On the preconditioned iterative solution of the distributed sparse linear systems in SIMRA

Ivar Lie [*] Torbjørn Utnes [†] Ragnhild Blikberg [‡]

May 26, 2004

1 Introduction

The numerical formulation of SIMRA, described in detail in [3], ends up by solving large sparse linear systems for both the advection-diffusion equations in the model and the pressure equation. The two types of equations have different properties, but they are both large and sparse with a banded structure. Due to the size of these linear systems, the only possibility is to solve them by preconditioned iterative methods.

The parallel version of SIMRA (pSIMRA) puts restrictions on the type of preconditioners that can be successfully applied. We use a domain decomposition technique to solve the linear systems in pSIMRA. Domain decomposition can be applied both on the continuous and the discrete level, and in pSIMRA we have chosen to apply domain decomposition on the linear algebra level, i.e. we distribute the global matrices to the processors and apply a domain decomposition preconditioner. This method will be described in detail below.

In this report we will also describe in some detail a few preconditioners that can be applied to distributed sparse linear systems. We have tried several of them in the numerical experiments, and the numerical results will be presented elsewhere. The preconditioners described here are only a few of the many possible ones. For others, see the large body of litterature on the subject, e.g. [6] and the references therein, or the proceedings of the domain decomposition conferences in order to get an impression.

What we have tried is to apply relatively simple preconditioners with limited amount of communication, and also to use knowledge from the field of problems to which pSIMRA is applied.

2 The basic framework

The framework we use for solving large sparse linear systems on a parallel platform is that of a distributed sparse linear system, see e.g. [9]. This is intimately connected to domain decomposition methods, and can be applied to irregularly structured sparse linear systems.

Assume that we have a finite element discretization of a PDE in a certain domain. To solve such linear systems on a distributed memory computer, it is common to partition the finite element mesh by a graph partitioner or by hand, and assign a cluster of elements which represents a physical subdomain to each processor.

[*]Storm Weather Center A/S, P.O.Box 7222, 5020 Bergen
[†]SINTEF IKT, Anvendt matematikk, 7465 Trondheim
[‡]Parallab, BCCS, UNIFOB A/S, Bergen

Each processor assembles only the local equations associated with the elements assigned to it. The general assumption is that each processor holds a set of equations of the linear system and a vector of variables associated with these rows.
For a subdomain Ω_i whose elements is assigned to processor i, we distinguish between three types of unknowns:

1. Internal variables, coupled with local equations
2. Interdomain interface variables, coupled with external equations as well as local equations
3. External interface variables, belonging to other processors and coupled with local equations

The term local variables, the variables stored processor i, consist of types 1 and 2. Type 3 variables are not stored in processor i, but they belong to the equations "owned" by processor i. If we order the unknowns in Ω_i such that type 1) comes first and then type 2), we can partition the matrix for Ω_i as shown in figure 1: Here the matrix A_i acts on the local variables

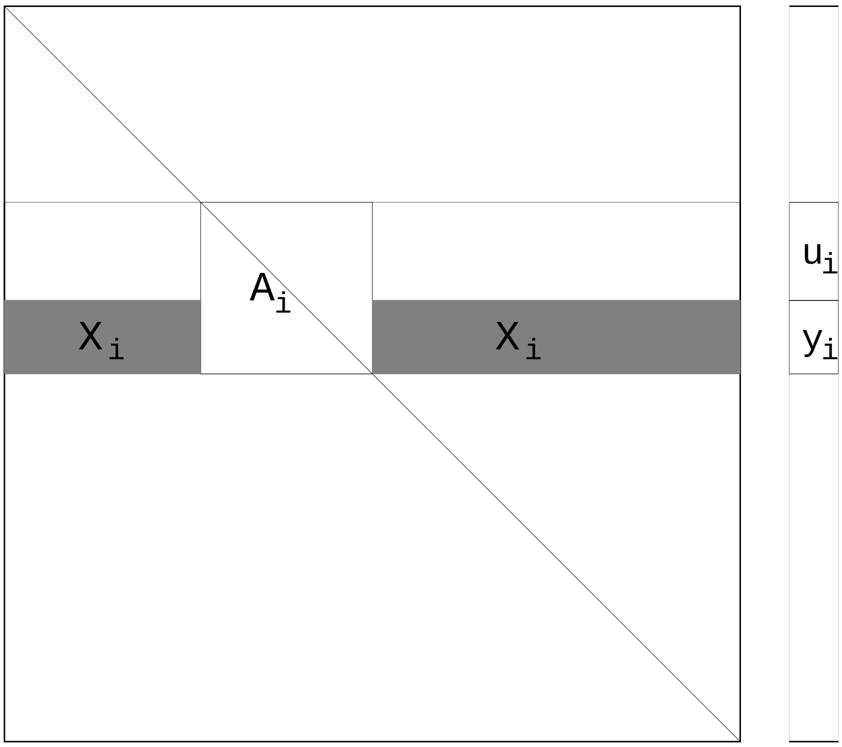


Figure 1: A partitioned sparse matrix

and the interface matrix X_i acts on the external variables which have to be received from other processors before a matrix-vector product can be completed.

The local vector x_i, $i = 1, \ldots, p$ is split into a subvector u_i of internal variables and a vector y_i of interdomain interface variables. Similarly for a right hand side b_i of the linear system involving the local matrix:

$$x_i = \begin{bmatrix} u_i \\ y_i \end{bmatrix}, \qquad b_i = \begin{bmatrix} f_i \\ g_i \end{bmatrix}.$$

The local matrix A_i can be split accordingly:

$$A_i = \begin{bmatrix} B_i & E_i \\ F_i & C_i \end{bmatrix},$$

and a linear system involving the locally stored matrix can be written

$$\begin{bmatrix} B_i & E_i \\ F_i & C_i \end{bmatrix} \begin{bmatrix} u_i \\ y_i \end{bmatrix} + \begin{bmatrix} 0 \\ \sum_{j \in \mathcal{N}_i} E_{ij} y_j \end{bmatrix} = \begin{bmatrix} f_i \\ g_i \end{bmatrix}, \tag{1}$$

where the term $\sum_{j \in \mathcal{N}_i} E_{ij} y_j$ is the contribution from the neighboring processors to Ω_i. Referring to the partitioning in figure 1, we have

$$\sum_{j \in \mathcal{N}_i} E_{ij} y_j = X_i y_{i,ext},$$

where $y_{i,ext}$ is the vector of all external interface variables.

When we use an iterative solver (which involves a global matrix-vector product), we see that this can be done in a distributed fashion by communicating the necessary interface variables between the subdomains, and that we have a computation-communication overlap since the local part of the matrix-vector product can be done independently in each subdomain.

3 Some preconditioners

We saw above how a global matrix-vector product can be computed. Obviously, for preconditioners consisting of global matrix vector products, the preconditioning operation can be carried out in the same way. On the other hand, global solves involving accurate or approximate factorization can not be done efficiently.

Below we describe some preconditioners involving either local solves or global solves of a small linear system together with local solves. As mentioned in the introduction, this is definitely not meant to be an overview of possible preconditioners, but we hope that we have selected preconditioners that are well-suited to the linear system we try to solve.

3.1 Point and block Jacobi preconditioning

Jacobi preconditioning is the simplest method we can use. It is fast since it only involves a scaling of the rows of the matrix, and it is perfectly parallel. The problem is evidently that Jacobi preconditioning is inefficient, but it may be competitive with more sophisticated preconditioners, and in particular in the parallel context, when the matrix is rather well-conditioned.

For an illustration, consider the case of 4 processors. We can write the matrix in block form,

$$\mathcal{A} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}, \tag{2}$$

where some of the off-diagonal blocks may be zero matrices. The general Jacobi preconditioner is written as,

$$M_J = \begin{bmatrix} \hat{A}_{11} & & & \\ & \hat{A}_{22} & & \\ & & \hat{A}_{33} & \\ & & & \hat{A}_{44} \end{bmatrix}. \tag{3}$$

Solving the system $M_J z = r$ for z becomes a fully parallel computation. Since this is a preconditioning, there is no need to solve the system with high precision. One can for example use a SSOR iteration, or some incomplete factorization. In case the matrix $\mathcal{A}$ is constant during the integration, such an incomplete factorization can be computed at the initial time, and only the backward and forward solves have to be performed.
We can now choose the matrices $\hat{A}_{ii}$ in several ways to construct a preconditioner:

- Point Jacobi: $\hat{A}_{ii} = \mathrm{diag}(A_{ii})$, where diag() denotes the diagonal of the argument.
- (Complete) Block Jacobi: $\hat{A}_{ii} = A_{ii}$.
- Incomplete block Jacobi: $\hat{A}_{ii}$ is some part of A_{ii}, for example a few bands in the case of a banded matrix.

Numerical experiences have shown that using a complete block Jacobi is not necessarily better than using a simpler approximation. This is presumably due to lack of diagonal dominance in $\mathcal{A}$. Hence there is no simple answer what type of block Jacobi to use, this has to be experimented with.

3.2 Parallel preconditioners of SSOR type

Consider the global matrix split as follows

$$A = D + L_B + L_N + U_B + U_N, \tag{4}$$

where D is the diagonal, L_B the strictly lower part of the matrix of the local unknowns in a subdomain and L_N is the stricly lower part of the global matrix that represents the coupling between the unknowns in the subdomains. The same goes for the strictly upper parts U_B and U_N. This is illustrated in figure 2 in the case of three subdomains, where the shaded portions of the matrix are those with elements representing the coupling between the unknowns in the subdomains.

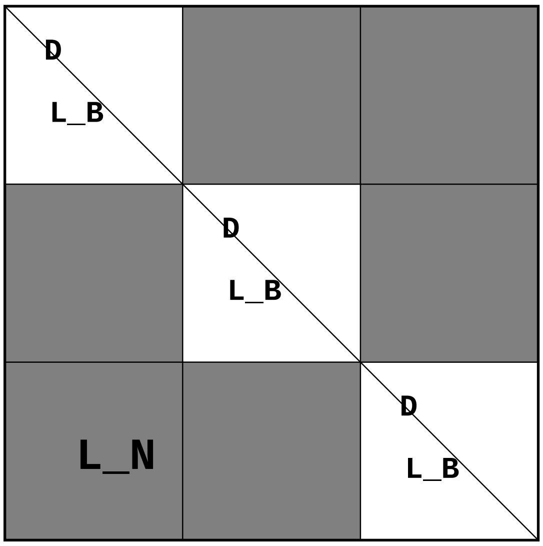


Figure 2: Matrix structure

The global and local SSOR preconditioning matrices are

$$M^g_{SSOR} = (D + \omega(L_B + L_N))D^{-1}(D + \omega(U_B + U_N)), \tag{5}$$

and

$$M^l_{SSOR} = (D + \omega L_B)D^{-1}(D + \omega U_B), \tag{6}$$

respectively. The SSOR process is an inherently sequential process, since it consists of forward and backward solves for the unknowns. When using the SSOR process as a preconditioner, one can use the so-called Eisenstat trick to save a significant part of computation. This "trick" requires that we are computing a preconditioned matrix multiplication as one operation, $w = M^{-1}A\,v$ for an arbitrary vector v. This means that the "standard" formulation of preconditioned CG that we use can not apply this "trick". However, there exist reformulations of the CG algorithm that make the application of the Eisenstat "trick" possible, but this may not be the most efficient way of expressing CG in the parallel context.

Parallelization of classical iterative schemes like SSOR has a long history, and one of the first techniques attempted was reordering of the matrix. We will discuss red-black ordering of the subdomains below. Domain decomposition techniques represent the other main avenue to parallel preconditioners, and if applied on the discrete level, it actually may be thought of as a reordering technique. For more information on parallell preconditioners, see for example [7, ch.12,13] and [2].

There is a basic fact that we want to stress: There is no preconditioner that works locally and has the same power as a global preconditioner. Hence communication is *required.* So whatever approach we use to mimick the global preconditioner, the real issue is the communication required.

An example of this is the parallel SSOR technique described in [5]. In this case the M^l_{SSOR} matrix is augmented with two matrices that are supposed to represent the couplings that the local SSOR cannot take into account. It reads:

$$M_w = (I - X_L)^{-1}(D + \omega L_B)D^{-1}(D + \omega U_B)(I - X_U)^{-1}. \tag{7}$$

If we let

$$X_L = L_N(D + \omega L_B), \qquad X_U = (D + \omega U_B)U_N,$$

and use the Neumann series

$$(I - X_L)^{-1} = I + X_L + X_L^2 + \cdots$$

we get the following preconditioning matrix if we use only the first two terms in the Neumann series:

$$M_{w1} = ((D + \omega L_B) + L_N)\, D^{-1}\, ((D + \omega U_B) + U_N)\,. \tag{8}$$

This is precisely the global SSOR matrix, and we have to perform the sequential process (or cheat in some way) in order to compute the preconditioner.

If we take three terms in the Neumann expansion, we get for the first factor of the preconditioning matrix:

$$\left((D + \omega L_B) + L_N + L_N^2(D + \omega L_B)^{-1}\right),$$

which is equally problematic to work with.

The same problem arises for the inexact inverse type of preconditioning: The application of the preconditioner requires communication of variables from neighboring subdomains. The part of the matrix needed is purely local.

Consider now SSOR when we order the subdomains in a red-black fashion. The local numbering in the subdomains can be whatever is convenient for the local solves needed. After this ordering, the global matrix can be written

$$\begin{bmatrix} D_R & L^T \\ L & D_B \end{bmatrix},$$

where D_R is a block diagonal matrix for the "red unknowns" and D_B is the same for the "black unknowns". The couplings between these two classes of unknowns are represented by the strictly block triangular matrix L and its transpose. The matrix D_R consists of blocks $D_R^{(i)}$, $i = 1, \ldots, n_r$, where n_r is the number of "red unknowns", and similarly for D_B.

The forward solve in the red-black SSOR is now

$$\begin{bmatrix} D_R & 0 \\ \omega L & D_B \end{bmatrix} \begin{bmatrix} y_r \\ y_b \end{bmatrix} = \begin{bmatrix} b_r \\ b_b \end{bmatrix}.$$

We see that the "red solves" $D_R y_r = b_r$ is computed first, then this result is applied to the matrix ωL, and then the "black solves" $D_B y_b = b_b - \omega L\, y_r$ are performed.

Similarly, the backward solve is:

$$\begin{bmatrix} D_R & \omega L^T \\ \omega 0 & D_B \end{bmatrix} \begin{bmatrix} x_r \\ x_b \end{bmatrix} = \begin{bmatrix} w_r \\ w_b \end{bmatrix}.$$

Now the "black solves" $D_B x_b = w_b$ are performed first, and then x_b is applied to ωL^T followed by the "red solves" $D_R x_r = w_r - \omega L^T x_b$. The middle step in the SSOR process amounts to multiplication by the local matrix (a block in D_R or D_B) and is therefore completely parallel. The following pseudocode illustrates what computation takes place in a processor:

```
if (myproc == red) then
   Local solve:  D_R^(i) y_{r,i} = b_{r,i}
   Send y_{r,i} to neighboring black processors
else !  black
   Receive y_r from neighboring red processors
   Compute:  b_b = b_b − ω L y_r
   Local solve:  D_B^(j) y_{b,j} = b_{b,j}
endif

Compute w = D y !  same for all processors

if (myproc == black) then
   Local solve:  D_B^(j) x_{r,j} = w_{r,j}
   Send x_{r,j} to neighboring red processors
else !  red
   Receive x_b from neighboring black processors
   Compute:  w_r = w_r − ω L^T x_b
   Local solve:  D_R^(i) x_{r,i} = w_{r,i}
endif
```

In the forward and backward solves only half the processors will be active at any time. This can be improved using more than two colors, but the logic will be more complicated. We note that the communication is always nearest neighbor because for a subdomain with a color, the neighbors have the other color.

Also, the computation of the modified right hand sides, for example $w_r = w_r - \omega L^T x_b$, takes place in parallel for all the processors of the same color.

The local solves can be performed with any reasonable method, for example SSOR-preconditioned conjugate gradients or only SSOR itself. Since we are computing a preconditioner it is not necessary to perform the local solves very accurately. Direct methods will be prohibitively costly in this context, unless the number of unknowns is small. So the computation of the preconditioner consists of an outer iteration (actually only 1 iteration), skecthed above, and inner iterations for the local solves. If we use conjugate gradients for the local solves, the same algorithm is used both at the global and local level.

3.3 Additive Schwarz preconditioning

Additive Schwarz preconditioners are the simplest (and in many cases the least efficient) preconditioners of domain decomposition type, and is based on local solves with a globally updated right hand side:

ALGORITHM 1 (ADDITIVE SCHWARZ)

1. *Update the local residual:* $r_i = (b - Ax)_i$
2. *Solve* $A_i \delta_i = r_i$
3. *Update local solution:* $x_i = x_i + \delta_i$

We see that this can be regarded as a defect correction method, and that we obtain the perfect preconditioning if the local solves are replaced by a global solve. The communication between the subdomains takes place in step 1), where the global residual is computed. This operation involves the distributed matrix-vector product as described in section 2.

The local solves in step 2) can be done by either a direct method or by an iterative solver with an ILU-type of preconditioning for example. The additive preconditioner is simple and therefore relatively fast, although compared to more sophisticated preconditioners it requires more iterations. See for example [8] for illustrative numerical experiments.

Note that the steps in Algorithm 1 above are also a part of the preconditioned conjugate gradient (PCG) method, although step 3) is slightly different. This means that the additive Schwarz preconditioner can be used directly in PCG with only local solves, but with distributed matrix-vector products. This is often called PCG acceleration of the iteration process in Algorithm 1. The local solves could be done by another PCG solver.

Overlapping subdomains give more efficient preconditioners, and with respect to the above algorithm, this means that the local solves are done for the unknowns of the extended domain. The impliciation of this is that the local matrix has to be assembled for larger subdomains than the nonoverlapping subdomains. An attractive variant is the so-called restricted additive Schwarz preconditioner, described briefly below.

3.4 Restricted Additive Schwarz preconditioning

The Restricted Additive Schwarz (RAS) preconditioning requires less communication than the usual Additive Schwarz (AS) preconditioner. RAS was actually found accidentally, when an AS method inwhich some of the communication was removed converged faster both in terms of iteration counts and CPU time, see [1].

RAS preconditioning makes use of overlapping domains (In our case LARGE_P domains, see [4]). This implies that the local matrix has to be assembled for larger subdomains than the non-overlapping subdomains. Even if this implies some extra work, one eliminates the communication of information from other neighboring subdomains.

The algorithm is given below:

ALGORITHM 2 *Restricted Additive Schwarz preconditioning*

1. *Input:* v, *Output:* $z = M^{-1}v$.
2. *For* $i = 1, ..., s$ *Do*
3. *Compute* $z_{i,local} = (R_i^{0^T} A_{i,large\_p}^{-1} R_i^{\delta})\, v$
4. *End Do*
5. $z = \sum_{i=1}^{s} z_{i,local}$

Here R_i^0 is the restriction operator corresponding to the local non-overlapping subdomain ($\delta = 0$), and R_i^{δ} is the restriction operator corresponding to the δ-overlapping subdomain (For the LARGE_P subdomain used in paper by Holstad, $\delta = 1$).

In practice, the restriction operators are applied implicitly. Computing step 3 of the RAS algorithm is done in the following manner:

a) $v_{i,large\_p}$ is exchanged with $R_i^{\delta} v$ since $v_{i,large\_p} = R_i^{\delta} v$.
($v_{i,large\_p}$ can be found in advance, before calling RAS.)

b) Compute $z_{i,large\_p} = A_{i,large\_p}^{-1}\, v_{i,large\_p} \quad \Leftrightarrow \quad$ Solve $A_{i,large\_p}\, z_{i,large\_p} = v_{i,large\_p}$
($A_{i,large\_p}$ can be found in advance, before calling RAS.)

c) $z_{i,local} = R_i^{0^T} z_{i,large\_p}$
The elements corresponding to the overlapping subdomain are removed, and $z_{i,large\_p}$ is reduced to $z_{i,local}$.

Note that no communication is needed within step a)-c), and that the For-loop in step 2-4 therefore can be executed in parallel.

The global matrix used when solving $A_{i,large\_p}\, z_{i,large\_p} = v_{i,large\_p}$ can be given as in Figure 3. This equation system can for instance be solved by an inner PCG method, with a simple preconditioning, or directly using a direct solver (like MUMPS [14]).

3.5 Schur complement techniques

Schur complement techniques refer to methods which iterate on the interface unknowns only, implicitly using the internal unknowns as intermediate variables. Since all the interface unknowns are solved simultaneously, the couplings of all subdomains are taken explicitly into account. Hence we expect this type of preconditioner to be more efficient in terms of number of iterations than additive Schwarz which uses only weak coupling. There are quite a few Schur complement type of preconditioners described in the literature, see e.g. [6, ch 3.3] and [7, ch 13.2]. We will present two methods for the distributed sparse linear systems described in section 2.
The starting point is (1) which by eliminating the unknown u_i through $u_i = B_i^{-1}(f_i - E_i y_i)$ becomes

$$S_i y_i + \sum_{j \in \mathcal{N}_i} E_{ij} y_j = g_i - E_i B_i^{-1} f_i = g_i', \tag{9}$$

where S_i is the local Schur complement $S_i = C_i - F_i B_i^{-1} E_i$. The local systems (9) can be

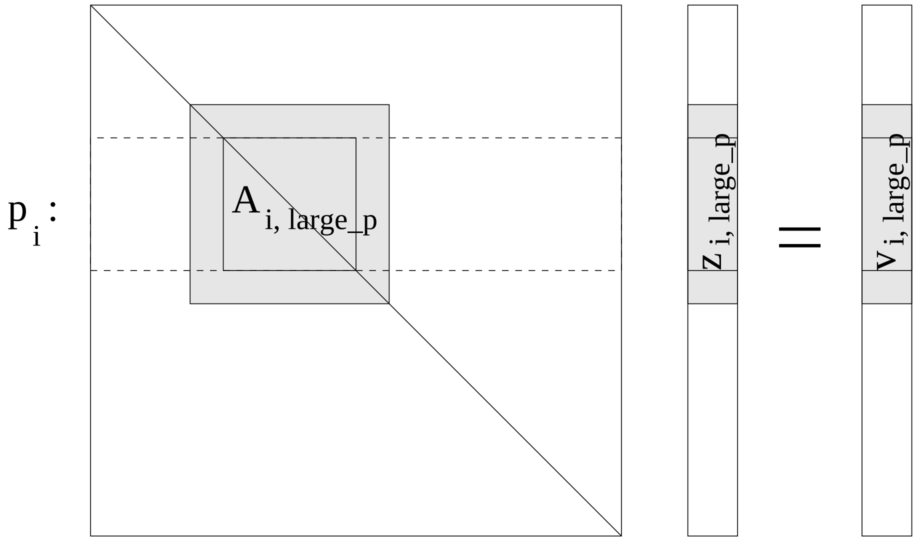


Figure 3: *A LARGE_P view of a distributed sparse matrix*

assembled into a global system,

$$\begin{bmatrix} S_1 & E_{12} & \dots & E_{1p} \\ E_{21} & S_2 & \dots & E_{2p} \\ \vdots & & \ddots & \vdots \\ E_{p1} & E_{p1-,2} & \dots & S_p \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix} = \begin{bmatrix} g'_1 \\ g'_2 \\ \vdots \\ g'_p \end{bmatrix}, \tag{10}$$

where the diagonal blocks S_i are dense in general. The off-diagonal blocks E_{ij} are sparse. The system (10) can be written in short notation

$$S\,y = g',$$

where S is the global Schur complement matrix.

Note that if we solve (10) exactly, and then use the local system $B_i u_i = f_i - E_i y_i$ to solve for the internal variables u_i, we obtain an exact solution of the distributed linear system. Schur complement preconditioners are in fact based approximately solving (10). There are several classes of such approximations, and we will give a brief description below.
A template for a Schur complement iteration is as follows:

ALGORITHM 3 (SCHUR COMPLEMENT ITERATION - TEMPLATE)

1. *Compute local right hand sides* $g'_i = g_i - F_i B_i^{-1} f_i$
2. *Solve the global Schur complement system* $S\,y = g'$
3. *Solve the local system for* u_i*:* $B_i u_i = f_i - E_i y_i$

We will now consider step 2), and begin with rewriting (10) as

$$y_i + S_i^{-1} \sum_{j \in \mathcal{N}_i} E_{ij} y_j = S_i^{-1} \left(g_i - F_i B_i^{-1} f_i \right), \qquad i = 1, \dots, p) \tag{11}$$

This can be viewed as a block-Jacobi preconditioned version of the global Schur complement system. The global system can be solved by an appropriate iterative solver, requiring a solve with S_i at each step.

Since S_i is dense in general, direct methods are good candidates. One technique is based on the observation that a LU-factorization of S_i can be derived from the LU-factorization of A_i, see [8] for details. Another technique is based on constructing an approximate inverse for S_i, see again [8] for details.

Another class of Schur complement techniques is based on the so-called expanded Schur complement system. For this system we split the internal variables vector u_i in subdomain Ω_i into a vector of interior points $\tilde{u}_i$ and a vector of local interface points $\hat{u}_i$. The local system (1) can be permuted accordingly:

$$\begin{bmatrix} \tilde{B}_i & \hat{E}_i & E_i' \\ \hat{F}_i & \hat{C}_i & E_i'' \\ F_i' & F_i'' & C_i \end{bmatrix} \begin{bmatrix} \tilde{u}_i \\ \hat{u}_i \\ y_i \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \sum_{j \in \mathcal{N}_i} E_{ij} y_j \end{bmatrix} = \begin{bmatrix} \tilde{f}_i \\ \hat{f}_i \\ g_i \end{bmatrix}. \tag{12}$$

In (12), the matrix $\tilde{B}_i$ is block diagonal. By eliminating the interior unknowns $\tilde{u}_i$ in each processor, the expanded Schur complement system for the unknown vector

$$z_i = \begin{bmatrix} \hat{u}_i \\ y_i \end{bmatrix}$$

is formed:

$$S_i z_i + \begin{bmatrix} 0 \\ \sum_{j \in \mathcal{N}_i} E_{ij} y_j \end{bmatrix} = \begin{bmatrix} \hat{f}_i - \hat{F}_i \tilde{B}_i^{-1} \tilde{f}_i \\ g_i - F_i' \tilde{B}_i \tilde{f}_i \end{bmatrix} \equiv g_i', \tag{13}$$

These local system can now be assembled into a global Schur complement system, exactly as for (10). We can write the global system as a block-Jacobi preconditioned system

$$z_i + S_i^{-1} \begin{bmatrix} 0 \\ \sum_{j \in \mathcal{N}_i} E_{ij} y_j \end{bmatrix} = S_i^{-1} g_i', \tag{14}$$

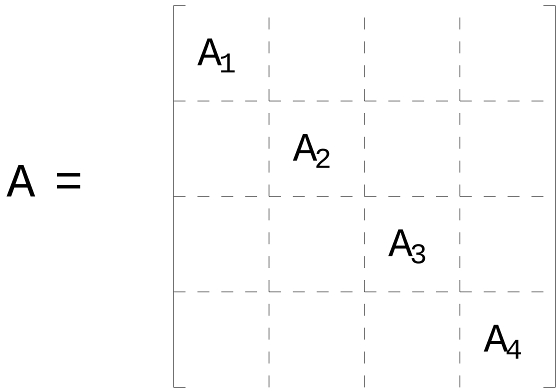
which can be solved by the methods described above: An iterative solver with either LU-factorizations or approximate inverses to compute S_i^{-1}. In [9] the Algebraic Recursive Multilevel Solver (ARMS) is used for this purpose, and numerical experiments are reported.

Even better convergence properties has been obtained with a distributed ILU preconditioner for the global Schur complement system, see [9] for a description.

The expanded Schur complement preconditioner involves more unknowns, namely the inter-domain interface unknowns, and is therefore expected to be more efficient than the "ordinary" Schur complement preconditioners. However, the computational complexity is higher, and in terms of CPU time the expanded preconditioners are not always competitive.

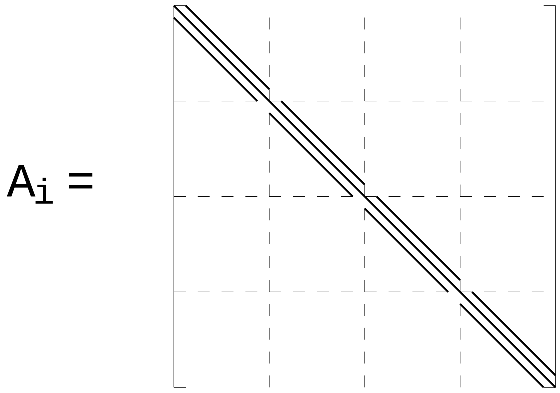
3.6 Line Jacobi and one-dimensional ADI preconditioning

Consider the global pressure matrix $A \in \mathbb{R}^{n \times n}$,



,

with 4 “stripes” in this example.
Because of the numbering of the unknowns (z-column by z-column), the coupling between the unknowns in the z-direction for a local matrix $A_i \in \mathbb{R}^{n_i \times n_i}$ is:



with 4 columns in this illustration. There is a symmetric and positive definite tridiagonal matrix for each column.

Factorization of the collection of these tridiagonal matrices can be done locally on one processor, and this factorization is done once since the pressure matrix is constant.
We will take a look at two one-dimensional preconditioners, namely one-dimensional ADI, described in [10] and [11], and line Jacobi described in [11]. The advantage of these preconditioners is that they require minimal communication (we will see that below), since the tridiagonal matrices are computed and used locally.
The starting point is the matrix splitting,

$$A = T + R, \tag{15}$$

where T is the collection of all the symmetric tridiagonal matrices above, and R is the remaining part which represents the coupling in the x-and y-direction. We also write

$$\hat{A}_i = T_i + R_i, \qquad T_i, R_i, \hat{A}_i \in \mathbb{R}^{n_i \times n}, \tag{16}$$

for a matrix “stripe” $\hat{A}_i$, which contains both the local matrix A_i and the remaining part representing the interdomain coupling. Note that the matrix T_i consists of a tridiagonal matrix plus zeros on each side. Let $\hat{T}_i$ denote this tridiagonal matrix, hence $\hat{T}_i \in \mathbb{R}^{n_i \times n_i}$. For an arbitrary vector $z \in \mathbb{R}^n$, we have that

$$T_i z = \hat{T}_i z_i, \qquad z_i \in \mathbb{R}^{n_i}. \tag{17}$$

The preconditioning operation in PCG is

$$\text{Solve} \qquad M\, z = r,$$

and the ideal situation would be if $M = \hat{A}_i$ for processor i,

$$(T_i + R_i) z = r_i. \tag{18}$$

Let us introduce a relaxation scheme in (18), and denote iterates with superscripts,

$$T_i z^{(k+1)} + R_i z^{(k)} = r_i.$$

Now using (17), we can write,

$$\hat{T}_i z_i^{(k+1)} + R_i z^{(k)} = r_i. \tag{19}$$

This can be solved efficiently since $\hat{T}_i$ can be factorized efficiently once and for all. A more convenient form is

$$\hat{T}_i z_i^{(k+1)} + \hat{A}_i z^{(k)} - T_i z^{(k)} = r_i,$$

or

$$\hat{T}_i (z_i^{(k+1)} - z_i^{(k)}) + \hat{A}_i z^{(k)} = r_i, \tag{20}$$

again by using (17). Hence one iteration consists of a matrix-vector multiplication and a local tridiagonal solve, so communication is only involved in the matrix-vector multiplication.
When will this iteration converge? If we write (19) for two consecutive iterations we get,

$$\hat{T}_i (z_i^{(k+1)} - z_i^{(k)}) + R_i (z^{(k)} - z^{(k-1)}) = 0.$$

If we write $\delta_i^{(k+1)} = z_i^{(k+1)} - z_i^{(k)}$ and $\delta^{(k)} = z^{(k)} - z^{(k-1)}$, the result is,

$$\delta_i^{(k+1)} = -\hat{T}_i^{-1} R_i \delta^{(k)} = -\hat{T}_i^{-1} (\hat{A}_i - T_i) \delta^{(k)} = -\hat{T}_i^{-1} \hat{A}_i \delta^{(k)} + \delta_i^{(k)},$$

by using (17). We partition the matrix stripe $\hat{A}_i$ into the local matrix and the matrix involving the couplings to other subdomains:

$$\hat{A}_i = \hat{A}_{i,loc} + A_{i,coup},$$

and by observing that $\hat{A}_{i,loc} z = A_i z_i$, we get

$$\begin{aligned} \delta_i^{(k+1)} &= -\hat{T}_i^{-1} A_i \delta_i^{(k)} - \hat{T}_i^{-1} A_{i,coup} \delta^{(k)} + \delta_i^{(k)} \\ &= \left(I - \hat{T}_i^{-1} A_i \right) \delta_i^{(k)} - \hat{T}_i^{-1} A_{i,coup} \delta^{(k)}. \end{aligned}$$

Taking norms and using the triangle inequality gives

$$\|\delta_i^{(k+1)}\| \leq \|I - \hat{T}_i^{-1} A_i\| \|\delta_i^{(k)}\| + \|\hat{T}_i^{-1} A_{i,coup} \delta^{(k)}\|,$$

or

$$\frac{\|\delta_i^{(k+1)}\|}{\|\delta_i^{(k)}\|} \leq \|I - \hat{T}_i^{-1} A_i\| + \|\hat{T}_i^{-1} A_{i,coup}\| \frac{\|\delta^{(k)}\|}{\|\delta_i^{(k)}\|} \tag{21}$$

We see that the convergence is determined by two terms, the first term involves the local unknowns, and the second term contains the coupling to other subdomains. If the matrix A is diagonally dominant, both terms will be less than one, and the iteration will converge. However, the SIMRA pressure matrix is *not* diagonally dominant, and hence the terms on the right hand side of (21) will most likely be larger than one and the iteration will diverge.

A remedy to this situation is to weaken the coupling to the unknowns that are involved in the matrices on the right hand side of (21). One method achieving this is the so-called damped Jacobi iteration which is extensively used as a smoother in multigrid methods. This method is also analyzed for a finite element discretization of the Stokes problem in [13]. In the line Jacobi case, we can define the damped line Jacobi iteration as a modification of (20):

$$\hat{T}_i(z_i^{(k+1)} - z_i^{(k)}) = \alpha \left(r_i - \hat{A}_i z^{(k)} \right), \qquad 0 < \alpha < 1. \tag{22}$$

To analyze the convergence, introduce the increments $\delta_i^{(k+1)} = z_i^{(k+1)} - z_i^{(k)}$ and the corresponding quantity for the global vector $\delta^{(k+1)}$. The iteration (22) can now be written,

$$\delta_i^{(k+1)} = \alpha \hat{T}_i^{-1} \left(r_i - \hat{A}_i z^{(k)} \right) = \alpha \hat{T}_i^{-1} \left(r_i - \hat{T}_i z_i^{(k)} - R_i z^{(k)} \right),$$

and the same for the previous iteration:

$$\delta_i^{(k)} = \alpha \hat{T}_i^{-1} \left(r_i - \hat{T}_i z_i^{(k-1)} - R_i z^{(k-1)} \right).$$

Subtracting these two formulas gives

$$\delta_i^{(k+1)} - \delta_i^{(k)} = -\alpha \hat{T}_i^{-1} \hat{T}_i \delta_i^{(k)} - \alpha \hat{T}_i^{-1} R_i \delta^{(k)},$$

or

$$\delta_i^{(k+1)} = (1-\alpha)\delta_i^{(k)} - \alpha \hat{T}_i^{-1} R_i \delta^{(k)},$$

which leads to the following convergence estimate:

$$\frac{\|\delta_i^{(k+1)}\|}{\|\delta_i^{(k)}\|} \le (1-\alpha) + \alpha \|\hat{T}_i^{-1} R_i\| \frac{\|\delta^{(k)}\|}{\|\delta_i^{(k)}\|}. \tag{23}$$

Hence we see that for α sufficiently small, the couplings will be weakened and the iteration will converge. On the other hand, α cannot be too small either since the term $(1-\alpha)$ occurs in the estimate.

Another way to improve the convergence of the basic line Jacobi iteration is to use an expression for the coupling matrix R_i which leads to weakened coupling. Instead of using (16) to find R_i in the iteration, we propose the two-parameter expression:

$$R_i = \alpha \hat{A}_i - \beta T_i,$$

for real, positive numbers α and β. Intuitively, by setting α to a small number we will weaken the coupling, and we have the free parameter β at our disposal to optimize the convergence. We repeat the derivation above,

$$\begin{aligned} \delta_i^{(k+1)} &= -\alpha \hat{T}_i^{-1} A_i \delta_i^{(k)} - \alpha \hat{T}_i^{-1} A_{i,coup} \delta^{(k)} + \beta \delta_i^{(k)} \\ &= \left(\beta I - \alpha \hat{T}_i^{-1} A_i \right) \delta_i^{(k)} - \alpha \hat{T}_i^{-1} A_{i,coup} \delta^{(k)}. \end{aligned}$$

This gives the error estimate:

$$\frac{\|\delta_i^{(k+1)}\|}{\|\delta_i^{(k)}\|} \leq \|\beta I - \alpha \hat{T}_i^{-1} A_i\| + \|\alpha \hat{T}_i^{-1} A_{i,coup}\| \frac{\|\delta^{(k)}\|}{\|\delta_i^{(k)}\|}, \tag{24}$$

and we see that taking α small reduces the second term in the error estimate. Choosing β is not so obvious, but taking $\beta = 1 + \alpha$ gives the following first term on the right hand side of (24),

$$\|I - \alpha \hat{T}_i^{-1} A_i^{(2)}\|,$$

where we have put $A_i = \hat{T}_i + A_i^{(2)}$, so $A_i^{(2)}$ represents the couplings to the local unknowns not in $\hat{T}_i$, i.e. the horizontal couplings. On the other hand we see from (24) that if α is very small, it makes good sense to make β small also, in order to reduce the first term on the right hand side. The iteration process itself now reads:

$$\hat{T}_i z_i^{(k+1)} + \alpha \hat{A}_i z^{(k)} - \beta \hat{T}_i z_i^{(k)} = r_i,$$

or

$$\hat{T}_i (z_i^{(k+1)} - z_i^{(k)}) + \alpha \hat{A}_i z^{(k)} + (1 - \beta) \hat{T}_i z_i^{(k)} = r_i. \tag{25}$$

Comparing this with (20) we see that the right hand side in the linear system is more complicated since a multiple of $\hat{T}_i z_i^{(k)}$ is added. This term is however local so no more communication is involved.

The ADI method is based on the same splitting as (15), (16), and is originally a method for solving the ODE system

$$\frac{\partial u}{\partial t} = A\,u - b.$$

The method reads for a timestep Δt:

$$(I - \frac{\Delta t}{2} R) u^{k+1/2} \quad = \quad (I + \frac{\Delta t}{2} T) u^k - \frac{\Delta t}{2} b \tag{26a}$$

$$(I - \frac{\Delta t}{2} T) u^{k+1} \quad = \quad (I + \frac{\Delta t}{2} R) u^{k+1/2} - \frac{\Delta t}{2} b \tag{26b}$$

An iteration solving elliptic problems, in our case $A\,u = b$, can be derived if the timestep $\frac{\Delta t}{2}$ is replaced by some acceleration parameter β.
To construct a 1D "vertical" preconditioner from (26a), (26b), we use only (26b) with $u^{k+1/2} = u^k$:

$$(I - \beta T) u^{k+1} = (I + \beta R) u^k - \beta b.$$

For the preconditioning operation $M\,z = r$ in each processor, this becomes,

$$\begin{aligned} (I - \beta \hat{T}_i) z_i^{(k+1)} &= (I + \beta R_i) z^{(k)} - \beta r_i, \\ &= (I + \beta \hat{A}_i - \beta T_i) z^{(k)} - \beta r_i, \end{aligned}$$

which leads to:

$$(I - \beta \hat{T}_i)(z_i^{(k+1)} - z_i^{(k)}) = \beta (\hat{A}_i z^{(k)} - r_i) \tag{27}$$

The matrix $(I - \beta \hat{T}_i)$ can be factored and used in the same way as $\hat{T}_i$ itself. We see that even though both preconditioners (20) and (27) use the factorization of the tridiagonal matrix as a key element, they are quite different in form. For ADI, the factor β acts as a weight for the matrix and the right hand side, whereas for line Jacobi only solution with the T_i matrix is used.

With the same notation as above we can write the iteration for two consecutive iterations as

$$(I - \beta\hat{T}_i)\delta_i^{(k+1)} = (I + \beta R_i)\delta^{(k)}, \tag{28}$$

and with $R_i = \hat{A}_i - T_i$ this becomes,

$$(I - \beta\hat{T}_i)\delta_i^{(k+1)} = (I - \beta\hat{T}_i)\delta_i^{(k)} + \beta\hat{A}_i\delta^{(k)}$$

Note that the identity matrix in this case is a part of the identity matrix of the size of the global matrix A. Now we can obtain directly:

$$\delta_i^{(k+1)} = \delta_i^{(k)} + (I - \beta\hat{T}_i)^{-1}\hat{A}_i\delta^{(k)}. \tag{29}$$

From (29) we cannot infer that the iteration converges, so we try the same technique as for line Jacobi, namely to weaken the coupling between the unknowns.
The first variant is to throw in a damping factor $\alpha < 1$ in the iteration in (27):

$$(I - \beta\hat{T}_i)(z_i^{(k+1)} - z_i^{(k)}) = \alpha\beta(\hat{A}_i z^{(k)} - r_i) \tag{30}$$

Another variant is to use a more flexible expression for the coupling matrix R_i, in this case using two more parameters:

$$R_i = \alpha\hat{A}_i - (1-\rho)T_i - \frac{\rho}{\beta}I. \tag{31}$$

Inserting this expression into (3.6), we obtain,

$$(I - \beta\hat{T}_i)\delta_i^{(k+1)} = (1-\rho)(I - \beta\hat{T}_i)\delta_i^{(k)} + \alpha\beta\hat{A}_i\delta^{(k)}, \tag{32}$$

and by multiplying by $(I - \beta\hat{T}_i)^{-1}$ and taking norms we get an error estimate of the form,

$$\frac{\|\delta_i^{(k+1)}\|}{\|\delta_i^{(k)}\|} \leq (1-\rho) + \alpha\beta\|(I - \beta\hat{T}_i)^{-1}\hat{A}_i\|\frac{\|\delta^{(k)}\|}{\|\delta_i^{(k)}\|} \tag{33}$$

and we see that if $0 < \rho < 1$ and α chosen small enough, we get convergence.
The modified iteration process using (31) is,

$$\begin{aligned}
(I - \beta\hat{T}_i)z_i^{(k+1)} &= \left[I + \beta\left(\alpha\hat{A}_i - (1-\rho)T_i - \frac{\rho}{\beta}I\right)\right] z^{(k)} - \beta r_i, \\
&= \alpha\beta\hat{A}_i z^{(k)} - \beta r_i + \left(I - \beta(1-\rho)\hat{T}_i - \rho I\right) z_i^{(k)}, \\
&= \beta(\alpha\hat{A}_i z^{(k)} - r_i) + (1-\rho)\left(I - \beta\hat{T}_i\right) z_i^{(k)},
\end{aligned}$$

so we can write the iteration process as:

$$\left(I - \beta\hat{T}_i\right)(z_i^{(k+1)} - z_i^{(k)}) = \beta(\alpha\hat{A}_i z^{(k)} - r_i) + \rho\left(I - \beta\hat{T}_i\right) z_i^{(k)}. \tag{34}$$

As for modified line Jacobi, we see that computing the right hand side of the linear system is more complicated, but local.

Numerical experiments with the original versions of these two preconditioners are reported in [11], where it is found that line Jacobi performs better for most test cases. The line Jacobi method is also used as preconditioner for the semi-implicit version of the WRF model [12]. No experiments with the modified versions have been reported, since these modified versions are believed to be new.

Line Jacobi preconditioning is simpler to use than ADI because there are fewer parameters to vary. For the ADI method the coupling parameters α and ρ must be specified, the former should be sufficiently small to make the influence from the other subdomains weak enough, and ρ should not be taken too small otherwise the coupling between the unknowns in the local subdomain will be weak and the preconditioner will be of poor quality. The acceleration parameter β must also be specified, in practice tailored to the specific matrix and the problem size. The number of iterations is usually small, the goal is to make a good preconditioner to a least possible cost.

3.7 Two–level Preconditioning

The preconditioners described above, for example the non–overlapping domain decomposition with a (vertical) line Jacobi preconditioning, and an overlapping restricted additive Schwarz (RAS) preconditioning using a point Jacobi approximation, are all examples of one-level preconditioners, meaning that the preconditioners are constructed based on only one grid, namely the model grid.

In the present section a **coarse grid correction** is described as an extension to the one–level preconditioners.

The pressure variable is treated as constant over each element in the simulation code. This implies that the restriction/prolongation operators must be constructed differently from the more common linear node–based system. A description is given about possible constructions of these operators.

3.7.1 Coarse grid correction

It is known from multigrid methods that coarse grids can remove low–frequency errors efficiently, while accuracy is ensured by means of fine grids. Similar effects will be present for iterative domain decomposition methods. For standard one–level methods an additional weakness implies that the convergence deteriorates as the number of subdomains increases. The global information propagation happen only between immediate neighboring subdomains during one iteration. A more rapid global information propagation mechanism can be obtained by including a coarse grid correction, thus implying a faster convergence, see e.g. Quarteroni and Valli [6] and Smith et al. [16].

A coarse grid correction can be constructed as follows: Assume that the linear system to be solved is given by

$$A_F u_F = f \tag{35}$$

where index 'F' indicates that this is the 'fine grid'.

We can construct a coarse grid with the corresponding coarse grid coefficient matrix A_C. Let R^T be an interpolation operator from the coarse to the fine grid. The operator R is then a so-called restriction operator, meaning an interpolation operator from the fine to the coarse grid. There is a direct analogy to multigrid, and we may alternatively write these operators as

$$R = I_F^C, \quad R^T = I_C^F$$

where I_F^C indicates interpolation from F–grid to C–grid, and I_C^F interpolation from C–grid to F–grid.

Assume that the solution error or residual is known on the fine grid. We can then approximate the residual on the coarse grid, and calculate a coarse grid correction. A simple coarse grid correction is given by

$$u_F \leftarrow u_F + R^T A_C^{-1} R\,(f - A_F u_F) \tag{36}$$

This means that we calculate the residual on the fine grid, restrict it to the coarse grid, then solve the coarse grid problem, and finally interpolate the coarse grid solution back to the fine grid.

Two–level preconditioner The coarse grid correction may be included in the method by simply adding the coarse grid preconditioner to the one–level subdomain preconditioner. We denote the preconditioner for each subdomain by $M_i^{-1} = B_i$, and may then write an **additive subdomain preconditioning** as

$$z_F = \sum_i B_i r_F \tag{37}$$

where the residual is given by $r_F = f - A_F u_F$. The present subdomain preconditioners in SIMRA may all be written in this general form.

The corresponding two–level method may be written in either additive or multiplicative form. The **additive two–level preconditioning** is given by

$$z_F = \left(R^T A_C^{-1} R + \sum_i B_i \right) r_F \tag{38}$$

This implies that the additive two–level preconditioner is defined by

$$M^{-1} = R^T A_C^{-1} R + \sum_i B_i \tag{39}$$

A **multiplicative two–level preconditioning** may be defined by solving for each level separately, e.g. first the subdomain part (additive), then updating the residual and finally the coarse grid correction:

$$z_F^{n+1/2} = \sum_i B_i r_F \tag{40}$$

$$z_F^{n+1} = z_F^{n+1/2} + \theta R^T A_C^{-1} R \left(r_F - A_F z_F^{n+1/2} \right) \tag{41}$$

In this case a relaxation factor θ has also be included in the coarse grid correction.

Possible convergence improvements by including this kind of coarse grid correction have been discussed e.g. by Smith et al.in [16] and by Frommer & Szyld [15] in case of additive Schwarz and RAS subdomain preconditioning.

The convergence will depend strongly on the particular interpolation and coarse grid operator. In addition, the geometrical representation on the coarse grid must be of importance. oth of these points represent issues that should be investigated further in the context of SIMRA.

The main items to be performed in the two–level preconditioner are summarized below:

<u>FINE GRID computations</u>

- For each subdomain, solve precondition system $z_i = B_i * r_i$
  and find corresponding residuals rsd_i

- Gather all sub-residuals into global residual: $rsd_i \rightarrow rsd_f$
  and transform from local to global ordering

COARSE GRID computations

- Restriction of residual to coarse grid: $rsd_f \rightarrow rsd_c$
- Solve global coarse grid problem: $zc = A_c^{-1} * rsd_c$
- Prolongation to fine grid: $zc \rightarrow zc_f$
  and transform from global to local ordering
- Scatter result to each subdomain: $zc_f \rightarrow zc_i$
- Update solution: $z_i \leftarrow z_i + zc_i$

3.7.2 Transformations between grids

In the SIMRA code pressure is assumed constant over each element. Due to this Q_0 representation the pressure field is discontinuous between elements. In the following we discuss possible transformation algorithms between coarse and fine grids for this kind of variable. It should be stressed that possible better transformation algorithms should be investigated in future versions.

Restriction: Fine–to–Coarse grid transformation The restriction operator that transforms the pressure from fine to coarse grid is chosen as follows. For each coarse element e_c we loop over all fine elements within this coarse element to calculate the average relation

$$P_C(e_c) = \frac{\sum_{e_f \in e_c} P_F(e_f) * \Omega(e_f)}{\sum_{e_f \in e_c} \Omega(e_f)} \tag{42}$$

This gives an element averaged value for the coarse grid value based on the fine grid element values within each coarse element. The procedure is repeated for all coarse elements.

Prolongation: Coarse–to–Fine grid transformation Due to the Q_0 elements it is not obvious how to interpolate from coarse to fine grid. Two different ideas are proposed:

Alternative 1:

- First inject values from the coarse grid to the fine grid.
- Then interpolate on the fine grid, by averaging over elements in each direction. This procedure can be refined by use of weighting, and by choosing the number of elements in the interpolating procedure according to the number of fine elements within each coarse element.

Alternative 2:

- On the coarse grid, re-interpolate from element based to node based values by using a weighted element size distribution.
- Then interpolate from the node based coarse grid to the midpoints of the fine grid, assuming these midpoints to represent element averaged values.

Alternative 1 Injection gives the same value for element e_f as for the coarse element e_c where it is situated:

$$\hat{P_F}(e_f) = P_C(e_f \in e_c) \tag{43}$$

The interpolation of the preliminary values $\hat{P_F}$ on the fine elments may be written in symbolic formm as

$$P_F(e_f) = \frac{\sum_{e_f \in \Omega_{intp(e_f)}} P_F(e_f) * w_F(e_f)}{\sum_{e_f \in \Omega_{intp}} w_F(e_f)} \tag{44}$$

where $\Omega_{intp(e_f)}$ represents the domain of elements to be used for interpolation of $P_F(e_f)$, and w_F denotes the weighting.

Alternative 2 For each coarse element node n_c the following element weighted re–distribution is preformed:

$$\hat{P_C}(n_c) = \frac{\sum_{e_c \in n_c} P_C(e_c) * \Omega(e_c)}{\sum_{e_c \in n_c} \Omega(e_c)} \tag{45}$$

where $\hat{P_C}(n_c)$ is the re–interpolated coarse grid pressure for node n_c.

In order to interpolate values from the coarse to the fine grid, we use the re–distributed node values described above. The fine grid elements use constant values over each element, similar to the coarse grid. To perform the interpolation we first interpolate to the fine grid nodes, assuming trilinear interpolation, and then average over each fine element. In short notation,

$$\begin{aligned} \hat{P_F}(n_f) &= \sum_{k \in e_c \& e_f \in e_c} N_k(\xi) \hat{P}_{Ck} && (46) \\ P_F(e_f) &= \sum_{k \in e_f} N_k(0) \hat{P}_{Fk} && (47) \end{aligned}$$

where ξ denotes the non–dimensional coordinates of e_f, and $\xi = 0$ is the element midpoint. The index k denotes function evaluation at node points.

3.7.3 Remarks

It seems that an efficient coarse grid correction strongly depends on two issues: (1) The transformation operators between grids; (2) the geometrical representation on the coarse grid. The present implementation may be improved on both points, and this could be an interesting issue for future studies.

References

[1] X.-C. Cai, M.Sarkis: *A restricted additive Schwarz preconditioner for general sparse linear systems*, SIAM J.Sci.Comput (21)2, pp.792-797, 1999.

[2] I.S.Duff, H.A.van der Vorst:*Preconditioning and Parallel Preconditioning*, Technical Report TR/PA/98/23, CERFACS, 42 avenue G.Coriolis, 31057 Toulouse Cedex, France.

[3] A.Holstad: *The mathematical and numerical formulation of SIMRA*, http://balder.ntnu.no/ttp/

[4] A.Holstad: *The Parallel Program Structure of pSIMRA*, http://balder.ntnu.no/ttp/

[5] T.Washio, K.Hayami: *Parallel block preconditioning based on SSOR and MILU*, Num.Lin.Alg.Appl. 1(1994), pp. 533-553.

[6] A.Quarteroni, A.Valli: *Domain Decomposition Methods for Partial Differential Equations*, Oxford University Press, Oxford, 1999.

[7] Y.Saad: *Iterative Methods for Sparse Linear Systems*, Second Edition with corrections, 2000, http://cs.umn.edu/∼saad/books.html

[8] Y.Saad, M.Sosonkina: *Distributed Schur Complement Techniques for General Sparse Linear Systems*, SIAM J.Sci.Comput. 21(4), pp. 1337-1356, 1999.

[9] Z.Li, Y.Saad, M.Sosonkina: *pARMS: a Parallel Version of the Algebraic Multilevel Solver*, UMSI-2001-100, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN, 2001.

[10] W.C.Skamarock, P.K.Smolarkiewicz, J.B.Klemp: *Preonditioned Conjugate-Residual Solvers for Helmholtz Equations in Nonhydrostatic Models*, Mon.Wea.Rev. 125(1997), pp.587-599.

[11] S.J.Thomas et al.: *A New Adiabatic Kernel for the MC2 model*, Mon.Wea.Rev. 127(1999).

[12] http://www.wrf-model.org

[13] L.Mansfield: *Damped Jacobi preconditioning and coarse grid deflation for the conjugate gradient iteration on parallel computers*, SIAM J.Sci.Stat.Comput. 12(1991), pp.1314-1323.

[14] MUMPS: a MUltifrontal Massively Parallel sparse direct Solver. http://www.enseeiht.fr/lima/apo/MUMPS

[15] A.Frommer, D.B.Szyld: *An algebraic convergence theory for restricted additive Schwarz methods using weighted max norms*. Research Report 00-3-23, Temple University, Department of Mathematics. 2000, http://www.math.uni-wuppertal/SciComp/

[16] B.Smith, P.Bjørstad, W.Gropp: *Domain decomposition. Parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, 1996.