Domain Decomposition Solvers for Large Scale Industrial Finite Element Problems*

Petter E. Bjørstad[1], Jacko Koster[2], and Piotr Krzyżanowski[3**]

[1] Parallab, University of Bergen, 5020 Bergen, Norway. petter@ii.uib.no
[2] Parallab, University of Bergen, 5020 Bergen, Norway. jak@ii.uib.no
[3] Institute of Applied Mathematics, Warsaw University, Banacha 2, 02-097 Warszawa, Poland. przykry@mimuw.edu.pl

Abstract. The European research project PARASOL aimed to design and develop a public domain library of scalable sparse matrix solvers for distributed memory computers. Parallab was a partner in the project and developed a domain decomposition code for solving large scale finite element problems in a robust, yet efficient way. Although the PARASOL project finished in June 1999, Parallab has continued the development of the solver. In this paper, we report on the present status of the solver and show its performance on some challenging industrial problems.

1 Introduction

The domain decomposition solver DD is a distributed memory combined iterative/direct solver of large sparse systems of linear algebraic equations with a symmetric positive definite matrix. Though these matrices may be general in nature, DD has been developed primarily for solving finite element problems, such as solid elasticity and shell problems, even on disconnected subdomains.

The DD software package currently features two state-of-the-art domain decomposition techniques. The first technique is based on iterative substructuring with 1-level and 2-level Neumann-Neumann preconditioners [7, 10–12, 15]. The 2-level method features a variety of coarse spaces. Some of these coarse spaces are designed for specific classes of problems, but others are general purpose self-adaptive algorithms for which almost no additional problem information is required. The second technique supported by DD is to solve the whole partitioned system by using Additive Average preconditioners, see [3, 4]. In this paper however, we restrict the discussion to the iterative substructuring algorithms and the Neumann-Neumann preconditioners.

* This work has been partially supported by the PARASOL project (EU ESPRIT IV LTR Project 20160).
** The work of this author has also been partially supported through the Polish Scientific Comittee research grant KBN 2P03A02116.

2 Iterative substructuring in DD

Iterative substructuring algorithms are described in detail in [14]. In this section, we briefly mention the main features and the sources of parallelism that are exploited by the DD solver.

For iterative substructuring algorithms, the problem must be partitioned into nonoverlapping subdomains. The DD solver accepts a partitioning from the user or computes one internally (using METIS 4.0). DD accepts problems in assembled, substructured, and elemental format, but uses a common internal substructure data format. Let the physical domain be decomposed into N nonoverlapping subdomains i, $i = 1, \ldots, N$, and let the stiffness matrix $K^{(i)}$ and right-hand side $f^{(i)}$ for subdomain i be defined as:

$$K^{(i)} := \begin{pmatrix} K_{II}^{(i)} & K_{IR}^{(i)} \\ K_{RI}^{(i)} & K_{RR}^{(i)} \end{pmatrix}, \qquad f^{(i)} := \begin{pmatrix} f_I^{(i)} \\ f_R^{(i)} \end{pmatrix} . \tag{1}$$

Here, the matrix $K_{II}^{(i)}$ contains the contributions of the internal nodes, the matrix $K_{RR}^{(i)}$ the contributions of the interface nodes, and the off-diagonal matrix $K_{RI}^{(i)}$ contains the links between internal nodes and the interface nodes. $K_{IR}^{(i)}$ is the transpose of $K_{RI}^{(i)}$. The right-hand side vector $f^{(i)}$ is defined accordingly.

In a parallel environment, the subdomain matrices and vectors first need to be assigned to the available processors. DD allows a flexible mapping of the subdomains so that a good balance of the processor work load can be achieved. For example, if the number of subdomains is larger than the number of processors, more than one subdomain can be assigned to the same processor. This processor will treat these subdomains independently; that is, the subdomains are not merged. If the number of subdomains is smaller than the number of processors or if the subdomains differ widely in work and/or storage, more than one processor can be assigned to the same subdomain. These processors will perform the operations for this subdomain in parallel. In the remainder of this paper, we assume (unless stated otherwise) a one-to-one mapping of subdomains to processors.

Substructuring algorithms contain a preprocessing phase in which the internal unknowns in each substructure are eliminated. This is done by solving the N linear systems $K_{II}^{(i)} \cdot v_I^{(i)} = f_I^{(i)}$ for $v_I^{(i)}$ in parallel. DD uses the multifrontal direct sparse solver MUMPS for this. MUMPS is a software package that (like DD) was developed as part of the PARASOL project [1] and that is still being further extended (see elsewhere in this volume [2]).

The preprocessing phase also computes for each substructure i a right-hand side vector of the form $g_R^{(i)} = f_R^{(i)} - K_{RI}^{(i)} \cdot v_I^{(i)}$. Vector $g_R^{(i)}$ is defined for the interface variables of substructure i.

Finally, the preprocessing phase computes for each substructure i a local Schur complement matrix

$$S_i := K_{RR}^{(i)} - K_{RI}^{(i)} \cdot [K_{II}^{(i)}]^{-1} \cdot K_{IR}^{(i)} . \tag{2}$$

DD can construct these matrices in two ways. Either S_i is computed explicitly (using MUMPS) and available as a dense matrix, or the action of S_i on a vector is available indirectly through evaluation of the right-hand side of Equation (2). In the latter case, a matrix-vector product with matrix S_i requires three matrix-vector products and one forward/back substitution with the triangular factors of $K_{II}^{(i)}$.

After the preprocessing, a linear system remains to be solved that involves all the unknowns that lie on the interfaces between the subdomains. More precisely, DD solves the interface problem

$$S \cdot u_R = g_R \ , \quad \text{where} \quad S := \sum_{i=1}^{N} R_i S_i R_i^T \ , \quad g_R := \sum_{i=1}^{N} R_i g_R^{(i)} \ , \tag{3}$$

for interface vector u_R, by using a parallel Preconditioned Conjugate Gradient (PCG) iteration, see for example [9]. Here, R_i denotes the prolongation operator associated with subdomain i. In the sequel, a vector x_i denotes the interface vector x (that is defined for all interface variables) restricted to the interface variables of subdomain i, i.e., $x_i = R_i^T x$.

The interface matrix S in Equation (3) needs not be assembled. This provides a source of parallelism that can be exploited in the construction of the preconditioner and the PCG iteration. For example, a matrix-vector product $y = S \cdot x$ is computed as

$$y = \sum_{i=1}^{N} R_i \cdot (S_i x_i) \ , \qquad x_i = R_i^T \cdot x \ .$$

This requires N matrix-vector products $S_i x_i$, one for each subdomain, that can be computed in parallel, followed by one communication step in which the local contributions are summed.

After the interface solution vector u_R is computed, some postprocessing is still necessary to compute the internal solution vectors. Therefore, DD solves the N linear systems $K_{II}^{(i)} \cdot w_I^{(i)} = -K_{IR}^{(i)} \cdot R_i^T u_R$ for local vectors $w_I^{(i)}$ in parallel. For each system, this requires a matrix-vector product to compute the right-hand side and a forward/back substitution with the factors of $K_{II}^{(i)}$ that were computed during the preprocessing phase. Finally, the vectors $v_I^{(i)}$ and $w_I^{(i)}$ are summed to obtain the interior solution vectors $u_I^{(i)}$.

The pre- and postprocessing phases are 'embarassingly' parallel. The operations for a substructure do not depend on data from any of the other substructures that is not already available. Therefore, no communication is necessary between the processors. The PCG iteration is implemented in parallel and requires synchronization of the processors at each step of the iteration. The number of iterations that is needed strongly depends on the problem and the preconditioner that is used. In the next section, we describe the more important features of the Neumann-Neumann preconditioners that are available in DD.

3 The Neumann-Neumann preconditioners

The DD solver provides 1-level and 2-level Neumann-Neumann preconditioners for the PCG iteration. We refer the reader to [14] for an introduction to Neumann-Neumann preconditioning and to [7, 10–12] for some detailed analyses. The pseudo-code below shows how a preconditioned (local) interface vector $y_i = R_i^T y$ is obtained from a (global) interface residual vector x. The parts of the computation that are performed in parallel and/or require synchronization of the processors are indicated on the right. The matrices D_i are diagonal matrices and define a partition of unity, i.e., $\sum_{i=1}^{N} R_i D_i = I$.

foreach subdomain $i = 1, \ldots, N$
  /∗ apply 1-level Neumann-Neumann to x: ∗/
  $y_i = D_i \cdot x_i = D_i \cdot R_i^T x$ — parallel
  $x_i = S_i^{-1} \cdot y_i$ /∗ 'Neumann solve' ∗/ — parallel
  $z_i = D_i \cdot x_i$ — parallel
  $y_i = R_i^T \cdot z = R_i^T \cdot \sum_{i=1}^{N} R_i z_i$ — neigh-neigh comm.
  if 2-level Neumann-Neumann **then**
    /∗ balance the residual on the coarse level: ∗/
    $w_i = R_i^T \cdot Sy = R_i^T \cdot \sum_{i=1}^{N} R_i \cdot (S_i y_i)$ — parallel + neigh-neigh comm.
    $w_i = x_i - w_i$ — parallel
    $t_i =$ `CoarseSolve(`w_i`)` — gather-scatter comm.
    $y_i = t_i + y_i$ — parallel
  end if
end foreach

The 1-level method preconditions an interface residual vector x by

$$y = \sum_{i=1}^{N} R_i D_i S_i^{-1} D_i R_i^T x \ .$$

This requires the solution to the Neumann problems $S_i x_i = y_i$ with Neumann matrix S_i. For this, DD factorizes the Neumann matrices using either MUMPS or LAPACK, depending on the sparsity of the matrix. However, the Neumann matrices can be singular. To avoid this, DD optionally regularizes the Neumann matrices by increasing the diagonal entries with a small constant α times the largest diagonal entry (see also [8]). DD then factorizes this modified matrix. We note that the regularization does not change the problem, but only the preconditioner. Alternatively, the direct solver MUMPS can handle linear systems with rank-deficient matrix and DD optionally makes use of this.

The 2-level Neumann-Neumann preconditioner requires the solution to a coarse problem (`CoarseSolve()` in the pseudo-code). Several options are available in DD to construct a coarse matrix that enables PCG to converge quickly for a range of problems. Mandel [12] showed that the coarse problem can be formulated in terms of N local coarse spaces $Z_i = \text{span}\{z_i^k : k = 1, \ldots, n_i\}$, one for each substructure i, where each local coarse space must satisfy $Ker\, S_i \subset Z_i$. The DD solver contains efficient coarse operators for problems for which the local coarse spaces Z_i (or rigid body motions) are known in advance. This is for example the case for diffusion equation and (2D and 3D) elasticity problems. However, DD also contains coarse operators that do not require such detailed *a priori* knowledge from the user about the local coarse spaces. This makes the Neumann-Neumann preconditioner quite flexible and it enables us to implement several variants of the preconditioner in a single framework.

A flexible strategy is to construct the local coarse spaces from selected eigenvectors of the local Schur operators [6]. This requires the computation of the eigenvectors that correspond to selected eigenvalues of a Neumann problem posed on each subdomain. Since the Schur matrices are assigned to different processors, these eigenanalyses can be performed in parallel. In general, larger eigenvector based coarse spaces are more expensive to construct but improve convergence of the PCG iteration. Hence, it is important to find a balance between the dimension of the coarse space and the number of iterations. Therefore, in [6] a new coarse operator is proposed that adjusts the number of computed eigenvectors adaptively to the number of small eigenvalues found (for each S_i individually). An eigenvalue λ is considered small if

$$\lambda \;\leq\; \varLambda \;:=\; \theta \cdot \max_i \lambda_{min} K_{II}^{(i)} \; . \tag{4}$$

Here, θ is a user-defined parameter. The threshold $\varLambda$ is obtained by first computing the smallest eigenvalue for each of the matrices $K_{II}^{(i)}$, followed by a reduction operation to get the largest. This reduction operation requires a synchronization of the processors. The smallest eigenvalues (and eigenvectors) for each S_i or Neumann matrix are then computed in blocks of k values until the largest computed eigenvalue exceeds $\varLambda$. We denote this coarse option by `ADAPT(`k`)`.

If a good bound n_i for the dimension of $Ker\, S_i$ is known *a priori*, then it usually suffices to compute n_i eigenvectors. The DD solver contains a coarse space option that computes for each Neumann matrix a pre-defined number of vectors. This coarse option does not require synchronization of the processors. We denote this option by `DETER(`k`)`, where $n_i = k$, for all $i = 1, \ldots, N$.

The coarse space constructors `DETER(`k`)` and `ADAPT(`k`)` were designed to make DD a general purpose and robust domain decomposition solver. Besides these eigenvector based coarse constructors, a variant of the cross-point method described in [11] is also implemented. This method was initially designed for plate and shell problems, but is also suitable for solid elasticity problems [13]. Another coarse option is available for direct substructuring (by defining the coarse problem as the complete interface problem), and an option in which the direct solver MUMPS computes a(n exact) null space for the Schur complement matrices. More details can be found in the forthcoming report [5].

After the vector sets Z_i are determined, the (sparse) coarse matrix can be constructed. The order of the coarse matrix is equal to the sum of the local coarse space dimensions ($\sum_{i=1}^{N} n_i$) and the entries in the matrix are of the form

$$(R_j D_j z_j^l)^T \cdot S \cdot R_i D_i z_i^k \;,$$

where i, j are neighbouring subdomains or subdomains that have a common neighbour subdomain. The entries of the coarse matrix can be computed in parallel, but only after the local coarse vectors are exchanged between neighbouring subdomains. DD makes use of higher level BLAS routines where possible. After the entries of the coarse matrix are assembled, this matrix is factorized. This is done in parallel (provided the order of the matrix is large enough).

4 Experimental results

We performed numerous experiments on various types of finite element problems. These include solid, plate, and shell elasticity problems (see [6]) and large scale industrial problems provided by partners in the PARASOL project. In this section, we illustrate the performance of the DD solver on three large scale problems. The experimental results were obtained on the 128-processor (195 MHz) SGI Origin 2000 at Parallab. More experimental results can be found in [5]. The DD software is written in Fortran 90 and uses MPI for message passing.

The tables in this section use the following notation. N denotes the number of subdomains, #procs denotes the number of processors, #iter denotes the number of PCG iterations, and 'coarse' denotes the dimension of the coarse space. Furthermore, 'pre' denotes the elapsed time (in seconds) for the preprocessing phase, which includes the factorization and elimination of the internal unknowns, the construction of the Schur complement matrices, and the construction of the 2-level preconditioner, 'CG' denotes the elapsed time for the PCG iteration, and 'total' denotes the total elapsed time for solution required by DD. (The times for the postprocessing are much smaller and not listed here.) DD 3.0 is the version of the code that is currently being developed. DD version 2.5 is the code that is part of the public domain PARASOL library that was released in August 1999. DD 2.5 uses for example only implicit Schur complement matrices.

The DD solver was given the partitioned stiffness matrices from Equation (1). Right-hand sides that were not available were generated randomly. No other information (e.g., the type of elements/unknowns, decomposition details, or geometry) was provided to the solver. The subdomains are mapped onto the processors in a cyclic way; i.e., subdomain i is assigned to processor $(i-1)$ **mod** #procs. Unless stated otherwise, all Schur complement matrices are computed explicitly and the following parameters are used. Regularization parameter $\alpha = 10^{-12}$. Threshold relaxation parameter $\theta = 0.005$ for the `ADAPT` coarse variant (see Equation (4)). The PCG iteration was stopped when the l_2-norm of the residual was reduced by at least six orders of magnitude.

MSC-BMW-crankshaft. The first test case comes from the automotive industry and is provided by MSC.Software (Munich, Germany). It is an elasticity problem with solid elements and contains 148,770 degrees of freedom. The problem was partitioned (by MSC.Software) into 2 and 4 subdomains. The 4-subdomain case contains two subdomains that consist of two disconnected pieces.

Table 1 shows the experimental results for the coarse space variants `DETER(12)` and `ADAPT(6)`, for varying numbers of processors. For a 3D elasticity problem, the deficiency of a Neumann matrix is normally 6, but since some of the subdomains consist of two disconnected pieces, we use parameter $k = 12$ for the `DETER`

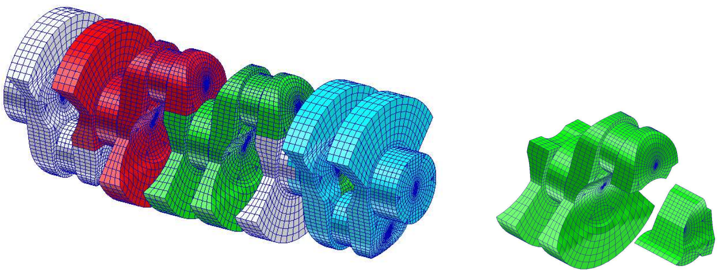

Fig. 1. The MSC-BMW-crankshaft test case decomposed into four subdomains. One of the disconnected subdomains is shown on the right.

Table 1. Results for the MSC-BMW-crankshaft test case.

DD version	N	#procs	DETER(12) coarse	#iter	times pre	CG	total	ADAPT(6) coarse	#iter	times pre	CG	total
3.0	2	2	24	19	243	6.6	256	24	19	243	6.9	256
3.0	4	4	48	22	123	13.8	140	54	20	120	8.9	131
3.0	4	1	48	22	438	38.8	493	54	20	445	30.8	490
3.0	4	2	48	22	224	27.8	262	54	20	241	20.4	269
3.0	4	4	48	22	123	13.8	140	54	20	120	8.9	131
3.0	4	8	48	22	96	12.3	112	54	20	99	11.6	114
2.5	2	2	24	19	659	130	793	24	19	672	115	790
2.5	4	4	48	22	353	113	468	54	20	404	88	496

option. The first set of two results (rows in the table) shows the default case where one subdomain is mapped onto one processor. For the 4-subdomain case, the `ADAPT(6)` option computes a slightly larger coarse problem (54 vs. 48). This results in a slightly smaller number of iterations (20 vs. 22). The speed-down in the elapsed time for the PCG iteration is not only due to the larger number of iterations for the 4-subdomain case. More precisely, the partitioning in two subdomains has an interface between the subdomains with 1533 degrees of freedom. For the partitioning into four subdomains however, the subdomains have local interfaces of 2073, 1845, 1461, and 651 degrees of freedom. These larger local interfaces increase the cost (wall-clock time) of the distributed matrix-vector product in the PCG iteration. We attribute the poor scaling of the PCG iteration also in part to inaccurate timings that we obtained at the time of experimentation (due to a heavy load on the machine).

The middle set of four results shows the performance of DD on the 4-subdomain case for 1, 2, 4, and 8 processors. For the run on eight processors, two processors are assigned to each subdomain; one processor (the slave) helps the other (the master) in the factorizations and forward/back substitutions that are performed by the direct solver MUMPS. We attribute the differences in the times for the `DETER(12)` and the `ADAPT(6)` option mainly to variations in the load of the machine at the time of experimentation. By comparing the first set of results for DD version 3.0 with the results for version 2.5, we can see a significant improvement in the performance of the solver over the past twelve months (even though the number of iterations remains the same).

DNV tubular joint (MT1). The second test case that we consider was provided by Det Norske Veritas (Oslo, Norway). The problem comes from the modelling of a tubular joint and contains a mixture of solid, shell, and transitional finite elements. The problem was partitioned by Det Norske Veritas into 58 subdomains of widely varying sizes and shapes. The problem contains 97,470 degrees of freedom; 13,920 lie on the interfaces between the subdomains.

Table 2. Results for the 58-subdomain MT1 test case using the `ADAPT(6)` coarse variant. $^{(a)}$: the solver uses only implicit Schur matrices; $^{(b)}$: the solver uses $\theta = 0.012$.

DD version	#procs	coarse	#iter	times pre	times CG	times total
3.0	1	1548	33	217.4	78.4	305.5
3.0	2	1548	33	115.5	40.2	160.7
3.0	4	1548	33	69.6	23.0	95.5
3.0	8	1548	33	42.2	12.8	56.5
3.0	16	1524	35	26.6	7.7	35.3
3.0	1	1548	33	217.4	78.4	305.5
3.0	1	1338	37	198.9	71.5	278.2
3.0	16	1524	35	26.6	7.7	35.3
3.0 $^{(a)}$	16	1548	33	88.5	27.0	116.9
3.0 $^{(b)}$	16	1836	22	41.8	7.9	50.8
3.0 $^{(a,b)}$	16	1836	22	103.3	23.0	127.6
2.5 $^{(a)}$	16	1850	22			380.0

The first set of results in Table 2 shows the experimental results using the `ADAPT(6)` coarse operator for various numbers of processors. The order of the coarse matrix is quite large (> 1500). Therefore, we experimented with a variant of the `ADAPT` coarse option in which we remove all the computed eigenvectors from the local coarse spaces that correspond to eigenvalues that exceed the threshold value Λ from Equation (4). This reduces (see the second set of results) the dimension of the coarse space for the 1-processor run from 1548 to 1338. As a result, the number of iterations goes up from 33 to 37, but the overall solution time decreases. This shows that the coarse space must be well tuned in order to preserve both the convergence and the performance of the DD solver.

We want to stress the importance of the ability to choose between explicit or implicit Schur complement matrices, which is now implemented in DD version 3.0. DD 3.0 computes 27 Schur complement matrices explicitly; the other 31 matrices are represented implicitly. For some time it has been believed that iterative substructuring algorithms should 'naturally' use implicit S_i. For the MT1 problem however, the use of implicit Schur complement matrices leads to a performance degradation of more than a factor of three (116.9 vs. 35.3 seconds total elapsed time).

Note however that the better performance of DD 3.0 is not only due to the use of a smaller coarse space and explicit Schur complement matrices. To illustrate this, we used DD 3.0 with (a) only implicit Schur complement matrices and (b) $\theta = 0.012$ in order to obtain a coarse space of dimension approximately 1850. As a result, the elapsed time for solution increases to 127.6 seconds. This is still significantly less than the 380 seconds required by DD 2.5. We attribute this to many algorithmic improvements in all parts of the code. These include for example the better use of the direct solver, faster construction of the coarse matrix, and reduced memory requirements. More details can be found in [5].

MSC-inline skater. The third test case was provided by MSC.Software and comes from the modelling of an inline skater. It contains 503,712 degrees of freedom and was partitioned (by MSC.Software) into 2, 8, and 16 subdomains. Some of the subdomains for the 8 and 16-subdomain case consist of disconnected parts. For example, the 16-subdomain test case contains eight subdomains that consist of two parts and two subdomains that consist of three parts.

Table 3. Results for the MSC-inline test case using the `ADAPT(6)` coarse variant.

DD version	N	#procs	coarse	#iter	times pre	CG	total
3.0	2	2	12	25	671	9.2	699
3.0	8	8	108	39	247	57.0	309
3.0	16	16	228	37	140	43.6	188
3.0	16	1	228	37	1185	328.0	1556
3.0	16	2	228	37	621	198.5	844
3.0	16	4	228	37	333	107.1	453
3.0	16	8	228	37	190	62.7	260
3.0	16	16	228	37	140	43.6	188
3.0	28	1	300	32	918	217.2	1177
3.0	28	2	300	32	459	123.0	606
3.0	28	4	300	32	249	64.2	326
3.0	28	8	300	32	151	39.9	198
3.0	28	16	300	32	98	29.5	131
2.5	2	2		22	1571	861	2446
2.5	8	8		31	594	257	851
2.5	16	16		30	391	126	517

Table 3 shows the experimental results for the `ADAPT(6)` coarse variant. The first set of (three) results shows the default case where one subdomain is mapped onto one processor. The largest local interface for the 2-, 8-, and 16-subdomain partitioning contains 1413, 3273, and 2958 degrees of freedom, respectively. The second set of results shows the performance of DD on the 16-subdomain test case for 1, 2, 4, 8, and 16 processors. For the third set of results, we performed the same runs as for the second set, but now the subdomains that consist of disconnected parts are split and each part is treated as a separate subdomain. This increases the number of subdomains from 16 to 28. The size of the largest local interface decreases from 2958 to 1977. It also leads to an increase in the dimension of the coarse space. This is due to the `ADAPT(6)` option that computes a multiple of six eigenvectors for each subdomain. In general, some of these eigenvectors correspond to eigenvalues that exceed the threshold Λ from Equation (4). As a result of the larger coarse space, the number of iterations decreases from 37 to 32. If we use the earlier mentioned variant of the `ADAPT(6)` option that deletes the eigenvectors with eigenvalues that exceed Λ, the dimension of the coarse space remains the same; i.e., 150 without splitting and 152 with splitting of the subdomains. The number of iterations is 56 for both cases.

5 Concluding remarks

The iterative substructuring technique in the DD solver in combination with the Neumann-Neumann preconditioners has proven capable of solving different kinds of problems in engineering and industry, in a robust, yet efficient way in a distributed memory environment. Iterative substructuring allows a high level of parallelism throughout the solution process. The new eigenvector based coarse spaces that have been designed for the 2-level preconditioner can be constructed efficiently in parallel and work well on various kinds of finite element discretizations. The DD solver uses the sparse direct solver MUMPS for the local and the coarse subproblems, because it is equipped with a range of functionalities that are usually not available in other sparse direct solvers and that have proven very useful in all phases of the solution process of our domain decomposition solver.

In our experiments with the large scale industrial problems, we observed that it is often beneficial to split large subdomains that consist of disconnected pieces. Furthermore, we also observed that it is usually more efficient to compute the Schur complement matrices explicitly than to use an implicit representation (given by the right-hand side of Equation (2)). Obviously, there is a balance in the general case. If the order of S_i is small compared to the order of the subdomain matrix $K^{(i)}$, then computing a matrix-vector product $S_i \cdot x$ with a small dense matrix S_i is cheaper than using forward/back substitution with the (large) factors of the matrix $K_{II}^{(i)}$. However, if the Schur complement matrix is relatively large, then constructing and using the dense matrix S_i is more expensive and an implicit representation of S_i is preferred. We have verified this with some 3D Poisson-type model problems (not shown in this paper).

We plan to continue the development of the DD solver. This will include algorithmic improvements that enhance the basic performance, tuning of the software, and the integration of new functionalities. We are currently looking at hybrid versions of the available coarse space variants.

The DD software is available in the public domain and up-to-date versions of the code can be obtained by contacting one of the authors.

References

1. P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. Technical Report RAL-TR-1999-059, Rutherford Appleton Laboratory, Chilton, Didcot, England, 1999. To appear in *SIAM J. Matrix Anal. Appl.*
2. P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. MUMPS: a general purpose distributed memory sparse solver. In *Proceedings of the PARA2000 Workshop on Applied Parallel Computing*. Springer-Verlag, 2000. Lecture Notes in Computer Science.
3. P. E. Bjørstad and M. Dryja. A coarse space formulation with good parallel properties for an additive Schwarz domain decomposition algorithm. Submitted to *Numerische Mathematik*, 1999.
4. P. E. Bjørstad, M. Dryja, and E. Vainikko. Robust additive Schwarz methods on unstructured grids. In P. E. Bjørstad, M. Espedal, and D. Keyes, editors, *Domain Decomposition Methods in Sciences and Engineering*. DDM.org, 1997. Ninth International Conference, Bergen, Norway.
5. P. E. Bjørstad, J. Koster, and P. Krzyżanowski. DD: a parallel domain decomposition solver for large scale industrial problems. User's Guide 3.0. Technical report, Parallab, University of Bergen, Norway, August 2000. In preparation.
6. P. E. Bjørstad and P. Krzyżanowski. An eigenvector based two-level Neumann-Neumann coarse space. Technical report, Warsaw University, Poland, May 2000.
7. Y.-H. De Roeck and P. Le Tallec. Analysis and test of a local domain decomposition preconditioner. In R. Glowinski, Y. Kuznetsov, G. Meurant, J. Périaux, and O. Widlund, editors, *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, pages 112–128. SIAM, 1991.
8. M. Dryja and O. B. Widlund. Schwarz methods of Neumann-Neumann type for three-dimensional elliptic finite element problems. *Comm. Pure Appl. Math.*, 48(2):121–155, 1995.
9. G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.
10. P. Le Tallec, J. Mandel, and M. Vidrascu. Balancing domain decomposition for plates. In *Domain Decomposition Methods in Scientific and Engineering Computing, University Park, 1993*, pages 515–524. Amer. Math. Soc., Providence, 1994.
11. P. Le Tallec, J. Mandel, and M. Vidrascu. A Neumann-Neumann domain decomposition algorithm for solving plate and shell problems. *SIAM J. Numer. Anal.*, 35(2):836–867, 1998.
12. J. Mandel. Balancing domain decomposition. *Comm. Numer. Methods Engrg.*, 9(3):233–241, 1993.
13. J. Mandel and P. Krzyżanowski. Robust Balancing Domain Decomposition. August 1999. Presentation at The Fifth US National Congress on Computational Mechanics, University of Colorado at Boulder, CO.
14. B. F. Smith, P. E. Bjørstad, and W. D. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, Cambridge, 1996.
15. M. Vidrascu. Remarks on the implementation of the Generalised Neumann-Neumann algorithm. In C.-H. Lai, P. E. Bjørstad, M. Cross, and O. B. Widlund, editors, *Domain Decomposition Methods in Sciences and Engineering*. DDM.org, 1999. Eleventh International Conference, London, UK.