INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
Int. J. Numer. Meth. Fluids **30**: 995–1008 (1999)

PARALLELIZATION OF THE ILU(0) PRECONDITIONER FOR CFD PROBLEMS ON SHARED-MEMORY COMPUTERS

LAURA C. DUTTO* AND WAGDI G. HABASHI

CFD Laboratory, Department of Mechanical Engineering, Concordia University, Montréal, Québec, Canada H3G 1M8

SUMMARY

The use of ILU(0) factorization as a preconditioner is quite frequent when solving linear systems of CFD computations. This is because of its efficiency and moderate memory requirements. For a small number of processors, this preconditioner, parallelized through coloring methods, shows little savings when compared with a sequential one using adequate reordering of the unknowns. Level scheduling techniques are applied to obtain the same preconditioning efficiency as in a sequential case, while taking advantage of parallelism through block algorithms. Numerical results obtained from the parallel solution of the compressible Navier–Stokes equations show that this technique gives interesting savings in computational times on a small number of processors of shared-memory computers. In addition, it does this while keeping all the benefits of an ILU(0) factorization with an adequate reordering of the unknowns, and without the loss of efficiency of factorization associated with a more scalable coloring strategy. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: CFD; preconditioners; ILU(0) factorization

1. INTRODUCTION

The ability of solving large, sparse, unsymmetric, typically ill-conditioned linear systems remains crucial for reducing the computational time of computational fluid dynamics (CFD) problems. Direct methods were in the past preferred to iterative ones, because of their predictability and reliability. The increase in size and in physical complexity of three-dimensional models, however, makes direct methods prohibitively costly, both in terms of storage and computation. Preconditioned iterative methods are, therefore, currently playing a major role in these kind of problems and are widely used to accelerate the solvers of a variety of discretized boundary value problems. The classical difficulties in developing general purpose preconditioners are amplified for parallel computations, with parallelism usually achieved by sacrificing efficiency. Thus, it is quite frequent to have scalable algorithms with good speed-ups in a parallel environment, but with little savings when compared with more powerful sequential methods.

Although the behavior of classical preconditioners is not fully predictable in general, approaches such as the well-known ILU(0) factorization are frequently used because of their

* Correspondence to: CFD Laboratory, Department of Mechanical Engineering, Concordia University, Montréal, Québec, Canada H3G 1M8.

CCC 0271–2091/99/160995–14$17.50
Copyright © 1999 John Wiley & Sons, Ltd.
Received December 1997
Revised July 1998

simplicity and reasonable memory requirements. In this case, the preconditioner is a product of two sparse triangular matrices L and U, where the sparse structure of $L+U$ is chosen to be identical to that of the system matrix. This work presents some results using this type of preconditioner on commonly available parallel machines or networks of workstations, with 2–8 processors sharing a fairly large aggregate memory. Although the incomplete Gaussian factorization is essentially a sequential preconditioner, some parallelism could be achieved both during the factorization and the backward–forward solution step *without modifying its behavior*. In particular, some reordering strategies can be used for breaking its sequential nature [1–3]. If the lower and upper triangular parts of the system matrix could be put in a block form, where the diagonal blocks are diagonal matrices, both the incomplete factorization and the solution of each triangular factor could be done simultaneously inside each block of equations.

In this work, level scheduling techniques are applied to the parallel solution of the compressible Navier–Stokes equations, for a small number of processors in shared-memory computers. The main advantage of these kinds of ordering is that the coloring *does not modify* the preconditioner itself. The efficiency of the proposed technique depends on the matrix, and no effort is done here to attempt to modify its structure. Instead, the initial ordering of equations *is supposed to be suitable to a good sequential ILU(0) factorization*, giving a robust sequential preconditioner, and additional savings are obtained using a small number of processors.

The matrix A is supposed to be a sparse large-scale system of order N. If its zero–non-zero structure is symmetric, as it is commonly the case for finite element methods, about half of the preprocessing step can be reduced. In this case, a row and column reordering for the lower part of A, where the diagonal blocks are diagonal matrices, is also an appropriate reordering for the upper part of the system. Nevertheless, the symmetry is not a necessary condition here, and this study can be applied to non-symmetric matrices as well. For the actual implementation of methods presented here, the compressed sparse row (CSR) storage format was used to store the non-zero coefficients of the matrix (see SPARSKIT2 [4] for details on sparse storage schemes). However, the algorithms can be rewritten for other storage schemes. See [1] for modifications to the compressed sparse column (CSC) or jagged diagonal (JD) storage schemes. In the following sections, only lower triangular systems are considered most of the time. The development for upper triangular systems is similar.

2. BLOCKING ALGORITHMS FOR SPARSE TRIANGULAR SYSTEMS

Given an $N \times N$ matrix $A=(A_{ij})$ it is possible to define an induced directed graph $G(A)=\langle V, E\rangle$. The set V has N vertices $\{a_1, \ldots, a_N\}$ and E is a collection of ordered pairs of elements of V such that $\langle a_i, a_j\rangle \in E$ if $i \neq j$ and if $A_{ij} \neq 0$. An element $\langle a_i, a_j\rangle \in E$ is called an edge of $G(A)$, and an arrow goes from a_i to a_j in the graph. Reciprocally, given $G=\langle V, E\rangle$, a directed graph without loops, the structure of a matrix $A=(A_{ij})$ could be associated with the graph in the usual way: $A_{ij} \neq 0$ if and only if either $i=j$ or $\langle a_i, a_j\rangle \in E$. The simple directed graph shown in Figure 1 will be used in the following to exemplify the strategies presented here.

A 'natural' ordering of the unknowns respects the direction of the arrows. The numbered graph (left) and the structure of the lower triangular matrix (right) associated with this ordering are shown in Figure 2.

Copyright © 1999 John Wiley & Sons, Ltd.

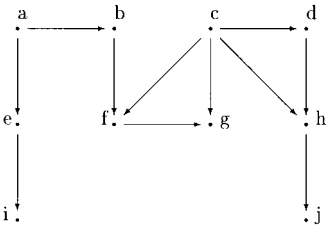

Figure 1. Example of a directed graph.

Let L be a sparse lower triangular matrix of order N. The forward substitution to solve the system $L\mathbf{w} = \mathbf{v}$ can be written in a compact form:

$$w_i = \frac{1}{l_{ii}} \left(v_i - \sum_{j<i,\ l_{ij} \neq 0} l_{ij} w_j \right), \quad i = 1, \ldots, N. \tag{1}$$

If L is dense, its coefficients are non-zero and each of the components $w_1, \ldots, w_{i-1}$ must be known to solve for w_i. However, when L is sparse, only few of its coefficients are non-zero and it may not be necessary to solve for all of the $i-1$ first components of $\mathbf{w}$ before solving for w_i. A set of contiguous values of the unknowns can be computed in parallel if they are mutually independent in $L\mathbf{w} = \mathbf{v}$, i.e. if in the associated graph the corresponding vertices are independent. In graph theory, *a partitioning* of the vertices of a graph into sets of independent vertices is called *coloring*. After coloring, the outer loop of the substitution can be parallelized *color by color*. In general, parallelization benefits from color sets that are as large as possible. Unfortunately, finding a coloring scheme that provides a minimum number of colors in an arbitrary graph is an NP-complete problem.

Several techniques of coloring are currently used to create sets of independent vertices in different kind of meshes, like the well-known red–black coloring for regular finite differences meshes or simply greedy algorithms for general unstructured finite element grids. See [5] for some coloring strategies well adapted to distributed-memory parallel computers. A multi-coloring algorithm applied to the graph of Figure 1 is shown in Figure 3, as well as its associated lower triangular matrix.

In this example, colors are represented by different figures around numbers. Thus, the first color is used for vertices numbered 1–4, the second one for vertices 5–8, and the last one for vertices 9 and 10. Unfortunately, several arrows are *reversed* when compared with the initial graph. This means that the ILU(0) factorization associated with this matrix *is not the same* as for the initial ordering. As it is known for the sequential case [6,7], reordering of the unknowns modifies the convergence rate of the preconditioned iterative methods [8]. It is out of the scope of this paper to compare the influence of various colorings and orderings on the convergence rate of linear methods. Instead, for a small to moderate number of processors on

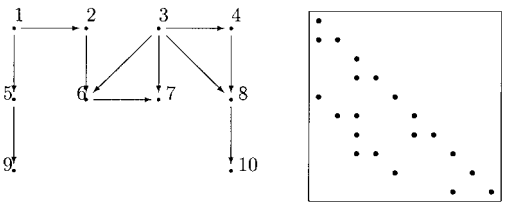

Figure 2. Natural order of the initial graph.

Copyright © 1999 John Wiley & Sons, Ltd.

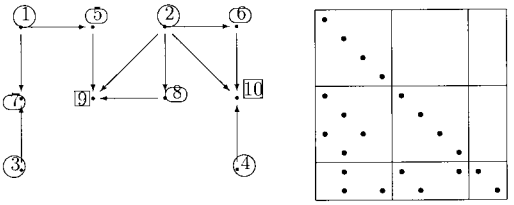

Figure 3. Multicoloring order of the initial graph.

shared-memory parallel computers, some coloring can be performed *without modifying* the preconditioner itself. A *level set* structure of the graph will be constructed in such a way that all vertices *in the same level* can be used at the same time. The efficiency of the proposed technique depends on the matrix itself, and no effort is done here to modify its structure. However, for the technique to be interesting, it is assumed that some previous effort was done *to find an ordering of the unknowns suitable* for an ILU(0) preconditioner.

The basic idea is to reorder the rows and columns of the coefficient matrix to obtain a block triangular system in which the diagonal blocks are diagonal matrices. If such a block partitioning of the matrix is possible, the inverses of the diagonal blocks will be easy to compute, and a block form of the sequential algorithm can be used. It is clear that the sequential algorithm is obtained without reordering if the dimension of the diagonal blocks is one. However, as it was pointed out in [2], the structure of many triangular systems allows a reordering of the rows into a moderate number of equivalence classes, where the rows in each class are independent and can be solved in parallel. Figure 4 shows an example of a lower triangular system where the diagonal blocks D_i, $i = 1, \ldots, m$ are diagonal matrices.

Assume that such non-trivial reordering is possible, and let $\mathscr{L}_1, \mathscr{L}_2, \ldots, \mathscr{L}_m$ (with $m \leq N$) be the equivalence classes defining the partitioning of the rows. The block forward substitution to solve $L\mathbf{w} = \mathbf{v}$ is summarized in Algorithm 2.1.

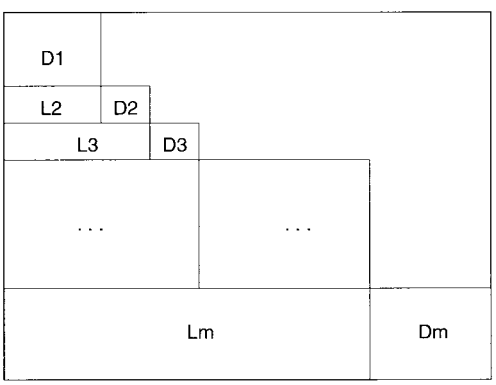

Figure 4. Block partitioning of a lower triangular system.

Copyright © 1999 John Wiley & Sons, Ltd.

Algorithm 2.1
Level set strategy for forward substitution.

For $v = 1, \ldots, m$:
  Computer in parallel for all $i \in \mathscr{L}_v$:
  $w_i = (1/l_{ii})(v_i - \Sigma_{j<i,\ l_{ij} \neq 0}\ l_{ij} w_j)$.
End loop v.

The number of levels represents the inherent number of sequential steps in solving the triangular system. If this number is small compared with the number of rows, the parallelism in level scheduling should be high.

A similar algorithm can be written for a backward substitution using the matrix U.

The parallel strategy given by algorithm 2.1 can also be applied to compute the incomplete Gaussian factorization of a general matrix A of order N, where its triangular factors L and U are restricted to have the same sparsity structure of A. If a level structure $\mathscr{L}_1, \mathscr{L}_2, \ldots, \mathscr{L}_m$ associated with the graph of the lower triangular part of A is available, the whole process can be described by Algorithm 2.2.

Algorithm 2.2
Level set strategy for incomplete Gaussian factorization

For $v = 1, \ldots, m$, do in parallel $\forall i \in \mathscr{L}_v$:
  For $k = 1, \ldots, i-1$ with $a_{ik} \neq 0$ do:
    $l_{ik} = -(a_{ik}/a_{kk})$,
    For $j = k+1, \ldots, N$ such that $a_{ij} \neq 0$ and $u_{kj} \neq 0$ do:
      $a_{ij} = a_{ij} - l_{ij} u_{kj}$
    End loop j.
  End loop k.
  Rename $u_{ij} = a_{ij}$, $j \geq i$.
End loop v.

3. LEVEL SCHEDULING METHODS

In this section, a class of reordering method is described that puts the coefficient matrix into a block form in which the diagonal blocks are diagonal matrices. These reorderings are called *level scheduling* methods after the way they are represented on the adjacency graph of the matrix. Important special cases are *forward level scheduling*, in which each row in a lower triangular system is solved at the earliest possible level, and *backward level scheduling*, in which each row is solved at the latest possible level.

Reordering the nodes of the graph is equivalent to reordering the rows and columns of the coefficient matrix. In general, the numerical values of the incomplete factors of A depend on the order in which computation takes place and thus, a physical reordering of the matrix is usually performed. For level scheduling methods, this is not necessary because the values of the coefficients are independent from the order of computation inside each set, provided that the level structure is respected. Instead, two additional `integer` vectors help to set up the

Copyright © 1999 John Wiley & Sons, Ltd.

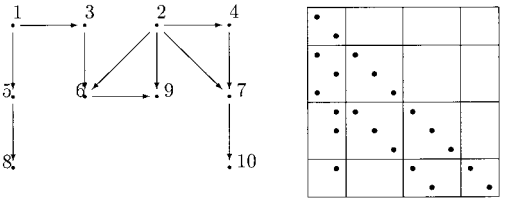

Figure 5. Forward level scheduling of the initial graph.

partitioning from which the set of rows in each level can be obtained. Thus, if the rows of a matrix are partitioned into m sets, a permutation vector `IRENU` of length N indicates the order in which the rows will be processed, and an index vector `LEVEL` of length $m+1$ points to the beginning of each level in `IRENU`. Therefore, the vth level for $1 \leq v \leq m$ is made up of the rows in locations `LEVEL`(v) to `LEVEL`($v+1$) -1 of `IRENU`, giving the set $\mathscr{L}_v$ used to describe algorithms 2.1 and 2.2. Hence the row numbers in these locations of `IRENU` are solved or factorized in iteration v of the outer loop of these algorithms.

Let $E(i)$ and $F(i)$ be two auxiliary sets:

$$E(i) = \{ j < i | l_{ij} \neq 0 \}, \qquad F(i) = \{ j > i | l_{ji} \neq 0 \}.$$

$E(i)$ has the indices of non-zero coefficients in the row i, while $F(i)$ has the respective ones in the column i.

In *forward level scheduling* each node is assigned to the first level in which it can reside. To be more precise, let *root* be an imaginary node linking all the nodes having no predecessors. In this way, the *minimum depth* of a node can be defined as the length of the longest path from the root to that node.

The minimum depth of each node can be computed with one pass through the row structure of L. For each i such that $E(i) = \varnothing$, `mindep`(i) $= 1$. Otherwise,

$$\texttt{mindep(i)} = 1 + \max_{j \in E(i)} \{\texttt{mindep(j)}\}. \tag{2}$$

All rows with the same depth are put in the same level set, and the number of levels is simply:

$$m = \max\{\texttt{mindep(i)}, \quad \text{i} = 1, \text{N}\}.$$

Once the number of levels and the depth of each row have been determined as above, the remainder of the scheduling algorithm consists of finding an ordering of the rows by increasing depth, and setting up an index vector to the start of each new level.

A forward level scheduling applied to the initial graph of Figure 1 gives a graph and a block triangular lower matrix as in Figure 5.

It is easy to verify in this example that the order of the unknowns respects the direction of the arrows. This is true in general, so that the ILU(0) factorization is *the same* as for the initial ordering. The lower triangular system has the expected blocked form, where the diagonal blocks are diagonal matrices. In general, the dimension of the diagonal blocks are smaller than those of a matrix reordered by a coloring technique, needing more communication and synchronization points in a parallel strategy.

In forward level scheduling, each node is assigned to the level of its minimum depth. The opposite scheduling strategy would be to assign each node to the level of its maximum depth, where the maximum depth is the furthest distance from the root that a node may appear in the graph without increasing the number of levels of the graph. This strategy, called *backward level*

Copyright © 1999 John Wiley & Sons, Ltd.

scheduling, is about the same as forward level scheduling in the parallelism of the triangular solves. Assume the number of levels in a graph is m. The maximum depth can be defined recursively with one pass through the row structure of L. For each i such that $F(i) = \varnothing$, $\texttt{maxdep}(i) = m$. Otherwise,

$$\texttt{maxdep(i)} = \min_{j \in F(i)} \{\texttt{maxdep(j)}\} - 1. \tag{3}$$

The remainder of the backward scheduling algorithm is the same as for forward scheduling, i.e. find an ordering of the rows by increasing depth, and set up an index vector to the start of each new level.

Both of these strategies can be regarded as only two special cases of level scheduling. A node v_i for which $\texttt{mindep}(i) < \texttt{maxdep}(i)$ can appear in any level in the range $[\texttt{mindep}(i), \texttt{maxdep}(i)]$, provided that care is taken to ensure that the ancestors of v_i appear in earlier levels and that the descendants of v_i appear in latter levels. A node satisfying $\texttt{mindep}(i) < \texttt{maxdep}(i)$ can be considered as a free node, and the difference $\texttt{maxdep}(i) - \texttt{mindep}(i)$ is the *degree of freedom* of the node.

In this work, a simplified implementation of the *backward load balancing* strategy defined by Anderson [1], is proposed. The main idea is to make the number of nodes at each level a multiple of the number of processors by moving the free nodes between levels. First, the minimum and maximum depths of each node are computed and a linked list of nodes at each level is set up as for backward level scheduling. Beginning at the last level m, each set of nodes is analyzed. If for the current set $\mathscr{L}_v$ it is possible to take a positive multiple of p nodes with r nodes left over, the idea is to move to $\mathscr{L}_{v-1}$ the r nodes with the greatest degrees of freedom. In principle, if a node v_i is moved from level v to level $v-1$, all the ancestors of node v_i, which are in level $v-1$, need to be moved to a previous level set, and so on. To avoid excessive swapping, only nodes *having no ancestors in the previous level* are moved from $\mathscr{L}_v$ to $\mathscr{L}_{v-1}$. If there is not enough free nodes satisfying this condition, the sets $\mathscr{L}_v$ and $\mathscr{L}_{v-1}$ are unchanged. In this way, there is only a small overhead between this algorithm and the simple forward and backward scheduling methods, and hopefully some supplementary parallelism can be recuperated. The proposed algorithm is summarized in algorithm 3.1, where P is the available number of processors, $\#\mathscr{S}$ denotes the number of elements of $\mathscr{S}$, and $\text{mod}(q, p)$ is the unique integer r such that $0 \le r < p$ and $(p - r)$ is a multiple of q.

Algorithm 3.1
Simplified backward load balancing level scheduling strategy

- Build the backward level structure associated with L: $\mathscr{L}_1, \mathscr{L}_2, \ldots, \mathscr{L}_m$.
- For $v = m, m-1, m-2, \ldots, 2$ such that $\#\mathscr{L}_v > P$ and $r = \text{mod}(\#\mathscr{L}_v, P) > 0$ do:
  Define $\mathscr{M}_v = \{v_i \in \mathscr{L}_v \,|\, \texttt{maxdep}(i) > \texttt{mindep}(i)$ and v_i has no ancestors in the set $\mathscr{L}_{v-1}\}$.
  If $\#\mathscr{M}_v \ge r$, move r nodes from $\mathscr{M}_v$ to $\mathscr{L}_{v-1}$ and delete them from $\mathscr{L}_v$.
  End loop v.

In the actual implementation of algorithm 3.1, some operations can be omitted. For example, it is not necessary to build the whole set $\mathscr{M}_v$ at each time, but a queue vector can be used instead. Thus, at most, r nodes are saved at each time, chosen among those having the highest degrees of freedom. If some of them have ancestors in the previous level, they are rejected from the queue, and the following equations (ordered by degree of freedom) are taken into account. The process ends when r equations satisfy both conditions, or when there are no more equations to be considered.

Copyright © 1999 John Wiley & Sons, Ltd.

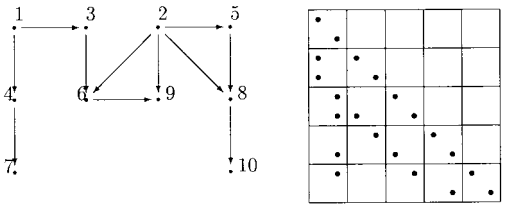

Figure 6. Load-balancing level scheduling of the initial graph.

Figure 6 shows a numbered graph and its associated lower triangular matrix obtained by the load balancing level scheduling algorithm applied to the initial graph of Figure 1. Here again the direction of the arrows in the graph are preserved, and so the ILU(0) factorization associated with this ordering *is the same* as for the initial ordering. Assuming that the number of processors is two, the theoretical speed-up is optimal with this ordering, because each level set has a number of vertices that is a multiple of the number of processors. In practice, however, the small size of the level sets penalizes this optimality.

4. NUMERICAL RESULTS

The test cases presented here are not arbitrarily chosen but come from specific CFD applications. The ILU(0) preconditioner is used in the computation of the steady solution of the compressible Navier–Stokes equations for subsonic laminar flows. The numerical tests were performed using FENSAP (Finite Element Navier–Stokes Analysis Package) [9]. This package, developed bv the CFD Laboratory at Concordia University, solves the compressible 2D and 3D Euler and Navier–Stokes equations in primitive variables form, in Cartesian or cylindrical co-ordinates. Finite element methods with hexahedra having bilinear shape functions were used for the space discretization. A fixed value of an artificial viscosity coefficient was used in each case. The system of equations was linearized using the Newton method, and at each non-linear iteration, the linear system was solved by right-preconditioned GMRES(k) [10], with $k = 50$ for all the test cases, except for the 2D Cascade, where $k = 100$, and the 2D NACA0010, where $k = 75$. The relative precision of the final residual in each linear system was 10^{-6}. For the overall non-linear problem, several Newton iterations are necessary to reduce

Table I. Description of the test cases

Description	Nodes	N	NNZ	M_∞	Re	Angle of attack (°)	
						α	β
2D cascade	4464	6347	166 725	0.20	100	0	90
2D cylinder	4854	6956	180 588	0.10	100	0	90
3D straight pipe	2739	8769	453 163	0.01	100	90	0
2D NACA0010	16 742	24 476	648 940	0.60	2000	0	90
3D concentric annulus	6000	21 392	2 157 000	0.01	100	0	90
3D lid-driven cavity							
(i) Grid $24 \times 24 \times 12$	8125	26 639	2 502 173	0.01	100	90	0
(ii) Grid $32 \times 32 \times 16$	18 513	63 679	6 202 685	0.01	100	90	0

Copyright © 1999 John Wiley & Sons, Ltd.

Table II. Information concerning level scheduling methods

Description	N	m	Ave Eq	Max Eq	Free Eq	Ave DOF
2D cascade	6347	729	8	16	2884	18
2D cylinder	6956	482	14	30	6442	24
3D straight pipe	8769	755	11	33	7853	52
2D NACA0010	24 476	940	26	46	6382	244
3D concentric annulus	21 392	3352	6	8	964	17
3D cavity $24\times24\times12$	26 639	588	45	87	3333	16
3D cavity $32\times32\times16$	63 679	796	79	148	5989	21

the norm of the residual equations to 10^{-10}. A description of the test cases is presented in Table I, where 'Nodes' is the number of nodes of the discretized domain, 'N' indicates the total number of unknowns of the system, 'NNZ' is the number of non-zero coefficients, 'Re' is the Reynolds number and 'M_∞' is the Mach number of the uniform flow.

The nunerical tests were run on two shared-memory SGI computers. Most of them were run on an IRIX 8 CPU Mips R4000, 32 bits, with 64 Kb of data cache size, 256 Kb of secondary data cache size and 128 Mb of memory. The 3D lid-driven cavity with a grid $32 \times 32 \times 16$ was run on a Power Challenge KL with 4 CPU Mips R8000, 64 bits, with 16 Kb of data cache size, 4 Mb of secondary data cache size and 512 Mb of memory. Computational times are shown in seconds. When the computations are carried out in parallel, the computational time consists of the maximum time among processors, including synchronization and communication time.

Some information related to the level scheduling reorderings concerning these test cases is shown in Table II. The order of the linear system N and the number of levels m are shown. 'Ave Eq' indicates the average number of equations by level, while 'Max Eq' is the maximum number of equations in a level; 'Free Eq' is the number of equations having a positive degree of freedom (DOF), and 'Ave DOF' indicates the average degrees of freedom of free equations.

The available parallelism depends strongly on the test case, as it can be seen from Table II. Thus, the 2D NACA0010 is very adequate for this kind of strategy, because of its relatively small number of levels and a large number of free equations. On the other hand, the 3D concentric annulus has a very small average number of equations by level, but it is almost constant. Also, free nodes are concentrated in a small percentage of sets, allowing little flexibility in modifying the level structure using algorithm 3.1.

A simple model is proposed to predict the behavior of block algorithms for several processors. The idea is to count the number of steps needed to solve a lower triangular system put in this form. Thus, for the sequential case, the number of steps is equal to the number of equations N. For the parallel case, it depends on the number of levels as well as the size of each one. If the number of equations inside $\mathscr{L}_v$ is q, with $q = kP + r$, the number of steps performed at level v is k for $r = 0$ and $k + 1$ for $0 < r < P$. The speed-up can be estimated by the number of equations over the total number of steps to be performed when using P processors. This model gives a coarse idea of the parallel behavior of the applied strategy. In particular, the number of operations inside each row is not taken into account. Nevertheless, finer estimations are not justified in practice, as it can be seen from the numerical results. The number of non-zero coefficients by row is small because of sparsity, and the total cost is dominated by the loop overhead and the synchronization costs.

Theoretical speed-ups using this simple model are shown in Table III. The differences between forward (or backward) and load balancing level scheduling are not significant in general. This observation is verified in the total computational time spent by each algorithm.

Copyright © 1999 John Wiley & Sons, Ltd.

Table III. Theoretical speed-up for level scheduling methods

Description	Forward					Load balancing				
	Number of processors									
	2	4	8	16	∞	2	4	8	16	∞
2D cascade	1.90	3.43	5.71	8.71	8.71	1.91	3.49	5.75	8.71	8.71
2D cylinder	1.94	3.62	6.39	10.15	14.43	1.96	3.69	6.47	10.29	14.43
3D straight pipe	1.92	3.54	6.06	9.23	11.61	1.96	3.67	6.30	9.33	11.61
2D NACA0010	1.96	3.78	6.93	12.39	26.04	1.96	3.80	7.12	12.30	26.04
3D concentric annulus	1.85	3.32	6.38	6.38	6.38	1.86	3.32	6.38	6.38	6.38
3D cavity $24 \times 24 \times 12$	1.97	3.85	7.40	13.46	45.30	1.99	3.93	7.66	14.13	45.30
3D cavity $32 \times 32 \times 16$	1.99	3.97	7.69	14.76	80.00	2.00	3.98	7.81	15.18	80.00

The expected speed-up for a large number of processors is also shown in Table III. In general, it is not interesting to use more than a moderate number of processors, say 6–10, but sometimes the method seems to handle as much as 80 processors. These results are not validated by the numerical experiments because of the reasons mentioned before, and only a small number of processors can benefit from this technique.

A summary of results is shown in Table IV. Several Newton iterations were necessary to reduce the residual norm by 10^{-10}, given by 'Newton Iterat'. On the other hand, 'Linear Iterat' shows the total number of linear iterations. 'CPU' is the total computational time, in seconds, for the sequential case. It includes not only preparing computations, such as the level scheduling reordering, but also assembly and factorization steps, repeated at each Newton iteration. For p processors, $p > 1$, the ratio between the time spent by the sequential run and the respective one using p processors is shown, to facilitate comparison. For $p > 1$, the computational time consists of the maximum time among processors, including communication.

The results presented in Table IV show that, while the parallel strategy used to factorize and solve the triangular systems is not interesting for a large number of processors, it gives speed-ups of up to 2.8 for four processors, and 3.9 for eight processors.

Table IV. Computational time using level scheduling recording schemes

Description	Newton Iterat	Linear Iterat	Number of processors						
			CPU	Speed-up					
			Forward				Load balancing		
			1	2	4	8	2	4	8
2D cascade	5	390	692.	1.49	1.90	2.17	1.46	2.01	2.16
2D cylinder	6	383	728.	1.59	1.95	2.23	1.44	2.11	2.46
3D straight pipe	7	193	2241.	1.83	2.88	3.92	1.82	2.87	3.99
2D NACA0010	6	797	4466.	1.44	1.84	2.01	1.44	1.82	1.97
3D concentric annulus	9	391	7159.	1.76	2.76	3.56	1.63	2.73	3.68
3D cavity $24 \times 24 \times 12$	7	447	7779.	1.62	2.59	2.88	1.66	2.66	2.96
3D cavity $32 \times 32 \times 16$	8	1213	4990.	1.79	1.95	—	—	—	—

Copyright © 1999 John Wiley & Sons, Ltd.

Table V. Detail of computational time spent in the linear system solution

Operations	2D NACA0010				3D concentric annulus			
	Number of processors							
	CPU	Speed-up			CPU	Speed-up		
	1	2	4	8	1	2	4	8
Matvec	472.	1.98	3.37	4.00	697.	2.03	3.43	3.93
Precond	1386.	1.58	2.43	2.70	1694.	1.63	2.39	2.58
Others	1343.	1.06	1.03	1.02	348.	1.03	1.03	0.99

The causes of the stagnation in the total computational time observed when the number of processors increases are twofold. First at all, the level scheduling scheme depends on the original ordering of the equations. If the number of levels is very high, the synchronization points are numerous, decreasing the efficiency of the parallel strategy. Second, the operations between vectors coming from BLAS Level 1 (`daxpy`, `ddot`, etc.) parallelize very poorly on shared-memory computers [11]. When the available number of processors increases, the computational cost of vector–vector operations dominates the total cost of the preconditioned iterative linear solver, and savings in the time involved in the preconditioning step cannot help in reducing the total cost of the algorithm. This is illustrated with results in Table V. 'Matvec' indicates the time spent during matrix–vector operations, 'Precond', during the forward–backward substitution and 'Others' states essentially the time spent during vector–vector operations. In the GMRES method, the last part is mainly the orthonormalization process of the Krylov basis vectors. The parallel behavior of different operations is similar for both test cases. For the 2D NACA0010 test case, vector operations are very time consuming during the solution step. They represent about 42% of the total time spent by the linear solver in a sequential run. This computational time does not diminish when the number of processors increases. This partialy explains why the global speed-up in this case is worse than for the 3D concentric annulus.

At the beginning of this work it was said that it was out of the scope of this research to compare the influence of different orderings of the unknowns in the global performance of the ILU(0) factorization. However, this is not a minor point in the global reduction of the computational time spent in the Navier–Stokes equations using ILU(0) preconditioners. The robustness and efficiency of the preconditioners is highly depending on the initial order of the unknowns. A 'good' general ordering for these kind of preconditioners does not exists, but it varies from one test case to another. Thus, before performing intensive computations on a test case, it is critical to choose an appropriate ordering of the unknowns resulting in a robust preconditioner. It is unuseful to have a preconditioner scaling very well, but with a very poor global perforrmance.

To give an idea of the importance of this point, several sequential preconditioners obtained using different orderings of the unknowns are compared for some of the test cases presented in this work. For details on the used orderings, see [7]. Also, see [5] for an interesting comparison of scalable orderings. The orderings used in this comparison are the following:

- *orig*: Original order of the unknowns, coming with the geometry. Its quality depends strongly on the effort done during the spatial discretization.

Copyright © 1999 John Wiley & Sons, Ltd.

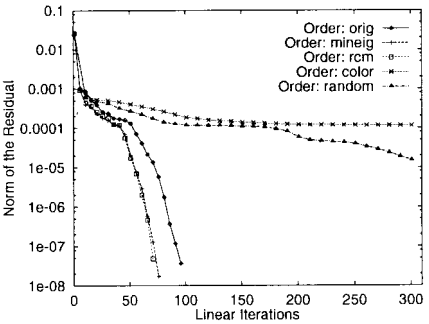

Figure 7. 2D-cascade: ILU(0) preconditioner with different ordering of unknowns.

- *rem*: Reverse Cuthill–McKee algorithm [12,13]. Classical algortithm used to minimize the bandwidth of a matrix.
- *mineig*: modification of the minimum degree algorithm [14] which reject the new relationships produced by the elimination of a vertex in the structure of the graph associated with the matrix [15].
- *random*: The equations are ordered in a random manner.
- *color*: Greedy multicoloring algorithm [4]. Neighboring equations have different colors.

The multicoloring algorithm parallelizes very well, because the number of colors used is quite small in all the cases, producing large sets to be processed in parallel. However, the behavior of ILU(0) using this ordering, as well as the *random* one, is really poor compared with *rcm* or *mineig* orderings.

Figures 7–9 show the convergence history of the linear system coming from the first Newton iteration of the 2D cascade, 2D NACA0010 and 3D concentric annulus respectively. The cost of each iteration is exactly the same for all the orderings used, because the number of non-zero coefficients is the same for each preconditioner.

As it can be seen, the *random* and *color* orderings perform very poorly. They produce less robust preconditioners, and even if *color* gives a well-scalable one, the parallel computation cannot beat the sequential algorithm in these test cases because of the degradation in the convergence of the preconditioned iterative solver. Therefore, even if the level scheduling algorithms seem to give a moderate parallelization, they can be very useful to effectively reduce the total computational time spent in the solution of the linear systems involved here when coupled with a good sequential preconditioner.

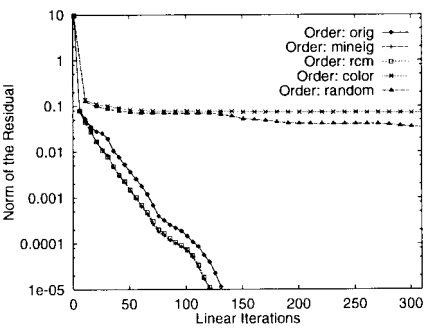

Figure 8. 2D-NACA0010: ILU(0) preconditioner with different ordering of unknowns.

Copyright © 1999 John Wiley & Sons, Ltd.

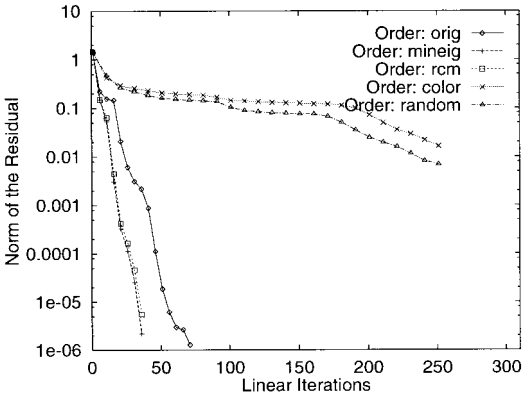

Figure 9. 3D-concentric annulus: ILU(0) preconditioner with different ordering of unknowns.

As mentioned at the beginning of this section, only one artificial viscosity cycle was done in each test case. This coefficient was adjusted such that the linear systems were 'solvable' when preconditioning with ILU(0) factorizations. The bigger the artificial viscosity coefficient is, the better conditioned is the linear system to solve. It is possible to choose a coefficient for which the ILU(0) factorization obtained with the *color* ordering performs better than here, allowing the user to take advantage of the better parallelization behavior associated with it. This can be useful, for example, in time marching algorithms with small time steps. In the cases presented here, however, a level scheduling ordering coupled with a performant sequential preconditioner is a better strategy for reducing the total computational time when using a small number of processors.

5. CONCLUSIONS

The use of incomplete Gaussian factorization as preconditioner is quite frequent when solving linear systems coming from CFD computations. because its efficiency and reasonable memory requirements. For the well-known ILU(0) factorization, the preconditioner is a product of two sparse triangular matrices L and U, where the sparsity structure of $L+U$ is chosen to be identical to that of the system matrix. This work presents some results using this preconditioner on commonly available shared-memory computers with a small number of processors sharing a fairly large aggregate memory. Level scheduling techniques are applied to the parallel solution of the compressible Navier–Stokes equations. The numerical results show that the parallel strategy used to factorize and solve the triangular systems, while not useful for a large number of processors, gives interesting savings in the computational time for a small number of processors (from 4 to 6).

Since the preconditioner is the same in the sequential and in the parallel case, the solver keeps all the benefits of a suitable ILU(0) factorization, without the loss of efficiency of a factorization obtained with a more scalable coloring strategy (as the red–black reordering). These methods involve some preprocessing overhead and are primarily of interest in solving many systems with the same coefficient matrix, as in an iterative procedure with ILU preconditioning, or with several different matrices all having the same structure. Both of these conditions are verified in the test cases considered here, and the preprocessing overhead becomes negligible in practice. The implementation of the level scheduling ordering is quite

Copyright © 1999 John Wiley & Sons, Ltd.

simple, and the additional memory requirements is only a couple of `integer` vectors of maximum length N for each triangular system to solve.

ACKNOWLEDGMENTS

The authors would like to acknowledge the financial support of NSERC (Canada) and FCAR (Québec) for this work. The provision by the Department of Mathematics of the University of Ottawa of space and computer connection resources to Dr Laura C. Dutto, Research Assistant Professor at the CFD Laboratory, is warmly acknowledged.

REFERENCES

1. E.C. Anderson, 'Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations', *Master's Thesis*, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1988 (also as *Technical Report 805*).
2. E.C. Anderson and Y. Saad, 'Solving sparse triangular systems on parallel computers', *Int. J. High Speed Comput.*, **1**, 73–96 (1989).
3. Y. Saad, 'Krylov subspace methods on supercomputers', *SIAM J. Sci. Stat. Comput.*, **10**, 1200–1232 (1989).
4. Y. Saad, SPARSKIT: A basic tool kit for sparse matrix computations. *Technical Report*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL, 1990.
5. C. Pommerell, M. Annaratone and W. Fichtner, 'A set of new mapping and coloring heuristics for distributed-memory parallel processors', *SIAM J. Sci. Stat. Comput.*, **13**, 194–226 (1992).
6. I.S. Duff and G.A. Meurant, 'The effect of ordering on preconditioned conjugate gradients', *BIT*, **29**, 1635–657 (1989).
7. L.C. Dutto, 'The effect of ordering on preconditioned GMRES algorithm, for solving the compressible Navier–Stokes equations', *Int. J. Numer. Methods Eng.*, **36**, 457–497 (1993).
8. G. Heiser, C. Pommerell, J. Weis and W. Fichtner, 'Three-dimensional numerical semiconductor device simulation: Algorithms, architectures, results', *IEEE Trans. Comput-Aided Des.*, **10**, 1218–1230 (1991).
9. C. Lepage, A Guide to FENSAP. Version 4.10. *Internal Technical Report*, CFD Laboratory, Concordia University, Montreal, Canada, May 1998.
10. Y. Saad and M. Schultz, 'GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems', *SIAM J. Sci. Stat. Comput.*, **7**, 856–869 (1986).
11. C. Robitaille, 'Une expérience de parallélisation d'un code d'eléments finis', *Master's Thesis*, Université Laval, Ste-Foy (Québec), Canada, April 1996.
12. E.H. Cuthill and J.M. McKee, 'Reducing the bandwith of sparse symmetric matrices', *Proc. 24th Natl. Conf. of the Association for Computing Machinery*, Brondon Press, New Jersey, 1969, pp. 157–172.
13. A. George, 'Computer implementation of the finite element method', *Ph.D. Thesis*, Departement of Computer Science, Stanford University, Stanford, USA, 1971. Also as *Research Report STAN CS-71-208*, Stanford University, Stanford, USA.
14. A. George and J.W.H. Liu, 'The evolution of the minimum degree ordering algorithm', *SIAM Review*, **31**, 1–19 (1989).
15. G. Martin, 'Méthodes de préconditionnement par factorisation incomplète', *Master's Thesis*, Université Laval, Québec, Canada, 1991.

Copyright © 1999 John Wiley & Sons, Ltd.