Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

BCYCLIC: A parallel block tridiagonal matrix cyclic solver [☆]

S.P. Hirshman [*], K.S. Perumalla, V.E. Lynch, R. Sanchez

Oak Ridge National Laboratory, Oak Ridge, Tennessee 37830, USA

ARTICLE INFO

Article history:
Received 30 December 2009
Received in revised form 27 April 2010
Accepted 30 April 2010
Available online 13 May 2010

Keywords:
Cyclic reduction
Block matrix
Dense blocks
Tridiagonal matrix
Thomas algorithm
Parallel computing

ABSTRACT

A block tridiagonal matrix is factored with minimal fill-in using a cyclic reduction algorithm that is easily parallelized. Storage of the factored blocks allows the application of the inverse to multiple right-hand sides which may not be known at factorization time. Scalability with the number of block rows is achieved with cyclic reduction, while scalability with the block size is achieved using multithreaded routines (OpenMP, GotoBLAS) for block matrix manipulation. This dual scalability is a noteworthy feature of this new solver, as well as its ability to efficiently handle arbitrary (non-powers-of-2) block row and processor numbers. Comparison with a state-of-the art parallel sparse solver is presented. It is expected that this new solver will allow many physical applications to optimally use the parallel resources on current supercomputers. Example usage of the solver in magneto-hydrodynamic (MHD), three-dimensional equilibrium solvers for high-temperature fusion plasmas is cited.

1. Introduction

1.1. Motivation and scope

The solution of the matrix equation $Ax = b$, with possible multiple right-hand sides b, arises routinely in the solution of linear and nonlinear partial differential equations (PDEs). In three-dimensional systems where two coordinates are angular and one is radial, the Fourier-transformed PDEs result in A being block tridiagonal when the underlying equations involve at most second order derivatives. The block size M is the additive Fourier extents in the two angular directions, and the number of block rows N is the number of discrete radial nodes. Such matrix structure emerges naturally in numerical simulations of hot thermonuclear plasmas confined by a magnetic toroidal field, such as those in a tokamak or stellarator. The nonlinear fluid equations that describe the equilibrium, stability and evolution of these plasmas are often discretized spectrally in the two periodic directions, while finite differences are used in the radial direction. In this case the highest derivative in the radial direction is second order, the resulting linear sub-problems that emerge when solving the nonlinear equations via iterative Newton–Krylov methods exhibit a block tridiagonal structure.

Examples of physics codes which require efficient block solvers include the ideal magnetohydrodynamic (MHD) nonlinear solvers VMEC [1] and SIESTA [2], and the linear full-wave ion cyclotron radio frequency (ICRF) code TORIC [3]. All of these codes solve equations of the form $Ax = b$, with the block tridiagonal matrix A consisting of large, dense blocks. Usually

[☆] This submission was sponsored by a contractor of the United States Government under contract DE-AC05-00OR22725 with the United States Department of Energy. The United States Government retains, and the publisher, by accepting this submission for publication, acknowledges that the United States Government retains, a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this submission, or allow others to do so, for United States Government purposes.

[*] Corresponding author. Tel.: +1 865 574 1289.
E-mail address: hirshmansp@ornl.gov (S.P. Hirshman).

0021-9991/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2010.04.049

multiple right sides b occur. Such a block solver is essential for the numerical efficiency of the SIESTA [2] code which computes high-resolution MHD equilibria in the presence of magnetic islands. Most of the computational time in SIESTA is spent in the inversion of large block matrices ($N > 100$, $M > 300$) which are repeatedly applied as preconditioners to accelerate the convergence of the linearized MHD equations toward an equilibrium state. Eventual simulations for plasmas in the International Thermonuclear Experimental Reactor (ITER, with $T \sim 15$ keV, $a \sim 2$ m, $B \sim 5$ T) will require even larger spatial resolution resulting in greater block row numbers and sizes. Therefore the efficient parallel inversion and storage of the factors of A is essential in SIESTA and was the primary motivation for the present code development.

Fig. 1 shows a SIESTA equilibrium calculation for pressure contours (magnetic surfaces) in a tokamak plasma with an internal $q = 1$ surface near the center of the plasma. The block size for this problem was $M = 273$ and there were $N = 101$ block rows. For this small sized problem, the serial calculation can be done in under 5 min on a desktop computer using Compaq Visual Fortran. Using ScaLAPACK (see below) to factor and invert the blocks improves the performance for this problem by about a factor of 5 before communication bottlenecks saturate the performance gain. As we discuss in Section 4, the new cyclic reduction routine BCYCLIC significantly extends the scalability for this, and larger, problem sizes.

1.2. Related work in parallel block tridiagonal matrix solvers

There has been considerable work in developing solution algorithms for tridiagonal matrices. Here, we briefly review work related to code development for block tridiagonal solvers. For a block tridiagonal matrix A, it is possible to obtain an exact inverse (direct solution) with no fill-in by using the well-known Thomas [4] *serial*algorithm, which is easily generalized for block sizes $M \gg 1$. While this is the fastest algorithm on a serial computer [5], it is not parallelizable since each solution step in the algorithm depends on the preceding one.

Many authors have considered efficient parallel block solvers for scalar block ($M = 1$) matrices based on cyclic reduction [6–8]. Cyclic reduction was first described by Heller [9] for block tridiagonal systems, although an efficient parallel code was not described. The new code BCYCLIC described here fills a software gap in the available codes for solving tridiagonal systems

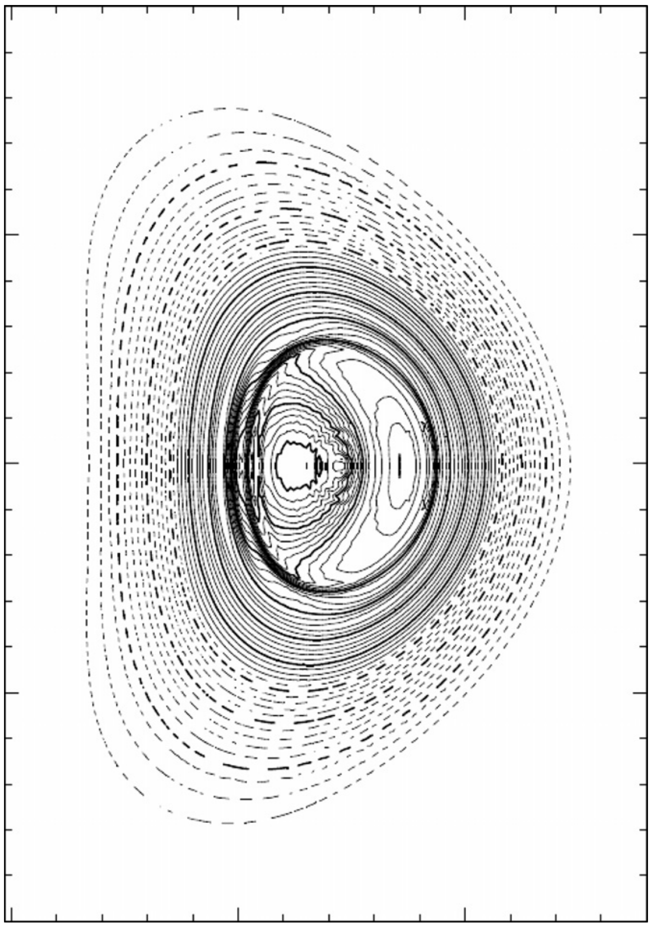

Fig. 1. Example of a tokamak equilibrium plasma with a $q = 1$ tearing mode creating an $m = 1$ island.

with large ($M \gg 1$), dense blocks. For example, the well-known BLKTRI code [10] uses cyclic reduction for the efficient solution of block tridiagonal matrices which arise from separable elliptic partial differential equations. It is however not well-suited for the present case of interest consisting of *dense* blocks. Other parallel block tridiagonal solvers [11] are also not optimized for these parameters.

ScaLAPACK [12,13] provides another method for efficiently solving dense block tridiagonal systems. Block factorization and solution based on ScaLAPACK are currently implemented in the SIESTA [2] MHD equilibrium code. This technique scales well with processor count only for very large matrix block sizes. For matrix blocks of interest for small 3D MHD problems ($M \sim 300$), scalability was found to be limited to about 5–10 processors. For larger block sizes, scalability is expected to improve. To improve scalability for the current (small) block sizes, we have developed the BCYCLIC code. It uses a combination cyclic reduction, for scalability in the block row dimension ($N \gg 1$), together with GotoBLAS for scalability in the block size, to achieve overall good scalability. Enhancements to use ScaLAPACK instead of LAPACK, while retaining the use of threaded BLAS (in the layer under LAPACK), are being developed. This will relieve single-node memory constraints, allowing the blocks to be larger than the memory of a single-node, and also permit idle processors to participate in later recursion levels of the cyclic reduction.

In addition to cyclic reduction, divide-and-conquer strategies have been used to parallelize the solution of tridiagonal equations [14]. As noted by Wang [15], this partition algorithm is unlikely to be more efficient than cyclic reduction unless $P \ll N$ (here, P is the number of MPI ranks, which would be nodes in the case of using OpenMP or GotoBLAS threads, but cores in the case of only MPI tasks). A parallel fast direct solver for block tridiagonal systems with separable matrices, based on divide-and-conquer, has been described in [16]. A parallel symmetric block tridiagonal divide-and-conquer algorithm was described in [17]. Divide-and-conquer has been used in conjunction with cyclic reduction by Lee [18] for the TORIC code [3].

The state-of-the art sparse direct solver SuperLU [19] can be used to perform the necessary inversion for the block matrix A. In Appendix A, the processor scaling of SuperLU is compared to BCYCLIC for typical block size ($M \sim 300$) and block rows ($N \sim 1000$) encountered in present day 3D MHD simulations. The PETSc [20] library provides a suite of parallel routines for solving partial differential equations. However, our present focus is on developing a highly-optimized parallel implementation for dense block-triadiagonal systems. Vendor-supplied libraries, such as CRAY Scientific Libraries [21] (CSL) and IBM Engineering and Scientific Subroutine Library [22] (ESSL), are commonly used for matrix inversion operations but are also not specifically tuned for the present problem.

1.3. New contributions

BCYCLIC uses cyclic reduction to obtain good processor scalability with respect to the number of block rows N. It is also optimized for speed in a parallel environment by using efficient non-blocking receives and sends to minimize inter-processor communication time. In this way, the overall computation time is not dominated by communication. In contrast to previous codes (and algorithms), BCYCLIC stores the necessary factored blocks of A with minimal fill-in so that solutions with multiple right sides can be efficiently computed, even when they are not known at factorization time. This is important, for example, when the matrix A is used repeatedly as a preconditioner as part of a nonlinear iterative scheme, as in the SIESTA [2] code. By using multithreading techniques to accelerate the computation of matrix–matrix products, parallelism is introduced to give additional scalability with respect to the block size M. This dual scalability – in both N and M – is unique to the present implementation. BCYCLIC can be used with arbitrary block rows N and MPI ranks P that are *not* restricted to powers of two as in the classical cyclic reduction algorithm.

1.4. Organization of paper

The rest of the paper is organized as follows. Our solution approach based on cyclic reduction is described in Section 2 with emphasis on efficient block storage and inter-processor communication optimization. Section 3 documents some additional implementation details, followed by Section 4 in which a performance study is presented for the solver executed on a parallel machine with multi-core nodes. The results are summarized in Section 5.

2. Our approach

In this section, we present our solution approach based on block cyclic reduction. First, we review the well-known Thomas sequential method, to place the current approach in context. This is followed by our parallelization scheme for the block cyclic reduction algorithm. This scheme when executed sequentially is nearly as efficient as the Thomas algorithm in both computational FLOPS (floating point operations) and memory usage, but requires (approximately) three times as many matrix–matrix products. The computational cost makes it slower than Thomas by about a factor of 2–3 on a serial machine. However, we later show that the real gain is that it gives a significant reduction in run time on a parallel machine (approximately $N/\log_2 N$), where it is assumed $P < N/2$, where, N is the number of block rows, and P is the number of processors over which the matrix block rows are distributed. This estimate assumes the code is structured so that communication times between processors do not dominate the run time. We show that this indeed can be achieved, and, in support of this observation, we present a detailed analysis of CPU time in a parallel execution environment.

2.1. Background: Thomas sequential recursion

The Thomas [4] algorithm is briefly reviewed for later comparison with the cyclic reduction method. The *i*th block row of the matrix *A* consists of the three $M \times M$ blocks denoted L_i, D_i, U_i, where L_i is the lower block, D_i is the diagonal block, and U_i is the upper block. At the boundaries, $L_1 = 0$ and $U_N = 0$. The block structure of *A* for the *i*th row is:

$$L_i x_{i-1} + D_i x_i + U_i x_{i+1} = b_i. \tag{1}$$

The Thomas algorithm uses the boundary conditions at $i = 1$ and $i = N$ to obtain a two-point recursion relation which can be iterated from one boundary to the other, followed by a "reverse sweep" to obtain the solution vector *x*. Generally, no block pivoting is done. First, x_1 is solved in terms of the unknown x_2, assuming D_1 is non-singular:

$$x_1 = D_1^{-1}(-U_1 x_2 + b_1). \tag{2}$$

Inserting this into the next block row equation, x_2 can be eliminated in terms of x_3. In general, the recurrence formula is:

$$x_i = -\Delta_i (U_i x_{i+1} + \beta_i). \tag{3a}$$

Here, Δ_i is a matrix and β_i is a vector. Inserting this into block row *i* yields the recurrence relations:

$$\begin{aligned} \Delta_i &= (D_i - L_i \Delta_{i-1} U_{i-1})^{-1}, \\ \beta_i &= b_i - L_i \beta_{i-1}. \end{aligned} \tag{3b}$$

Starting values for the recurrence relations in Eq. (3b) are $\Delta_0 = 0$ and $\beta_0 = 0$. Note this equation is intrinsically serial, since the computation at level *i* requires the previous level values.

The recurrence continues until the other boundary at $i = N$ is reached. There, $U_N = 0$ is used to initiate the back-solving process, via Eq. (3a):

$$x_N = -\Delta_N \beta_N. \tag{4}$$

Eq. (3a) is iterated *backwards* from $i = N - 1$ to $i = 1$ to complete the solution process.

Once Δ_i has been determined (and stored) for a particular matrix *A*, Eq. (3b) for β_i can be iterated forward for multiple right sides and the back-substitution in Eq. (3a) performed to obtain the unknowns, without any further matrix inversions or matrix–matrix products. Also, there is no fill-in, since D_i (and U_i) can be used to sequentially store Δ_i, (and $\Delta_i U_i$). Note the Thomas algorithm requires one matrix inversion and two matrix–matrix multiplications per block row.

2.2. Cyclic reduction of block rows

Cyclic reduction [6,9] is a "period doubling" algorithm that can be applied recursively to reduce the number of coupled block rows in Eq. (1) to one or a very small number compared with *N*. This reduced block system can then be either solved trivially – with a single block matrix inversion-or efficiently using the Thomas algorithm. Each step of the method can be performed in a parallel fashion. At each period doubling "bifurcation", the number of coupled equations to be solved is reduced by a factor of two and the spacing (in the original block row index space) of remaining block rows is doubled: hence the terminology "period doubling". Assuming that there are $P = N$ processors available, and inter-processor communication time is not a dominant factor, then the theoretical maximum speed-up factor due to cyclic reduction alone would be $R = N/\log_2 N$ on a parallel machine, compared to a single processor serial computation. The speed-up compared with the Thomas algorithm is not quite as large as this, since there are more matrix–matrix calculations per cyclic reduction step compared with Thomas (so *R* in practice might be only be in the range *R*/3 to *R*/2 compared to Thomas).

Following Ref. [9], note that cyclic reduction eliminates *all* the even-numbered block rows in Eq. (1) in terms of the odd-numbered rows. In Eq. (1), let $i = 2k$, for $k = 1, N/2$. For now, assume that $N \equiv 2^p$ is even. (This limitation will be relaxed later.) There are *p* + *1 levels*that correspond to *effective* decreasing system sizes (the number of remaining block rows to be processed) of $N, N/2, N/4, \ldots, N/2^p = 1$. At the μth level ($\mu = 0, \ldots, p - 1$), there are $N_\mu = 2^{p-\mu}$ rows of which half (the even ones) can be eliminated using Eq. (1), which is rewritten below for the *k*th even row:

$$x_{2k} = \hat{b}_{2k} - \widehat{L}_{2k} x_{2k-1} - \widehat{U}_{2k} x_{2k+1}, \tag{5a}$$

$$\begin{aligned} \hat{b}_{2k} &= D_{2k}^{-1} b_{2k}, \\ \widehat{L}_{2k} &= D_{2k}^{-1} L_{2k}, \\ \widehat{U}_{2k} &= D_{2k}^{-1} U_{2k}. \end{aligned} \tag{5b}$$

In Eq. (5a), $k = (1, \ldots, N_\mu/2)$. The boundary condition $\widehat{U}_{2N} = 0$ is implied by $U_{2N} = 0$.

Several features of Eq. (5) are noteworthy:

- *only* the inverses D_{2k}^{-1} for the *even* block row diagonal elements are required
- the D_{2k}^{-1} can be computed in *parallel* since at any level μ of the cyclic reduction, *all* the D_{2k} are known. Furthermore, there is no inter-processor communication required at this computation stage.

- the two matrix–matrix products needed to compute $\widehat{L}_{2k}$ and $\widehat{U}_{2k}$ can also be done in parallel, once the inverse blocks are computed.
- values of D_{2k}^{-1} (actually, its *LU* decomposition factors), $\widehat{L}_{2k}$, and $\widehat{U}_{2k}$ can overwrite the values of D_{2k}, U_{2k}, L_{2k}, respectively (since they are no longer needed for the back-substitution step) so that no fill-in occurs at this step.
- once the *odd* values of *x* are known (by a backwards recursion, which will be described below), then the *even* values are computed by efficient matrix–vector multiplications (no further expensive matrix inversions or matrix–matrix multiplications are required)

The number of computed inverses at the first reduction level ($\mu = 0$) is only half of the total Thomas inverses. When summed over *all* levels the number is the same as for the Thomas scheme. Likewise, total the number of matrix–matrix products for this step equals the number of Thomas products. Thus, in both time and memory utilization, the even part of the cyclic reduction equals the Thomas algorithm.

The next step in the cyclic reduction of Eq. (1) is to obtain an equation for the *odd*-indexed components of *x*, by letting $i = 2k - 1$, for $k = 1, N_\mu/2$, in Eq. (1) and using Eq. (5a) to eliminate the *even* components. The result is:

$$\widehat{L}_{2k-1}x_{2k-3} + \widehat{D}_{2k-1}x_{2k-1} + \widehat{U}_{2k-1}x_{2k+1} = \hat{b}_{2k-1}. \tag{6}$$

Here, the reduced matrix blocks (for *odd* indices now) are

$$\begin{aligned} \widehat{D}_{2k-1} &= D_{2k-1} - L_{2k-1}\widehat{U}_{2k-2} - U_{2k-1}\widehat{L}_{2k}, \\ \widehat{L}_{2k-1} &= -L_{2k-1}\widehat{L}_{2k-2}, \\ \widehat{U}_{2k-1} &= -U_{2k-1}\widehat{U}_{2k}, \end{aligned} \tag{7a}$$

and the reduced sources (for *odd* indices) are

$$\hat{b}_{2k-1} = b_{2k-1} - L_{2k-1}\hat{b}_{2k-2} - U_{2k-1}\hat{b}_{2k}. \tag{7b}$$

The boundary condition is $\widehat{L}_1 = 0$ (with $\widehat{U}_{2N-1} = 0$ a consequence $\widehat{U}_{2N} = 0$ as stated above). Note that no matrix inverses are required. Rather, four matrix–matrix products are needed to compute the *odd*-indexed reduced (hatted) matrix blocks. Interprocessor communication will now be required, since the calculation of the $(2k - 1)$ reduced matrices requires the *U* and *L* blocks from both the adjacent $2k - 2$ and $2k$ *even* rows. Communication can be minimized, if there are less than *N* processors, by storing even and odd block rows contiguously in memory on a given processor. Then only the "boundary" matrices from block rows on adjacent processors need to be communicated. While the effective *odd* diagonal elements can write over the original ones, we see from Eqs. (5a) and (7b) that both the original and reduced (hatted) *U* and *L* blocks must be retained for back-solving with multiple right sides. (The exception to this is when all the right sides are known at factorization time, which may *not* be the case when iterative matrix inversion is used as a preconditioner). So this reduction produces fill-in of one extra block per *level* (since only the *odd* values of the original blocks *U* and *L* must be stored), and an overall (summing over *all* levels) increase in storage of two blocks per row. Thus, the cyclic reduction increases the memory requirements compared to the Thomas algorithm by 66%, so the fill-in is still deemed to be manageable.

There are now four additional matrix–matrix products to evaluate in Eq. (7a). Added to the two required for the even reduction equations, this is a total of six compared with the Thomas algorithm's two. However, as noted previously, the number of matrix inversions is the same as for Thomas, so the overall numerical efficiency for each step of the cyclic reduction is greater than 1/3 that of Thomas. The parallelism of the cyclic algorithm compensates for this performance reduction.

Eq. (6) is self-similar to the original equation (1) but in the "period-doubled" index space *k* of *odd* indices. Setting $x_{2k-1} \equiv y_k$, for $k = 1, N_\mu/2$, with $\widehat{D}_{2k-1} \equiv \widetilde{D}_k$, etc., Eq. (6) becomes:

$$\tilde{L}_k y_{k-1} + \tilde{D}_k y_k + \tilde{U}_k y_{k+1} = \tilde{b}_k. \tag{8}$$

This self-similarity, in the period-doubled (odd) index space, allows one to apply recursively the same reduction technique to each successive level (cycle) of $N_{\mu+1} = N_\mu/2$ equations. In this way, one iterates through a period doubling sequence in which the number of equations is halved at each level: $N, N/2, N/4, \ldots, 1$. If at each level the communication time is dominated by the inversion and matrix–matrix multiplication times (implying an efficient use of parallel resources), then the time for the cyclic reduction will be *p* (in units of the time to compute a single inverse and matrix–matrix multiply step). This is compared with a time of *N* for only one processor. Thus, the theoretical parallel performance gain (ignoring communication time) is $R = N/p = N/ln_2N$, which is the scalability enhancement factor with respect to the number of block rows that can be expected from this algorithm. If the number of processors *P* is less than, or equal to, *N*, the performance gain will be reduced by approximately P/N, so $R = P/ln_2N$. More complete analysis of the parallel performance time is presented in Section 4.

The final level of the reduction process ($\mu = p$) results in a single diagonal block equation ($L = U = 0$) which can be trivially solved for the only remaining odd-indexed vector x_1. The solution $x_1 = \widehat{D}_1^{-1}\hat{b}_1$ is used to initiate back-substitution step.

Back-substitution at any level $\mu \leqslant p$ proceeds as follows. The even-indexed values are determined by substituting the *known* odd values into Eq. (5a), which requires two additional matrix–vector multiplications per even index. Note from Eqs. (6) and (8) that *all* indices at a *higher* level of the cyclic reduction originate from *only odd* indices at the (previous) level, since even-indexed rows do not propagate to the next level. Conversely, the even *and* odd indices at level μ uniquely populate *all* of the *odd* indices at the previous level $\mu - 1$. Therefore, once the solution at level μ is known, Eq. (5a) can be used to fill-in *all* the remaining even indices of level $\mu - 1$.

2.3. Parallel complexity of block cyclic reduction

In this performance analysis, both N (number of block rows) and P (number of processors) are assumed to be powers of two. Specifically, $N = 2^p$ and $P = 2^r$, for integers p and r. Execution time for non-powers of two for N can be bounded on the low-(and high-) side by using the nearest power of two that is less than (and greater than) N. A similar approach can be used for P that is not a power of two.

Recall that the algorithm first performs a series of forward "reduction" steps for each recursive bisection ("period doubling") until the sub-problem size has only one row block, and then unrolls the recursion via a corresponding series of backward "solve" steps. There are two zones within each of the reduction and solve phases as shown schematically in Fig. 2.

These zones correspond to $N' > P$ and $N' \leqslant P$, where N' is the number of remaining block rows at a particular level μ of the cyclic reduction. It is assumed that $N > P$ initially ($p > r$) so the algorithm begins in the first execution zone, where every processor contains two or more block rows to be processed. In general, in the first execution zone, each processor starts with $N/P = 2^{p-r}$ block rows and performs recursive bisections until each processor has exactly two row blocks.

The algorithmic complexity of the total elapsed time T_F in the forward reduction sweep, including both computation and communication is (definition of terms and derivation of which is provided in Appendix A.2):

$$T_F = (2^{p-r} + r - 1)\alpha M^3 + 2(p + r)\beta M^2 = (N/P - 1 + \log_2 P)\alpha M^3 + [2\log_2(NP)]\beta M^2. \quad (9)$$

For comparison, the serial Thomas algorithm computation time is

$$\begin{aligned} T_{\text{Thomas}} &= \alpha_{\text{Thomas}} N M^3, \\ \alpha_{\text{Thomas}} &= C_{inv} + 2C_{mul}. \end{aligned} \quad (10)$$

For a single processor ($P = 1$), the ratio of the cyclic reduction to Thomas algorithm times is $\alpha/\alpha_{\text{Thomas}} = (1 + 6C_{mul}/C_{inv})/(1 + 2C_{mul}/C_{inv})$. Since $C_{mul}/C_{inv} \sim 2.28$ on the SMOKY [23] computer (see Section 4.1), this computed ratio is 2.64 and is in close agreement with the observed value of 2.69. A minimum of three processors is required for the cyclic reduction algorithm to break-even with Thomas (see Fig. 3). For more processors, the parallel cyclic reduction algorithm is clearly prefer-

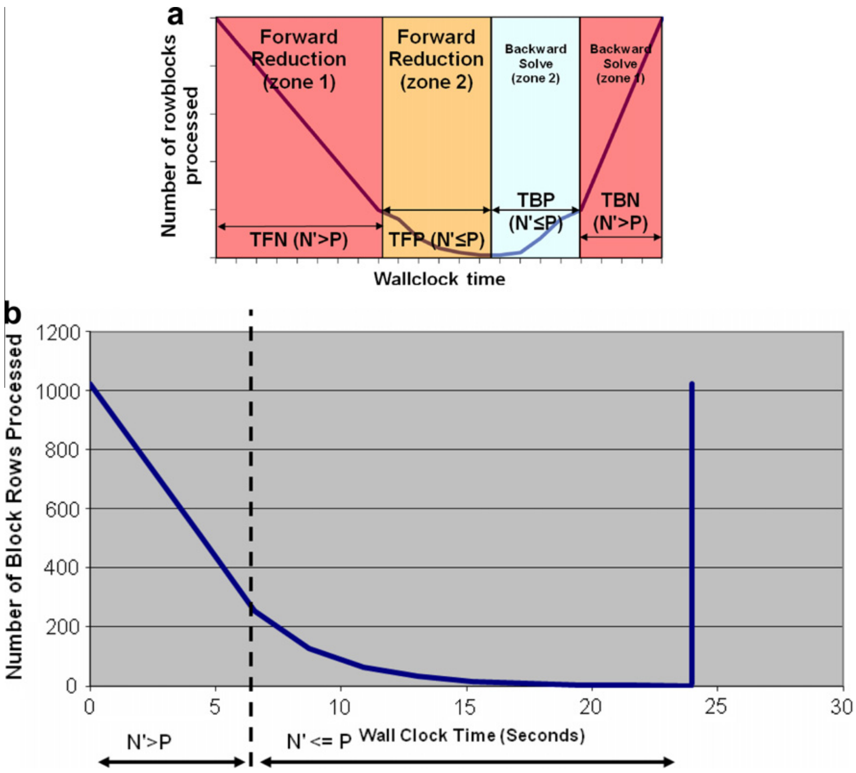

Fig. 2. (a) A schematic of the parallel work done over time. Schematic is not-to-scale, to show detail. (b) Actual (to-scale) parallel work done over time, for $N = 1024$, P-256, $M = 1024$.

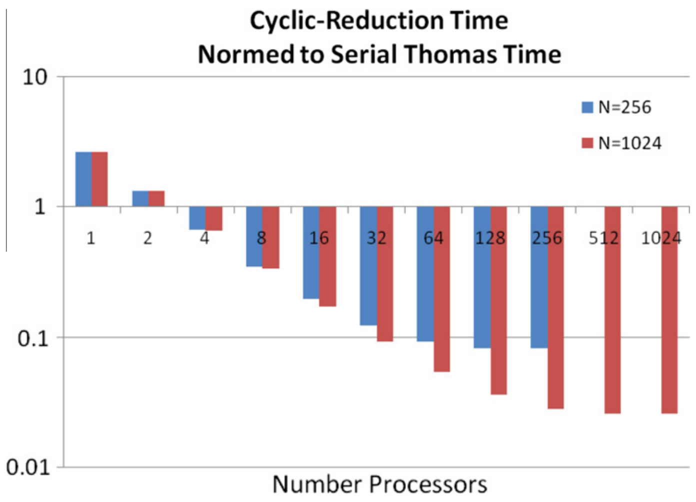

Fig. 3. Theoretical ratio of cyclic reduction time to serial Thomas time vs. number of processors, for small and moderate numbers of block rows, and ignoring inter-block communication ($M \to \infty$), based on Eqs. (9) and (10).

able. There is no performance gain for $P > N/2$. Any additional processors should be used to improve the factorization and matrix–matrix product calculations.

3. Additional implementation details

In this section we generalize the previous discussion to show how the cyclic reduction algorithm can be extended to the case when the number of processors P and/or the number of blocks rows N are not powers of two (it is still assumed $P \leqslant N$).

3.1. Distribution of block rows to processors

The distribution of block rows to processors is similar to the distribution for exact powers of two, with the simple modification that the first few processors each gets an extra row block mapped to it. Thus, the block rows are distributed to the processors according to the following rule (here, p refers to a specific processor):

$$\text{for } p = 1 \text{ to } p = mod(N,P) : \text{distribute } [N/P] + 1 \text{ block rows numbered consecutively,}$$
$$\text{for } p = 1 + mod(N,P) \text{ to } p = P : \text{distribute } [N/P] \text{ block rows numbered consecutively.}$$

Here, $[x] = floor(x)$ denotes the largest integer less than, or equal to, x.

3.2. Propagation of odd row indices between levels

According to the cyclic reduction at any level μ, the current even-numbered block rows are stored and no longer propagate to the next level $\mu + 1$. The odd rows propagate to the next level and map into the new consecutive block row numbering by the rule:

at level μ, odd row $n \Rightarrow$ row $(n + 1)/2$ at level $\mu + 1$

If N_μ is the number of block rows at level μ, then $N_{\mu+1} = [(N_\mu + l)/2]$. The inverse of this mapping is given by:

at level $\mu + 1$, row $n' \Rightarrow$ row $2n' - 1$ at level μ

Note that *all* indices at level $\mu + 1$ map to *only* the *odd* indices at the previous level μ. That is, in performing the back-substitution step at level μ to compute the even-index solutions according to Eq. (5a), one first maps the solution from level $\mu + 1$ to obtain *all* the odd values of x at the previous level μ. Then Eq. (5a) can be solved for the even-indexed $x's$ to complete the back-solve at that level. This back-solution is iterated until the complete solution at level $\mu = 0$ is obtained.

4. Performance study

4.1. Software and hardware platforms

To prepare for future multi-core computers with many cores per node, we have chosen a machine that contains a large number of cores per node (16 cores per node, 80 nodes). We implemented BCYCLIC using Fortran 90 and optimized it on the SMOKY [23] computer, which is a machine hosted by the National Center for Computational Sciences (NCCS) as a developmental computational resource at Oak Ridge National Laboratory (ORNL). SMOKY's current configuration is an 80 node Linux cluster consisting of four quad-core 2.0 GHz AMD Opteron processors per node, 32 GB of memory (2 GB per core), a gigabit Ethernet network with Infiniband interconnect, and access to Spider, the center wide LUSTRE-based file system.

Two levels of parallelism are used for the solver: (1) cyclic reduction is computed in parallel over the N block rows using separate MPI tasks; (2) GotoBLAS [24] (or OpenMP [25]) threads are used inside nodes (16 cores) to achieve parallelism in the factorization of the large ($M \times M$) dense blocks. The first level of parallelism has been described in the previous sections. The second level of parallelism involves the efficient calculation of matrix–matrix products and inverses. Note the four matrix–matrix products in Eq. (7a) are performed by the LAPACK routine DGEMM. This is the single most CPU-intensive routine, consuming over 50% of the time at each cyclic reduction level. Optimization can be achieved by doing this part of the calculation in parallel when separate threads are available. (The remaining time is spent in the matrix inversion and the two matrix–matrix multiplications in Eq. (5b).)

4.2. Threaded goto BLAS

On SMOKY, batch jobs can be submitted to use multiple threads. Thread usage is controlled by modifying the processors per node (ppn) count. For example, 1024 MPI tasks on 16 cores per node would be set by the command "nodes = 64: ppn = 16". Alternatively, for 64 MPI tasks, with 1 MPI task per node, the required command would be "nodes = 64: ppn = 1". In this latter case, sixteen threads can be assigned on each node. For OpenMP parallelism, "export OMP_NUM_THREADS = 16" would set 16 OpenMP threads per core. For GotoBLAS threads, "export GOTO_NUM_THREADS = 16" would set 16 GotoBLAS threads per core.

OpenMPI is the library used for MPI tasks on SMOKY. The MPI communication in BCYCLIC is done by non-blocking sends and receives of even-numbered row data overlapped with computation. The non-blocking receives are called at the beginning of the factorization routine and the non-blocking sends are called as soon as the required even-numbered block row is calculated instead of waiting for all the calculations to complete. For the cases studied, choosing the number of MPI tasks so there are initially four or more block rows per MPI task was optimal.

OpenMP directives were placed around the loops in BCYCLIC that call the time-intensive BLAS routines. However, for the same total number of processors, this results in essentially the same parallelism over the block rows that is done by MPI tasks alone. For example, calculating the factorization with 1024 MPI tasks takes approximately the same time as calculating with 64 MPI tasks using sixteen OpenMP threads each, since both methods parallelize the factorization over the number of block rows. Fig. 4 shows comparisons for different distributions of processors between MPI tasks and hybrid MPI tasks with OpenMP directives for several block sizes and numbers of block rows. The speed-up factors are very similar, but there are

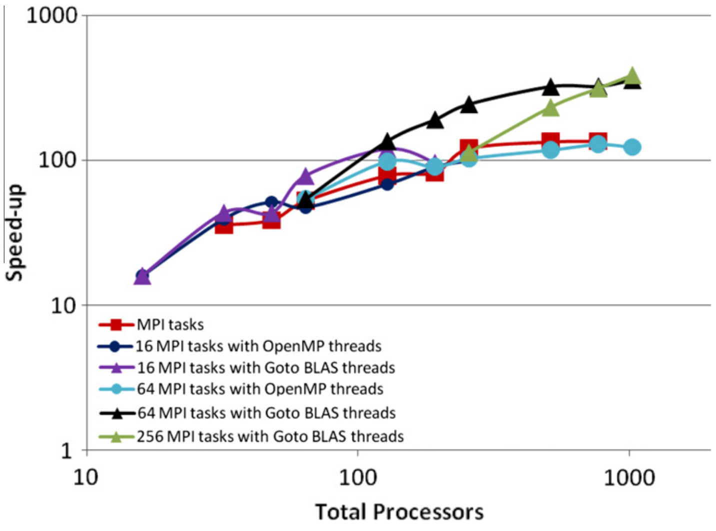

Fig. 4. Speed-up factor on SMOKY [23] vs. total number of processors for block size $M = 1000$ and block rows $N = 1000$.

advantages to using the hybrid method whenever possible. For instance, the hybrid jobs can use sixteen times more memory since the sixteen cores have shared memory. This implies that the linear block size M could be four times larger for the hybrid job. Another advantage appears whenever the communication time dominates the calculation. Then, the hybrid method would be faster since it requires less communication. In the future, as the number of cores per node is expected to increase, and threads will be able to spawn additional threads, we expect a combination of OpenMP threads spawning GotoBLAS threads will further increase the scalability of BCYCLIC calculations.

For parallelism within the blocks (factorization and matrix–matrix multiplication), the threaded GotoBLAS library was used. It contains both threaded Basic Linear Algebra Subprograms (BLAS) and threaded Linear Algebra Package (LAPACK) routines needed for cyclic reduction and has been built for SMOKY. The number of threads used by these routines is controlled with the GOTO_NUM_THREADS environmental variable. For the cyclic reduction factorization, the LAPACK routines, DGETRF and DGETRS, are used along with DGEMM. The cyclic reduction solver requires DGEMV from level two BLAS and DGETRS.

For large blocks ($M = 1000$) shown in Fig. 4, the optimal number of GotoBLAS threads is sixteen. Based on these observations, the optimal number of GotoBLAS threads is approximately the block size divided by 64. In all four cases, improved performance can be achieved using a combination of MPI tasks and GotoBLAS threads, compared with using only MPI tasks. Note that for 256 MPI tasks and four GotoBLAS threads (the green points in Fig. 4), the speed-up has not saturated but appears as though it would continue to increase if more threads would be available. Unfortunately, the processor limit of SMOKY is $80 \times 16 = 1280$, so it was not possible to verify this conjecture.

4.3. Performance comparison with SuperLU

SuperLU [19] is a state-of-the art library for solving sparse systems on high performance computers. The Distributed SuperLU version is designed to efficiently use MPI for communicating between processors. Here, the Distributed SuperLU version was compared to BCYCLIC using the function, pdgssvx_ABglobal, which replicates the matrices on all processors and should be faster than the distributed data version. First the matrix rows and columns are scaled to have unit norm. Then a row permutation is done to make the diagonal large relative to the off-diagonal elements and a column permutation to preserve the scarcity of the upper and lower factors. The system is solved directly by factoring the matrix and doing forward and back-substitutions and the solution is refined.

The block tridiagonal matrices were initialized with random values for blocks of size 273×273 in a tridiagonal system with 256 blocks for both the SuperLU and BCYCLIC for comparison. Fig. 5 shows the factorization time (the most time-consuming part of the calculation) comparison for this case. For a small number of cores the CPU time is very similar for SuperLU and BCYCLIC, but the scaling to large cores is much better for BCYCLIC. For this comparison, no GoTo BLAS threads were used (which would enhance the scalability of BCYCLIC even more).

4.4. Verification of parallel complexity

The time estimate in Eq. (13) is verified by first determining the values for α and β, and then executing the BCYCLIC code to compare the observed total communication and computation.

The following parameters were chosen for the numerical experiment: $N = 1024(=2^{p=10})$, $P = 256(=2^{r=8})$ and $M = 1024$. These represent a medium-range of the size of the problems of interest in 3D MHD equilibrium calculations [2].

The total computational time observed on the SMOKY cluster [23] for this problem size was 20.81 s, and the total time spent in communication was 15.03 s, giving a total runtime of 35.83.

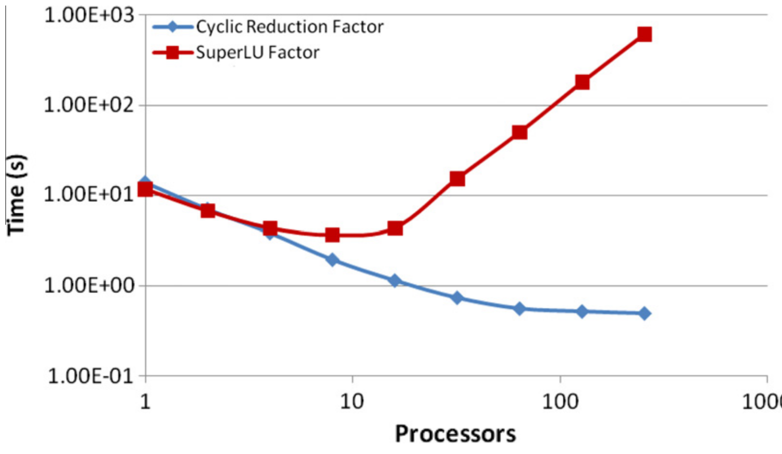

Fig. 5. Wall clock time for factoring a block tridiagonal matrix of size $M = 273$ and $N = 256$ for BCYCLIC and SuperLU.

The value for α can be empirically found by clocking the time it takes to compute one matrix multiplication and one matrix factorization of a 1024×1024 matrix. On SMOKY, for $M = 1024$, C_{mul} and C_{inv} were determined to equal 3.15×10^{-10} sec/element and 1.38×10^{-10} sec/element, respectively, giving $\alpha = 20.28 \times 10^{-10}$. This predicts the total computation time as $T_F^{\text{compute}} = (2^{10-8} + 8 - 1) \times 20.28 \times 10^{-10} \times 1024^3 = 23.95\,\text{sec}$, which is in good agreement with the observed time (20.81 s). The value for β can be estimated by dividing the byte size of one floating point value by the average bandwidth available per processor core. On SMOKY, each processor contains eight cores. Since each processor is connected to a 1 Gbps Ethernet link, each "endpoint" core obtains a proportional share of the total off-processor bandwidth. On average, the bandwidth may thus be estimated to be around 128 Mbps. Each floating point value takes 8 bytes, which is 64 bits, giving $\beta = 64/(128 \times 10^6) = 0.5 \times 10^{-6}$ sec/value. This predicts the total communication as $T_F^{\text{comm}} = 2(10 + 8) \times 0.5 \times 10^{-6} \times 1024^2 = 18.87\,\text{sec}$.

The predicted total runtime is $T_F = 42.82$ s, which matches the actual observed runtime of 35.83 s within the margin of error due to system artifacts such as variable messaging delays, operating system jitter, and cache-sensitive linear algebra routine run times.

5. Summary

A new implementation of a parallel algorithm for solving linear, block tridiagonal problems with multiple right-hand sides has been described. The algorithm is based on block cyclic reduction, which delivers a theoretical speed-up of $N/(3\log_2 N)$ with respect to the serial Thomas algorithm and a fill-in factor of 5/3. The implementation presented in this paper has been extended to deal with arbitrary numbers of blocks and processors (powers of two are required by the usual cyclic reduction technique). It recovers this theoretical enhancement – within a factor of two due to communication overhead-by using MPI-based parallelism over the block rows. Further parallelism is achieved by using OpenMP or GotoBLAS threads to perform the computationally-intensive matrix inversion and multiplication operations that must be done on individual blocks. This mixed programming model has been found to deliver better performance compared with MPI alone and is suitable for modern multi-core supercomputers. It is expected that this new algorithm will provide substantial advantages for the many codes which need to solve linear problems with dense, tridiagonal blocks either directly or as an intermediate step to provide a suitable preconditioner for nonlinear problems. Physical problems in which block tridiagonal matrices with many ($N \gg 1$) and large ($M \gg 1$) blocks appear are found in the simulation of toroidal plasmas confined by a magnetic field, such as in tokamak or stellarator, due to the double periodicity of the toroidal configuration. Our scheme will permit block sizes up to about $M \sim 40{,}000$ at double-precision on machines with up to 32 GB per node (like SMOKY). This can be considered to represent a large-sized for MHD equilibrium simulations.

Acknowledgments

This research has been sponsored by the US Department of Energy under Contract DE-AC05-00OR22725 with UT-Battelle, LLC.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

The authors are grateful to Dr. L. Chacon and Dr. S. Seal for thoughtful reviews of the manuscript.

Appendix A

A.1. Schematic illustration of the cyclic reduction algorithm in a parallel environment

We will now illustrate how the algorithm proceeds in multiple, alternating rounds of computation and communication. Fig. 6 is a schematic execution trace of the cyclic reduction algorithm with $N = 32 = 2^5$ block rows executing on $P = 4(2^2)$ processors. The blocks are distributed contiguously in memory and evenly to all processors. The processor number is shown in the first column. Computation and communication corresponding to row blocks are represented by rows in the table. Iterations of the algorithm progress left to right in columns. Each cell in the table, hence, represents one level (step) of the algorithm on that element. Green cells represent computation of terms in Eq. (5b) by even-numbered row blocks, and yellow cells denote communication of the results to the adjacent odd-numbered row blocks at that level. Blue cells represent the computation of modified (hatted) constants of Eq. (7b). Red cells represent the solve operation of Eq. (5a), and pink cells show the communication of solution vectors to adjacent even-numbered rows of the previous level.

The inter-processor communication of matrix blocks is shown by yellow-colored cells. Note that only the row blocks at the boundaries of each processor send and receive the data; the row blocks "inside" a single processor do not need to explicitly exchange data via inter-processor communication, since the data is available locally. Near the bottom of the chart, the row marked "Busy" indicates the number of processors that are actively engaged in computation at any given level. It is clear that all processors are active until after the level at which only one row block remains to be processed per processor.

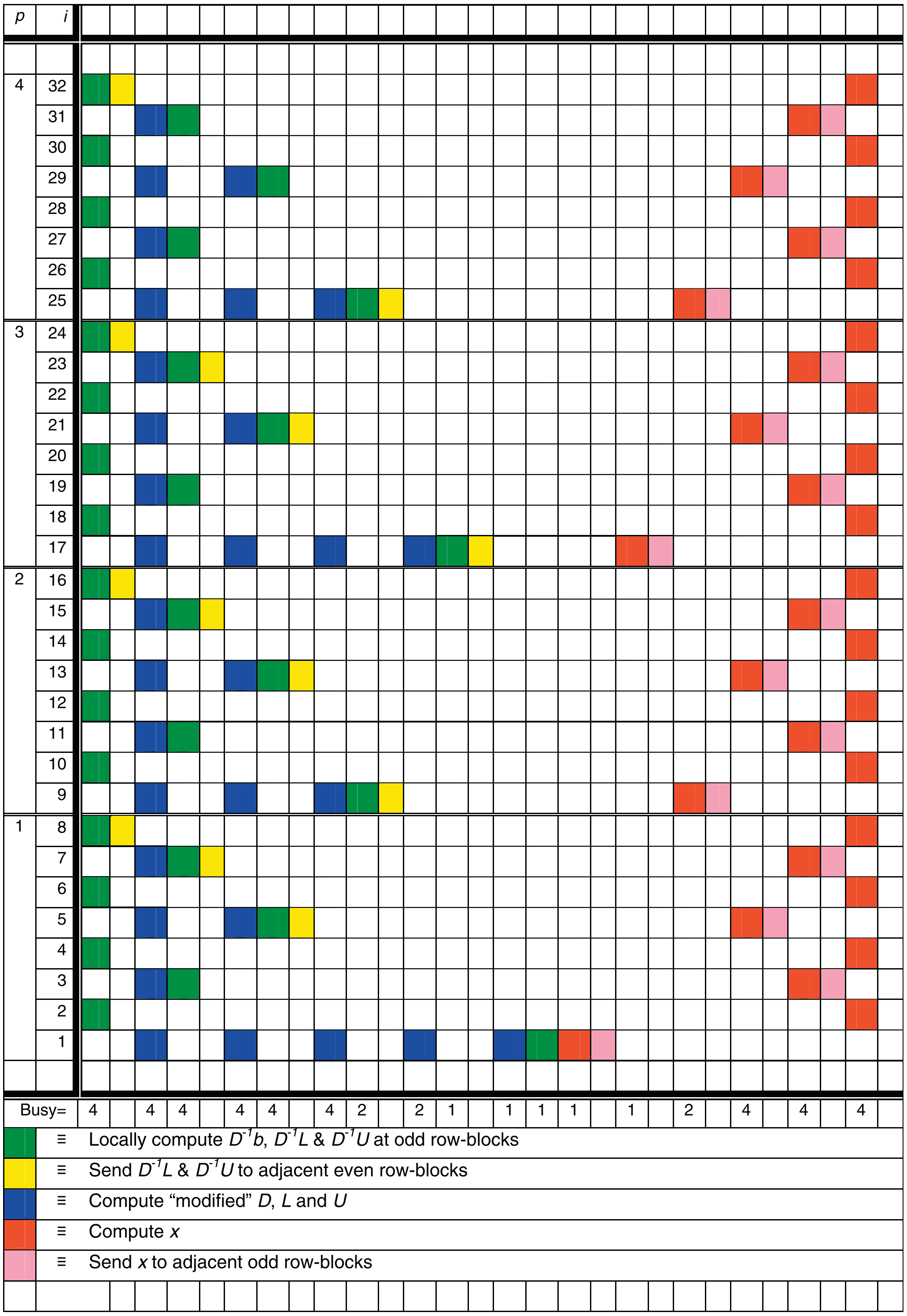

Fig. 6. Execution trace of a 32 row block problem with four processors.

A.2. Parallel complexity analysis

Here, additional detail is provided on the analysis of the parallel computation time, continuing from the discussion in Section 2.3.

Due to recursive bisection, even when starting with $p > r$, the algorithm will eventually reach a level for which $p \leqslant r$, which is the "second zone" of execution. Here, we focus on the reduction portion of the algorithm, since that is the most time-consuming portion. Each reduction step involves matrix–matrix multiplication and matrix inversion operations, which are computed in $O(M^3)$ time. Each solve step involves matrix–vector multiplication operations, which are computed in $O(M^2)$ time. To analyze CPU usage, we introduce the following definitions:

$$T_{FN}^{\text{compute}} \equiv \text{Computation time in the } (N > P) \text{ reduction zone},$$
$$T_{FN}^{\text{comm}} \equiv \text{Communication time in the } (N > P) \text{ reduction zone},$$
$$T_{FP}^{\text{compute}} \equiv \text{Computation time in the } (N \leqslant P) \text{ reduction zone},$$
$$T_{FP}^{\text{comm}} \equiv \text{Communication time in the } (N \leqslant P) \text{ reduction zone}.$$

The $N > P$ zone proceeds in $p - r$ steps. At level μ, $0 \leqslant \mu < p - r$, every processor processes $2^{p-r-\mu} > 1$ block rows. For each *even* block (Eq. (5b)), one matrix inversion, two matrix–matrix products, and one matrix–vector product are performed. For each *odd* block (Eq. (7a)), four matrix–matrix products and two matrix–vector products are performed. At the end of even row processing at each step, the modified L, U and b terms of the even rows are communicated to the neighboring processor, which involves sending two $M \times M$ matrices, and a M vector.

Let us assume perfectly concurrent execution by all processors. Let us also assume a matrix multiplication algorithm that multiplies two $M \times M$ matrices using $C_{mul}M^3$ floating point operations, and Gaussian elimination for matrix inversion which takes $C_{inv}M^3$ floating point operations for an $M \times M$ matrix. The matrix–vector product operation takes $C_{vmul}M^2$ operations. The constants C_{mul}, C_{inv} and C_{vmul} depend on the algorithm used as well as the central processing unit(s) available on the computing platform, and can be easily determined by calculations for different sizes of M.

For step μ, the time consumed for even rows is given by:

$$T_{even}^{\mu} = 2^{p-r-\mu-1}\left[C_{inv}M^3 + 2C_{mul}M^3 + C_{vmul}M^2\right]. \tag{11a}$$

The time for odd rows is given by:

$$T_{odd}^{\mu} = 2^{p-r-\mu-1}\left[4C_{mul}M^3 + 2C_{vmul}M^2\right]. \tag{11b}$$

The total computation time for all rows at step μ is:

$$T_{comp}^{\mu} = T_{even}^{\mu} + T_{odd}^{\mu} = 2^{p-r-\mu-1}\left[(C_{inv} + 6C_{mul})M^3 + 3C_{vmul}M^2\right] \approx 2^{p-r-\mu-1}\alpha M^3 (M \gg 1), \tag{11c}$$

where $\alpha = C_{inv} + 6C_{mul}$.

At the end of even row computation of step μ, the time to communicate the two $M \times M$ matrices in parallel is given by:

$$T_{comm}^{\mu} = \beta \cdot 2M^2, \tag{11e}$$

for some constant "communication factor" β that measures the average time to transmit one floating point number.

The total computation and communication times for the first zone ($N > P$) of the forward reduction are given by summing the times at all levels 0 through $p - r - 1$:

$$\begin{aligned} T_{FN}^{\text{compute}} &= \sum_{\mu=0}^{p-r-1} 2^{p-r-\mu-1}\alpha M^3 = (2^{p-r} - 1)\alpha M^3, \\ T_{FN}^{\text{comm}} &= \sum_{\mu=0}^{p-r-1} \beta \cdot 2M^2 = 2\beta(p-r)M^2. \end{aligned} \tag{12}$$

In the second zone ($N \leqslant P$), each step involves processing of at most one row block per processor, each involving matrix operations similar to the one for the first zone. In communication, however, four matrices are sent to neighbors (two each to top and bottom), unlike the two matrices sent to the top neighbor alone in the first zone. Thus, the total times for the second zone of the forward solve (ignoring M^2 terms in the computation time) are:

$$\begin{aligned} T_{FP}^{\text{compute}} &= \sum_{\mu=p-r}^{p} \alpha M^3 = r\alpha M^3, \\ T_{FP}^{\text{comm}} &= \sum_{\mu=p-r}^{p} \beta \cdot 4M^2 = 4\beta r M^2. \end{aligned} \tag{13}$$

The total execution time for the entire forward solve is given by adding Eqs. (12) and (13) for the two zones:

$$\begin{aligned} T_{F}^{\text{compute}} &= (2^{p-r} + r - 1)\alpha M^3, \\ T_{F}^{\text{comm}} &= 2(p+r)\beta M^2. \end{aligned} \tag{14}$$

Thus, the algorithmic complexity of the total elapsed time including computation and communication is:

$$T_F = (2^{p-r} + r - 1)\alpha M^3 + 2(p+r)\beta M^2 = (N/P - 1 + \log_2 P)\alpha M^3 + [2\log_2(NP)]\beta M^2. \tag{15}$$

For comparison, the serial Thomas algorithm computation time is

$$\begin{aligned} T_{\text{Thomas}} &= \alpha_{\text{Thomas}} N M^3, \\ \alpha_{\text{Thomas}} &= C_{inv} + 2C_{mul}. \end{aligned} \tag{16}$$

References

[1] S.P. Hirshman, W.I. van Rij, P. Merkel, Comput. Phys. Commun. 43 (1986) 143.
[2] S.P. Hirshman, R. Sanchez, V.E. Lynch, E.F. D'Azevedo, J.C. Hill, SIESTA: an scalable iterative equilibrium solver for toroidal applications, in: Proceedings of the 35th European Physical Society Conference on Plasma Physics, vol. 32, Hersonissos, 9–13 June, 2008, p. 1044.
[3] M. Brambilla, Quasilinear wave equations in toroidal geometry with applications to fast wave propagation and absorption at high harmonics of the ion cyclotron frequency, Plasma Phys. Controlled Fusion 45 (2005) 1411.
[4] L.H. Thomas, Elliptic problems in linear difference equations over a network, Watson Sci. Comput. Lab. Rept., Columbia University, New York, 1949.
[5] A.C. Hindmarsh, Solution of Block-Tridiagonal Systems of Algebraic Equations, UCID-30150, Lawrence Livermore Laboratory, Livermore, California, 1977.
[6] R.A. Sweet, SIAM J. Numer. Anal. 14 (1977) 706–719.
[7] W. Gander, G.H. Golub, in: Proceedings of the Workshop on Scientific Computing, Hong Kong, 10–12 March, 1997.
[8] H.S. Stone, Parallel tridiagonal equation solvers, ACM Trans. Math. Softw. 1 (1975) 289.
[9] D. Heller, Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, SIAM J. Numer. Anal. 13 (4) (1976) 484–496.
[10] P.N. Swarztrauber, R.A. Sweet, Efficient Fortran subprograms for the solution of elliptic equations, ACM Trans. Math. Softw. 5 (1979) 352–364.
[11] P. Garaud, J.-Didier Garaud, A Friendly and Free Parallel Block-Tridiagonal Solver: User Manual. <http://www.soe.ucsc.edu/research/report?ID=487>.
[12] ScaLAPACK Users' Guide, SIAM, 1997, ISBN: 0-89871-397-8.
[13] S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. Whaley, ScaLAPACK: a linear algebra library for message-passing computers, in: Proceeding of 1997 SIAM Conference on Parallel Processing, May 1997.
[14] A. Cleary, J. Dongarra, Implementation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems, Center for Research on Parallel Computation, Technical Report No. CRPC-TR97717, 1997.
[15] H.H. Wang, ACM Trans. Math. Softw. 7 (2) (1981) 170–183.
[16] T. Rossi, J. Toivanen, A Parallel Fast Direct Solver for Block Tridiagonal Systems with Separable Matrices of Arbitrary Dimension, University of Jyvaskyla, Department of Mathematics, Report 21, 1996.
[17] Y. Bai, R.C. Ward, A Parallel Symmetric Block-Tridiagonal Divide-and-Conquer Algorithm, Technical Report UT-CS-05-571, 2005. <http://www.cs.utk.edu/~library/2005.html>.
[18] Jungpyo Lee, The Parallelization of a Block-Tridiagonal Matrix System for an Electromagnetic Wave Simulation in TOKAMAK(TORIC) by MPI Fortran, MIT Department of Nuclear Engineering, Final Report-18.337, 2009.
[19] S. Li Xiaoye, James W. Demmel, SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Trans. Math. Softw. 29 (2) (2003) 110–140.
[20] Satish Balay, William D. Gropp, Lois Curfman McInnes, Barry F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A.M. Bruase, H.P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, p. 163, 1997.
[21] SGI/CRAY Scientific Computing Software Library. <http://www.sgi.com/products/software/irix/scsl.html>.
[22] IBM ESSL Manual. <http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.essl.doc/esslbooks.html>.
[23] SMOKY, Oak Ridge National Laboratory Center for Computational Science. <http://www.nccs.gov/computing-resources/smoky/>.
[24] <http://www.tacc.utexas.edu/resources/software/gotoblasfaq.php>.
[25] Barbara Chapman, Gabriele Jost, Ruud van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, 2007.