International Journal of High Speed Computing, Vol. 6, No. 1 (1994) 157–182
© World Scientific Publishing Company

IMPLEMENTATION OF A DOMAIN DECOMPOSITION METHOD WELL-SUITED FOR (MASSIVELY) PARALLEL ARCHITECTURES

P. CIARLET, JR
Commissariat à l'Energie Atomique
Centre d'Etudes de Limeil-Valenton
94195 Villeneuve-Saint-Georges Cedex, France

Received December 9, 1992

ABSTRACT

In this paper, we describe a way of obtaining a domain decomposition method for efficiently solving the linear system which arises from the discretization of symmetric elliptic problems using the finite difference approximation. The resulting linear system is solved iteratively by using the conjugate gradient method together with the capacitance matrix method. The algorithm is designed to be implemented on a parallel computer and to reach high parallelization rates together with nearly optimal speed-ups, i.e., close to the number of processors.

Keywords: Domain decomposition, conjugate gradient and capacitance matrix methods, parallel computers.

1. Introduction. A way of finding efficient algorithms well-suited for parallel implementation is to utilize domain decomposition techniques. These techniques are now widely used; see for example the proceedings of the annual domain decomposition meeting [1–4]. Partitioning the original domain into smaller subdomains, the original problem transforms into smaller connected subproblems on the subdomains.

In order to solve decoupled subproblems instead of connected ones, we will use the capacitance matrix method together with the conjugate gradient method (CG). The combination of the two leads to a method very similar to the preconditioned conjugate gradient method (PCG), in the sense that it minimizes a residual, depending both on the matrix of the original problem and on a preconditioner. From now on, we will use the term PCG method to denote the association of the capacitance matrix and the CG methods. Ideally, the preconditioner should have three properties:

1. it should be easily invertible,
2. it should be spectrally close to the original problem,
3. the subproblems should be decoupled for parallel efficiency.

Remark 1.1. Note that the first two properties ensure that the PCG method is efficient, whereas the third one is of computational nature, meaning that it is unnecessary on a strictly mathematical point of view.

We consider symmetric elliptic problems in a two dimensional domain. The domain is divided into subdomains called boxes, separators and crosspoints labeled in a checkerboard-like way. The white boxes are decoupled. The black boxes, separators and crosspoints can be easily decoupled, thus leading to a preconditioner satisfying at least the third property (see Fig. 1).

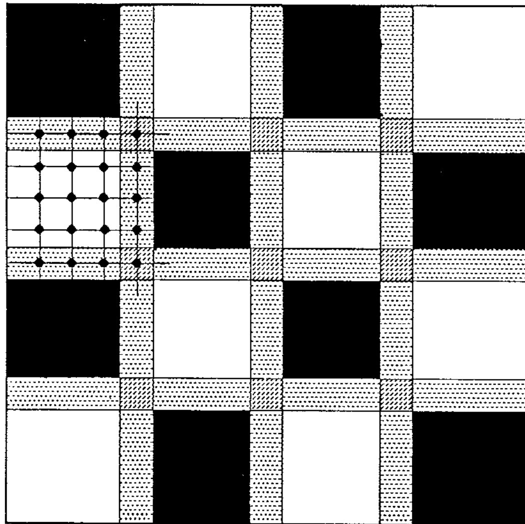

FIG. 1. *Partitioning of* $\Omega =]0,1[\times]0,1[$.

This kind of preconditioning technique with intersecting separators was analyzed in [9] and results of the sequential implementation were presented in [10]. The preconditioner was called the Neumann–Dirichlet preconditioner by the authors. Efficiency of this algorithm confirmed the theoretical predictions [15]. However, parallel implementation of this algorithm [11] showed only modest rate of parallelization.

In order to increase parallel efficiency we decided to introduce two significant modifications: (a) widen the choice of the imposed boundary conditions (used for the preconditioners) to mixed Neumann–Dirichlet–Dirichlet (possibly keeping Neumann–Dirichlet), (b) solve the system related to the second level preconditioner (C_c) iteratively instead of directly. Moreover, in order to get a symmetric preconditioner for all problems (which is not the case in [11]), we added a third modification: (c) we discretized the boundary

conditions on the subdomains with an upwind scheme instead of a centered one.

We implement the method on a *Sequent Symmetry S81* parallel computer. On the one hand, we study the behavior of the method on a mathematical point of view, i.e., the condition number and other indicators. On the other hand, we study the specific computational results, i.e., the speed-up as a function of the number of processors.

We begin with the mathematical theory, that is, the use of the conjugate gradient method together with the capacitance matrix technique. Then we briefly describe the parallel implementation of the code on the *Sequent Symmetry S81*. After that, we study three numerical examples on the unit square and verify that the parallelization rate is very close to 1 in these cases. The problems considered are the model problem, a problem with jumps of the coefficients on the separators and a problem with jumps of the coefficients inside the boxes. We also demonstrate that the family of preconditioners we use is much more efficient than a fully parallel method, the classical Preconditioned Conjugate Gradient method (with the Diagonal preconditioner).

2. The continuous and discrete problems. The aim of the remainder of this paper is to solve numerically the following problem:

$$-div(\mathcal{A}\, grad\, u) = \lambda \text{ in } \Omega, \quad \text{where } \mathcal{A}(x,y) = \begin{pmatrix} a(x,y) & 0 \\ 0 & b(x,y) \end{pmatrix} \tag{2.1}$$

$$u = 0 \text{ on } \partial\Omega . \tag{2.2}$$

where $\Omega =]0,1[\times]0,1[$ and a, b, σ, λ and μ are given functions, a and b being nonnegative over the domain. The coefficients of $\mathcal{A}$ can have jumps over Ω. Problems with other boundary conditions can be handled without difficulty.

We discretize the problem by using the standard finite difference method with a constant meshsize $h = \frac{1}{n+1}$, where n is an integer; n is the number of mesh points in each direction parallel to the x- or y-axis.

Then, by using the usual 5-point stencil approximation, we obtain a linear system with n^2 equations and n^2 unknowns:

$$Ax = f . \tag{2.3}$$

DEFINITION 2.1. Denote by $\text{tridi}_n(a_i, b_i, c_i)$ the tridiagonal matrix

$$\text{tridi}_n(a_i, b_i, c_i) = \begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{pmatrix}.$$

(i) If $a \equiv 1$ and $b \equiv 1$, then $A = \begin{pmatrix} T & -I & & & \\ -I & T & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & T & -I \\ & & & -I & T \end{pmatrix}$ of order n^2

where $T = \text{tridi}_n(-1, 4, -1)$ and I is the identity matrix of order n.

(ii) The unknown x is approximating the node values of u.

(iii) For the numerical experiments, f is computed the following way: once $u(x, y)$ and $\mathcal{A}(x, y)$ are chosen, define $\tilde{x}_{i,j} = u(x_i, y_j)$ and compute $f = A\tilde{x}$. The approximate solution $\tilde{x}$ is known and it is therefore possible to verify that the solution returned by the code is correct.

3. Partitioning. This domain decomposition technique has been previously introduced in [11].

The domain Ω is divided as shown in Fig. 1, with an even number of boxes in each direction. We denote by n_0^2 the total number of boxes.

Remark 3.1. Note that there is one and only one node in each small striped box (the so-called crosspoints).

DEFINITION 3.1. The white boxes are called *α–boxes*. The set of nodes of second, third and fourth types is called *connected β–box*. The black boxes and separators are called *β–boxes*. The striped boxes are called *c–boxes*.

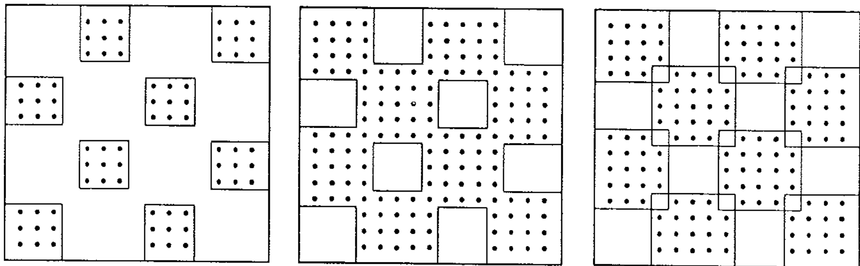

FIG. 2. *α–boxes, connected β–box and β–boxes.*

We consider four types of unknowns and we denote by n_i, J$i \in \{1, 2, 3, 4\}$, the number of nodes of each type:

(1) the nodes in the white boxes, $n_1 = \frac{1}{2}(n + 1 - n_0)^2$,

(2) the nodes in the black boxes, $n_2 = \frac{1}{2}(n + 1 - n_0)^2$,

(3) the nodes in the rectangular boxes called separators, $n_3 = 2(n + 1 - n_0)(n_0 - 1)$,

(4) the nodes in the small striped boxes called crosspoints, $n_4 = (n_0 - 1)^2$.

Now we rewrite A as a 4 by 4 block matrix, the indices corresponding to the types previously defined. That is, we first number the (1)-type nodes, then the (2)-type nodes, the (3)-type nodes and finally the (4)-type nodes, keeping inside each type the original ordering of the nodes. Moreover, from the definition of sets (1) and (2), it is easy to see that there is no relation between any (1)-type node and any (2)-type node arising from the 5-point stencil approximation; thus the block elements A_{12} and A_{21} are equal to 0 (the same holds for elements A_{14}, A_{24}, A_{41} and A_{42}). Finally

$$\tilde{A} = PAP^{-1} = \begin{pmatrix} A_{11} & 0 & A_{13} & 0 \\ 0 & A_{22} & A_{23} & 0 \\ A_{13}^T & A_{23}^T & A_{33} & A_{34} \\ 0 & 0 & A_{34}^T & A_{44} \end{pmatrix}. \tag{3.1}$$

From now on, we will use this numbering scheme and rename A the matrix $\tilde{A}$.

4. The capacitance matrix method. We would like to replace the original problem (2.3) by a set of independent linear problems, related to a matrix close to the original one in a certain way. Finding the solution of these problems will be much easier than trying a straightforward computation of the solution of (2.3).

Let B be a n^2 by n^2 matrix identical to A except the third row blocks; these n_3 rows will be chosen later, close to those of A (see (6.3)). In the remainder of this paper, we speak of B as a **preconditioner** for A. Now define I_3 the identity matrix of the subspace corresponding to the vectors which have null components except at the separator nodes and

$$S = \begin{pmatrix} 0 \\ 0 \\ I_3 \\ 0 \end{pmatrix} \text{ a } n^2 \text{ by } n_3 \text{ matrix}. \tag{4.1}$$

Definition 4.1. $C = S^T AB^{-1}S$ is an n_3 by n_3 matrix called the *capacitance matrix.*

Solving $Ax = f$ amounts to solving the following linear problems (see [5]):

$$By = g, \quad \text{with } g_i = f_i,\ i \in \{1, 2, 4\}, \tag{4.2}$$

$$C\omega = S^T(f - Ay), \tag{4.3}$$

$$Bx = g + S\omega. \tag{4.4}$$

Remark 4.1. In our numerical experiments, we chose $g_3 = 0$.

Remark 4.2. In practise, we do not compute C explicitely and we use the iterative method described in the next paragraph to solve (4.3).

The first and third equations involve solving a B-related linear problem. Our next step consists in replacing the second equation by using a method (derived of the classical CG method) also featuring B-related linear problems. Thus, solving (2.3) finally reduces to solving only B-related linear problems.

5. A preconditioned conjugate gradient method. If C is equal to HG, where both matrices are symmetric positive definite, then we can compute the solution of $C\omega = b$ by using the method described in [14]. In fact, it is exactly the CG method applied to $G^{\frac{1}{2}}HG^{\frac{1}{2}}x = f$ where $x = G^{\frac{1}{2}}\omega$ and $f = G^{\frac{1}{2}}b$.

Remark 5.1. The initial guess is chosen equal to 0 and the stopping criterion is

$$\sqrt{\frac{(r^k, Gr^k)}{(r^0, Gr^0)}} < \epsilon\,,$$

where r^k is the residual at iteration k and ϵ a given nonnegative number.

Remark 5.2. It is possible to modify this method by introducing the matrices A, B and S. The matrices H and G can be replaced by their respective values, i.e., $H = (S^T A^{-1} S)^{-1}$ and $G = S^T B^{-1} S$ (see for example [13]) and the method thus obtained only involves solves of B-related problems. The algorithm to solve $C\omega = b$ is described hereafter.

Let $\epsilon > 0$ be a given nonnegative number.

Initialisation

$$\omega^0 = 0$$
$$r^0 = b$$
$$p^0 = r^0$$
$$Bx = Sr^0$$
$$q^0 = S^T x$$
$$s^0 = q^0$$
$$z^0 = S^T A x$$

Do $k = 0, 1 \cdots$

$$\alpha^k = \frac{(r^k, s^k)}{(q^k, z^k)}$$

$$\omega^{k+1} = \omega^k + \alpha^k p^k$$

$$r^{k+1} = r^k - \alpha^k z^k$$

$$
\begin{aligned}
Bx &= Sr^{k+1} \\
s^{k+1} &= S^T x \\
\beta^k &= \frac{(r^{k+1}, s^{k+1})}{(r^k, s^k)} \\
p^{k+1} &= r^{k+1} + \beta^k p^k \\
q^{k+1} &= s^{k+1} + \beta^k q^k \\
z^{k+1} &= S^T A x + \beta^k z^k
\end{aligned}
$$

until $\dfrac{(r^{k+1}, s^{k+1})}{(r^0, s^0)} < \epsilon^2$.

Remark 5.3. The method we have described for solving $C\omega = b$ is based on the **classical** conjugate gradient method. The term **PCG** method is used as C depends on the choice of B which has to be "close" to A in a sense; as we said earlier, B should be easily invertible and spectrally close to A, so that first the condition number of C is small and the CG method converges rapidly, and secondly the subproblems corresponding to B are decoupled to enable parallel solves.

Remark 5.4. Let us denote by n_{it} the number of iterations of the PCG method. Then the number of times we use the B solver is equal to $4 + n_{it}$.

6. A family of preconditioners.

6.1. Boundary conditions and their discretization. In order to define our preconditioners, let us introduce

$$\mu u + \frac{\partial u}{\partial n} = 0\,, \quad \mu \geq 0\,, \tag{6.1}$$

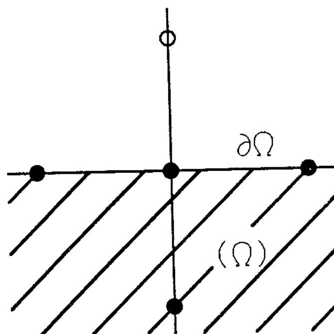

FIG. 3. *An example of a boundary.*

as boundary conditions. Now consider the case of a horizontal boundary as in Fig. 3.

Using an upwind scheme yields

$$\mu\, u_{i,j} + \frac{(u_{i,j+1} - u_{i,j})}{h} = 0\,. \tag{6.2}$$

6.2. First level preconditioners. Instead of using some continuous problems and then discretizing these to get preconditioners, let us skip the continuous part and directly define the matrix preconditioners

$$B = \begin{pmatrix} A_{11} & 0 & A_{13} & 0 \\ 0 & A_{22} & A_{23} & 0 \\ 0 & A_{23}^T & B_{33} & A_{34} \\ 0 & 0 & A_{34}^T & A_{44} \end{pmatrix}, \tag{6.3}$$

with

$$(B_{33})_{i,j} = (A_{33})_{i,j}\,, \quad \forall i \neq j$$
$$(B_{33})_{i,i} = (A_{33})_{i,i} - (1 - \mu\, h)|(A_{13})_{j_i,i}|\,,$$

j_i[1] such that $(A_{13})_{j_i,i} \neq 0$, $\mu \geq 0$.

Remark 6.1. B_{33} is a straightforward consequence of (6.2), by using it together with the 5-point stencil approximation.

Remark 6.2. One interesting feature that can be derived of the expression of $(B_{33})_{i,i}$ is that the relevent quantity which has to be taken into account is μh, not μ. In our numerical experiments, we denote μh by ρ and let it vary from 0 (purely Neumann boundary conditions) to 1, the increment being 0.2.

Recall that $H = (S^T A^{-1} S)^{-1}$ and $G = S^T B^{-1} S$. We have the following lemma:

LEMMA 6.1. *H and G are symmetric positive definite.*

Proof. • A being symmetric positive definite, so is H.

• G is a block diagonal submatrix of the inverse of

$$A_{\beta^c} = \begin{pmatrix} A_{22} & A_{23} & 0 \\ A_{23}^T & B_{33} & A_{34} \\ 0 & A_{34}^T & A_{44} \end{pmatrix},$$

a symmetric positive matrix. If we prove that A_{β^c} is positive-definite, then the proof is complete. Now, the following property holds

$$\forall i,\ (A_{\beta^c})_{i,i} \geq \sum_{j \neq i} |(A_{\beta^c})_{i,j}| \quad \text{and} \quad \exists i_0,\ (A_{\beta^c})_{i_0,i_0} > \sum_{j \neq i_0} |(A_{\beta^c})_{i_0,j}|\,.$$

[1] To understand why j_i exists and is unique, one can look at Fig. 3 and suppose that Ω stands for the (1)-type nodes, $\partial\Omega$ the (3)-type nodes and the complementary of $\bar{\Omega}$ the (2)-type ones. Then it is easy to see that a (3)-type node is linked to exactly one (1)-type node by the 5-point stencil.

Moreover, as it is irreducible (to define it the 5-point stencil is used on the *connected* β*–box*), it is positive-definite (see [8]). ■

Thus, we have found a way of replacing the original linear problem by a number (depending on the convergence rate of the PCG method) of linear problems related to B. Now, if we have the following notations: $x = \begin{pmatrix} x_\alpha \\ x_{\beta^c} \end{pmatrix}$ and $x_{\beta^c} = (\, x_2 \quad x_3 \quad x_4\,)^T$, then solving a problem like $Bx = g$ amounts to

$$A_{\beta^c} x_{\beta^c} = g_{\beta^c} \tag{6.4}$$

$$A_{11} x_\alpha = g_\alpha - A_{13} x_3\,. \tag{6.5}$$

6.3. Second level preconditioners. Ideally, we would like to solve **decoupled** subproblems on both the α*–boxes* and the β*–boxes*, which is not the case with B for the latter, as the unknowns of the *connected* β*–box* are connected to one another by A_{β^c} (it is an irreducible matrix). That is the reason why we introduce a second level preconditioner that decouples the β*–boxes*. The capacitance matrix method is applied once again. This time, we do generate the capacitance matrix C_c explicitly and choose to solve the system by a PCG method (PCGD), the preconditioner being the diagonal of C_c.

DEFINITION 6.1.

$$B_{\beta^c} = \begin{pmatrix} A_{22} & A_{23} & 0 \\ A_{23}^T & B_{33} & A_{34} \\ 0 & 0 & I_4 \end{pmatrix} = \begin{pmatrix} A_\beta & \begin{pmatrix} 0 \\ A_{34} \end{pmatrix} \\ (0 \quad 0) & I_4 \end{pmatrix}, \tag{6.6}$$

$$\text{where } A_\beta = \begin{pmatrix} A_{22} & A_{23} \\ A_{23}^T & B_{33} \end{pmatrix}.$$

Therefore, the second level capacitance matrix needs to be defined only on the crosspoints, that is

$$S_{\beta^c} = \begin{pmatrix} 0 \\ 0 \\ I_4 \end{pmatrix}, \quad \text{and } C_c = S_{\beta^c}^T A_{\beta^c} B_{\beta^c}^{-1} S_{\beta^c}\,.$$

LEMMA 6.2. $C_c = A_{44} - A_{34}^T (B_{33} - A_{23}^T A_{22}^{-1} A_{23})^{-1} A_{34}$

Proof.

$$B_{\beta^c}^{-1} = \begin{pmatrix} A_\beta^{-1} & -A_\beta^{-1} \begin{pmatrix} 0 \\ A_{34} \end{pmatrix} \\ (0 \quad 0) & I_4 \end{pmatrix}$$

$$= \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & -(B_{33} - A_{23}^T A_{22}^{-1} A_{23})^{-1} A_{34} \\ \cdot & \cdot & I_4 \end{pmatrix}.$$

Finally, $C_c = S_{\beta^c}^T A_{\beta^c} B_{\beta^c}^{-1} S_{\beta^c} = A_{44} - A_{34}^T (B_{33} - A_{23}^T A_{22}^{-1} A_{23})^{-1} A_{34}$. ■

It can be shown that the internal structure of this matrix is that of a matrix derived of a 9-point stencil approximation (on the crosspoints), see for example [7]. Now, let $x_\beta = (x_2 \quad x_3)^T$ and $x_c = x_4$. Solving $Bx = g$ (i.e., (6.4–5)) is equivalent to

$$
\begin{aligned}
A_\beta v_\beta &= g_\beta \\
C_c \omega_c &= g_c - A_{34}^T v_3 - A_{44} v_c \\
A_\beta x_\beta &= g_\beta - \begin{pmatrix} 0 \\ A_{34}\omega_c \end{pmatrix} \\
x_c &= \omega_c \\
A_{11} x_\alpha &= g_\alpha - A_{13} x_3 .
\end{aligned}
$$

Finding the solution of $Bx = g$ amounts to solving two problems on the β–*boxes*, one problem on the α–*boxes* and one problem on the crosspoints. The main advantage of this reformulation is that the problems on both the α–*boxes* or the β–*boxes* can be solved in parallel.

6.4. How to solve the decoupled subproblems in terms of boxes. Solving the subproblems in the α–*boxes* is classical, since these boxes are squares.

This is different concerning the β–*boxes*, for there are of the three types shown in Fig. 4 (after a possible rotation):

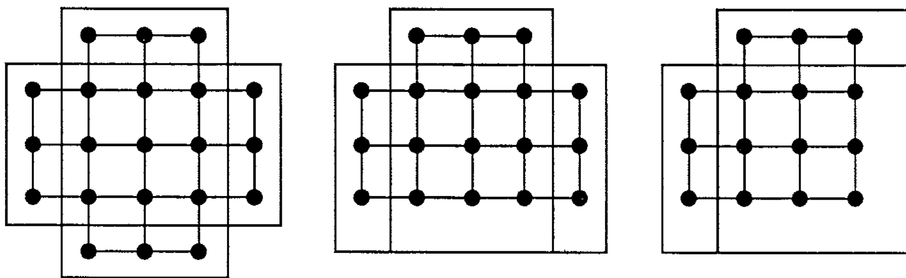

FIG. 4. *Different types of β–boxes.*

Then one can think of two methods to work out this difficulty. The first option consists in adding fictitious nodes so that the subdomains become squares; thus one gets bigger fictitious linear subproblems, but on the other hand all these problems are of the same size and no renumbering needs to be made.

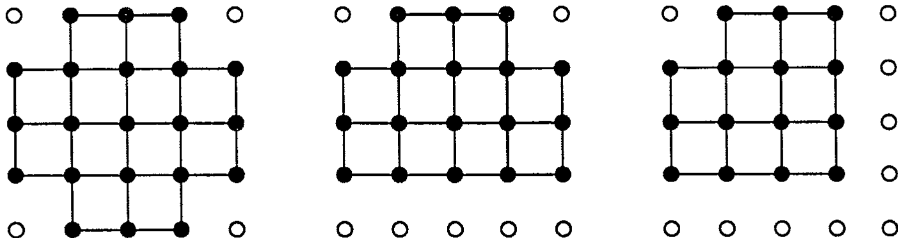

FIG. 5. *Adding fictitious nodes.*

The matrices associated with the fictitious subproblems are build the following way: keep the original relations between the real nodes and add relations of the type $x_f = 0$ for all $f \in F$, F denoting the set of fictitious nodes.

The second option consists in keeping the original subdomains and then renumbering the nodes in each subdomain. This time there are three different sizes of subproblems, all lower or equal to the size of the fictitious linear problems of the first option.

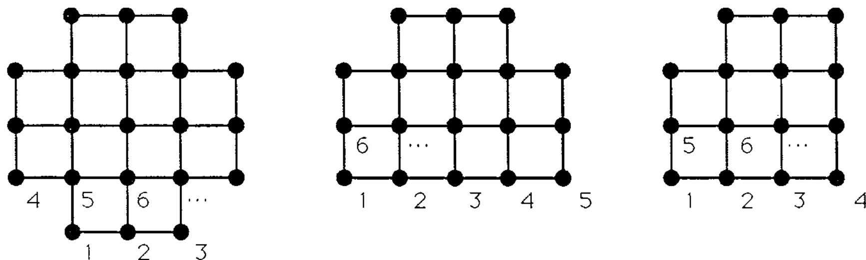

FIG. 6. *Renumbering the nodes.*

We have chosen the second option, although the first one is also feasible.

To solve the linear subproblems we used the Cholesky method by computing the accompanying decomposition once and for all at the beginning of the code and then using it afterwards. The subroutines correspond to the SPBFA and SPBSL subroutines of the LINPACK library.

7. Parallel implementation.

7.1. The computer. The program was run on a *Sequent Symmetry S81* parallel computer. This system can include up to thirty 32-bit CPUs, composed of an Intel 80386 and a floating-point support Intel 80387 together with an optional floating-point accelerator, the WTL 1167 processor.

At the time we went through our simulations, the system included twenty CPUs together with their floating-point accelerators.

Note that the *Symmetry* system is a multiprocessor whose operating system allows processors to work on a **shared memory** basis.

7.2. *KAP* preprocessing. In order to get a parallel version of the code, we used the *KAP* Fortran preprocessor (see [12]). The *KAP* optimizer identifies potential parallelism to a certain extent at the loop level and restructures the source code for parallel execution.

The optimizer converts the code using explicit syntax to represent the parallelism. It inserts the *Sequent* directives necessary to the implementation of the parallel code.

Unfortunately, we had to rewrite the most important subroutines of the code, i.e., the solvers, because the optimizer failed to parallelize these subroutines correctly. So we included some *Sequent* directives in the code before preprocessing.

7.3. Parallel programming. The parallel version of the code is achieved in three steps:

(i) we have first to identify the subroutines or parts of subroutine which have to be run sequentially.

In this case, we can prevent a possible failure of the optimizer by surrounding the concerned lines of the code with two directives (*C$ NOCONCURRENTIZE* and *C$ CONCURRENTIZE*). In this way, the optimizer goes on without modifying the concerned statements.

(ii) Then we utilize the *KAP* optimizer to parallelize the simplest subroutines or loops. By using the optimizer we can thus avoid fastidious writings.

(iii) Finally we parallelize the A_{11} and A_β solvers, i.e., the most critical subroutines, by adding *Sequent* directives.

Remark 7.1. Note that in our case, approximately 70% of the code are directly parallelized by the optimizer.

Remark 7.2. We have to solve subproblems in the β*–boxes* in parallel and then solving subproblems in the α*–boxes* in parallel. Both numbers of β*–boxes* and α*–boxes* are equal to $\frac{1}{2}n_0^2$. Thus, if p_{max} is the greatest number of processors available, n_0^2 has to be a multiple of $2p_{max}$ (if $p_{max} = 16$, then n_0^2 has to be a multiple of 64).

8. Problems to be solved. We choose to solve numerically the following three problems. The first one is the well-known Poisson equation, that is

Problem #1

$$a(x,\, y) = 1 \forall (x,\, y) \in \Omega$$
$$b(x,\, y) = 1 \forall (x,\, y) \in \Omega .$$
Dirichlet boundary conditions on $\partial\Omega$.

For the second problem, we use a strongly discontinuous coefficient in one direction.

Problem #2

$$a(x,\, y) = \begin{cases} 10^3 & \text{if } \frac{1}{4} \leq x \leq \frac{3}{4},\, 0 \leq y \leq 1 \\ 1 & \text{elsewhere} \end{cases}$$
$$(x,\, y) \;= 1 \;\forall (x,\, y) \in \Omega .$$
Dirichlet boundary conditions on $\partial\Omega$.

3	1	0.1
30	0.3	10
0.01	0.03	100

a(x,y)

0.3	3	0.03
10	0.01	100
30	1	0.1

b(x,y)

FIG. 7. *The coefficients of Problem #3.*

For the third problem, still with Dirichlet boundary conditions, we choose strongly discontinuous coefficients in both directions, with jumps occurring inside the subdomains in our numerical experiments.

In order to be able to compare the numerical results with a solution, we take

$$u(x,\, y) = x(1 - x)y(1 - y)\mathrm{e}^{-xy}$$

as the true solution. Then we compute the right hand side f of the linear problem using (iii) of paragraph 2.

Remark 8.1. Since the model problem (or Poisson equation) has constant coefficients, seven different types of boxes appear for the α–*boxes* on the one hand and seven more for the β–*boxes* on the other hand, independently of the number of boxes. Hence, one needs to compute and store only fourteen different Cholesky decompositions. Yet our choice is to compute and store all decompositions, as it is necessary for the other two problems.

9. Numerical results. As we have seen before in paragraph 5, we utilize a PCG method to solve the linear problems involving a capacitance matrix. We mentioned at the beginning of the algorithm that we have to choose a nonnegative number ϵ; we set it at 10^{-6} for solving both the C-problem and the C_c-problems (PCGD method).

Remark 9.1. In order to get a sufficient accuracy, we carried out the computations using the *Sequent* double precision type (i.e., 64 bits) for all the real variables of the code.

Once the problem is chosen, there are two parameters which may vary: the meshsize h and the number of boxes. So we tested the code for different values of (n, n_0), where $h = \frac{1}{n+1}$ and n_0 is the number of boxes in a direction parallel to the x- or y-axis. In the first subparagraph, the couple (n, n_0) is such that $n + 1 = 8n_0$. In the second subparagraph, n is set at 127 and the value of n_0 varies. The subject of the third subparagraph is the average number of iterations of the PCGD method applied to C_c.

In the first section of subparagraphs 9.1 and 9.2, we study the convergence and the efficiency of the algorithm from a mathematical point of view by taking a closer look at the number of iterations and at the condition numbers. In appendix A, we briefly consider the choice of an error indicator. In the second section, we take a closer look at the parallel efficiency of the code by studying the CPU time as a function of the number of processors and then the speed-up factor.

9.1. Boxes of a given size. Let the couple (n, n_0) take the following values:

$$(63, 8)\,, \quad (127, 16)\,, \quad (191, 24)\,, \quad \text{J where } n + 1 = 8n_0\,.$$

9.1.1. Number of iterations — Condition numbers. We now study the efficiency of the algorithm.

First of all, let us consider the number of iterations as a function of (n, n_0), necessary to satisfy the stopping condition for the PCG method concerning C.

The mostly interesting feature of these pictures is that the number of iterations increases linearly when ρ is not equal to zero, whereas the growth seems bounded when $\rho = 0$. This can be further investigated if ones consider the condition number of the matrix C, denoted by $\kappa(C)$. The lowest and greatest eigenvalues are computed by using the Lanczos factorisation (of C), deduced from the coefficients α^k and β^k of the PCG method. Again, we distinguish two cases, i.e., whether ρ is null or not.

When ρ is not equal to 0, then the condition number grows as $O(n^2)$. The main term of the condition number varies from $0.005n^2$ to $0.01n^2$ for Problem #1 and ρ between 0.2 and 1.0. With the same bounds for ρ, it goes from $2n^2$ to $4n^2$ for Problem #2 and $0.02n^2$ to $0.04n^2$ for Problem #3.

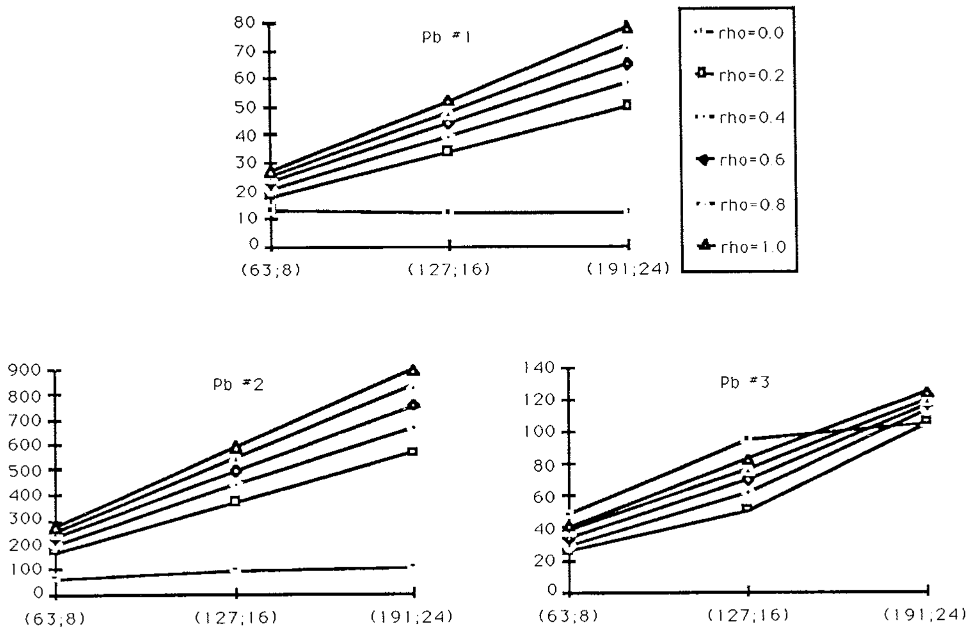

FIG. 8. *Number of iterations.*

Remark 9.2. Let us compare these condition numbers with $\kappa(A)$ for the three problems. The behaviour of the condition number is $0.41n^2$ for Problem #1, $190n^2$ for Problem #2 and $40n^2$ for Problem #3. The average gain is about 60 for Problem #1, 70 for Problem #2 and 800 for Problem #3.

On the contrary, if $\rho = 0$, the condition number is bounded for the three problems. For this particular case, we have added a numerical test with $n = 255$ and $n_0 = 32$ to emphasize this trend. We enter the results in the next table.

TABLE 1

The condition number $\kappa(C)$ with Neumann boundary conditions.

(n, n_0)	$(63, 8)$	$(127, 16)$	$(191, 24)$	$(255, 32)$
Problem #1	4.9	5.0	5.0	5.0
Problem #2	510	520	520	520
Problem #3	480	870	990	1000

CONJECTURE. For a given ratio $\frac{n+1}{n_0}$, the condition numbers verify $\kappa(C) = O(1)$ for $\rho = 0$ and $\kappa(C) = O(n^2)$ for a different ρ.

In appendix B, we are interested in the asymptotic rate of convergence $R_\infty(C)$, which is inversely proportional to the number of iterations.

9.1.2. CPU times — Speed-up factors. When the numerical tests were made, the computer was dedicated. In the first place, let us study a particular case. We list the CPU times in a table with two entries for Problem #3 and $\rho = 0.2$. Horizontally, we give the value of $(n,\, n_0)$ and, vertically, the number of processors p: 1, 2, 4, 8 or 16. The theoretical clock precision is 1 μ's, but different parallel executions of the same program produce different CPU times. So we keep the mean value for each case, which gives a precision of a few thousandths of the total CPU time. The results are given in seconds.

TABLE 2

CPU times: an example.

$(n,\, n_0)$	$(63,8)$	$(127,16)$	$(191,24)$
$p=1$	58.5	399.9	1731.1
$p=2$	30.0	202.9	874.4
$p=4$	15.7	103.1	441.4
$p=8$	8.6	54.1	228.2
$p=16$	5.5	30.7	126.3

TABLE 3

The speed-up factor $\tau(p)$.

$(n,\, n_0)$	$(63,8)$	$(127,16)$	$(191,24)$
$p=1$	1.0	1.0	1.0
$p=2$	2.0	2.0	2.0
$p=4$	3.7	3.9	3.9
$p=8$	6.8	7.4	7.6
$p=16$	10.6	13.0	13.7

Remark 9.3. It should be noted that the CPU times mentioned above are global times, meaning in particular that the initialization step is included. When the initialization is performed, a part of it has to be done sequentially, therefore worsening the speed-up factor. For example, for $n = 127$ and

$n_0 = 16$, the initialization takes 56.3 seconds with one processor and 5.0 seconds with sixteen processors. The induced speed-up factor is 11.3 whereas for the remainder of the execution time (respectively 343.5 and 25.8 seconds), the speed-up factor is 13.3.

From the values shown in this table, we can derive the speed-up factor $\tau(p)$, i.e., the ratio of the monoprocessor CPU time to the p-processor CPU time, $p \in \{1, 2, 4, 8, 16\}$.

The results concerning to speed-ups for the other two problems are very close and do not depend on the value of ρ. Now we are going to utilize Amdahl's law in order to determine the parallelization rate. Denote by $t(p)$ the total CPU time, p being the number of processors. We have the following equality: $t(p) = t_s + t_{//}(p)$, where t_s is the time spent in the sequential parts of the code and $t_{//}(p)$ is the time spent in the parallelized parts of the code. Obviously, there exists a constant t_0 such that $t_{//}(p) = \frac{t_0}{p}$. Let α be the parallelization rate (α being 0 when the code is totally sequential and 1 when it is fully parallel); we have $t_s = (1-\alpha)t(1)$ and $t_0 = \alpha t(1)$. From the last three equalities, we deduce Amdahl's law, giving the parallelization rate α as a function of the number of processors p and the speed-up factor $\tau(p)$:

$$\alpha = \frac{1 - \dfrac{1}{\tau(p)}}{1 - \dfrac{1}{p}} .$$

So, for a given p, we get α as a function of $\tau(p)$. In Table 4, we collect the parallelization rates corresponding to the CPU times obtained with sixteen processors with respect to the CPU times obtained with one processor.

TABLE 4

Parallelization rates.

(n, n_0)	$(63, 8)$	$(127, 16)$	$(191, 24)$
α_{16} (%)	96.6	98.5	98.9

The parallelization rates always remain very close to 1.

9.2. Varying boxsize. For a given n, let us study the effect of changing the value of n_0. Our choice is $n = 127$ and $n_0 \in \{8, 16, 32\}$. Thus, we obtain three different preconditioners for each problem and a fixed value of ρ (meshsize $h = \frac{1}{128}$). In terms of errors (of the computed solution with respect to the exact one), the three preconditioners behave quite similarly and the results of appendix A are still valid.

9.2.1. Condition numbers – number of iterations. In opposition to the previous subparagraph, let us consider first the condition numbers. One can observe two different kinds of behaviour, depending on whether or not ρ is equal to 0. If ρ is equal to 0, then the condition number (called κ^0) decreases as the number of boxes grows, except for Problem #3. For a different ρ, the condition number (called $\kappa^{\neq}$) increases. In the next two tables, the case of 16 boxes ($n_0 = 4$) is added as the CPU times are not our field of interest at the moment.

TABLE 5

The condition number κ^0.

(n, n_0)	(127,4)	(127,8)	(127,16)	(127,32)
Problem #1	9.7	7.2	4.9	3.2
Problem #2	670	650	520	400
Problem #3	23	540	870	960

It is hard to derive a conclusion from what goes on, because the behaviour depends on the problem. In order to be able to draw a general rule, we have tried other problems, some with jumps at the boundary of the boxes, and some with jumps inside the boxes. For all the problems with jumps on the boundary we have tested so far, the condition number κ^0 decreases when n_0 is sufficiently large. The behaviour is more erratic for the problems with jumps inside the boxes.

Now, let us study the case of a nonnegative ρ. This time, our choice is Problem #2 and $\rho = 0.4$.

TABLE 6

An example of the condition number $\kappa^{\neq}$.

(n, n_0)	(127,4)	(127,8)	(127,16)	(127,32)
Problem #2	9.7×10^3	1.8×10^4	3.4×10^4	6.3×10^4

$\kappa^{\neq}$ grows, in this particular case, but only very slowly. The growth of $\kappa^{\neq}$ is bounded above by $O(n_0)$, i.e., $\kappa^{\neq}$ is a function of n_0 that can be bounded above by linear functions of n_0. The same conclusion holds true for other problems with jumps at the boundary of the boxes, but again, no general behaviour can be derived for problems with jumps inside the boxes.

Let us consider now the number of iterations when n_0 varies.

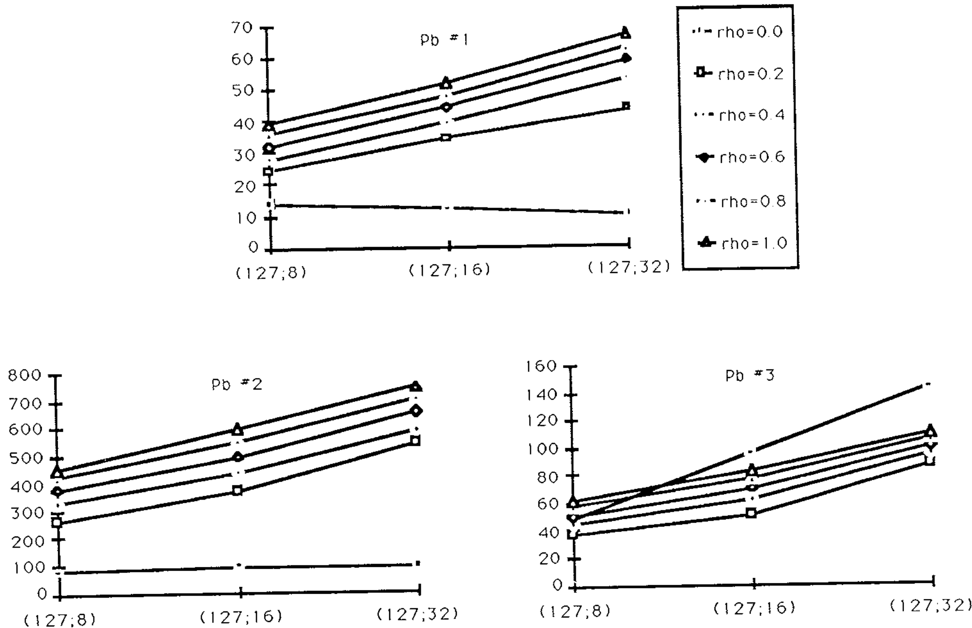

FIG. 9. *Number of iterations.*

The trend concerning the number of iterations follows more or less that of the condition number. For Problems #1 and #2 and $\rho = 0$, the number of iterations decreases when n_0 grows. In the remaining cases, it is a growing function of n_0.

9.2.2. CPU times – Speed-up factors. Let us give first the CPU times obtained for our test problem, Problem #3 ($\rho = 0.2$). The number of processors equals either one or sixteen.

TABLE 7

CPU times: another example.

(n, n_0)	(127,8)	(127,16)	(127,32)
$p = 1$	468.2	399.9	879.0
$p = 16$	34.5	30.7	67.7

Remark 9.4. There is an apparent contradiction between the number of iterations and the CPU times. This contradiction can be solved if two factors are taken into account. First, finding the solution of problems with matrix C_c. The number of unknowns (equal to n_4) grows as n_0^2. So if the PCGD solver is inefficient, the time spent in the subroutine corresponding to this

solver can increase very fast. This will be further investigated in the next subparagraph. The second factor is the size of the subproblems on the boxes. The number of floating operations involved to solve a problem of matrix A_{11} or A_β once their Cholesky decomposition is computed is $k\frac{n^3}{n_0}$, k being a constant number. Thus, the number of floating operations is reduced by a factor of two when n_0 goes from eight to 16 (or 16 to 32). The combination of these two factors, the first one being significant for n_0 big and the second one for n_0 small, is an a posteriori justification of the previous results

We concentrate now on the speed-up factors and the parallelization rates for sixteen processors.

TABLE 8

$\tau(16)$ and α_{16}.

(n, n_0)	(127,8)	(127,16)	(127,32)
$\tau(16)$	13.6	13.0	13.0
α_{16} (%)	98.8	98.5	98.5

The figures show that the code is almost fully parallel.

9.3. The PCGD method applied to C_c. The quantity of interest in this subparagraph is the average number of iterations needed to solve one C_c problem. This average number is denoted by n_c. The results concerning the second level PCG method are divided in three parts, each of these about a family of (n, n_0): a fixed number of nodes by box: $n + 1 = 8n_0$, a fixed meshsize: $n = 127$ and a fixed number of boxes: $n_0 = 16$. Before we enumerate the numerical results, it is interesting to keep in mind that the pattern of matrix C_c follows that of a 9-point stencil approximation. This matrix has been devised to transmit some information between crosspoints. Thus, it is natural to hope that it is not too diagonally dominant. A by-product of this fact is that if matrix C_c has been well designed, then the PCG method with the diagonal preconditioner will not work very efficiently.

Remark 9.5. In our numerical tests, one interesting feature is that n_c is always a decreasing function of ρ, with a sharp maximum for $\rho = 0$. To emphasize this general remark, let us take an example: $n = 191$ and $n_0 = 24$.

9.3.1. The case $n + 1 = 8n_0$. If $\rho = 0$, then the growth of n_c is a linear function of n. The increasing rate is about $\frac{1}{4}$ for Problem #1, two for Problem #2 and $\frac{1}{3}$ for Problem #3. For a different ρ, n_c is a constant number for Problem #1 and, again, increases linearly for the other problems (a rate of $\frac{1}{20}$ for Problem #2 and $\frac{1}{10}$ for Problem #3).

TABLE 9

$n_c(\rho)$: an example.

Problem	Pb #1	Pb #2	Pb #3
$\rho = 0$	45	294	85
$\rho \geq 0.2$	$2 \leq n_c \leq 5$	$13 \leq n_c \leq 21$	$19 \leq n_c \leq 24$

9.3.2. The case $n = 127$. The growth is linear (as a function of n_0) for any value of ρ and any problem considered. The rate goes approximatively from two to ten when $\rho = 0$. For other ρ's, it is less than one.

9.3.3. A constant number of boxes. The number of boxes kept here is 256 ($n_0 = 16$). If ρ is equal to 0, an interesting feature is that n_c is practically a constant number. In other cases, it even decreases (for a given ρ) when n grows, although very slowly.

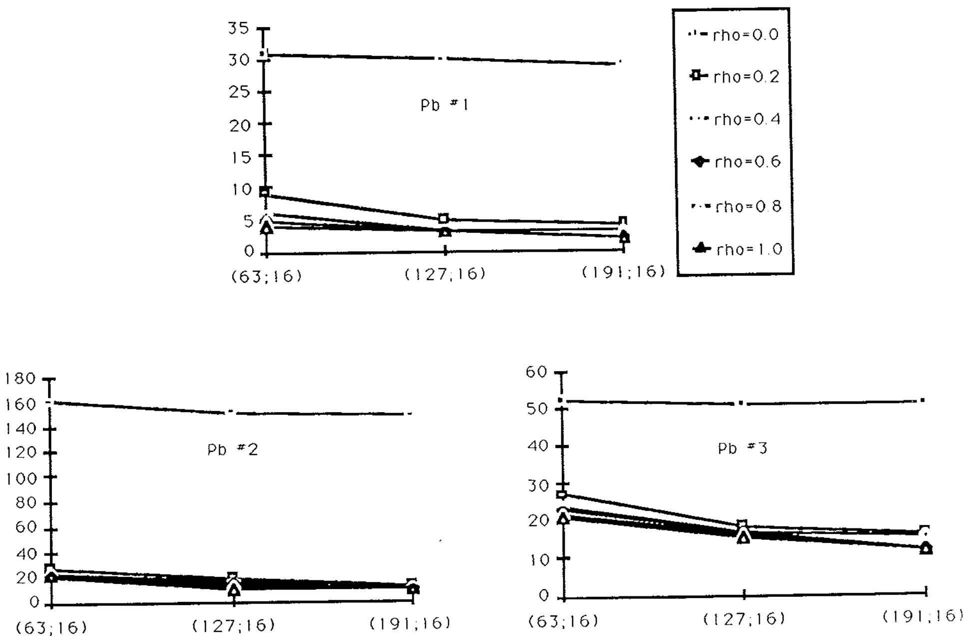

FIG. 10. *n_c for 256 boxes.*

It is therefore likely that matrix C_c goes towards a limit as the meshsize h goes to 0. The PCGD method applied to C_c-related problems leads to a constant number of iterations.

9.3.4. A few more remarks. The choice of the PCGD method may seem quite odd to solve the C_c-related problems. Let us probe a few other possibilities we have tested. For example, one can argue that this iterative method may not be as efficient as a direct method, such as a (direct) solver for which the inverse of C_c is explicitly computed, or one using the Cholesky decomposition of C_c.

There are two advantages in using the first option. First, the method is fully parallel (even at the initialization step when the inverse is computed). Secondly, if n_4 (number of crosspoints) is not too big, it is also very efficient in terms of CPU time. The main disadvantage relies on its memory requirements and on the initialization step. The storage requirements are equal to n_4^2, that is $(n_0-1)^4$. And the initialization step becomes rapidly prohibitive, if one recalls that the Cholesky decomposition of C_c has to be computed and then they are $(n_0-1)^2$ systems to invert (although in parallel) to get C_c^{-1}. Nevertheless, this option is the fastest one for $n_0 \leq 16$ and $\rho = 0$ (because n_c is big in that case), but do not forget the storage requirements in $O(n_0^4)$.

On the other hand, the Cholesky decomposition requires a storage in $O(n_0^3)$ as the halfbandwidth of C_c is $n_0 + 1$. But its main disadvantage is that it is sequential. Actually, this gives the worst CPU times.

One may also have chosen the PCG method with another preconditioner. So, we have also tested a nine point ILU factorization of C_c. This is a much better preconditioner than its diagonal, but on the other hand it is sequential. So, even for $\rho = 0$ (n_c big), the diagonal preconditioner is slightly more efficient than the nine point ILU one.

10. Comparison with the PCG method with a diagonal preconditioner. It is natural to compare the methods implemented in this paper with some well tried parallel method. Our choice is the PCGD method. The meshsize h is either equal to $\frac{1}{64}$, $\frac{1}{128}$ or $\frac{1}{192}$. In the next table, we summarize the results in terms of CPU times for our three problems and sixteen processors.

TABLE 10

CPU times: the PCGD method.

Problem	Pb #1	Pb #2	Pb #3
$h=\frac{1}{64}$	3.5	32.9	5.2
$h=\frac{1}{128}$	26.9	294.1	43.5
$h=\frac{1}{192}$	98.2	1106.9	161.5

First, let us now consider the case of a constant number of nodes by box. As usual, take $n+1 = 8n_0$. In order to compare the PCGD method

with the best preconditioner in each case, our choice for ρ is 0 for Problems #1 and #2, 0.2 for Problem #3. This can be explained in the following way for the latter: although the condition number $\kappa(C)$ is bounded above when $\rho = 0$, the problem on the crosspoints (matrix C_c) is much harder to solve in this case; this more than compensates for $\rho = 0.2$ for which the respective behaviours are quite the opposite. In the next figure, we expose the ratios of the CPU time needed to solve the problem with our preconditioner to the CPU time required when using the PCGD method.

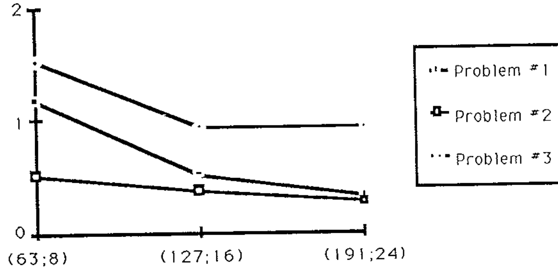

FIG. 11. *Ratios of CPU times: a constant number of nodes by box.*

Secondly, let the number of boxes be set: $n_0 = 16$, that is a total of 256 boxes. The values of ρ are taken as in the previous case. Again, the figure pictures the ratios of CPU times.

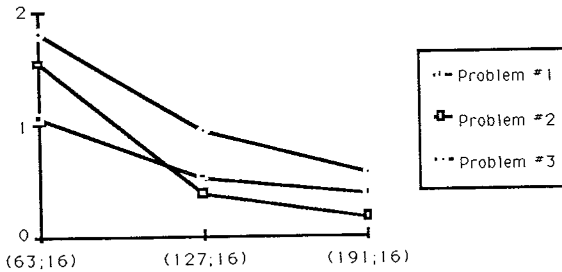

FIG. 12. *Ratio of CPU times: 256 boxes.*

The general trend is that the ratios decrease quite sharply, being less than one as soon as n is equal to 127 (a meshsize of $\frac{1}{128}$). Moreover, the gain is important for at least two of the Problems, as the amount of time needed to solve Problem #1 is reduced by a factor of three (that is the inverse of the ratios previously defined) and more than a factor of four to solve Problem #2. Finally, concerning Problem #3, the gains are somewhat slimmer in the first case (the factor is only of 1.1) and more interesting for the second case (a factor of two).

11. Conclusion. We gave a detailed account of what to expect when using the partitioning presented in paragraph 3. The partitioning (with n_0^2 boxes) and the two level PCG method are easily implementable. Moreover, the code is highly parallel, due to the partitioning. Thus, when the program is implemented on a shared memory computer with a *Symmetry S81*-like internal structure, one can hopefully reach high parallelization rates in any case. With regard to the family of preconditioners, we saw that, when the ratio of the number of meshpoints to the number of boxes is a constant, $\kappa(C)$ (the condition number of the preconditioned system) is bounded above if the boundary conditions are purely Neumann, whereas it is $O(n^2)$ in other cases (mixed boundary conditions). On the other hand, the PCG method applied to solve the second level problems (matrix C_c defined on the crosspoints) is much more efficient when applied to preconditioners with mixed boundary conditions. That is why the best preconditioner is sometimes obtained with mixed boundary conditions. Finally, and for the three problems we studied, the preconditioners performed better than the PCG method with the diagonal of the matrix as a preconditioner.

Acknowledgements. The author wants to thank both Dr. Gérard Meurant for his advice and his interest in this paper and Prof. Wlodzimierz Proskurowski for his helpful remarks.

REFERENCES

[1] R. Glowinski, G. Golub, G. Meurant and J. Périaux, eds., *Domain decomposition methods for partial differential equations*, Siam (1988).

[2] T. F. Chan, R. Glowinski, J. Périaux and O. Widlund, eds., *Domain decomposition methods for partial differential equations*, Siam (1989).

[3] T. F. Chan, R. Glowinski, J. Périaux and O. Widlund, eds., *Domain decomposition methods for partial differential equations*, Siam (1990).

[4] R. Glowinski, Y. A. Kuznetsov, G. Meurant, J. Périaux and O. Widlund, eds., *Domain decomposition methods for partial differential equations*, Siam (1991).

[5] B. Buzbee, F. Dorr, A. George and G. Golub, *The direct solution of the discrete Poisson equation in irregular regions*, Siam J. Numer. Anal., 8, No. 4 (1971) pp. 722–736.

[6] Tony F. Chan and Howard C. Elman, *Fourier analysis of iterative methods for elliptic problems*, Siam Rev., 31, No. 1 (Mar. 1989), pp. 20–49.

[7] P. Ciarlet, Jr, *Méthodes itératives de résolution de problèmes elliptiques en 2D adaptées à des architectures (massivement) parallèles*, CEA-N 2688.

[8] P. Ciarlet and B. M. et J.-M. Thomas, *Exercices d'analyse numérique matricielle et d'optimisation avec solutions*, Masson (1986).

[9] M. Dryja, W. Proskurowski and O. Widlund, *Method of domain decomposition with crosspoints for elliptic finite element problems*, in Proc. of the Int. Symp. on Optimal Algorithms (Apr. 1986).

[10] M. Dryja, W. Proskurowski and O. Widlund, *Numerical experiments and implementation of a domain decomposition method with crosspoints for the model problem*, in Advances in computer methods for partial differential equations (1987).

[11] M. HAGHOO AND W. PROSKUROWSKI, *Parallel implementation of a domain decomposition method*, Technical Report, CRI 88–06 (1988).
[12] KUCK AND ASSOCIATES, INC., *KAP/Sequent User's Guide Version 6*, KAI (1988).
[13] W. PROSKUROWSKI, *Remarks on spectral equivalence of certain discrete operators*, in [2], pp. 103–113.
[14] W. PROSKUROWSKI AND O. WIDLUND, *A finite element-capacitance matrix method for the Neumann problem for Laplace's equation*, Siam J. Sci. Statist. Comput., 1, No. 4 (Dec. 1980), pp. 410–425.
[15] O. B. WIDLUND, *Iterative substructuring methods: algorithms and theory for elliptic problems in the plane*, in [1], pp. 113–128.

Appendix A. One can choose three error indicators a priori, corresponding respectively to the three norms l^1, l^2 or l^∞. At that point, it is not clear which is the most well-suited norm to study the error. In order to level this difficulty out, let us proceed as follows. Recall that the exact solution is defined in the paragraph 8. The code computes the values of the components of x which should be (ideally) the values of u at the nodes of the mesh (see the definition of the right-hand side).

DEFINITION A.1. Let

$$\tilde{u}_{k,l} = u\left(\frac{k}{n+1}, \frac{l}{n+1}\right), \quad \forall (k, l) \in \{1, \dots, n\} \times \{1, \dots, n\} \quad \text{and}$$
$$\epsilon_{k,l} = |\tilde{u}_{k,l} - x_i|, \quad \forall (k, l) \in \{1, \dots, n\} \times \{1, \dots, n\}, \quad i = k + (n-1)j\,.$$

DEFINITION A.2. Let

$$\|v\|_1 = \sum_{k,l=1}^{n} |v_{k,l}|, \; \|v\|_2 = \sqrt{\sum_{k,l=1}^{n} v_{k,l}^2} \quad \text{and} \quad \|v\|_\infty = \max_{1 \le k,l \le n} |v_{k,l}|\,.$$

DEFINITION A.3. Define the following indicators

$$\epsilon_1 = \frac{\|\epsilon\|_1}{\|\tilde{u}\|_1}, \quad \epsilon_2 = \frac{\|\epsilon\|_2}{\|\tilde{u}\|_2} \quad \text{and} \quad \epsilon_\infty = \frac{\|\epsilon\|_\infty}{\|\tilde{u}\|_\infty}\,.$$

Now that these definitions are made, we can compare the three error indicators. Actually, the indicators are very close to one another in all occurences. The error is of the order of 10^{-8} for Problem #1, 10^{-6} for Problem #2 and 10^{-5} for Problem #3 and does not depend much on the preconditioner used nor on the meshsize. That is the reason why each one of them is as good as the other one and the choice of any of these indicators is free.

Appendix B. In this appendix, we estimate the inverse of the number of iterations of the PCG method necessary to solve the C-related problem. It

is proportional to $R_\infty(C)$, which is called the asymptotic rate of convergence (see [6]). It is defined as

$$R_\infty(C) = -\ln\left(\frac{\sqrt{\kappa(C)}-1}{\sqrt{\kappa(C)}+1}\right).$$

Two cases may occur, whether $\kappa(C)$ is bounded or not. In the first case, $R_\infty(C)$ can be clearly provided a nonnegative lower bound. The number of iterations is therefore bounded (with an upper bound for each problem) when $\rho = 0$ for any value of (n, n_0) (with $n+1 = 8n_0$). On the contrary, if the condition number goes to infinity, then we have

$$\begin{aligned}\frac{\sqrt{\kappa(C)}-1}{\sqrt{\kappa(C)}+1} &= \frac{1-\frac{1}{\sqrt{\kappa(C)}}}{1+\frac{1}{\sqrt{\kappa(C)}}} = \left(1-\frac{1}{\sqrt{\kappa(C)}}\right)\left(1-\frac{1}{\sqrt{\kappa(C)}}+o\left(\frac{1}{\sqrt{\kappa(C)}}\right)\right)\\ &= 1-\frac{2}{\sqrt{\kappa(C)}}+o\left(\frac{1}{\sqrt{\kappa(C)}}\right),\end{aligned}$$

where $g \overset{x_0}{=} o(f)$ means that $\lim_{x\to x_0} \frac{g(x)}{f(x)} = 0$.

Thus, when $\kappa(C)$ is not bounded,

$$R_\infty(C) \sim \frac{2}{\sqrt{\kappa(C)}}.$$

We know that, when $\rho \neq 0$, the condition number is equivalent to $\eta(\rho)n^2$, where $\eta(\rho)$ depends only on ρ (and the ratio $\frac{n+1}{n_0}$). Moreover $h = \frac{1}{n+1}$, so

$$R_\infty(C) \sim \frac{2}{\sqrt{\eta(\rho)}}h.$$