# Parallelization of Linear Tridiagonal System: A Cyclic Reduction Approach

Bowen Zhu, Linglai Chen

May 2023

## 1 Introduction

Tridiagonal systems of linear equations arise in various fields. They are particularly common in the numerical solution of partial differential equations, where discretization methods often lead to tridiagonal systems. The solution of tridiagonal systems of linear equations is a fundamental problem in numerical linear algebra with applications in various fields such as fluid dynamics, heat transfer, and financial mathematics. Traditional direct methods like LU or Cholesky factorization can be used to solve these systems with a complexity of $\Theta(n)$, where $n$ is the order of the system. However, these methods cannot exploit the potential parallelism.

In this project, we explore the cyclic reduction algorithm, an alternative approach that leverages the structure of tridiagonal systems to effectively solve them in parallel. While the total work of cyclic reduction is $\Theta(n \log n)$, which is more than traditional methods, it can exploit up to $n$-fold parallelism and requires only $\Theta(\log n)$ time in best case.

## 2 Tridiagonal Systems

A tridiagonal system is a special type of linear system where the coefficient matrix has non-zero elements only on the main diagonal, the diagonal above it, and the diagonal below it. The system can be represented as follows:

$$\begin{bmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix} \tag{1}$$

A tridiagonal system of order $n$ is typically solved using LU or Cholesky factorization. These methods incur no fill and require $\Theta(n)$ operations, but they yield a serial thread of length $\Theta(n)$ through the task graph, hence providing no opportunity for parallelism.

# 3 Cyclic Reduction Method

Cyclic reduction is an alternative method for solving tridiagonal systems that can be parallelized. The method involves transforming the original system into an equivalent system with fewer equations, where the new system can be solved more efficiently.

The $i$-th equation of the original tridiagonal system

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = y_i \tag{2}$$

is transformed into

$$\bar{a}_i x_{i-2} + \bar{b}_i x_i + \bar{c}_i x_{i+2} = \bar{y}_i \tag{3}$$

where

$$\begin{aligned} \bar{a}_i &= \alpha_i a_{i-1}, & \bar{b}_i &= b_i + \alpha_i c_{i-1} + \beta_i a_{i+1}, \\ \bar{c}_i &= \beta_i c_{i+1}, & \bar{y}_i &= y_i + \alpha_i y_{i-1} + \beta_i y_{i+1}, \end{aligned}$$

and $\alpha_i = -a_i/b_{i-1}$, $\beta_i = -c_i/b_{i+1}$. After transforming each equation in the system, the matrix of the resulting new system has a different form

$$\begin{bmatrix} \bar{b}_1 & 0 & \bar{c}_1 & & & & \\ 0 & \bar{b}_2 & 0 & \bar{c}_2 & & & \\ \bar{a}_3 & 0 & \bar{b}_3 & 0 & \bar{c}_3 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & \bar{a}_{n-2} & 0 & \bar{b}_{n-2} & 0 & \bar{c}_{n-2} \\ & & & \bar{a}_{n-1} & 0 & \bar{b}_{n-1} & 0 \\ & & & & \bar{a}_n & 0 & \bar{b}_n \end{bmatrix} \tag{4}$$

This new system can be re-ordered to group equations with odd indices together and those with even indices together. The matrix of the system then has the following form:

$$\begin{bmatrix} \bar{b}_1 & \bar{c}_1 & & & & & & \\ \bar{a}_3 & \bar{b}_3 & \ddots & & & & & \\ & \ddots & \ddots & \bar{c}_{n-3} & & & & \\ & & \bar{a}_{n-1} & \bar{b}_{n-1} & 0 & & & \\ & & & 0 & \bar{b}_2 & \bar{c}_2 & & \\ & & & & \bar{a}_4 & \bar{b}_4 & \ddots & \\ & & & & & \ddots & \ddots & \bar{c}_{n-2} \\ & & & & & & \bar{a}_n & \bar{b}_n \end{bmatrix} \tag{5}$$

After this transformation, the resulting new system is reordered to place odd indices before even indices. By reordering the system, it breaks into two

independent tridiagonal subsystems that can be solved simultaneously. Each resulting tridiagonal subsystem can in turn be solved using the same technique, resulting in a recursive algorithm and an opportunity for parallel computation. This can continue until the subsystems are trivially solvable, at which point we can simply obtain the solution via traditional solvers.

# 4 Parallelism in Implementation

Given the parallelism opportunities within the cyclic reduction algorithm, it is highly beneficial to utilize the power of multiple processors if available. We implemented a parallelized tridiagonal solver using the cyclic reduction method in Julia. Our implementation for solving tridiagonal systems exploits two sources of parallelism: simultaneous transformation of equations in the system and simultaneous solution of multiple tridiagonal subsystems. We combine multithreading and multiprocessing to exploit both the inter-processor and intra-processor parallelism inherent in modern computing systems. Multithreading is used to transform equations in parallel, while multiprocessing is used to solve multiple subsystems in parallel. This results in a highly parallel implementation that can significantly reduce the time to solve large tridiagonal systems on multi-core and multi-processor systems.

## 4.1 Simultaneous Transformation of Equations

The system of equations is first transformed using the method of cyclic reduction. Each equation in the system can be transformed independently, providing an opportunity for parallelism. Thus, we perform these updates simultaneously across multiple threads using a multithreaded loop in Julia. In our implementation, we use the `Threads.@threads` macro, which automatically manages the distribution of iterations to threads and the synchronization between them.

**for** each index $i$ in the system, in parallel **do**
    Compute new coefficients $\bar{a}_i$, $\bar{b}_i$, $\bar{c}_i$, $\bar{y}_i$ using cyclic reduction formulas
**end for**

## 4.2 Simultaneous Solution of Multiple Tridiagonal Subsystems

Once the system has been transformed, it is split into two separate subsystems: one for odd-indexed equations and another for even-indexed equations. These two subsystems can be solved independently, providing another opportunity for parallelism.

The implementation uses Julia's multiprocessing capabilities to solve the subsystems concurrently. We utilized Julia's Distributed library, which provides the `@spawnat` macro, a powerful tool for managing distributed computations. Crucially, `@spawnat` is non-blocking, which enables us to split the computation between the odd and even subsystems across different processors. In particular,

as long as there are additional processors available, our implementation executes the next function call to solve the odd-indexed subsystem on the current processor, and at the same time assign a new processor to execute the function call to solve the even-indexed subsystem.

This is done recursively, so that each level of the recursion spawns new tasks for the next level. This continues until all available processors are being used, after which no new tasks are spawned and each processor is responsible for solving all of its remaining subsystems.

**if** there are still processors available **then**
    Spawn task to solve odd-indexed subsystem on the current processor
    Spawn task to solve even-indexed subsystem on a new processor
**else**
    Solve odd-indexed subsystem on the current processor
    Solve even-indexed subsystem on the current processor
**end if**
Get solutions of odd and even-indexed subsystems
Merge solutions into final solution vector $x$

## 4.3 Switching Back to Traditional Solvers

When the size of the matrix becomes sufficiently small, the benefits of further recursive calls in the cyclic reduction algorithm diminish. This occurs when the number of equations reduces to a point where there is no significant parallelism to exploit. In such scenarios, it becomes more efficient to switch to traditional methods for solving tridiagonal systems, such as LU and Cholesky factorization. Switching to traditional methods for small subsystems allows us to avoid the unnecessary overhead associated with parallel communication, synchronization, and further recursive calls. It ensures that computational resources are efficiently utilized and that the most appropriate solution method is employed for the given problem size.

When the matrix size reaches this threshold, we can halt the recursive calls and solve the subsystems directly using traditional methods. By incorporating a termination condition in the parallel cyclic reduction algorithm, we can dynamically determine the appropriate point at which to switch from parallel execution to traditional serial methods. This condition is typically based on the size of the subsystems or the number of equations remaining.

Careful consideration and benchmarking are required to determine the optimal threshold for stopping the recursive calls and employing traditional methods. It is important to note that the threshold at which we transition to traditional methods will depend on various factors, including the specific characteristics of the problem, the hardware architecture, and the available computational resources. In fact, this decision can be made once all processors have been assigned their respective subsystems to solve. At this stage, the parallelism has been fully utilized, and it is potentially more beneficial to employ fast and optimized linear solvers such as LU and Cholesky factorization.

Another group of methods for truncated tridiagonal systems is to solve each subsystem roughly (e.g. ignoring nonzero elements outside of main diagonals), obtain an approximate solution for the whole system, and use some iterative numerical linear algebraic methods to compute a more precise solution. Given the sparsity of tridiagonal matrices, each iteration in an iterative method will be fast as the most computationally intensive part, the matrix-vector product, takes only $O(N)$ operations. However, many iterative linear algebraic methods require certain properties of matrices to solve. Therefore, such methods shall only be used on matrices whose properties satisfy the iterative methods to be used.

# 5 Complexity Analysis

In order to understand the efficiency and effectiveness of the cyclic reduction algorithm, it's important to understand the complexity and parallelization potential of the method.

## 5.1 Computational Complexity

The cyclic reduction method operates in $\log n$ steps, where $n$ is the number of equations in the tridiagonal system. This is due to the divide-and-conquer strategy employed by the algorithm, in which the original system is recursively broken down into smaller, independent tridiagonal systems. At each step, the system size is effectively halved, leading to the logarithmic number of steps.

In each of these $\log n$ steps, the algorithm performs transformations on the equations of the system. The transformations involve calculating new coefficients for the system based on the old ones, and each transformation requires a constant number of operations. Since these transformations are performed for each equation in the system, the total number of operations in each step is proportional to the number of equations, i.e., $\Theta(n)$.

Therefore, the total work done by the cyclic reduction method is the product of the number of steps and the work done per step, leading to a total computational complexity of $\Theta(n \log n)$. This is more than the $\Theta(n)$ work required by the LU or Cholesky factorization methods for tridiagonal systems when executed serially. However, this comparison does not take into account the parallelization potential of the cyclic reduction method, which can significantly speed up its execution on parallel computing architectures.

## 5.2 Parallelization Potential

The main advantage of the cyclic reduction method is its high degree of parallelism. In each step of the method, the transformations applied to the equations of the system are independent of each other, meaning that they can be performed concurrently. This allows the method to exploit up to $n$-fold parallelism, with $n$ being the size of the system.

In the best case, the time required by the cyclic reduction method is then only $\Theta(\log n)$, which corresponds to the depth of the divide-and-conquer operation. This is a significant reduction compared to the $\Theta(n)$ time required by the serial execution of LU or Cholesky factorization.

### 5.3 Early Truncation

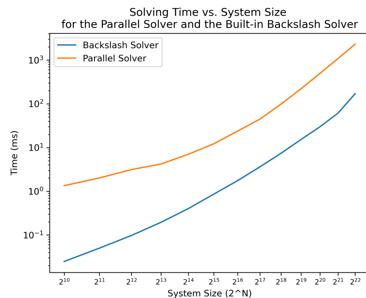
In many cases, the matrix of the tridiagonal system becomes approximately diagonal in fewer than $\log n$ steps of the cyclic reduction method. When this happens, the remaining off-diagonal elements are sufficiently small that they can be ignored without significantly affecting the accuracy of the solution.

In such cases, the reduction can be truncated, meaning that the remaining steps of the method are skipped and the system is solved either directly or iteratively. This can significantly reduce the actual work done by the method and the time required to solve the system, while still attaining an acceptable level of accuracy. The potential for early truncation is another factor that contributes to the efficiency and effectiveness of the cyclic reduction method.

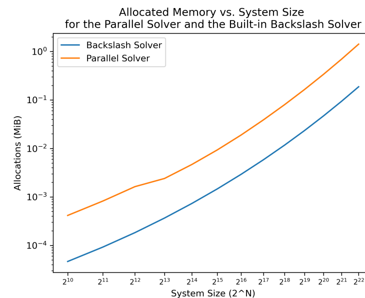
## 6 Results

### 6.1 Random Matrices

The code was tested on randomly generated matrices and accuracy was confirmed by comparing them to the reference solution. Running an unoptimized implementation shows that while the parallel solver is faster than the sequential cyclic reduction solver given a large system size, it cannot beat the built-in backslash solver in Julia. The time and memory comparison is shown in figure 6.



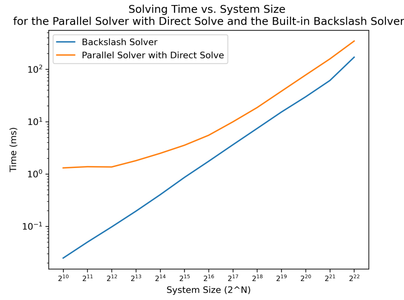

(a) Timings of original cyclic reduction solver



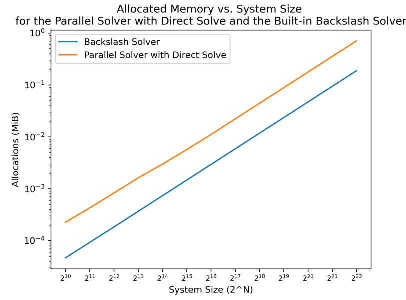

(b) Allocations of original cyclic reduction solver

Figure 1: Performances of Cyclic reduction

Experiment results indicate that using `Threads.@threads` actually slows down the solver, so we decided to use `@simd` instead. We also tried to incor-




(a) Timings of Early Termination with direct solver




(b) Allocations of Early Termination with direct solver

Figure 2: Performances of Early Termination with Direct Solver

porate early termination of the recursion and switching back to the built-in backslash solver. Figure 2 shows the results of early termination with direct solvers. As matrix sizes increase, both the timing and allocation memory sizes of the backslash solver and our termination-early-with-direct-solver form log-log linear relationships with matrix size. Asymptotically, the timing/memory ratio of our solver to backslash is constant. While we aimed to achieve parallelization in our recursive tridiagonal solver, the timing results indicate that the current implementation still did not obtain the desired speedup. This lack of performance improvement could be discussed in terms of both the multithreading and multiprocessing effort.

### 6.1.1 Multithreading Overhead

Using multithreading in every recursive call leads to the creation of thousands of threads. The overhead associated with managing these threads could negatively impact performance, nullifying the benefits of parallelism. Each thread creation, synchronization, and termination consumes both time and system resources, and the for loop for updating the the coefficients may not take as much time by itself. Furthermore, the shared memory architecture of multithreading may introduce contention and cache coherence issues. In our implementation, the sheer number of threads could have caused the multithreading overhead to outweigh the advantages of parallelism, thereby hindering performance improvement.

### 6.1.2 Multiprocessing Trade-offs

In our attempt to adapt the tridiagonal solver for multiprocessing, we introduced a trade-off. The original algorithm has a complexity of $O(n)$, while our adaptation increased the complexity to $O(nlogn)$. This additional overhead might have impacted the performance negatively.

We tested our implementation using 8 processors for parallelization. When the input size $N$ is large, the benefits of $p$-fold parallelism might not outweigh the additional factor of log n introduced by our adaptation. In practice, the number of processors is likely to be smaller than $logn$. In other words, the $O(nlogn)$ complexity might have dominated the performance, causing the lack of speedup.

## 6.2 Application: 1D Poisson Equation

As an application, we consider the simple finite difference scheme for 1D Poisson Equation. Specifically, we consider the following ODE problem:

$$u''(x) = f(x), x \in (0,1); u(0) = u(1) = 0$$

where $f(x) = \exp(\sin(x))$.

To solve this ODE problem numerically, we consider uniform points $x_i = \frac{i}{N+1}$, where $N$ is the number of grid points we select and $i = 1, 2 \cdots N$. Let $h = \frac{1}{N+1}$. By approximating $u''(x_i) \approx \frac{-U_{i-1}+2U_{i+1}-U_{i+1}}{h^2}$ where $U_i$ represents the approximated value of $u(x_i)$, the ODE system above can be transformed to the following linear system:

$$\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_{n-1} \\ U_n \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ f(x_n) \end{bmatrix} h^2 \tag{6}$$

Due to time limit, we haven't visualized our results. However, we've implemented this scheme with cyclic reduction and confirmed our code's accuracy. The implementation can be found in the file "spawnat_ODE_timing_test.jl" at this Github repository.

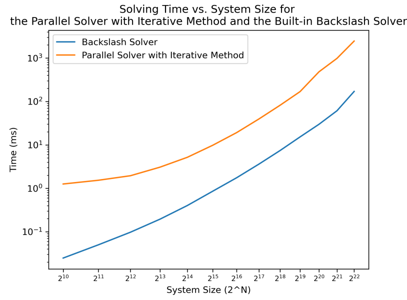
## 6.3 Termination with Iterative Methods

We also test how well cyclic reduction performs when combined with iterative methods. Here we adopt the Jacobi method. Suppose we want to solve $Ax = b$, and $A = L + D + U$, where $L$, $D$, and $U$ represent the lower, diagonal, and upper part of $A$, the Jacobi method suggests the following iterative formula:

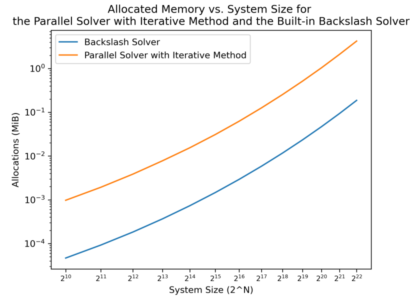
$$x^{k+1} = D^{-1}(b - (L+U)x^k)$$

A sufficient condition for the Jacobi method to work is that the matrix is diagonally dominant (i.e. $\sum_{j\neq i} |a_{i,j}| < |a_{i,i}|$). To ensure convergence, we first initialize the sub-diagonal and super-diagonal of our matrix with random numbers on $[-1, 0]$. Then we initialize the diagonal by setting $a_{i,i} = |a_{i-1,i}| + |a_{i+1,i}| + 1$. Jacobi is thus guaranteed to converge in our experiments.

The results obained from an 8 cores machine are shown in 3. Again, asymptotically, the memory/timing ratio of our method to backslash is constant.




(a) Timings of early termination with Jacobi solver




(b) Allocations of early termination with Jacobi solver

Figure 3: Performances of Early Termination with Jacobi Solver

# 7 Future Works: Reusing Arrays and Reducing Allocations

## 7.1 Design and Procedure

The algorithm initiates with four arrays $a$, $b$, $c$, and $y$, each of size $N$. Here, $a$, $b$, and $c$ represent the diagonals of the tridiagonal matrix, while $y$ signifies the RHS vector.

The function `solve_tridiagonal!` computes modified arrays $\bar{a}$, $\bar{b}$, $\bar{c}$, and $\bar{y}$, each of the same size $N$ based on $a$, $b$, $c$, and $y$. After this computation, $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{y}$ are partitioned into odd and even indexed sub-arrays, each of size $N/2$. These newly formed sub-arrays are then passed as inputs into two recursive calls of the `solve_tridiagonal!` function.

To illustrate the recursion process, let's follow through at least three levels of recursions:

- **Level 1:** The algorithm begins with arrays $a$, $b$, $c$, and $y$, each of size $N$. The function `solve_tridiagonal!` modifies these arrays into $\bar{a}$, $\bar{b}$, $\bar{c}$, and $\bar{y}$, each of size $N$ and then splits them into two sub-arrays each of size $N/2$. There are now two recursive calls, each taking arrays of size $N/2$ as input.

- **Level 2:** For each of the two recursive calls from Level 1, the function further modifies and splits the input arrays of size $N/2$ into two new sub-arrays each of size $N/4$. At this level, there are a total of $2 \times 2 = 4$ recursive calls, each taking arrays of size $N/4$ as input.

- **Level 3:** Similar to Level 2, for each of the four recursive calls from Level 2, the function modifies and splits the input arrays of size $N/4$ into two new sub-arrays each of size $N/8$. At this level, there are a total of $2 \times 4 = 8$ recursive calls, each taking arrays of size $N/8$ as input.

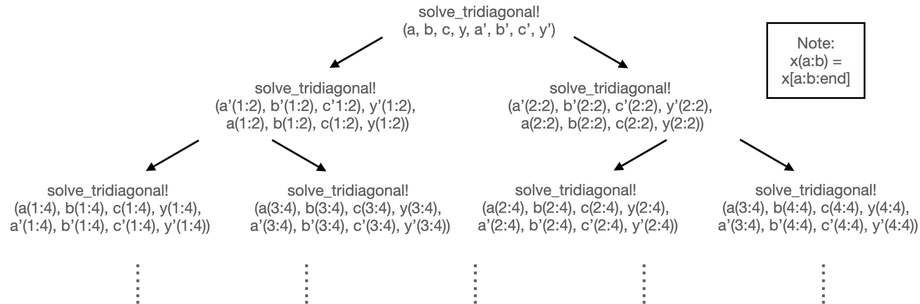



Figure 4: An illustration of our allocation reduction process.

By the time we reach Level 3, we can see a clear pattern: each level of recursion doubles the number of recursive calls while halving the size of the arrays involved in each call. Thus, the total size of the arrays involved in each level of recursion remains constant, equal to $2N$ (i.e., $N$ for $a$ and $N$ for $\bar{a}$). This holds true for the arrays $b$, $c$, and $y$ as well.

Since in theory, each layer of the recursive calls uses arrays with total length of 2N (N for $a$ and N for $\bar{a}$), we hope to reduce allocations by preallocating two arrays arr1 and arr2 of size $N$ and them splitting them and reusing them throughout the subsequent recursion.

More specifically, $arr1$ and $arr2$ are pre-allocated for $a$ and $\bar{a}$, respectively, before the initial call to the solver. These arrays are used in an interleaved fashion in every layer of recursion.

Let’s illustrate the first three layers again, focusing on the specific indices (assuming the 1-based indexing used in Julia):

- **Level 1:** At the initial call, $arr1$ is used for $a$ and $arr2$ for $\bar{a}$. The indices in $\bar{a}$ are then divided into odd (2k+1) and even (2k+2) parts. These correspond to the odd and even indices in the original $a$ array.

- **Level 2:** In the subsequent recursive calls, the halves of $arr2$ (which was $\bar{a}$ in the previous call) are used for $a$ and the halves of $arr1$ are used for $\bar{a}$. Specifically, for the new odd recursive call from the odd part of the previous layer, the indices are 4k+1; for the new even recursive call from the odd part of the previous layer, the indices are 4k+3; for the new odd recursive call from the even part of the previous layer, the indices are 4k+2; and for the new even recursive call from the even part of the previous layer, the indices are 4k+4.

- **Level 3:** At this level, the four quarters of $arr1$ are used for $a$ and the four quarters of $arr2$ for $\bar{a}$. The indices follow a similar pattern as described for Level 2 but are further divided (for example, $8k + 1$ for the odd call of the odd call of the odd call), leading to a total of 8 recursive calls at this level.

Through this recursive process, the data originally stored in $arr2$ (which was $\bar{a}$) will become the new $a$ in the subsequent recursive calls. This interleaved usage of $arr1$ and $arr2$ facilitates efficient memory utilization by effectively reusing the preallocated arrays $arr1$ and $arr2$ and thus minimizing the need for additional allocations.

An illustration of this procedure is given in figure 4.

**Algorithm 1** Tridiagonal Solver with Interleaving

1: **procedure** SOLVETRIDIAGONAL!($a, b, c, y, a_-, b_-, c_-, y_-, spawn, spawnid$)
2: $N \leftarrow \text{size}(a, 1)$
3: **if** $N$ large **then**
4: Initialize $a_-, b_-, c_-, y_-$
5: Compute the modified $a_-, b_-, c_-, y_-$ based on $a, b, c, y$
6: Partition $a_-, b_-, c_-, y_-$ into odd and even indexed sub-arrays
7: $n \leftarrow \frac{N}{2}$
8: **if** $spawn > 0$ **then**
9: $x_{\text{oddTask}} \leftarrow \text{spawnAt}(\text{spawnId}, \text{SolveTridiagonal!}(a_{-\text{odd}}, b_{-\text{odd}}, c_{-\text{odd}}, y_{-\text{odd}},$
10: $a_{\text{odd}}, b_{\text{odd}}, c_{\text{odd}}, y_{\text{odd}}, \text{spawn} - 1, \text{spawnId})$
11: $x_{\text{evenTask}} \leftarrow \text{spawnAt}(\text{spawnId} + 2^{\text{spawn}-1},$
12: $\text{SolveTridiagonal!}(a_{-\text{even}}, b_{-\text{even}}, c_{-\text{even}}, y_{-\text{even}}, a_{-\text{even}}, b_{\text{even}}, c_{\text{even}}, y_{\text{even}},$
13: $\text{spawn} - 1, \text{spawnId} + 2^{\text{spawn}-1})$
14: $x[1:2:\text{end}] \leftarrow \text{fetch}(x_{\text{oddTask}})$
15: $x[2:2:\text{end}] \leftarrow \text{fetch}(x_{\text{oddTask}})$
16: **else**
17: Compute $x_{\text{odd}}$ and $x_{\text{even}}$ without spawning new tasks
18: $x[1:2:\text{end}] \leftarrow \text{fetch}(x_{\text{oddTask}})$
19: $x[2:2:\text{end}] \leftarrow \text{fetch}(x_{\text{oddTask}})$s
20: **return** $x$
21: **end if**
22: **else**
23: Solve using backslash
24: **return** $x$
25: **end if**
26: **end procedure**

## 7.2 Limitations

Through numerical experiments, we found that strict synchronization is required for the above design to produce the correct outputs consistently. In particular, the calls in each subsequent layer must wait for the preceding layer to finish its computation before performing their own computation. Thus there are limitations that prevents the above implementation from giving a better performance:

1. **Synchronization Requirement:** Due to the data dependency, the design necessitates synchronization, implying each layer of calls cannot start

before the previous layer of calls have completed their computation. This requirement of synchronization can limit the speed of the execution, and severely limits the parallel processing potential of the system.

2. **Spatial Locality:** When the size of the array is large, the interleaved design might lose spatial locality, potentially leading to inefficient memory use.

# 8 Conclusion

In summary, the cyclic reduction method provides an efficient way to solve tridiagonal systems of linear equations in parallel. Although it requires more operations than serial methods such as LU or Cholesky factorization when executed serially, the parallelism it offers makes it a suitable choice for large systems where parallel computation can be effectively exploited. By exploiting the inherent parallelism of the algorithm, we can solve large systems with a complexity of $\Theta(n \log n)$, which is a significant improvement over sequential methods for large $n$. The Julia implementation, with its use of both multi-threading and multi-processing, demonstrates how to effectively parallelize this algorithm to leverage the power of modern computing resources.

# 9 Code

Our codes are available in this Github repository.