Scalable Computing: Practice and Experience
Volume 10, Number 4, pp. 373–383. http://www.scpe.org
ISSN 1895-1767
© 2009 SCPE

PARALLEL NUMERICAL SOLUTION OF ABD AND BABD LINEAR SYSTEMS ARISING FROM BVPS*

PIERLUIGI AMODIO† AND GIUSEPPE ROMANAZZI‡

Abstract. We consider linear systems with coefficient matrices having the ABD or the Bordered ABD (BABD) structures. These systems arise in the discretization of BVPs for ordinary and partial differential equations with separated and non-separated boundary conditions, respectively. We describe the cyclic reduction algorithm for the solution of BABD linear systems which allowed us to write the codes *BABDCR* and *GBABDCR* (the latter code is suitable for matrices with a more generic BABD structure). A comparison of the *GBABDCR* code with respect to the well-known sequential code *COLROW* on ABD linear systems is then analysed. We report some tests on an OpenMP Fortran 90 parallel version of the *GBABDCR* code and finally we discuss about the use of *GBABDCR* inside the BVP code *BVP_SOLVER*.

Key words: boundary value problems, ABD and BABD linear systems, parallel solution

1. Introduction. The discretization of Boundary Vale Problems for ordinary and partial differential equations leads often to linear systems with Almost Block Diagonal (ABD) structure in case of separated boundary conditions, and Bordered ABD structure (BABD) in case of non-separated conditions (see [3, 18, 24]).

BABD linear systems

$$Ax = f \tag{1.1}$$

have the coefficient matrix A with the following sparsity structure

$$A = \begin{pmatrix} B_a & & & & & & B_b \\ V_1 & & & & & & \\ & V_2 & & & & & \\ & & V_3 & & & & \\ & & & V_4 & & & \\ & & & & \ddots & & \\ & & & & & \ddots & \\ & & & & & & V_N \end{pmatrix}. \tag{1.2}$$

We recognize the boundary blocks B_a and B_b on the first row, and the block rows V_i which have some columns that overlap the blocks on the adjacent rows. For this reason, each block V_i can be represented as

$$V_i = \begin{pmatrix} S_{i-1} & T_i & R_i \end{pmatrix}, \tag{1.3}$$

where the columns of S_{i-1} and R_i overlap columns of V_{i-1} and V_{i+1}, respectively. Also V_1 and V_N have the same structure in (1.3): the first columns of V_1 (i. e., S_0) are overlapped with those of B_a while the last columns of V_N (i. e., R_N) are overlapped with those of B_b, see (1.2). We set the size of each block V_i equal to $n_i \times (m_{i-1} + k_i + m_i)$, where m_i is the number of overlapped columns between the blocks V_i and V_{i+1}, and k_i

*Work developed within the project "Numerical methods and software for differential equations"
†Dipartimento di Matematica, Università di Bari, Bari, Italy (amodio@dm.uniba.it)
‡Departamento de Matemática, Universidade de Coimbra, Coimbra, Portugal (roman@mat.uc.pt)

is the number of the non-overlapped columns of V_i; the size of the boundary blocks B_a and B_b is $n_0 \times m_0$ and $n_0 \times m_N$, respectively. Since A is a square matrix we have

$$\sum_{i=0}^{N} n_i = m_0 + \sum_{i=1}^{N} (m_i + k_i); \tag{1.4}$$

it is also supposed that N is much larger with respect to each n_i, m_i and k_i.

The ABD structure differs from the BABD structure for the presence of boundary blocks that have some null rows (the non-null rows of B_a correspond to null rows of B_b and viceversa). In this case it is preferable to refer to B_a and B_b as blocks of size $n_a \times m_0$ and $n_b \times m_N$, respectively. We impose then that no rows are overlapped between B_a and B_b and we set $n_0 = n_a + n_b$ in (1.4). Moreover, B_b is located after V_N as the last block row so that the coefficient matrix can be represented as

$$A = \begin{pmatrix} B_a & & & & & & \\ V_1 & & & & & & \\ & V_2 & & & & & \\ & & V_3 & & & & \\ & & & V_4 & & & \\ & & & & \ddots & & \\ & & & & & \ddots & \\ & & & & & V_N & \\ & & & & & & B_b \end{pmatrix}. \tag{1.5}$$

We note that ABD linear systems can be also solved by BABD solvers; this implies an higher computational cost and fill-in, see [24]. Conversely, a doubling of the size of each block is required for solving a BABD linear system using an ABD solver.

Since ABD linear systems are easier to solve than BABD systems, historically the former problem has received much more attention and several codes have been proposed. We quote the package *SOLVEBLOK* [14] that uses Gaussian elimination with partial pivoting to ensure stability and requires fill-in. On the contrary, the packages *COLROW* and *ARCECO* in [15] are based on a modified version of Varah's alternate row and column stable elimination [27] which exploits the structure of the ABD matrices to avoid fill-in. In particular, *COLROW* solves ABD linear systems with blocks V_i of constant size $n \times (2m + k)$, so that we have $n = m + k$ and $n_i = n$, $m_i = m$ and $k_i = k$ for each $i = 1, \ldots, N$.

Most nonlinear BVP packages employ ABD packages. The BVP code *COLSYS* [9] uses *SOLVEBLOK* to solve ABD linear systems arising from the use of orthogonal spline collocation (OSC) at Gauss points with B-spline bases. *COLNEW* [11] uses a modified version of *SOLVEBLOK* to solve ABD linear systems arising from the application of OSC at Gauss points with monomial spline bases. Both the Mono Implicit Runge Kutta (MIRK) code with defect control *MIRKDC* [17] and its new implementation *BVP_SOLVER* [28] (the latter solves a wider class of BVPs with respect the former) use *COLROW* as solver for the obtained ABD systems. Modified versions of *COLROW* are used in the deferred correction code *TWPBVP* [13]. Other versions of *COLROW* are used in COLMOD [26], a modified version of *COLNEW*, and in *ACDC* [12], that uses automatic continuation and OSC at Lobatto points to solve singularly perturbed BVODEs.

For what concerns the numerical solution of BABD systems (1.1)-(1.2), a common idea is to double the size of the system (that is the number of unknowns) in order to obtain an equivalent ABD system that can be solved with ABD solvers. Of course, this involves a high computational cost. In fact, since the computational cost of a general ABD solver is cubic with the dimension of the blocks, then its cost for solving a BABD linear

system is eight times larger with respect to the solution of an ABD linear system with blocks of the same size as the given BABD system.

The first available package for solving BABD systems is *RSCALE*; it is a shared memory parallel code used inside *PMIRKDC* [21], a parallel version of *MIRKDC*. In [7, 8] the cyclic reduction algorithm is applied to the BABD matrix (1.2) with a simplified structure: the blocks S_i and R_i are square of dimension m and the blocks T_i are null. This structure is also used by *RSCALE*. The coefficient matrix considered is then block lower bidiagonal with an additional block in the right-upper corner. The sequential code *BABDCR*, introduced in [8] and available at the url

`http://www.netlib.org/toms/859`.

solves this kind of linear systems. Moreover, in [4] we have generalized the code *BABDCR* to solve BABD linear systems with the general structure (1.1)-(1.2) where each block V_i have size $n \times (2m+k)$, with $k > 0$ and $n = m + k$. This latter code is called *GBABDCR* and is available on the net at the url

`http://www.pitagora.dm.uniba.it/~romanazzi/babdcr.html`.

In [4, 5, 8, 24] *BABDCR* and *GBABDCR* have been compared to *RSCALE* and *COLROW* for solving BABD systems. The theoretical and numerical results show that *COLROW* has a computational cost which is till 3 times larger than *GBABDCR*. This means that each code for BVPs with separated boundary conditions (for example, *BVP_SOLVER*) using *COLROW* to solve the associated ABD linear systems, may be generalized to the solution of BVPs with non-separated boundary conditions by just replacing (the calls to) the linear solver *COLROW* with *GBABDCR*. Moreover, *BABDCR* performs better (resulting faster and more precise) and has the same degree of parallelism with respect to *RSCALE*. Therefore, the efficiency of *PMIRKDC* can be improved by replacing *RSCALE* with a parallel version of *BABDCR*.

This work originates from the observation that nowadays personal computers have motherboards with more CPUs (or core-processors), and this permits to overcome the physical limitations (such as speed and memory) of a single core-processor. Moreover these multi-core processors are easily accessible, and we can use them to speed-up the execution of each numerical code if the underlying algorithm is parallelizable. In our case, since cyclic reduction has an obvious parallel implementation, we can speed up the solution of ABD linear systems by implementing *GBABDCR* on such parallel architectures.

We propose, in fact, a shared memory implementation of *GBABDCR* and compare it with *COLROW* (that is not parallelizable) on multi-core computers. We believe that the replacement of *COLROW* in the existing (previously cited) BVODE packages with the parallel implementation of *GBABDCR*, can lead to a twofold advantage: the reduction of the computational cost when the code is run on multi-core processors, and the solution with no extra cost of BVODEs with non-separated boundary conditions.

The paper is organized as follows: in the next section we briefly sketch the cyclic reduction algorithm applied to general BABD linear systems, in Section 3 we explain the strategy used to parallelize the code on shared and distributed memory computers, and finally in the last section we compare the shared memory parallel implementation of *GBABDCR* with *COLROW* in the solution of ABD systems and we discuss the performance of *BVP_SOLVER* when *GBABDCR* replaces *COLROW*.

2. The cyclic reduction algorithm. Let us rewrite the coefficient matrix (1.2)-(1.3) as

$$A = \begin{pmatrix} B_a & & & & & & & & & B_b \\ S_0 & T_1 & R_1 & & & & & & & \\ & & S_1 & T_2 & R_2 & & & & & \\ & & & & S_2 & T_3 & R_3 & & & \\ & & & & & & & \ddots & & \\ & & & & & & & & S_{N-1} & T_N & R_N \end{pmatrix}. \tag{2.1}$$

In accordance with the structure of (2.1) we define the right hand side f of the linear system (1.1) as $f = \begin{pmatrix} f_0^T & f_1^T & \dots & f_N^T \end{pmatrix}^T$ where each f_i is of length n_i, and the solution vector

$$x = \begin{pmatrix} z_0^T & w_1^T & z_1^T & \dots & w_{N-1}^T & z_N^T \end{pmatrix}^T,$$

where z_i and w_i are of length m_i and k_i, respectively.

We solve the system (1.1) using a block cyclic reduction algorithm, that is, a recursive approach that reduces the original linear system to subsystems with a smaller number of unknowns. In this process the first and the last unknowns, z_0 and z_N, are always among the unknowns of the successive reduced systems; moreover, the first row containing the boundary blocks is unchanged in the reduction process. The boundary blocks B_a and B_b are then maintained in the first row of each reduction step.

Following [4], we consider a reduction step to eliminate, locally in each block V_i, the odd unknowns w_i of the solution vector x. We observe that, since A is a non-singular matrix, the blocks T_i of size $n_i \times k_i$, have full rank k_i because they are not overlapped by adjacent V_i blocks. We may then compute the factorization:

$$\widetilde{P}_i T_i = \begin{pmatrix} \widetilde{L}_i \\ \widetilde{G}_i \end{pmatrix} \widetilde{U}_i = \begin{pmatrix} I & \\ \widetilde{F}_i & I \end{pmatrix} \begin{pmatrix} \widetilde{L}_i \widetilde{U}_i \\ O \end{pmatrix} \tag{2.2}$$

where $\widetilde{P}_i$ is a suitable permutation matrix, $\widetilde{L}_i$ and $\widetilde{U}_i$ are square matrices, and $\widetilde{F}_i = \widetilde{G}_i \widetilde{L}_i^{-1}$.

Multiplying V_i on the left by $\widetilde{P}_i$ and the inverse of the lower triangular matrix in the last term of (2.2) we obtain

$$\begin{pmatrix} I & \\ -\widetilde{F}_i & I \end{pmatrix} \widetilde{P}_i \begin{pmatrix} S_{i-1} & T_i & R_i \end{pmatrix} = \begin{pmatrix} \widetilde{S}_{i-1} & \widetilde{L}_i \widetilde{U}_i & \widetilde{R}_i \\ \widehat{S}_{i-1} & & \widehat{R}_i \end{pmatrix}. \tag{2.3}$$

Analogously, we perform the same operations on the right-hand side f_i, thus obtaining corresponding vectors $\tilde{f}_i$ and $\widehat{f}_i$ for the right side. The row with the boundary blocks and the second row of (2.3) (for each $i = 1, \ldots, N$) give the linear system

$$\begin{pmatrix} B_a & & & & B_b \\ \widehat{S}_0 & \widehat{R}_1 & & & \\ & \widehat{S}_1 & \widehat{R}_2 & & \\ & & \ddots & \ddots & \\ & & & \widehat{S}_{N-1} & \widehat{R}_N \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix} = \begin{pmatrix} f_0 \\ \widehat{f}_1 \\ \widehat{f}_2 \\ \vdots \\ \widehat{f}_N \end{pmatrix}. \tag{2.4}$$

which has dimension equal to $\displaystyle\sum_{i=0}^{N} m_i = n_0 + \sum_{i=1}^{N} (n_i - k_i)$ and no longer depends on the unknowns w_i. These unknowns will be computed in the last step of the back-substitution phase (when all the z_i will be known), by using the first row of (2.3):

$$\tilde{L}_i \tilde{U}_i w_i = \tilde{f}_i - \tilde{S}_{i-1} z_{i-1} - \tilde{R}_i z_i.$$

Factorization (2.3) does not require additional memory since $\widetilde{F}_i$ may be saved together with L_i and U_i in place of T_i. Therefore, this first reduction should be considered as a (completely parallelizable) initial step to be applied to ABD or BABD matrices in order to simplify their structure.

Returning to the solution of (1.1), system (2.4) may be further on reduced by considering the cyclic reduction algorithm in [7, 8] (even if blocks $\widehat{S}_i$ and $\widehat{R}_i$ are not square). At first we compute the LU factorization of the $(n_i - k_i + n_{i+1} - k_{i+1}) \times m_i$ matrix (of rank m_i)

$$P_i \begin{pmatrix} \widehat{R}_i \\ \widehat{S}_i \end{pmatrix} = \begin{pmatrix} L_i \\ G_i \end{pmatrix} U_i = \begin{pmatrix} I & \\ F_i & I \end{pmatrix} \begin{pmatrix} L_i U_i \\ O \end{pmatrix}$$

that, applied to the block rows of index i and $i+1$ in (2.4), gives

$$\begin{pmatrix} I & \\ -F_i & I \end{pmatrix} P_i \begin{pmatrix} \widehat{S}_{i-1} & \widehat{R}_i & \\ & \widehat{S}_i & \widehat{R}_{i+1} \end{pmatrix} = \begin{pmatrix} \bar{S}_{i-1} & L_i U_i & \bar{R}_i \\ S'_{i-1} & & R'_i \end{pmatrix}. \tag{2.5}$$

The second row of the right hand-side of (2.5) is independent of z_i and allows to obtain, for $i = 1, 3, 5, \ldots$, a reduced system similar to (2.4) but with $\lceil N/2 \rceil + 1$ block rows and unknowns z_i with even indices. Conversely,

the first row in (2.5) may be used to compute z_i from z_{i-1} and z_{i+1}. Factorization (2.5) requires additional memory for the fill-in blocks F_i.

Iterating this last step on the successively reduced systems we obtain, after $\lceil \log_2 N \rceil$ steps, a 2×2 block (full) linear system

$$\begin{pmatrix} B_a & B_b \\ S_0^* & R_N^* \end{pmatrix} \begin{pmatrix} z_0 \\ z_N \end{pmatrix} = \begin{pmatrix} f_0^* \\ f_N^* \end{pmatrix}. \tag{2.6}$$

The factorization and the solution of (2.6) gives the first and the last unknowns of the vector x. Successively, a back substitution phase allows us to compute, in reverse order, all the other unknowns.

If m and k are constant (with $n = m + k$), the computational cost of this algorithm is

$$(\tfrac{14}{3}m^3 + 4m^2k + 2mk^2 + \tfrac{2}{3}k^3)N. \tag{2.7}$$

If $k = 0$ (all columns are overlapped), we have from (2.7) that the cost is $\frac{14}{3}m^3N$ as for the *BABDCR* algorithm, see [8]. The additional memory requirement (fill-in) is always m^2N (it does not depend on k) since the factorization of the blocks T_i in (2.2) does not require fill-in.

The typical dimensions of the BABD linear system arising from BVPs consists of small dimensions k, m of each row block, with respect to a large number of row blocks N.

In order to make a fair comparison, the computational cost of *COLROW* is

$$(\tfrac{5}{3}m^3 + 4m^2k + 3mk^2 + \tfrac{2}{3}k^3 + mn_an_b - (2m + k)kn_a)N, \tag{2.8}$$

where n_a and n_b, with $n_a + n_b = m$, denote the number of rows of the initial and of the final boundary block, respectively; remember that *COLROW* solves only ABD linear systems. Supposing $n_a = n_b = m/2$, we have that *GBABDCR* costs at most 2.4 times more than *COLROW* and this ratio decreases when k increases (if $k = m$ the ratio is 1.4).

3. Parallel implementation. In this section we analyze and compare the main properties of parallel cyclic reduction algorithms written for shared and distributed memory architectures.

The cyclic reduction algorithm has a straightforward parallel implementation which has been described in several papers. See, for example, [1, 2, 6] where parallel cyclic reduction is used to solve tridiagonal systems and [5, 7, 24] where it is applied to BABD systems.

Let p the number of processors used[1], the first phase of parallel reduction requires that each processor reduces a contiguous set of block rows,

$$\begin{pmatrix} S_{i-1} & T_i & R_i & & & & & \\ & & S_i & T_{i+1} & R_{i+1} & & & \\ & & & & & \ddots & & \\ & & & & & & S_{j-1} & T_j & R_j \end{pmatrix}, \tag{3.1}$$

to a single row

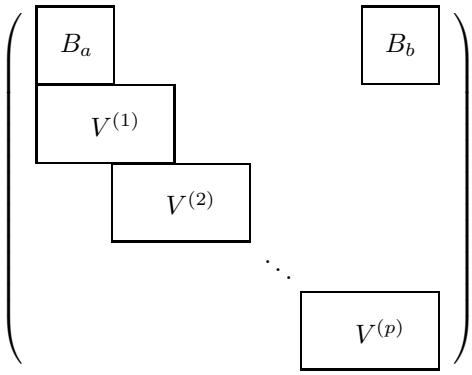
$$\begin{pmatrix} S_{i-1}^* & R_j^* \end{pmatrix}. \tag{3.2}$$

In particular, $q = N - \lceil N/p \rceil p$ processors reduce $\lfloor N/p \rfloor$ block rows and the remaining $p - q$ processors reduce $\lceil N/p \rceil$ block rows each. Note that, when q is non-zero, a different workload per processor is required.

At the end of this phase, each processor stores the block row (3.2) in the first and last m^2 locations of the assigned contiguous block rows (3.1) in place of S_{i-1} and R_j. The fill-in of these reductions is stored locally in a private array of each processor, when a distributed memory architecture is used, or consists of contiguous blocks generated by each single processor, in case of a shared memory architecture.

[1]We use the generic term "processor" to refer to each core per physical processor used in the parallel architecture considered.

Therefore, the first phase of reduction produces a reduced BABD system with coefficient matrix



, (3.3)

where $V^{(i)}$ is computed by the i-th processor; this phase is then completely parallelizable.

Let us now analyze the parallel solution of the system with the coefficient matrix (3.3). Supposing, for sake of simplicity, that p is a power of 2, the parallel MPI algorithm (as in [7, 24, 5]), for a distributed memory architecture, requires $\log_2 p$ synchronizations, communications between processors, and reduction steps. On shared memory architectures, clearly we do not require communications between processors because both the coefficient matrix and the right-hand side of the original system (1.1) are shared among processors. However, $\log_2 p$ synchronizations between processors are necessary.

This part of the algorithm determines, in both the architectures, the 2×2 block system (2.6). The solution of (2.6) is the only sequential part of the algorithm and is followed by the back-substitution phase which still has much parallelism inside.

On a distributed memory architecture it is made possible that all the back-substitution phase requires no more synchronization or communication among the processors by considering bidirectional communications in the reduction phase (see Figure 3.1 and [6] for more details). This means that a copy of (2.6) is solved (concurrently) on all the processors.

On the other hand, on shared memory architectures, the number of processors is halved at each of the last $\log_2 p$ reduction steps and (2.6) is solved on a single processor. Then the number of processors is doubled for the first $\log_2 p$ steps of back-substitution with a synchronization after each step.

In conclusion, if $p \ll N$ the total operation count for each processor is essentially $1/p$ times the cost of (2.7). On shared memory architectures, the number of synchronizations is $2 \log_2 p$. The parallel code does not require additional memory with respect to the sequential code and only a few local variables are needed on each processor. On distributed memory architectures, the number of synchronizations is only $\log_2 p$ but each processor needs to maintain $\log_2 p$ blocks of size $m \times 2m$ to avoid synchronization in the back-substitution and, moreover, $\log_2 p$ communications of $m \times 2m$ arrays and vectors of length m are necessary in the reduction phase.

Since distributed parallel architectures can have even hundreds processors, the main advantage of the parallel algorithm for these architectures is the possibility to strongly reduce the cost of the reduction and back-substitution phases by increasing the number of processors used. In such a case, however, a moderate overhead (of order $\log_2 N$) appears due to the presence of synchronizations and communications.

As already mentioned, shared memory architectures are nowadays much easier to access. They have a limited number of processors (up to 4 in our tests), but the parallel algorithms for these architectures have a double advantage: there is no need to send the initial data to the local memory of each processor and there is no large communication step after each synchronization. This means that, for a given number of processors, these algorithms can lead to a better speed-up than that observed on distributed memory architectures.

4. Numerical results. In this section we report some numerical tests to compare *COLROW* with the shared memory (OpenMP) parallel version of *GBABDCR* (here called *GBABDCR_OMP*) in the solution of ABD systems on multi-core computers. Our aim is to stress that *GBABDCR_OMP* may be efficiently employed in each BVP solver that requires the solution of ABD/BABD systems on the new multi-core personal computers.

For this reason we analyze the execution times of sequential runs performed on an alpha EV6.7 (21264A, 667 MHz) and of parallel runs performed on an Intel Core 2, Quad CPU Q9550, (2.83 GHz) multi-processor with 4 cores. We consider ABD linear systems with blocks of the same size and with the same overlap ($m_i = m$, $k_i = k$ and $n_i = n = k + m$). Moreover, we set the number of initial and final conditions equal to $m/2$.

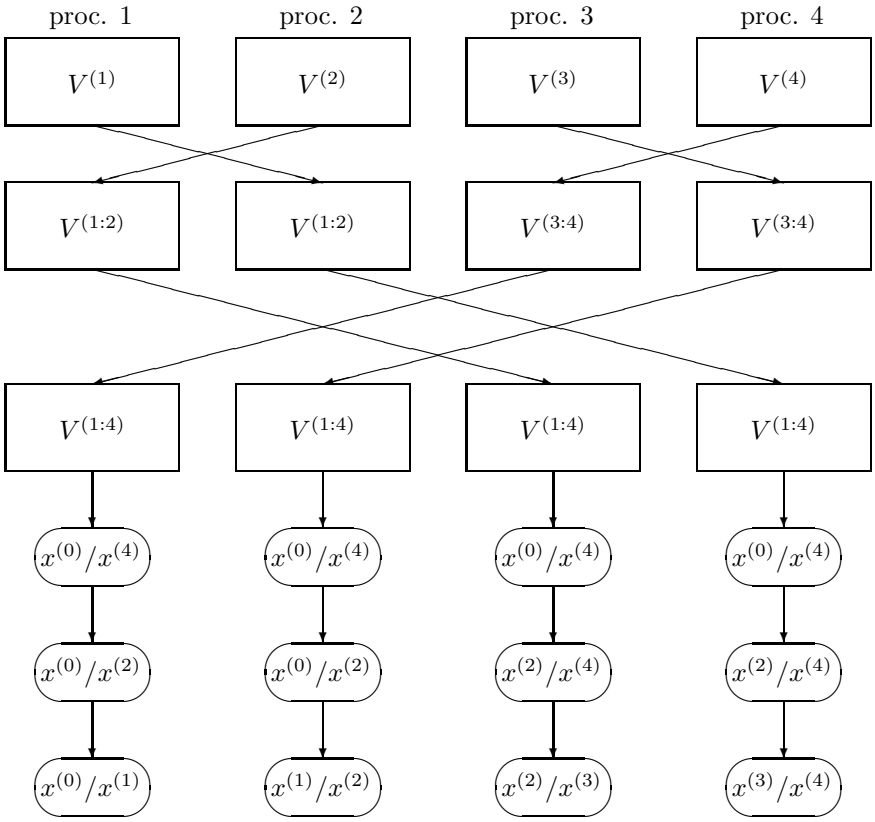


FIG. 3.1. *Communications among processors on a distributed memory architecture with* $p = 4$.

In Tables 4.1 and 4.2 we compare the elapsed time of some *GBABDCR* and *COLROW* runs using the sequential computer to estimate the theoretical operation counts. We observe that, for $k = 0$ the ratio between the elapsed time of *GBABDCR* and *COLROW* decreases from 3.26 to 2.32 (the expected value is 2.4) as we increase n while for $k = m$ the ratio decreases from 4.75 to 1.64 (the expected value is 1.4). A reason of this strange behaviour is due to the large use in *GBABDCR* of BLAS3 routines [16] (such as DGEMM, DTRSM) which have poor performance for small matrices (see tests in [22, 23]) and make this algorithm more efficient for large k and m only.

TABLE 4.1
Elapsed times (in seconds) of GBABDCR and COLROW for solving ABD linear systems with $N = 10000$ *and variable* $n = m$ *(*$k = 0$*)*

n	GBABDCR	COLROW	GBABDCR/COLROW
4	0.0795	0.0244	3.2600
8	0.2525	0.0947	2.6675
12	0.5768	0.2294	2.5149
16	1.0753	0.4377	2.4565
20	1.6946	0.7291	2.3243

These results show therefore that, for small size n of each block, it is quite difficult that *GBABDCR* overcomes the performance of *COLROW* on parallel machines. Effectively for $n < 8$ we were not able to lower *COLROW* timings by using *GBABDCR_OMP* on 4 cores.

In Tables 4.3-4.6 we show the performance of *GBABDCR_OMP* running on shared memory architectures for $N = 10000, 20000, 40000$ and $n = 8, 16$.

We note that, when $N \gg p$, the computational cost of the parallel algorithm scales with the number of processors and in all the considered cases *GBABDCR_OMP* on 4 cores is faster than *COLROW*. Therefore, on

TABLE 4.2
Elapsed times of GBABDCR and COLROW for solving ABD linear systems with $N = 10000$ and variable $n = 2m$ $(k = m)$

n	GBABDCR	COLROW	GBABDCR/COLROW
4	0.0986	0.0207	4.7529
8	0.1840	0.0712	2.5822
12	0.3582	0.1940	1.8465
16	0.5558	0.3235	1.7179
20	0.9118	0.5546	1.6441

TABLE 4.3
Elapsed times of GBABDCR_OMP and COLROW on a quad-core machine for ABD linear systems with $n = m = 8$ $(k = 0)$ and variable N. Speed-up is computed as the ratio between the elapsed time of COLROW and GBABDCR_OMP

		GBABDCR_OMP			speed-up	
N	*COLROW*	1 core	2 cores	4 cores	2 cores	4 cores
10000	0.091	0.199	0.102	0.062	0.892	1.468
20000	0.186	0.398	0.204	0.132	0.912	1.409
40000	0.360	0.797	0.410	0.224	0.878	1.607

a multicore architecture, we can effectively speed-up the performance of any BVP solver when the size of the problem is sufficiently large, just replacing *COLROW* with *GBABDCR_OMP*. Since a code for BABD systems may be directly applied to ABD systems, the modifications in the code are limited to a few instructions.

In particular, in *BVP_SOLVER* we have inserted a subroutine that creates the non-separated boundary blocks of the linear system starting from the given separated boundary conditions and we have added a fill-in vector which is only required by *GBABDCR_OMP*. We point out that in *BVP_SOLVER* the resulting linear systems have a BABD structure with $k = 0$ and $m = n$.

As an example, we have considered the solution of a 8×8 nonlinear system of equations describing fluid injection through one side of a long vertical channel (see [10, Example 1.4]). We have taken into account this problem with the following parameters

$$R = 1000, \quad P = 0.7 * R, \quad \text{METH} = 2$$

and fixed equal to 1 both the initial guess for the solution and the unknown parameter A.

Table 4.7 shows the results obtained by setting the exit tolerance equal to 10^{-6}, 10^{-7} and 10^{-8}. A few comments need to be done on this example. The number of requested linear system solutions (SOLV) is about 8 times the number of factorizations (FACT). Since this size of the ODE is just $n = 8$ and in any ABD/BABD code the computational cost of the factorization phase (FACT) is about n times that of the solution phase (SOLV), then we expect the execution time of the two phases to be approximately equivalent.

From Table 4.7 we observe that the percentage of time required by SOLV using *GBABDCR* is higher than that requested by FACT (the opposite happens when we use *COLROW*) and it is therefore difficult to obtain a large speed-up with the replacement of the linear algebra solver. By considering only the linear algebra part (see % LIN_ALG in the table) of *BVP_SOLVER*, *GBABDCR* reduces its elapsed time of a factor in the range [1.5, 1.7] when we use 4 cores instead of 1. This is anyway sufficient to obtain on 4 cores an execution faster than *COLROW*.

We have to specify that on BVPs of size smaller than 8 we were not able to improve timings obtained with *COLROW*. The size of the considered problem is anyway small, and this implies two negative aspects: linear algebra is not the most time consuming part of the algorithm and, for such small dimensions, *COLROW* is much better than *GBABDCR*. Nevertheless, it is clear that parallelism is really more useful to reduce timing in presence of very large nonlinear BVPs.

A better speed-up is, in fact, obtained when we solve BVODE of larger size. Table 4.8 shows, for example, some results of the application of the *BVP_SOLVER* to a kidney problem of size 21×21 that describes the mass and the energy balance of a renal counter-flow system, see [10]. We note that the Linear Algebra part requires a larger portion of computation (between 60% and 70%) with respect to the previous example and, therefore, the use of *GBABDCR* sensibly improves the performance of *BVP_SOLVER*, when a multi-core computer is

TABLE 4.4
Elapsed times of GBABDCR_OMP and COLROW on a quad-core machine for ABD linear systems with $n = m = 16$ ($k = 0$) and variable N. Speed-up is computed as the ratio between the elapsed time of COLROW and GBABDCR_OMP

		GBABDCR_OMP			speed-up	
N	*COLROW*	1 core	2 cores	4 cores	2 cores	4 cores
10000	0.584	1.038	0.525	0.299	1.112	1.953
20000	1.174	2.054	1.049	0.552	1.119	2.127
40000	2.332	4.131	2.098	1.092	1.112	2.136

TABLE 4.5
Elapsed times of GBABDCR_OMP and COLROW on a quad-core machine for ABD linear systems with $n = 2m = 8$ ($k = m$) and variable N. Speed-up is computed as the ratio between the elapsed time of COLROW and GBABDCR_OMP

		GBABDCR_OMP			speed-up	
N	*COLROW*	1 core	2 cores	4 cores	2 cores	4 cores
10000	0.060	0.122	0.066	0.041	0.909	1.463
20000	0.120	0.249	0.137	0.078	0.876	1.539
40000	0.240	0.500	0.271	0.175	0.886	1.371

used. Finally, we emphasize that the performance of *BVP_SOLVER* can be further improved with *GBABDCR*, by considering that the original problem has three nonseparated boundary conditions and, therefore, it can be recast so that a 18×18 BABD linear system is obtained.

REFERENCES

[1] P. Amodio, L. Brugnano. Parallel factorizations and parallel solvers for tridiagonal linear systems. *Linear Algebra Appl.* **172**, 347–364, 1992.

[2] P. Amodio, L. Brugnano, T. Politi. Parallel factorizations for tridiagonal matrices. *SIAM J. Numer. Anal.* **30**, 813–823, 1993.

[3] P. Amodio, J. R. Cash, G. Roussos, R. W. Wright, G. Fairweather, I. Gladwell, G. L. Kraut and M. Paprzycki. Almost block diagonal linear systems: sequential and parallel solution techniques, and applications. *Numer. Linear Algebra Appl.* **7**, no. 5, 275–317, 2000.

[4] P. Amodio, I. Gladwell and G. Romanazzi. Numerical solution of general Bordered ABD linear systems by cyclic reduction. *J. Numer. Anal., Industrial and Applied Mathematics* **1**, no. 1, 5–12, 2006.

[5] P. Amodio, I. Gladwell and G. Romanazzi. An algorithm for the solution of Bordered ABD linear systems arising from Boundary Value Problems. *Multibody Dynamics 2007, ECCOMAS Thematic Conference*, C. L. Bottasso, P. Masarati, L. Trainelli (eds.), Milano (Italy), June 25–28, 2007.

[6] P. Amodio, N. Mastronardi. A parallel version of the cyclic reduction algorithm on a Hypercube. *Paral. Comput.* **19**, 1273–1281, 1993.

[7] P. Amodio and M. Paprzycki. A cyclic reduction approach to the numerical solution of boundary value ODEs. *SIAM J. Sci. Comput.* **18**, no. 1, 56–68, 1997.

[8] P. Amodio and G. Romanazzi. BABDCR: a Fortran 90 package for the solution of Bordered ABD systems. *ACM Trans. Math. Software* **32**, no. 4, 597–608, 2006. (Available on the web address "http://www.netlib.org/toms/859")

[9] U. M. Ascher, J. Christiansen and R.D. Russell. Algorithm 569: COLSYS: Collocation Software for Boundary-Value ODEs. *ACM Trans. Math. Software* **7**, no. 2, 223–229, 1981.

[10] U. M. Ascher, R.M.M. Mattheij, and R.D. Russell. Numerical Solution of Boundary Value Problems for Ordinary Differential Equations. SIAM Classics In Applied Mathematics, 1995.

[11] G. Bader and U. Ascher. A new basis implementation for a mixed order boundary value ODE solver. *SIAM J. Sci. Statist. Comput.* **8**, no. 4, 483–500, 1987.

[12] J. R. Cash, G. Moore and R.W. Wright. An automatic continuation strategy for the solution of singularly perturbed nonlinear boundary value problems. *ACM Trans. Math. Software* **27**, no. 2, 245–266, 2001.

[13] J. R. Cash and R. W. Wright. A deferred correction method for nonlinear two-point boundary value problems: Implementations and numerical evaluation. *SIAM J. Sci. Statist. Comput.* **12**, no. 4, 971–989, 1991.

[14] C. De Boor and R. Weiss. SOLVEBLOK: A package for solving almost block diagonal linear systems. *ACM Trans. Math. Software* **6**, no. 1, 80–87, 1980.

[15] J. C. Diaz, G. Fairweather and P. Keast. FORTRAN packages for solving certain almost block diagonal linear systems by modified alternate row and column elimination. *ACM Trans. Math. Software* **9**, no. 3, 358–375, 1983.

[16] J. J. Dongarra, J. Du Croz, I. S. Duff and S. Hammarlin, Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Software* **16**, 18–28, 1990.

[17] W. H. Enright and P. H. Muir. Runge-Kutta software with defect control for boundary value ODEs. *SIAM J. Sci. Comput.* **17**, no. 2, 479–497, 1996.

TABLE 4.6

Elapsed times of GBABDCR_OMP and COLROW on a quad-core machine for ABD linear systems with $n = 2m = 16$ ($k = m$) and variable N. Speed-up is computed as the ratio between the elapsed time of COLROW and GBABDCR_OMP

		GBABDCR_OMP			speed-up	
N	*COLROW*	1 core	2 cores	4 cores	2 cores	4 cores
10000	0.370	0.468	0.253	0.145	1.463	2.552
20000	0.742	0.938	0.506	0.278	1.466	2.669
40000	1.485	1.875	0.977	0.552	1.520	2.690

TABLE 4.7

Total elapsed times of BVP_SOLVER using COLROW and GBABDCR_OMP as linear algebra solver on a nonlinear 8×8 BVP. # FACT and # SOLV denote, respectively, the number of linear system factorizations and solutions performed by the code. % FACT and % SOLV denote, respectively, the percentage of the total elapsed time required by linear system factorizations and solutions in BVP_SOLVER. % LIN_ALG = % FACT + % SOLV denotes the percentage of the total elapsed time required by the linear algebra part.

TOL	1e-8	1e-7	1e-6
$N_{\max}$	63357	19281	6466
# FACT	21	20	20
# SOLV	170	161	151
COLROW	30.886	3.409	0.895
% FACT	5.83%	14.66%	21.16%
% SOLV	2.73%	9.20%	17.19%
% LIN_ALG	8.56%	23.86%	38.35%
GBABDCR (1 core)	31.831	3.737	1.067
% FACT	6.86%	14.84%	21.60%
% SOLV	5.42%	16.80%	29.56%
% LIN_ALG	12.28%	31.64%	51.16%
GBABDCR (2 cores)	30.881	3.435	0.917
GBABDCR (4 cores)	30.484	3.260	0.846

TABLE 4.8

Total elapsed times of BVP_SOLVER using COLROW and GBABDCR_OMP as linear algebra solver on a nonlinear 21×21 BVP. # FACT and # SOLV denote, respectively, the number of linear system factorizations and solutions performed by the code. % FACT and % SOLV denote, respectively, the percentage of the total elapsed time required by linear system factorizations and solutions in BVP_SOLVER. % LIN_ALG = % FACT + % SOLV denotes the percentage of the total elapsed time required by the linear algebra part.

TOL	1e-6	1e-4
$N_{\max}$	36128	3016
# FACT	285	177
# SOLV	738	481
COLROW	366.78	25.15
% FACT	56.7%	60.2%
% SOLV	13.7%	13.1%
% LIN_ALG	70.4%	73.3%
GBABDCR (1 core)	415.95	21.88
% FACT	39.5%	40.5%
% SOLV	21.7%	25.2%
% LIN_ALG	61.2%	65.7%
GBABDCR (2 cores)	246.51	14.42
GBABDCR (4 cores)	126.95	13.52

[18] G. Fairweather and I. Gladwell. Algorithms for almost block diagonal linear systems. *SIAM Rev.* **46**, no. 1, 49–58, 2004.
[19] B. Garrett and I. Gladwell. Solving bordered almost block diagonal systems stably and efficiently. *J. Comput. Methods Sci. Engrg.* **1**, 75–98, 2001.
[20] K. R. Jackson and R. N. Pancer. The parallel solution of ABD systems arising in numerical methods for BVPs for ODEs. Techinical Report n. 255/91, Computer Science Department, University of Toronto, 1992.
[21] P. H. Muir, R. N. Pancer and K.R. Jackson. PMIRKDC: a parallel mono-implicit Runge-Kutta code with defect control for boundary value ODEs. *Paral. Comput.* **29**, 711–741, 2003.
[22] J. R. Herrero and J. J. Navarro. Improving Performance of Hypermatrix Cholesky Factorization. In *9th International Euro-Par Conference*, 461–469, August 2003.
[23] A. Remón, E. S. Quintana-Ortí and G. Quintana-Ortí. Cholesky factorization of band matrices using multithreaded BLAS. In B. Kågström, E. Elmroth, J. Dongarra, J. Wásniewski, eds., PARA 2006. Lect. Notes Comp. Sci. **4699**, 608–616. Springer, Heidelberg.
[24] G. Romanazzi. Numerical Solution of Bordered Almost Block Diagonal linear systems arising from BVPs. Ph.D. Thesis, Università di Bari, 2006.
[25] M. Snir, S.W. Otto, S. Huss-Lederman, D. Walker and J. Dongarra. *MPI: The Complete Reference.* The MIT Press, Cambridge, Massachusetts, 1996.
[26] R. W. Wright, J. Cash and G. Moore. Mesh selection for stiff two-point boundary value problems. *Numer. Algorithms* **7**, 205–224, 1994.
[27] J. M. Varah. Alternate row and column elimination for solving certain linear systems. *SIAM J. Numer. Anal.* **13**, 71–75, 1976.
[28] L. F. Shampine, P. H. Muir and H. Xu. A User-Friendly Fortran BVP Solver. *J. Numer. Anal. Ind. and Appl. Math.* **11**, no. 2, 1201–217, 2006

Edited by: Luigi Brugnano
Received: August 27, 2009
Accepted: September 20, 2009