How to Compute the Velocity of a Rotating Flow

L. K. Lundin

April 2, 1997

Abstract

Let A be an m-by-n matrix with $m > n$. We consider how to solve $\min_x ||Ax-b||_2$ by use of the Preconditioned Conjugate Gradient algorithm for Least Squares (PCGLS) on a distributed parallel computer. The vector x represents the unknown velocity field of a 3-dimensional rotating flow. Two levels of structure in A are revealed and exploited in the computation. The aspects of using a parallel computer for this problem are discussed, and results for a wide range of parallel computers are presented. The parallel speed-up is typically 80% of the number of processors used, depending on the architecture.

Keywords: Cauchy-Riemann equations, overdetermined linear system of equations, conjugate gradients, least squares, parallel computers.

1 Introduction

Suppose $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ with $m > n$. In the following we will investigate how to efficiently solve

$$\min_x ||Ax - b||_2 \tag{1}$$

where A and b arise from a specific physical problem, namely the rotating flow of an incompressible fluid inside a closed cylinder. The fluid is brought into motion by letting the lid at one end rotate with a constant velocity. If the state of the fluid is known through its velocity at an initial time and at the boundary the problem is to compute the velocity and vorticity at a given later time.

One way to do this is to use a second-order finite-difference discretization of the Navier-Stokes equations in a velocity-vorticity formulation. With this approach one can advance the solution through time in two steps. In the first step the vorticity at a new time level is computed from the vorticity and velocity at the previous time level. In the second step the velocity at the new time level is computed from the vorticity at the new time level. The second step, which is the most computationally demanding [6], leads to (1).

2 The Matrix A

To find the elements of A we need to look at equations of the flow. The governing equations are the Navier-Stokes equations in a vorticity-velocity formulation:

$$\begin{aligned} \frac{\partial \omega}{\partial t} + \underline{\nabla} \times (\omega \times v) &= \frac{1}{R_e}\underline{\nabla} \times \underline{\nabla} \times \omega && (2) \\ \underline{\nabla} \times v &= \omega && (3) \\ \underline{\nabla} \cdot v &= 0. && (4) \end{aligned}$$

Equation (2) which is used in the first step of the solution method will not be discussed here. Equation (3) is the definition of vorticity. Equation (4) is a continuity equation that expresses that the net flow of an incompressible fluid through an arbitrary closed part of the cylinder is zero. The velocity is found as a solution to the last two equations, which are of Cauchy-Riemann type. The reason for this name is that the original Cauchy-Riemann equations define a 2-dimensional field that is both irrotational and solenoidal, i.e. $\underline{\nabla} \times v = 0$ and $\underline{\nabla} \cdot v = 0$. The equations in our case differ by being 3-dimensional and by the ω on the right hand side of (3).

Our Cauchy-Riemann equations in cylindrical coordinates are

$$
\begin{aligned}
\frac{\partial(rU)}{\partial r} + \frac{\partial V}{\partial \theta} + r\frac{\partial W}{\partial z} &= 0 && (5)\\
\frac{\partial W}{\partial \theta} - r\frac{\partial V}{\partial z} &= r\omega_1 && (6)\\
\frac{\partial U}{\partial z} - \frac{\partial W}{\partial r} &= \omega_2 && (7)\\
\frac{\partial(rV)}{\partial r} - \frac{\partial U}{\partial \theta} &= r\omega_3. && (8)
\end{aligned}
$$

These equations constitute an overdetermined system, with four equations and three unknowns, (U, V, W). Note that due to the cylindrical geometry there is a periodic boundary condition in the θ-direction.

2.1 Discretization.

The r, θ and z-intervals of the domain are partitioned into N_r, $N_\theta - 2$ and $N_z - 1$ subintervals, respectively. The definition of N_θ and N_z may seem awkward at this point but will be practical for the later discretization of (2). The gridpoints are indexed by i, j and k, where $i = 0, 1, \ldots, N_r$, $j = 1, 2, \ldots, N_\theta - 2$ and $k = 1, 2, \ldots, N_z$.

The spatial resolution is therefore

$$(\Delta r, \Delta\theta, \Delta z) = (\frac{R}{N_r}, \frac{2\pi}{N_\theta - 2}, \frac{H}{N_z - 1}).$$

We discretize our Cauchy-Riemann equations by a second-order finite-difference scheme that leads to an overdetermined linear system of equations. The discretized versions of equations (5), (6), (7) and (8) will be denoted Z0, Z1, Z2 and Z3, respectively.

2.1.1 Summary of the Discretized Equations.

Our discretized Cauchy-Riemann equations will be denoted

$$Ax = b, \tag{9}$$

where the vector x represents the unknown velocities. We have $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$, where the number of equations, m, is of the order $4N_r N_\theta N_z$ and the number of unknowns, n, is of the order $3N_r N_\theta N_z$.

Thus the linear system of equations is overdetermined, with roughly 4 equations per 3 unknowns.

Since $Ax = b$ is overdetermined we will henceforth seek the solution to (1).

2.2 The Structure of A

The rows in A correspond to the equations Z0, Z1, Z2, Z3. Our ordering of the equations is based on the grid coordinates (i, j, k). An equation connecting a number of grid points is said to have coordinate (i, j, k), when (i, j, k) is the smallest grid coordinate involved in the equation.

One way to order the rows of A, is to scan all the grid points in the order ijk, and then for each grid point insert those of the four Z-equations that are applicable at that instance of ijk. Such a row ordering will be denoted a $Zijk$-ordering, where Z runs through (0,1,2,3). In FORTRAN this ordering could be obtained by

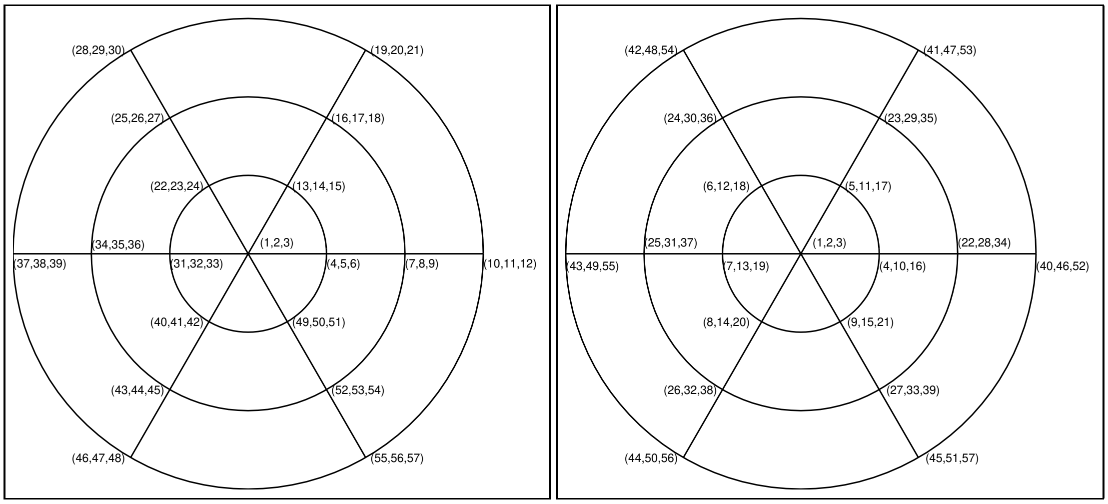

Figure 1: An example (one k-level) of a dijk- and a jdik-ordering of the unknowns. The triplet at each gridpoint consists of the indices of (U, V, W).

```
DO k=1,nz
  DO j=1,ntheta-2
    DO i=0,nr
      DO Z=0,3
         .
         .
         .
```

Another ordering is the $jZik$ ordering which is obtained from the FORTRAN example above by moving the j-loop inside the Z-loop. For each of the row orderings one can choose to scan the Z-equations in the order 0,1,2,3, or in any permutation of this order.

The columns can likewise be ordered be scanning the grid points. For each grid point there are three unknowns (U, V, W). As in the row ordering, one can obtain a column ordering by scanning the grid points in the order ijk and then for each grid point insert the triplet of unknowns. The unknowns can be scanned in the order (U, V, W) or in any permutation of this order. Since each component in (U, V, W) is the velocity in a certain direction, the column ordering described above will be referred to as a $dijk$-ordering. In FORTRAN this could be obtained by

```
DO k=2,nz-1
  DO j=1,ntheta-2
    DO i=0,nr-1
      DO d=1,3
         .
         .
         .
```

The column analogue of the $jZik$-ordering is the $jdik$-ordering, which is obtained from the FORTRAN example above by moving the j-loop inside the d-loop.

In these letter combinations the rightmost letter is referred to as the most significant index and the leftmost as the least significant index. Figure 1 shows a $dijk$- and a $jdik$-ordering of the unknowns at one horizontal level. It turns out that a combined $jZik/jdik$-ordering has some very convenient features.

To understand why, we make a few important observations regarding the structure of A:

1 $a_{r,c}$ is independent of k.

2 A row in A involves only unknowns at k and at $k+1$.

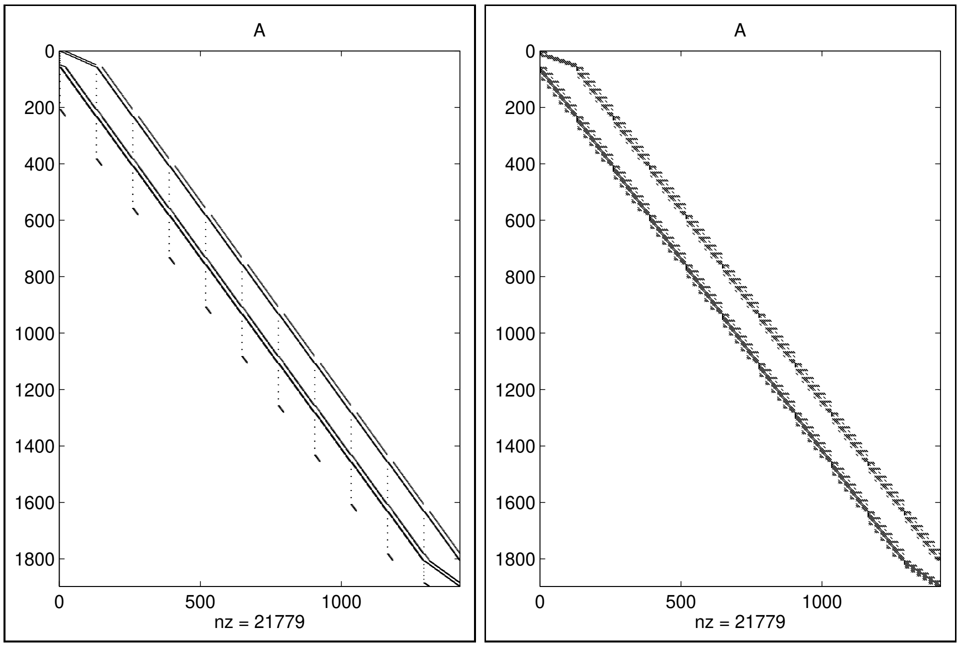

Figure 2: A $Zijk/dijk$- and a $jZik/jdik$-ordered A, $(N_r, N_y, N_z) = (7, 9, 13)$

3 A row in A involves only unknowns at i and at $i{+}1$, $i = 0, 1, \ldots, N_r{-}1$.

4 A row in A involves either unknowns at j and at $j{+}1$, or at $j = N_\theta{-}2$ and $j = 1$.

5 $a_{r,c}$ is independent of j iff $i > 0$ (the center axis of the cylinder, where $i = 0$, is a special case).

This can be seen by inspection of $Z0, Z1, Z2, Z3$.

2.2.1 The Macro Structure of A

A consequence of the first observation is that a row ordering where k is the most significant index will result in an A consisting of equally sized blocks of rows, and the blocks corresponding to each of $k = 2, 3, \ldots, N_z{-}2$ will contain the same coefficients. The first and last row blocks, corresponding to $k = 1$ and $k = N_z{-}1$, are special cases, since each contain only a subset of the rows in the other blocks. If furthermore a column ordering with k as the most significant index is chosen, then the same thing applies for the columns. They will consist of equally sized column blocks, and for each of $k = 3, 4, \ldots, N_z{-}2$ these column blocks will consist of the same coefficients. The first and last column blocks, corresponding to $k = 2$ and $k = N_z - 1$, are special cases. These column blocks have only a subset of the nonzeros in the other blocks. With such an ordering A can therefore be partitioned into $(N_z - 1) \times (N_z{-}2)$ block matrices, which we denote

$$A_{k_1,k_2},\ k_1 = 1, 2, \ldots, N_z - 1,\ k_2 = 2, 3, \ldots, N_z - 1$$

A consequence of observation 2 above is that only block matrices $A_{k,k}$, $k = 2, 3, \ldots, N_z{-}1$ and $A_{k,k+1}$, $k = 1, 2, \ldots, N_z{-}2$ are nonzero. This can be seen in figure 2, where an example of two orderings with k as the most significant index can be seen.

Observations 1 and 2 together imply that A consists of only 4 distinct submatrices, replicated along the 'diagonal' of A. The first, $A_{1,2}$, will be denoted A_1. The second $A_{k,k+1}$, $k = 2, 3, \ldots, N_z{-}2$, will be denoted A_2. The third, $A_{k,k}$, $k = 2, 3, \ldots, N_z{-}2$, will be denoted A_3. The last, A_{N_z-1,N_z-1}, will be denoted A_4. The number of columns in the four submatrices is the same, namely the number of unknowns at any k-level in the cylinder, $n_s = 3(1{+}(N_r{-}1)(N_\theta{-}2)$. The number of rows in A_2 and in A_3 is the number of equations, $m_s = (4N_r - 3)(N_\theta - 2)$, in a k-level, $k = 2, 3, \ldots, N_z - 2$. The number of rows in A_1 and A_4 is somewhat smaller.

It is straightforward to give the four submatrices a physical interpretation. A_1 is the coupling of the unknowns at $k = 2$ to the boundary at $k = 1$. A_2 is the coupling of the unknowns to each other within the same k-level. A_3 is the coupling between the unknowns in level k and level $k+1$. A_4 is the coupling of the unknowns at $k = N_z - 1$ to the boundary at $k = N_z$.

It is convenient to redefine A_1 and A_4 by adding zero-rows until they have m_s rows, so that $A_l \in \mathbb{R}^{m_s \times n_s}$, $l = 1, 2, 3, 4$. Let $E_l \in \mathbb{R}^{(N_z-1)\times(N_z-2)}$, $l = 1, 2, 3, 4$ be defined by

$$
\begin{aligned}
(E_1)_{i,j} &= \begin{cases} 1, & j = i = 1 \\ 0, & \text{otherwise} \end{cases} \\
(E_2)_{i,j} &= \begin{cases} 1, & j = i - 1,\ i = 2, \ldots, N_z - 2 \\ 0, & \text{otherwise} \end{cases} \\
(E_3)_{i,j} &= \begin{cases} 1, & j = i,\ i = 2, \ldots, N_z - 2 \\ 0, & \text{otherwise} \end{cases} \\
(E_4)_{i,j} &= \begin{cases} 1, & i = N_z - 1,\ j = N_z - 2 \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

Then

$$A = \sum_{l=1}^{4} E_l \otimes A_l \tag{10}$$

where $\otimes$ denotes the Kronecker product. It should be noted that the extra rows in A_1 and in A_4 do not change the solution to (1), provided that zeros are inserted into b accordingly.

That A can be written in this form has two advantages:

- The storage requirement is $O(N_r N_\theta)$ instead of $O(N_r N_\theta N_z)$.
- The computer implementation of A times a vector can be made efficient.

2.2.2 The bandwidth of A

We will later need to use A in a matrix-vector product implemented on a distributed parallel computer. When computing a matrix-vector product on a distributed parallel computer, a narrow bandwidth of A is desirable since this requires less communication. We have seen that k being the most significant index in both the row and column ordering leads to a relatively narrow (Kronecker) structure. This leaves room for several orderings with various bandwidths. Figures 3 and figure 4 show two such orderings, a $Zijk/dijk$- and a $jZik/jdik$-ordering. These orderings result in very different bandwidths of the submatrices, and therefore also differing bandwidths for A. This difference in bandwidth can be explained by observations 2, 3 and 4. In the j-direction, and in this direction only, unknowns across the entire interval may be involved in a single equation. This means that a narrow bandwidth of the submatrices can be obtained by having j as the least significant index in both the row- and column-ordering, as in the $jZik/jdik$-ordering.

2.2.3 The Micro Structure of A

The $jZik/jdik$-ordering not only gives A a Kronecker structure and a narrow bandwidth; it turns out that each of the submatrices A_l are highly structured as well. Recall that the ordering of A_l is jZi/jdi. This means that A_l can be partitioned into blocks, each block corresponding to the coefficients in some Z-equation with respect to one of the velocity components U, V or W at some value of i, k, for all of $j = 1, 2, \ldots, N_\theta - 2$. For $i > 0$ these blocks are square with size $(N_\theta - 2) \times (N_\theta - 2)$. The reason is that there are $N_\theta - 2$ instances of a Z-equation at i, k, and $N_\theta - 2$ unknowns in each of the 3 dimensions at i, k for $i > 0$. At $i = 0$ there is just one column in the block since there is just one unknown per dimension at $i = 0$. The interesting thing about these blocks is that the nonzeros in each block are

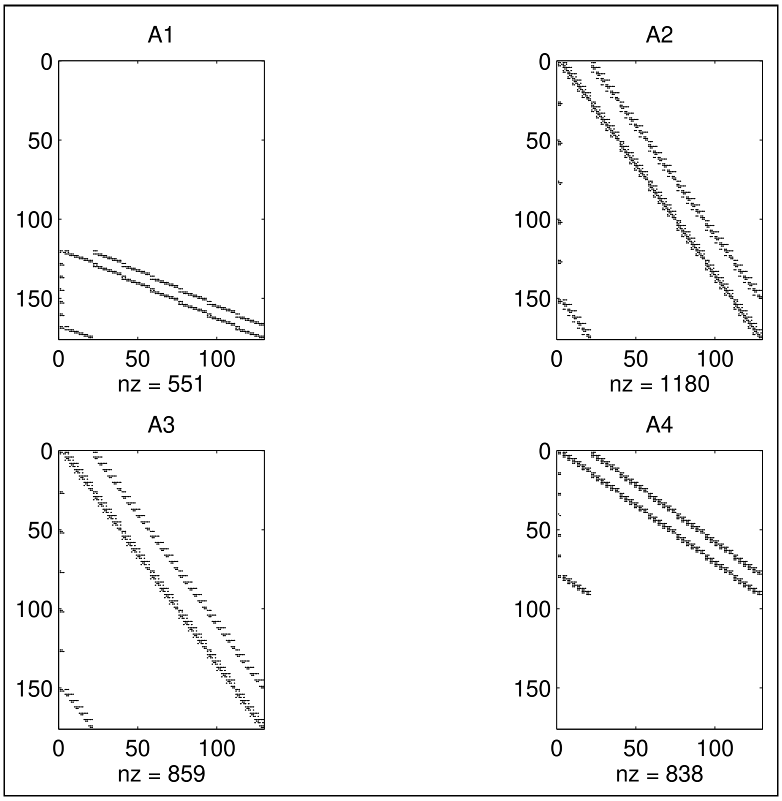

Figure 3: The submatrices of a $Zijk/dijk$-ordered A, $(N_r, N_y, N_z) = (7, 9, 13)$

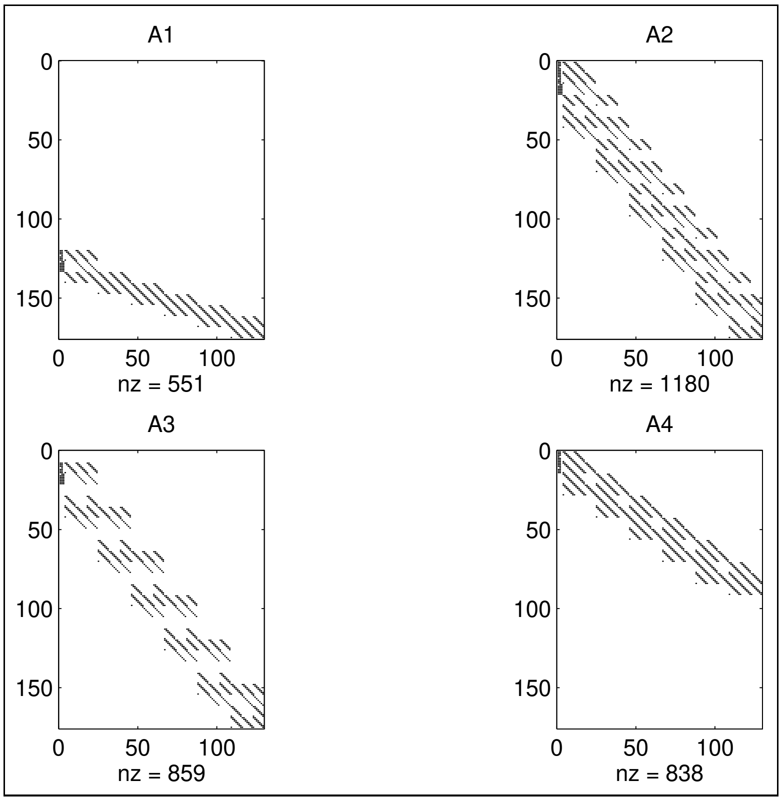

Figure 4: The submatrices of a $jZik/jdik$-ordered A, $(N_r, N_y, N_z) = (7, 9, 13)$

independent of j iff $i > 0$ (observation 5). This means that for these blocks the coefficients along any diagonal will be the same, i.e. that these matrices will have a Toeplitz structure. In fact, due to the periodic boundary on the angular dimension, the matrices will be circulant. For $i > 0$ the unknowns in (9) appear in two different ways (η denotes one of the velocity components U, V or W):

- As pairs indexed $\eta_{i,j,k}$ and $\eta_{i,j+1,k}$. The circulant matrix corresponding to this will have nonzeros only on the main diagonal, on the first super diagonal and the shifted element in the first column of the last row. The two nonzeros in each row will either be equal or their sum will be zero.

- $\eta_{i,j,k}$ is the only component of its type at i, k. The block corresponding to this will have nonzeros only on the main diagonal. The nonzeros will all have the same value.

Figure 4 gives an impression of the blocked circulant structure of the A_l matrices. The thin dense block in the three leftmost columns indicate the special case at $i = 0$.

There are roughly $4N_r$ different blocks at $i > 0$ and no more than $N_\theta - 2$ (distinct) values in each of the 7 blocks at $i = 0$, and the total number of distinct values in A is therefore roughly $4N_r + 7N_\theta$, much less than the number of nonzeros in A which is about $48 N_r N_\theta N_z$. For $(N_r, N_\theta, N_z) = (63, 66, 129)$, a resolution used in some of our tests, the storage requirement for A using compressed storage by columns is 190MB using single precision. By exploiting the described structure of A, the storage requirement is less than 5kb, a reduction of almost 40,000 times.

3 Parallel Iterative Solution of Equation (1)

We will solve (1) by use of the Preconditioned Conjugate Gradient method for Least Squares (PCGLS), a commonly used method for solving large sparse overdetermined linear systems of equations [2].

We will only consider the diagonal preconditioner,

$$S_{j,j} = ||a_j||_2,$$

where a_j is the j'th column of A.

With the notation used by Björck in [2] one step of PCGLS is (with approximate FLOP-counts):

a)	$p^{(k+1)} = s^{(k)} + \beta_k p^{(k)}$	$2n$ FLOPs
b)	$t^{(k)} = S^{-1} p^{(k+1)}$	n FLOPs
c)	$q^{(k)} = A t^{(k)}$	$32n$ FLOPs
d)	$\alpha_k = \frac{\gamma_k}{q^{(k)T} q^{(k)}}$	$\frac{8}{3}n$ FLOPs
e)	$r^{(k+1)} = r^{(k)} - \alpha_k q^{(k)}$	$\frac{8}{3}n$ FLOPs
f)	$x^{(k+1)} = x^{(k)} + \alpha_k t^{(k)}$	$2n$ FLOPs
g)	$\tilde{s}^{(k)} = A^T r^{(k+1)}$	$32n$ FLOPs
h)	$s^{(k+1)} = S^{-T} \tilde{s}^{(k)}$	n FLOPs
i)	$\gamma_{k+1} = s^{(k+1)T} s^{(k+1)}$	$2n$ FLOPs
j)	$\beta_{k+1} = \frac{\gamma_{k+1}}{\gamma_k}$	1 FLOPs

The iterations are initialized by setting $\beta_1 = 0$, $r^{(1)} = b - Ax^{(1)}$ and, with $k = 0$, executing steps g), h) and i).

The stopping criterion is

$$||s^{(k+1)}||_2 < \epsilon(||A^TA||_F||x^{(k+1)}||_2 + ||b||_2),$$

where ϵ is a user supplied stop-tolerance. $||A^TA||_F$ is set to 1, which is an acceptable approximation because of the diagonal preconditioning. To compute $||x^{(k+1)}||_2$ an extra $2n$ FLOPs are needed. The total FLOP-count for 1 PCGLS-iteration is therefore approximately $80n$ FLOPs.

3.1 Some Uniprocessor Considerations.

The matrix A enters PCGLS only through steps c) and g), i.e. as matrix-vector products with A and A^T. Since steps c) and g) are the most computationally demanding, effort has been made to compute these matrix-vector products efficiently. We have exploited the structure of A to achieve a fast computation of steps c) and g). In both steps we iterate through the nonzeros of A, and for each of these nonzeros we perform a multiply-add operation. A and $A^T$are iterated via 3 nested loops, one for each of the discrete spatial coordinates (i, j, k). The order of the loops reflect the row/column ordering of A, i.e. first k in the outermost loop, then i, and finally j in the innermost loop. For $i > 0$ the nonzeros of A are independent of j, which means that inside the innermost (j-) loop only the elements of the input and output vectors have to be accessed. This leads to an implementation of the matrix-vector products that is several times faster[1] than what can be achieved with an optimized implementation of the sparse matrix-vector product in [1].

After having implemented a matrix-vector product that exploits the structure of A, the amount of time to perform the remaining operations in PCGLS can no longer be neglected. The remaining operations are all BLAS-1. Such operations perform poorly on cache based architectures when their input arrays are not in the cache. For some of the operations this situation is inevitable. Consider for instance step f), the update of the solution x. In each iteration x is only accessed once, and with the exception of the rare case where all vectors and A can be stored in cache simultaneously x will not be in cache when it is to be updated. On four different cache based architectures the time to perform the BLAS-1 operations of PCGLS therefore amounts to 45%-68% of the total time spent in PCGLS depending on the actual architecture - a significant amount of time.

3.2 Parallelization Strategy.

To be able to run PCGLS on a variety of distributed parallel computers, the code is parallelized by use of explicit communication (by use of the communication standard MPI - version 1). The computations are divided across the vertical coordinate k such that each process handles variables and equations corresponding to a number of neighboring k-levels. The parallelization over k-levels makes all the vector-updates i.e. steps a), b), e), f), h) and j) perfectly parallel. The remaining operations require communication:

- The inner products in step d) and i).
- The matrix-vector products in step c) and g).

For a review of the concepts of parallelism in Conjugate Gradient methods see [5].

3.3 The Inner Products

We will analyze step i); the result applies to step d) as well. Each process owns (that is stores and updates) a part of s. Each process can compute a local inner product (LIP) from its part of s. This intermediate result then needs to be summed up globally by every

[1] On a IBM RS6000 model 590 the implementation is 7 times faster

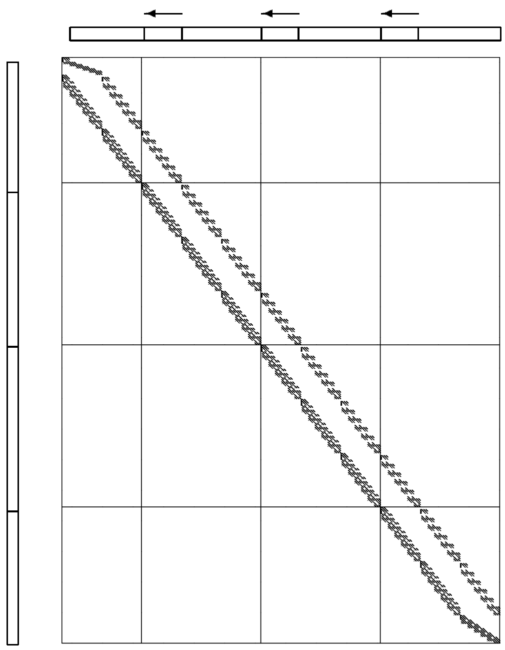

Figure 5: The distribution of work in step c).

process. This global reduction requires that the LIP of every process is distributed to all the other processes. This global communication can be implemented by a single MPI-call (MPI_ALLREDUCE). Since no process can obtain the global result before all processes have initiated the communication, the inner products are said to be synchronizing. We will see that this is the only necessary synchronizing communication, and the above formulation of PCGLS therefore has 2 synchronization points. Apart from the involved communication, synchronization points are inconvenient. Parallel performance is sensitive to fluctuations in the computational speed of individual processes, and the impact of improper load balance may be increased. Furthermore synchronization points decrease the possibility of the cache optimization known as blocking. Due to the stopping criterion $x^T x$ needs to be computed as well. The LIP of $x^T x$ is computed like $s^T s$, but a third synchronization point can be avoided by communicating the LIP's of $x^T x$ along with those of $s^T s$. This means that the stopping criterion can be evaluated after the completion of step i).

3.4 The Matrix-Vector Products

The required communication in a matrix-vector product is highly dependent on the bandwidth of the matrix in question. To parallelize our matrix-vector products we therefore need to examine the bandwidth of A and A^T.

3.4.1 The product with A

Figure 5 shows an example of the work in step c) distributed on four processes along with the parts of t that need to be communicated. In this context it is convenient to think of the processes as arranged in a pipe topology. With MPI it is possible to specify to the runtime system the chosen topology of the processes and thus aid the runtime system to obtain an optimal mapping of the processes onto the physical topology of the processors. Note that only the distinct values of A are stored. Since all distinct values of A are found in each k-level, every process stores exactly the same data concerning A. So the distribution of work in step c) merely defines the blocks of A with which the multiplication is local and those with which the multiplication requires communication. Since the unknowns connected by (9) are at the most one k-level apart, the communication in the matrix-vector products only has to take place between neighboring processes. This is illustrated in the example plot: the A_3-matrices corresponding to the coupling between two k-levels distributed on different processes require communication in the multiplication. The rest of the multiplication can be done without communication. As indicated in figure 5, the nonlocal part of the multiplication is done by letting every process send the relevant part of t to its left neighbor. If N_z and the number of processes N_p satisfy $N_z - 1 = N_p N_k$ then the multiplication with the submatrices of A will be distributed as follows:

First processor:	A_1, $(N_k - 1)$ A_2's, $(N_k - 1)$ A_3's
Last processor:	A_4, $(N_k - 1)$ A_2's, $(N_k - 1)$ A_3's
Remaining processes:	N_k A_2's, N_k A_3's

This distribution strategy has the advantage of being both simple ($N_p - 1$ messages are sent concurrently) and yielding a rather good load-balance: 2 processes have slightly less work than the remaining and the remaining all have the same amount of work.

Below we list some observations regarding the communication of t that can be exploited to reduce the communication overhead.

- The same segment of t needs to be sent in every iteration. The overhead of such repeated communication can be reduced by using persistent communication (an MPI term).
- The communication does not need to be synchronizing, and we therefore use asynchronous communication (called non-blocking in MPI).
- The overhead of the communication is minimal if the sender can assume that the receiver is ready when the sending starts (called ready-mode in MPI). The sending requires that the relevant part of t is computed in the previous step. The reception can however be initiated in the previous iteration. Since there is then a synchronization point between the initiation of the reception and the sending, ready-mode communication is guaranteed to work.
- After the sending of t is initiated the sender does not need to update t until after the synchronization point in step d). After step d) it is certain that the matching reception has completed, and the sender can therefore update t in the next iteration without having to check that the send has completed.

From the above we see that the synchronization effect of the inner products can be useful.

3.4.2 The product with A^T

Figure 6 shows the work (using the same A as in the previous example) in step g) distributed on four processes along with the parts of r that need to be communicated. The situation is very much the same as in step c): the same A_3-matrices require communication, the difference being that the A_3's are now transposed. If again $N_z - 1 = N_p N_k$ then the multiplication with the submatrices of A^T will be distributed as follows:

First processor:	A_1, $(N_k - 1)$ A_2's, $(N_k - 2)$ A_3's
Last processor:	A_4, N_k A_2's, $(N_k - 1)$ A_3's
Remaining processes:	N_k A_2's, N_k A_3's

The same 2 processes as before have slightly less work than the remaining and the remaining all have the same amount of work.

The overhead in communicating r can be reduced in the same four ways described above.

3.5 Comments on Load Balance

The described parallelization distributes levels 1 through $N_z - 1 = N_k N_p$ (there are no equations or unknowns at $k = N_z$) among the processes. This corresponds to distributing N_k k-levels of equations per processor. Since there are no unknowns at $k = 1$ the process

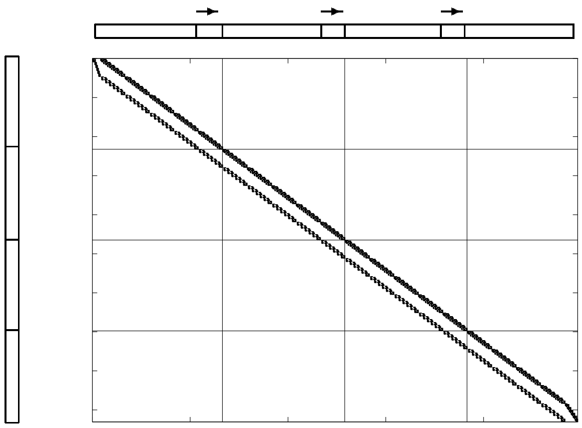

Figure 6: The distribution of work in step g).

that owns the equations at k-level 1 will have less BLAS-1 operations to perform than the remaining processes. Since the BLAS-1 operations have ended up being relatively time consuming, this means that the first process is expected to spend a significant amount of time waiting in the synchronization points, especially in the cases where there are only a few k-levels per processor.

3.6 Overlapping Communication with Computation

With MPI it is possible to reduce the communication overhead by specifying that the communication should be overlapped with computation, as far as this is possible on the computer being used. The idea is to initiate the communication as early as possible, then do as much useful computation as possible while hoping that the communication takes place concurrently. At some point the computation cannot continue without the completion of the communication, and the program has to wait for the communication to end. In the following we will analyze and exploit the potential for such overlapping.

Communication of t The reception initialization of t can be overlapped with the send of r in the previous iteration. To be able to start sending the segment of t, only the relevant parts of steps a) and b) need to be performed (and step j in the previous iteration must therefore be completed). The remaining parts of steps a) and b) can be performed concurrently with the communication. The local part of step c) (see figure 5) can also be performed concurrently with the communication. This produces part of q, and the LIP of this part of q can therefore also be computed concurrently with the communication. After this, the communication needs to finish before c) and d) can be completed.

Communication of r The reception initialization of r can be overlapped with the send of t. To be able to start sending r only the relevant part of step e) needs to be performed (and step d) must be completed). The remaining part of step e) and all of step f) can therefore be overlapped with the communication of r. The local part of step g) (see figure 6) can also be overlapped with communication. The corresponding part of both step h) and i) can therefore also be overlapped with communication. After this the communication needs to finish before steps g), h) and i) can be completed.

Step f), the update of x, is fully overlapped with the communication needed in step g). [5] describes how step f) instead can be overlapped with the communication in one of the inner products. In our case step f) could be overlapped with the communication in the inner product in step i). If step f) were to be overlapped with step i), then the LIP's of $x^T x$ would have to be communicated in step d) in the next iteration, thus adding an unnecessary half iteration. Note that of the time spent in i), only the actual communication time can be overlapped with step f), while the time spent waiting in a synchronization point due to load imbalance cannot be reduced by changing the order of computation. Let t_i denote the communication time in step i), t_r denote the time from the sending of r is initiated until the beginning of the the wait for the corresponding reception (this is the time interval in step g that can be overlapped with computation), let t_o denote the parts of steps e), g), h) and i) that can be overlapped with this communication, and let t_f denote the time spent in step f). Then it is easy to show that the approach in [5] will be more effective only if $t_i < t_r - t_o$ and $t_i < t_f$. The test results show that there really is nothing to be gained from overlapping step f) with the communication in step i). This is fortunate in so far as the MPI-1 standard does not define how global communication can be overlapped with computation.

3.7 Reducing the Communication Costs

The communication needed for the matrix-vector products is between neighboring pairs of processes, and the overhead of this communication is therefore likely to be independent of the number of processes. This is not the case with the communication needed for the inner products: an increasing number of processes will require an increasing amount of communication. This is one reason why it would be desirable to reduce the number of synchronization points from two to one. Along with the reasons mentioned in subsection 3.3, this motivates the finding of a formulation of PCGLS that has only one synchronizing inner product. In [3] an alternative formulation of the standard Conjugate Gradient algorithm with just one synchronization point is derived. Using the same approach we have derived and implemented an alternative formulation of PCGLS also with only one synchronization point. This formulation of PCGLS relies on the orthogonality of the residual vectors $s^{(k-1)}$ and $s^{(k)}$ of the normal system in two consecutive iterations. In spite of the fact that this is not guaranteed numerically, our alternative formulation of PCGLS does not show any signs of instability. But unfortunately the algorithm has not really proven itself effective. On a thin node equipped IBM SP-2 the two formulations are equally fast with 4 processors. With 32 processors the reduction in wall-clock time is 14%, but on all the other parallel computers in table 1 the alternative formulation is slower than the original in all of the test cases. We can think of three reasons for this:

- The reformulation introduces an extra BLAS-1 operation. Note that BLAS-1 operations have become relatively time consuming in our implementation.
- After the rearrangement of the computations, the communication needed for the multiplication with A could only be overlapped with little computation, so an increased amount of time would have to be spent waiting for the multiplication with A to finish. This has little or no effect on a SP-2, since the overlapping of communication and computation is not effective on this architecture [4].
- The effectiveness of the communication scheme for the matrix-vector product relies somewhat on the two synchronization points. Therefore the removal of a synchronization point leads to a less effective communication scheme for the matrix-vector product.

The fact that both uniprocessor performance and the communication overhead in the alternative formulation is reduced leads however to an increased parallel speed-up of this formulation. On the thin node SP-2 the increase in parallel speed-up is 25% on 32 processors.

4 Numerical Experiments

All calculations have been done in single precision[2], which is faster than double precision and for our purposes sufficient in terms of accuracy. We have conducted a number of experiments

[2] Single-precision on a Cray is equal to double precision on the other tested machines.

Machine	No. of processors	Relative solution time
SGI PC R8000	16	0.87
Cray T3D	64	1.00
IBM SP-2 Wide 66 Mhz	16	1.27
IBM SP-2 Thin 66 Mhz	32	1.46
SGI PC R4000	16	1.67
Cray C92A	1	3.44
SGI PC R10000	6	3.55
Meiko CS/2	14	3.70
Cray Y-MP	1	9.31

Table 1: Relative solution time for various machines using PCGLS with two synchronization points.

with spatial resolutions from $(N_r, N_\theta, N_z)=(7,7,7)$ up to $(N_r, N_\theta, N_z)=(127,130,145)$. In these experiments the boundary conditions have been changed to a flow that has an analytical solution. In all cases the solver returned the expected solution.

The following experiments have been conducted to investigate the performance of the solver. All timings are elapsed wall-clock times. This is likely to show a worse performance on machines that are not completely unloaded, and performance may therefore be better than the measurements indicate. The reason for using wall-clock timing is that any other timing may leave out time where the computation is stopped due to time-sharing on the computer, even though the communication is ongoing.
We now consider a test case with a spatial resolution of $(N_r, N_\theta, N_z)=(63,66,129)$. This results in a rather large system ($n \approx 1.5 \cdot 10^6$) but such a resolution is believed to be necessary to obtain a good model of the flow. With $x^{(0)} = 0$ and $\epsilon = 3 \cdot 10^{-5}$ the solver requires 5400 iterations and yields a solution where the largest relative error in a component is 10^{-4}. On a 64 processor Cray T3D this takes 250 seconds. In the performance tests the number of iterations has been fixed at 100. The reason is that on a timesharing machine, longer runs are very likely to be interrupted by processes of other users, making the timings less accurate.

The computers used in the tests are listed along with their number of processors in table 1. This table also shows the time spent on the 100 iterations relative to the time spent on the 64 processor Cray T3D. The table includes two uniprocessor vector machines; the gain from the parallel implementation of PCGLS can be seen from this. The relative solution time using PCGLS with one synchronization point is 1.28 on the thin node IBM SP-2.

Figure 7 shows the parallel speed-up for the seven parallel machines. Since the maximum number of processors per machine varies from 6 to 64, the results are shown in two plots: one with measurements with at most 16 processors, and another with measurements with at least 16 processors. Some comments on the speed-ups are appropriate:

- The R4000 and R8000 equipped SGI Power Challenges exhibit super-linear speed-up even with this rather large system. This happens because each CPU has a much larger cache than the other machines and because of the low storage requirement for A.
- The speed-up of the R10000 equipped SGI Power Challenge at 6 processors and that of the Meiko at 12 and 14 processors[3] is low due to bad load-balance. In the described test case load-balance is only optimal with 2^p processors.
- The speed-up of the IBM SP-2 (both variants) is quite low. This may be because the SP-2 nodes compute quite fast. The timings for 1 processor vary with less than 1%, while the timings with 32 processors vary with more than 20%. We therefore believe the relatively poor speed-up also is caused by the fact that the inter-processor communication in our SP-2 experiments is competing for the bandwidth of the inter-processor connection against the communication between processors used by others.

[3]Tests are conducted with one CPU per node.

- The speed-up on the T3D decreases significantly when the number of processors increases from 32 to 64. With 64 processors there are only two levels per processor, and therefore not much computation to overlap with the communication. This reveals a weakness within the chosen parallelization strategy: with a massively parallel computer a large number of levels is necessary to obtain a good speed-up.

We have measured how the time is spent on communication in these tests. Figures 8, 9, 10 and 11 show the amount of the total solution time spent in each of the communication steps. In each figure the first plot shows the start-up time for the processor-to-processor communication in the two matrix-vector products while the second shows the time to complete that communication. The remaining two show the time spent in the inner product communication. The scaling on the y-axis is the same in all plots and is determined by the largest amount of time that any one of the processors spends on communication. These measurements have been excluded for the R4000 and R10000 variants of the SGI Power Challenge, since they are similar to those of the R8000 variant. The same holds for the wide node variant of the IBM SP/2.

The plots confirm that there is too little work for the first and last processors. The first processor on both the thin node SP-2 and the CRAY T3D spends more than 50% of the total time on communication. The effect of this substantial inefficiency is however diminished by the large number of processors that spend a much smaller amount of time on communication. Secondly, they show that nothing can be gained by overlapping the inner product communication with the update of x: the reduction in the inner product wait would simply cause an equal increase in the matrix-vector product wait. The time spent on completion of the communication in step g) is generally smaller than that in step c). This is because the communication in step g) is overlapped with a larger amount of computation, including the update of x. The plots furthermore show widely varying characteristics of the parallel overhead on the different architectures. The amount of time spent with the global sums in steps d) and i) is much larger for the thin node SP-2 than for the other machines. This explains why PCGLS with one synchronization point is faster on this machine. Finally, the measurements on the SGI R8000 were conducted on a (lightly) loaded machine, which may explain the fluctuations in the timings.

It is common to measure the parallel scalability of an algorithm by measuring the scaled speed-up. A scaled speed-up is obtained by increasing the number of processors while keeping the amount of work per processor constant. With a perfectly parallel algorithm such an experiment should produce timings that are independent of the number of processors. An easy way to construct such an experiment would be to assign N_k k-levels to each processor, and then simply increase the vertical resolution along with the number of processors. Due to the special cases in our Cauchy-Riemann equations at the first and last k-level, this approach will result in an uneven amount of work per processor. Since this makes it so difficult to analyze the scaled speed-ups we have decided not no use them.

The scalability of the solver is therefore evaluated on the basis of the standard speed-ups. We conclude that on a massively parallel computer like the CRAY T3D it is feasible to use up to 64 processors and no more. On the other machines it is feasible to use as many processors as we have tested and possibly up to 64, provided that the machines are unloaded or lightly loaded. With these restraints it is safe to say that the solver has a good parallel scalability.

5 Conclusion

By appropriate reordering we have discovered two levels of structure in the matrix A. This has been exploited to achieve a very substantial reduction in the storage requirement for A.

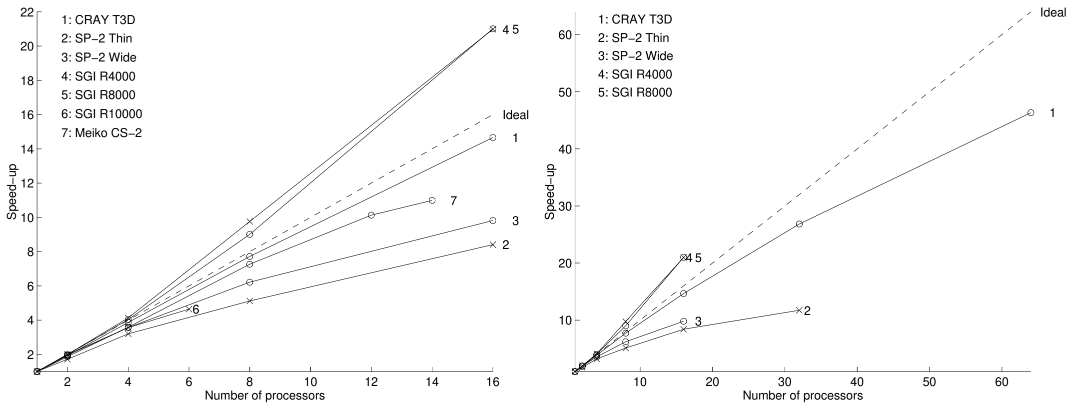

Figure 7: Parallel speed-up on up to 16 and up to 64 processors.

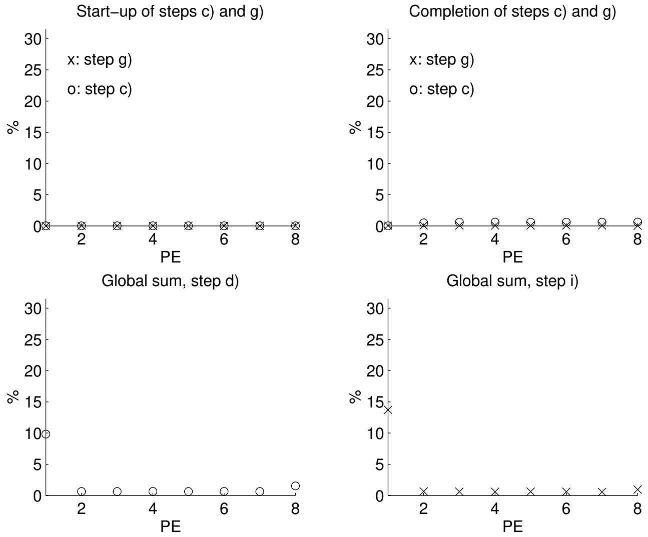

Figure 8: Parallel overhead on 8 Meiko CS/2 processors.

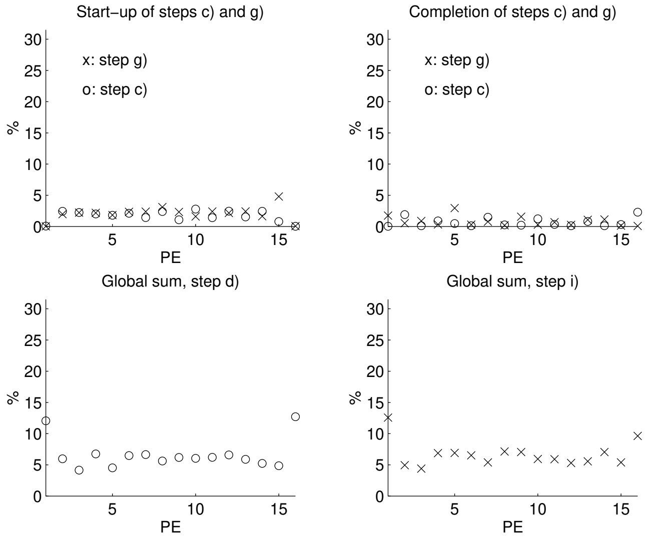

Figure 9: Parallel overhead on 16 SGI R8000 Power Challenge processors.

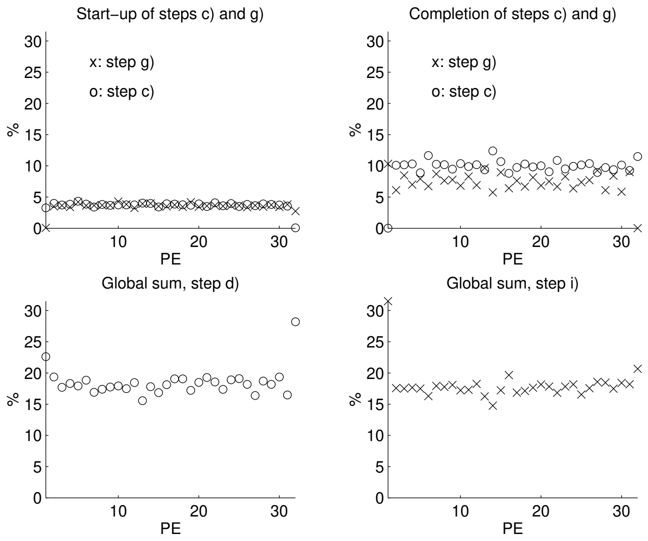

Figure 10: Parallel overhead on 32 IBM Thin SP/2 processors.

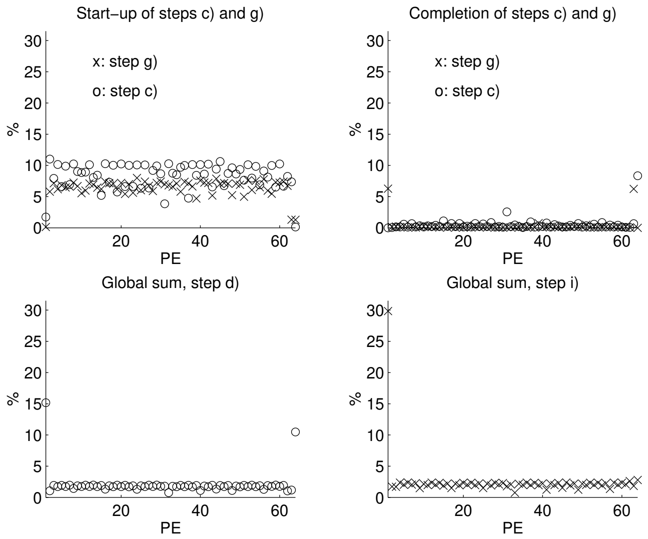

Figure 11: Parallel overhead on 64 Cray T3D processors.

In the example used in our numerical experiments the reduction is almost 40,000 times when compared to standard sparse matrix techniques. The structure has also been exploited to obtain a matrix-vector multiply that is several times faster than what can be obtained with standard sparse techniques.

The Preconditioned Conjugate Gradient algorithm for Least Squares (PCGLS) has been implemented on a variety of parallel computers. Effort has been made to rearrange the computations so as to maximize the overlapping of communication with computation. The success of this depends on the chosen computer architecture.

In an attempt to reduce the communication overhead, an alternative formulation of PCGLS with just one synchronizing inner product has been derived and implemented. This alternative implementation did not show instability, but the obtained reduction in communication overhead introduced another communication overhead along with a computational overhead. For this reason this reformulation produces faster results only on one out of seven tested machines.

The parallel scalability of our PCGLS implementation is good on a variety of computers with up to 64 processors.

6 Acknowledgements

The described research was carried out during a stay at CERFACS from September 1996 to January 1997.

Michele Benzi, CERFACS, has been a great source of inspiration, invaluable in questions concerning linear algebra, and in general as good an advisor as one could wish for.

References

[1] R. Barrett et al., "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", SIAM, pp. 62-63, 1994

[2] Å. Björck, "Numerical Methods for Least Squares Problems", SIAM, pp. 288-294, 1996

[3] E. D'Azevedo, V. Eijkhout and C. Romine, "Reducing Communication Costs in the Conjugate Gradient Algorithm on Distributed Memory Multiprocessors", in LAPACK Working Note 56, Tech. report, Computer Science Department, University of Tennessee, Knoxville, TN, 1993.

[4] Cornell Theory Center, "MPI: Message Passing Interface Library", `http://www.tc.cornell.edu/UserDoc/Software/PTools/mpi/`

[5] J. W. Demmel, M. T. Heath and H. A. van der Vorst, "Parallel numerical linear algebra", in A. Iserles (Ed.), Acta Numerica 1993, pp. 171-179, 1993

[6] J. N. Sørensen, M. O. L. Hansen and E. Jensen, "Simulation of Fluid Dynamical Flow Problems", PARA '94 J. Wasniewski (Ed.), Lecture Notes in Computer Science vol. 879, pp. 458-468, 1994.