

PARALLEL BLOCK ITERATIVE SOLVERS FOR HETEROGENEOUS COMPUTING ENVIRONMENTS

M. ARIOLI

*Istituto di Analisi Numerica, CNR
via Abbiategrasso 209
27100 Pavia ITALY
arioli@bond.ian.pv.cnr.it*

A. DRUMMOND

*Centre Européen de Recherche
et de Formation Avancée en Calcul
Scientifique (CERFACS)
42 Ave. G. Coriolis
31057 Toulouse, FRANCE
drummond@cerfacs.fr*

I.S. DUFF

*CERFACS, and also
Rutherford Appleton Laboratory
Chilton, Didcot, Oxon
OX11 0QX ENGLAND
duff@cerfacs.fr*

D. RUIZ

*Ecole Nationale Supérieure
d'Electrotechnique, d'Electronique,
d'Informatique, et d'Hydraulique
de Toulouse (ENSEEIH)
31000 Toulouse, FRANCE
ruiz@enseeiht.fr*

ABSTRACT. We study the parallel implementations of a block iterative method in heterogeneous computing environments for solving linear systems of equations. The method is a generalization of the row-projection Cimmino method where blocks are obtained by partitioning the original linear system of equations. The method is referred to as the Block Cimmino method. In addition, we accelerate the Block Cimmino convergence rate by the Block Conjugate Gradient method (Block-CG). Firstly, we present three different implementations of the Block-CG to compare different parallelization strategies of the method. Secondly, we present a scheduler to balance the computational load of the parallel distributed implementation of the Block Cimmino iteration. The computational resources used in these experiments are a BBN TC2000, and a network of Sun Sparc 10 and IBM RS-6000 workstations. We use PVM 3.3 to handle the interprocessor heterogeneous communication.

KEYWORDS. Block iterative methods, Cimmino method, conjugate gradient method, heterogeneous environments, parallel algorithms, PVM.

1 INTRODUCTION

The Block Conjugate Gradient method (Block-CG) ([13, 3]) is an extension of the classical conjugate gradient method. The Block-CG algorithm simultaneously searches the solution to a linear system of equations in different Krylov subspaces. Moreover, the Block-CG method can also be used to simultaneously find more than one solution to linear systems of equations with multiple right-hand sides. Given the nature of the Block-CG method, we propose three different strategies for its parallel distributed implementation. The main differences between the three Block-CG implementations are the amount of computation performed in parallel, the communication scheme, and distribution of tasks among processors.

The Cimmino method ([7]) is a row projection method in which the solution of the linear systems of equations is obtained through row-projections of the original matrix in given subspaces. In the Block Cimmino method ([2, 5]), the blocks are obtained by dividing the linear system of equations into subsystems. At every iteration, it computes one projection per subsystem and uses these projections to construct an approximation to the solution of the linear system. The Block-CG method can also be used to accelerate the Block Cimmino convergence rate ([11, 3]). Therefore, we present an implementation of a parallel distributed Block Cimmino method where the Cimmino iteration matrix is used as a preconditioner for the Block-CG algorithm.

In the parallel distributed Block Cimmino implementation, we have developed a scheduler that assigns units of work of different sizes to a set of heterogeneous processors. With the scheduler, we generate heuristics to balance the work load among processors taking into consideration the size of the units of work and the potential communication between these units. Processors can be clustered to take advantage of particular network interconnections. Thus, for the scheduler, we classify the processors into single processor nodes, shared memory clusters, and distributed memory clusters.

In Section 2, we present the three parallel Block-CG implementations with the results obtained from these implementations. Later, in Section 3 we present a scheduler for the parallel distributed implementation of the Block Cimmino method. Lastly, we present conclusions and general observations in Section 4.

2 PARALLEL DISTRIBUTED BLOCK-CG

The Block Conjugate Gradient algorithm (Block-CG) is an extension of the classical conjugate gradient algorithm from [12], and in turn it belongs to a broad class of conjugate direction methods for solving linear systems of equations of the form:

$$\mathbf{H}x = k, \tag{2.1}$$

where \mathbf{H} is an $n \times n$ symmetric positive definite matrix (SPD). These methods guarantee, in absence of roundoff errors, to reach a solution to (2.1) in a finite number of steps.

In the Block-CG method one considers more than one right hand side vector, leading to the linear systems :

$$\mathbf{H}X = K, \tag{2.2}$$

in which \mathbf{X} and \mathbf{K} are now matrices of order $n \times s$, and s is the block size. For this reason the method is called Block-CG.

Algorithm 2.1 is a stabilized Block-CG [3, 14], and from this algorithm we have developed three different parallel distributed implementations.

Algorithm 2.1 (Stabilized Block Conjugate Gradient)

Begin

$\mathbf{X}^{(0)}$ is arbitrary, $\mathbf{R}^{(0)} = \mathbf{K} - \mathbf{H}\mathbf{X}^{(0)}$

$\overline{\mathbf{R}}^{(0)} = \mathbf{R}^{(0)}\gamma_0^{-1}$ such that $(\overline{\mathbf{R}}^{(0)T} \overline{\mathbf{R}}^{(0)} = \mathbf{I})$

$\overline{\mathbf{P}}^{(0)} = \overline{\mathbf{R}}^{(0)}\beta_0^{-1}$ such that $(\overline{\mathbf{P}}^{(0)T} \mathbf{H}\overline{\mathbf{P}}^{(0)} = \mathbf{I})$

for $j = 0, 1, \dots$, until convergence

do

$\lambda_j = \beta_j^{-T}$

$\mathbf{X}^{(j+1)} = \mathbf{X}^{(j)} + \overline{\mathbf{P}}^{(j)}\lambda_j (\prod_{i=j}^0 \gamma_i)$

$\overline{\mathbf{R}}^{(j+1)} = (\overline{\mathbf{R}}^{(j)} - \mathbf{H}\overline{\mathbf{P}}^{(j)}\lambda_j)\gamma_{j+1}^{-1}$ such that $(\overline{\mathbf{R}}^{(j+1)T} \overline{\mathbf{R}}^{(j+1)} = \mathbf{I})$

$\alpha_j = \beta_j \gamma_{j+1}^T$

$\overline{\mathbf{P}}^{(j+1)} = (\overline{\mathbf{R}}^{(j+1)} + \overline{\mathbf{P}}^{(j)}\alpha_j)\beta_{j+1}^{-1}$ such that $(\overline{\mathbf{P}}^{(j+1)T} \mathbf{H}\overline{\mathbf{P}}^{(j+1)} = \mathbf{I})$

enddo

End

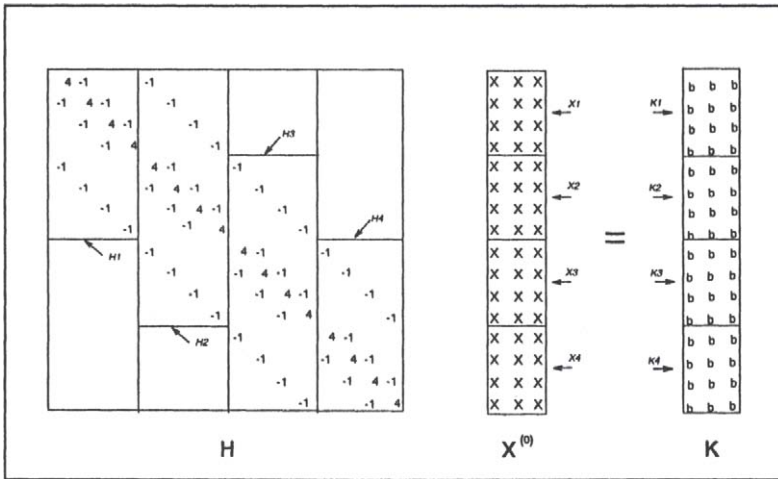


Figure 1: Example of a linear system of equations with a block size of 3 partitioned into 4 subsystems (e.g., $l = 4$).

For the parallel distributed implementations of Block-CG, we first partition the matrix \mathbf{H} from (2.2) into l submatrices $\mathbf{H}_1, \mathbf{H}_2, \dots, \mathbf{H}_l$, in a way that every submatrix \mathbf{H}_i has dimensions $n_i \times k_i$. The k_i dimension of every submatrix \mathbf{H}_i comes from a column partition

of the original matrix H , and the n_i dimensions are determined by the number of rows with non-zero elements inside a column partition. Similarly, we partition the $X^{(0)}$, and K matrices into l submatrices with dimensions $k_i \times s$ each. Afterwards, we distribute the l sets of $\langle H_i, X_i^{(0)}, K_i \rangle$ among p processors. An example of a matrix partitioning strategy is shown in Figure 1.

In Algorithm 2.1 the parallel computations of the matrices: $HX^{(0)}$, $HP^{(j)}$, β_j , and γ_j required to build in full the $P^{(j)}$, and $R^{(j)}$ matrices. The places in the algorithm where the matrices $HX^{(0)}$, $HP^{(j)}$, β_j , and γ_j are computed require interprocess communication and synchronization, and these places can penalize the efficiency of the parallel implementation of the Block-CG method.

In our first implementation, we minimize the number of required communications us-

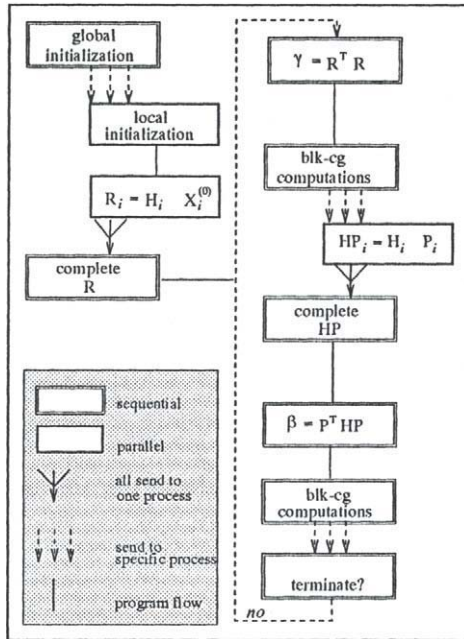


Figure 2: Master-Slave : centralized Block-CG implementation.

ing a *master-slave* computational approach in which the master performs the Block-CG algorithm 2.1 with the help of p slaves to perform the $HP^{(j)}$ products. In Block-CG algorithm, the most expensive part in term of computations is the calculation of the $HP^{(j)}$ products and in this implementation we only parallelized these products. We refer to this implementation as “Master-Slave: centralized Block-CG.” Figure 2 illustrates the flow of computations for the Master-Slave: centralized Block-CG implementation.

As a second implementation, we consider a *master-slave* computing approach in which each of the p slaves perform iterations of the Block-CG Algorithm 2.1 in a set of matrices $\langle H_i, X_i^{(0)}, K_i \rangle$. The role of the master in this case is to gather partial results from the

$R_i^{(j)T} R_i^{(j)}$ and $P_i^{(j)T} HP_i^{(j)}$ products in order to build the γ_j and β_j matrices respectively. At the same time, each slave has information about other slaves with whom it needs to exchange information to build locally a part of the full $HP_i^{(j)}$ matrix. The implementation is illustrated in Figure 3. We will refer to this implementation as “Master-Slave: distributed Block-CG.”

Lastly, we develop an implementation based on an *all – to – all* computing model. This

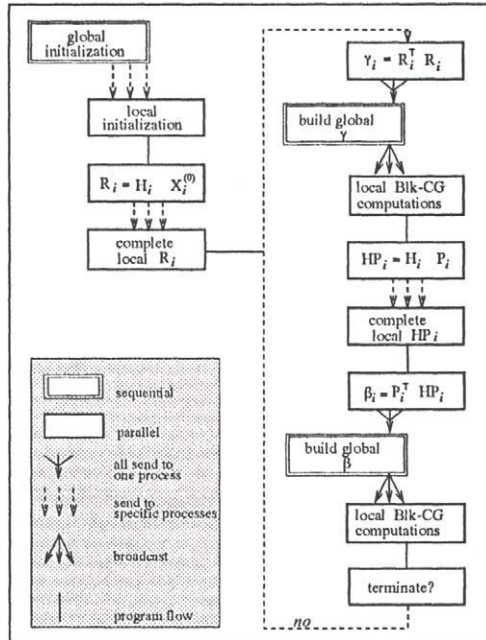


Figure 3: Master-Slave: Distributed Block-CG implementation.

time the motivation is to reduce the communication bottlenecks created by having a processor that acts as master and needs to receive messages from p slaves and broadcast the results back. In this implementation, we have an *all – to – all* communication for computing the γ_j and β_j matrices which means that, after the communication, the same γ_j and β_j information is local to every processor. To compute the full $HP_i^{(j)}$ matrix each processor communicates with only the processors that have information relevant to its computations. Figure 4 is an illustration of this implementation of the Block-CG algorithm. We refer to this implementation as the “All-to-All Block-CG”.

In all three implementations the interprocessor communication has an impact on performance. Therefore, we analyse the amount of information that needs to be communicated in each implementation.

Let m_k be the number of processors with whom the k -th processor must communicate in order to compute the full $HP_i^{(j)}$ or $HX_i^{(0)}$ matrix. Notice that each processor only needs to communicate a part of its local information with its m_k neighbour processors. In the

Master-Slave: centralized Block-CG implementation, these products are handled differently and each processor sends results back to the processors executing the master role. Table 1 summarizes the number of messages sent per iteration of the Block-CG.

We observe in Table 1 that the Master-Slave: centralized Block-CG implementation needs

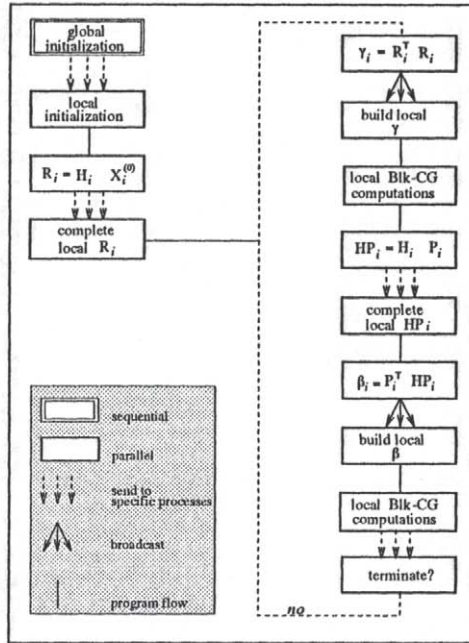


Figure 4: All-to-All Block-CG implementation.

Name of Implementation	Number of messages for product			Total Number of messages
	$HP^{(j)}$	$R^{(j)T}R^{(j)}$	$P^{(j)T}HP^{(j)}$	
Master-Slave: centralized Block-CG	$2p$	0	0	$2p$
Master-Slave: distributed Block-CG	$\sum_{k=1}^p m_k$	$2p$	$2p$	$\sum_{k=1}^p m_k + 4p$
All-to-All Block-CG	$\sum_{k=1}^p m_k$	$p \times (p - 1)$	$p \times (p - 1)$	$\sum_{k=1}^p m_k + 2p^2 - 2p$

Table 1: Number of messages sent at every iteration

the least amount of messages per iteration. However, in the Master-Slave: centralized Block-CG the length of every message is $n_i \times s$ (master processor sends $P^{(j)}$ and each slave processor sends back $HP^{(j)}$). As stated before, in the All-to-All Block-CG and Master-Slave distributed Block-CG implementations, the processors communicate directly with their neighbours to compute the $HP^{(j)}$ products. These messages are in almost all

cases smaller than $n_i \times s$ except when the matrix H is a full matrix. The length of the messages used to exchange the inner product is $s \times s$.

In the Master-Slave: centralized Block-CG, the master processor assembles the full $HP^{(j)}$ from partial results sent by slave processors. In the other two implementations, the assembly of the full matrices happens in parallel because each slave processor builds the part of the full matrix it needs. Furthermore, the overhead of assembling the $HP^{(j)}$ matrix in a centralized way increases as the number of subproblems and degree of parallelism increase.

The results shown in Tables 2 and 3 were run on a BBN TC2000 computer. We ran the

Laplace Matrix 4096 x 4096 (Block size = 4, 171 iterations) Elapsed Time of sequential version = 279142						
Number of PE's	All-to-All		Mstr-Slv: distributed		Mstr-Slv: centralized	
	Elps. Time	Speed-up	Elps. Time	Speed-up	Elps. Time	Speed-up
1	278827	1.001	279436	0.999	-.*	-.*
2	143083	1.951	143419	1.946	301884	0.925
4	71393	3.910	71244	3.918	278184	1.003
8	40755	6.849	38798	7.195	273320	1.021
12	40668	6.864	29747	9.384	279414	0.999
16	57759	4.833	25452	10.967	283649	0.984

Table 2: Test matrix generated from a discretization on a 64×64 grid: Laplace's equation. Times shown in table are in microseconds.

LANPRO Matrix 960 x 960 (Block size = 4, 138 iterations) Elp. Time of sequential version = 64869						
Number of PE's	All-to-All		Mstr-Slv: distributed		Mstr-Slv: centralized	
	Elps. Time	Speed-up	Elps. Time	Speed-up	Elps. Time	Speed-up
1	64980	0.998	65455	0.991	-.*	-.*
2	34063	1.904	34347	1.889	61942	1.047
4	19531	3.321	19451	3.335	53964	1.202
8	14108	4.598	12667	5.121	52175	1.243
12	20943	3.097	11319	5.730	53566	1.211
16	48054	1.350	11874	5.463	58400	1.110

Table 3: This matrix comes from The Harwell-Boeing Sparse Matrix Collection, and it is obtained from a biharmonic operator on a rectangular plate with one side fixed and the others free. Times shown in table are in microseconds.

experiments with 1, 2, 4, 8, and 16 processors. We used two SPD matrices for running the experiments. The first matrix is the result of a discretization on a 64×64 grid: Laplace's equation. The matrix is sparse of order 4096, and has 20224 nonzero entries. The second matrix, LANPRO of order 960 with 8402 nonzero entries, comes from the Harwell-Boeing Sparse Matrix Collection [9].

The sequential time reported in Tables 2 and 3 is the result of running a sequential implementation of Block-CG without any routines for handling parallelism. The sequential implementation uses the same BLAS and LAPACK [1] routines as the parallel Block-CG

implementations.

We can see in Tables 2 and 3 that the larger the problem size is, the better the speed-ups we get with the Master-Slave: distributed Block-CG implementation. This is not the case for the Master-slave: centralized Block-CG implementation, for which the performance decreases as we increase the size of the problem, and the overhead from monitoring the parallelism by the master processor negates all the benefits from performing the $HP^{(j)}$ products in parallel.

In the All-to-All Block-CG implementation, we have chosen to perform redundant computations in parallel instead of waiting for a master processor that gathers, computes and broadcasts the results from computations. As can be seen in Tables 2 and 3, an increase in the degree of parallelism penalizes the performance of the implementation due to the accompanying increase in interprocessor communication.

We conclude from these experiments that the Master-Slave: distributed Block-CG implementation performs better than the other two implementations because the amount of work performed in parallel justifies better the expense of communication. Furthermore, we use this implementation to accelerate the rate of convergence of the Block Cimmino iterative solver to be presented in the next section.

3 PARALLEL DISTRIBUTED BLOCK CIMMINO

The Block Cimmino method is a generalization of the Cimmino method [7]. Basically, we partition the linear system of equations:

$$\mathbf{A}x = b, \quad (3.1)$$

where \mathbf{A} is a $m \times n$ matrix, into l subsystems, with $l \leq m$, such that:

$$\begin{pmatrix} \mathbf{A}^1 \\ \mathbf{A}^2 \\ \vdots \\ \mathbf{A}^l \end{pmatrix} x = \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^l \end{pmatrix} \quad (3.2)$$

The block method ([5, 2]) computes a set of l row projections, and a combination of these projections is used to build the next approximation to the solution of the linear system. Now, we formulate the Block Cimmino iteration as:

$$\begin{aligned} \delta^{i(k)} &= \mathbf{A}^{i+} b^i - \mathbf{P}_{\mathcal{R}(\mathbf{A}^{iT})} x^{(k)} \\ &= \mathbf{A}^{i+} (b^i - \mathbf{A}^i x^{(k)}) \end{aligned} \quad (3.3)$$

$$x^{(k+1)} = x^{(k)} + \nu \sum_{i=1}^l \delta^{i(k)} \quad (3.4)$$

In Equation (3.3), the matrix \mathbf{A}^{i+} refers to the Moore-Penrose pseudoinverse of \mathbf{A}^i defined as: $\mathbf{A}^{i+} = \mathbf{A}^{iT} (\mathbf{A}^i \mathbf{A}^{iT})^{-1}$. However, the Block Cimmino method will converge for any other pseudoinverse of \mathbf{A}^i and in our parallel implementation we use a generalized pseudo-

inverse [6], $\mathbf{A}_{\mathbf{G}^{-1}}^{i-} = \mathbf{G}^{-1} \mathbf{A}^{iT} (\mathbf{A}^i \mathbf{G}^{-1} \mathbf{A}^{iT})^{-1}$, where \mathbf{G} is an ellipsoidal norm matrix. The $\mathbf{P}_{\mathcal{R}(\mathbf{A}^{iT})}$ is an orthogonal projector onto the range of \mathbf{A}^{iT} .

We use the augmented systems approach, [4] and [10], for solving the subsystems (3.3)

$$\begin{bmatrix} \mathbf{G} & \mathbf{A}^{iT} \\ \mathbf{A}^i & 0 \end{bmatrix} \begin{bmatrix} u^i \\ v^i \end{bmatrix} = \begin{bmatrix} 0 \\ b^i - \mathbf{A}^i x \end{bmatrix}$$

with solution:

$$v^i = - (\mathbf{A}^i \mathbf{G}^{-1} \mathbf{A}^{iT})^{-1} r^i, \text{ and } u^i = \mathbf{A}_{\mathbf{G}^{-1}}^{i-} (b^i - \mathbf{A}^i x) = \delta^i \tag{3.5}$$

The Block Cimmino method is a linear stationary iterative method, with a symmetrizable iteration matrix [11]. The use of ellipsoidal norms ensures the positive definiteness of the Block Cimmino iteration.

An SPD Block Cimmino iteration matrix can be used as a preconditioning matrix for the Block-CG method. The use of Block-CG in this case accelerates the convergence rate of the Block Cimmino method. We recall that Block-CG will simultaneously search the next approximation to the system's solution in s -Krylov subspaces and, in the absence of roundoff errors, will converge to the system's solution in a finite number of steps. We use the Master-Slave Distributed Block-CG implementation presented in the previous section to develop a parallel block iterative solver based on the Cimmino iteration. At first, we solve the system (3.5) using the sparse symmetric linear solver MA27 from the Harwell Subroutine Library [8]. The MA27 solver is a frontal method which computes the \mathbf{LDL}^T decomposition. The MA27 solver has three main phases: Analyse, Factorize, and Solve. These MA27 phases are called from the parallel Block Cimmino solver.

First of all, the parallel Block Cimmino solver builds the partition of the linear system of equations (3.1) into (3.2) and generates the augmented subsystems. The solver then examines the augmented subsystems to count the number of nonzero elements inside each of them and identifies the column overlaps between the different subsystems. The number of nonzero elements per subsystem gives a rough estimation of the amount of work that will be performed on the subsystem. The column overlaps determine the amount of communication between the subsystems. In addition, the solver gathers information from the computing environment either supplied by the user or acquired from the message passing programming tool.

Processors are classified into single processor, shared memory clusters, and distributed memory clusters. We assume that the purpose of clustering a group of processors is to take advantage of a specific communication network between the processors. The information from the processors is sorted in a tree structure where the root node represents the startup processor, intermediate level nodes represent shared or distributed clusters, and the leaf nodes represent the processors. Processors in the tree are sorted from left to right by their computer power.

The tree of processors, the augmented subsystems, the number of nonzero elements per subsystem and the information from the column overlaps are passed to a static scheduler. The scheduler first sorts all the subsystems by their number of nonzero elements. Later, subsystems are assigned to processors following a postorder traversal visit of the tree of processors (e.g., first visit leaf nodes then the parent node in the tree). A cluster node receives

a number of subsystems to solve equal to the number of processors it has. In this case, a first subsystem is assigned to the cluster and the remaining ones are chosen from a pool of not-yet-assigned subsystems. To choose amongst the candidate subsystems, we consider the amount of column overlaps between them and the subsystems already assigned to the cluster and, then, we select the candidate subsystem with the highest factor of overlapping. This choice aims to concentrate the communications between subsystems inside a cluster. Every time a subsystem is assigned to a processor or cluster, we update a workload factor per processor. This workload factor is useful in the event that there are more subsystems than processors. The subsystems that remain in the not-yet-assigned pool after the first round of work distribution are assigned to the least loaded processor or cluster one at the time. Every time the least loaded processor is determined from the workload factors.

After assigning all the subsystems to processors, these subsystems are sent through messages to the heterogeneous network of processors. Each processor calls the MA27 Analyse and Factorize routines on the set of subsystems it has been assigned. Afterwards, it performs the Block Cimmino iteration on these subsystems checking the convergence conditions at the end of every iteration. The same parallel computational flow from Figure 3 is used in the parallel Block Cimmino solver. The only difference is a call to the MA27 Solve subroutine to solve the augmented subsystems and compute a set of projections δ^i .

The scheduler may redistribute subsystems to improve the current workload distribution. This redistribution may take place after the MA27 Analyse phase, MA27 Factorize phase, or during the Block Cimmino iterations. Moreover, the user specifies to the scheduler the different stages of the parallel solver where redistribution is allowed. Given the high expense of moving a subsystem between processors (move all the data structures involved in the solution of a subsystem, and update the neighbourhood information), we recommend allowing redistribution only before the MA27 Factorization phase started because there are many data structures that are created per subsystem during the solve phase and sometimes the time to relocate these data structures across the network is more expensive than letting the unbalanced parallel solver finish its execution.

In Table 4, we present some preliminary results of the Block Cimmino solver. We ran in a heterogenous environment of five SUN Sparc 10, and three IBM RS600 workstations. We used one of the IBM workstations to monitor the executions (master processor). The first test matrix is GRE1107 from the Harwell-Boeing Sparse matrix collection [9]. This matrix is partitioned into 7 blocks (6 block of 159 rows and one of 153 rows) using a block size of 8 for Block-CG.

As a second test matrix, we consider a problem that comes from a two dimensional wing profile at transonic flow (without chemistry effects). The problem is discretized using a mesh of 80 by 32 points. This leads to an unsymmetric, diagonal dominant, block tridiagonal matrix of order $80 \times 32 \times 3$. In this case we test with three different partitionings. We use block size of 4 for the Block-CG algorithm only to increase the problem granularity since the problem converges very fast even with a block size of one for Block-CG.

The numbers inside parenthesis in Table 4 show the relations between an execution time with a given number of slave processors and the execution time of the same problem with a single slave processor. We do not anticipate speed-ups in a network of workstations and we expect the Parallel Block Cimmino solver to perform better in parallel heterogeneous environments where we can take advantage of clusters of processors, and very different pro-

cessing capabilities. Besides, we conclude that the Parallel Block Cimmino will provide in a "reasonable" time a solution to a problem that cannot be solved in a single processor.

N.Slaves		Transonic Flow								
IBM	SUN	GRE1107		10 Blks			16 Blks		11 Blks	
1	0	205449	(1.0)	78079	(1.0)	77757	(1.0)	77579	(1.0)	
1	2	161969	(1.3)	53352	(1.5)	75297	(1.0)	48492	(1.6)	
0	3	201517	(1.0)	47479	(1.6)	64349	(1.2)	41226	(1.9)	
1	4	249802	(0.8)	40966	(1.9)	44352	(1.8)	50895	(1.5)	
0	5	256320	(0.9)	33916	(2.3)	36200	(2.1)	42721	(1.8)	

Table 4: Preliminary results of the parallel Block Cimmino solver. Times shown in table are in milliseconds.

References

- [1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 1992.
- [2] M. Arioli, I. S. Duff, J. Noailles, and D. Ruiz. A block projection method for sparse matrices", *SIAM J. Scientific and Statistical Computing* 1992, **13**, pp 47-70.
- [3] M. Arioli, I. S. Duff, D. Ruiz, and M. Sadkane. Block Lanczos techniques for accelerating the block Cimmino method *CERFACS TR/PA/92/70*, Toulouse, France, 1992.
- [4] R.H. Bartels, G.H. Golub, and M.A. Saunders. Numerical techniques in mathematical programming. In *Nonlinear programming* J. B. Rosen, O.L. Mangasarian, and K. Ritter, eds., Accademic Press, New York, 1970.
- [5] R. Bramley and A. Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM J. Scientific and Statistical Computing* 1992, **13**, pp 168-193.
- [6] S.L. Campbell and C.D. Meyer, Jr. *Generalized inverses of linear transformations*. Pitman, London, 1979.
- [7] G. Cimmino. Calcolo approssimato per le soluzioni dei sistemi di equazioni lineari. *Ricerca Sci. II*, **9**, I, pp 326-333, 1938.
- [8] I.S. Duff and J.K. Reid. The multifrontal solution of indefinite sparse linear systems. *ACM Trans. Math. Softw.* **9**, pp 302-325, 1983.
- [9] I.S. Duff, R.G. Grimes and J.G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release 1). *RAL 92-086* Central Computing Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, 1992
- [10] G.D. Hachtel. Extended applications of the sparse tableau approach - finite elements and least squares. In *Basic question of design theory* W.R. Spillers, ed., North Holland, Amsterdam, 1974.
- [11] L.A. Hageman and D. M. Young. *Applied Iterative Methods*. Academic Press, London, 1981.
- [12] M. R. Hestenes and E. L. Stiefel. Methods of conjugate gradient for solving linear systems. *Nat. Bur. Std. J. Res.* **49**, pp 409-436, 1952.
- [13] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications* 1980,**29**, pp 293-322.
- [14] D. Ruiz. Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment. *CERFACS TH/PA/92/6*. Toulouse, France, 1992.