Revisiting Cyclic Reduction and Parallel Prefix-Based Algorithms for Tridiagonal Systems of Equations

Sudip K. Seal[1], Kalyan S. Perumalla[1], Steven P. Hirshman[2]

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Abstract

Direct solvers based on *prefix computation* and *cyclic reduction* algorithms exploit the special structure of tridiagonal systems of equations to deliver better parallel performance compared to those designed for more general systems of equations. This performance advantage is even more pronounced for block tridiagonal systems. Complexity analyses of both algorithms based on the problem size and the number of processors alone are inadequate to capture the effect of block sizes on their relative runtimes. This paper re-examines these algorithms taking the effects of block size into account. Depending on the block size, the parameter space spanned by the number of block rows, the block size and the processor count is shown to favor one or the other of the two algorithms. A critical block size that separates these two regions is shown to exist and its dependence both on problem dependent parameters and on machine-specific constants is established. Empirical verification of these analytical findings are carried out on up to 2,048 cores of a Cray XT4 system.

Keywords: block tridiagonal matrix, cyclic reduction, prefix computation, parallel solver

1. Introduction

A matrix equation of the form $Ax = b$ in which x and b are vectors of length N and A is an $N \times N$ matrix whose only non-zero elements are those along its three central diagonals is referred to as a tridiagonal system of equations. They arise naturally in many important scientific applications and a number of fast solvers for such systems have been developed over the years. In a more generalized variant, called a block tridiagonal system, the matrix A consists of an $N \times N$ array of blocks where each block is an $M \times M$ array of numbers and the elements not belonging to its three central block diagonals are all zeros. For generality, we assume blocks can be dense. Fast solvers become especially critical to runtime performance when such dense block tridiagonal systems need to be solved multiple number of times corresponding to changing A or b during the simulation of a physical process.

A variety of parallel tridiagonal solvers is available, either as part of larger linear algebra packages or standalone [1, 2, 3, 4, 5, 6]. These state-of-the-art solvers are highly effective on matrix structures such as general dense or sparse matrices. Customized solvers based on algorithms that take advantage of the tridiagonal structure are, however, known to deliver superior runtime performance and scalability. A better understanding of the performance gains from customized block tridiagonal solvers is motivated by the need for fast and scalable parallel solvers for dense block tridiagonal systems of equations that form a critical computational core in multiple application domains. This scaling need is more pronounced in the context of the computing capabilities of today's large state-of-the-art computing platforms.

Two classes of algorithms for block tridiagonal systems of equations, based on cyclic reduction and parallel prefix computations, respectively, are particularly amenable to efficient and scalable parallelization. Both require $O(\lg N)$

Email addresses: sealsk@ornl.gov (Sudip K. Seal), perumallaks@ornl.gov (Kalyan S. Perumalla), hirshmansp@ornl.gov (Steven P. Hirshman)

[1]Computational Sciences and Engineering Division

[2]Fusion Energy Division

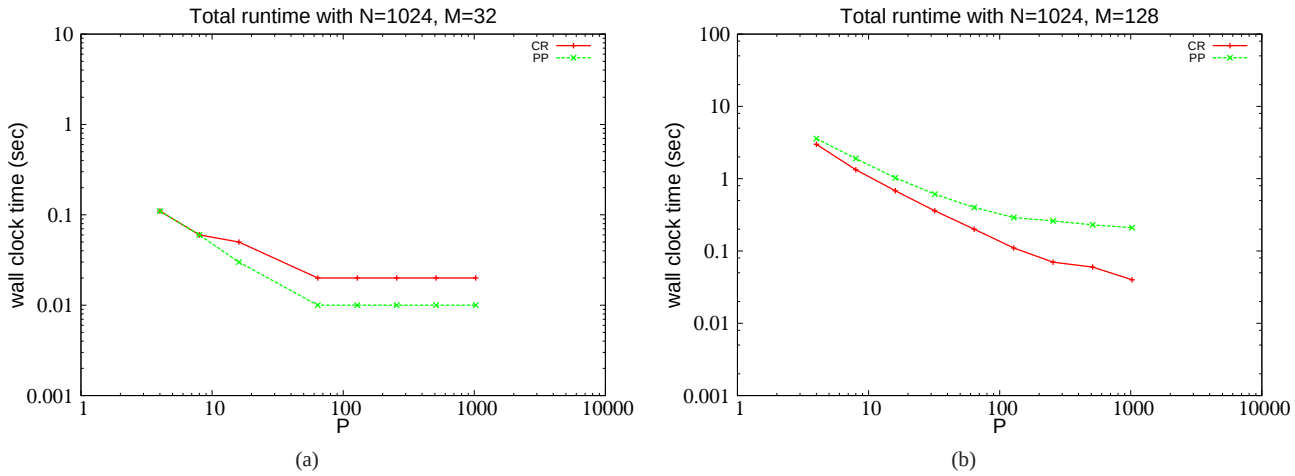


Figure 1: Performance of a parallel prefix (PP) based solver and a cyclic reduction (CR) based solver on a log-log scale, with block rows, $N = 1024$, and block size: (a) $M = 32$ and (b) $M = 128$.

steps. Complexity analysis of both algorithms for a system with N row blocks on P processors suggests a better runtime for parallel prefix-based solvers. This is found to be true for smaller block sizes. Interestingly, the opposite is found to be true when block sizes are large. An illustration of this behavior is shown in Figure 1 in which the parallel prefix-based solver outperforms the cyclic reduction-based algorithm when $M = 32$ and vice versa when $M = 128$. Performance analyses of the two algorithms that allows users to make the right choice of a parallel solver for block tridiagonal systems based on the parameters N, P and M, when $M > 1$, is lacking in the literature. The findings in this paper fill this gap.

The Thomas algorithm [7] is one of the first sequential algorithms to exploit the special structure of tridiagonal systems of equations. But, its inherently serial nature precluded it from any practical parallel implementation. Algorithms based on divide-and-conquer approaches were subsequently introduced in [8, 9, 10, 11]. These formed the foundations of a large body of research [12] on parallel solvers for tridiagonal systems of equations. Most closely related to this paper is the work in [13] which provides tight bounds for both cyclic reduction and parallel prefix-based algorithms. However, to the best of our knowledge, no prior work reports the effect of block sizes on the relative performances of direct solvers based on these two algorithms.

2. Formulations

For an $N \times N$ block tridiagonal matrix with block size M, let L_i, D_i and U_i denote the lower, main and upper diagonal blocks, respectively, in block row i. Using this notation, the i^{th} row of the block tridiagonal matrix can be written as:

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

$$D_1 x_1 + U_1 x_2 = b_1 \tag{2}$$

$$L_N x_{N-1} + D_N x_N = b_N \tag{3}$$

where b_i is the i^{th} block of the right hand side vector.

2.1. Cyclic Reduction

In a cyclic reduction-based approach, the boundary conditions on the block matrices are set to $L_1 = U_N = 0$. For the even indices $i = 2k$ ($i \leq k \leq N/2$), Eqn (1) yields:

$$x_{2k} = \tilde{b}_{2k} - \tilde{L}_{2k} x_{2k-1} - \tilde{U}_{2k} x_{2k-1} \tag{4}$$

where

$$\tilde{b}_{2k} = D_{2k}^{-1} b_{2k}\,,\ \tilde{L}_{2k} = D_{2k}^{-1} L_{2k}\,,\ \tilde{U}_{2k} = D_{2k}^{-1} U_{2k}$$

A similar equation for the odd indices $i = 2k-1$ ($i \le k \le N/2$) can be written and then Eqn (4) used to eliminate the even indexed terms. This yields:

$$\tilde{L}_{2k-1} x_{2k-3} + \tilde{D}_{2k-1} x_{2k-1} + \tilde{U}_{2k-1} x_{2k+1} = \tilde{b}_{2k-1} \tag{5}$$

where:

$$\begin{aligned}
\tilde{D}_{2k-1} &= D_{2k-1} - L_{2k-1}\tilde{U}_{2k-2} - U_{2k-1}\tilde{L}_{2k} \\
\tilde{L}_{2k-1} &= -L_{2k-1}\tilde{L}_{2k-2} \\
\tilde{U}_{2k} &= -U_{2k-1}\tilde{U}_{2k} \\
\tilde{b}_{2k-1} &= b_{2k-1} - L_{2k-1}\tilde{b}_{2k-2} - U_{2k}\tilde{b}_{2k}
\end{aligned}$$

Note that Eqn (5) is "similar" to Eqn (1) but the number of equations that need to be solved for has been reduced by half. This step is recursively applied until only a single equation remains which is solved for x_1 using the appropriate boundary conditions.

This solution is used to initiate a backward solve phase in which the recursion tree is traversed in the reverse direction. At each recursive step during this backward traversal, the even indexed unknowns are computed by substituting the now known odd-indexed values.

2.2. Prefix Product

In a prefix formulation, $L_1 = U_N = I$ and $x_0 = x_{N+1} = 0$ at the boundaries. This makes Eqn (1) valid for all $1 \le i \le N$ and it can be written as:

$$x_{i+1} = -U_i^{-1} D_i x_i - U_i^{-1} L_i x_{i-1} + U_i^{-1} b_i$$

assuming that U_i is non-singular for all $1 \le i \le N$. In matrix form, this can be rewritten as:

$$\begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} = \begin{bmatrix} -U_i^{-1}D_i & -U_i^{-1}L_i & U_i^{-1}b_i \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} \tag{6}$$

Let:

$$Y_{i+1} = \begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix},\ B_i = \begin{bmatrix} \tilde{D}_i & \tilde{L}_i & \tilde{b}_i \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix},\quad \begin{aligned} \tilde{D}_i &= -U_i^{-1}D_i \\ \tilde{L}_i &= -U_i^{-1}L_i \\ \tilde{b}_i &= U_i^{-1}b_i \end{aligned} \tag{7}$$

Using Eqn (7), the matrix equation in Eqn (6) can be rewritten as:

$$Y_{i+1} = B_i Y_i = B_i B_{i-1} Y_{i-1} = \cdots = B_i B_{i-1} B_{i-2} \cdots B_1 Y_1 = S_i Y_1 \text{ where } S_i = B_i B_{i-1} B_{i-2} \cdots B_1 \tag{8}$$

Note that the partial matrix-matrix products S_i's can be evaluated using a parallel prefix computation. The partial results Y_{i+1} can, therefore, be computed using the partial matrix product S_i and Y_1. Thus, the latter must be made available at each processor to realize the final solution efficiently. For this, a parallel prefix algorithm in which each processor has both the total as well as the partial prefix products at the end of the prefix computation is used so that each processor has the total prefix product, $S_N = B_N B_{N-1} B_{N-2} \cdots B_1$, in addition to the local S_i's. From Eqn (8), it follows that:

$$Y_{N+1} = S_N Y_1 \Rightarrow \begin{bmatrix} x_{N+1} \\ x_N \\ 1 \end{bmatrix} = \begin{bmatrix} S_N^{11} & S_N^{12} & S_N^{13} \\ S_N^{21} & S_N^{22} & S_N^{23} \\ 0 & 0 & 1 \end{bmatrix}_N \begin{bmatrix} x_1 \\ x_0 \\ 1 \end{bmatrix}$$

Using the boundary conditions $x_0 = x_{N+1} = 0$ yields:

$$x_{N+1} = S_N^{11} x_1 + S_N^{12} x_0 + S_N^{13} \Rightarrow x_1 = -[S_N^{11}]^{-1} S_N^{13} \tag{9}$$

Thus:

$$Y_1 \quad = \quad \begin{bmatrix} x_1 & \hat{0} & 1 \end{bmatrix}^T \qquad \text{where } \hat{0} \text{ is a } M \times 1 \text{ zero-vector.}$$

At the end of the prefix sum, each processor already has S_N and, hence, S_N^{11} and S_N^{13}, so that each processor can independently compute Y_1 using Eqn (9) and then the X_i's (hence, x_i's) locally for all i's that are mapped to that processor.

3. Algorithms

For ease of presentation, we will assume that $N = 2^n$ and $P = 2^q$, for some non-negative integers n and q with the understanding that both algorithms generalize to N and P values that are non-powers of two. We will also assume that $N \geq P$ (or $n \geq q$). In both algorithms, $\frac{N}{P}$ block rows are assigned to each processor initially.

3.1. Cyclic Reduction

Forward Phase I: This phase consists of the first $\lg(\frac{N}{P}) = n - q$ recursive steps of the algorithm. In each step, the even indexed unknowns are eliminated in terms of the odd indexed ones using Eqn (4). At any intermediate step i, where $1 \leq i \leq (n - q)$, there are $N/2^{i+1}$ reordered odd-indexed rows and as many even-indexed row. For each reordered even row $2k$ in step i, $\tilde{b}_{2k}$, $\tilde{L}_{2k}$ and $\tilde{U}_{2k}$ are computed (for suitable chosen k within bounds). $\tilde{L}_{2k}$ and $\tilde{U}_{2k}$ at the boundary rows are then sent to neighboring processsors. For each reordered odd row $2k - 1$ in step i, $\tilde{b}_{2k-1}$, $\tilde{D}_{2k-1}$, $\tilde{L}_{2k-1}$ and $\tilde{U}_{2k-1}$ are then computed (for suitably chosen k within bounds). At the end of $n - q$ recursive steps, each processor contains exactly one row block and the algorithm enters its next phase.
Forward Phase II: Each step of this phase is identical to the ones in the previous phase with the modification that the logical reordering of the remaining rows after the completion of each step spans across processor boundaries. Due to the recursive bisection of the problem, only half the number of processors remain active compared to the previous step until at the end only one processor remains active and $(P - 1)$ remain idle. The computations remain the same but the communication pattern and load balance differ. In this phase, each step processes at most one block row per processor with matrix operations similar to the previous phase. However, four matrices are sent to neighbors (two each to top and bottom). This phase ends when finally there is only one active processor.
Backward Phases I and II: At the end of the previous phase, a local portion of x is available at processor 0. This value is then propagated to processor $P/2 - 1$ which in turn uses it for back substitution to evaluate its local portion of x. The newly computed values of x are then similarly propagated by traversing the recursion tree in the backward direction in the reverse direction from the bottom most level to the topmost level). Once the back substitution traverses $\lg P$ levels form the leaf level of the recursion tree, the algorithm enters its compute intensive communication efficient final phase. In this phase, the remaining $\lg(\frac{N}{P})$ levels are traversed in a direction opposite to that in forward phase I, all the while computing new values at each level by back-substituting with values computed in the previous level. Communication is required only for the boundary blocks. Finally, when the top level is reached, the solution of the system of equations is realized.
Total Runtime: Runtime T_{cr}, of this cyclic reduction-based tridiagonal solver can be shown (see Section Appendix A.1) to be:

$$T_{cr} \propto (C_{in} + 6C_{mm})\, M^3 \left(\tfrac{N}{P} + \lg P\right) + 2\beta M^2 \lg(NP) \tag{10}$$

where C_{mm}, C_{mv}, C_{vv} and C_{in} denote the amortized cost per floating point number for matrix-matrix multiplication, matrix-vector multiplication, vector-vector multiplication and matrix-inversion, respectively and β is the average time to transmit one floating point number between any two processing elements across the network.

3.2. Prefix Product

Initialization: Each processor is assigned $\frac{N}{P}$ block rows. For each local block row i, U_i^{-1} is computed. Thereafter, for each local block row, $\tilde{D}_i$, $\tilde{L}_i$ and $\tilde{b}_i$ are computed and the matrices B_i, as defined in Section 2.2 are constructed.
Serial Prefix Computation: Each processor performs a local prefix product of its B_i matrices and stores it in a matrix S_i^s where $1 \leq i \leq \frac{N}{P}$ refers to local indices. There is no communication in this step.
Parallel Prefix Computation: Each processor k maintains two matrices S_k^p and T_k^p. They are both initialized to the last prefix product, $S_{N/P}^s$, computed in the previous step. This phase has $\log_2 P = \lg P$ stages. In each stage $s \in [0, \lg P - 1]$, processors with ranks k and n exchange T_k^p. If processor ranked k receives communication from a lower ranked processor n, then S_k^p is updated as $S_k^p \leftarrow S_k^p T_n^p$. In any case, T_k^P is always updated as $T_k^p \leftarrow T_k^p T_n^p$.
Finalization: Each processor sends the matrix the partial sum S_k^p to its neighbor to the right. For each local block row index $1 \leq i < \frac{N}{P}$ in processor rank k, S_i^s is updated as $S_i^s \leftarrow S_i^s S_{k-1}^p$. Using the total product matrix T_k^p, x_1 (and hence Y_1) is locally computed using Eqn (9). For each local i, Y_i (and hence x_i) is then locally computed using S_i^s and Y_1 from Eqn (13). The final solution is now available across all the processors.
Total Runtime: Runtime T_{pp}, of this prefix product-based tridiagonal solver can be shown (see Section Appendix A.2) to be:

$$T_{pp} \propto (C_{in} + 18C_{mm})\, M^3 \left(\tfrac{N}{P}\right) + 24C_{mm}M^3 \lg P + \rho\beta M^2 \lg P \tag{11}$$

4. Influence of Block Size on Performance

When the block size M is treated as a constant in Eqn (10) and Eqn (11) along with C_{in} and C_{mm}, the asymptotic runtimes of the two algorithms, based solely on the parameter set $\{N, P\}$, are:

$$T_{cr} = O(\tfrac{N}{P} + \lg(NP)) \quad \text{and} \quad T_{pp} = O(\tfrac{N}{P} + \lg P)$$

These runtimes are misleading. As shown in Figure 1, the relative runtimes of the two algorithms can be seen to depend on the choice of M, N and P. A more careful analysis of their runtimes based on the augmented parameter set $\{M, N, P\}$, therefore, becomes necessary to understanding the performance of one algorithm relative to the other. Accordingly, let $C = \max\{C_{mm}, C_{in}\}$. Then, Eqn (10) and Eqn (11) yields:

$$T_{cr} = C_1 M^2 \left(7CM\left(\tfrac{N}{P} + \lg P\right) + 2\beta \lg(NP)\right) \tag{12}$$

$$T_{pp} = C_2 M^2 \left(19CM\left(\tfrac{N}{P}\right) + 24CM \lg P + \rho\beta \lg P\right) \tag{13}$$

where C_1 and C_2 are positive constants. Comparing the two yields the following result:

Result 1. *$T_{pp} \leq T_{cr}$ when*

$$M \leq \alpha_{arch} \frac{\lg N}{\lg P} \tag{14}$$

where $N \geq P$ and

$$\alpha_{arch} = \frac{4\beta}{C} \cdot \frac{C_1}{(19C_2 - 7C_1)} \tag{15}$$

Proof. Using Eqn (10) and Eqn (11) and setting $T_{pp} \leq T_{cr}$ yields:

$$\begin{aligned} M &\leq \frac{1}{C} \cdot \frac{(2C_1 - \rho C_2)\beta \lg P + 2\beta C_1 \lg N}{(19C_2 - 7C_1)\frac{N}{P} + (24C_2 - 7C_1)\lg P} = \beta K \left[\frac{P \lg(NP)}{N + P \lg P}\right] \text{ where } K = \frac{2C_1}{C(19C_2 - 7C_1)} \\ &\leq \beta K \left[\frac{P \lg(NP)}{P \lg P}\right] = \beta K \left[1 + \frac{\lg N}{\lg P}\right] \leq 2\beta K \frac{\lg N}{\lg P}\ , \text{ when } N \geq P \end{aligned}$$

Since $\alpha_{arch} = 2\beta K$, the result is proven. □

Result (1) reveals two important observations, namely: (a) a critical value of M separates the parameter space spanned by M, N and P into two distinct sub-spaces such that one favors a parallel prefix-based solver while the other favors a cyclic reduction-based approach, and (b) for an identical implementation of underlying array operations (e.g., BLAS), the constant α_{arch} depends entirely on machine-specific constants. Accordingly, the critical M value depends on two factors – σ_{arch} that is completely machine dependent and $\frac{\lg N}{\lg P}$ whose value depends purely on a problem specification. In particular, when $N = P$, this critical value of M depends only on machine dependent constants.

To compare the weak scaling properties of the two algorithms, let $P = N^{\gamma}$ where $0 < \gamma \leq 1$. We exclude the sequential case of $P = 1$ ($\gamma = 0$) here. Also, we limit our discussion to cases for which $N \geq P$ ($\gamma \leq 1$). The following result is of particular interest in practice[3]:

Result 2. *For large N and $M \rightarrow N$, the iso-granular runtime of both algorithms increase linearly with the problem size.*

Proof. Let $\kappa = N^{1-\gamma} = \frac{N}{P}$. The ratio κ remains constant in weak scaling. Substituting for P in Eqn (12) and Eqn (13) yields:

$$\begin{aligned} T_{cr} &= C_1 M^2 \left(7\kappa CM + 7CM\gamma \lg N + 2(1+\gamma)\beta \lg N\right) \\ T_{pp} &= C_2 M^2 \left(19\kappa CM + 24CM\gamma \lg N + \rho\gamma\beta \lg N\right) \end{aligned}$$

Differentiating with respect to N yields:

$$\frac{dT_{cr}}{dN} = C_1 M^2 \left(7C\gamma \frac{M}{N} + \frac{2(1+\gamma)\beta}{N}\right) \quad \text{and} \quad \frac{dT_{pp}}{dN} = C_2 M^2 \left(24C\gamma \frac{M}{N} + \frac{\rho\gamma\beta}{N}\right)$$

When N is large (the second term becomes negligible) and $M \rightarrow N$ (the first term approaches a constant), both $\frac{dT_{cr}}{dN} = \textit{constant}$ and $\frac{dT_{pp}}{dN} = \textit{constant}$. This indicates that when both the block size M and the number of block rows N are sufficiently large, the iso-granular runtime of both algorithms increase linearly with the problem size. □

A good measure of the strong scalability of an algorithm is its computation-to-communication ratio. A larger computation-to-communication ratio indicates a greater degree of parallelism in the algorithm. Let the computation-to-communication ratio for the two algorithms be denoted by $\alpha_{cr} = T_{cr}^{comp}/T_{cr}^{comm}$ and $\alpha_{pp} = T_{pp}^{comp}/T_{pp}^{comm}$, respectively.

Result 3. $\alpha_{cr} \leq \alpha_{pp} \; \forall \; M$.

Proof. Using Eqn (A.-5), Eqn (A.-4), Eqn (A.-8) and Eqn (A.-7), it follows that:

$$\alpha_{cr} = M \cdot \frac{7C}{2\beta} \cdot \left(\frac{N + P\lg P}{P\lg N + P\lg P}\right) \quad \text{and} \quad \alpha_{pp} = M \cdot \frac{C}{\rho\beta} \cdot \left(\frac{19N}{P\lg P} + 24\right)$$

For equal block sizes, this yields:

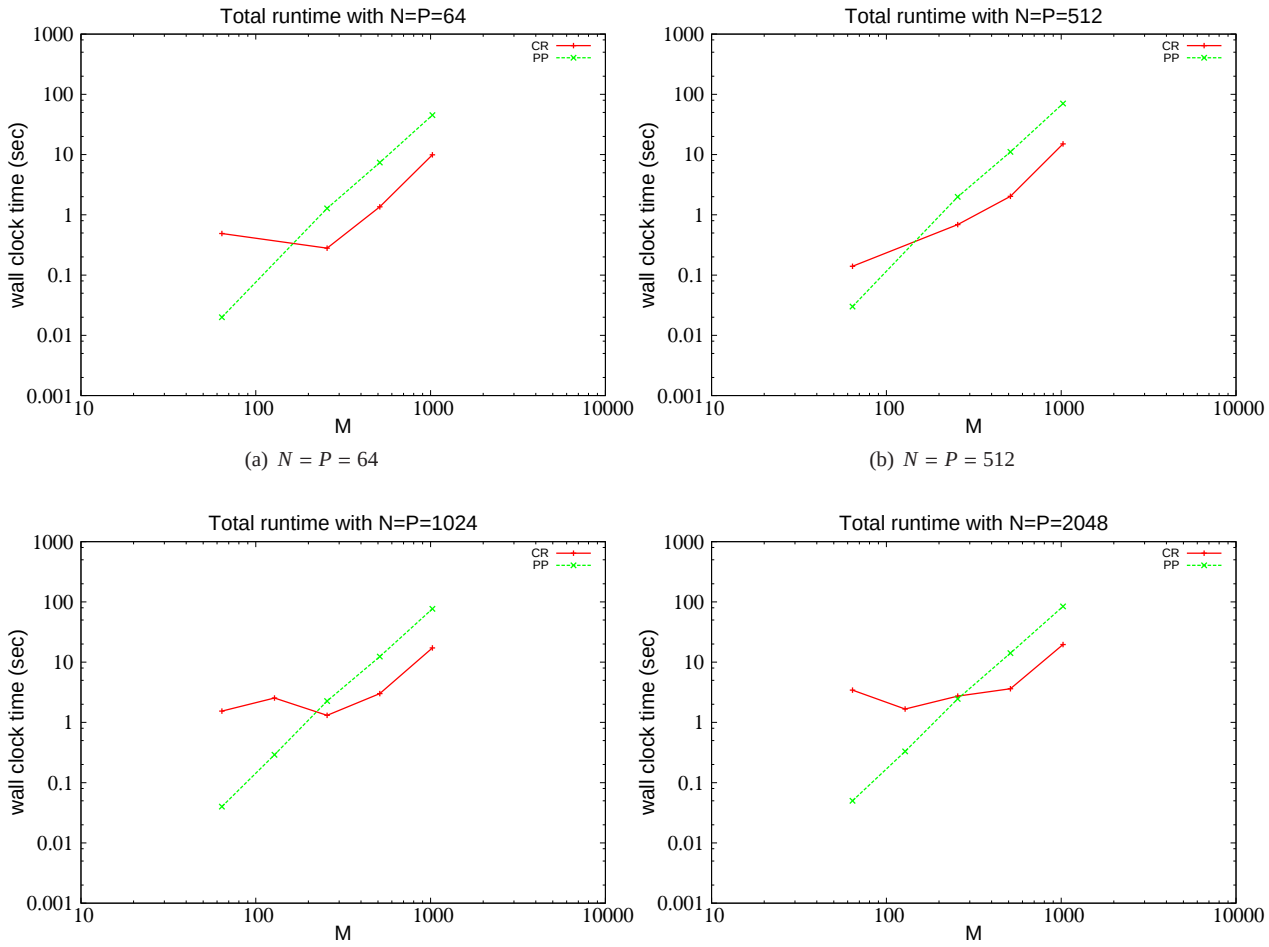
$$\frac{\alpha_{cr}}{\alpha_{pp}} = \frac{7}{2\rho} \cdot \frac{N + P\lg P}{\lg N + \lg P} \cdot \frac{\lg P}{19N + 24P\lg P} \leq \frac{7}{2\rho} \cdot \frac{N + P\lg P}{\lg P} \cdot \frac{\lg P}{19N + 19P\lg P} = \frac{7}{38\rho} < 1$$

since $4 \leq \rho \leq 9$, as shown in Appendix A.1. Therefore, $\alpha_{cr} \leq \alpha_{pp}$. □

Result (3) indicates that a parallel prefix-based solver exhibits better scalability (in the strong sense) than one based on cyclic reduction for the same block size.

5. Experimental Verification

To verify the results from the previous section, both algorithms were implemented and executed on a Cray XT4 machine with a quad-core 2.3 GHz single socket AMD Opteron processor (Budapest) and 8 GB of memory in each compute node. The nodes are connected via a high-bandwidth SeaStar interconnect. Solutions obtained by both solvers in the experiments reported here were numerically stable.



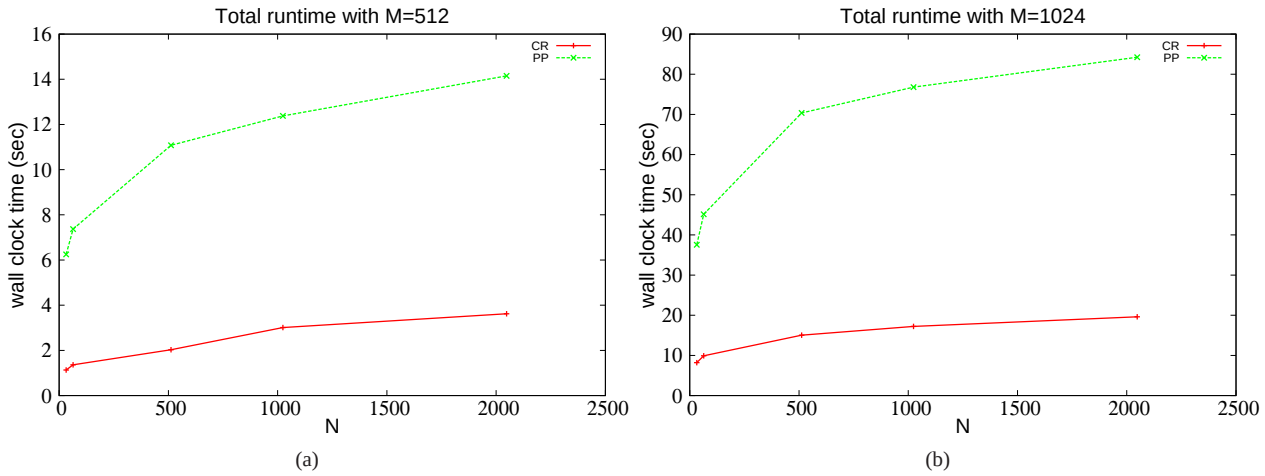
(a) $N = P = 64$ (b) $N = P = 512$

(c) $N = P = 1024$ (d) $N = P = 2048$

Figure 2: Runtimes of cyclic reduction- and parallel prefix-based solvers with varying M. CR denotes cyclic reduction and PP denotes parallel prefix.

In Figure 2, runtimes of the cyclic reduction- and parallel prefix- based algorithms are reported for varying block sizes. A unit granularity ($N = P$) was maintained for these experiments. From Eqn (14), this implies that the critical block size M becomes independent of the values of N and P. The critical block size is indeed found to be almost identical (at $M \sim 150$) when Figure 2(a) and Figure 2(b) for $N = P = 64$ and $N = P = 512$ are compared. Similarly, for Figure 2(c) and Figure 2(d) in which the cases $N = P = 1024$ and $N = P = 2048$ are compared. In the latter case, the critical block size $M \sim 170$. Note that the critical block sizes when $N = P = 1024$ is roughly the same as when $N = P = 2048$, at $M \sim 150$. Similarly, it remains the same when $N = P = 64$ and $N = P = 512$, at $M \sim 170$. Since both algorithms were executed with the same granularity on the same hardware platform, this difference appears to violate Eqn (15) which states that α_{arch} is purely machine dependent and should have no dependence on P. The reason for this apparent contradiction is that though Eqn (15) is ideally true, in practice, the value of β used in the analysis is not a true constant. It changes, albeit slowly, depending on the scale of the network sub-system being exercised during an execution. The average cost of transmitting a floating point number across the network tends to grow with increasing P. Since $\alpha_{arch} \propto \beta$, the critical value of M also increases. It is this drift that is observed above when P changes from 64 to 2048.

[3] An extremely practical use of this result is in estimating processor time to be requested when submitting a batch job on larger number of processors but with the same granularity.

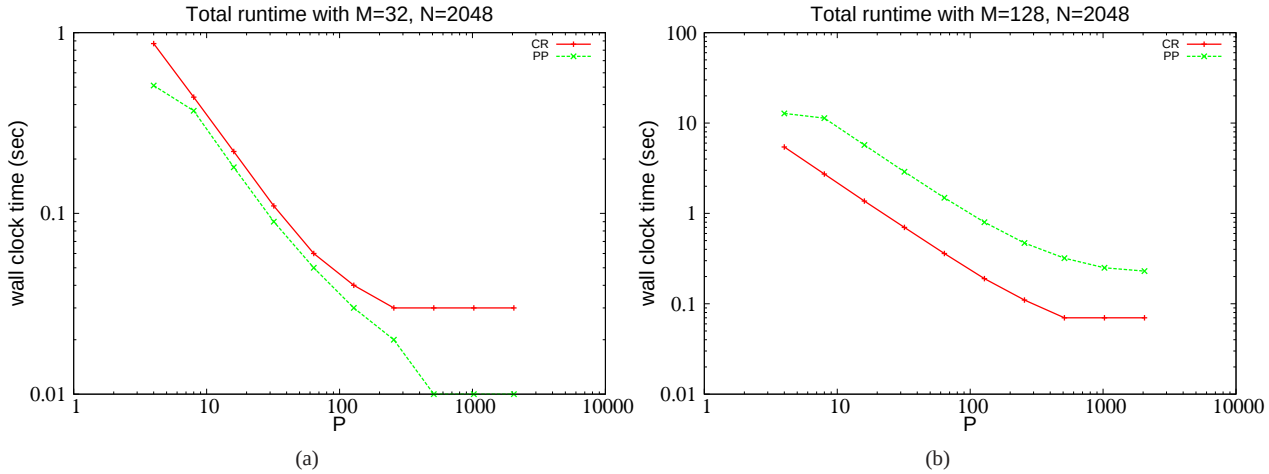


(a) (b)

Figure 3: Weak scaling results for (a) $M = 512$ and (b) $M = 1024$ as N and P become large while maintaining a constant granularity $\frac{N}{P}$.

In Figure 3, iso-granular scaling behavior of the two algorithms is shown. Since M is much larger than the critical value, the runtimes of the cyclic reduction-based solver outperforms the prefix computation-based solver. While maintaining the same granularity (in other words, using proportionately larger number of processors with increasing problem size), the runtime of both solvers increases super-linearly with N when N is small but for larger values of N, the runtime change becomes linear with N in accordance with Result (2). Figure 3 illustrates this for $M = 512$ and $M = 1024$.

Strong scaling behavior of both algorithms is shown in Figure 4 for $N = 2048$. Result (3) suggests that irrespective of the block size M, a prefix computation-based solver should scale better than a cyclic reduction-based solver when the number of processors is increased while maintaining a constant problem size. This is shown in Figure 4 for two block sizes, namely, $M = 32$ which is smaller than the critical block size and $M = 128$ which is larger. Note that, in accordance with Result (3), the number of processors beyond which adding more processors no longer benefits the runtime performance is reached earlier by the cyclic reduction-based solver compared to the prefix computation-based one in both cases. It should be noted that *better (strong) scalability does not necessarily imply better parallel runtime*. Figure 4(b) is such an example in which which, despite better scalability, the prefix computation-based solver under-performs compared to the cyclic reduction-based solver.



(a) (b)

Figure 4: Strong scaling results with $N = 2048$ and (a) $M = 32$ and (b) $M = 1024$.

6. Conclusions

Analyses of cyclic reduction and parallel prefix-based solvers for block tridiagonal systems of equations based on the parameter set $\{N, P\}$ suggest better runtimes for a parallel prefix-based solver. However, in practice, this is not always found to be true. Motivated by the need for the fastest block tridiagonal solvers in applications such as fusion simulations and to better understand their parallel performance, an in-depth analysis of the two specialized algorithms was carried out based on an augmented parameter set $\{M, N, P\}$. The analysis reveals that the performance difference between the two algorithms stems from a trade-off between computation and communication overheads that is determined by the block size.

We show that a critical block size that determines this trade-off point emerges naturally. It separates the parameter space spanned by $\{M, N, P\}$ into distinct regions that favor one or the other algorithm. In addition, the dependence of this critical block size on the parameters of the problem as well as on architecture specific machine constants was derived. Empirical results, based on implementations that scale to 2,048 cores of a Cray XT4, were obtained. Theses results convincingly agree with the theoretical findings.

The results reported here fill a practical gap in the understanding of two important direct solvers specialized for block tridiagonal systems of equations and, in particular, are expected to aid domain scientists in choosing the appropriate solver for their simulations.

7. Acknowledgements

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

This effort has been supported by research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Appendix A. Complexity Analysis

For complexity analysis, we will adopt the following notations. Let C_{mm}, C_{mv}, C_{vv} and C_{in} denote the implementation- and architecture- dependent amortized cost per floating point number for matrix-matrix multiplication, matrix-vector multiplication, vector-vector multiplication and matrix-inversion, respectively. We will assume that identical implementations of these array operations are used in both the algorithms. In addition, let the average time to transmit one floating point number between any two processing elements across the network be denoted by β.

Appendix A.1. Complexity of Cyclic Reduction Algorithm

Forward Phase I: In this phase, $N/2^l > P$ for all $0 \leq l < (n-q)$, where l denotes the level of recursion. At each such level, a processor processes 2^{n-q-l} block rows. For each even row, one matrix inversion, two matrix-matrix products and one matrix-vector products are performed on block-sized matrices. For each odd row, four matrix-matrix products and two matrix-vector products on block-sized matrices are performed locally at each processor. Thus, if T_E^l and T_O^l denotes the computation costs to process the even and odd rows in level l, respectively, then:

$$\begin{aligned} T_E^l &\propto 2^{n-q-1-l}\left[C_{in}M^3 + 2C_{mm}M^3 + C_{mv}M^2\right] \\ T_O^l &\propto 2^{n-q-1-l}\left[4C_{mm}M^3 + 2C_{mv}M^2\right] \end{aligned}$$

Summing over all the levels in this phase, the total computational cost of forward phase I can be shown to be:

$$T_{FI}^{comp} \propto (2^{n-q}-1)\left[C_{in} + 6C_{mm}\right]M^3 = \left[C_{in} + 6C_{mm}\right]M^3\left(\tfrac{N}{P} - 1\right)$$

upto leading orders in M. After each even row is processed, the modified L_{2k}, U_{2k} and b_{2k} (for suitably chosen k within bounds for the level in question) are communicated to the neighboring processor. This involves sending two $M \times M$ matrices and a vector of length M. Thus, the communication cost in each level of this phase is equal to $\beta \cdot 2M^2$ upto leading orders in M. Thus, the total communication cost of forward phase I is:

$$T_{FI}^{comm} \propto 2\beta(n-q)M^2 = 2M^2\beta \lg\left(\tfrac{N}{P}\right)$$

Forward Phase II: In this phase, $N/2^l \leq P$ for all $(n-q) \leq l < n$, where l denotes the level of recursion. In each recursion step, a processor processes at most one row block, each involving matrix operations identical to those in the previous phase. Thus, the total computation time across all the $\lg P = q$ stages of forward phase II is:

$$T_{FII}^{comp} \propto [C_{in} + 6C_{mm}]\, M^3 \lg P$$

Unlike the two matrices sent to the one neighbor alone in the first zone, each active processor in a level sends two matrices each to two neigbors for a total of four matrices. The total communication cost of forward phase II is:

$$T_{FII}^{comm} \propto 4\beta q M^2 = 4\beta M^2 \lg P$$

Backward Phases I and II: These two phases do not involve array operations and all communications involve vector messages. As such, both the computation and communication costs of these two phases are linear in M, and hence ignored in this analysis.

Total Runtime: Summing up all costs above, the total computation and communication costs of a tridiagonal solver based on cyclic reduction are:

$$T_{cr}^{comp} \propto [C_{in} + 6C_{mm}]\, M^3 \left(\tfrac{N}{P} + \lg P\right) \quad \text{(A.-5)}$$

$$T_{cr}^{comm} \propto 2\beta M^2 \lg(NP) \quad \text{(A.-4)}$$

and the total runtime $T_{cr} = T_{cr}^{comp} + T_{cr}^{comm}$.

Appendix A.2. Complexity of Prefix Product Algorithm

Initialization: In the initialization step, a $M \times M$ matrix, U_i is inverted only once, followed by two matrix-matrix multiplications and one matrix-vector multiplication for a total cost of $(C_{in}M^3 + 2C_{mm}M^3 + C_{mv}M^2)$. In general, there are $\frac{N}{P}$ block rows per processor for which the total initialization cost is:

$$(C_{in} + 2C_{mm})M^3(\tfrac{N}{P})$$

up to leading orders in M. There is no communication cost in this step.

Serial Prefix Computation: Each block matrix B_i is of the form:

$$\begin{bmatrix} B_{M\times M}^{11} & B_{M\times M}^{12} & B_{M\times 1}^{13} \\ B_{M\times M}^{21} & B_{M\times M}^{22} & B_{M\times 1}^{23} \\ 0_{1\times M} & 0_{1\times M} & 1 \end{bmatrix}$$

where the subscript of each block sub-matrix refers to the number of rows times the number of columns in it. The number of floating point operations needed to multiply two such matrices B_i and B_j is, therefore, $(8C_{mm}M^3 + 4C_{mv}M^2 + 2C_{vv}M + 1)$. When $N > P$, each processor has $\frac{N}{P}$ such matrices whose partial products are computed after $\frac{N}{P} - 1$ matrix-matrix multiplications for a total cost of:

$$8C_{mm}M^3(\tfrac{N}{P} - 1)$$

Parallel Prefix Computation: In each of the $\lg P$ parallel stages, three matrix-matrix multiplications take place – one in the lower ranked receivers and two in the higher ranked receivers – for a total cost of:

$$3 \cdot 8C_{mm}M^3 \lg P = 24C_{mm}M^3 \lg P$$

As is typical of any parallel prefix computation, each step has $P/2$ unique sender-receiver pair. In each of the $\lg P$ stages, every processor sends to exactly one processor (and receives from exactly one processor). Thus, there are $\lg P$ rounds of communication with a message size of $4M^2 + 4M + 1 = \rho M^2$ (where $4 \leq \rho \leq 9$)[4] in each. Thus, the total communication cost for the parallel prefix stage is:

$$\rho\beta M^2 \lg P$$

Finalization: In the final step, each processor computes $(\frac{N}{P} - 1)$ matrix-matrix multiplications, one matrix-inversion and two matrix-vector multiplications for a total cost of:

$$8C_{mm}M^3(\tfrac{N}{P} - 1)$$

There is one round of communication with $\rho\beta M^2$ cost in this step.
Total Runtime: Summing up all costs above, the total computation and communication costs of a tridiagonal solver based on parallel prefix computation are:

$$T_{pp}^{comp} \propto [C_{in} + 18C_{mm}]\, M^3 \left(\tfrac{N}{P}\right) + 24C_{mm}M^3 \lg P \quad \text{(A.-8)}$$

$$T_{pp}^{comm} \propto \rho\beta M^2 \lg P \quad \text{(A.-7)}$$

and the total runtime $T_{pp} = T_{pp}^{comp} + T_{pp}^{comm}$.

References

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[2] S. Balay, K. Buschelman, V. Eijkhout, W. Gropp, D. Kaushik, M. Knepley, L. C. Mcinnes, B. Smith, H. Zhang, PETSc 2.0 Users Manual, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.2297`.

[3] SGI/CRAY Scientific Computing Software Library, `http://www.sgi.com/products/software/irix/scsl.html`.

[4] IBM ESSL Manual, `http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.essl.doc/esslbooks.html`.

[5] MUMPS, `http://graal.ens-lyon.fr/MUMPS/`.

[6] SuperLU, `http://crd.lbl.gov/~xiaoye/SuperLU/`.

[7] L. H. Thomas, Elliptic Problems In Linear Difference Equations Over A Network , Watson Sci. Comput. Lab. Rep., Columbia University.

[8] R. W. Hockney, A Fast Direct Solution Of Poisson's Equation Using Fourier Analysis, Journal of the ACM 12 (1) (1965) 95–113.

[9] H. S. Stone, An Efficient Parallel Algorithm For The Solution Of A Tri-diagonal Linear System of Equations, Journal of the ACM 20 (1) (1973) 27–38.

[10] D. Heller, Some Aspects Of The Cyclic Reduction Algorithm For Tri-diagonal Linear System, SIAM Journal of Numerical Analysis 13 (4) (1976) 484–496.

[11] H. H. Wang, A Parallel Method For Tri-diagonal Equations, ACM Trans. on Mathematical Software 7 (2) (1981) 170–183.

[12] H. X. Lin, A Bibliography of Parallel Algorithms for Tri-diagonal Systems, `http://ta.twi.tudelft.nl/wagm/users/lin/Biblio/tri_sol.html`.

[13] E. Santos, Optimal Parallel Algorithms for Solving Tridiagonal Linear Systems, in: Procs. of Euro-Par, 1997, pp. 700–709.

[4] Since $\rho = (4M^2 + 4M + 1)/M^2 = 4 + 4/M + 1/M^2$, the limits are obtained by considering the boundary values, 1 and ∞, of M.