



ELSEVIER

Parallel Computing 21 (1995) 1725–1748

PARALLEL
COMPUTING

An efficient parallel discrete PDE solver

Y. Notay^{1,*}

*Service de Métrologie Nucléaire Université Libre de Bruxelles (C.P. 165) 50, Av. F.D. Roosevelt,
B-1050 Brussels, Belgium*

Received 8 July 1994; revised: 8 May 1995

Abstract

We present a parallel iterative solver for discrete second order elliptic PDEs. It is based on the conjugate gradient algorithm with incomplete factorization preconditioning, using a domain decomposed ordering to allow parallelism in the triangular solves, and resorting to some special recently developed parallelization technique to avoid communication bottleneck for the computation associated to the internal boundary nodes. Numerical results are given for a transputer network with up to 512 processors and a few workstation cluster.

Keywords: Partial differential equation; Conjugate gradient algorithm; Incomplete factorization; Preconditioning; Distributed memory multiprocessor

1. Introduction

We present here a parallel solver for linear systems arising from the discretization of second order two or three dimensional elliptic PDEs of the type

$$\begin{aligned} -\partial_x a_x \partial_x u - \partial_y a_y \partial_y u (-\partial_z a_z \partial_z u) &= f \quad \text{in } \Omega \\ u &= f_1 \quad \text{on } \Gamma_1 \subset \partial\Omega \\ \frac{\partial u}{\partial n} &= f_2 \quad \text{on } \Gamma_2 = \partial\Omega \setminus \Gamma_1 \end{aligned} \quad (1.1)$$

¹ Supported by the “Fonds National de la Recherche Scientifique”, Chercheur qualifié.
* Email: ynotay@ulb.ac.be

As is well known, for such systems, the best methods on sequential computers are difficult to parallelize while the truly parallel algorithms compare unfavourably in term of global arithmetic work, making the problem of designing a parallel solver very challenging, especially if one targets efficiency for a large number of processors or on systems like workstation clusters for which the relative cost of the communication is very high.

Clearly, two philosophies may be followed: either one implements a method of choice from the sequential point of view, trying to minimize the effects of communication, synchronization an imperfect loading balancing, or one tries to derive a truly parallel method for which the increase of the total number of arithmetic operations is kept minimal.

Up to now, the former approach has given satisfying results for parallel computers with moderated number of processors (see for instance [4,14]), while the second one seems more adapted to massively parallel systems (see [6,11], among many others).

The solver presented here is based on the conjugate gradient algorithm and belongs to the latter category.

As usual with this method, the tricky point is the choice of the preconditioner, on which relies both the numerical efficiency of the method and its intrinsic parallelism, the remaining of the algorithm being not difficult to parallelize on most architectures.

Incomplete factorization preconditioners allow an efficient reduction of the number of iterations (see e.g. [2,5,8,20]), but require two triangular solves per iterations, which, when using classical ordering schemes, prevents large scale parallelization, even if interesting implementations were developed for a moderate number of processors thanks to a careful dependency analysis of the involved recursions [4,14]. The latter implementations require in addition relatively fast communication for small messages and are therefore not necessarily adapted to all systems.

That is why we consider here the use of “domain decomposed” orderings. This consists in dividing the discrete domain in as many subdomains as available processors and in numbering separately the interior nodes of the different subdomains on the one hand, and the internal boundary nodes on the other hand, in such a way that the part of the computation related to the interior nodes reduces to local triangular solves which may be carried out independently.

Since the ordering changes with the number of processors, some degradation is expected in the convergence rate but, as will be seen, limited compared with the gain in parallelism, see also [9,10,19,22].

Here, we consider the use of this principle within the framework of the results of [21].

In the latter paper, a parallelization technique is proposed which is based on the equivalence between any iterative scheme applied to the global system with the same iterative scheme applied to an augmented system, in which internal boundary nodes are represented a number of times equal to the number of subdomains to which they belong.

The advantage is that the algorithm on this augmented system is straight forward to parallelize since it involves only purely local computations and basic communication routines similar to that needed without preconditioning. In particular, one avoids the bottleneck — potentially caused by the computation associated to internal boundary nodes [23].

Another advantage of this technique is that it leads naturally to consider ordering schemes that are more efficient from the conditioning point of view than those used in standard implementations.

In [21], we focus on the theoretical proof of the equivalence between the iterations on the augmented system and the corresponding process on the true system, the discussion of the needed assumptions in practical contexts being limited to general considerations.

Here, we address the particular case of eq. (1.1), assuming a rectangular (2D) or parallelipedal (3D) discretization grid with $n_x \times n_y (\times n_z)$ nodes and $p_x \times p_y (\times p_z)$ process grid. In this context, we show in Section 2 and 3 how the framework developed in [21] leads to a very nice parallel solution algorithm. Its efficiency on a transputer network with up to 512 processors is discussed in sections 4 while in Section 5 we give the results obtained on a 8 workstation cluster.

2. The parallel solution algorithm in 2D

We divide the physical domain Ω according to the processor grid in $p_x \times p_y$ subdomains by $(p_x - 1)$ vertical and $(p_y - 1)$ horizontal lines which are enforced to be node lines. Each subdomain is assigned to a processor which will deal with a local grid whose nodes are all nodes, including internal boundary nodes, belonging to the concerned subdomain.

Hence, nodes at internal boundaries are replicated on the different processors sharing them, and the union on the local node sets is larger than the true node set.

In [21], it is proposed in such case to apply directly the iterative process on this augmented variable set, the equivalence with a sequential algorithm applied to the true variable set being proved under some easy to check assumptions.

With that technique, replicated nodes are not assigned to one processor in particular. Rather, each vector involved in the iterative process is either a “distributed” or “replicated” representation of the corresponding vector in the equivalent sequential algorithm. The distributed representation of a vector is such that the true value at a given gridpoint is recovered by summing the values at the different nodes corresponding to that gridpoint, whereas the replicated representation is that for which the value at any node is a copy of the true value at the corresponding gridpoint.

Note that exchange the distributed representation of a vector for its replicated representation requires communication and corresponds to the following opera-

tion, where x_p denotes the local representation of the vector x on processor p :

FOR ALL p :

$$\text{for all } i: (x_p)_i \leftarrow (x_p)_i + \sum_{q \neq p} \sum_{\substack{j \\ i \text{ and } j \text{ correspond} \\ \text{to a same gridpoint}}} (x_q)_j \quad (2.1)$$

(for all i mean for all node i in the local grid associated to processor p).

Consider then the following standard implementation of the conjugate gradient algorithm; here A is the system matrix, b the right hand side, $x^{(0)}$ the initial approximation and B the preconditioner.

$$r^{(0)} = b - Au^{(0)}$$

For $k = 0, 1, \dots$ until convergence:

$$g^{(k)} = B^{-1}r^{(k)}$$

$$\alpha_k = (g^{(k)}, r^{(k)}); \delta_k = \alpha_k / \alpha_{k-1} (\delta = 0)$$

$$d^{(k)} = g^{(k)} + \delta_k d^{(k-1)} (d^{(0)} = g^{(0)})$$

$$t^{(k)} = A d^{(k)}$$

$$\gamma_k = (t^{(k)}, d^{(k)}); \beta_k = \alpha_k / \gamma_k$$

$$u^{(k+1)} = u^{(k)} + \beta_k d^{(k)}$$

$$r^{(k+1)} = r^{(k)} - \beta_k t^{(k)}$$

The main trick is that $g^{(k)}$, $u^{(k)}$, $d^{(k)}$ are replicated whereas b , $r^{(k)}$ and $t^{(k)}$ are distributed. Hence, as easily checked (see [21] for a formal proof), the inner products α_k and γ_k , because they both involve a replicated and a distributed vector, are recovered by summing on all nodes without discriminating interface nodes.

On the other hand, this choice of replicated and distributed representations is consistent if the multiplication by the system matrix is implemented in such a way that it takes a replicated vector as input to produce a distributed vector as output, whereas, conversely, the preconditioning step has to accept a distributed input vector and produce a replicated output vector.

As will be seen below, such an implementation of the matrix vector multiplication is very easy and requires no communication. Hence, all local communications are delayed to the preconditioning step, and the major result in [21] is that, in this scheme, one may efficiently implement incomplete factorization preconditioners using only the kind of communication required to exchange the distributed representation of a vector for its replicated representation (cf. (2.1)).

However, in general, this operation is not performed as such. Rather, each pair of processor (p, q) is labelled either 'f', 'l' or 'o' in such a way that (2.1) is

equivalent to:

FOR ALL p : for all i :

$$(x_p)_i \leftarrow (x_p)_i + \sum_{\substack{q \neq p \\ \text{label}(p,q) = 'f'}} \sum_j (x_q)_j \quad (2.2)$$

i and j correspond to a same gridpoint

followed by

FOR ALL p : for all i :

$$(x_p)_i \leftarrow (x_p)_i + \sum_{\substack{q \neq p \\ \text{label}(p,q) = 'l'}} \sum_j (x_q)_j. \quad (2.3)$$

i and j correspond to a same gridpoint

Here we assume, as written above, $p_x \times p_y$ processor grid, and the label of each pair of processor sharing some node is chosen on the basis of the following classification of the local boundaries of the part of the domain assigned to each processor. For a processor in position (i_p, j_p) , we define

$$\left. \begin{matrix} hf \\ hl \\ vf \\ vl \end{matrix} \right\} \begin{matrix} \text{boundary} \\ \text{as the set} \\ \text{of nodes} \\ \text{belonging} \\ \text{to the} \end{matrix} \left\{ \begin{matrix} \text{bottom (resp. top)} \\ \text{top (resp. bottom)} \\ \text{left (resp. right)} \\ \text{right (resp. left)} \end{matrix} \right\} \text{boundary} \left\{ \begin{matrix} \left\{ \begin{matrix} j_p \text{ is odd} \\ \text{(resp. even)} \end{matrix} \right\} \\ \left\{ \begin{matrix} i_p \text{ is odd} \\ \text{(resp. even)} \end{matrix} \right\} \end{matrix} \right\}$$

when

This classification is such that the common boundary between (i_p, j_p) and $(i_p, j_p + 1)$ is either the *hf* boundary on both processors (j_p even) or the *hl* boundary on both processors (j_p odd).

Hence, we may set without ambiguity label $((i_p, j_p), (i_p, j_p \pm 1)) = 'f'$ if their common boundary is the *hf* boundary, and label $((i_p, j_p), (i_p, j_p \pm 1)) = 'l'$ if it is the *hl* boundary. Similarly, we set label $((i_p, j_p), (i_p \pm 1, j_p)) = 'f'$ if their common boundary is the *vf* boundary and label $((i_p, j_p), (i_p \pm 1, j_p)) = 'l'$ if it is the *vl* boundary.

Finally, the labelling is completed by setting

$$\text{label}((i_p, j_p), (i_p \pm 1, j_p \pm 1)) = \begin{cases} 'f' & \text{if } \text{label}((i_p, j_p), (i_p, j_p \pm 1)) = 'f' \\ & \text{and } \text{label}((i_p, j_p), (i_p \pm 1, j_p)) = 'f' \\ 'l' & \text{if } \text{label}((i_p, j_p), (i_p, j_p \pm 1)) = 'l' \\ & \text{and } \text{label}((i_p, j_p), (i_p \pm 1, j_p)) = 'l' \\ 'o' & \text{otherwise.} \end{cases}$$

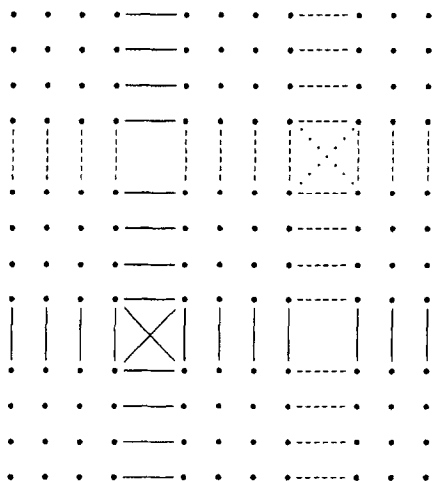
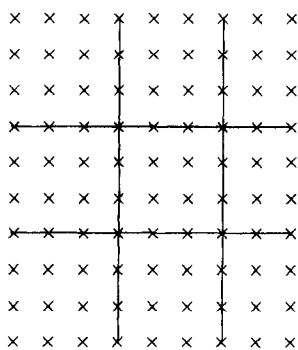


Fig. 1. 1.9×10 discretization grid with the division of the corresponding domain in 9 subdomains (above), and associated local grids (below); nodes i, j corresponding to a same gridpoint are joined straight lines if the label of the pair formed by their respective processors is 'l', and by dashed lines if it is 'f'.

These choices are illustrated on Fig. 1. for a 3×3 processor grid. The interested reader will easily check that all requirements stated in [21] are satisfied.

The algorithm to be executed on each processor is then as follows where A_p and U_p , which will be described below, are defined on the local grid of each processor and may be seen as respectively a "local" system matrix and a "local" upper triangular factor. $P_p = \text{diag}(U_p)$ and $\Delta_p^{(f)}$ is the diagonal matrix defined by $(\Delta_p^{(f)})_{ii} = \frac{1}{m_i^{(f)}}$ with, for all node i on processor p ,

$$m_i^{(f)} = 1 + \#\{q \neq p \mid \text{label}(p, q) = 'f' \text{ and the gridpoint corresponding to } i \text{ is also represented on } q\}.$$

For the operations corresponding to (2.2) and (2.3), we use respectively the condensed notation $x_p \leftarrow \Sigma_p^{(f)}(x)$ and $x_p \leftarrow \Sigma_p^{(l)}(x)$. These operations, together with the global summation of partial inner products and the multiplication by $\Delta_p^{(f)}$, are the only ones which distinguish this algorithm from a purely local conjugate gradient solve. A left arrow indicates a step requiring communication.

FOR ALL p :

$$r_p^{(0)} = b_p - A_p u_p^{(0)}$$

For $k = 0, 1, \dots$ until convergence:

$$g_p^{(k)} = r_p^{(k)}$$

$$g_p^k \leftarrow \Sigma_p^{(f)}(g_p^{(k)})$$

$$g_p^{(k)} := (U_p^t)^{-1} g_p^k$$

$$g_p^{(k)} \leftarrow \Sigma_p^{(f)}(g_p^{(k)})$$

$$g_p^{(k)} := (P_p^{-1} U_p)^{-1} \Delta_p^{(f)} g_p^{(k)}$$

$$g_p^{(k)} \leftarrow \Sigma_p^{(f)}(g_p^{(k)})$$

$$\alpha_p^{(k)} = (g_p^{(k)}, r_p^{(k)})$$

$$\alpha_k \leftarrow \sum_q (\alpha_p^{(k)}); \delta_k = \alpha_k / \alpha_{k-1} (\delta = 0)$$

$$d_p^{(k)} = g_p^{(k)} + \delta_k d_p^{(k-1)} (d_p^{(0)} = g_p^{(0)})$$

$$t_p^{(k)} = A_p d_p^{(k)}$$

$$\gamma_p^{(k)} = (t_p^{(k)}, d_p^{(k)})$$

$$\gamma_k \leftarrow \sum_q (\gamma_p^{(k)}); \beta_k = \alpha_k / \gamma_k$$

$$u_p^{(k+1)} = u_p^{(k)} + \beta_k d_p^{(k)}$$

$$r_p^{(k+1)} = r_p^{(k)} - \beta_k t_p^{(k)}$$

The conditions under which this process is equivalent to a standard PCG solution performed on the global system are given below:

- (1) The global system matrix A and right hand side b correspond to the assembly of the different local system matrices and right hand sides. That is, an element a_{ij} in A (b_i in b) has to be the sum of the corresponding contributions in A_p (b_p) from all the processors p where both gridpoints i and j are (the gridpoint i is) represented.

- (2) When two processors have a set of gridpoints in common, the order relation between the corresponding nodes is the same in both local orderings.
- (3) When one or two gridpoints are shared by more than one region, the corresponding diagonal or offdiagonal element takes the same value in all concerned local matrices U_p , so that all of them are the restriction to the local grid of a matrix U defined on the global grid. Zeroing in the latter any entry between two gridpoints not belonging to a same region, it turns out to be unique and $B = U^t P^{-1} U$, with $P = \text{diag}(U)$, is the corresponding global preconditioner.
- (4) Calling successors of a node i in a local grid the nodes j , necessarily with larger indexes, such that $u_{ij}^{(p)} \neq 0$ in the corresponding local matrix U_p , all nodes of the hl (respectively vl) boundary, whenever internal (i.e. whenever common with another processor), have only as successors nodes belonging to the hl (resp. vl) boundary.
- (5) Calling precursors of a node i in a local grid the nodes j , necessarily with smaller indexes, such that $u_{ji}^{(p)} \neq 0$ in the corresponding local matrix U_p , all nodes belonging to the hf (respectively vf) boundary, whenever internal, have only as precursors nodes belonging to the hf (resp. vf) boundary.
- (6) The value of the initial approximation at any node in a local grid is equal to the value of the global initial approximation $u^{(0)}$ at the corresponding gridpoint.

Under these conditions, The results in [21] prove that the approximate solution obtained at some node in a local grid is equal, at any step, to the approximate solution that one would obtain for the same step at the corresponding gridpoint by performing a sequential PCG solution of the system $Au = b$ with $u^{(0)}$ as initial approximation and B as preconditioner.

(1) is not difficult to satisfy in practice and we shall assume here that the discretization process directly provides local matrices A_p and right hand sides b_p , whose assembly gives the global system matrix and right hand side; when using a finite element discretization or the finite difference point scheme box integration [16], it suffices indeed to perform a local discretization on the subdomains, treating internal boundaries as if they were external with natural boundary conditions.

To manage (2) while minimizing the constraints raised by (4) and (5), we decided to order the nodes of each local grid lexicographically, starting at the intersection of the hf and vf boundaries, and progressing first in the x direction, i.e. along the hf boundary. This is illustrated on Fig. 2 for the example of Fig. 1. One easily checks that all connections raised by a 5-point scheme are compatible with (4) and (5) when using a generalized SSOR factorization (that is when U and A have same upper triangular part).

However, this does not mean that our solver is limited to 5 point finite difference or linear finite element discretizations. Indeed, as the existence and conditioning properties of approximate factorization preconditioners are guaranteed only if the factorized matrix is a Stieltjes matrix (i.e. symmetric positive definite with nonpositive offdiagonal entries), it is a common technique to use as

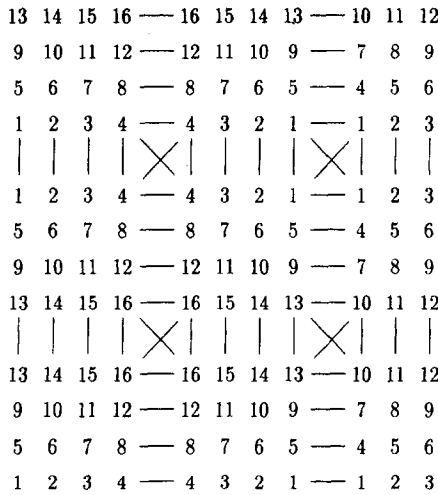


Fig. 2. Local orderings for the example of Fig. 1. Nodes corresponding to a same gridpoint are joined by straight lines.

preconditioner for a further scheme the incomplete factorization of the corresponding 5 point finite difference or linear finite element matrix, see e.g. [2]. Other techniques used to derive Stieltjes approximations may also be used to discard some selected negative offdiagonal entries, see [17].

Hence, we may assume here, without loss of generality, that, together with the local system matrices A_p , it is furnished local approximations \tilde{A}_p whose connections are all compatible with (4) and (5) and such that the matrix \tilde{A} corresponding to their assembly is a Stieltjes matrix.

It remains to satisfy (3). Here, we choose to consider generalized SSOR factorizations, so that the first step of the factorization procedure consists in setting the offdiagonal entries in the upper triangular part of the local matrices U_p equal to the corresponding value in \tilde{A} , by means of the information stored in \tilde{A}_p and basic communication between neighbour processors.

The data dependency in the corresponding U is then as illustrated on Fig. 3 for the example of Fig. 1 & 2 with a 5 point grid. It corresponds to that obtained with a domain decomposed ordering, except that here part of the boundaries are ordered first (the hf and vf ones), and the remaining last (hl & vl), which results in some benefit compared with classical implementations where all boundaries are ordered last, a p processors ordering in the latter framework being more or less equivalent to a $4p$ processors ordering in the present implementation.

Letting π be the vector which stores the diagonal part of U (i.e. $\pi_i = u_{ii}$ for all i), the sequential algorithm to compute it is

$$\begin{aligned}
 & \text{initialize: } \pi_i = a_{ii} \text{ for all } i \\
 & \text{for } i = 1, \dots, n: \\
 & \text{for all } j > i: u_{ij} \neq 0
 \end{aligned}
 \tag{2.4}$$

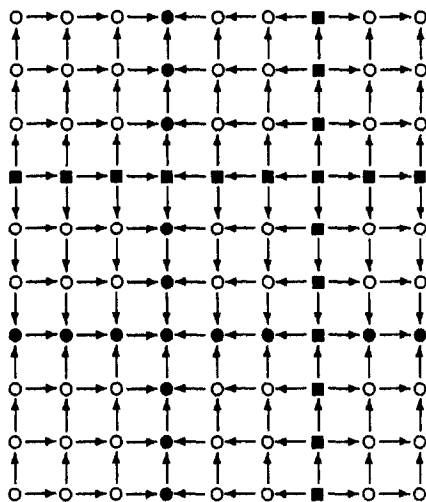


Fig. 3. Data dependency in U for the example of Figs. 1 & 2 with a 5-point grid and a generalized SSOR factorization. All connections in U are represented by an arrow which originates from the node with smallest index. Internal boundary nodes are represented by a square if they are “ordered first” and by a filled circle if they are “ordered last”.

$$\pi_j := \pi_j - \frac{u_{ij}^2}{\pi_i} - \omega_i \frac{u_{ij}}{\pi_i} (\sigma_i - u_{ij}) \tag{2.5}$$

where

$$\sigma_i = \sum_{j>i} u_{ij}$$

and where ω_i depends on the chosen incomplete factorization method; $\omega_i \equiv 0$ corresponds to the IC method [15], $\omega_i \equiv 1$ to the MIC method (without perturbations) (see e.g. [8]) and $\omega_i \equiv \omega$ for some $0 < \omega < 1$ to the RIC method [3]. For the numerical tests in the text section, we chose the DRIC method because of its particular robustness to solve discrete second order elliptic PDEs [18,20]. It is slightly more sophisticated and uses

$$\omega_i = \min \left(\frac{2(1 - \alpha)\pi_i}{-\sigma_i} - 1, 1 \right)$$

where α , $0 < \alpha \leq 1$ if some input parameter (see Section 4).

Here, we need an algorithm to compute π by means of local computation with mere communication procedures. Note that, by requirement (3), we need to obtain the replicated representation of π . In this view, a very first step is to obtain the

replicated representation of σ . This is easy with

FOR ALL p :

$$\text{for all } i: (\sigma_p)_i = \sum_{j>i} \tilde{a}_{ij}^{(p)}$$

(which gives its distributed representation) followed by

FOR ALL p :

$$\sigma_p \leftarrow \Sigma_p^{(f)}(\sigma)$$

$$\sigma_p \leftarrow \Sigma_p^{(l)}(\sigma)$$

(where we use the same notation as in the parallel solution algorithm above, and take advantage that (2.1) is equivalent to (2.2) followed by (2.3) to reuse routines needed during the solution process).

Then, the approximate factorization algorithm (2.4), (2.5) may be achieved by applying the following algorithm where, because of the needed communications, the computations for the nodes on the uf and hl boundaries are separated from that of the remaining nodes. On each processor p , n_p is the last node in the local ordering, i.e. the node which intersects both hl and ul boundaries.

FOR ALL p :

– for all i : $(\pi_p)_i = \tilde{a}_{ii}^{(p)}$

– for all i not belonging to the hl nor the ul boundary:

$$(\pi_p)_i \leftarrow (\pi_p)_i + \sum_{q \neq p} \sum_j \begin{matrix} (\pi_q)_j \\ \text{\small } i \text{ and } j \text{ correspond} \\ \text{\small to a same gridpoint} \end{matrix}$$

– for all i in increasing order, whenever i does not belong to the hl nor the ul boundary

for all $j > i$: $u_{ij} \neq 0$

$$(\pi_p)_j := (\pi_p)_j - \frac{(u_{ij}^{(p)})^2}{(\pi_p)_i} - \omega_i \frac{u_{ij}^{(p)}}{(\pi_p)_i} ((\sigma_p)_i - u_{ij}^{(p)})$$

– for all $i \neq n_p$ belonging to either the hl or the ul boundary:

$$(\pi_p)_i \leftarrow (\pi_p)_i + \sum_{q \neq p} \sum_j \begin{matrix} (\pi_q)_j \\ \text{\small } i \text{ and } j \text{ correspond} \\ \text{\small to a same gridpoint} \end{matrix}$$

– for all i in increasing order, i belonging to either the hl or the vl boundary:

for all $j > i, j \neq n_p: u_{ij} \neq 0$

$$(\pi_p)_j := (\pi_p)_j - \frac{(u_{ij}^{(p)})^2}{(\pi_p)_i} - \omega_i \frac{u_{ij}^{(p)}}{(\pi_p)_i} ((\sigma_p)_i - u_{ij}^{(p)})$$

if $u_{in_p} \neq 0$:

$$(\pi_p)_{n_p} := (\pi_p)_{n_p} - \frac{(u_{in_p}^{(p)})^2}{m_i^{(l)}(\pi_p)_i} - \omega_i \frac{u_{in_p}^{(p)}}{m_i^{(l)}(\pi_p)_i} ((\sigma_p)_i - u_{in_p}^{(p)})$$

$$(\pi_p)_{n_p} \leftarrow (\pi_p)_{n_p} + \sum_{q \neq p} \sum_{n_q} (\pi_q)_{n_q}$$

n_p and n_q correspond to a same gridpoint

where

$m_i^{(l)} = 1 + \#\{q \neq p \mid \text{label}(p,q) = 'l' \text{ and the grid point corresponding to } i \text{ is also represented on } q\}$.

(in fact, $m_i^{(l)} = 2$ for any concerned node except when the boundary to which it belongs is not common with another processor, that is an external boundary; this division is required because other wise contributions would be added multiple times at next step).

Remark. It is relevant to compare here briefly our approach with the interesting method proposed by Radicati di Brozolo et al. [24,12].

In the latter papers, it is suggested to split the unknowns into overlapping blocks, preferably in such a way that all nonzero entries in the system matrix appear in at least one block. Then, to each block is associated a “local” incomplete factorization, either by taking the restriction to the block of a beforehand computed global factorization, or by just performing an incomplete factorization of the “local” part of the system matrix.

The preconditioning step $g = B^{-1}r$ consists then in computing (in parallel) local triangular solves, the global vector g being set equal to the local ones in the nonoverlapped parts and to the average of them in the overlapped parts.

Clearly, there are similarities between this *Overlapped Partitioned ILU (OPI)*, and our suggestion to iterate on an augmented system, and one may even think that our method, although originating from a different point of view, is just a sophisticated version of OPI.

Now, the interesting point is that the main potential drawbacks of the OPI method are avoided by using our approach. Indeed:

- in [24], the authors obtain actually the best iteration counts when using as local triangular factors the restriction of a global ILU which has to be computed sequentially.

Our parallel factorization algorithm allows to compute a global factorization by means of local computation.

- Even when using the restriction of a globally defined factorization, the averaging process in the OPI method implies the loss of the equivalence with the sequential algorithm, so that one may have some doubts on the convergence properties, especially for difficult problems with discontinuities or in case of many processors.

Here, using only (2.2), (2.3) which are not more complicated to program than a mere average procedure, we maintain the equivalence with the sequential process. It means that any deterioration of the convergence rate can only be caused by the variation of the global ordering with the number of processors. The numerical tests below show that the effect of this variation is very limited compared with the gain in parallelism.

This conclusion is in addition supported by recent theoretical results in [22], where some suggestions are further given to improve the convergence and obtain a completely scalable algorithm.

- Because of its rather empirical basis, the OPI approach seems usable only in combination with an all purpose method like IC. Hence in any event, the number of iterations will grow with the number of unknowns at least as $\mathcal{O}(n^{1/2})$. By preserving the equivalence with a globally defined incomplete factorization, our technique is compatible with any factorization method, in particular with those like DRIC for which one may prove that the number of iterations grows not more than $\mathcal{O}(n^{1/4})$. Note that, see [22], this still holds for the multiprocessor orderings considered here, even through this number of iterations slightly increases for fixed n as the number of processors increases.

3. The algorithm in 3D

In 2D, both directions x and y play exactly the same role, and the 3D algorithm is just an extension of the 2D one to a third demension.

In fact, the parallel solution algorithm given in Section 2 applies as such, provided that we adapt the labelling of the pair of processors to the case of a 3D grid distributed on a 3D processor grid.

To this aim, we first extends our classification of the local boundaries: for a

processor in position (i_p, j_p, k_p) ,

$$\text{its } \left. \begin{matrix} \left. \begin{matrix} xf \\ xl \\ yf \\ yl \\ zf \\ zl \end{matrix} \right\} \text{ boundary} \\ \text{is the set of} \\ \text{nodes be-} \\ \text{longing to} \\ \text{the} \end{matrix} \right\} \left. \begin{matrix} \left. \begin{matrix} \text{left (resp. right)} \\ \text{right (resp. left)} \\ \text{back (resp. front)} \\ \text{front (resp. back)} \\ \text{bottom (resp. top)} \\ \text{top (resp. bottom)} \end{matrix} \right\} \text{ boundary} \\ \text{when} \end{matrix} \right\} \left. \begin{matrix} \left. \begin{matrix} i_p \text{ is odd} \\ \text{(resp. even)} \end{matrix} \right\} \\ \left. \begin{matrix} j_p \text{ is odd} \\ \text{(resp. even)} \end{matrix} \right\} \\ \left. \begin{matrix} k_p \text{ is odd} \\ \text{(resp. even)} \end{matrix} \right\} \end{matrix} \right\}.$$

Then, as in 2D, $label((i_p, j_p, k_p), (i_p \pm 1, j_p, k_p)) = 'f'$ if their common boundary is the xf boundary whereas $label((i_p, j_p, k_p), (i_p \pm 1, j_p, k_p)) = 'l'$ if it is the xl boundary. We proceed similarly for the two other directions.

Next, $label((i_p, j_p, k_p), (i_p \pm 1, j_p \pm 1, k_p))$, $label((i_p, j_p, k_p), (i_p \pm 1, j_p, k_p \pm 1))$ and $label((i_p, j_p, k_p), (i_p, j_p \pm 1, k_p \pm 1))$ are determined by applying, as in 2D the rules

$$\left\{ \begin{matrix} label(p,q) = 'f' \text{ and } label(q,r) = 'f' \Rightarrow label(p,r) = 'f' \\ label(p,q) = 'l' \text{ and } label(q,r) = 'l' \Rightarrow label(p,r) = 'l' \\ label(p,q) = 'f' \text{ and } label(q,r) = 'l' \Rightarrow label(p,r) = 'o' \end{matrix} \right.$$

Finally, it is not difficult to check that the same rules uniquely determine $label((i_p, j_p, k_p), (i \pm 1, j_p \pm 1, k_p \pm 1))$ for any relevant (i_p, j_p, k_p) .

The conditions under which the parallel solution algorithm in Section 2 is equivalent to a standard PCG solution performed on the global system are then the same conditions (1)-(6) given above, except that here (4) is to be applied to the xl , yl and zl boundaries and (5) to the xf , yf and zf boundaries.

The local orderings are still lexicographic ones, here starting at the corner intersecting the xf , yf and zf boundaries. As in the preceding case, this implies that all connections raised by a 7 point scheme are compatible with the given conditions. Hence, considering again generalized SSOR factorizations, the off-diagonal entries in the local upper triangular factors are just equal to the corresponding entries in the global system matrix.

The parallel facotrization algorithm to determine the diagonal part is here slightly more complicated, although the same technique that have led to the algorithm in Section 2 can still be employed. We refer to [21] for a general parallel factorization algorithm.

4. Numerical results on Parsytec GCel

We tested the method described in Sections 2 and 3 on a transputer (T 805) network (Parsytec GCel-3 \ 512 multiprocessor).

We exclusively used the communication routines provided by the PARIX virtual topology library, which is invoked by the program to build a processor grid with user specified dimensions, see [7] for details. We therefore did not consider the use of low level programming language to reduce the communication cost.

The program was written in double precision FORTRAN. We did not consider scaling to save some multiplications during the iterations, and, more generally, no particular optimization effort was made, so that the timing results given below cannot pretend to be optimal and should be used only as a relative criteria.

The stopping criterion was $\|r^{(k)}\|_{B^{-1}} < 10^{-6} \|b\|_{B^{-1}}$, where $r^{(k)}$ is the residual, b the right hand side and $\| \cdot \|_{B^{-1}} = \sqrt{(\cdot, B^{-1} \cdot)}$. This makes the test dependent of the preconditioner, but avoids the computation of any additional inner product, reason for which this test serves as basis for the “natural” stopping criterion proposed in [1]. In practice, we observed that this dependence has only a limited influence on the number of iterations. Another advantage of this criterion in our context is that $\|r^{(k)}\|_{B^{-1}}$ may be directly computed on the augmented variable set since $\|r^{(k)}\|_{B^{-1}} = \alpha_k$ for the quantity α_k computed as in the algorithm given in Section 2.

4.1. 2D problems

As test problems, we considered the PDE (1.1) with $\Omega = (0,1) \times (0,1)$ and

PROBLEM 1: $a_x = a_y = f \equiv 1$ in Ω , $u = 0$ on $\partial\Omega$

PROBLEM 2:

$$a_x = a_y = \begin{cases} 100 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \\ 1 & \text{elsewhere,} \end{cases}$$

$$f = \begin{cases} 100 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \\ 0 & \text{elsewhere,} \end{cases}$$

$$\begin{cases} u = 0 & \text{for } 0 \leq x \leq 1, y = 0 \\ \frac{\partial u}{\partial n} = 0 & \text{on the remaining part of the boundary} \end{cases}$$

PROBLEM 3:

$$a_x = 1 \text{ and } a_y = \begin{cases} .001 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \\ 1 & \text{elsewhere,} \end{cases}$$

$$f = \begin{cases} 1 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \\ 0 & \text{elsewhere,} \end{cases}$$

$$\begin{cases} u = 0 & \text{for } 0 \leq x \neq 1, y = 1 \text{ and } 0 \neq y \neq 1, x = 1 \\ \frac{\partial u}{\partial n} = 0 & \text{on the remaining part of the boundary} \end{cases}$$

In all cases, we considered the 5 point finite difference approximation (point scheme integration [16]) with uniform mesh size h in both directions. We used as preconditioner for the conjugate gradient solution, the domain decomposed generalized SSOR, DRIC factorization described in Section 2 with parameter $\alpha = h$ following the recommendations in [20].

For comparison purpose, we give also the results obtained for the Jacobi preconditioner with 256 processors. For convenience, we implemented it within the framework described in Section 2, using

$$g_p^{(k)} \leftarrow \Sigma_p^{(f)}(g^{(k)})$$

$$g_p^{(k)} \leftarrow \Sigma_p^{(l)}(g^{(k)})$$

$$g_p^{(k)} := D_p^{-1} g_p^{(k)}$$

for the preconditioning step, where D_p is the replicated representation of the diagonal of A on processor p . The communications are then the same as that needed with an optimal implementation², but, since one iterates on an augmented variable set, one may have an extra arithmetic cost of at most 25%, 12.5%, 6% and 3% for respectively $h^{-1} = 128, 256, 512$ and 1024 ³.

The observed computing time for the solution part, as well as the number of iterations, are reported in Table 1. In any case, we used square processor grids ($p_x = p_y$) and zero initial approximations.

From the table, it is seen that the computing time decreases in any case when the number of processors increases. We do not give efficiencies because the comparative run on a single processor was in most cases not possible, due to excessive memory requirements. However, one may check that, when the number of processors is multiplied by 4, the computing time is reduced by a factor more or less equal to 3, with large variations due to the irregularities in the increase of the number of iterations. For the smallest problems and the largest number of processors, this factor is further reduced because the relative communication cost increases prohibitively.

This may appear not really impressive, but it is precisely one of the major feature of our solver that the amount of parallelism offered is automatically adapted to the number of available processors, so that the global arithmetic work increases continuously with the latter, but without any minimal number of processors required to run faster than the sequential execution of a comparison method. In fact, our solver, in the 1 processor case, reduces to the preconditioned conjugate gradient method with a generalized SSOR-DRIC factorization preconditioner computed with respect the standard lexicographic ordering, which may fairly be

²That is 1 real data exchange per internal boundary node and per iteration, with respect a box partitioning which minimizes the number of internal boundary nodes.

³These estimations assume a perfect load balancing for the optimal implementation, which is seldomly attainable in practice.

Table 1

Solution time (in seconds) and number of iterations for 2D problems on Parsytec GCel

PROBLEM 1								
h^{-1}	128		256		512		1024	
# proc.	time	(# it)	time	(# it)	time	(# it)	time	(# it)
1	47.1	(36)	—	(52)	—	(77)	—	(114)
4	9.79	(29)	60.1	(45)	—	—	—	—
16	3.10	(32)	16.3	(46)	97.1	(71)	—	—
64	1.29	(42)	6.44	(66)	36.6	(103)	220.	(161)
256	1.02	(58)	3.14	(90)	14.2	(140)	75.0	(209)
(Jacobi) 256	3.48	(203)	12.0	(409)	66.0	(827)	464.	(1671)

PROBLEM 2								
h^{-1}	128		256		512		1024	
# proc.	time	(# it)	time	(# it)	time	(# it)	time	(# it)
1	74.7	(56)	—	(82)	—	(123)	—	(182)
4	17.6	(51)	99.9	(74)	—	—	—	—
16	5.77	(60)	32.2	(91)	190.	(139)	—	—
64	2.41	(89)	11.1	(114)	62.4	(176)	356.	(261)
256	1.73	(100)	5.24	(150)	22.4	(222)	118.	(330)
(Jacobi) 256	6.78	(452)	25.9	(910)	146.	(1840)	1027.	(3705)

PROBLEM 3								
h^{-1}	128		256		512		1024	
# proc.	time	(# it)	time	(# it)	time	(# it)	time	(# it)
1	80.7	(61)	—	(88)	—	(127)	—	(183)
4	24.5	(71)	142.	(105)	—	—	—	—
16	7.02	(73)	38.2	(108)	221.	(162)	—	—
64	2.98	(98)	13.8	(142)	74.8	(211)	423.	(310)
256	2.45	(132)	6.51	(187)	27.8	(275)	142.	(397)
(Jacobi) 256	8.59	(618)	34.1	(1162)	200.	(2556.)	1441.	(5203)

considered as a comparison method for discrete problems of the type (1.1) solved with the conjugate gradient algorithm, see [18, 20].

Moreover, from our tests, it appears that this number of iterations do generally not increases when #proc. increases from 1 to 4 (in accordance with the theoretical and experimental results in [9,19,22]). Then, it increases slightly less than $\mathcal{O}((\#proc.)^{1/4})$, and, for 256 processors, remains not greater than about twice that

required by the single or four processor version, which represents a quite reasonable cost for such a massively parallel method.

For large system, this cost is unavoidable, due to the limited amount of memory available on each processor. Classical computers do not suffer from this penalty, but increasing problem sizes means increasing use of virtual memory and slowing down of the whole process.

This penalty is also avoided with implementations like that described in [4,14], where the incomplete factorization preconditioners with lexicographic ordering are parallelized by means of a wavefront like method. However, it turns out that such techniques require relatively fast communication for small messages, and can in addition not be efficient for large numbers of processors, since they require a division of the discrete domain into strips; for instance a strip partitioning in 256 processors is not possible for $h^{-1} = 128$, and causes a severe load balancing problem for $h^{-1} = 512$, the number of columns to handle being actually 513 when both left and right sides of the domain have been specified Neumann boundary conditions. The same remark holds for the implementation of block preconditionings like that proposed in [13].

Comparing now our solver with a standard Jacobi preconditioned conjugate gradient solution, it is clear that the latter is outmatched by far, even taking into account the factorization cost on the one hand and the non optimality of our implementation of the Jacobi preconditioning on the other hand. Actually, our incomplete factorization preconditioner allows as usual to save arithmetic work, but also broadcast communication for the update of the inner products (since the number of iterations is cut down by a factor up to ten), and communication between neighbour processors (since, per iteration, our solver requires only 50% more such communication than the amount needed for a mere multiplication by the system matrix).

The number of iterations required with our preconditioner is $\mathcal{O}(h^{-1/2})$ for fixed number of processors, and thus globally slightly better than $\mathcal{O}(h^{-1/2}(\# \text{ proc.})^{1/4})$, while, for the Jacobi preconditioner, it grows like $\mathcal{O}(h^{-1})$, which is less interesting even if the number of processors is increased together with the problem size so as to maintain the load per processor constant.

We did not make direct comparison with other popular purely parallel preconditioners like polynomial preconditioners or those based on red-black or multicolor orderings. However, it is clear that these methods can improve the Jacobi method only by a constant factor, the associated number of iterations remaining $\mathcal{O}(h^{-1})$, see e.g. [6]. On the other hand, such preconditioners involve, per iteration, more communications than the one used in our solver within the framework of the implementation described here, see for instance [11]. Hence, we believe that our solver performs also better than the latter methods for examples like those considered here.

Finally, it is interesting to include here some comments on the results in [22], which were not available at the time this manuscript was prepared. First, it is shown there that the scalability of the algorithm is much improved by exchanging the rule $\alpha = h$ for $\alpha = h\sqrt{\# \text{ proc.}}/2$. The gain increases with the number of

processors, and, for instance, in the case of 16×16 processor grid, from 20 to 30% iterations appear to be saved by changing nothing in the algorithm but the choice of the parameter α .

Next, it is shown there that a perfectly scalable algorithm is obtained by including some coarse grid correction, whose design appear to be compatible with the implementation technique considered here.

These two facts give another argument to maintain the equivalence of “parallel” preconditioners with well understood “global” preconditioners, any progress made in the design of the latter being then directly available to improve their parallel version.

4.2. 3D problems

We considered also the PDE (1.1) with $\Omega = (0,1) \times (0,1) \times (0,1)$ and

PROBLEM 4: $a_x = a_y = a_z = f \equiv 1$ in Ω , $u = 0$ on $\partial\Omega$

PROBLEM 5:

$$a_x = a_y = a_z = \begin{cases} 100 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \\ 1 & \text{elsewhere,} \end{cases}$$

$$f = \begin{cases} 100 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \\ 0 & \text{elsewhere,} \end{cases}$$

$$\begin{cases} u = 0 & \text{for } 0 \leq x, z \leq 1, y = 0 \\ \frac{\partial u}{\partial n} = 0 & \text{on the remaining part of the boundary} \end{cases}$$

PROBLEM 6:

$$a_x = a_y = a_z = \begin{cases} .001 & \text{in } (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \times (1,1) \\ 1 & \text{elsewhere,} \end{cases}$$

$$f = \begin{cases} 1 & \text{in } (\frac{1}{4}) \times (\frac{1}{4}, \frac{3}{4}) \times (1,1) \\ 0 & \text{elsewhere,} \end{cases}$$

$$\begin{cases} u = 0 & \text{for } 0 \leq x, y \leq 1, z = 0, 0 \leq y, z \leq 1, x = 0 \\ & \text{and } 0 \leq x, z \leq 1, y = 0 \\ \frac{\partial u}{\partial n} = 0 & \text{on the remaining part of the boundary} \end{cases}$$

In all cases, we considered the 7 point finite difference approximation (point scheme integration [16]) with uniform mesh size h in all directions. We used as preconditioner for the conjugate gradient solution, the domain decomposed generalized SSOR, DRIC factorization described in Section 3 with parameter $\alpha = h$.

The observed computing time for the solution part, as well as the number of

Table 2
Solution time (in seconds) and number of iterations for 3D problems on Parsytec GCel

PROBLEM 4						
h^{-1}	32		64		128	
# proc.	time	(# it)	time	(# it)	time	(# it)
1	62.2	(21)	—	(31)	—	(45)
8	7.83	(18)	93.8	(28)	—	—
64	2.20	(19)	18.1	(29)	198.	(49)
512	1.01	(24)	5.99	(43)	47.9	(69)
(Jacobi) 512	1.95	(63)	12.8	(127)	131.	(259)

PROBLEM 5						
h^{-1}	32		64		128	
# proc.	time	(# it)	time	(# it)	time	(# it)
1	127.	(37)	—	(55)	—	(81)
8	17.0	(33)	182.	(50)	—	—
64	4.15	(36)	36.0	(58)	367.	(91)
512	1.88	(45)	9.89	(70)	75.5	(108)
(Jacobi) 512	4.91	(156)	32.6	(316)	326.	(639)

PROBLEM 6						
h^{-1}	32		64		128	
# proc.	time	(# it)	time	(# it)	time	(# it)
1	87.6	(27)	—	(41)	—	(62)
8	15.4	(30)	164.	(45)	—	—
64	4.17	(36)	34.3	(55)	339.	(84)
512	1.71	(41)	9.08	(64)	67.2	(96)
(Jacobi) 512	4.49	(143)	30.5	(293)	305.	(600)

iterations, are reported in Table 2. In any case, we used cubic processor grids ($p_x = p_y = p_z$) and zero initial approximations.

For comparison purpose, we give also the results obtained for the Jacobi preconditioner with 512 processors.

Compared with the 2D results, it turns out that the increase of the number of iterations with the number of processors is here much more moderate. This explains because, in both cases, the leading parameters appears to be the number of processors per line in the grid. Therefore, as one may check by comparing

Tables 1 and 2, the deterioration in convergence observed for 512 ($= 8 \times 8 \times 8$) processors in 3D is similar to that observed for only 64 ($= 8 \times 8$) processors in 2D.

Nevertheless, if the parallelization is more easy from this point of view, the timing results indicate that the communication overhead is here more significative. We explain this by two factors. On the one hand, the internal boundaries are here the faces of a cube instead of the edges of a square, so that each processor has to communicate with 6 neighbours rather than 4, while the messages are in addition longer. On the other hand, on the considered machine, the physical processor grid is a two dimensional one in which each processor is directly connected to only four others, so that part of the interprocessor communications concern non neighbour processors and are therefore slower.

However, these problems are inherent to 3D calculation and validate our choice to first and foremost minimize the communications.

The comparison with the Jacobi preconditioning raises the same comments as in 2D, except that here the saving of iterations is somewhat less spectacular, essentially because it is above all function of the mesh size which is more modest than in 2D examples.

5. Numerical results on IBM cluster

We performed also some tests on a 8 workstation IBM RS6000 cluster, using the public domain PVM software.

Here, each processor appears sufficiently powerful to run efficiently sequentially, while the communications remain relatively slow compared with the speed of floating point calculation. This represents generally a major limitation on such systems, and makes the comparison of a parallel program with a sequential execution somewhat challenging in spite of the modest number of processors.

For this comparison, we considered some of the problems referred in the preceding section, using the same stopping criterion, discretization scheme, approximate factorization method and solution process.

The results are given in Table 3. We always used zero initial approximation and square (2D Problems 1 and 2) or cubic (3D Problems 4 and 5) processor grids, except in 2D cases when $\# \text{proc.} = 8$, where we used $p_x = 4$ and $p_y = 2$.

Looking at the CPU times, it turns out that the parallelization is especially interesting when the memory requirements of the sequential version reach the available in core memory.

This explains why, for the largest dimensions, efficiencies are always greater than 1!

On the other hand, efficiencies may appear relatively poor for the smallest load per processor, but one has to take into account that this concerns only cases where the program executes sequentially in less than a very few seconds, i.e. cases for which the interest of the parallelization is anyway doubtful.

Finally, we obtain efficiencies slightly less than 1 in intermediate cases, showing that our method allows a successful parallelization even below the limit where the sequential version suffers from the recourse to virtual memory. This conclusion

Table 3

Solution time (in seconds) and number of iterations for Problems 1, 2, 4 and 5 on IBM cluster. %CPU is the ratio between the CPU time spent for the solution part (reported in the first column) and the total elapsed time during the execution of the whole program

PROBLEM 1									
h^{-1}	144			288			576		
# proc.	CPU	(# it)	%CPU	CPU	(# it)	%CPU	CPU	(# it)	%CPU
1	1.36	(38)	39%	7.92	(56)	38%	47.7	(82)	30%
4	.333	(32)	12%	1.92	(48)	21%	10.5	(72)	29%
8	.246	(35)	6%	1.09	(51)	16%	5.94	(76)	22%

PROBLEM 2									
h^{-1}	144			288			576		
# proc.	CPU	(# it)	%CPU	CPU	(# it)	%CPU	CPU	(# it)	%CPU
1	2.20	(60)	25%	12.5	(88)	46%	76.8	(131)	44%
4	.635	(54)	17%	3.19	(79)	47%	16.8	(115)	70%
8	.514	(55)	8%	1.88	(82)	18%	9.41	(120)	39%

PROBLEM 4						
h^{-1}	36			72		
# proc.	CPU	(# it)	%CPU	CPU	(# it)	%CPU
1	2.24	(23)	14%	96.9	(33)	4%
8	.321	(19)	12%	3.61	(31)	32%

PROBLEM 5						
h^{-1}	36			72		
# proc.	CPU	(# it)	%CPU	CPU	(# it)	%CPU
1	4.35	(39)	44%	195.	(59)	2.7%
8	.644	(36)	13%	6.61	(54)	32%

could be strengthened if the memory per processor were higher. Indeed, we had here 32 MB per node, which is relatively small for such powerful processors and such memory intensive computations.

We also report on the real time needed to execute the program. Indeed, although it is subject to large variations from run to run, this is one of the parameter the user is interested in when parallelizing an application. Moreover, it

appears that part of the costs inherent to the execution of the program are hidden when considering only the user CPU time, maybe because a process which is waiting for data is automatically unloaded even if no other program is running on the machine. So, the tendency to have low use of CPU time when parallelizing with small load per processor on the one hand, or when resorting to virtual memory on the other hand, has been confirmed by various runs.

This left unchanged our preceding conclusions, but makes them much more dramatic. For instance, in real time, the 8 processor version is always slower than the 4 processor one for 2D problems with $h^{-1} = 144$. On the other hand, for 3D problems with $h^{-1} = 72$, this means that the sequential version requires actually, for Problem 4, 40 min against 11 sec for the 8 processor version, and, for Problem 5, more than 2 hours against 22 sec when parallelizing.

Acknowledgement

This work presents research results of the Belgian Incentive Program “Information Technology”- Computer Science of the future, initiated by the Belgian State-Prime Minister’s Service-Federal Office for Scientific, Technical and Cultural Affairs (Contract No. IT/IF/14). The Scientific responsibility is assumed by its author.

This work was also supported by IBM through a research contract between ULB and IBM.

We thank the Interdisciplinary Center for Computer Based Complex System Research, Amsterdam (IC³A), for the facilities provided on their computer systems.

References

- [1] S. Ashby, M. Holst, T. Manteuffel, and P. Saylor, The role of the inner product in stopping criteria for the conjugate gradient method, Tech. Rep. UCRL-JC-112586, Lawrence Livermore National Laboratory, Livermore, CA 94551, 1992.
- [2] O. Axelsson and V. A. Barker, Finite Element Solution of Boundary Value Problems. Theory and Computation, Academic Press, New York, 1984.
- [3] O. Axelsson and G. Lindskog, On the eigenvalue distribution of a class of preconditioning methods, *Numer. Math.*, 48 (1986), 479–498.
- [4] P. Bastian and G. Horton, parallelization of robust multigrid methods: ILU factorization and frequency decomposition method, *SIAM J. Sci. Stat. Comput.*, 12 (1991), 1457–1470.
- [5] R. Beauwens and R. Wilmet, Conditioning analysis of positive definite matrices by approximate factorizations, *J. Comput. Appl. Math.*, 26 (1989), 257–269.
- [6] T. Chan, C.-C. J. Kuo, and C. Tong, Parallel elliptic preconditioners: Fourier analysis and performance on the Connection Machine, *Comput. Phys. Comm.*, 53 (1989), 237–252.
- [7] M. Clauss et al., PARIX 1.2, Software Documentation, PARSYTEC Computer GmbH, Aachen, 1993.
- [8] P. Concus, G.H. Golub, and G. Meurant, Block preconditioning for the conjugate gradient method, *SIAM J. Sci. Statist. Comput.*, 6 (1985), 220–252.
- [9] I.S. Duff and G.A. Meurant, The effect of ordering on preconditioned conjugate gradients, *BIT*, 29 (1989) 635–657.

- [10] V. Eijkhout, Analysis of parallel incomplete point factorizations, *Lin. Alg. Appl.*, 154–156 (1991) 723–740.
- [11] H. Elman and E. Agroń, Ordering techniques for the preconditioned conjugate gradient method on parallel computers, *Comput. Phys. Comm.*, 53 (1989), 253–269.
- [12] S. Filippone, M. Marrone, and G. Radicati di Brozolo, Parallel preconditioned conjugate-gradient type algorithms for general sparsity structures, *Journal of Computer Mathematics*, (1992) to appear.
- [13] M.M. Magolu, Implementation of parallel block preconditionings on a transputer-based multiprocessor, *Future Generation Computer Systems*, 11 (1995).
- [14] P. Manneback and J. Qin, Algorithmic on a distributed memory mimd computer: a case study, in *Proceedings of Transputers '92* M.B. et al., ed., Amsterdam, 1992, IOS Press, pp. 172–178.
- [15] J.A. Meijerink and H. A. Van Der Vorst, An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix, *Math. Comp.*, 31 (1977), 148–162.
- [16] S. Nakamura, *Computational Methods in Engineering and Science*, John Wiley & Sons, New York, 1977.
- [17] Y. Notay, Résolution itérative de systèmes linéaires par factorisations approchées, PhD thesis, Service de Métrologie Nucléaire, Université Libre de Bruxelles, Brussels, Belgium, 1991.
- [18] Y. Notay, A new incomplete factorization method, in *Incomplete Decomposition (ILU)-Algorithms, Theory and Applications*, W. Hackbusch and G. Wittum, eds., vol. 41 of *Notes on Numerical Fluid Mechanics*, Vieweg, Braunschweig, 1993, pp. 103–112.
- [19] Y. Notay, Ordering methods for approximate factorization preconditioning, Tech. Rep. IT/IF/14-11, Université Libre de Bruxelles, 1993.
- [20] Y. Notay, DRIC: a dynamic version of the RIC method, *Journ. Num. Lin. Alg. with Appl.*, 1 (1994), pp. 511–532.
- [21] Y. Notay, Parallel implementation of preconditioned iterative schemes by means of overlapping decomposition, submitted for publication, 1995.
- [22] Y. Notay and A. Van De Velde, Coarse-grid acceleration of parallel incomplete factorization preconditioners. submitted for publication, 1995.
- [23] J. Ortega, Orderings for conjugate gradient preconditionings, *SIAM J. Optimization*, 1 (1990), pp. 565–582.
- [24] G. Radicati Di Brozolo and Y. Robert, Parallel conjugate gradient-like algorithms for solving sparse nonsymmetric systems on a vector multiprocessor, *Parallel Computing*, 11 (1989), pp. 223–239.