An Implementation of Balancing Domain Decomposition

Frederic d'Hennezel

CRPC-TR93389
March 1994

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

CRPC Technical Report

CRPC-TR93389

AN IMPLEMENTATION OF BALANCING DOMAIN DECOMPOSITION METHOD

Frédéric d'Hennezel

August 1993

An implementation of balancing domain decomposition.

F. d'Hennezel

August 1993

1. Introduction

In this technical report, a implementation of the balancing domain decomposition method, introduced by Mandel [1992] is presented. The result is a kernel routine that can be used to solve linear system arising from various discretizations of elliptic partial differential equations. It is an algebraic method in the sense that are only needed in the input:

- The linear system restricted to the unknowns of each subdomain,
- A numbering of the interface unknowns which are by definition the unknowns belonging to at least two subdomains, and their correspondence to the unknown number in each subdomain.

A technique introduced by Le Tallec and Vidrascu [1993], allows to build the coarse grid operator automatically, using only the above data, for any type of domain shape, interpolation or type of elliptic problem (equations, systems...).

Throughout this paper the notations of Mandel [1992] are recalled in section 2. In the section 3, is described how the preconditioner for the Schur complement matrix is constructed automatically, using the ideas of Le Tallec & Vidrascu [1992]. The convergence of the method depends on the type of the discretized problem, finite element, finite difference, mixed finite element etc... In this report we describe the implementation of the domain decomposition method. This implementation does not change anything to the known convergence properties of the method (see Mandel [1992] for the case of finite elements). Here we describe in the section 4 the application of the presented implementation for the resolution of a flow problem using a mixed finite element method.

2. Notations.

We consider the linear system of algebraic equations

$$Ax = f, \tag{1}$$

arising from a discretized elliptic, self adjoin boundary value problem on a domain Ω. The matrix A is assumed to be symmetric and positive definite. The unknowns in the linear system are related to the value of the solution at some points of the domain Ω.

The domain is split into non-overlapping subdomains $\Omega_1, ..., \Omega_k$. The topologic interface is $\cup_{i=1}^{k} \partial\Omega_i$. At the discrete level, the interface is composed with discrete unknowns that belong to at least two subdomains. The other unknown are the *interior unknown*. We suppose that in the discretization scheme, interior unknowns belonging to two different subdomains are not related to each other.

Let x_i be the vector of unknowns corresponding to all the elements in subdomain Ω_i, and let N_i denote the 0-1 rectangular matrix that maps the degrees of freedom x_i into global degrees of freedom x; then

$$x_i = N_i^T x$$

and

$$A = \sum_{i=1}^{k} N_i A_i N_i^T, \tag{2}$$

where A_i is the local matrix $n_i \times n_i$ corresponding to subdomain Ω_i. In each subdomain, x_i is denoted whether $\overline{x}_i$ if it is an interior unknown, or $\dot{x}_i$ if it is an interface unknown. The local matrices A_i and the 0-1 matrices N_i are then split accordingly:

$$x_i = \begin{pmatrix} \overline{x}_i \\ \dot{x}_i \end{pmatrix}, \quad A_i = \begin{pmatrix} \overline{A}_i & B_i \\ B_i^T & \dot{A}_i \end{pmatrix}, \quad N_i = \left(\overline{N}_i, \dot{N}_i\right).$$

Remark. In order to verify (2), the coefficient of A for the interface unknowns are split into the different $\overline{A}_i$. In the case of finite element, this is trivially done if A_i is obtain by subassembly. In other case (finite difference methods for example), this splitting has to be define. □

After eliminating the $\dot{x}_i$ in (1), the linear system becomes

$$Su = g, \tag{3}$$

where S is the assembly of the Schur complements,

$$S = \sum_{i=1}^{k} \overline{N}_i S_i \overline{S}_i^T, \quad S_i = \overline{A}_i - B_i \dot{A}_i^{-1} B_i^T. \tag{4}$$

We assume that the local matrices A_i are symmetric and positive semi definite, with the sub matrices $\dot{A}_i$ non singular. Then the Schur complements S_i are also positive semi definite.

The space of interface unknowns is denoted by V, and V_i is the space of interface unknown for the subdomain Ω_i.

The balancing domain decomposition method of Mandel [1992], refers to a particular preconditioner for a conjugate gradient algorithm applied to the system (3).

This preconditioner is define with the following local matrices:

- A_i, $i = 1, ..., k$
- D_i, $i = 1, ..., k$ which describe a decomposition of the unity on the space V in the following sense:

$$\sum_{i=1}^{k} \overline{N}_i D_i \overline{N}_i^T = I. \tag{5}$$

- Z_i, $i = 1, ..., k$ of size $m_i \times n_i$ such that

$$\text{Null } S_i \subset \text{ Range } Z_i. \tag{6}$$

3. Construction of the balancing preconditioner.

3.1 Definition of the matrices D_i, $i = 1, ..., k$.

In order to verify the equation (5), it is enough to take for D_i the diagonal matrix with diagonal element equal to the inverse of the number of subdomains in which the unknown belongs.

The definition of the matrices D_i should reflect the coefficient variation of the elliptic problem across the interface. There are various way to take into account this variation, see for example Dryja and Widlund [1993]. In general, the matrices D_i are defined such that they can be easily computed automatically. The experience shows that diagonal matrix are sufficient in most of the cases.

3.2 Definition of the matrices Z_i, $i = 1, ..., k$.

The preconditioner requires to find a solution u_i of the problem

$$S_i u_i = s_i. \tag{7}$$

In general the matrix S_i is only semi definite. The balancing preconditioner will provide a right hand side such that the above problem has a solution.

When it has a solution, solving the problem (7) or the following matrix problem is equivalent.

$$A_i \begin{pmatrix} \dot{u}_i \\ \overline{u}_i \end{pmatrix} = \begin{pmatrix} 0 \\ s_i \end{pmatrix} \tag{8}$$

If the matrix A_i has one or several zero eigenvalues, some equations in (8) can be suppressed in order have definite system. Once solved this reduced linear system, by setting the solution equal to zero on the indices corresponding to the suppressed lines of the system, we obtain a solution of (8).

After reordering the unknowns, the matrix can be written

$$A_i = \begin{pmatrix} E_i & G_i \\ G_i^T & F_i \end{pmatrix}, \tag{9}$$

where E_i correspond to the reduced part of the system.

Then, the matrices defined by:

$$Z_i = -E_i^{-1} G_i$$

verify the property (6).

In the proposed implementation of the algorithm, the definition of the matrices Z_i is automatic.

A modified Crout factorization of the matrices A_i is used. During the factorization, each time a zero pivot is encountered it is substituted with a very large positive number denoted PN.

Let us denote $\widetilde{A}_i = L_i^T O_i L_i$ the resulting factorized matrix. If the value of PN is large enough, the solution of the linear system

$$\widetilde{A}_i \begin{pmatrix} \dot{u}_i \\ \overline{u}_i \end{pmatrix} = \begin{pmatrix} 0 \\ s_i \end{pmatrix}, \tag{10}$$

will be practically zero on the unknowns where the value PN has been substituted, and the solution will also be a solution of (8). Furthermore, by solving the system

$$\widetilde{A}_i \begin{pmatrix} \dot{u}_i \\ \overline{u}_i \end{pmatrix} = PN \begin{pmatrix} 0 \\ \lambda_i \end{pmatrix}, \tag{11}$$

we obtain

$$\overline{u}_i = Z_i \lambda_i .$$

Remark. One might think that iterative method for the solution of the local problems are more adequate. This us probably true in quite a number of case. On the other hand domain decomposition algorithm are intended to be used with and increasing number of subdomain and a fixed number of unknowns per subdomain. Therefore the issue is more in the definition and the resolution of the coarse grid problem. Here the use of direct method allows to simplify the input of the domain decomposition solver. □

4. Application to the solution of a flow problem discretized by mixed finite elements.

4.1. Presentation of the discrete problem

A new mixed finite element method has been developed for flow problems, which gives a continuous approximation of the velocity field. The theoretical study of this method, including error estimates is presented in Arbogast and Wheeler [1993]. We consider here the case where the tensor K appearing in Darcy's law is diagonal. The variational formulation is the following:

$$\begin{cases} (Ku, v) - (p, \nabla v) &= \displaystyle\int_{\partial\Omega} q.v.n \quad \forall v \in V \\ (\nabla u, w) &= (f, w) \quad \forall w \in W \end{cases} \tag{12}$$

In this method, the space V and W are defined on each element respectively by:

$$Q_{1,2,2} \otimes Q_{2,1,2} \otimes Q_{2,2,1} \text{ and } Q_{0,0,0}$$

A nodal basis is chosen such that the functions of V are continuous. The degrees of freedom for the functions of W are located at the center of the cells.

The linear system arising from this mixed finite element discretization is the following: find the couple (u, p) solution of:

$$\begin{pmatrix} M & Q \\ Q^T & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} q \\ f \end{pmatrix}. \tag{13}$$

In the system (12), the inner product (Ku, v) is computed with a quadrature rule. The points of integration are located on the degrees of freedom of the velocities, and the weights are chosen such that the function of V are integrated exactly. It leads of a diagonal sub matrix M. Thus, velocity unknown u can be eliminated in (13), in order to have an *explicit* linear system in term of the unknown p

$$Q^T M^{-1} Q\ p = Q^T M^{-1}\ q - f. \tag{14}$$

Once this linear system solved, the velocity u can be computed directly using the relation

$$u = M^{-1}\,(q - Q\ p)\,. \tag{15}$$

The matrix $Q^T M^{-1} Q$ of the linear system can be interpreted as a finite difference matrix for the pressure unknown p. The implementation of balancing domain decomposition presented here can be used

4.2. Application of the balancing domain decomposition.

The matrix $Q^T M^{-1} Q$ has to be split in different sub matrices $(Q^T M^{-1} Q)_i$ corresponding to different subdomains in order to solve the system using balancing domain decomposition.

To apply the method we have to define interior unknowns and interface unknowns. The unknowns of the linear system (14) are the pressures at the center of the cells. In order to have interface unknowns, the interface between the subdomain will be geometrically located "in the middle" of the cells.

The coefficients of the matrix $(Q^T M^{-1} Q)_i$ corresponding to interface unknowns in the decomposition, are divided by the number of time this unknown appears in a subdomain. Thus, like in (2), it is possible to write:

$$Q^T M^{-1} Q = \sum_{i=1}^{k} N_i (Q^T M^{-1} Q)_i N_i^T,$$

where the matrices N_i maps the local degrees of freedom into global degrees of freedom.

The first appendix to this report is the user's guide for the balancing domain decomposition code presented here. The second appendix is the user's guide for the mixed finite element method presented here.

References

Arbogast, T.; Wheeler, M. [1993]: to appear.

Dryja, M.; Widlund, O. [1993]: Schwarz method of Neumann-Neumann type for the three-dimensional elliptic finite element problems, to appear.

Mandel, J. [1992]: Balancing domain decomposition, to appear in *Communications on Applied Numerical Methods.*

Le Tallec, P.; Vidrascu, M. [1993]: Méthodes de décomposition de domaine en calcul des structures, *Colloque national en calcul des structures*, 11-14 mai 1993, Giens.

Appendix 1. Algebraic balancing domain decomposition: user's guide.

1. Introduction.

Consider the linear system of algebraic equations

$$Ax = b$$

arising from finite element or finite difference discretizations of a linear, elliptic, self adjoin boundary value problem on a domain Ω. The domain Ω is divided into non overlapping subdomains $\Omega_1, ..., \Omega_k$. Each unknown of the linear system correspond to a function value at a point of the domain. We then have two kinds of unknowns: the *interface unknowns* belonging to at least two subdomains, and the others called *interior unknowns*.

Using the same notation as in Mandel [1992], the matrix A is written:

$$A = \sum_{i=1}^{k} N_i A_i N_i^T \tag{16}$$

where A_i is the local matrix for the unknowns of the subdomain Ω_i, and N_i denotes the $0-1$ matrix that maps the unknowns numbers of the subdomain Ω_i to the unknown numbers of the global system. Note that since A is written as a sum, the coefficients for the interface unknowns have to be divided in the different A_i. In the case of finite elements, A_i is directly obtained by subassembly process. In the case of a finite difference stencil, the strategy for splitting the coefficients of A corresponding to the interface unknowns is not clear. A straightforward choice is to divide those coefficients by the number of time they appear in one of the local matrices $(A_i)_{i=1}^k$. It is also necessary to split the right hand side in the same way.

The domain decomposition algorithm is an iterative method that requires at each iteration, to solve independent linear systems with respect to the local matrices A_i. Here, those linear systems are solved with a direct method; skyline storage is used for the matrices A_i.

2. Skyline storage of local matrices.

The matrix elements are stored in a vector array. Since the matrix is symmetric, it is not necessary to store the upper triangular part. In addition, each line is stored from the first column that has a non zero element. To "read" such a matrix from a vector array, it is enough to know the address in that array of each diagonal element of the matrix.

For example, let us consider the following symmetric matrix where a + denotes a non zero element.

$$\begin{pmatrix} + & + & & & & \\ + & + & + & + & & \\ & + & + & & & + \\ & + & & + & + & \\ & & & + & + & + \\ & & + & & + & + \end{pmatrix}$$

Then the next figure shows the addresses of the elements that are stored in the vector array.

$$\begin{pmatrix} 1 & & & & & \\ 2 & 3 & & & & \\ & 4 & 5 & & & \\ & 6 & 7 & 8 & & \\ & & & 9 & 10 & \\ & & 11 & 12 & 13 & 14 \end{pmatrix}$$

To be able to read this matrix from a simple vector array of 14 elements, one needs the following array of indices:

$$arr(0) = 0,\ arr(1) = 1,\ arr(2) = 3,\ arr(3) = 5,$$

$$arr(4) = 8,\ arr(5) = 10,\ arr(6) = 14.$$

Note that $arr(0)$ is always equal to zero and that $arr(i) - arr(i-1)$ gives the number of elements stored on the line i.

This defines the skyline storage of the symmetric matrix. With this type of storage, it is possible to perform a Choleski factorization or a Crout factorization. A C++ class named *profilMatrix* was implemented for the skyline storage of symmetric matrix.

Remark. The efficiency of the Choleski or Crout factorization using a skyline storage is highly dependent on the average number of element stored on each line. This number is usually called the *bandwidth*. Thus, it is the task of the user to choose a numbering for the unknowns on the local problems in such a way that this bandwidth is as small as possible. For example in the case of a logically rectangular grid, it is better to number first the direction having less grid blocks.

3. Interface data structure.

The balancing domain decomposition method is a preconditioned conjugate gradient method applied to the Schur complement matrix S of A. Let S_i be the Schur complement matrix of the local matrix A_i. As in the equation (3) of Mandel [1992],

$$S = \sum_{i=1}^{k} \overline{N}_i S_i \overline{N}_i^T, \tag{17}$$

where $\overline{N}_i$ denotes the $0-1$ matrix that maps the local interface unknowns numbers to the global interface unknown numbers.

Then, to determine the action of S in a general case, we need to define three different sets of unknowns:

- The set of unknowns for S, numbered from one to the global number of interface unknowns.
- The set of unknowns for S_i, numbered independently in each subdomain from one to the local number of interface unknowns.
- The set of unknowns of A_i, numbered independently on each subdomain, from one to the total number of local unknowns.

The matrices A_i are assembled on independent meshes with some given numbering of the unknowns (as was said before a good numbering will reduce the bandwidth of the local matrices). Then, to apply the balancing domain decomposition algorithm, we need to define the numbering for S and S_i. It is done in the following way.

For each subdomain Ω_i, the C++ class *local* of type *bdryIntfc* is defined which gives:

- The global number $local.nbIntfcGloc$ of interface unknowns for S.
- The local number $local.nbIntfcLoc$ of interface unknowns for S_i.
- For $i = 1$ to $local.nbIntfcLoc$

  Its number in the local mesh $local.sdom(i)$, which is the unknown number for the local matrix A_i.

  Its number on the interface $local.intfc(i)$, which is the unknown number for the global Schur complement S.

This class is initialized in the C++ routine *recosd*. This routine works for logically rectangular grids.

Remark. The numbering for A_i and S_i are independent from one subdomain to another. Therefore, storing vectors corresponding to those matrices on different processors of a distributed memory machine is easy. Vectors corresponding to S are define on the interface; there is no natural way to partition them. In the actual program a copy of such a vector is stored for each subdomain. This type of storage is efficient with a "reasonable" number of subdomains.

4. Input and output parameters for $DD\_algorithm$.

DD_algorithm is the subroutine of a node program running under PVM, which solves a linear system using balancing domain decomposition.

- The class *pll* of type *pllinfo* contains the information for the message passing calls in *DD_algorithm*. This parameter does not appear explicitly in the argument list of *DD_algorithm*; it is define as an external and has to be initialized before *DD_algorithm* is called. *pll*, contains the following information.

  The number *procnb* of processes running under the current application.

  The number $tasknb \in \{0, ..., procnb - 1\}$ of the current process.

  The array of integer *tids*, containing the tid numbers of the different processes. Those number are affected by PVM in the master program.

  The maximum number of neighbor *maxnbr* for all the subdomains.

  The array of integer *mynbr* of size *maxnbr* which contains the number of the process on which is the designated neighbor. If the designated neighbor does not exist *mynbr* is equal to -1. In the case of a logically rectangular grid in 3D, *maxnbr* =26.

- The class $matrix_{neumann}$ of type *profilMatrix* (see §2) contains in the input, the matrix restricted to the subdomain corresponding to the current process.

- The class *local* of type *bdryIntfc* (see §3) contains in the input the data structure describing the numbering for S_i, A_i and S.

- The class *p* of type *doubleArray* contains in the input the right hand side of the linear system restricted to the subdomain corresponding to the current process. In the output, it gives the solution of the linear system on the subdomain corresponding to the current process. The solution on the interface unknowns is the same on every subdomains.

Appendix 2. Mixed finite element method for continuous approximation of the velocity field: User's Guide.

1. Introduction

A new mixed finite element method has been developed for flow problems, which gives a continuous approximation of the velocity field. We consider here the case where the tensor appearing in Darcy's law is diagonal. The linear system arising from this mixed finite element discretization is the following: find the couple (u, p) solution of:

$$\begin{pmatrix} M & Q \\ Q^T & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} q \\ f \end{pmatrix}. \tag{18}$$

Due to the quadrature rule used in the scalar product of the velocities, the submatrix M is diagonal. Then, we can easily eliminate the velocity unknown u in (1) and compute the matrix of the linear system in term of the unknown p:

$$Q^T M^{-1} Q\, p = Q^T M^{-1}\, q - f. \tag{19}$$

Once this linear system solved, the velocity u can be computed algebraically:

$$u = M^{-1}\,(q - Q\, p)\,. \tag{20}$$

The implementation has been done for rectangular meshes in three dimensions. It consists in two main routines:

- *assMatrix*, which assemble the matrix and the right hand side of the linear system (2).
- *expVel*, which compute the velocities explicitly from the pressure solution of (2) using (3).

2. Global numbering of the velocity unknowns.

On a rectangular mesh, it is very simple to have a numbering of the cells. Also, the pressure unknowns that are located at the center of each cells can have the same numbering.

The velocity unknowns are located on the vertices of the cells, on the edges of the cells and on the faces of the cells. In order to be able to distinguish those degrees of freedom, we need to number them locally on each element and globally on the mesh.

The local numbering is the same on every elements. The global numbering is defined through a map, depending on the number of the cell and the local number in that cell.

On a given rectangular mesh, we now describe this global numbering. For vertices, edges or faces we always count the degrees of freedom in the order of increasing coordinates.

- The vertex unknowns are numbered first
  - The x component of the velocity field on the vertices
  - The y component of the velocity field on the vertices
  - The z component of the velocity field on the vertices
- Then the unknowns on the edges parallel to the x axis
  - The y component of the velocity field of these edges
  - The z component of the velocity field of these edges
- Then the unknowns on the edges parallel to the y axis
  - The x component of the velocity field of these edges
  - The z component of the velocity field of these edges
- Then the unknowns on the edges parallel to the z axis
  - The x component of the velocity field of these edges
  - The y component of the velocity field of these edges
- Then the unknowns on the faces orthogonal to the x axis
- Then the unknowns on the faces orthogonal to the y axis
- Then the unknowns on the faces orthogonal to the z axis

As said above, the local numbering of the velocity unknowns is the same on every element; it is equal to the global numbering of a mesh with a single element.

The map defining the global numbering from the cell number and the local number is computed once for all by the routine *contVel::initIndex*. It is stored in the member array *index*.

3. The input-output parameter for *assMatrix*.

Here are described the input parameters of the routine *assMatrix* to be used *without* domain decomposition.
mesh of type *fdgrid* contains three arrays for the vertex coordinates of the cells, and three arrays for the sizes of the cells in each direction.

tensor of type *contVel* contains the values of the diagonal tensor used in the assembly of the matrix M. Due to the type of quadrature rule used in the variational formulation, the tensor value is needed only at the location of the velocity degrees of freedom. For this reason, the type of *tensor* is the same than for the velocities. Consequently, the user has to specify the value of the tensor using the global numbering initialized in *contVel::initIndex* (see §2).

lambda of type *bdryVal* gives for each of the six faces of the rectangular domain the type of boundary condition. It can be Neumann $lambda(i) = 0$, or Dirichlet $lambda(i) = 1$. $i = 1$ and 2 denote the faces orthogonal to the x axis of coordinates, $i = 3$ and 4 denote the faces orthogonal to the y axis of coordinates and $i = 5$ and 6 denote the faces orthogonal to the z axis of coordinates.

lbdval of type *contVel* contains the values for the Dirichlet or/and the Neumann boundary conditions and the map for the global numbering of the velocities (see §2). To initialize the member array *lbdval.index*, the user has only to type the command *lbdval.initIndex()* after declaring *lbdval*. The value of the boundary conditions, Dirichlet or Neumann, have to be given at the location of the velocity degrees of freedom that are on the boundary, with respect to this numbering.

p of type *cArray3d* contains on the output, the right hand side of the linear system in term of pressure unknowns.

assMatrix is a function returning an object of type *profilMatrix*, which contains the matrix for a the linear system (2). It is a skyline storage (see d'Hennezel [1993],§2). For example, calling *LtL(profilMatrix &)* performs a Choleski factorization of the matrix. Then, calling *invLtL(profilMatrix &, doubleArray &)* with the factorized matrix and the right hand side *p* gives in *p* the solution of the linear system.

5. The input-output parameter for *expVel*.

For sake of clarity, it has been chosen that *expVel* and *assMatrix* would be independent subroutines. Therefore the matrices M and Q are assembled again. *expVel* has a void return type, but has the same list of arguments than *assMatrix*.

mesh of type *fdgrid*. In input; the same than for *assMatrix*.

tensor of type *contVel*. In input; the same than for *assMatrix*.

lambda of type *bdryVal*. In input; the same than for *assMatrix*.

lbdval of type *contVel*. In input; the same than for *assMatrix*. In output, it contains the velocity field computed with the algebraic equation (3).

p of type *cArray3d*. In input; contains the pressure solution of the linear system (2).

7. Using *assMatrix* and *expVel* with the domain decomposition solver *DD_algorithm*.

When *DD_algorithm* (see d'Hennezel [1993]) is used, *assMatrix* and *expVel* are

used as the subroutines of a node program for each subdomain (even though there is no message passing in either of those two subroutines).

The unknowns of the linear system (2) are the pressures at the center of the cells. In other words, the interface unknowns which appear at least in two different subdomains, are in the middle of cells. Therefore those cells appear in as many subdomains as the interface unknowns they contain.

For this reason, using *assMatrix* in a domain decomposition program, requires a definition of *mesh* with one layer of cells overlapping along the interface.

The input parameter *lambda* of type *bdryVal*, contains the boundary condition for the six faces of the subdomain. In the case of domain decomposition, faces of the subdomain can be the interface with another subdomain. In that case, we have $(lambda(i) = 3$.

Using *lambda*, the subroutine *ScaleTensor* called in *assMatrix* will modify the value of tensor on the boundary of the subdomain that is on the interface. In that way, when the local matrices for the subdomains are added to form the global matrix; the same interface coefficient is not going to be counted several times.

The solution of the linear system p is identical from one subdomain to another for the same interface unknown. Using this p as an input parameter in *expVel* with the other argument identical as in *assMatrix*, will give in the output *lbdVal*, the velocity field. Along the interface, the velocity field is not computed on degrees of freedom of the boundary. But since there is an overlap of the cells on the interface, those missing velocities are computed inside the neighbor subdomain.

The computation in *assMatrix* and *expVel* are completely independent from one subdomain to another. Therefore, not message passing call is done either in *assMatrix* or *expVel*.

All the routines for *assMatrix*, *expVel* and *DD_algorithm* are in:

hennezel/pdom/bdd

The program *slave.C*, is a node program using PVM. It uses *DD_algorithm* with *assMatrix* and *expVel*. It is a simple example, where the tensor is constant and the global domain is a cube of side one. The user gives to parameters; the number of cells in each direction and the decomposition in each direction. The boundary conditions and the right hand side are specified "in line" in the program. *Makefile* compiles all the routines needed for the executable *slave*.

The host program calling under PVM the node program *slave* is *master.C*.