \, d\Gamma
\]

(13)

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

\[
q_i = \frac{\partial u}{\partial x} = \int_{\Gamma} q \ \frac{\partial u^*}{\partial x} \, d\Gamma - \int_{\Gamma} u \ \frac{\partial q^*}{\partial x} \, d\Gamma
\]

(14)

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

2.5. Fundamental Solution

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

\[
u^*(\xi, x) = \frac{1}{2\pi} \ln \frac{1}{r} \quad q^*(\xi, x) = \frac{1}{2\pi} r_\eta
\]

(15)

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

3. Parallel programming libraries

Parallel systems are usually classified as shared or distributed memory models. In the shared memory model all the processors are able to directly access all of the memory in the machine while in distributed memory each processor in the system is only capable of addressing memory directly associated with it.

3.1. LAPACK

LAPACK \([\text{L]}\), or Linear Algebra Package, is a library of subroutines for solving linear equations systems, least squares, eigenvalue and singular value problems. Dense and band matrices are supported as well as real and complex data. LAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.
It is designed to be efficient on a wide range of modern high-performance computers, from PCs and workstations to shared memory multiprocessors and vector processors. LAPACK routines are written so that as much as possible of the computations is performed by calls to the BLAS routines.

**BLAS.** The BLAS, or Basic Linear Algebra Subprograms, include subroutines for common linear algebra computations such as vector, matrix-vector and matrix-matrix operations.

Highly efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The BLAS form a low-level interface between LAPACK software and different machine architectures. Thus, BLAS enable LAPACK routines to achieve high-performance with portable codes.

### 3.2. ScaLAPACK

ScaLAPACK is the distributed-memory version of LAPACK. The goals of both libraries are efficiency, scalability, reliability, portability, flexibility and easy of use.

ScaLAPACK can solve systems of linear equations, linear least squares, eigenvalue and singular value decomposition problems. ScaLAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

The building blocks of the ScaLAPACK library are distributed-memory versions of the BLAS, called the parallel BLAS or PBLAS and a set of Basic Linear Algebra Communications Subprograms, BLACS.

**PBLAS.** The ScaLAPACK strategy for combining efficiency with portability is to construct the software so that as much as possible of the computation is performed by calls to PBLAS. The PBLAS routines perform global computation by relying on BLAS for local computation and the BLACS for communication.

To simplify the design of ScaLAPACK and because BLAS has proven to be a useful tool outside LAPACK, the ScaLAPACK project chose to build a parallel set of BLAS called PBLAS, which performs message-passing and whose interface is as similar to the BLAS as possible. This decision has permitted the ScaLAPACK code to be quite similar and sometimes nearly identical to the analogous LAPACK code.

The ScaLAPACK project hopes that PBLAS will provide a distributed memory standard, just as BLAS have provided a shared memory standard. This would simplify and encourage the development of high performance and portable parallel numerical software as well as providing manufactures with a small set of routines to be optimized.

**BLACS.** The BLACS, Basic Linear Algebra Communication Subprograms, is a message-passing library designed for linear algebra.

The computation model consists of one or two-dimensional process grid where each process stores pieces of the matrices and vectors. The BLACS include synchronous send/receive routines to communicate a matrix or submatrix from one process to another, to broadcast submatrices to many processes or to compute global reductions: sums, maxima and minima.

There are also routines to construct change or query the process grid. Since several ScaLAPACK algorithms require broadcasts or reductions among different subsets of grids, BLACS permits a process to be a member of several overlapping or disjoint process grids, each one labeled by a context.

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of memory hierarchy. The efficiency of the ScaLAPACK software depends on the use of block-partitioned algorithms and on efficient implementations of BLAS and BLACS provided by computer vendors for their machines. Thus, the BLAS and BLACS form a low-level interface between ScaLAPACK software and different machine architectures. Above this level, all of the ScaLAPACK is portable.

**Message Passing Libraries.** In distributed memory systems, to access information in memory connected to other processors, the user must pass messages through some network connecting the processors. Such systems are usually programmed with message passing libraries such as PVM, Parallel Virtual Machine, and MPI, Message Passing Interface.

ScaLAPACK software is designed so that it can be used with clusters of workstations through a networked environment and with a heterogeneous computing environment via MPI or PVM. Indeed, ScaLAPACK can run on any machine that supports either PVM or MPI.

### 3.3. OpenMP

OpenMP is a standard and portable Application Programming Interface (API) for writing shared memory and distributed shared memory parallel programs. One advantage of this parallelization model 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.
4. A case study

Here, an existing Fortran computer program presented by Brebbia and Dominguez [5] is reviewed and rewritten. The program is a simple code for the solution of potential problems using constant elements.

The main program defines some general variables and arrays, integer and real, as follows:

```fortran
program POCONBE
integer :: N,L,info
integer,parameter :: NX=8000
integer,dimension(NX) :: KODE,ipiv
real,dimension(NX+1) :: X,Y
real,dimension(NX) :: XM,YM,FI,DFI
real,dimension(NX,NX) :: G,H
real,dimension(20) :: CX,CY,POT,FLUX1,FLUX2

call INPUT
call GHMAT
call SGESV (N,1,G,NX,ipiv,DFI,NX,info)
if (info /= 0) stop
call INTER
call OUTPUT
end program POCONBE
```

In order to compute the off-diagonal coefficients, the routine GHMAT 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 subroutine INTER to compute potential and fluxes at internal points.

SGESV is a LAPACK routine which solves the linear system of equations $Ax = b$ by factorizing $A$ and overwriting $b$ with the solution $x$.

In an optimized version of the program, EXTIN and LOCIN are inlined into GHMAT and loop optimizations are applied [9]:

```fortran
subroutine GHMAT
integer :: i,j,k,kk
real :: ax,ay,by,sl,eta1,eta2,ra,rd1,rd2,rdn1,rdn2,rdn3,rdn4
real,matrix dimension(4) :: GI = (/0.86113631,-0.86113631, 0.33998104,-0.33998104/), &
                           OME = (/0.34785485,0.34785485, 0.65214515,0.65214515/)

      X(N+1) = X(1)
      Y(N+1) = Y(1)
      do i=1,N
          XM(i) = (X(i) + X(i+1)) * 0.5
          YM(i) = (Y(i) + Y(i+1)) * 0.5
      enddo
      G=0.;H=0.
      do j=1,N
          if (i /= j) then
              ra   = sqrt((XM(i)-xco1)*(XM(i)-xco1) + &
                           (YM(i)-yco1)*(YM(i)-yco1))
              temp1 = 1. / ra
              rd1   = (xco1 - XM(i)) * temp1
              rd2   = (yco1 - YM(i)) * temp1
              rdn1  = rd1 * eta1 + rd2 * eta2
              G(i,j) = (log(temp1) * OME(1) + &
                           log(tmp4) * OME(4)) * sl
              H(i,j) = - (rdn1 * OME(1) * tmp1 + &
                           rdn4 * OME(4) * tmp4) * sl
          else
              G(j,j) = 2. * sl * (1. - log(sl))
              H(j,j) = 3.1415926
          endif
      enddo
      call SGEMV ('No transpose',N,N,1.,H,NX,FI,1,0.,DFI,1)
end subroutine GHMAT
```

The following profiler output shows the initial execution times of the whole program and of each function separately:

```
------------------------------------------------
<table>
<thead>
<tr>
<th></th>
<th>%</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>POCONBE</td>
<td>100</td>
<td>269.547</td>
</tr>
<tr>
<td>SGESV</td>
<td>70</td>
<td>188.791</td>
</tr>
<tr>
<td>GHMAT</td>
<td>29</td>
<td>80.715</td>
</tr>
<tr>
<td>INTER</td>
<td>0</td>
<td>0.041</td>
</tr>
<tr>
<td>INPUT</td>
<td>0</td>
<td>0.000</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>0</td>
<td>0.000</td>
</tr>
</tbody>
</table>
------------------------------------------------
```
The results produced by the program profiling show that SGESV routine takes the greatest portion of execution time and should be the first part of the code to get programmer’s attention. In larger systems, the solver has a major importance and changes in this part of the code bring the best results in the overall performance.

4.1. A shared-memory implementation

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 shared-memory parallel version of the solver, done with a simple compiler option.

The LAPACK routines are written as a single thread of execution. However, LAPACK can accommodate shared-memory machines, provided parallel BLAS are available.

4.2. A distributed-memory implementation

ScaLAPACK provides parallel implementations for the serial LAPACK routines. Hence, the SGESV routine can be replaced by its equivalent parallel routine PSGESV. In both cases the letter S in the routines names refers to single precision real data type. The libraries also provide routines for double precision, complex and double complex cases. The next two letters, GE, indicate a general, i.e., non-symmetric matrix. ScaLAPACK can also handle banded, tridiagonal and other matrix types. Simple driver routines have names ending by SV.

Four basic steps are required to call a ScaLAPACK routine:

- Initialize the process grid.
- Distribute the matrix on the process grid.
- Call ScaLAPACK routine.
- Release the process grid.

The application here in study can be rewritten using these steps, each one detailed below.

Process Grid. The P processes of an abstract parallel computer are often represented as a one-dimensional linear array of processes labeled 0, 1, ..., P-1. It is often more convenient to map this one-dimensional array of processes into a two-dimensional rectangular grid. This grid will have P_r rows and P_c columns, where P_r * P_c = P. A process can now be referenced by its row and column coordinates within the grid. The processes can be mapped to the process grid by using row-major order where the numbering of the processes increases sequentially across each row. Similarly, the processes can be mapped in a column-major order whereby the numbering of the processes proceeds down each column of the process grid.

Contexts. In ScaLAPACK, and thus BLACS, each process grid is enclosed in a context. A context partitions the communication space. A process grid can safely communicate even if other, possibly overlapping process grid is also communicating. But a message sent from one context cannot be received in another context.

In most aspects, the terms process grid and context can be used interchangeably. The slightly difference here is that the user can define two identical process grids, but each will be enclosed in its own context, so that they are distinct in operation, even though they are indistinguishable from a process grid standpoint.

A call to BLACS routine BLACS_PINFO determines the number of process and the current process number. The BLACS_GET and BLACS_GRIDINIT routines get a default system context and initialize the process grid using a row-major ordering. The BLACKS_GRIDINFO routine allows the user to identify the process coordinates, row and column. Two BLACS routines, BLACS_GRIDEXIT and call BLACS_EXIT, release the process grid and free the BLACS context. The process grid can alternatively be initialized with routine SL_INIT.

Data distribution. ScaLAPACK requires that all global data, vectors and matrices, be distributed across the processes prior to invoking the routines. It is the user’s responsibility to perform this data distribution.

The storage schemes of global data structures in ScaLAPACK are conceptually the same as LAPACK. Global data is mapped to the local memories of processes assuming specific data distributions. The local data on each process is referred to as the local array.

The ScaLAPACK software library provides routines that operate on three types of matrices: in-core dense matrices, in-core narrow band matrices and out-of-core dense matrices. On entry, these routines assume that the data has been distributed on the processors according to a specific data decomposition scheme. Conventional arrays are used to store locally the data when it resides in the processor’s memory.

The data layout information as well as the local storage scheme for these different matrix operands is conveyed to the routines via a simple array of integers called an array descriptor.

Array Descriptors. Each global array to be distributed across the process grid must be assigned to an array descriptor. This array stores the information required to establish the mapping between each global array entry and its corresponding process and memory location.

This array descriptor is most easily initialized with a call to a ScaLAPACK TOOLS routine called DESCINIT and must be set prior to the invocation of a ScaLAPACK routine.
**In-core Dense Matrices.** The choice of an appropriate data distribution heavily depends on the characteristic or flow of the computation in the algorithm.

The two main issues in choosing a data layout for dense matrix computations are:

- Load balance, or splitting the work evenly among the processors.
- Use of Level 3 BLAS during computations on a single processor, to account for the memory hierarchy on each processor.

Dense matrix computations feature a large amount of parallelism, so that a wide variety of distribution schemes have the potential for achieving high performance. The block-cyclic data layout selected for the dense algorithms implemented in ScaLAPACK is principally due to the scalability, load balance and communication properties.

The block-partitioned computation proceeds in consecutive order just like a conventional serial algorithm. This essential property of the block cyclic data layout explains why the ScaLAPACK design has been able to reuse the numerical and software expertise of the sequential LAPACK library.

The one-dimensional block cyclic column distribution scheme divides the columns into groups of block size NB and distributes these groups in a cyclic manner. This layout includes two special cases, the one-dimensional block column distribution and the one-dimensional cyclic column distribution, where NB = N / P and NB = 1, respectively.

The two-dimensional block cyclic distribution scheme used in the ScaLAPACK library for dense matrix computations is a mapping of a set of blocks onto the processes.

The application can then be written as shown:

```fortran
program POCONBE
    integer       :: N,L
    integer,parameter :: NX=8000
    integer,dimension(NX) :: KODE
    real,dimension(NX+1)  :: X,Y
    real,dimension(NX)    :: FI,XM,YM
    real,dimension(20)    :: CX,CY

    ! BLACS variables
    integer :: IAM,NPROCS,ICTXT
    integer :: NPROW,NPCOL,MYROW,MYCOL

    ! array descriptors
    integer,parameter :: NB=64,MXLLD=MXLOCC=4064
    integer,dimension(9) :: DESCG,DESCH,DESCX

    ! local arrays
    integer       :: NP,NQ,i,j,k,l,info
    integer,dimension(NX) :: ipiv
    real,dimension(MXLLD,MXLOCC) :: H,G
    real,dimension(MXLLD)        :: LFI,DFI

    ! external function
    integer,external :: NUMROC

    ! process number and number of processes
    call BLACS_PINFO(IAM,NPROCS)
    NPROW = INT(SQRT(REAL(NPROCS)))
    NPCOL = NPROCS / NPROW
    if (MOD(NPROCS,NPROW*NPCOL) > 0) then
        NPROW = 1
        NPCOL = NPROCS
    endif

    ! get default system context and
    ! initialize the process grid
    call BLACS_GET(0,0,ICTXT)
    call BLACS_GRIDINIT(ICTXT,'Row',NPROW,NPCOL)
    call BLACS_GRIDINFO &
    (ICTXT,NPROW,NPCOL,MYROW,MYCOL)

    call INPUT
    NP = NUMROC(N,NB,MYROW,0,NPROW)
    NQ = NUMROC(N,NB,MYCOL,0,NPCOL)

    ! init array descriptors
    call DESCINIT &
    (DESCg,N,N,NB,0,0,ICTXT,MXLLD,info)
    if (info/=0) write (*,*) 'DESCINIT (g) : ',info
    call DESCINIT &
    (DESCh,N,N,NB,0,0,ICTXT,MXLLD,info)
    if (info/=0) write (*,*) 'DESCINIT (h) : ',info
    call DESCINIT &
    (DESCx,N,1,NB,1,0,0,ICTXT,MXLLD,info)
    if (info/=0) write (*,*) 'DESCINIT (x) : ',info
    if (info/=0) stop
    call GHMAT

    ! solve the linear system A X = B
    call PSGESV &
    (N,1,G,1,1,DESCG,ipiv,DFI,1,1,DESCX,info)
    if (info/=0) then
        write (*,*) 'PSGESV : ',info
        stop
    endif

    call INTER
    call OUTPUT
    call BLACS_GRIDEXIT(ICTXT)
    call BLACS_EXIT(0)
end program POCONBE

The generation of the system of equations in a distributed-memory version must also be rewritten in order to accommodate the two-dimensional block cyclic distribution:

```fortran
subroutine GHMAT
    integer :: i,j,k,l,info
    integer,dimension(NX) :: ipiv
    real,dimension(MXLLD,MXLOCC) :: H,G
    real,dimension(MXLLD)        :: LFI,DFI

    do j=1,MXLOCC
        do i=1,MXLLD
            G(i,j) = 0.
        enddo
    enddo
    ! ..
end subroutine GHMAT
```
4.3. The shared-model with OpenMP

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.

The performance of the shared memory version can be enhanced with the parallelization of the generation of the system of equations using OpenMP directives. This implementation is already presented in a previous work [8] and its performance will be compared ahead.

5. Experimental results

The applications run on a SGI's Origin 2000 with 16 R10000 250 MHz processors and 8 Gb memory. The operational system is IRIX and the compiler is MIPSpro. Profiler is SpeedShop.

A ScaLAPACK library tuned for the system was downloaded from www.netlib.org and installed in the home user directory.

For a system of equations of 8000x8000 elements and using a shared-memory parallel version of the solver, the program presented the following processing times:

<table>
<thead>
<tr>
<th>Procs.</th>
<th>User</th>
<th>System</th>
<th>Wall</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>862.341</td>
<td>3.587</td>
<td>14:30.98</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>878.876</td>
<td>3.040</td>
<td>7:58.58</td>
<td>1.82</td>
</tr>
<tr>
<td>4</td>
<td>913.357</td>
<td>3.011</td>
<td>4:44.13</td>
<td>3.07</td>
</tr>
<tr>
<td>8</td>
<td>948.206</td>
<td>3.824</td>
<td>3:02.79</td>
<td>4.76</td>
</tr>
<tr>
<td>16</td>
<td>1090.721</td>
<td>3.607</td>
<td>2:16.41</td>
<td>6.39</td>
</tr>
</tbody>
</table>

The distributed memory implementation presented here shown the following results:

<table>
<thead>
<tr>
<th>Procs.</th>
<th>User</th>
<th>System</th>
<th>Wall</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>862.341</td>
<td>3.587</td>
<td>14:30.98</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>944.393</td>
<td>5.262</td>
<td>7:58.12</td>
<td>1.93</td>
</tr>
<tr>
<td>4</td>
<td>998.318</td>
<td>10.460</td>
<td>4:15.14</td>
<td>3.41</td>
</tr>
<tr>
<td>8</td>
<td>1052.448</td>
<td>23.128</td>
<td>2:18.44</td>
<td>6.29</td>
</tr>
<tr>
<td>16</td>
<td>1199.217</td>
<td>95.945</td>
<td>2:16.41</td>
<td>9.92</td>
</tr>
</tbody>
</table>
It must be noticed that in the shared memory version only the solver is parallelized while in the distributed memory version, the generation of the system of equations is also parallelized.

A second shared-memory implementation, including the parallelization of the generation of the system of equations using OpenMP directives, shown the results below:

<table>
<thead>
<tr>
<th>Procs.</th>
<th>User</th>
<th>System</th>
<th>Wall</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>862.341</td>
<td>3.587</td>
<td>14:30.98</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>865.046</td>
<td>4.295</td>
<td>7:20.85</td>
<td>1.98</td>
</tr>
<tr>
<td>4</td>
<td>898.354</td>
<td>4.023</td>
<td>3:50.15</td>
<td>3.78</td>
</tr>
<tr>
<td>8</td>
<td>946.537</td>
<td>4.150</td>
<td>2:02.65</td>
<td>7.10</td>
</tr>
<tr>
<td>16</td>
<td>1151.94</td>
<td>4.279</td>
<td>1:16.35</td>
<td>11.40</td>
</tr>
</tbody>
</table>

6. Conclusions

The development of parallel applications for shared and distributed memory systems can be achieved in a simple and effortless way by using tuned parallel libraries.

The solution of linear equation systems are usually the most time consuming routine in BEM codes and this work shows a practical approach in the parallelization of existing or new applications using a tuned parallel solver.

The implementation of the shared memory version of the program does not demand any change in the code and brings speedups as good as the distributed version. The latter, however, needs considerable code rewriting to distribute data among processes.

Once the solver is an already tuned routine, an algorithm change can be applied to increase performance of the solution of the system of equations. Generally, iterative solvers have more modest storage requirements than direct methods and may also be faster, depending on the iterative method and the problem. The Generalized Minimal Residual, GMRES, algorithm is an efficient method to solve dense non-symmetric systems of equations produced by BEM codes [2][3][10][11][12].

Improvement of program performance is the main reason to the parallelization of a computer program. However, the concepts presented here also address the problem of running applications in memory limited systems. In this case, the size of the arrays in the program is a major concern and memory restraints may be prevalent over processing time.

LAPACK and ScaLAPACK are standard and portable libraries freely available for a variety of architectures. Thus, parallel boundary elements can be developed and distributed without other considerations than the knowledge of data distribution schemes.

Acknowledgements

The authors are indebted to SIMEPAR - Sistema de Meteorologia do Paraná by the use of its supercomputing facilities and support during the course of this work. M.T.F. Cunha is supported by a CAPES grant from the Ministry of Education, Brazil.

References: