Implementation of a Parallel Tridiagonal Solver for Linear system of Equations arising in Physicell-BioFVM

Trabajo realizado por:
Shardool Kulkarni

Dirigido por:
Dr. Riccardo Rossi
Dr. Gaurav Saxena

Máster en:
MASTER'S DEGREE IN NUMERICAL METHODS IN ENGINEERING

Barcelona, 20 June 2021

Departamento de Ingeniería Civil y Ambiental

TRABAJO FINAL DE MASTER

Master's Degree
Numerical Methods in
Engineering

Master Thesis

Implementation of a Parallel Tridiagonal Solver for Linear system of Equations arising in Physicell-BioFVM

Author:
Shardool Kulkarni

Professors:
Riccardo Rossi
Gaurav Saxena
Miguel Ponce De Leon

June 21, 2021
Academic Year 2020-2021

Acknowledgements

I would like thank my mentors at the Barcelona Super Computing Center, Dr. Gaurav Saxena and Dr. Miguel Ponce De Leon for their continuous guidance and supervision which helped me complete this work. I would also like to thank Dr. Alfonso Valencia and the whole of his Computational Biology group for giving me an opportunity to work with them. I am also grateful to Professor Riccardo Rossi for being my on-campus supervisor and for his timely support.

I am also thankful to my family and friends who have helped and encouraged me during the two years of my Master Course.

Abstract

PhysiCell/BioFVM is an open source and agent-based software package supporting 2D/3D simulations for multi-cellular biological systems. It is completely written in C++ and enjoys shared-memory parallelization through OpenMP. An attempt is being made to re-structure the code-base to add support for Distributed parallelization through MPI. The biggest bottleneck in the current version of the simulation package is the serial Thomas solver that is used to solve the triadiagonal system of linear equations resulting from the Finite Volume Discretization Method (FVM) of the reaction-diffusion equations modelling the secretion, ingestion of substrates from cells/agents. Our aim in this project is to replace the serial solver with an efficient distributed-parallel solver. For this purpose we experiment with the Cyclic Reduction (CR) algorithm on a shared-memory system but after understanding its limitations with regard to the problem size, we settle on a modified version of the Thomas solver as our preferred choice in distributed-parallel settings. Our experiments show that the Cyclic Reduction algorithm implemented using OpenMP is able to outperform the serial Thomas solver at a certain thread count on a single node. However, we do not extend this algorithm to support distributed parallelism due to the aforementioned problem. Further, we implement an MPI-only version of the modified Thomas algorithm that promises good scalability on multiple nodes of our HPC cluster - the MareNostrum 4 (MN4) supercomputer at the Barcelona Supercomputing Center (BSC). We project and optimistically conclude that for large problem sizes and a high core count, the parallel modified Thomas algorithm can offer significant reduction in the time to solution for complex 3D simulations in PhysiCell/BioFVM.

Keywords: Tridiagonal Matrices, Direct Solvers, Linear Equations, Cyclic Reduction, Thomas Algorithm, OpenMP, MPI

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Simulating multi-cellular systems is an essential challenge in Computational Biology. Knowing how cells grow, divide, die, move and interact, helps biologists solve fundamental problems in Biology [2]. Cells are influenced by external biochemical and biophysical signals continuously. PhysiCell [1] is a virtual laboratory that helps biologists simulate various multi-cellular systems at different Spatio-temporal levels. It is a powerful, open-source, C++ based multi-cellular systems simulator. For example, it simulates the diffusion, uptake, and secretion of substrates in the micro-environment; it also simulates the mechanical interactions and several intra-cellular processes. PhysiCell has been parallelized with OpenMP for a shared-memory system only, and its performance scales linearly with the number of cells. Simulations up to $10^5 - 10^6$ cells are feasible on quad-core desktop workstations; larger simulations are attainable on a single HPC (High-Performance Computing) compute node.

Simulating diffusion is the most demanding computational step in the PhysiCell simulations. It requires solving a Partial Differential Equation (PDE) after a Finite Volume Discretization (FVM) is solved as a multiple linear systems of equations. Optimizing this step can greatly reduce the time taken to simulate experiments in PhysiCell. PhysiCell can be visualized to be made up of two layers: BioFVM [3] and PhysiCell itself. The solver for the linear systems of equations is a part of BioFVM. The work presented here focuses on parallelizing (and thus optimizing) the solution for tridiagonal sparse linear systems. Such systems of equations are found in many problems in computational mechanics, fluid simulations and processes that can be modeled with Poisson equations. The aim of the project is to provide an efficient parallel solver for large, sparse, tridiagonal systems which can replace the current serial solver in the BioFVM layer of PhysiCell.

1.1 Numerical Solution of a Linear system of Equations

Consider a linear system of equations which is represented by a matrix equation

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1}$$

Here A is a non-singular, sparse, square matrix $\mathbb{R}^{n\times n}$. $\mathbf{b} \in \mathbb{R}^n$ is a given vector. Such systems can be solved using Direct or Iterative methods. A standard way of solving Equation 1 is to invert the matrix A and to multiply it with the forcing vector $\mathbf{b}$. Computing an inverse, even for a sparse matrix, is a computationally intensive operation with a complexity of $\mathcal{O}(n^2)$ [6]. Further, the inverse of a sparse matrix *can* be dense matrix and storing A^{-1} may also be demanding on the memory resources. Solving a matrix system by inverting a matrix would also not take into account the sparse nature of the matrix - an important property that can be exploited to obtain the solution faster. Thus, practically we use direct or iterative methods to solve such linear systems.

In a direct method matrix A is decomposed into simpler constituents which are easier to manipulate to obtain the solution. Iterative methods begin with an assumed solution and improve the solution in successive steps to obtain an accurate solution (within a given tolerance limit). Some examples of direct methods include LU decomposition, Cholesky Factorization, and Gaussian Elimination etc, while the Jacobi, Gauss-Seidel, Conjugate Gradient method, SOR (Successive Over Relaxtion) and Krylov family of methods fall in the category of Iterative methods [8].

1.2 Direct Solvers

A sparse linear system is often solved by direct solvers, the simplest of which is the Gaussian Elimination method. In the Gaussian Elimination Method we factorise the coefficient matrix based on the matrix properties [4]. In LU decomposition, we factorize A into triangular matrices L and U i.e. $A = LU$. The two triangular systems $Ly = b$ and $Ux = y$ are solved to obtain the solution x. In Cholesky factorization, we factorise $A = LL^T$. Here L is the lower triangular matrix and L^T, is the conjugate transpose of L. In Cholesky Decomposition, A has to be a symmetric positive definite (SPD) matrix.

A number of solvers have been developed to solve sparse matrices, which use parallel computing . MUMPS (Multifrontal Massively Parallel Sparse Direct Solver) [5] is a distributed mutifrontal package for solving sparse linear systems. It solves SPD, as well as general symmetric matrices. Direct Solvers are suited for smaller systems where the number of variables are up-to N $= 1 \times 10^6$.

1.3 Thomas Algorithm

Thomas Algorithm is also called the tridiagonal matrix algorithm (TDMA). In this algorithm we apply the Gaussian Elimination to a tridiagonal system of equations. A tridiagonal Matrix is a special type of banded matrix that has one sub and super-diagonal. A banded matrix is a sparse matrix whose non-zero entries are confined to a diagonal band.

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_1 & b_2 & c_2 & 0 & 0 \\ 0 & a_2 & b_3 & c_3 & 0 \\ 0 & 0 & a_3 & b_4 & c_4 \\ 0 & 0 & 0 & a_4 & b_5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} \tag{2}$$

The matrix form of Equation 1 can be represented as in Equation 2 and the latter itself can be written as a set of linear equations in the following form.

$$b_i x_{i-1} + a_i x_i + c_i x_{i+1} = y_i \qquad\qquad i = 1, 2, ..., n \tag{3}$$

The Thomas Algorithm solves the linear system in two phases: (1) Forward Elimination and (2) Backward Substitution. Let us consider a modification in the first two equations (see Equation 3),

$$Eq_{i=2}\dot{b}_1 - Eq_{i=1}\dot{a}_1$$

here, Eq$_i$ refers to the i^{th} equation of 3 which gives

$$(b_2 b_1 - c_1 a_2)x_2 + c_2 b_1 x_3 = d_2 b_1 - d_1 a_2 \tag{4}$$

Thus, we have effectively eliminated x_1 from the second equation. Similarly, we can eliminate x_2 with the help of the modified second equation 4 and the third matrix equation.

$$(b_3(b_1b_2 - c_1a_2) - c_2b_1a_3)x_3 + c_3(b_1b_2 - c_1a_2)x_4 = y_3(b_1b_2 - c_1a_2) - (y_2b_1 - y_1a_2)a_3 \tag{5}$$

This procedure is repeated till the n^{th} equation (Forward Elimination). The last equation will involve just one unknown x_n. Once this is solved, the remaining modified equations can be solved with backward substitution.

The coefficients are calculated after dividing by b_i as follows

$$a\prime_i = 0 \qquad b\prime_i = 1 \qquad c\prime_1 = \frac{c_1}{b_1} \tag{6}$$

$$c\prime_i = \frac{c_i}{(b_i - c\prime_{i-1}a_i)} \qquad y\prime_1 = \frac{y_1}{b_1} \qquad y\prime_i = \frac{y_i - y\prime_{i-1}a_i}{(b_i - c\prime_{i-1}a_i)} \tag{7}$$

This gives the following system of equations

$$x_i + c\prime_i x_{i+1} = y\prime_i \qquad i = 1...n-1 \tag{8}$$

$$x_n = y\prime_n \qquad i = n \tag{9}$$

The Thomas Algorithm is an extremely efficient algorithm for the solution of a sparse Tri-diagonal matrices. It's complexity is $\mathcal{O}(n)$. The operation count [20] of the Thomas Algorithm is $5N$ multiplications and $3N$ additions for any system size N .

1.4 Parallel computing

Parallel computing involves the use of multiple instances of resources to solve a single problem. This is accomplished by dividing the problem into multiple smaller parts, solving them with or without coordination and then combining the solutions of the sub-parts to obtain the final solution. Since the advent of parallel computing the sizes of linear sparse linear systems that we are able to solve has increased by a factor of thousands. The number of variables have increased from about 200 in 1970s to more than a billion in the past decade [9]. To take advantage of multiple instances of resources, the algorithm must be designed in way that such parallel computations are possible. Due to increasing demands of computing resources which arise from an increasingly precise simulations of real life problems, a significant body of research work is dedicated to parallel high-performance machines. Such machines work in large computer clusters and are classified as supercomputers.

1.4.1 OpenMP

OpenMP (Open Multiprocessing) is the standard Application Program Interface (API) for parallel programming on shared memory architectures. It is a very flexible interface for developing parallel applications that are based on the concept of multi-threading. In this programming paradigm, we specify a number of threads that work concurrently to perform a given task. Communication is carried out using shared memory i.e. the process level memory is visible to all the threads and communication is done by reading from/writing to the shared memory. In addition to global data, OpenMP threads can also declare data exclusive to each thread. OpenMP is based on the concept of parallel regions i.e. the master thread (equivalent to the process) can *fork* multiple threads on entering a parallel region. When a thread enters a parallel region it becomes the leader. At the end of the parallel region the threads *join* and the leader resumes the execution of the sequential code (fork-join model). This is illustrated in figure 1

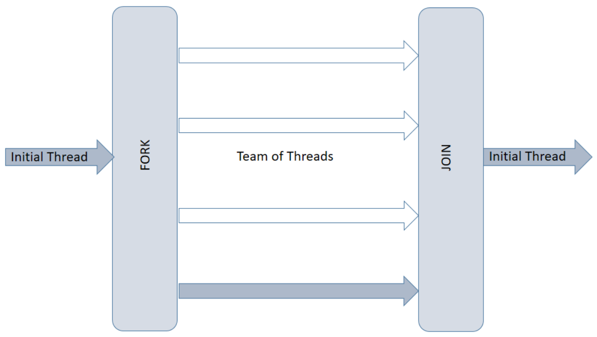


Figure 1: Fork Join model

OpenMP has various types of work-sharing constructs to share the work of the parallel region among threads [10]. The parallel loop construct distributes the iterations of one or more nested loops into smaller chunks and assigns it to the threads in the team. At the end of the loop construct an implicit barrier synchronizes the threads (the barrier can be removed by using the `nowait` construct). The splitting of the iterations among threads depends on the schedule used in the loop construct [10]. In a *Static* schedule, the iteration blocks are mapped to executing threads in a round robin fashion. In a *Dynamic* schedule the mapping of iteration blocks to threads happens on a “first come, first serve" basis. Choosing between static and dynamic scheduling depends on factors such as the specific requirements based on the algorithm and work-load balancing. If each iteration takes a fairly even time to complete, static scheduling is preferred whereas if some iterations take much more or much less time than the mean time for all iterations, static scheduling may cause a load imbalance and dynamic scheduling may give better results.

1.4.2 Message-Passing Interface (MPI)

Message-Passing Interface (MPI) is a standard and aims to be a portable message-passing system developed for distributed computing. There are multiple popular implementations of the MPI standard, namely, OpenMPI, MPICH and the Intel MPI etc. The MPI interface allows one to create parallel programs by specifying how data is exchanged between sending and receiving processes.

MPI communication can be classified as Point-to-Point or Collective communication [11]. `MPI_Send` and `MPI_Recv` are examples of blocking point-to-point communication. Collective communication involves the participation of all processes in a communicator where a communicator is a group of MPI processes. The default communicator containing all the processes is called `MPI_COMM_WORLD`. Some examples of such collective communication routines are `MPI_Bcast`, in which one process sends the same data to all the processes in a communicator. This is what is knows as a Broadcast. `MPI_Scatter` is a collective routine where a root process sends a part of the data in the sending buffer to all other processes in a particular communicator. Whereas broadcasting involves sending the same information to different communicators, `MPI_Scatter` sends chunks of a given array to different processes. `MPI_Gather` is the inverse of `MPI_Scatter` where one process receives data from all other processes in a communicator. This can be visualised as shown in Figure 2.

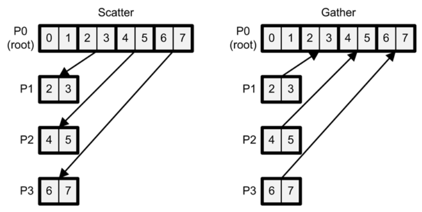


Figure 2: MPI Scatter and Gather operations

1.5 Laws of Scaling

In parallel computing or High Performance Computing (HPC), we aim to solve problems of large sizes which cannot be solved on a Desktop computer. For this we use many supercomputer nodes, which themselves have multiple cores. The aim is to minimize the time to solution for the given problem on a particular architecture. The reduction in the time to solution when an increasing number of Processing Elements (PEs) or nodes or cores are utilized leads to the concept of Scalability. Scalability is for a constant problem size is measured in terms of Speedup, which is defined as:

$$S_p = \frac{T_1}{T_p} \tag{10}$$

where, T_1 is time taken to run a program on a single core and T_p is the time taken to run the program in parallel on p cores. In an ideal scenario we wish to achieve a linear Speedup for all problem sizes but this is rarely achieved as most problems are not embarrassingly parallel.

1.5.1 Amdahl's law and Strong Scaling

Amdahl's law [18] states that the Speedup is limited by the serial fraction of the code. Let F_s be the serial fraction of the given code and $F_p = 1 - F_s$ be the parallel fraction. If we assume that we can *perfectly* parallelise the parallel fraction so that the total execution time on P processors is

$$T_p = T_1(F_s + \frac{F_p}{P}) \tag{11}$$

Using the definition of Speedup, we can write

$$S_p = \frac{1}{F_s + \frac{F_p}{P}} \tag{12}$$

If we increase the number of cores to a very large value, the equation above changes to

$$S_\infty = \frac{1}{F_s} \tag{13}$$

Therefore, the maximum Speedup obtained for a certain problem size is limited by the serial fraction of the code. The type of parallel scaling where we fix the problem size and add more compute resources is termed Strong scaling.

1.5.2 Gustafson's Law and Weak Scaling

Now, instead of a fixed problem size, we look at what happens when we increase the size of the problem with the number of cores. Gustafson [19] proposed that the serial part of the code does not increase as we increase the problem size and thus, it does not affect the scaling. This is a different point of view at looking at measuring scaling. We begin with a parallel program and see how long it would take to execute it in serial.

$$T_1 = T_p F_s + P F_p T_p \tag{14}$$

Plugging in the definition of Speedup, we obtain

$$S_p = F_s + P F_p = P + (1 - P) F_s \tag{15}$$

This kind of scaling is called Weak Scaling where we fix the problem size per process and look at how the Speedup is affected when we increase the number of cores.

1.6 Outline

The thesis begins with the description of the equations which are solved by BioFVM solver. We look at the PDE, the domain over which it is described and the final form which results in a linear system of equations. We then look at the Cyclic Reduction Algorithm in serial and the OpenMP parallel version of the same. The modified Thomas Algorithm which can be solved in parallel with distributed memory is also discussed in this chapter.

Next, we look at the results of the experiments performed after implementing these methods. We are looking at the Speedup achieved, the efficiency of each algorithm and also the scaling. The OpenMP parallel version of CR is compared to the existing serial Thomas Algorithm. For the MPI parallel version of Thomas Algorithm, we explore the limits of scaling beyond which adding more cores is redundant.

2 Algorithm Description

2.1 The equations solved by BioFVM

BioFVM is a standalone solver for Partial Differential Equations (PDE) simulating diffusion uptake and secretion of several substrates in 2D and 3D domains. Physicell uses BioFVM to simulate the chemical microenvironment with a vector of reaction-diffusion PDEs using both bulk and cell-centered sources and sinks.

$$\frac{\partial \boldsymbol{\rho}}{\partial t} = \boldsymbol{D}\nabla^2\boldsymbol{\rho} - \lambda\boldsymbol{\rho} + S(\boldsymbol{\rho}^* - \boldsymbol{\rho}) - U\boldsymbol{\rho} + \Phi \tag{16}$$

$$(\boldsymbol{D}.\nabla\boldsymbol{\rho}).n = 0 \qquad\qquad \text{on } \partial\Omega$$

$$\boldsymbol{\rho}(x, t_0) = \boldsymbol{g} \qquad\qquad \text{in } \Omega$$

In Equation 16, $\boldsymbol{\rho}$ is the vector of densities (i.e. the different substrates or molecules), Φ represents the sources and the uptakes by the cell, $\boldsymbol{\rho}^*$ is the vector of constant saturation densities (the densities at which the cells stop secreting). Likewise, $\mathbf{D}$ and λ are vectors of constant diffusion coefficients and decay rates. $\mathbf{S}$ is the the bulk supply rate, and $\mathbf{U}$ is the bulk uptake function. All vector-vector products $\mathbf{ab}$ are element-wise products (Hadamard Product). BioFVM solves Equation 16 by discretizting the domain and then using the Finite Volume Method.

2.1.1 Domain discretization and notation

The domain in interest is discretized to solve the PDEs, this can be done in two dimensional and three dimensional problems. The discretization is explained further. Let $\{\Omega_i\}_{i=0}^N$ be a set of voxels satisfying $\bigcup_{i=0}^N \Omega_i = \Omega$, with centroids $\{\mathbf{x}_i\}_{i=0}^N$, volumes $\{V_i = |\Omega_i|\}_{i=0}^N$, and boundaries $\{\Sigma_i = \partial\Omega_i\}_{i=0}^N$. For each voxel i, let N_i be the index set of neighboring voxels. For any voxel i and for each $j \in N_i$, let $\Sigma_{ij} = \Sigma_i \cap \Sigma_j$ be the shared boundary between Ω_i and Ω_j, with centroid $\mathbf{x}_{ij}$, outward normal vector (into Ω_j) $\mathbf{n}_{ij}$, and surface area $S_{ij} = |\Sigma_{ij}|$. Define $\Delta\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$ and $\Delta x_{ij} = |\Delta\mathbf{x}_{ij}|$. Let $\mathbf{D}_{ij} = \mathbf{D}(\mathbf{x}_{ij})$ [3].

For a fixed time step size Δt, let $t_n = t_0 + n\Delta t$. For any voxel Ω_i and any time $t \geq t_0$, define $\mathbf{u}_i^n = \int_{\Omega_i} \boldsymbol{\rho}^n(\mathbf{x})dV$, and denote the mean density $\boldsymbol{\rho}^{\boldsymbol{n}}$ at time t_n in voxel Ω_i by

$$\boldsymbol{\rho}_i^n = \frac{\mathbf{u}_i^n}{V_i} = \frac{\int_{\Omega_i} \boldsymbol{\rho} dV}{V_i}$$

For any other function $\mathbf{f}(\mathbf{x}, t)$ (e.g., bulk sources), denote $\mathbf{f}_i^n = \mathbf{f}(\mathbf{x}_i, t_n)$ for any voxel Ω_i and discretized time t_n.

2.1.2 Operator splitting

Here, we are splitting the overall operator into simpler operators which can them be solved individually [12]. To obtain the solution of $\boldsymbol{\rho}^{n+1}$ from the previous time step $\boldsymbol{\rho}^n$ we use first order operator splitting as shown below.

$$\frac{\boldsymbol{\sigma} - \boldsymbol{\rho^n}}{\Delta t} = \nabla.(\boldsymbol{D}.\nabla\boldsymbol{\sigma}) - \lambda\boldsymbol{\sigma} \tag{17}$$

We then apply the Finite Volume Method (FVM) with the Neumann Boundary conditions and assume a *Voronoi mesh* which is given by, $x_{ij} = \frac{1}{2}(x_i + x_j)$ and $\bar{n}_{ij} = \frac{\boldsymbol{\Delta x_{ij}}}{\Delta x_{ij}}$for the neighbouring cells/voxels i and j. After integrating Equation 17 and applying the divergence theorem we obtain the following form.

$$\frac{1}{\Delta t}\int_{\Omega_i}(\boldsymbol{\sigma}^* - \boldsymbol{\rho^n})dV = \int_{\partial\Omega_i}(\boldsymbol{D}.\nabla\boldsymbol{\sigma}^*)\boldsymbol{.n}ds - \int_{\Omega_i}\lambda\boldsymbol{\sigma}^* dV \tag{18}$$

We now divide by the volume V_i to obtain the implicit discretization of the Finite Volume Method

$$(1 + \Delta t\boldsymbol{\lambda} + \Delta t\sum_{j=1}^{N_i}\frac{S_{ij}}{\Delta x_{ij}V_i}\boldsymbol{D_{ij}}).\boldsymbol{\sigma_i^*} - \Delta t\sum_{j=1}^{N_i}\frac{S_{ij}}{\Delta x_{ij}V_i}\boldsymbol{D_{ij}}.\boldsymbol{\sigma_j^*} = \rho_i^n \tag{19}$$

which indicates, a large, sparse linear system.

2.2 The Cyclic Reduction Algorithm

Cyclic Reduction (CR) algorithm [7], is used to solve a tridiagonal system of equations. This algorithm is similar to the Gaussian Elimination, but the advantage over the original Thomas Algorithm is that CR can be implemented in parallel. The main idea of the cyclic reduction technique is to group the odd and even numbered entries and then successively eliminate the even numbered unknowns. Thus, reducing the number of equations to half in one step. We then follow the same rule in successive steps until we are left with one equation. We solve for that equation, and use that solution to trace back the other unknowns. This section describes the cyclic reduction algorithm in detail, first we will look at the serial version of the algorithm and then we shall move on to the parallel cyclic reduction.

We shall look at the specific steps involved with the help of a small example. Consider a tridiagonal system of equations i.e. :

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = F_i \qquad\qquad i = 1...n \tag{20}$$

here a_i, b_i, c_i and F_i are given and $n = 2^{p-1}$ here p is an integer, if n does not satisfy that condition we can add trivial equations such as $x_i = 1, i = n+1, ...2^p - 1$

In the first step we are looking to eliminate the odd numbered unknowns $x_1, x_3, x_5...x_n$. We will then rewrite the equations and renumber the unknowns. In the steps that follow, the renumbered odd-unknowns are eliminated. The aforementioned procedure is repeated until we get one equation and one unknown. The solution of the last equation is substituted in the equations from which the last equation was obtained to find the remaining unknowns. To generalize, we aim to eliminate unknowns, x_{i-1} and x_{i+1} from equation i by utilizing equations $i-1$ and $i+1$. We combine form of equations from system in 20 as triplets [20].

Consider an example with $n = 2^3 - 1 = 7$. We take the first three equations and multiply them with α_2, β_2, and γ_2 respectively to get

$$\begin{aligned}\alpha_2 b_1 x_1 + &\alpha_2 c_1 x_2 = \alpha_2 F_1\\ \beta_2 a_2 x_1 + &\beta_2 b_2 x_2 + \beta_2 c_2 x_3 = \beta_2 F_2\\ &\gamma_2 a_3 x_2 + \gamma_2 b_3 x_3 + \gamma_2 c_3 x_4 = \gamma_2 F_3\end{aligned} \tag{21}$$

We aim to eliminate x_1 and x_3 from the equations hence we choose $\beta_2 = 1$, $\alpha_2 b_1 + \beta_2 a_2 = 0$ and $\beta_2 c_1 + \gamma_2 b_3 = 0$.

Adding the first three equations, we get

$$\underbrace{(\alpha_2 c_1 + \beta_2 b_2 + \gamma_2 a_3)}_{\hat{b}_2} x_2 + \underbrace{\gamma_2 c_3}_{\hat{c}_2} x_4 = \underbrace{\alpha_2 F_1 + \beta_2 F_2 + \gamma_2 F_3}_{\hat{F}_2} \ . \tag{22}$$

To form the second triplet we combine equations 3, 4 and 5.

$$\underbrace{\alpha_4 a_3}_{\hat{a}_4} x_2 + \underbrace{(\alpha_4 c_3 + \beta_4 b_4 + \gamma_4 a_5)}_{\hat{b}_4} x_4 + \underbrace{\gamma_4 c_5}_{\hat{c}_4} x_6 = \underbrace{a_4 F_3 + \beta_4 F_4 + \gamma_4 F_5}_{\hat{F}_4}, \tag{23}$$

Again, we choose $\beta_4 = 1$, $\alpha_4 b_3 + \beta_4 a_4 = 0$ and $\beta_2 c_1 + \gamma_2 b_3 = 0$

For the third and last triplet we obtain,

$$\hat{a}_6 x_4 + \hat{b}_6 x_6 = \hat{F}_6 \tag{24}$$

and for this triplet, we have chosen $\beta_6 = 1$, $\alpha_6 b_5 + \beta_6 a_6 = 0$, and $\beta_6 c_6 + \gamma_b b_7 = 0$.

The three resulting equations 22, 23, and 24 also form a tridiagonal system of equations as shown below.

$$\begin{aligned}\hat{b}_2 x_2 + &\hat{c}_2 x_4 = \hat{F}_2\\ \hat{a}_4 x_2 + &\hat{b}_4 x_4 + \hat{c}_4 x_6 = \hat{F}_4\\ &\hat{a}_6 x_4 + \hat{b}_6 x_6 = \hat{F}_6\end{aligned} \tag{25}$$

We repeat the procedure mentioned above to get one final equation

$$\alpha_4^* x_4 = F_4^* \tag{26}$$

The value of x_4 from Equation 26, can be substituted in Equation(s) 25 to determine the unknowns x_2 and x_6. The odd unknowns can be subsequently found using 21

. Figure 3 illustrates the CR algorithm for the case of $N = 7$. Each column of circles in Figure 3 can be considered as a level of the problem to be solved. Thus, when $N = 7$, the Cyclic Reduction algorithm can be visualized as having 3 levels. This is the worst case scenario, as there are cases where we will find the solution much earlier, before all the steps are completed.

Algorithm 1: Cyclic Reduction

```
Input  : matrix size: N, Vectors $a[]$, $b[]$, $c[]$ representing the sub-diagonal,
         main-Diagonal and super-Diagonal of the matrix A and the forcing
         vector $F[]$, $i = 0, 1, ...N - 1$
Output: row vector $x[]$ here $i = 0, 1, 2, ..., N - 1$

 Forward Reduction for $i = 0$ to $log_2(N + 1)$ - 1 do
    for $j = 2^{i+1} - 1$ to $N - 1$, $j = j + 2^{i+1}$ do
        offset ⟵ $2^i$;
        id1 ⟵ $j-$offset;
        id2 ⟵ $j+$offset;
        alpha ⟵ $a[i]$ / $b[i]$;
        beta ⟵ $c[i]$ / $b[i]$;
        a[j] ⟵ -a[id1]*alpha;
        b[j] ⟵ b[j] - c[id1]*alpha - a[id2]*gamma;
        c[j] ⟵ -c[id1]*gamma;
    end
end
id ⟵ $(N - 1)$ / 2;
x[id] ⟵ $F[id]$ / $b[id]$;
Backward substitution for $i = log_2(N + 1) - 2$ to 0, $i - -$ do
    for $j = 2^{i+1} - 1$ to $N - 1$, $j = j + 2^{i+1}$ do
        offset ⟵ $2^i$;
        id1 ⟵ $j-$offset;
        id2 ⟵ $j+$offset;
        if id1 - offset < 0 then
            x[id1]⟵(F[id1]-c[id1]*x[id1+offset])/b[id1];
        end
        else
            x[id1] ⟵ (F[id1] - a[id1]*x[id1-offset] - c[id1]*x[id1+offset])/b[id1];
        end
        if id2 + offset $\geq N$ then
            x[id2]⟵(F[id2]-a[id1]*x[id2-offset])/b[id2];
        end
        else
            x[id2] ⟵ (F[id2] - a[id2]*x[id2-offset] - c[id2]*x[id2+offset])/b[id2];
        end
    end
end
```

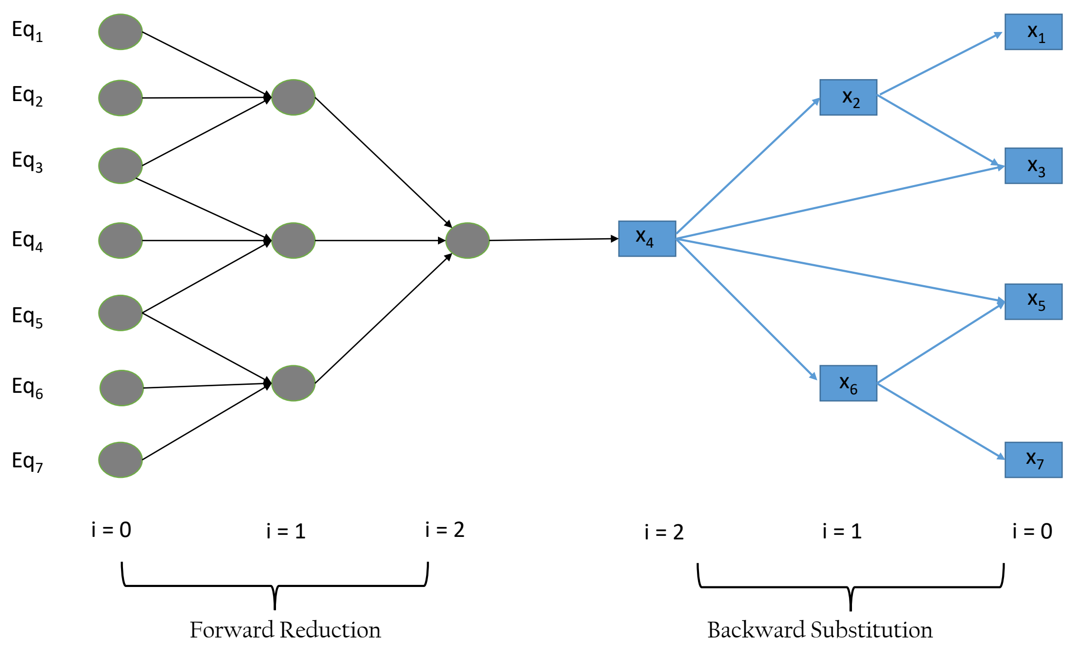


Figure 3: Illustration of the Cyclic Reduction algorithm for $N = 7$ unknowns

2.3 OpenMP version of Cyclic Reduction

We now describe the shared memory parallelization of the Cyclic Reduction algorithm using OpenMP (Open Multiprocessing). For the two stages of cyclic reduction we distribute the work i.e. $W = N/p$ equations among threads, where N is the size of the system we want to solve, and p is the number of threads. This can be seen in Algorithm 1, we use three arrays to store the tridiagonal matrix, an array for the forcing vector and one array to store the solutions. All arrays and the system size N are *shared* among the threads to satisfy the data dependencies.

In Figure 4, on the left, we illustrate how the data is accessed in the Forward Reduction stage for the case of an 7-equation system. If we have $p = 2$, we will have each thread handling four equations. The equations are mapped consecutively to the threads as we use a *static scheduling*. An implicit *barrier* at the end of each step keeps the reduction stage synchronized. In the same figure, the blue squares indicate how the threads access the data in the Backward Substitution stage, where the work is shared among the threads in the same way as in the Forward Reduction stage.

Looking closely at Algorithm 1, we can notice that both Forward Reduction and Backward Substitution phases have two nested loops. The outer for loop changes the level at which the calculations are taking place and these levels are denoted by dashed boxes in Figure 4. The inner loop moves vertically on a particular 'level'. Thus, for a 'loop-level' parallel construct such as `#pragma omp parallel for` in C++, we must focus on the inner loops which traversed vertically, for both Forward Reduction and Backward Substitution.

The placement of the OpenMP parallel for is illustrated in Listing 1

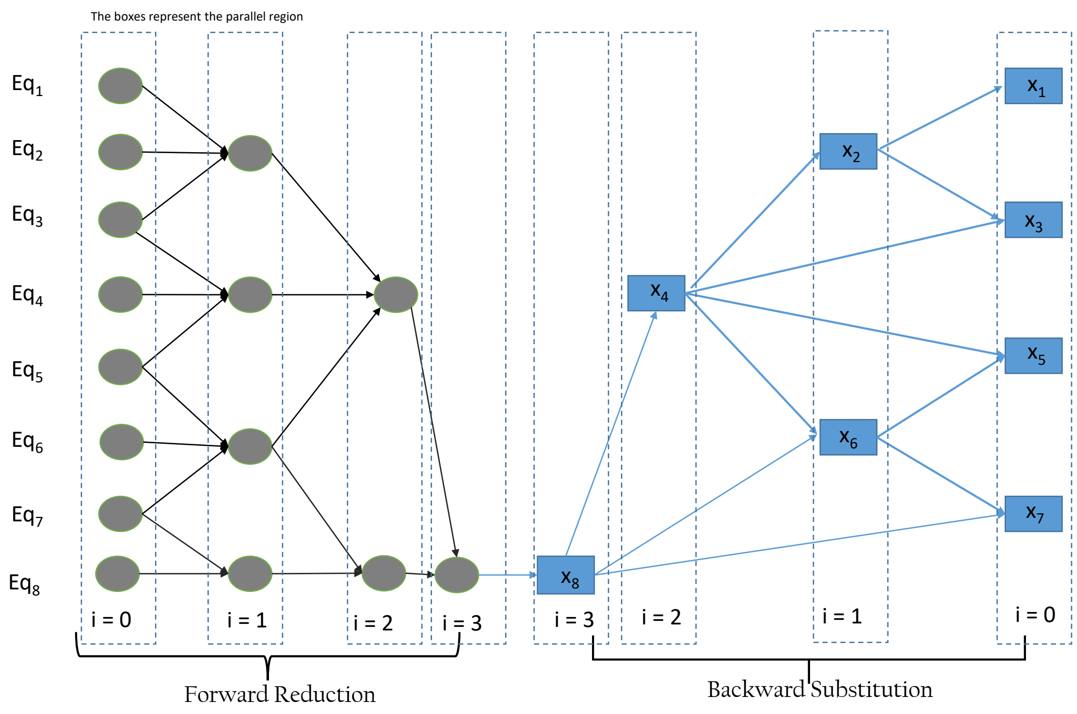


Figure 4: Work shared among the threads in the Forward Reduction and Backward Substitution stage of cyclic reduction for an $N = 7$ equation system.

Listing 1: OpenMP implementation of Cyclic Reduction. The complete code can be found at this Github page

```
//Forward Reduction
for(i=0;i<logSize;i++){
    int step = pow(2,i+1);
    #pragma omp parallel shared (a, c, b, F, size) private(j,index1, index2,
        alpha, gamma)
    {
    #pragma omp for
    for(j=pow(2,i+1)-1;j<size;j=j+ step){
    // code
    }
}
}
// Backward Substitution
for(i=log2(size+1)-2;i>=0;i--){
    int step = pow(2,i+1);
    #pragma omp parallel shared(x,F,a,c,b, size) private(j,index1, index2,
        alpha, gamma)
    {
    #pragma omp  for
    for(j=pow(2,i+1)-1;j<size;j=j+ step){
    //code
        .
}
}
}
```

2.4 Modified Thomas Algorithm

The standard Thomas algorithm has already been described in the previous section. The pseudo-code for the Thomas algorithm is given by Algorithm 2.

Algorithm 2: Thomas Algorithm

```
Input  : matrix size: N, Vectors $a[]$, $b[]$, $c[]$ representing the sub-diagonal,
         main-Diagonal and super-Diagonal of the matrix A and the forcing
         vector $d[]$ here $i = 0, 1, ..., N - 1$
Output: row vector $d[i]$ here $i = 0, 1, 2, ..., N - 1$

$d_0^* \longleftarrow d_0 \,/\, b_0$
$c_0^* \longleftarrow c_0 \,/\, b_0$
for $i = 1, 2...N - 1$ do
    $r \longleftarrow 1 \,/\, (b_i - a_i c_{i-1})$
    $d_i^* \longleftarrow r(d_i - a_i d_{i-1})$
    $c_i^* \longleftarrow r c_i$
end
for $i = (N - 2)...1$ do
    $d_i \longleftarrow d_i^* - c_i^* d_{i+1}$
end
```

here, the RHS vector d is overwritten with the solution.

Studies [14] [15] have discussed parallel methods to partition large tridiagonal systems of equations into subsystems. In this current work, we study a modified algorithm developed by László et al [16] and further improved by Kim et al [17]. The algorithm implemented, can be described in the following steps

1. Every core transforms the partitioned sub-matrices in the tridiagonal systems of equations into the modified forms by applying the modified Thomas algorithm.
2. We construct a reduced tridiagonal system of equations by collecting the first and last row of every modified sub-matrix. This is done using `MPI_Gather` MPI collective communication
3. The reduced tridiagonal system constructed in step 2 is solved using the Thomas Algorithm, see Algorithm 2.
4. The solutions of reduced tridiagonal systems in Step 3 are distributed to each core using the `MPI_Scatter` MPI collective communication call.
5. Finally, we solve for the remaining unknowns of the modified sub-matrices (Step 1) by using the solutions obtained in Steps 3 and 4.

Consider a tri-diagonal system of equations of a given size N. We divide this system among P cores such that each core stores a system of size m ($m = N/P$). Each core has an index j such that $0 \leq j \leq P - 1$. The matrix Equation 28 shows an $N = 12$ variable system divided among three cores ($P = 3$) such that the number of rows per core are $m = 4$.

In Step 1, the partitioned tridiagonal system is transformed into a system with modified sub-matrices. This is done by the modified Thomas Algorithm this is presented in the step

1 of Algorithm 3. The tridiagonal system for each core can be represented by the following equation:

$$a_i^j x_{i-1}^j + b_i^j x_i^j + c_i^j x_{i+1}^j = F_i^j \tag{27}$$

,where $x_0^j = x_m^{j-1}$ for $j \geq 1$ and $x_{m+1}^j = x_1^{j+1}$ for $j \leq P-1$. The modified Thomas algorithm begins with the elimination of the lower diagonal elements from the third row instead of the second row as in the standard Thomas Algorithm. Thus, the tridiagonal system is transformed into a system in which the first element of all rows is non-zero. This is shown in Equation 29. All the main diagonal elements are normalized to 1. Then, for the upper part, the upper diagonal elements are eliminated from $i = (m-2)$, resulting in non-zero values in the last column.

$$\left(\begin{array}{cccc|cccc|cccc}
b_1 & c_1 & & & & & & & & & & \\
a_2 & b_2 & c_2 & & & & & & & & & \\
 & a_3 & b_3 & c_3 & & & & & & & & \\
 & & a_4 & b_4 & c_4 & & & & & & & \\
\hline
 & & & a_5 & b_5 & c_5 & & & & & & \\
 & & & & a_6 & b_6 & c_6 & & & & & \\
 & & & & & a_7 & b_7 & c_7 & & & & \\
 & & & & & & a_8 & b_8 & c_8 & & & \\
\hline
 & & & & & & & a_9 & b_9 & c_9 & & \\
 & & & & & & & & a_{10} & b_{10} & c_{10} & \\
 & & & & & & & & & a_{11} & b_{11} & c_{11} \\
 & & & & & & & & & & a_{12} & b_{12}
\end{array}\right)
\left(\begin{array}{c}
u_1 \\ u_2 \\ u_3 \\ u_4 \\ \hline u_5 \\ u_6 \\ u_7 \\ u_8 \\ \hline u_9 \\ u_{10} \\ u_{11} \\ u_{12}
\end{array}\right)
=
\left(\begin{array}{c}
d_1 \\ d_2 \\ d_3 \\ d_4 \\ \hline d_5 \\ d_6 \\ d_7 \\ d_8 \\ \hline d_9 \\ d_{10} \\ d_{11} \\ d_{12}
\end{array}\right) \tag{28}$$

$$\left(\begin{array}{cccc|cccc|cccc}
\mathbf{1} & & & \mathbf{c_1^*} & & & & & & & & \\
a_2^* & 1 & & c_2^* & & & & & & & & \\
a_3^* & & 1 & c_3^* & & & & & & & & \\
\mathbf{a_4^*} & & & \mathbf{1} & \mathbf{c_4^*} & & & & & & & \\
\hline
 & & & \mathbf{a_5^*} & \mathbf{1} & & & \mathbf{c_5^*} & & & & \\
 & & & & a_6^* & 1 & & c_6^* & & & & \\
 & & & & a_7^* & & 1 & c_7^* & & & & \\
 & & & & \mathbf{a_8^*} & & & \mathbf{1} & \mathbf{c_8^*} & & & \\
\hline
 & & & & & & & \mathbf{a_9^*} & \mathbf{1} & & & \mathbf{c_9^*} \\
 & & & & & & & & a_{10}^* & 1 & & c_{10}^* \\
 & & & & & & & & a_{11}^* & & 1 & c_{11}^* \\
 & & & & & & & & \mathbf{a_{12}^*} & & & \mathbf{1}
\end{array}\right)
\left(\begin{array}{c}
\mathbf{u_1} \\ u_2 \\ u_3 \\ \mathbf{u_4} \\ \hline \mathbf{u_5} \\ u_6 \\ u_7 \\ \mathbf{u_8} \\ \hline \mathbf{u_9} \\ u_{10} \\ u_{11} \\ \mathbf{u_{12}}
\end{array}\right)
=
\left(\begin{array}{c}
\mathbf{d_1^*} \\ d_2^* \\ d_3^* \\ \mathbf{d_4^*} \\ \hline \mathbf{d_5^*} \\ d_6^* \\ d_7^* \\ \mathbf{d_8^*} \\ \hline \mathbf{d_9^*} \\ d_{10}^* \\ d_{11}^* \\ \mathbf{d_{12}^*}
\end{array}\right) \tag{29}$$

$$\left(\begin{array}{cccccc}
1 & c_1^{*j-1} & & & & \\
a_4^{*j-1} & 1 & c_4^{*j-1} & & & \\
\hline
 & a_1^{*j} & 1 & c_1^{*j} & & \\
 & & a_4^{*j} & 1 & c_4^{*j} & \\
\hline
 & & & a_1^{*j+1} & 1 & c_1^{*j+1} \\
 & & & & a_4^{*j+1} & 1
\end{array}\right)
\left(\begin{array}{c}
u_1^{j-1} \\ u_4^{j-1} \\ \hline u_1^j \\ u_4^j \\ \hline u_1^{j+1} \\ u_4^{j+1}
\end{array}\right)
=
\left(\begin{array}{c}
d_1^{*j-1} \\ d_4^{*j-1} \\ \hline d_1^{*j} \\ d_4^{*j} \\ \hline d_1^{*j+1} \\ d_4^{*j+1}
\end{array}\right) \tag{30}$$

The modified pair of equations from every core are denoted with a super-script (asterisk). For the reduction phase, we collect all the modified equations from each core, these are first and last equations, denoted by $i = 1$ and $i = m$. These first and last modified equations from each core are combined as shown in Equation 30. The system of equations obtained is solved using the standard Thomas Algorithm. Notice for this we have reduced the size of the system by almost half for this particular example, as we look at larger values of N, we expect a considerable reduction in the number of variables that need to solved in the reduced system of equation. Another thing to note is that the matrix in 30 is still Symmetric, Positive Definite and tridiagonal just like the original system in Equation 28.

Finally, after we have obtained the solution for the reduced system see Equation 30. We send it back to the cores and obtain the other solutions. This is summarised in Algorithm 3.

Algorithm 3: MPI Parallel version of Thomas Algorithm

```
**Input :** matrix size: N, Vectors $a[]$, $b[]$, $c[]$ representing the sub-diagonal,
         main-Diagonal and super-Diagonal of the matrix A and the forcing
         vector $d[]$ here $i = 1, ..., N$
**Output:** row vector $d[i]$ here $i = 1, 2, ..., N$

**Step 1 : The Modified Thomas Algorithm**
*This step takes place at a particular core j such that* $0 \le j \le P - 1$
$d_1^* \longleftarrow d_1 / b_1$; $c_1^* \longleftarrow c_1 / b_1$; $a_1^* \longleftarrow a_1 / b_1$
$d_2^* \longleftarrow d_2 / b_2$; $c_2^* \longleftarrow c_2 / b_2$; $a_2^* \longleftarrow a_2 / b_2$
**for** $i = 3...m$ **do**
    $r \longleftarrow 1 / (b_i - a_i c_{i-1})$
    $d_i \longleftarrow r(d_i - a_i d_{i-1)}$
    $c_i \longleftarrow r c_i$
    $a_i \longleftarrow -r(a_i a_{i-1})$
**end**
**for** $i = (m - 2)...2$ **do**
    $d_i \longleftarrow d_i - c_i d_{i+1}$
    $c_i \longleftarrow -c_i c_{i+1}$
    $a_i \longleftarrow a_i - c_i a_{i+1}$
**end**
$d_1 \longleftarrow r(d_i - a_i d_{i-1})$ ; $c_1 \longleftarrow -r c_1 c_2$ ; $a_1 \longleftarrow r a_1$

**Step 2: Forward Communication**
Collect coefficients $a_i$, $b_i$, $c_i$ and $d_i$ for $i = 1$ and $m$ from each core

**Step 3 : Solving the reduced system**
Obtain The solution for the reduced system with the help of the Thomas Algorithm

**Step 4 : Backward Communication**
Distribute the solution of the reduced system, $d_i$ for $i = 1$ and $m$ back to each core

**Step 5: Update the other solutions**
**for** $i = 2...m - 1$ **do**
    $d_i \longleftarrow d_i - a_i d_1 - c_i d_m$
**end**
```

3 Results

3.1 Experimental Testbed

We conduct our experiments on the MareNostrum 4 (MN4) supercomputer cluster located at and maintained by the Barcelona Supercomputing Center (BSC). MN4 has a total of 3456 nodes. Each node has two 24-core Intel Xeon Platinum 8160 Skylake generation processors (sockets) and a total working memory of 96 GB (per node). The underlying Operating System (OS) is the SUSE Linux Enterprise Server 12 SP2 OS that also supports the IBM General Parallel File System (GPFS). The compute nodes are interconnected with the Intel Omni-Path technology (OPA) having a maximum bandwidth of 100 Gbits/sec. We use gcc/8.1.0 as the compiler and impi/2017.4 (Intel MPI 2017.4) as the MPI implementation for all our experiments. Further, each simulation is carried out three times and the average run time is considered.

3.2 Performance Measurement

We express the performance of the program as a function of the total run-time and, as and when deemed appropriate, using the Speedup. The Speedup ($S_p(n)$) of a parallel program can using p cores and for a constant problem of size n is shown below.

$$S_p(n) = \frac{T_{serial}}{T_{parallel}} \tag{31}$$

The Efficiency ($E_p(n)$) of the parallel program using p cores and at a problem of size n can be expressed as follows:

$$E_p(n) = \frac{S_p(n)}{p} \tag{32}$$

The Efficiency expresses the processor utilization of a parallel program. In other words, it measures the fraction of time for which a processor is usefully exploited. Since the minimum Speedup for a parallel program using p cores can be one i.e. the program is completely serial and the maximum is p (ideal Speedup), $\frac{1}{p} \leq E_p(n) \leq 1$.

3.3 Example Problem

Our first aim is to check the correctness of our Cyclic Reduction (CR) algorithm implmentation by solving a specific but standard problem. Thus, we consider a a 1D Poisson problem and use the Finite Difference Method (FDM) to discretize the PDE to obtain a tridiagonal system of linear equations. Consider the Poisson equation,

$$-\nabla^2\phi(x) = \rho(x) \qquad\qquad x \in [a,b] \subset \ R \tag{33}$$

, with the given boundary conditions as $\phi(a) = \alpha$ and $\phi(b) = \beta$. We discretize the domain with $N+1$ equidistant points where $h = \frac{b-a}{N}$ is the distance between two neigbouring points. We set $a = 0$ and $b = 2\pi$. Using the Central Difference approximation in Equation 33 to convert the derivatives to differences we obtain:

$$-\frac{\phi(i+1) - 2\phi(i) + \phi(i-1)}{h^2} = \rho(i) \qquad\qquad i = 1, 2, ..., N \tag{34}$$

The system of equations represented by Equation 34 results in a tridiagonal system of linear equations and can be solved using the CR algorithm. Assuming $\rho(x) = 2\sin(x) + x\cos(x)$, the exact solution for $\phi(x)$ is given by $\phi(x) = x\cos(x)$. The numerical solution obtained from the CR algorithm implementation is compared to the exact solution and we show both the solutions in figure 5. For a problem of size $N = 64$ i.e. 64 unknowns, the Mean Square

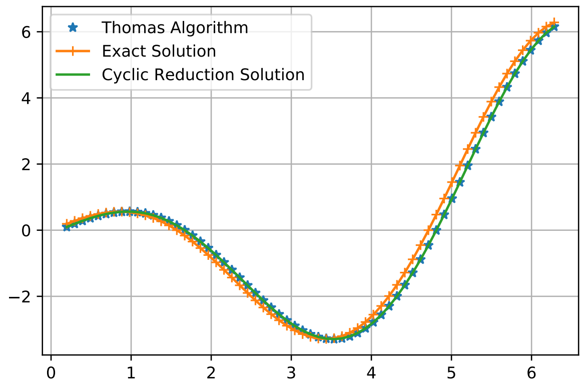


Figure 5: The comparison of the exact against Thomas Algorithm solution and Cyclic Reduction algorithm solution

Error (MSE) obtained is equal to 0.07263626 (see Figure 5). This error can be attributed to the Finite Difference Method as $h = 0.0982$ is not sufficiently small but the error between the solution obtained from the Thomas solver and the CR is computed to be 3.074×10^{-28}.

3.4 Parallel implementation of the Cyclic Reduction algorithm

This section summarises the results obtained for the OpenMP parallel version of the Cyclic Reduction algorithm. As described in the previous section, we discretize the Poisson problem using FDM to obtain a Tridiagonal system of Equations of size N i.e. the matrix A of the matrix equation $Ax = b$ is of the size $N \times N$. In BioFVM/PhysiCell, the linear solver is called multiple times in the execution of a particular simulation. Owing to the Locally One Dimensional (LoD) formulation adopted by PhysiCell if we have N voxels (Volumetric Pixels which are equivalent to N unknowns) in the X direction and N (Volumetric Pixels which are equivalent to N unknowns) voxels in the Y direction then we solve an $N \times N$ system of equations N^2 times. It is not necessary to have an equal number of voxels in all directions. In this study a particular $N \times N$ system of equations is solved N times to test its performance.

In our experiments, the size of the matrix N increases from 500^2 to 128000^2. The serial execution of the algorithm is taken as the base case and the number of threads (t) are varied from $t = 2$ to 48 (maximum cores per node in MN4). The different problem sizes and the number of threads that the problem is evaluated at are shown in Table 1. It can further be noted that in all the experiments, we run these threads on a single core.

Problem Size (N)	Number of Threads (t)
500	1,2,4,8,16,24,32,48
1000	1,2,4,8,16,24,32,48
2000	1,2,4,8,16,24,32,48
4000	1,2,4,8,16,24,32,48
8000	1,2,4,8,16,24,32,48
16000	1,2,4,8,16,24,32,48
32000	1,2,4,8,16,24,32,48
64000	1,2,4,8,16,24,32,48
128000	1,2,4,8,16,24,32,48

Table 1: Problem size (N) and the number of threads from 1 to 48 . Each problem size on the left has been evaluated at all the the threads on the right.

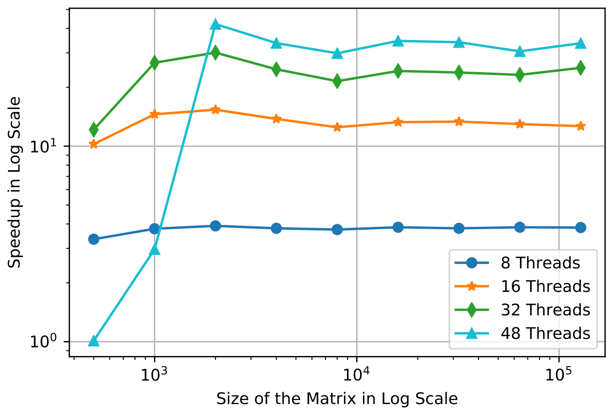


Figure 6: Speedup vs Problem Size in Log-Log Scale

Figure 6, shows the Speedup Vs the size of the matrix (Log scale) for a few chosen number of threads. We observe that as the number of threads increase, the Speedup increases. As more threads perform calculations simultaneously, the computations per thread decrease, resulting in a better Speedup. The highest Speedup is 33.60 which is achieved with 48 threads for the problem size of $N = 128000$.

For a fixed number of threads as the problem size is increased, the Speedup more or less remains constant after an initial increase. In case of small problem sizes the serial Thomas algorithm outperforms the parallel CR algorithm as the amount of work that each thread is assigned is not substantial and some threads remain idle when the level at which the equations are being solved increases (see Figure 3). Thus, for a problem size of $N \leq 1000$, we observe that fewer threads performs better.

Figure 7 displays the variation in parallel efficiency against the number of threads. We observe that the efficiency is almost 100% when the number of threads are ≤ 8 but it drops down to 70% with 48 threads. The two dimensional heat map in Figure 8 attempts to

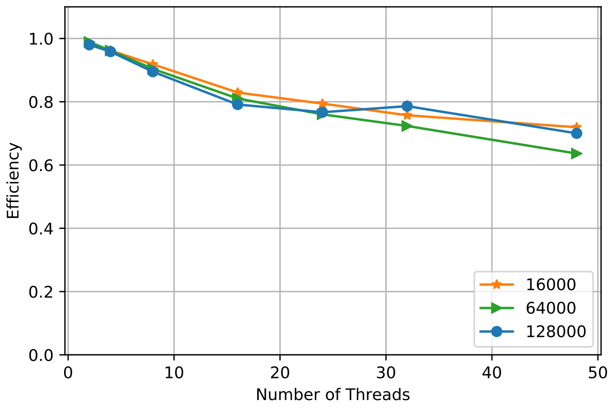


Figure 7: Efficiency vs Number of Threads for problems of sizes $N = 16000, 64000, 128000$

show a clearer picture of how the efficiency drops as we increase the number of threads for a constant problem size. This can be attributed to the increase in cache-contention and a reduced amount of work per-thread. It can be noted that although efficiency decreases with an increasing number of threads for a constant problem size, the time to solution decreases.

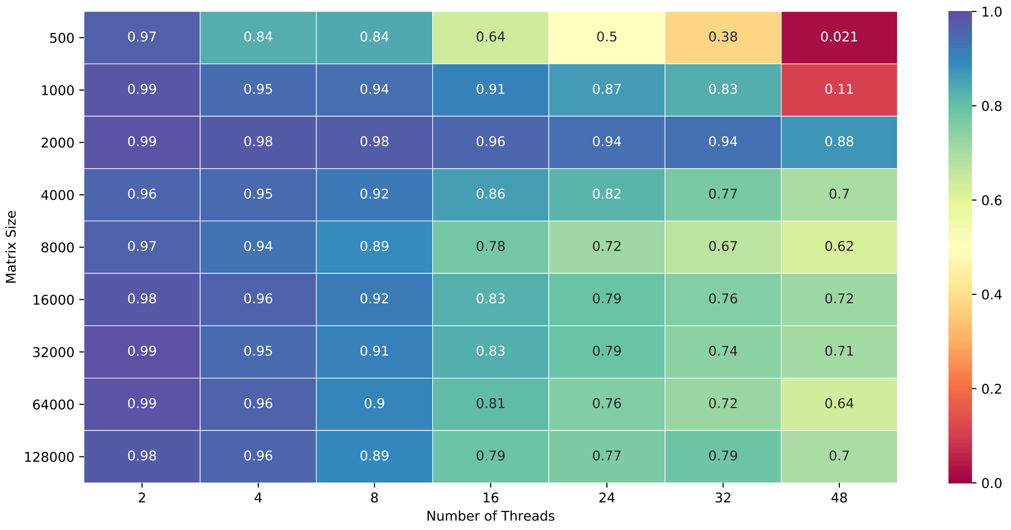
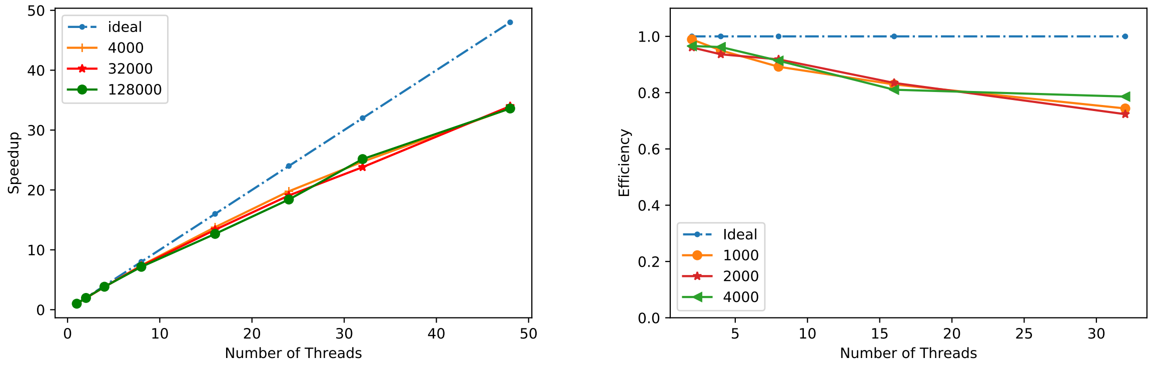


Figure 8: Parallel efficiency for multiple problem sizes executed with increasing thread count

Figure 9, shows the Strong and Weak scaling results for the Parallel CR algorithm. As mentioned earlier for Strong scaling, the total problem size is kept constant and the number of threads is increased [18]. In Weak scaling the problem size relative to the number of threads/ cores is kept constant. In figure 9a, we see that the Speedup begins to deviate from the ideal value when the number of threads is ≥ 8. It can be seen that for the case of 48 threads, we achieve a Speedup of around 33 for all the three cases.

In figure 9b, we look at the change in efficiency when the problem size per thread (N/t)



(a) Strong Scaling (b) Weak Scaling

Figure 9: Scalibility results for the Parallel Cyclic Reduction

is a constant. We achieve the best efficiency when the number of threads is the lowest ($t = 2$) and it decreases as we increase the number of threads. In figure 7, a similar trend was observed for a constant problem size. The drop in efficiency for all the three cases $N/t = 1000$, $N/t = 2000$ and $N/t = 4000$, very similar they start at nearly 100% and drop to around 75%. This drop in efficiency is the inverse of the relative time taken to compute the solution, remember $E \propto 1/T_p$, as the efficiency drops, computation time increases. In an ideal case for a constant problem size per thread we expect the efficiency and the relative time taken to remain a constant as we increase the number of threads, but we observe that there is a deviation from the ideal.

3.5 CR Vs the serial Thomas Algorithm

We now compare our OpenMP implementation of the CR algorithm with the most efficient direct algorithm for solving a tridiagonal system of linear equations in serial i.e. the Thomas algorithm. The serial Thomas solver is faster than the serial version of Cyclic Reduction [20]. From Figure 10, it can be seen that the serial Thomas Algorithm effectively shows similar execution time as the Parallel Cyclic Reduction algorithm with 8 threads but the latter outperforms the former when the number of threads is greater than eight. This result is important as it gives a cut-off to the minimum number of threads that we need to use for the OpenMP version of Cyclic Reduction to achieve better performance than the serial Thomas Algorithm.

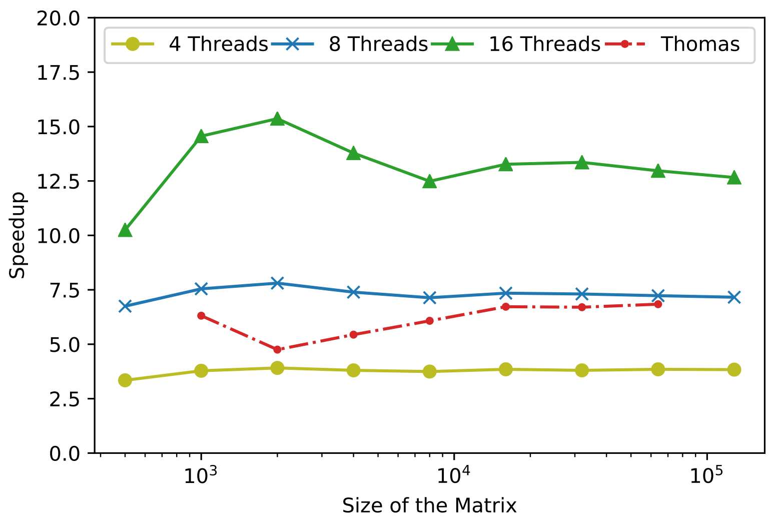


Figure 10: Cyclic Reduction Vs Thomas Solver Speedup

3.6 Modified Thomas Algorithm

The Modified Thomas Algorithm explained in the previous chapter is implemented. The conditions of the tests are similar to those explained in the previous section. The wall clock time for solving systems with increasing sizes is recorded. The problem size increases from 500^2 to 12800^2 and the number of cores used for execution of the distributed memory program is varied from $p = 2$ to $p = 48$. The variation of MPI processes for various problem sizes is shown below in Table 2.

Problem Size	Number of cores
500	1,2,4,8,16,24,32,48
1000	1,2,4,8,16,24,32,48
2000	1,2,4,8,16,24,32,48
4000	1,2,4,8,16,24,32,48
8000	1,2,4,8,16,24,32,48
16000	1,2,4,8,16,24,32,48
32000	1,2,4,8,16,24,32,48
64000	1,2,4,8,16,24,32,48
128000	1,2,4,8,16,24,32,48

Table 2: Problem size (N) and the number of MPI cores. Each problem size on the left has been evaluated at every value of number of cores on the right.

Figure 11 shows the Speedup vs the size of matrix in log scale for three different values of the number of cores. The Speedup increases for all the cases as we increase the number of cores i.e. MPI processes. The algorithm achieves better Speedup at larger matrix sizes as the local computations increase when we solve larger problems. The case for $p = 48$ (48 MPI processes) in Figure 11 performs worse than other cases with less than 48 cores when the matrix size is smaller possibly because the major time is spent in communication

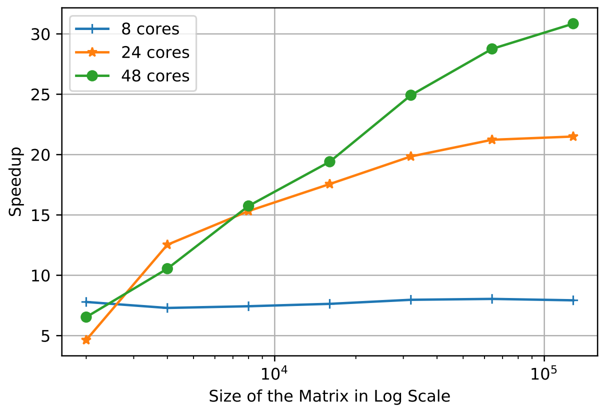


Figure 11: Speedup for different values of Matrix sizes

between cores. The highest Speedup achieved is around 30.6 for 48 cores and the problem size of $N = 12800$.

In figure 12, we examine the Efficiency of the Modified Thomas solver for a few cases. The reader may recall that the efficiency of parallel program is the ratio of the Speedup achieved to the number of cores. Efficiency measures the fraction of time for which a processor is usefully utilized. We can observe that when the number of cores are small, the efficiency is almost 100%. The efficiency drops down from around 1 to 0.4 in case of $N = 16000$. The drop in efficiency is slower when we increase the matrix size as the local work per processor increases.

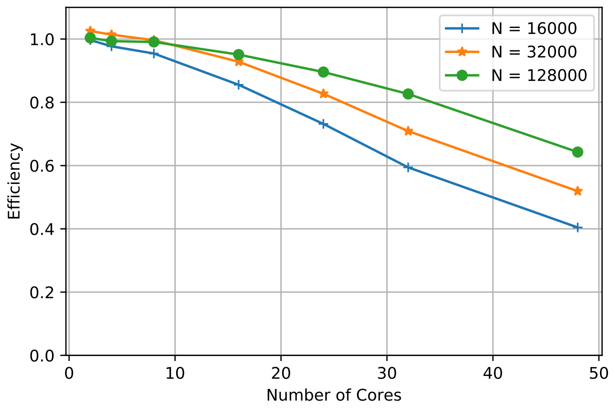


Figure 12: Efficiency at different core count for problems of sizes $N = 16000, 32000, 128000$ in the Modified Thomas Algorithm (MPI-only implementation)

In the global heat-map 13 of the efficiency data for the Modified Thomas Algorithm we see a clear trend that for fewer processor the efficiency is ≈ 1. The efficiency drops as we increase the number of cores in all the cases. However, the drop in efficiency is less gradual as the size of the matrix system increases. For the largest value of $N = 128000^2$, the minimum efficiency recorded is 0.64 for 48 cores.

Figure 14 shows the scalibility of the Modified Thomas Algorithm. We initially keep the

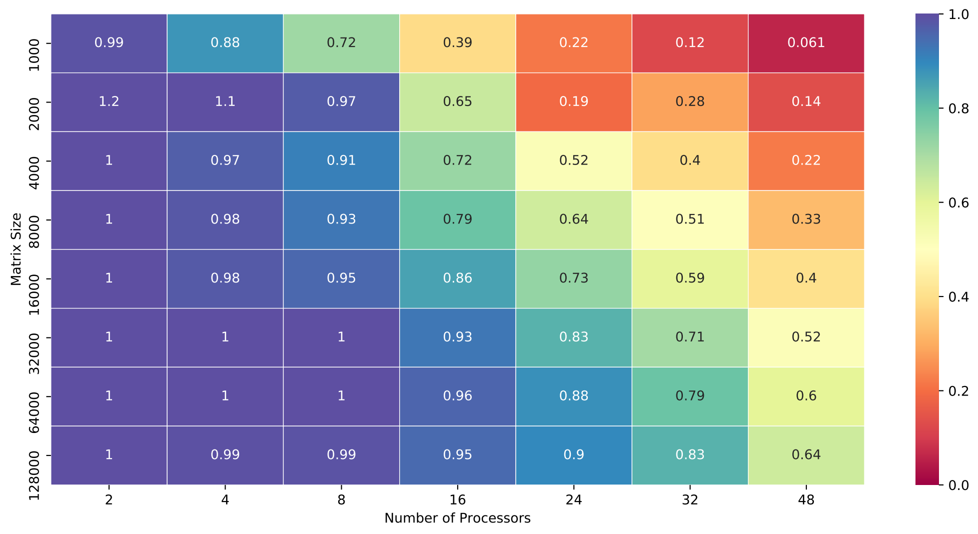
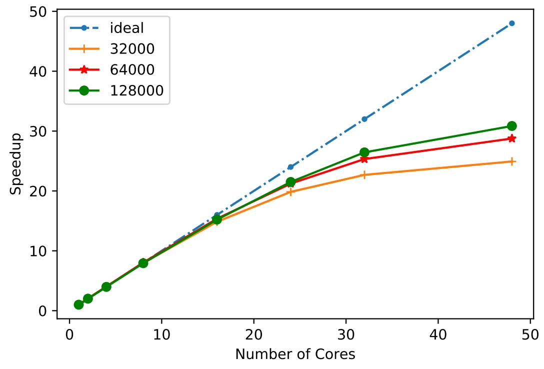


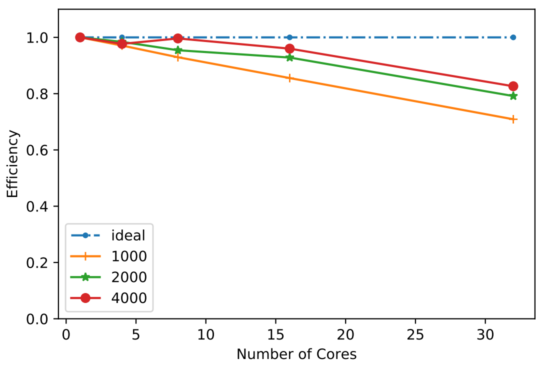
Figure 13: Global heat map of efficiency data

problem size constant and increase the number of MPI processes (see Figure 14a) to examine how the Speedup changes (Strong Scaling). The dotted straight line represents and ideal Speedup. The scaling coincides with the ideal scaling curve for all three cases up to 16 cores and deviates from the ideal as we further increase the number of cores. We also observe that for larger values of problem size N, the deviation from the ideal Speedup is smaller. From this, we can conclude that as we increase the problem size N, the chunk of parallel computation increases faster as compared to the chunk of serial computation (and communication).

The figure on the right 14b, we look at how the efficiency changes for a constant problem size per core (N/c). We achieve the best efficiency when the number of cores is the lowest and it decreases as we increase the number of cores. The drop in efficiency is the largest when $N/c = 1000$, from around 100% to 70%. This drop decreases as we increase the N/c, for $N/c = 4000$ the efficiency drops from around 100% to 82.6%. This would also result in smaller increase in the relative time taken to compute the solution. Thus we can say that for larger problem size per cores we exhibit less deviation from ideal scaling.



(a) Strong Scaling



(b) Weak Scaling

Figure 14: Scalibility results for the Modified Thomas Algorithm (MPI-only implementation)

3.7 Limits of Strong Scaling

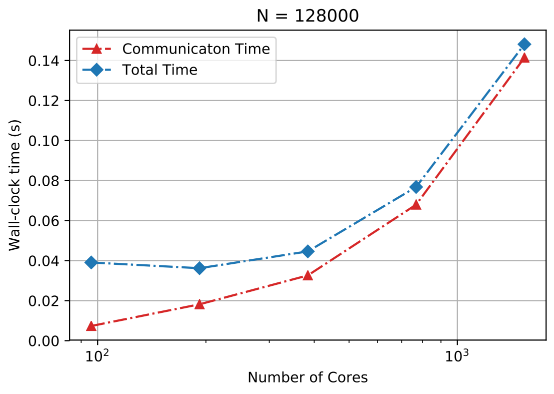
We now examine Strong scaling at a high core count i.e. we utilize multiple nodes to understand and identify the point where Strong scaling reaches a limit beyond which, increasing the number of cores is redundant. At such a point, the communication between cores and the fraction of the program which is serial becomes the dominating part. It can be observed from the graph in Figure 14a that as we increase the number of cores we reach the maximum limit of the Speedup value. The problem size is increased to up-to one million unknowns and the the system is solved on up-to 32 nodes, (each node has 48 cores). The details are presented in Table 3.

Problem Size	128000	256000	512000	1024000		
Nodes	2	4	6	8	16	32
Cores	96	192	288	384	768	1536

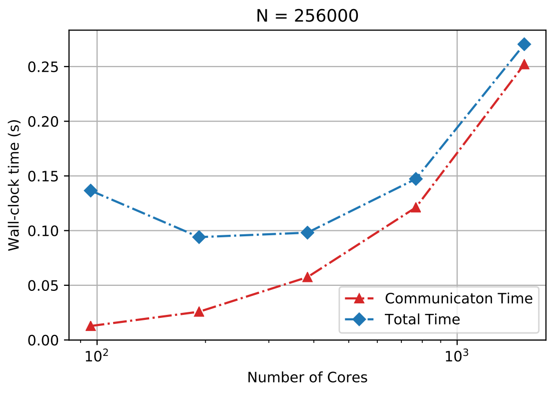
Table 3: Problem sizes for the Modified Thomas Algorithm and the number of nodes (and cores) at which it is evaluated.

In Figures 15a, 15b, 16a and 16b, we look at the variation of the total wall clock time and communication times for *large* problem sizes. We can see that in all the cases, the communication time increases as the number of cores are increased. This is expected as we employ more and more nodes to solve a problem of constant size.

For a problem of size $N = 128k$ (see Figure 15a), we can see that the total time is decreases steadily till we reach 192 cores and beyond that we see a rise in the total time taken to complete the simulation. For the problem of size $256k$ (see Figure 15b) the minimum for the total-time curve is at 384 cores, after which the required time increases. For $N = 512k$ unknowns (see Figure 16a), we see decreasing solution times till up-to 768 cores, after which it increases. Finally, for $N = 1024k$ unknowns, the total time is decreases till 768 cores, post which it increases. Thus, these graphs represent the Strong scaling limit for a given problem being solved on a specific architecture and communication network.

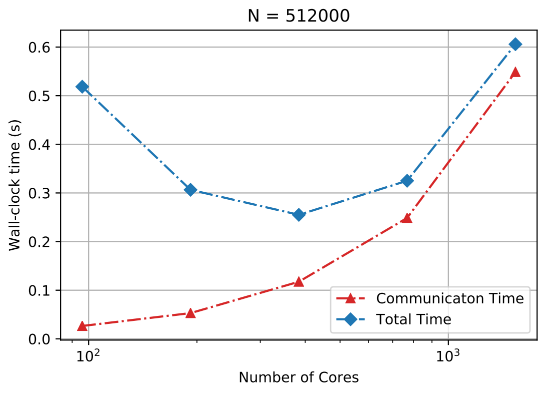


(a) Problem size = 128k

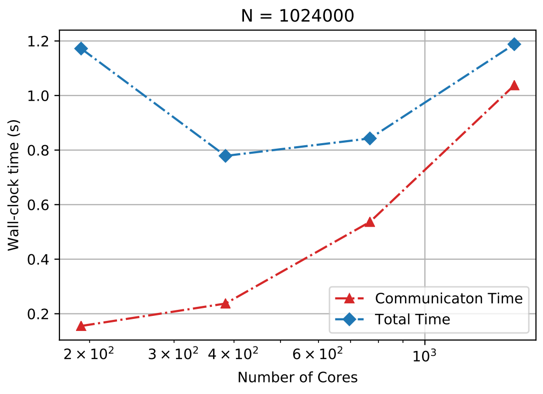


(b) Problem size = 256k

Figure 15



(a) Problem size = 512k



(b) Problem size = 1024k

Figure 16: Limits of scaling

4 Conclusion

This work aimed to find a suitable replacement for the serial Thomas Algorithm, which is used to solve the PDEs simulating the Reaction-Diffusion, uptake and secretion in the BioFVM solver used by the PhysiCell environment. For this problem, we first explored the Cyclic Reduction (CR) Algorithm. The serial version of this algorithm was parallelized using shared memory paradigm using the OpenMP API.

We first check that this solver gives favorable results for a standard Poisson problem and verify its correctness. Experiments were run for different problem sizes and a range of number of threads. We have shown that the CR algorithm with OpenMP parallelization exhibits favorable Speedup and thus, shows good scalability to be solved on a large cluster of computers for large problem sizes. The highest Speedup exhibited by our implementation is 33.60, which is achieved with 48 threads for the problem size of $N = 128000$. The variation in Efficiency, which quantifies the processor utilization, is also evaluated. We observe the highest efficiency for the least number of threads, and the efficiency drops for a larger number of threads. We also observe that for a greater number of threads, efficiency increases as we increase the problem size. Finally, we take a look at how this algorithm scales by keeping a constant problem size and looking at the Speedup for strong scaling and look at how much the Speedup deviates from the ideal. For weak scaling, we look at the efficiency obtained for a constant problem size per thread. We report that the efficiency drops down to 75% from the ideal as we increase the number of threads. We have also observed that when the number of threads is greater than 8, this algorithm performs better than the serial Thomas Algorithm. Thus, it can prove to be a suitable replacement for the existing solver in BioFVM.

Finally, we also look at a Modified version of the Thomas Algorithm described by Lazlo [16] to explore the distributed memory solution. Here we have used divided the matrix system into chunks and applied the Thomas Algorithm to a *Reduced* system of equations. We have used MPI communication to parallelize this algorithm. After running experiments for different ranges of problem sizes and on a range of a number of cores, we find that this method also shows good scalability. We achieve a Speedup of around 30.6 with 48 cores and for the problem size of $N = 128000$. Again, we look at variation in Efficiency, and we find results similar to those reported for the Cyclic Reduction method. For the fewest cores, we get the highest efficiency, and it drops as we increase the number of cores, similar to what was observed previously. We look at the strong and weak scaling for this algorithm, and observe that the deviation from the ideal is slower for a higher value of matrix size per cores. In the last study, we look at the strong scaling at a high core count. We show that increasing the number of cores beyond a particular number of cores is redundant and the execution time, in fact, increases as more time is spent for communication between cores.

We have investigated two algorithmic alternatives to the serial Thomas Algorithm used by Bio-FVM solver. One which implements shared memory paradigm and one with a distributed memory. Both of these methods have shown to achieve significant Speedup and show good scaling. We have thus presented with two suitable alternatives to improve the performance of the Bio-FVM solver.

5 Future Work

The work done so far can expanded further on the following fronts

- The MPI parallel version of the Thomas Algorithm can be further parallelized in a shared memory environment using OpenMP. This is expected to increase the Speedup obtained in this thesis even further. The PhysiCell environment has implemented OpenMP parallelization to it's other components so an OpenMP + MPI parallel hybrid version would smoothly replace the Thomas Solver and offer much better performance.
- The two algorithms mentioned in this work can be combined to form another MPI + OpenMP parallel hybrid of the Thomas Algorithm and Cyclic Reduction Algorithm. In Algorithm 3, we mention that in step three we solve the reduced system of equations using the standard Thomas Algorithm. This can instead be solved using the OpenMP parallel version of Cyclic Reduction. As mentioned in the results section, the parallel version of CR does perform better than the serial Thomas Algorithm when the number of threads are more than 8. Thus, when dealing with very large systems which will use multiple nodes and thousands of cores in parallel we can consider this option of hybrid parallelization.
- The numerical methods described in this work have shown good scalability and do achieve a better performance than the Thomas Algorithm. As a consequence, both algorithms are good candidates to be integrated with the PhysiCell environment, in the BioFVM solver. This integration would require some further work for smooth operation . Further, specific experiments must be conducted to test the performance of the integration with PhysiCell. This can be done on the standard examples provided by PhysiCell.
- Finally, it would also possible to explore iterative methods to solve Tri-diagonal systems of equations. For instance, Iterative Methods such as Krylov Methods for solving large linear systems can be used instead of the Direct methods which PhysiCell currently implements.

A Appendix

Parallel Cyclic Reduction run-time data

The average of time recorded for three runs for Cyclic Reduction algorithm solving an $Ax = b$ system

N	Serial	T = 2	T = 4	T = 8
500	0.0368184	0.0189883	0.0110119	0.00545635
1000	0.140423	0.0710694	0.0371366	0.0186122
2000	0.543681	0.27479	0.13899	0.0696449
4000	2.18072	1.13496	0.573753	0.294991
8000	9.00221	4.66075	2.40367	1.26133
16000	36.1252	18.4106	9.38645	4.91774
32000	147.075	74.0552	38.7015	20.1261
64000	592.319	299.438	153.914	81.9176
128000	2385.03	1216.81	622.113	333.122

N	T = 16	T = 24	T = 32	T=48	Thomas
500	0.00359276	0.00305026	0.00303234	0.0364326	0.005063
1000	0.00964518	0.00675952	0.00525653	0.0472844	0.022249
2000	0.0353916	0.0240437	0.0180714	0.0129162	0.114401
4000	0.15825	0.110263	0.0882286	0.0648032	0.400822
8000	0.720978	0.522945	0.418772	0.301663	1.48167
16000	2.72286	1.89531	1.49066	1.04604	5.373
32000	11.0137	7.70948	6.1782	4.32667	21.95
64000	45.6791	32.4678	25.585	19.3943	86.6017
128000	188.389	129.588	94.8107	70.9637	

Modified Thomas Algorithm run-time data

The average of time recorded for three runs for Modified Thomas Algorithm solving an $Ax = b$ system

N	Serial	Procs = 2	Procs = 4	Procs = 8
500	0.00242	0.000204	0.000129	0.000113
1000	0.001194	0.000602	0.00034	0.000208
2000	0.005477	0.002355	0.001223	0.000703
4000	0.0186	0.009305	0.004789	0.00255
8000	0.075866	0.037791	0.019273	0.010203
16000	0.303937	0.152449	0.077777	0.039812
32000	1.24737	0.608305	0.30757	0.156505
64000	5.00169	2.49914	1.2428	0.621809
128000	19.889	9.90813	5.00563	2.5091

N	Procs = 16	Procs = 24	Procs = 32	Procs = 48
500	0.000137	0.000169	0.000551	0.000315
1000	0.000191	0.000229	0.000304	0.000411
2000	0.000525	0.001181	0.000602	0.000838
4000	0.001605	0.001484	0.001471	0.001763
8000	0.006001	0.004951	0.004636	0.004818
16000	0.022207	0.017308	0.015984	0.015659
32000	0.08398	0.062848	0.054981	0.050052
64000	0.325638	0.235615	0.197465	0.173918
128000	1.30736	0.92513	0.752134	0.644724

References

[1] A. Ghaffarizadeh, R. Heiland, S.H. Friedman, S.M. Mumenthaler, and P. Macklin, *PhysiCell: an open source physics-based cell simulator for 3-D multicellular systems*, PLoS Comput. Biol. 14(2): e1005991, 2018 . DOI: 10.1371/journal.pcbi.1005991.

[2] John Metzcar, Yafei Wang, Randy Heiland, and Paul Macklin *A Review of Cell-Based Computational Modeling in Cancer Biology* JCO Clinical Cancer Informatics 2019 :3, 1-13

[3] Ghaffarizadeh, Ahmadreza, Samuel H. Friedman and P. Macklin. *BioFVM: an efficient, parallelized diffusive transport solver for 3-D biological simulations.* Bioinformatics 32 (2016): 1256 - 1258.

[4] Gene H. Golub, Charles Van Loan. *Matrix Computations* Third Edition 1996, John Hopkins Press

[5] Patrick Amestoy, Iain Duff, Jacko Koster, and Jean-Yves L'Excellent. *spacer MUMPS: A Multifrontal Massively Parallel Solver* 1998

[6] Volker Strassen (Aug 1969). *Gaussian elimination is not optimal.* Numerische Mathematik. 13 (4): 354–356. doi:10.1007/BF02165411. S2CID 121656251.

[7] Gander, W., Golub, G.H.: *Cyclic reduction—history and applications.* Scientific computing (Hong Kong, 1997), pp. 73–85. Springer, Singapore (1997)

[8] Trefethen, L. N., Bau, D. (1997). *Numerical Linear Algebra.* SIAM. ISBN: 0898713617

[9] J. Scott, *Sparse Direct Solvers 1: The Challenge. 43th Woudschoten Conference, 10 2018.* STFC Rutherford Appleton Laboratory and the University of Reading.

[10] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 2001.

[11] Wes Kendall, *MPI Tutorial*, https://mpitutorial.com/ Accessed : 2021-05-30

[12] G.I. Marchuk. *Splitting and alternating direction methods* volume 1 of Handbook of Numerical Analysis, pages 197–462. Elsevier, 1990.URL http://www. sciencedirect.com/science/article/pii/S1570865905800353

[13] B.L. Buzbee, G.H. Golub and C.W. Nielson, *On direct methods for solving Poisson's equations*, SIAM J. Numer. Anal. 7, 627-656 (1970).

[14] Stefan Bondeli, *Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations*, Parallel Computing, Volume 17, Issues 4–5, 1991,

[15] L. Brugnano, *A parallel solver for tridiagonal linear systems for distributed memory parallel computers*, Parallel Computing, Volume 17, Issue 9, 1991,

[16]] E. László, M. Giles, J. Appleyard, *Manycore algorithms for batch scalar and block tridiagonal solvers* ACM Trans. Math. Softw. 42 (4) (2016) 31:1–31:36.

[17] Ki-Ha Kim, Ji-Hoon Kang, Xiaomin Pan, Jung-Il Choi, *PaScaL TDMA: A library of parallel and scalable solvers for massive tridiagonal systems*, Computer Physics Communications, Volume 260, 2021,

[18] Amdahl, Gene M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS Conference Proceedings. (30): 483–485. doi: 10.1145/1465482.1465560

[19] Gustafson, John L. (1988). *Reevaluating Amdahl's law*, Communications of the ACM. 31 (5): 532–533. doi: 10.1145/42411.42415

[20] George Em Karniadakis, Robert M. Kirby *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*, Cambridge University Press. Oct, 2013. DOI:https://doi.org/10.1017/CBO9780511812583

[21] Joe Pitt-Francis, Jonathan Whiteley *Guide to Scientific Computing in C++*, Springer Publications, 2012. DOI 10.1007 978-1-4471-2736-9

[22] J. D. Hunter, *Matplotlib: A 2D Graphics Environment*, Computing in Science Engineering, vol. 9, no. 3, pp. 90-95, 2007.