JOURNAL OF COMPUTATIONAL PHYSICS **142,** 148–162 (1998)
ARTICLE NO. CP985916

Parallel Multigrid Preconditioning of the Conjugate Gradient Method for Systems of Subsurface Hydrology

Leesa Brieger* and Giuditta Lecca†

**Geophysics Group, †Environment Group, CRS4, I-09123 Cagliari, Italy*
E-mail: leesa@crs4.it, giuditta@crs4.it

Received July 17, 1997; revised December 24, 1997

Parallel preconditioners are considered for improving the convergence rate of the conjugate gradient method for solving sparse symmetric positive definite systems generated by finite element models of subsurface flow. The difficulties of adapting effective sequential preconditioners to the parallel environment are illustrated by our treatment of incomplete Cholesky preconditioning. These difficulties are avoided with multigrid preconditioning, which can be extended naturally to many processors so that the preconditioner remains global and effective.The coarse grid correction which defines the multigrid preconditioner is outlined and its parallel implementation with the distributed finite element data structure is presented, along with some examples of its use as a parallel preconditioner. © 1998 Academic Press

Key Words: multigrid preconditioning; parallel preconditioners; conjugate gradient method; parallel iterative methods.

1. INTRODUCTION

Mathematical models of flow and transport processes in porous media must be large and complex if they are to furnish realistic simulations of the increasingly urgent concerns of groundwater flow and contaminant transport in subsurface repositories. Numerical simulations can be used to follow the movement of groundwater and study water management strategies; they can be used to describe the migration of contaminants and predict the response of a system to remediation scenarios. The increased computing capacity—in speed and/or memory—of parallel machines, from networks of PCs to supercomputers, can put these large real-world simulations within reach.

Models of subsurface flow phenomena are often based on finite element (FE) formulations of the equations of flow and transport in porous media. FE codes use a large proportion of total computation time assembling and then solving the resulting large sparse systems;

the linear solver kernels are often the most computationally important components of the codes. A typical FE simulation of 3D subsurface flow with implicit time-stepping results in many large, sparse, symmetric positive definite linear systems to solve per time step, so that efficiency of the linear solver is particularly important to performance. We thus consider specifically the adaptation of these computational kernels to parallel machines [1, 2]. The environment we consider is coarse-grain parallelism, typical of networks of workstations and machines like the 28-node SP2 which we use for running our examples.

The FE model used to generate the linear systems for our study is based on the 3D nonlinear fluid continuity equation, commonly known as Richards equation [3], which describes water movement in porous media,

$$\sigma\frac{\partial\psi}{\partial t} = \nabla\cdot[K\nabla(\psi+z)]+q, \tag{1}$$

where ψ is the pressure head, $\sigma(\psi)$ is the general storage term, $K(\psi)$ is the unsaturated hydraulic conductivity tensor, z is elevation above a reference depth, and q is the injected/extracted volumetric flow rate per unit volume of porous medium.

The linear solver under study here is the conjugate gradient (CG) method [4] for solving sparse symmetric positive definite systems, such as those arising from our FE models. We are specifically looking for preconditioners for improving the convergence rate of the CG method when these systems are distributed over the processors of a parallel machine; we are investigating the efficiency of preconditioners adapted to the distributed computing environment. We distinguish between *effectiveness* of the preconditioning algorithm and its *efficiency* when implemented on a given machine. The effectiveness of a preconditioner is a measure of how much it speeds up convergence of the CG method (reducing the necessary number of iterations), while its efficiency is a measure of how it influences total calculation time to convergence (number of iterations balanced against cost per iteration) for that implementation.

The distribution of the numerical problem onto the parallel machine is driven by the FE discretization itself; a subdomain consisting of a cluster of elements is attributed to each processor. This is described in Section 2. While the parallel data structure is determined by a decomposition of the computational domain, it is slightly different from that used in typical domain decomposition techniques; there is not redundancy of information in the storage of coefficient matrices. Boundary nodes that are shared by more than one domain have *partial* values associated to them in the matrices on the respective processors. This determines the structure of interprocessor communications which are then used for implementing the conjugate gradient solver as a global method numerically independent of the domain decomposition.

We can use the same communication structure and mechanisms to generalize a typical sequential CG preconditioner for a parallel application in a very simple way; this is considered in Section 3. We have done so for the powerful sequential incomplete Cholesky (IC) preconditioner and present some of the results of the investigation in Section 4. However, the parallelization of such preconditioners in this manner can lead to a *local* algorithm, one whose effectiveness (convergence rate) depends, most unfortunately, on subdomain structure and thus, among other things, on the number of processors available at the moment of a run.

If we consider a multigrid (MG) preconditioner as presented in Section 5, it is possible to use the same communication structure to implement a natural parallel extension which remains global, i.e. numerically independent of number and structure of subdomains. Its effectiveness is that of the sequential algorithm; its *efficiency* depends further on implementation, problem size, subdomain structure, etc. Aside from its interest as a parallel algorithm, MG preconditioning for the CG method (MGCG) is intriguing inasmuch as this hybrid endows the conjugate gradient solver with multigrid's convergence properties, renowned for independence of mesh size [5]. The simple MG preconditioner we have implemented is presented along with the results of our investigation in Section 5. Comments on the extension of this preconditioner to CG-type solvers for nonsymmetric systems are also included in this section.

Our goal has been to study the parallel preconditioners in a general context, i.e. not restricted to regular or structured grids. However, our current investigations of the parallel multigrid preconditioner, taken up in Section 6, have all been carried out on regular grids, as a stepping stone to more general applications on unstructured grids. In fact, aside from the grid coarsening, which can also be adapted to unstructured grids, nothing in these algorithms depends on any structure in the underlying grid. What *does* depend heavily on grid structure is the communication pattern for a given implementation, and this is a determining factor in the efficiency of a code. Thus, some preprocessing of an unstructured grid would be in order to enhance communication efficiency. Some possibilities are considered in the next section.

2. PARALLEL DATA STRUCTURE

We use the structure of the FE discretization to distribute our numerical problem over a multiprocessor machine, attributing a cluster of finite elements to each processor. This attribution can be made without regard to the underlying geometry of the domain, at the admitted risk of complicating communication patterns and so rendering less efficient the application. We have used, for example, the following "automatic" method of domain decomposition: with ne elements and np processors, we define $k = \lfloor ne/np \rfloor$ and attribute the first k elements to the first processor, the second k elements to the second processor, etc., and let the npth processor contain $k + r$ elements, where $r = ne \bmod(np)$. A subdomain is just that cluster of elements (and the corresponding nodes) assigned to a given processor. This distribution of the problem onto the processors is a nongeometric domain decomposition which, when the size of the problem is large compared to the number of processors, is almost perfectly load-balanced. It also results in a *run-time* calculation of the domain decomposition, desirable when the number of processors is not fixed and can vary from run to run.

In our examples using structured regular grids, this domain decomposition has not led to a particular inefficiency in communication patterns. However, on an unstructured grid it would likely yield extremely irregular and inefficient interprocessor communications. When run-time calculation of the domain decomposition is a priority because, for example, the number of available processors changes with every run, ordering algorithms for reducing band width [6] should be applied to grid elements or nodes; this should help the automatic domain decomposition with communication efficiency for any number of subdomains. When run-time calculation of the domain decomposition is not important, other interface-minimizing domain decomposition techniques can be applied separately to partition the problem into a fixed number of subdomains [7].

For our parallel data structure, once the problem has been distributed onto the processors, only *local* assembly of the coefficient matrices is ever carried out; no global coefficient matrix is actually calculated. In our matrix storage scheme, there is no redundancy of information between processors; boundary nodes that are shared between domains have partial values associated to them in the matrices on the respective processors. (The global matrix would result from additively collecting the partial matrices from among the processors; $A = \sum_{k=1}^{np} A_k$, where np is the number of processors. This is not calculated, however.) In addition, we use a compressed sparse row (CSR) storage scheme to compact and store the local submatrices on each processor.

Vectors are distributed among processors according to the distribution of nodes among subdomains. A global vector $\mathbf{y} = (y_1, y_2, \ldots, y_N)^T$, where N is the number of nodes in the grid, is distributed into vectors $\mathbf{y}_k$ on the processors; $\mathbf{y}_k = (0, \ldots, 0, y_{i_1}, 0, \ldots, 0, y_{i_2}, 0, \ldots, 0, y_{i_{n_k}}, 0, \ldots, 0)^T$, where the nonzero elements y_{i_j} correspond to the n_k grid nodes which are resident on processor k. (For the implementation, these vectors are compacted and stored without the zero entries.) Notice that a global vector so distributed does engender a redundancy of information between processors; y_j will appear as an element of vector $\mathbf{y}_k$ for every processor k on which node j is resident. On the other hand, a distributed vector need not contain global values; if the $\mathbf{y}_k$ store results of local operations, then on each processor the elements of $\mathbf{y}_k$ will be local/partial values. This will often be the case in the parallel operations outlined below for which the the local vectors $\mathbf{y}_k$ contribute to a global vector $\mathbf{v}$ without redundancy: $\mathbf{v} = \sum_{k=1}^{np} \mathbf{y}_k$. In the following we drop the vector notation and write simply $v = \sum_{k=1}^{np} y_k$.

Since the distributed data structure is just another storage scheme for the large sparse matrices of the FE application and not a traditional domain decomposition, then implementing the CG method as a global method, independent of grid structure, is immediate. For example, to evaluate the matrix–vector multiplication $y = Ad$ (an operation on which the CG method for solving $Ax = b$ heavily depends), where d is a distributed vector of global values and A is a partially assembled distributed matrix, we use the additive operation described above to define the global vector y. First a partial result is calculated on each processor, $y_k = A_k d_k$ for $k = 1, \ldots, np$; this step is parallel, requiring no interprocessor communication. In terms of the implementation, y_k is just a notational representation of that portion of y which is resident on processor k; thus at this point y is the distributed vector of local values, containing partial results on each processor. Then interprocessor communications are invoked to collect the partial results and distribute them back into y: $y = \sum_{k=1}^{np} y_k$, or, implementationally speaking, $y \leftarrow \sum y$. After this, y is a distributed vector of global values. Our current implementation treats the collection of partial results as a *single node gather* problem and the distribution of the results after addition as a *single node scatter* problem [8].

Inner products are handled similarly, if more simply. In order to evaluate $s = (x, y)$, where x and y are distributed vectors of global values, first a local result is calculated, $s_k = (x_k, y_k)$, then it is additively collected into a scalar result which is distributed to all the processors. We treat the necessary communication steps as *single node accumulation* (collection of partial results) and *single node broadcast* (distribution of the global result) [8]. Note that the calculations at this step must take into account the redundancy present in the distributed storage of global vectors.

3. PARALLEL CG PRECONDITIONERS

In our study of parallel iterative solvers, we have adapted existing sequential preconditioning algorithms to coarse-grain parallel architecture. Preconditioned CG requires an additional operation with respect to CG without preconditioning, the calculation of $z = M^{-1}r$, where M^{-1} is a symmetric positive definite preconditioning matrix and r is a residual vector generated by the CG method. The better M^{-1} approximates A^{-1}, the faster the convergence of the preconditioned system. Interprocessor communications are inevitably an obstacle to parallel computing efficiency, so we have implemented the preconditioning operations using exactly these communication patterns and mechanisms already in place for the CG method itself. This is also desirable for programming ease.

In our parallel CG code, r is a distributed global vector. As long as the preconditioning matrix is stored in the same distributed, partially assembled manner as the coefficient matrix A, then the calculation of $z = M^{-1}r$ proceeds as the aforementioned matrix–vector multiplication $y = Ad$, using exactly the same communications and handling to additively collect local results into a global, distributed vector, $z = \sum_{k=1}^{np} z_k$, where $z_k = M_k^{-1} r_k$, $k = 1, \ldots, np$. On the other hand, it is possible to carry out preconditioned CG without explicitly calculating a preconditioning matrix. If we solve $Az = r$ only roughly for z, then $z \approx A^{-1}r$. Since a preconditioning matrix M^{-1} should approximate A^{-1}, we can consider that $z = M^{-1}r$, i.e. that M^{-1} is defined implicitly in this approximation of $A^{-1}r$. The only real constraint is that M^{-1}, even implicitly defined, must nevertheless be a symmetric, positive definite matrix for this preconditioner to be valid. If $Az = r$ is approximately solved with an iterative method using matrix–vector multiplications, then again the preconditioning step will use exactly the same communication calls and handling as implemented for the original CG method.

4. INCOMPLETE CHOLESKY PRECONDITIONING

IC is particularly effective in sequential applications for preconditioning systems typical of our problems and we consider it first. With the incomplete Cholesky-preconditioned conjugate gradient method (ICCG), the preconditioning matrix is defined as $M^{-1} = (LL^T)^{-1}$, where the lower triangular matrix L comes from an incomplete Cholesky decomposition of A, being allowed nonzero entries only where A has nonzero entries, so that $A \approx LL^T$. To minimize memory requirements, only the sparse matrix L is stored, and then the calculation of $z = M^{-1}r$ depends on the recursive forms for evaluating $y = L^{-1}r$ and $z = (L^T)^{-1}y$ from L. While extremely effective, every step of this algorithm is rigorously sequential, including the decomposition itself, which depends on recursion to define the elements of L.

How then to fit such a sequential list of instructions onto a parallel machine? With our distributed data structure, an obvious way of adapting this algorithm to several processors is to apply it locally—in parallel—processor by processor and then to collect local results into a global preconditioner. Implemented this way, an incomplete decomposition is computed on each processor, $A_k \approx L_k L_k^T$, and $z_k = (L_k L_k^T)^{-1} r_k$ is computed locally, so that each processor is independently executing its own sequential list of instructions. Then with interprocessor communications (via our usual communication handling), results are collected into a global vector $z = \sum_{k=1}^{np} z_k$, yielding the global preconditioner $M^{-1} = \sum_{k=1}^{np} (L_k L_k^T)^{-1}$, an approximation of $(LL^T)^{-1}$, and an even rougher approximation of A^{-1}.

While the simplicity of this approach is appealing and we parallelized ICCG in this manner, the preconditioner itself now depends on the number of processors and its effectiveness depends on the number and structure of the subdomains: the more submatrices there are, the more roughly M^{-1} approximates A^{-1} and the poorer the convergence; the more independent the submatrices from each other, the better M^{-1} approximates A^{-1} and the faster the convergence of the method. The method is no longer global, and our tests with this

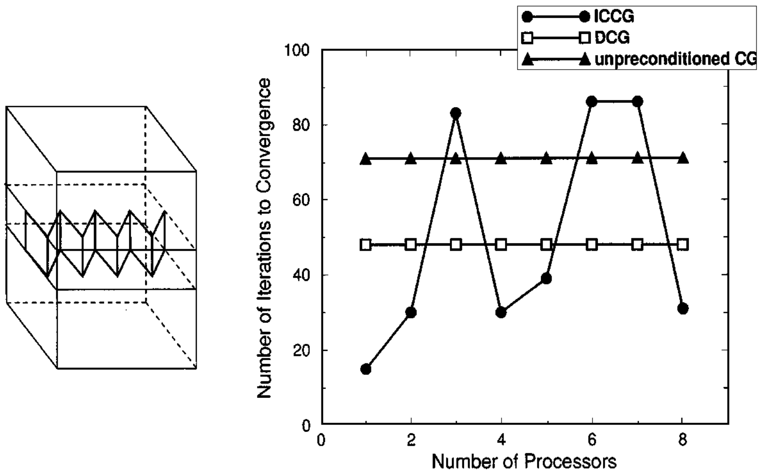

FIG. 1. Convergence behavior for parallel local ICCG for 3D flow system (28,577 unknowns) on the SP2; diagonal preconditioning and unpreconditioned CG included for comparison. Image on left illustrates a problematic domain decomposition for parallel ICCG.

preconditioner have shown a vulnerability to subdomain geometry in ways that surprised us. Rather than a gradual degradation of the parallel local ICCG method with an increase in the number of processors, we intermittently observed a convergence rate much worse than for CG without preconditioning! (See Fig. 1.) Thus, not only is this parallel preconditioner potentially ineffective at enhancing convergence rate, it can actually degrade performance of the CG method!

This phenomenon was flagrant in our test case, perhaps because of some particularities of the example, a small system (28,577 unknowns) which came from the simulation of 3D flow in a homogeneous medium (parallelepiped volume) with a point sink at a bottom corner and a point source at the opposite top corner. The convergence behavior for ICCG applied to this example is shown in Fig. 1. The numerical 3D finite element mesh for the test was generated by projecting a regular 2D triangular mesh, so that it is composed of layers of regular tetrahedra. The grid is 16 layers deep, composed of 19,200 elements (1200 per layer) and distributed among processors using the runtime domain decomposition described above with $k = \lceil 19200/np \rceil$ elements per processor (of the numbers of processors tested, only when $np = 7$ is there not a perfectly equal number of elements distributed to each processor); element numbering is layer-by-layer. Whenever the number of processors results in a layer-wise domain decomposition, the effectiveness of the preconditioner is good; on the contrary, whenever the number of processors results in a subdomain boundary which cuts a layer, as shown in the figure, the preconditioner actually degrades performance of the CG solver.

The conclusion is that this parallel adaptation of ICCG is not robust and cannot be used as a general-purpose preconditioner. It is included here for comparison and to demonstrate the potentially disastrous effect of locality on the parallel algorithm.

5. MULTIGRID PRECONDITIONING

The multigrid preconditioner we have implemented for the CG method consists of a simple coarse grid correction for approximately solving the system $Az = r$. Since we want the preconditioning matrix M^{-1} to approximate A^{-1}, we use this as the evaluation of

$z = M^{-1}r$. We must carry out operations for solving $Az = r$ in such a way that the implicitly defined M^{-1} is always a symmetric positive definite matrix, which we can do if we are careful with the coarse grid correction [9].

For completeness, we include here a short outline of the coarse grid correction method of solving a system $Ax = b$ using two grid levels. Suppose that x^* is the exact solution and that, using some relaxation method, we generate an approximation x; call this the presmoothing step. The unknown error is $e = x^* - x$, the residue is given by $res = b - Ax$, and the two are related in the following way: $Ae = res$. By solving this residual equation on a coarser discretization than the one that defines the original system, we can expect to generate a useful correction to x. Then, applying the relaxation method to the corrected approximation, we improve and smooth it; call this postsmoothing. With this multigrid approach, low-frequency components of the error are damped out more efficiently than by using the relaxation method on a single grid alone [10].

Since we are using the multigrid method to solve $Az = r$ for purposes of preconditioning and not to calculate an accurate solution of this system, we allow the residual equation $Ae = res$ to be solved only approximately on the coarser grid using a predefined number of relaxation steps; a typical multigrid solver would iterate to convergence at this step. Also for purposes of preconditioning, we use damped Jacobi as our relaxation method.

Prolongation and restriction are the intergrid transfer operations necessary to map a vector $v^{(c)}$ on the coarse grid to a vector v on the fine grid and vice versa. We use linear interpolation to define the prolongation operator P and full weighting [10] to define the restriction operator R, so that $R = \alpha P^T$, where α is some positive constant. Figure 2 illustrates the 2D version of these operators. The residual equation on the coarse grid necessitates the definition of the matrix $A^{(c)}$, a coarse-grid representation of the original matrix A. We have chosen simply to discretize the continuous operators on the coarser discretization to produce the symmetric, positive definite matrix $A^{(c)}$.

5.1. *Properties of the Preconditioning Matrix*

The use of the MG method for solving $Az = r$ as a preconditioner to the CG method is outlined below. Recall that the damped Jacobi method for solving $Az = r$ is defined by the

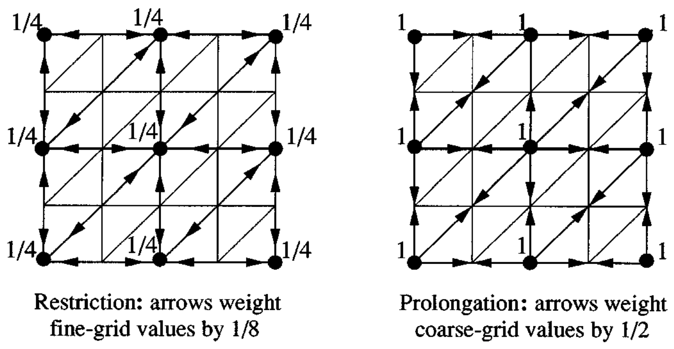
1/4
1/4
1/4
1/4
1/4
1/4
1/4
1/4
1/4
Restriction: arrows weight fine-grid values by 1/8
1
1
1
1
1
1
1
1
1
Prolongation: arrows weight coarse-grid values by 1/2

FIG. 2. Intergrid operators on a regular 2D triangular grid: the full weighting restriction operator weights by 1/8 the contributions from a coarse node's fine-grid neighborhood and by 1/4 its own fine-grid value to define its new value on the coarse grid; the prolongation operator uses linear interpolation between every two coarse-grid nodes to define fine-grid nodes (coarse-grid nodes retain their value—weight 1—on the fine grid).

splitting

$$A = \tilde{P} - \tilde{Q}, \tag{2}$$

where

$$\tilde{P} = \frac{1}{\omega} D, \quad \tilde{Q} = \frac{1}{\omega} D - A,$$

D is the diagonal of A, and ω is the relaxation parameter. The method is given by

$$z_{k+1} = H z_k + \tilde{P}^{-1} r, \tag{3}$$

where $H = \tilde{P}^{-1}\tilde{Q}$. For A symmetric positive definite, the method is convergent if and only if $\tilde{P} + \tilde{Q}$ is positive definite (see [11]). A sufficient condition for this is $0 < \omega < 1$, although, depending on the eigenstructure of A, this condition could prove unnecessarily restrictive. We use damped Jacobi also for the coarse grid iterations, with the same damping parameter ω, $0 < \omega < 1$, for the two grid levels, so that characteristics of the relaxation method are identical on the two grids. (Coarse-grid operators for damped Jacobi are tagged with (c) in the following.)

The steps in the coarse grid correction preconditioner:

- *presmoothing.* Apply m passes of damped Jacobi to $Az = r$, generating an approximate solution $z_m = H^m z_0 + Q_m r$, where $H = \tilde{P}^{-1}\tilde{Q} = I - \omega D^{-1} A$ and $Q_m = \sum_{j=0}^{m-1} H^j \tilde{P}^{-1}$; calculate residue $res = r - Az_m$.
- *restriction.* Restrict the fine-grid residue: $rhs^{(c)} = R(res)$; this will be the right-hand side of the residual equation on the coarse grid.
- *smoothing.* Apply n passes of damped Jacobi to the coarse-grid residual equation, generating a coarse-grid approximation to the error:

$$e_n^{(c)} = (H^{(c)})^n e_0^{(c)} + Q_n^{(c)} rhs^{(c)}.$$

- *prolongation.* Interpolate $e_n^{(c)}$ back to the fine grid: $e_n = P(e_n^{(c)})$.
- *correction*: $\tilde{z}_m \leftarrow z_m + e_n$.
- *postsmoothing*. Apply k passes of damped Jacobi to additionally smooth the corrected solution: $z = H^k \tilde{z}_m + Q_k r$.

With more than two grid levels, the algorithm is recursive; a coarse grid correction using a third-level coarser grid would be used to improve the solution of $Ae = res$ on the second-level grid, etc. The simple, recursive use of the coarse grid correction defines the multigrid V-cycle. We have not considered using more complex MG schemes for preconditioning.

If we use the zero vector as the starting point for the damped Jacobi iterations on the fine and coarse grids, i.e. $z_0 = 0$ and $e_0^{(c)} = 0$, then the entire sequence of operations in the above multigrid algorithm can be represented as a matrix, let us call it M^{-1}, operating on the right-hand side r, so that $z = M^{-1} r$. Under appropriate conditions, this matrix will be symmetric and positive definite and the MG method will be suitable for preconditioning the CG method. Consider

$$\begin{aligned} z &= H^k \tilde{z}_m + Q_k r \\ &= H^k (z_m + e_n) + Q_k r \end{aligned}$$

$$\begin{aligned}
&= H^k\left(Q_m r + P e_n^{(c)}\right) + Q_k r \\
&= H^k Q_m r + Q_k r + H^k P Q_n^{(c)} rhs^{(c)} \\
&= H^k Q_m r + Q_k r + H^k P Q_n^{(c)} R(res) \\
&= H^k Q_m r + Q_k r + H^k P Q_n^{(c)} R(I - AQ_m)r. \qquad (4)
\end{aligned}$$

Then, because $I - AQ_m = (H^T)^m$ and $P = (1/\alpha)R^T$, z can be written as $z = M^{-1}r$, where

$$M^{-1} = H^k Q_m + Q_k + \frac{1}{\alpha} H^k R^T Q_n^{(c)} R(H^T)^m. \qquad (5)$$

As long as this matrix is symmetric positive definite, we may use the coarse grid correction outlined above for CG preconditioning. Since the matrix $\tilde{P}$ of our splitting of A is symmetric, as is $\tilde{P}^{(c)}$ from the splitting of $A^{(c)}$, then the matrices Q_k, $Q_n^{(c)}$, and $H^k Q_m$ are symmetric for all positive integers k, m, and n. If, in addition, we impose that $k = m$, then the matrix $H^k R^T Q_n^{(c)} R(H^T)^m$ is also symmetric. Thus, as long as we perform an equal number of pre- and postsmoothing passes of our relaxation method, M^{-1} is a symmetric matrix.

Now to check its positive definiteness. Since we require that $k = m$, the preconditioning matrix can be rewritten

$$\begin{aligned}
M^{-1} &= H^m Q_m + Q_m + \frac{1}{\alpha} H^m R^T Q_n^{(c)} R\left(H^T\right)^m \\
&= Q_{2m} + \frac{1}{\alpha} H^m R^T Q_n^{(c)} R\left(H^T\right)^m. \qquad (6)
\end{aligned}$$

Recall that we require $0 < \omega < 1$ as relaxation parameter for damped Jacobi—on both grid levels. This ensures the positive definiteness of $\tilde{P} + \tilde{Q}$ on the fine grid and of $\tilde{P}^{(c)} + \tilde{Q}^{(c)}$ on the coarse grid, guaranteeing that Q_{2m} is always positive definite and that $Q_n^{(c)}$ is positive definite for n even. Since $\tilde{P}^{(c)}$ is a positive definite matrix, then $Q_n^{(c)}$ will also be positive definite for n odd. Thus $H^m R^T Q_n^{(c)} R(H^T)^m$ and, consequently, M^{-1} are positive definite for all positive integers n and m. (See [11] for a review of properties for the splitting of symmetric positive definite matrices.)

Therefore, using damped Jacobi as the relaxation method with damping parameter ω between 0 and 1, with the same number of relaxation passes for pre- and postsmoothing, we are guaranteed the positive definiteness of M^{-1} and we may use the two-grid coarse grid correction as a CG preconditioner. In a similar way, we may inductively show that the coarse grid correction, subject to the same conditions as outlined above, on a hierarchy of many grids, also defines a symmetric positive definite preconditioning matrix and so it can serve as a CG preconditioner.

5.2. *Performance of MGCG*

Once the coarse grid correction has been adopted as a CG preconditioning strategy, even within the constraints of defining a symmetric positive definite preconditioning matrix, a variety of MGCG methods that can be used. These depend on the relaxation method chosen, the number of grids used to implement coarse grid corrections, the number of relaxation passes executed on each grid level, etc. The choices in the implementation of a method will

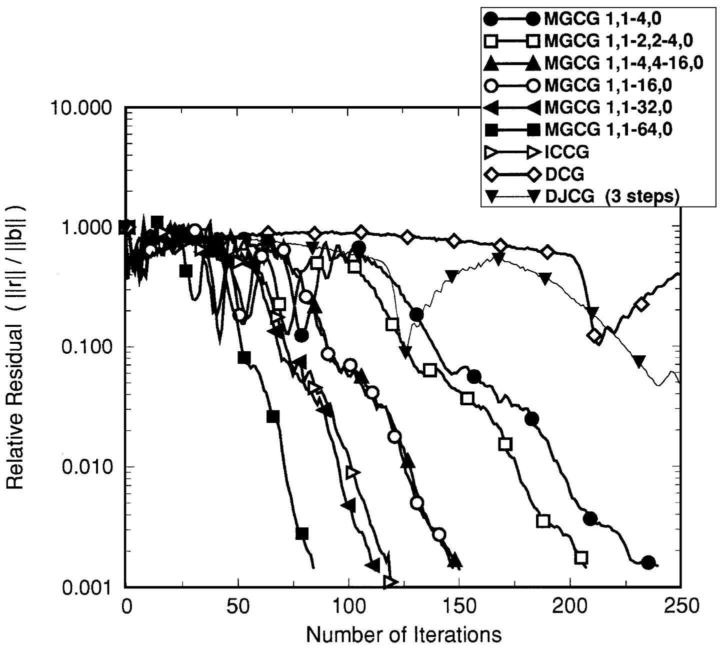

FIG. 3. MGCG behavior on a system of 126,225 unknowns; DCG, ICCG, and DJCG are included for comparison. In the legend, $1, 1 - j, k(-m, n)$ indicates one pre- and one postsmoothing pass on the finest grid level, j pre- and k postsmoothing passes on the coarser level (and m pre- and n postsmoothing passes on the coarsest level when there is a third-level grid). Note: these are all sequential implementations.

depend on the balance between effectiveness of the preconditioner and the cost of using it. To minimize the cost and complication of an MG preconditioner, it is tempting to consider only the most rudimentary coarse grid implementation—and indeed, this may be sufficient in some cases. Yet, to endow the MGCG method with multigrid's renowned independence from mesh size and so to produce an effective method when problems are large and difficult, it may be necessary to admit costlier MG methods.

In our test cases, even with the loosest coarse grid corrections (few grid levels and few relaxation passes on them) it is possible to define effective preconditioners. We have typically used only two or three grid levels. Figure 3 shows the behavior of several MGCG variants applied to a system of 126,225 unknowns generated by the simulation of 3D flow in a porous medium containing obstacles of very low hydraulic conductivity. For each MGCG method, the number of grids and the number of relaxation passes on each grid are indicated in the legend: $1, 1 - j, k(-m, n)$ indicate one presmoothing and one postsmoothing pass on the finest grid level, with j presmoothing and k postsmoothing passes on the coarser level (and m presmoothing and n postsmoothing passes on the coarsest level if there is a third-level grid). In each case, on the coarsest grid where only smoothing takes place, the number of postsmoothing steps is, of course, zero. Results for diagonal and IC preconditioned CG (DCG and ICCG) are included for comparison, along with an example of a simple one-grid damped Jacobi (DJCG) preconditioner, included to better illustrate the contribution of the *multigrid* correction to preconditioner effectiveness.

For a given computational effort, a method can be made more effective if more of this effort is displaced to coarser grids. It is even possible to improve effectiveness and reduce computational effort by distributing the computation over the grids. This has a price, however, in the increased memory requirements necessary for additional grid levels. Consider the relative cost and effectiveness of the MGCG methods of Fig. 3 as sequential implementations. Because of the structure of our regular grid in these test cases, each coarse grid contains eight times fewer nodes than the next finer grid. Thus we approximate that eight passes of damped Jacobi at a coarse grid level is roughly equivalent to one pass of damped Jacobi at the next finer grid level. The simple one-grid damped Jacobi preconditioned method DJCG costs more in these terms than the two-grid MGCG (1, 1–4, 0) method per iteration, but it requires 434 iterations to converge, as opposed to only 239 for MGCG(1, 1–4, 0). The MGCG(1, 1–16, 0) method uses the equivalent of four fine-grid matrix–vector multiplications, while, for the same effectiveness, MGCG(1, 1–4, 4–16, 0) uses the equivalent of only slightly more than three fine-grid matrix–vector multiplications. MGCG(1, 1–4, 4–16, 0) is less costly in computing time, but demands additional memory in order to achieve this.

To what extent then do these MGCG methods inherit mesh-size independence? Since we had three grid levels for the example problem of Fig. 3, we were able to run the two-grid MGCG methods on two different mesh discretizations of the same problem. Table 1 shows the number of iterations, n_s and n_l, necessary for the various methods (including DCG and ICCG) to reach convergence on the small problem (17,289 unknowns) and on the "large" problem (126,225 unknowns), respectively. For a method truly independent of mesh size, the ratio of these two, n_l/n_s, would be 1. For our methods, we see, instead, a tendency toward mesh-size independence as the MGCG method uses a more accurate (and costlier) coarse grid correction.

Another thing to notice is that MG preconditioning is not necessarily restricted to the CG method for symmetric positive definite systems. With other Krylov methods for nonsymmetric systems, we may also use coarse grid corrections to define preconditioners and without the preoccupation of a preconditioning matrix which must be symmetric positive definite. The definition of relaxation methods adapted to the coarse grid corrections may be less straightforward than for the symmetric positive definite case, but even nonconvergent relaxations could prove useful for preconditioning.

TABLE 1
Effect of Problem Size on Convergence Rate

Iterative method	No. iterations n_s for small problem	No. iterations n_l for large problem	n_l/n_s
DCG	298	[a]	—
ICCG	56	119	2.13
MGCG:			
1, 1–2, 0	131	[a]	—
1, 1–4, 0	115	239	2.08
1, 1–8, 0	96	188	1.96
1, 1–16, 0	78	147	1.88
1, 1–32, 0	69	112	1.62
1, 1–64, 0	53	84	1.58

Note. Each two-grid MGCG method is implemented twice for a sample problem: once on a small mesh of 17,289 unknowns and once on a larger mesh of 126,225 unknowns. Variants of MGCG differ from each other in the number of coarse-grid relaxation passes. n_l/n_s is a measure of independence of each method from mesh size.

[a] No convergence to within desired tolerance.

6. PARALLEL MGCG

As pointed out above, with MG preconditioning for a problem distributed over several processors, we would like to use the same communication patterns and mechanisms as are already implemented for the CG method itself. First consider the operations necessary to implement MG and the interprocessor communications they require. The intergrid prolongation operator, based on linear interpolation between nodes of a coarse-grid vector $v^{(c)}$ to define the vector v at fine-grid nodes, does not require any message passing; as long as this parallel operator is acting on a *global* distributed vector for which there is redundancy of information on nodes shared between processors, the linear interpolation can be carried out locally on each processor without any need of interprocessor communications. See the 2D prolongation operator in Fig. 4 for help in visualizing why this should be so.

The interprocessor communications used by the parallel MG preconditioner arise then during the matrix–vector multiplication of the damped Jacobi relaxation and during application of the intergrid restriction operator. The matrix–vector multiplication is the same as already in place, so of course, it does use exactly the same communication mechanisms already established for the CG method. As for the restriction operator, consider again Fig. 4 which illustrates the parallel intergrid operators for a regular 2D triangular grid. This operator uses full weighting to define a coarse-grid vector $v^{(c)}$ from a vector v defined on the next finer grid; weighted contributions from values at the fine-grid neighbors of a coarse node are added to define a coarse-grid value at that node (see also Fig. 2). Recall from the outline of the coarse grid correction scheme given in Section 5.1 that the restriction operator is only ever used to calculate vectors of the form

$$\begin{aligned} rhs^{(c)} &= R(res) \\ &= R(r - Az) \\ &= R(r) - R(Az) \\ &= r^{(c)} - y^{(c)}. \end{aligned}$$

The calculation of $y = Az$ itself entails first the computation of local/partial values on each processor ($y_k = A_k z_k$), normally followed by the communication step in which partial values

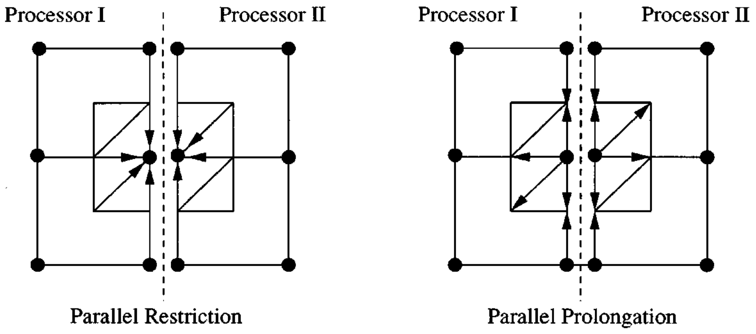

FIG. 4. Parallel intergrid operators: effect of parallel restriction and prolongation at boundaries. At shared nodes global vectors contain redundant information (and so prolongation requires no communication) and local vectors contain partial information (and so restriction requires a matrix–vector multiplication-style communication).

are collected and added and the results distributed into global vector $y = \sum_{k=1}^{np} y_k$. Knowing that the result $y = Az$ is in this case destined for restriction, we are better off avoiding that last communication step and applying the restriction operator locally to the partial values in the y_k. Only then should the results be collected into a global distributed vector:

$$\begin{aligned} y^{(c)} &= R(y) \\ &= \sum_{k=1}^{np} R(y_k) \\ &= \sum_{k=1}^{np} R(A_k z_k). \end{aligned}$$

The restriction operator thus acts locally on a distributed vector of partial values; then with the collection of partial results into a global vector $y^{(c)}$, we arrive at the same result as if we had applied the restriction operator to the global vector $y = Az$. This implementation results in the communication step being carried out on the coarser level, so that it is even less expensive—in terms of communication cost—to evaluate $R(Az)$ than to evaluate Az. Thus, supposing that r is also stored in terms of its partial values on the processors, then the evaluation of $R(res) = R(r - Az)$ can be carried out, not only using the same communication scheme as already in place, but with some economy of effort.

The relative costs of the MGCG methods of Fig. 3 should be reconsidered now in terms of the parallel implementations of these algorithms. Since interprocessor communications are the most expensive (slowest) part of these parallel applications, we evaluate relative costs only in terms of these communications. Because each coarse grid contains eight times fewer nodes than the next finer grid, we suppose that this translates into subdomain boundaries with roughly four times fewer nodes than the same boundary at the next finer level. Thus we estimate that messages passed at a coarse level will be four times smaller and so four times less costly than at the next finer level. Each pass of damped Jacobi requires a matrix–vector multiplication, replete with all the necessary communications—except at the step just before a restriction, when we can delay the communications until the restriction step and so apply them on the coarser grid.

Thus the simple one-grid damped Jacobi preconditioner is more costly than the two-grid MGCG(1, 1–4, 0) method and remains much less effective. The MGCG(1, 1–16, 0) method uses the equivalent of $5\frac{1}{4}$ fine-grid matrix–vector multiplications, while, for the same effectiveness, MGCG(1, 1–4, 4–16, 0) uses the equivalent of $4\frac{1}{4}$ fine-grid matrix–vector multiplications. In the typical trade-off between computing efficiency and memory, the more effective versions have more important memory requirements for handling additional grids.

One can also see that MGCG(1, 1–32, 0) (rather costly at around nine fine-grid matrix–vector multiplications per CG iteration) is about as effective as the typically very effective *sequential* ICCG. The big difference is that parallelizing the IC preconditioner in the simple way we have done results in a completely unreliable algorithm, while parallelizing the MG preconditioner just as simply changes nothing in the algorithm, and it retains all its effectiveness. In addition, the MGCG method should enjoy some useful speedup when implemented as a parallel method.

The curves in Fig. 5 show the speed of solution (1/[solution time]) versus the number of processors for two parallel MGCG methods applied to another test system of 126,225

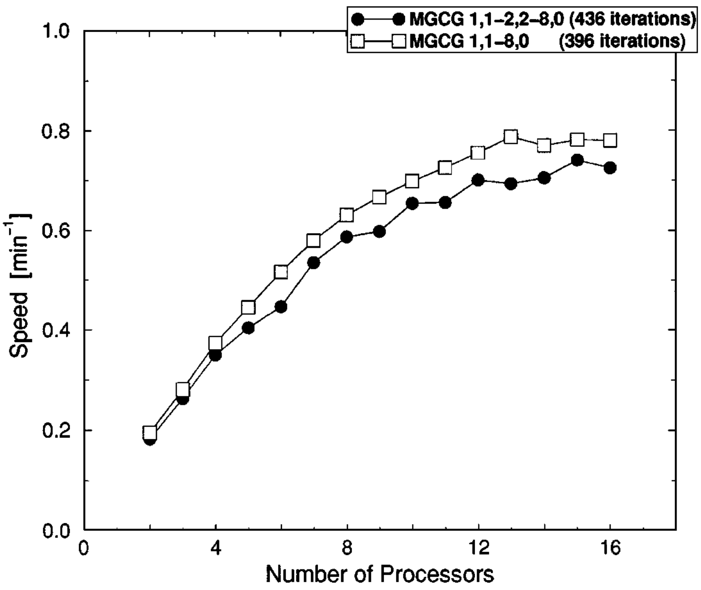

FIG. 5. Speed (1/[solution time]) of solving a test system with a two-grid and with a three-grid parallel MGCG method vs number of processors used for solving the system.

unknowns. The increase of this speed when computing with more processors flattens out as increased communication requirements overcome the benefit of increased concurrency. We have not optimized the present implementations of parallel MGCG; we would expect to profit considerably in computing time by doing so. In addition, the benefit of parallel computation (speedup) is typically much more marked on very large systems; we have not yet run MGCG for very large problems and so we have not yet tested its full potential. In fact, we expect parallel MGCG to have its greatest impact on very large and difficult problems.

ACKNOWLEDGMENTS

This work was supported by the Sardinian Regional Authorities and the Center for Advanced Studies, Research and Development in Sardinia (CRS4), with additional support for Giuditta Lecca from the EC AVICENNE Project AVI-CT93-2-073. The authors thank Carlo Nardone and Ernesto Bonomi for helpful hints during the preparation of this article.

REFERENCES

1. L. Brieger and G. Lecca, Data parallelism in finite element computation, in *Proc. of the X International Conference on Computational Methods in Water Resources,* edited by A. Peters, B. Herrling, U. Meissner, G. Wittum, C. A. Brebbia, W. G. Gray, and G. F. Pinder (Kluwer Academic, Dordrecht, 1994), Vol. II, p. 1515.
2. L. Brieger and G. Lecca, Parallel multigrid preconditioning for finite element models of groundwater flow, in *Proc. of the XI International Conference on Computational Methods in Water Resources*, edited by A. A. Aldama, J. Aparicio, C. A. Brebbia, W. G. Gray, I. Herrera, and G. F. Pinder (Kluwer Academic, Dordrecht, 1996), Vol. I, p. 507.

3. G. Gambolati, G. Pini, G. Putti, and C. Paniconi, Finite element modeling of the transport of reactive contaminants in variably saturated soils with LEA and non-LEA sorption, in *Environmental Modelling*, edited by P. Zannetti (Comput. Mech., Southampton, UK, 1994), Vol. II, p. 173.
4. M. Hestenes and E. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bur. Stand.* **49**, 409 (1952).
5. W. Hackbusch, *Solution of Large Sparse Systems of Equations* (Springer-Verlag, New York, 1994).
6. A. D. P. Z. Chinn, J. Chvatalova, and N. E. Gibbs, The bandwidth problem for graphs and matrices—a survey, *J. of Graph Theory* **6**, 223 (1982).
7. C. Farhat and H. Simon, TOP/DOMDEC: A software tool for mesh partitioning and parallel processing, Technical Report CU-CSSC-93-11, Center for Space Structures and Controls, University of Colorado, Boulder, Colorado, 1993.
8. D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation* (Prentice Hall, Englewood Cliffs, New Jersey, 1989).
9. O. Tatebe, The multigrid preconditioned conjugate gradient method, in *Sixth Copper Mountain Conference on Multigrid Methods*, edited by N. D. Nelson, T. A. Manteuffel, and S. F. M. Cormick (NASA, Hampton, VA, 1993), Vol. CP 3224, p. 621.
10. W. L. Briggs, *A Multigrid Tutorial* (SIAM, Philadelphia, 1987).
11. J. M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems* (Plenum Press, New York, 1988).