ORNL/LTR-2020/40

Optimizing the Accelerated Recursive Doubling Algorithm for Block Tridiagonal Systems of Equations

Muktaka Joshipura
Sudip K Seal

August 7, 2020

ORNL/LTR-2020/40

Computer Science and Mathematics Division

Optimizing the Accelerated Recursive Doubling Algorithm for Block Tridiagonal Systems of Equations

Muktaka Joshipura
Sudip K. Seal

Date Published: August, 2020

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

ABSTRACT

The need to solve block tridiagonal systems with hundreds or thousands of right-hand sides for the same block tridiagonal matrix is common in a variety of disciplines. To meet this need, the Accelerated Recursive Doubling Algorithm was developed. After a right-hand side independent phase, the algorithm allows for the quick, online calculation of solutions for different right-hand sides. In this work, we present methods to optimize the Accelerated Recursive Doubling Algorithm in memory usage and computation time in a hybrid parallelization model. The right-hand side independent phase of the naïve implementation takes $\geq \frac{11}{3}$ the amount of memory required to store the tridiagonal matrix, while our implementation reduces the fraction to $\approx \frac{5}{3}$. The right-hand side dependent phase of the naïve implementation takes ≥ 6 times the amount of memory required to store the right-hand side, while our implementation reduces the fraction to ≈ 3. The computation time for the independent phase is reduced to $\approx \frac{2}{3}$ times that of the naïve implementation, while the computation time for the dependent phase is reduced to $\approx \frac{5}{9}$. With increasing numbers of shared-memory threads q on every distributed processing element, we have $O(q)$ theoretical speedup.

1. Introduction

A block tridiagonal system is a linear system $Ax = b$ such that A is a block tridiagonal matrix. A block tridiagonal matrix is a block matrix such that the only non-zero blocks are on the block diagonal or adjacent to the main block diagonal.

To describe the dimensions of the matrix A, we define two parameters M and N. M refers to the width of each square block. N refers to the number of block rows.

1.1 L_i, D_i, and U_i matrices

In order to reference specific blocks, we label each block in the block-tridiagonal matrix as either being 'lower' (L), 'diagonal' (D), or 'upper' (U). We distinguish the blocks on each block row by a block-row index. Therefore, the layout of A is as follows:

$$A = \begin{bmatrix} D_1 & U_1 & 0 & \cdots & \cdots & \cdots & 0 \\ L_2 & D_2 & U_2 & 0 & \cdots & \cdots & 0 \\ 0 & L_3 & D_3 & U_3 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & L_N & D_N \end{bmatrix}. \tag{1}$$

For convenience in using block-row indices, we define $L_1 = U_N = I$.

1.2 x_i and b_i vectors

We also section the solution vector x and the right-hand side vector b into smaller $M \times 1$ vectors, one for each of N block rows, laid out as follows:

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}. \tag{2}$$

Again, for convenience in using block-row indices, we define $x_0 = x_{N+1} = 0$.

1.3 X_i vectors and B_i matrices

Both the Accelerated and the non-accelerated Recursive Doubling Algorithms rely on a method of solving for x_{i+1} once we have obtained a solution for x_i and x_{i-1}, presented in [1]. This is done by multiplying a matrix B_i—determined from L_i, D_i and U_i—with a vector X_i—determined from x_i and x_{i-1}.

X_i and B_i were defined for $1 \leq i \leq N$ such that

$$X_i = \begin{bmatrix} x_i \\ x_{i-1} \\ 1 \end{bmatrix} \qquad B_i = \begin{bmatrix} -U_i^{-1}D_i & -U_i^{-1}L_i & U_i^{-1}b_i \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3}$$

We notice that the construction of B_i requires the following assumption to be made:
Assumption 1. For all $1 \leq i \leq N$, the matrix U_i is invertible.

The property that allows us to determine x_{i+1} from x_i and x_{i-1} is stated as a lemma as
Lemma 1. *For all* $1 \leq i \leq N$, *we have* $X_{i+1} = B_i X_i$.

The proof of this lemma is found in [1].

This lemma tells us that once x_1 is known, we can pack x_1 and $x_{1-1} = x_0 = 0$ into X_1, calculate $X_2 = B_1 X_1$, and extract x_2 from the result. We can then calculate every subsequent x_i in serial, thus obtaining the entirety of the solution vector x.

1.4 S_i matrices

The property described in Lemma 1 yields the possibility of determining the value of all x_i in parallel once x_1 is determined. This is used in both versions of the Recursive Doubling Algorithms. In order to do this, a prefix product of B_i, labeled S_i, is calculated, which is then multiplied with X_1 to obtain X_i. This property is presented in [1].

The prefix product $S_i = B_i B_{i-1} \ldots B_1$ is defined recursively for $1 \leq i \leq N$ as

$$S_i = \begin{cases} B_1 & i = 1 \\ B_i S_{i-1} & i > 1 \end{cases}. \tag{4}$$

The property that allows X_i to be calculated from X_1 using S_i is stated as a theorem as

Theorem 1. *For all* $1 \leq i \leq N$*, we have* $X_{i+1} = S_i X_1$.

This property follows trivially from the definition of x_i in Equation 4 and the property of B_i in Lemma 1.

The method for determining x_1, and by extension X_1 as used in both versions of the Recursive Doubling Algorithm is

Theorem 2. $x_1 = -[S_N^{11}]^{-1} S_N^{13}$ *and so* $X_1 = \begin{bmatrix} -[S_N^{11}]^{-1} S_N^{13} \\ 0 \\ 1 \end{bmatrix}$.

A proof for this theorem can be found in [2].

However, the use of this property relies on the assumption that
Assumption 2. S_N^{11} is invertible

1.5 C_i and F_i matrices

The Accelerated Recursive Doubling Algorithm presented in [2] accelerates the computation of solving for x for multiple right-hand sides b by separating the computation dependent on b (right-hand side dependent) and the computation independent of the right-hand side. In order to do this, B_i is separated into two matrices, one right-hand side independent (C_i), and one right-hand side dependent (F_i).

To be precise, for all $1 \leq i \leq N$, we decompose B_i into C_i and F_i as

$$C_i = \begin{bmatrix} -U_i^{-1} D_i & -U_i^{-1} L_i & 0 \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad F_i = \begin{bmatrix} 0 & 0 & U_i^{-1} b_i \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5}$$

C_i and F_i matrices have the following properties that help simplify expressions involving their products, as presented in [2].
Lemma 2. *For* $1 \leq i \leq N$ *and* $1 \leq j \leq N$*, we have* $F_i F_j = 0$.
Lemma 3. *For* $1 \leq i \leq N$ *and* $1 \leq j \leq N$*, we have* $F_i C_j = F_i$.

These lemmas are presented without proof because they can be verified trivially.

1.6 Q_i matrices

In the Accelerated Recursive Doubling algorithm, S_i is constructed by combining a result that only depends on A, and a result that only depends on the right-hand side b. The result that only depends on A is the Q_i matrix, which is the prefix product of C_i matrices.

The prefix product $Q_i = C_i C_{i-1} \ldots C_1$ is defined recursively for $1 \leq i \leq N$ as

$$Q_i = \begin{cases} C_1 & i = 1 \\ C_i Q_{i-1} & i > 1 \end{cases} \tag{6}$$

Q_i matrices have the following property that helps simplify expressions, stated as

Lemma 4. *For* $1 \leq i \leq N$ *and* $1 \leq j \leq N$*, we have* $F_i Q_j = F_i$*.*

This lemma follows trivially from the definition of Q_i and the property in Lemma 3.

1.7 E_i matrices

The right-hand side dependent result that is used to construct S_i in the Accelerated Recursive Doubling Algorithm is the E_i matrix, defined recursively for $1 \leq i \leq N$ as:

$$E_i = \begin{cases} F_1 & i = 1 \\ F_i + C_i E_{i-1} & i > 1 \end{cases} \tag{7}$$

Using this formulation, we can start from E_1 and determine all subsequent E_i using the recursive formulation. This formulation of E_i is simpler but equivalent to the original formulation in [2].

A property that helps simplify expressions is
Lemma 5. *For* $1 \leq i \leq N$ *and* $1 \leq j \leq N$*, we have* $F_i E_j = 0$*.*

Proof. We can prove this property by induction.

Let $1 \leq i \leq N$ be a valid index.

Base case: $j = 1$

By Lemma 2, we have

$$F_i F_1 = 0$$

Inductive step

Let the lemma be true for values of j up to and including k. This means that $F_i E_k = 0$. We have to prove that $F_i E_{k+1} = 0$.

Since $k \geq 1$, we have $k + 1 \geq 2$. So, by the definition of E_i, we have $E_{k+1} = F_{k+1} + C_{k+1} E_k$.

$$\begin{aligned} F_i E_{k+1} &= F_i(F_{k+1} + C_{k+1} E_k) = (F_i F_{k+1}) + (F_i C_{k+1}) E_k \\ &= 0 + F_i E_k \text{ (By Lemma 2 and 3)} \\ &= 0 \text{ (By the inductive hypothesis).} \end{aligned}$$

□

Using the value of Q_i, determined during the right-hand side independent computation, and the value of E_i, determined during the right-hand side dependent computation, we can construct the value of S_i using this property.
Theorem 3. *For all* $1 \leq i \leq N$*, we have* $Q_i + E_i = S_i$*.*

Proof. We can prove this property using induction.

Base case: $i = 1$

By definition of S_i, Q_i and E_i, we have

$$S_1 = B_1 \qquad Q_1 = C_1 \qquad E_1 = F_1$$

Therefore,

$$Q_1 + E_1 = C_1 + F_1 = B_1 = S_1$$

Inductive step

Let the theorem be true for all i up to and including k. Therefore, $S_k = Q_k + E_k$. We have to prove that $S_{k+1} = Q_{k+1} + E_{k+1}$.

By definition of C_i and F_i, we have $Q_{k+1} = C_{k+1} + F_{k+1}$.

By definition of S_i and the inductive hypothesis, we have

$$S_{k+1} = B_{k+1}S_k = (C_{k+1} + F_{k+1})S_k = (C_{k+1} + F_{k+1})(Q_k + E_k).$$

Expanding, we get

$$\begin{aligned}(C_{k+1} + F_{k+1})(Q_k + E_k) &= C_{k+1}Q_k + C_{k+1}E_k + F_{k+1}Q_k + F_{k+1}E_k \\ &= Q_{k+1} + C_{k+1}E_k + F_{k+1} + 0 \\ &\text{(Using Lemma 4 and Lemma 5)}\end{aligned}$$

Since $k \geq 1$, we know that $k + 1 > 1$. So, we have $E_{k+1} = C_{k+1}E_k + F_{k+1}$. Plugging in the value of E_{k+1}, we get:

$$S_{k+1} = Q_{k+1} + E_{k+1}.$$

We have thus shown the inductive step to be true, which proves the theorem. □

1.8 Z_i matrices

The original formulation of E_i allows for the calculation of all E_i in parallel. For this formulation, we must define another kind of matrix, Z_i, defined for $1 \leq i \leq N$ as:

$$Z_i = \sum_{k=1}^{i} Q_i^{-1} F_i. \tag{8}$$

Further, we define $Z_0 = 0$.

The existence of Z_i therefore depends on the invertibility of Q_k for all $1 \leq k \leq i$. In order to get the most minimal set of assumptions required to ensure this property, we investigated the conditions in which Q_i are invertible. We discovered the following property.

Lemma 6. *For $1 \leq i \leq N$, the matrix Q_i is invertible if and only if $C_1, \ldots, C_i$ are invertible.*

Proof. By the properties of the determinant and the definition of Q_i given in Equation 6, we have

$$\det Q_i = \prod_{k=1}^{i} \det C_i \tag{9}$$

Part 1: $C_1, \ldots, C_i$ are invertible $\implies$ Q_i is invertible

If $C_1, \ldots, C_i$ are invertible, then $\det C_1, \ldots, \det C_i$ are non-zero. This means that the product of the determinants is also nonzero. Therefore, by Equation (9), we know that $\det Q_i \neq 0$, and so Q_i is invertible.

Part 2: Not all of $C_1, \ldots, C_i$ are invertible $\implies$ Q_i is not invertible

Let $C_j \in \{C_1, \ldots, C_i\}$ such that it is a non-invertible matrix. Then, $\det C_j = 0$. By Equation (9), we know that $\det Q_i = 0$, and so Q_i is not invertible. □

Since the invertibility of Q_i matrices is conditional on the invertibility of C_i matrices, we investigated the invertibility of C_i matrices. We discovered the following property.

Lemma 7. *Given that U_i is invertible, for $1 \leq i \leq N$, the matrix C_i is invertible if and only if L_i is invertible.*

Proof.

Part 1: L_i is invertible $\implies$ C_i is invertible

A matrix is invertible if and only its inverse exists. Let L_i be invertible. Therefore L^{-1} exists and so does the matrix

$$D = \begin{bmatrix} 0 & I & 0 \\ -L_i^{-1}U_i & -L^{-1}D_i & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Now, we have

$$DC_i = \begin{bmatrix} 0 & I & 0 \\ -L_i^{-1}U_i & -L^{-1}D_i & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -U_i^{-1}D_i & -U_i^{-1}L_i & 0 \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = I$$

Therefore D is the inverse of C_i and C_i is invertible.

Part 2: L_i is not invertible $\implies$ C_i is not invertible

A matrix K is invertible if and only if there are no non-zero vectors y such that $Ky = 0$. Since L_i is not invertible, there is at least one non-zero vector y such that $L_i y = 0$.

Let $Y = \begin{bmatrix} 0 \\ y \\ 0 \end{bmatrix}$.

Since $y \neq 0$, we know that $Y \neq 0$.

Then,

$$C_iY = \begin{bmatrix} -U_i^{-1}D_i & -U_i^{-1}L_i & 0 \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} -U_i^{-1}L_iy \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Since there exists a non-zero vector Y such that $C_iY = 0$, we know that C_i is not invertible. □

Therefore, we were able to determine that the following condition is both necessary and sufficient for the invertibility of Q_i and therefore the existence of Z_i for all $1 \leq i \leq N$:
Assumption 3. For $1 \leq i \leq N$, the matrix L_i is invertible.

Note: A method suggested in [2] to make ARDA numerically stable for the same classes of matrices for which Cyclic Reduction is stable is to take the LU-decomposition of A, and then solve $Ax = LUx = b$ in two steps: First solving $Ly = b$, and then solve $Ux = y$.

This is impossible to do by the ARDA algorithm since L is lower-triangular, violating Assumption 1 U is upper-triangular, violating Assumption 3.

The original formulation of E_i as described in [2] given in the theorem below, which also states the equivalence of the formulations described in this work and [2].

Theorem 4. *For $1 \leq i \leq N$, the matrix $E_i = Q_iZ_{i-1} + F_i$.*
Proof.

Base case: $i = 1$

From Equation (7), we have

$$E_i = E_1 = F_1.$$

The right hand side evaluates to

$$Q_iZ_{i-1} + F_i = Q_1Z_0 + F_1 = 0 + F_1 = F_1.$$

Therefore, the proposition is true for $i = 1$.

Inductive step

Assume that $E_i = Q_iZ_{i-1} + F_i$ for $i = k$. We have to prove that $E_{k+1} = Q_{k+1}Z_k + F_{k+1}$. From Equation (7), we have

$$\begin{aligned} E_{i+1} &= F_{i+1} + C_{i+1}E_i = F_{i+1} + C_{i+1}(Q_iZ_{i-1} + F_i) \\ &= F_{i+1} + C_{i+1}(Q_iZ_{i-1} + Q_iQ_i^{-1}F_i) = F_{i+1} + C_{i+1}Q_i(Z_{i-1} + Q_i^{-1}F_i). \end{aligned}$$

By Equation (6), we have $C_{i+1}Q_i = Q_{i+1}$, and by Equation (8) we have $Z_{i-1} + Q_i^{-1}F_i = Z_i$. Therefore, we get

$$E_{i+1} = Q_{i+1}Z_i + F_i.$$

□

2. Original Algorithm

The Accelerated Recursive Doubling Algorithm as described in [2] is described below for reference.

2.1 Independent phase

Algorithm 1: Independent Phase of the Non-optimized Accelerated Recursive Doubling Algorithm

```
1. Assign $K_r = \frac{N}{P}$ block rows of $A$ to each processor.
2. For each local block row $k$, compute $U_k^{-1}$, and use it to calculate $C_k$.
3. Calculate the local prefix product of the $C_k$ products to obtain local $Q_k$'s.
4. Perform a prefix product of the total products across all processors.
5. Use the received prefix product to update the local prefixes to obtain global $Q_k$'s.
6. Invert $Q_k$ to determine $Q_k^{-1}$.
```

2.2 Dependent phase

Algorithm 2: Dependent Phase of the Non-optimized Accelerated Recursive Doubling Algorithm

```
1. Assign $K_r = \frac{N}{P}$ block rows of $b$ to each processor.
2. For each local block row $k$, use $U_k^{-1}$ to compute $F_k$.
3. For each local block row $k$, compute $V_k = Q_k^{-1} F_k$.
4. Calculate the local prefix sum of the $V_k$'s to obtain local $Z_k$'s.
5. Perform a prefix sum of the local totals across all processors.
6. Use the received prefix sum to update the local prefix sums to obtain global $Z_k$'s.
7. Use $Q_k$, $Z_k$ and $F_k$ to calculate $E_k$.
8. Use $Q_k$ and $E_k$ to calculate $S_k$.
9. On the processor that is assigned the final block row $N$, use $S_N$ to calculate $x_1$, and broadcast $x_1$ to
   all processors.
10. Use $x_1$ to construct $X_1$.
11. Use $S_k$ and $X_1$ to determine $X_k$, and thus determine $x_k$.
```

3. Algorithm Setup

3.1 Block row assignment and indexing

The N block rows are distributed between the P distributed processing elements evenly. The number of block rows assigned to a given processing element r is $K_r \approx \frac{N}{P}$. Block rows can be indexed in two ways:

- Global index: Represented by i, this is the row index of the block row in A. The global index follows 1-based indexing.
- Local index: Represented by k, this is the index of the block row among the block rows assigned to the distributed processing element. The local index follows 0-based indexing.

3.2 Shared-memory threading

The highly parallelizable nature of the Accelerated Recursive Doubling algorithm lends to the possibility of using shared-memory threads in conjunction with the distributed-memory algorithm described in [2].

We consider q shared-memory threads running on every distributed processing element. The K_r block rows assigned to the distributed processing element are sectioned evenly between the threads. The thread t is responsible for block rows with local indexes $\mathtt{ts}_t \leq k < \mathtt{te}_t$.

3.3 Registers and Instructions

To increase the efficiency of the algorithm, the strategy chosen was to represent the algorithm in terms of atomic operations similar to those found in an assembly language. Operations like inverting, multiplying, and adding matrices become the 'instructions', while memory blocks of various sizes are considered to be 'registers'. Memory is optimized by reducing the number of 'registers' required, while computation is optimized by reducing the number or cost of the 'instructions' in the program.

In this implementation, memory is assigned once when the matrix A is input, and freed once the solution to the right-hand side is output. All memory is contiguous, where consecutive registers have consecutive addresses in memory.

The memory is laid out with the following registers in order

1. **Full block registers (size M^2K_r each):** R^0, R^1, R^2, R^3, R^4
   - R^0_k refers to the k-th block of size M^2 in R^0.
   - R^{01} refers to the register formed by combining registers R^0 and R^1 end-to-end.
   - R^{01}_k refers to the k-th block of size $2M^2$ in R^{01}, interpreted as a matrix of dimensions $M \times 2M$.
   - $R^{01}_k[0]$ refers to the 0-th block of size $M \times M$ in R^{01}_k.
   - R^0_k, R^1_k, R^2_k are assumed to be initialized with the appropriate L_i, D_i, and U_i respectively, at the beginning of the independent phase execution.
2. **Receive register (size $4M^2$):** R^r.
3. **Thread registers (size $4M^2q$ each):** R^{t0}, R^{t1}
   - R^{t0}_t refers to the t-th block of size $4M^2$ in R^{t0}.
   - R^{t0}_{v0} refers to the first block of size $2M$ in R^{t0}.
   - R^{t0}_{v1} refers to the block of size M immediately after R^{t0}_{v0}.
4. **Vector register (size MK_r each):** R^{v0}, R^{v1}, R^{v2}.
   - R^Z refers to the register formed by combining registers R^{v0} and R^{v1} end-to-end.
   - R^{v0} is assumed to be initialized with the appropriate b_i at the beginning of every dependent phase execution.

4. Optimizations

In this work, we present a large improvement in time and memory complexity of the algorithm. In order to do this, we exploit the structure and properties of the intermediate matrices and vectors calculated. We also present a method to efficiently perform parallel prefixes for the requirements of this algorithm in a shared-memory paradigm.

4.1 Optimizations in the Independent Phase

4.1.1 C_i matrix compression

A C_i matrix is determined from L_i, D_i, and U_i matrices and is used as an input to a parallel prefix product algorithm in order to determine Q_i. C_i matrices are of size $(2M + 1) \times (2M + 1)$, and are of the form

$$C_i = \begin{bmatrix} C_i^{11} & C_i^{12} & 0 \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{10}$$

Therefore, C_i can be compressed into a smaller matrix, given by

$$C_i' = \begin{bmatrix} C_i^{11} & C_i^{12} \end{bmatrix}. \tag{11}$$

A C_i' matrix can now be stored in a single cell of a concatenated register like R^{01}.

4.1.2 Q_i matrix compression

A Q-type matrix is a matrix produced by the product of several C_i matrices. They are of of size $(2M + 1) \times (2M + 1)$, and are of the form

$$Q = \begin{bmatrix} Q^{11} & Q^{12} & 0 \\ Q^{21} & Q^{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{12}$$

Therefore, a Q_i matrix is a Q-type matrix.

These matrices are isomorphic over matrix multiplication to smaller $2M \times 2M$ matrices, which are constructed as

$$Q' = \begin{bmatrix} Q^{11} & Q^{12} \\ Q^{21} & Q^{22} \end{bmatrix}. \tag{13}$$

Theorem 5. *If Q_A is a Q-type matrix, and C is a C_i matrix, then $Q_B = CQ_A$ is such that the second block row of the Q' representation of Q_B is equal to the first block row of the Q' representation of Q_A.*

Proof.

$$Q_B = CQ_A.$$

Multiplying on the left on both sides of the above equation with $\begin{bmatrix} 0 & I & 0 \end{bmatrix}$, we get

$$\begin{aligned}
\begin{bmatrix} 0 & I & 0 \end{bmatrix}\begin{bmatrix} Q_B^{11} & Q_B^{12} & 0 \\ Q_B^{21} & Q_B^{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} &= \begin{bmatrix} 0 & I & 0 \end{bmatrix}\begin{bmatrix} C^{11} & C^{12} & 0 \\ I & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} Q_A^{11} & Q_A^{12} & 0 \\ Q_A^{21} & Q_A^{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} Q_B^{21} & Q_B^{22} & 0 \end{bmatrix} &= \begin{bmatrix} I & 0 & 0 \end{bmatrix}\begin{bmatrix} Q_A^{11} & Q_A^{12} & 0 \\ Q_A^{21} & Q_A^{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
\begin{bmatrix} Q_B^{21} & Q_B^{22} & 0 \end{bmatrix} &= \begin{bmatrix} Q_A^{11} & Q_A^{12} & 0 \end{bmatrix}.
\end{aligned}$$

From the definition of Q_i' in Equation (13) and the above result, we get

$$\begin{aligned}
Q_B' = \begin{bmatrix} Q_B^{11} & Q_B^{12} \\ Q_B^{21} & Q_B^{22} \end{bmatrix} &= \begin{bmatrix} Q_B^{11} & Q_B^{12} \\ Q_A^{11} & Q_A^{12} \end{bmatrix} \\
Q_i' &= \begin{bmatrix} Q_A^{11} & Q_A^{12} \\ Q_A^{21} & Q_A^{22} \end{bmatrix}.
\end{aligned}$$

We observe that the second block row of Q_B' is the first block row of Q_A'. □

This means that if $Q_1', Q_2', \ldots, Q_k'$ were the Q' representations of prefix product of some ordered collection of C_i matrices, the second block row of Q_m' would be equal to the first block row of Q_{m-1}' if $i > 1$, and would be equal to the first block row of the identity matrix $\begin{bmatrix} I & 0 \end{bmatrix}$ if $i = 0$.

This further means that when a Q' is calculated as a result of a prefix operation, and stored among adjacent prefixes, we can further compress the Q' matrix into a Q'' matrix given by

$$Q'' = \begin{bmatrix} Q^{11} & Q^{12} \end{bmatrix}.$$

In this case, since we only have to calculate the first block row, the time complexity of this multiplication is only $T_{C' \times Q'} = T_{Q'' \times Q'} = 4C_{\text{mul}}M^3$ instead of $8C_{\text{mul}}M^3$.

However, when a Q' is calculated by multiplying two other Q''s and not stored among adjacent prefixes, we need to calculate and store the full Q' result. The time complexity of this operation is therefore $T_{Q' \times Q'} = 8C_{\text{mul}}M^3$.

For every block row $1 \leq i \leq N$, we define the Q'' representation of Q_i as Q_i''.

4.1.3 Q^{-1} compression

Because of the isomorphism, we know that even Q_i^{-1} matrices are representable as Q' matrices, which are two block-rows wide.

As described in [2], only the first block column of the inverse of a Q_i^{-1} is used. Therefore, we can omit storing and calculating other block columns of the matrix.

Since the Q' representation of Q_i (represented by Q_i') are block matrices, we can use the following property of block matrices

$$Q_i' = \begin{bmatrix} Q^{11} & Q^{12} \\ Q^{21} & Q^{22} \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ then } Q^{-1} = \begin{bmatrix} A^{-1}(I + B(D - CA^{-1}B)^{-1}CA^{-1}) & -A^{-1}B(D - CA^{-1}B) \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}. \quad (14)$$

Assuming that A and $D - CA^{-1}B$ are invertible. This occurs when Q_i^{11} and $Q_i^{22} - Q_i^{21}(Q_i^{11})^{-1}Q_i^{12}$ are invertible.

We define $V = D - CA^{-1}B$ and $W = BV^{-1}CA^{-1}$ for brevity.

We therefore can calculate only the first block column of Q_i^{-1} like so:

$$Q_i^{-1} = \begin{bmatrix} A^{-1}(I + W) \\ -V^{-1}CA^{-1} \end{bmatrix}. \quad (15)$$

However, this optimization requires the following assumption to be made:
Assumption 4. For $1 \le i \le N$,

4.2 Optimization in the Dependent Phase

4.2.1 Evaluating x_i

Theorem 6. *For* $1 \le i \le N$*,*

$$x_i = Q_{i-1}'' \begin{bmatrix} x_0 + Z_i^{13} \\ Z_i^{23} \end{bmatrix}$$

Proof. From Theorem 1, we have

$$\begin{aligned} X_{i+1} &= S_i X_i \\ \begin{bmatrix} x_{i+1} \\ x_i \\ 1 \end{bmatrix} &= (Q_i + E_i)X_1 = (Q_i + Q_i Z_{i-1} + F_i)X_1 = (Q_i(I + Z_{i-1}) + F_i)X_1 \\ &= Q_i(I + Z_{i-1})X_1 + F_i X_1 \\ &= \begin{bmatrix} Q_i^{11} & Q_i^{12} & 0 \\ Q_i^{21} & Q_i^{22} & 0 \\ 0 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} x_0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} Z_i^{13} \\ Z_i^{23} \\ 0 \end{bmatrix} \right) + \begin{bmatrix} F_i^{13} \\ 0 \\ 0 \end{bmatrix}. \end{aligned}$$

Multiplying both sides of the above equation with $\begin{bmatrix}0 & 1 & 0\end{bmatrix}$ on the left, we get

$$\begin{aligned}
\begin{bmatrix}0 & 1 & 0\end{bmatrix}\begin{bmatrix}x_{i+1}\\ x_i\\ 1\end{bmatrix} &= \begin{bmatrix}0 & 1 & 0\end{bmatrix}\left(\begin{bmatrix}Q_i^{11} & Q_i^{12} & 0\\ Q_i^{21} & Q_i^{22} & 0\\ 0 & 0 & 1\end{bmatrix}\left(\begin{bmatrix}x_0\\ 0\\ 1\end{bmatrix} + \begin{bmatrix}Z_i^{13}\\ Z_i^{23}\\ 0\end{bmatrix}\right) + \begin{bmatrix}F_i^{13}\\ 0\\ 0\end{bmatrix}\right)\\
x_i &= \begin{bmatrix}Q_i^{21} & Q_i^{22} & 0\end{bmatrix}\left(\begin{bmatrix}x_0\\ 0\\ 1\end{bmatrix} + \begin{bmatrix}Z_i^{13}\\ Z_i^{23}\\ 0\end{bmatrix}\right) + 0\\
&= \begin{bmatrix}Q_i^{21} & Q_i^{22}\end{bmatrix}\left(\begin{bmatrix}x_0\\ 0\end{bmatrix} + \begin{bmatrix}Z_i^{13}\\ Z_i^{23}\end{bmatrix}\right) = Q''_{i-1}\begin{bmatrix}x_0 + Z_i^{13}\\ Z_i^{23}\end{bmatrix}.
\end{aligned}$$

□

4.3 Shared-memory parallel prefix

A parallel prefix problem on an array X of length n over a binary, associative operator $\odot$ is to determine an array P of length n such that $P[k] = X[0] \odot X[1] \odot X[2] \odot \cdots \odot X[k]$.

This can be done in shared memory for q threads in the following method, known as the 'hypercube' method:

Algorithm 3: Naïve Shared-Memory Parallel Prefix

```
$TT$ is an array of length $q$, the number of threads;
$TP$ is an array of length $q$;
$b \leftarrow 1$ is an integer;
/* Thread-local Prefix Phase */
for $t \leftarrow 0$ to $q - 1$ in parallel do
    $P[ts_t] \leftarrow P[ts_t]$;
    for $i \leftarrow ts_t + 1$ to $te_t - 1$ do
        $P[i] \leftarrow P[i - 1] \odot X[i]$;
    end
    $TT[t] \leftarrow P[te_t - 1]$;
    $TP[t] \leftarrow P[te_t - 1]$;
end
if $q > 1$ then
    /* Cross-thread Prefix Phase */
    while $(1 \ll b) < q$ do
        for $t \leftarrow 0$ to $q - 1$ in parallel do
            $i \leftarrow t \wedge (1 \ll b)$;
            if $i < q$ then
                if $i < t$ then
                    $TT[t] \leftarrow TT[i] \odot TT[t]$;
                    $TP[t] \leftarrow TP[i] \odot TT[t]$;
                else
                    $TT[t] \leftarrow TT[t] \odot TT[i]$;
                end
            end
        end
        $b \leftarrow b + 1$;
    end
    /* Thread-local Update Phase */
    for $t \leftarrow 1$ to $q - 1$ in parallel do
        for $i \leftarrow ts_t$ to $te_t - 1$ do
            $P[i] \leftarrow TP[t - 1] \odot P[i]$
        end
    end
end
```

Therefore, where $T_\odot$ is the time complexity of the implementation of the $\odot$ operator, and M_X is the number of block cells in each element of X, the time complexity of Algorithm 3 is

$$T_{npp}(n, q) = \begin{cases} 2\left(\frac{n}{q} + \log_2(q)\right) T_\odot + 2C_{\text{copy}}(qM_X) & q > 1 \\ nT_\odot + 2C_{\text{copy}}(M_X) & q = 1 \end{cases}. \tag{16}$$

The memory complexity of Algorithm 3 is

$$M_{npp}(n, q) = 2C_{\text{cell}} M_X (n + q). \tag{17}$$

At every stage, the values in TT are available somewhere in TP, so we can preclude TT entirely by modifying the Cross-thread Prefix Phase.

Algorithm 4: Better Shared-memory Parallel Prefix

```
$TP$ is an array of length $q$;
/* Thread-local Prefix Phase as in Algorithm 3 without copying to $TT$ */
$\ldots$;
/* Better Cross-thread Prefix Phase */
for $t \leftarrow 0$ to $\lfloor \frac{q}{2} \rfloor$ in parallel do
    $n \leftarrow 1 \ll 0$ is a two's complement signed integer;
    while $n < q$ do
        /* ~ is bitwise complement and & is bitwise AND */
        /* − is two's complement negation */
        $target \leftarrow ((t \,\&\, (-n)) \ll 1) + n + (t \,\&\, \sim (-n))$;
        if $target < q$ then
            $source \leftarrow (t \,\&\, (-n)) - 1$;
            /* For an illustration on the source-target pairings, see Figures 1
               and 2 */
            $TP[target] \leftarrow TP[source] \odot TP[target]$;
        end
        $n \leftarrow n \ll 1$;
        barrier;
    end
end
/* Thread-local Update Phase as in Algorithm 3 */
```

Since this algorithm eliminates the need to copy values to TT, this reduces the time complexity to

$$T_{bpp}(n, q) = \begin{cases} \left(2\frac{n}{q} + \log_2(q)\right) T_{\odot} + C_{\text{copy}}(q M_X) & q > 1 \\ n T_{\odot} + C_{\text{copy}} M_X & q = 1 \end{cases}. \tag{18}$$

Since the array TT is eliminated, the memory complexity is reduced to

$$M_{bpp}(n, q) = C_{\text{cell}} M_X (2n + q). \tag{19}$$

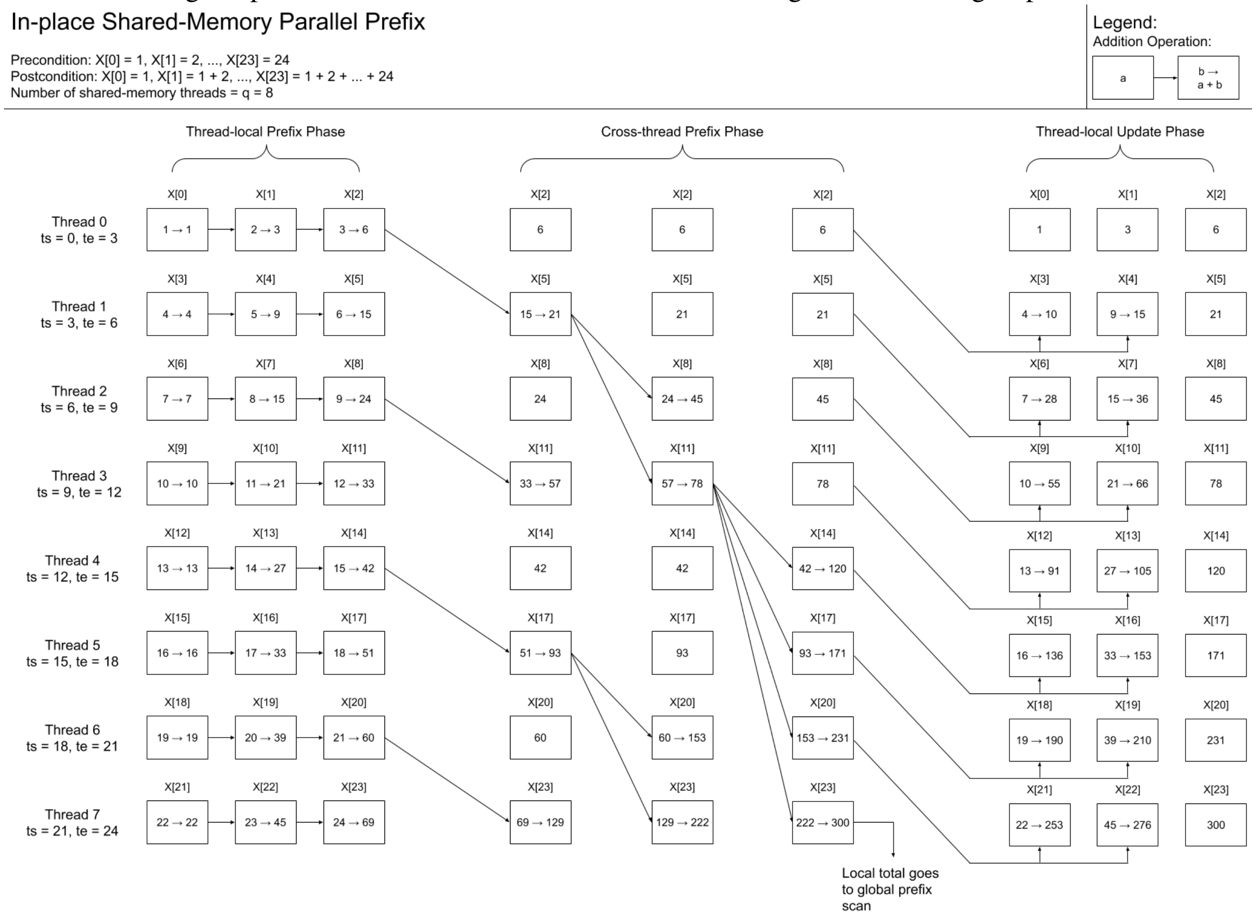
4.3.1 Optimizations for in-place prefix operations

The prefix scan in the Right-hand side Dependent Phase is a prefix sum of the last column of $Q_i^{-1} F_i$ for all i, in order to determine Z_i'. In this case, X is the array of the last columns of all $Q_i^{-1} F_i$, and the operation $\odot$ is vector addition. Vector addition can be performed in place.

However, the specific requirements of the Right-hand Side Dependent Phase of the Accelerated Recursive Doubling Algorithm causes Algorithm 3 to have the following drawbacks:

Figure 1. In-place Shared-Memory Parallel Prefix

Calculating the prefix sum of the first 24 natural numbers using 8 threads using in-place addition


In-place Shared-Memory Parallel Prefix
Precondition: X[0] = 1, X[1] = 2, ..., X[23] = 24
Postcondition: X[0] = 1, X[1] = 1 + 2, ..., X[23] = 1 + 2 + ... + 24
Number of shared-memory threads = q = 8
Legend:
Addition Operation:
a
b → a + b
Thread-local Prefix Phase
Cross-thread Prefix Phase
Thread-local Update Phase
Thread 0 ts = 0, te = 3
Thread 1 ts = 3, te = 6
Thread 2 ts = 6, te = 9
Thread 3 ts = 9, te = 12
Thread 4 ts = 12, te = 15
Thread 5 ts = 15, te = 18
Thread 6 ts = 18, te = 21
Thread 7 ts = 21, te = 24
Local total goes to global prefix scan

1. The parallel prefix algorithms in ARDA do not require X to be preserved, while the algorithm preserves X and P.

2. The algorithm does not take advantage of the fact that the operation $\odot$ is in-place.

These drawbacks can be mitigated by modifying Algorithm 3 as shown in Figure 1 for in-place operations. Figure 1 shows the parallel prefix algorithm for in-place addition on the first 24 natural numbers. The following modifications can be made:

Algorithm 5: Modified Parallel Prefix Algorithm for in-place operations

```
1. The prefix is calculated in-place, clobbering the original values of $X$. This saves the memory
   required for $P$.
2. The algorithm removes the need for $TP$ by using $X[te_t - 1]$ to store the value that would
   otherwise be stored in $TP[t]$.
```

Since this modification eliminates the need to copy values to TP, this reduces the time complexity further to

$$T_{ipp}(n,q) = \begin{cases} \left(2\frac{n}{q} + \log_2(q)\right) T_{\odot} & q > 1 \\ nT_{\odot} & q = 1 \end{cases}. \tag{20}$$

Since the arrays P and TP is eliminated, the memory complexity is reduced to

$$M_{ipp}(n,q) = nC_{\text{cell}}M_X. \tag{21}$$

Note on threads: The number of concurrent threads running during the Cross-thread Prefix Phase is reduced to $\lfloor \frac{q}{2} \rfloor$. This means that thread synchronization is faster, and more memory bandwidth is available to the prefix operations.

The number of threads performing during the Thread-local Update Phase is reduced to $q-1$. Since the local total available at $X[te_{q-1}-1]$, the q-th thread can be assigned to performing the cross-rank parallel prefix operation, providing for some computation-communication overlap.

4.3.2 Optimizations for out-of-place prefix operations

The prefix scan in the Right-hand side Independent Phase is a prefix product of C'_i over all i to determine Q''_i. In this case, X is the array of C'_i, and the operation $\odot$ is matrix multiplication. Matrix multiplication, as implemented in BLAS `_gemm` routines, is performed out of place, unlike vector addition.

To make the algorithm behave like as the operation was in place, we could store the result of the matrix multiplication in auxiliary memory and then copy the result, but one copy per operation can be expensive as the size of the matrix increases.

Also, we cannot use X as a substitute for TP as in the case of the out-of-place prefix operations. The cross-thread prefix phase requires multiplying Q'-type (single-prime) matrices, which are inferred from Q''-type (double-prime) or C'-type stored in X.

However, we have two thread registers that can store q matrices of type Q', and we can use both R^{01} and R^{34} to store Q'' type registers. Therefore, we can make the following modifications to the algorithm, illustrated in Figure 2 and described below.

Algorithm 6: Modified Parallel Prefix Algorithm for out-of-place operations

1. Every cell in the array X and TP has two sections, each with an associated color: white and black. All $X[i]$ are currently in a white section, and the black sections are all empty.
2. Whenever $X[i]$ or $TP[i]$ is used as an operand, the value in the white section is used.
3. Whenever $X[i]$ or $TP[i]$ is written to, the value is stored in the black section. Then, the colors of the sections of that cell are flipped.
4. If $q = 1$, the colors of the every section are flipped once after the Thread-local Prefix Phase.

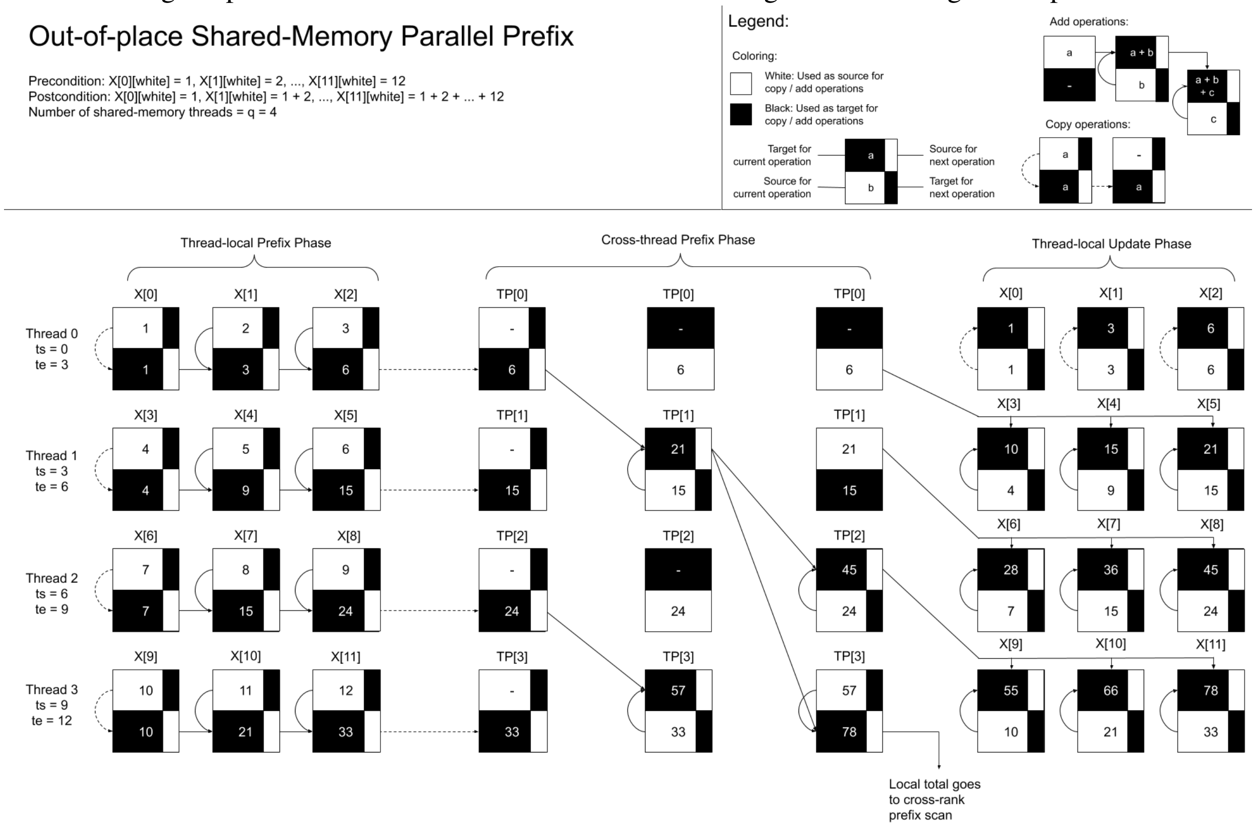
The copy between X and TP in Figure 2 can be eliminated by referring to $X[te_t - 1]$ for the value at $TP[t]$ until $TP[t]$ is written to.

The time complexity of the algorithm is therefore:

$$T_{oopp}(n,q) = \begin{cases} \left(2\frac{n}{q} + \log_2(q)\right) T_{\odot} & q > 1 \\ nT_{\odot} & q = 1 \end{cases}. \tag{22}$$

Figure 2. Out-of-place Shared-Memory Parallel Prefix

Calculating the prefix sum of the first 12 natural numbers using 4 threads using out-of-place addition


Out-of-place Shared-Memory Parallel Prefix
Precondition: X[0][white] = 1, X[1][white] = 2, ..., X[11][white] = 12
Postcondition: X[0][white] = 1, X[1][white] = 1 + 2, ..., X[11][white] = 1 + 2 + ... + 12
Number of shared-memory threads = q = 4
Legend:
Coloring:
White: Used as source for copy / add operations
Black: Used as target for copy / add operations
Target for current operation
Source for current operation
Source for next operation
Target for next operation
Add operations:
Copy operations:
Thread-local Prefix Phase
Cross-thread Prefix Phase
Thread-local Update Phase
Thread 0 ts = 0 te = 3
Thread 1 ts = 3 te = 6
Thread 2 ts = 6 te = 9
Thread 3 ts = 9 te = 12
Local total goes to cross-rank prefix scan

The memory complexity is

$$M_{oopp} = 2C_{\text{cell}} M_X (n + q). \tag{23}$$

Note on threads: Like in the in-place case, the number of concurrent threads running during the Cross-thread Prefix Phase is reduced to $\lfloor \frac{q}{2} \rfloor$. However, the number of threads performing during the Thread-local Update Phase remains q. But, as in Figure 2, Thread 0 is expected to finish early since copying is faster than performing operations like matrix multiplications. Thread 0 can then perform the cross-rank parallel prefix for some computation-communication overlap.

5. Program

5.1 Independent Phase Part 1

In the first part of the Independent Phase, we have received L_i, D_i, and U_i, and we calculate C'_i. For any given k, the calculation of C'_k only relies on L_k, D_k, and U_k. Therefore, we can complete this part using an embarrassingly parallel algorithm as shown in Algorithm 9.

Algorithm 7: Independent Phase Part 1

```
for $t \leftarrow 0$ to $q - 1$ in parallel do
    for $k \leftarrow ts_t$ to $te_t - 1$ do
        Execute each instruction in Table 1;
    end
end
```

Table 1. Instructions for Independent Phase Part 1

Instruction	R_k^0	R_k^1	R_k^2	R_k^3	R_k^4	R_k^{t0}	Complexity
—	L_i	D_i	U_i	—	—	—	—
$R_k^2 \leftarrow \texttt{inv}(R_k^2, R_t^{t0})$	L_i	D_i	U_i^{-1}	—	—	—	$C_{\text{inv}} M^3$
$R_k^{34}[0] \leftarrow -R_k^2 R_k^1$	L_i	D_i	U_i^{-1}	$\left(C_i^{11}, —\right)$		—	$C_{\text{mul}} M^3$
$R_k^{34}[1] \leftarrow -R_k^2 R_k^0$	L_i	D_i	U_i^{-1}	$\left(C_i^{11}, C_i^{12}\right) = C'_i$		—	$C_{\text{mul}} M^3$

The time complexity of first part of the independent phase is

$$T_{ip1}(M, N, P, q) = \frac{N}{Pq} \left(M^3 (C_{\text{inv}} + 2C_{\text{mul}}) \right). \tag{24}$$

5.2 Independent Phase Part 2

In the second part of the Independent Phase, we have C'_i, and we must calculate Q''_i. This can be done efficiently using a parallel prefix scan. The prefix scan has the following parts:

1. Compute the shared-memory prefix product of all C'_i on a given processing element to obtain 'local' Q''_i.
2. Compute the distributed-memory prefix product of the 'local' Q'_i's that represent the local total.
3. Update the 'local' Q''_i using the result of the previous step to get Q''_i.

Step 1 is implemented using Algorithm 6. The sections of $X[i]$ are $R^{34}[i]$ and $R^{01}[i]$ with the R^{34} section initially colored white. The sections of $TP[i]$ are $R^{t0}[i]$ and $R^{t1}[i]$ with the R^{t0} section initially colored white.

For the particular application of calculating the prefix sum of C_i matrices, we can further optimize the storage and computation time for the parallel prefix algorithm. In the Thread-local Prefix and the Thread-local Update stages, the resultant matrices are products of collections of C_i matrices. This means we can omit calculating the second block rows of the resultant matrices, and infer the second block rows from previous matrices. This means that the cost of the prefix operation is not uniform over the prefix operation.

This means Equation (22) cannot be used to express the time complexity of this step. The time complexity can be calculated to be:

$$\begin{aligned} T_{ip21}(M,N,P,q) &= T_{C'\times Q''}(M)\frac{N}{Pq} + \begin{cases} T_{Q'\times Q'}(M)\log_2(q) + T_{Q''\times Q'}(M)\frac{N}{Pq} & q > 1 \\ 0 & q = 1 \end{cases} \\ &= M^3 C_{\text{mul}} \begin{cases} 8\frac{N}{pq} + 8\log_2(q) & q > 1 \\ 4\frac{N}{q} & q = 1 \end{cases}. \end{aligned} \tag{25}$$

Step 2 is implemented using a distributed-memory library function like `MPI_Exscan` using a simulated in-place matrix multiplication operator, which first multiplies two matrix into some auxiliary storage (R_0^{t0}) and copies the result into the memory held by an operand. The result is received in the receive register R^r. The time complexity of this step is therefore:

$$T_{ip22}(M,N,P,q) = (M^3(8C_{\text{mul}}) + M^2(4C_{\text{copy}}))\log_2(P) + (\tau + 4\mu M^2)\log_2(P). \tag{26}$$

The $4M^2C_{\text{copy}}$ term can be eliminated by using a parallel prefix method optimized for out-of-place operations like in Step 1. However, this was not done in this study because the convenience of using a library function outweighed the potential gain by the complex optimization.

Step 3 can be written as an embarrassingly parallel algorithm as given in Algorithm 8.

Algorithm 8: Independent Phase Part 2 Step 3

```
for $t \leftarrow 0$ to $q-1$ do
    for $k \leftarrow ts_t$ to $te_t - 1$ do
        $R_k^{34} \leftarrow R_k^{01} R^r$;
    end
end
```

Therefore, the time complexity of this step

$$T_{ip23}(M,N,P,q) = M^3\left(4\frac{N}{Pq}C_{\text{mul}}\right). \quad (27)$$

The time complexity of the second part of the independent phase is

$$T_{ip}(M,N,P,q) = M^2(4C_{\text{copy}} + 4\mu)\log_2 P + \tau\log_2 P + \quad (28)$$

$$M^3\left(8C_{\text{mul}}\log_2 P + \begin{cases}\frac{N}{Pq}(12C_{\text{mul}}) + 8C_{\text{mul}}\log_2 q & q>1\\ \frac{N}{P}(8C_{\text{mul}}) & q=1\end{cases}\right). \quad (29)$$

5.3 Independent Phase Part 3

This phase can be implemented using an embarrasingly parallel algorithm as follows:

Algorithm 9: Independent Phase Part 1

```
for $t \leftarrow 0$ to $q-1$ in parallel do
    for $k \leftarrow ts_t$ to $te_t - 1$ do
        Execute each instruction in Table 1;
    end
end
```

In the following table, `inv`(R^A, R^B) is a function that inverts the matrix stored in R^A, in place, using R^B as a workspace.

Table 2. Instructions for Independent Phase Part 3

Instruction	R_k^0	R_k^1	R_k^{t0}	R_k^{t1}	Complexity
$R_t^{t0} \leftarrow R_{K-k}^{34}[0]$	—	—	A	—	$C_{\text{copy}}M^2$
$R_t^{t0} \leftarrow \texttt{inv}(R_k^{t0}, R_t^{t1})$	—	—	A^{-1}	—	$C_{\text{inv}}M^3$
$R_k^0 \leftarrow R_{K-k+1}^{34}[0]R_t^{t0}$	CA^{-1}	—	A^{-1}	—	$C_{\text{mul}}M^3$
$R_t^{t1} \leftarrow R_{K-k+1}^{34}[1]$	CA^{-1}	—	A^{-1}	D	$C_{\text{copy}}M^2$
$R_t^{t1} \leftarrow R_t^{t1} - R_k^0R_{K-i}^{34}[1]$	CA^{-1}	—	A^{-1}	V	$C_{\text{mul}}M^3$
$R_t^{t1} \leftarrow \texttt{inv}(R_t^{t1}, R_k^1)$	CA^{-1}	—	A^{-1}	V^{-1}	$C_{\text{inv}}M^3$
$R_k^1 \leftarrow -R_t^{t0}R_k^0$	CA^{-1}	$-V^{-1}CA^{-1}$	A^{-1}	V^{-1}	$C_{\text{mul}}M^3$
$R_t^{t1} \leftarrow -R_{K-k}^{34}[1]R_k^1$	CA^{-1}	$-V^{-1}CA^{-1}$	A^{-1}	W	$C_{\text{mul}}M^3$
$R_t^{t1} \leftarrow R_t^{t1} + I$	CA^{-1}	$-V^{-1}CA^{-1}$	A^{-1}	$I+W$	$C_{\text{add}}M$
$R^0 \leftarrow R_t^{t0}R_t^{t1}$	$A^{-1}(I+W)$	$-V^{-1}CA^{-1}$	A^{-1}	$I+W$	$C_{\text{mul}}M^3$

The final results in R_k^0 and R_k^1 form the first column of the inverse of Q_i, as described in the Optimizations in the Independent Phase section.

The time complexity of the third part of the independent phase is

$$T_{ip3}(M, N, P, q) = (M^3(2C_{\text{inv}} + 5C_{\text{mul}}) + M^2(2C_{\text{copy}}) + M(C_{\text{add}}))\frac{N}{Pq}. \tag{30}$$

5.4 Dependent Phase Part 1

In the first part of the Dependent Phase, we have b_i, U_i^{-1}, and the first column of Q_i^{-1} and we calculate z_i. For any given k, the calculation of z_k only relies on b_i, U_k^{-1}, and the first column of Q_k^{-1}. Therefore, we can implement this part using an embarrassingly parallel algorithm as shown in Algorithm 10.

Algorithm 10: Dependent Phase Part 1

```
for $t \leftarrow 0$ to $q - 1$ in parallel do
    for $k \leftarrow ts_t$ to $te_t - 1$ do
        Execute each instruction in Table 3;
    end
end
```

Table 3. Instructions for Dependent Phase Part 1

Instruction	R_k^{v0}	R_k^{v1}	R_k^{v2}	Complexity
—	b_i	—	—	—
$R_k^{v2} \leftarrow R^2 R^{v0}$	b_i	—	F_i^{13}	$C_{\text{mul}}M^2$
$R_k^Z[0] \leftarrow R_k^0 R_k^{v2}$	$\left((Q_i^{-1}F_i)^{13}, —\right)$		F_i^{13}	$C_{\text{mul}}M^2$
$R_k^Z[1] \leftarrow R_k^1 R_k^{v2}$	$\left((Q_i^{-1}F_i)^{13}, (Q_i^{-1}F_i)^{23}\right) = z_i$		F_i^{13}	$C_{\text{mul}}M^2$

The time complexity of this part of the algorithm is

$$T_{dp1}(N, M, P, q) = \frac{N}{Pq}(M^2(3C_{\text{mul}})). \tag{31}$$

5.5 Dependent Phase Part 2

In the second part of the Dependent Phase, we have z_i, and we must calculate Z_i'. This can be done using a parallel prefix scan. The prefix scan has the following parts:

1. Compute the shared-memory prefix sum of all z_i to obtain ‘local’ Z_i.
2. Compute the distributed-memory prefix sum of the ‘local’ Z_i’s that represent the local totals.

3. Update the 'local' Z_i using the result of the previous step to get Z_i'.
4. On the processing element that has Q_N, calculate $x_1 = -(Q_N^{11})^{-1}F_N$ and broadcast it to all processing elements.

Step 1 is implemented using Algorithm 5. The register R^Z is represented by X.

Substituting $T_\odot = M(2C_{\text{add}})$, $n = \frac{N}{P}$ in Equation (20), we get

$$T_{dp21}(M,N,P,q) = \begin{cases} M\left(2\frac{N}{Pq} + \log_2(q)\right)(2C_{\text{add}}) & q > 1 \\ M\left(\frac{N}{P}(2C_{\text{add}})\right) & q = 1 \end{cases}. \tag{32}$$

Step 2 is implemented using a distributed-memory library function like `MPI_Exscan` using a vector addition operator, where the result is received in R_{v0}^{t0}. The time complexity of this step is therefore:

$$T_{dp22}(M,N,P,q) = M(2C_{\text{add}})\log_2(P) + (\tau + 2\mu M)\log_2(P). \tag{33}$$

Step 3 can be implemented as an embarrassingly parallel algorithm as given in Algorithm 11.

Algorithm 11: Dependent Phase Part 2 Step 3

```
for $t \leftarrow 0$ to $q-1$ do
    for $k \leftarrow ts_t$ to $te_t - 1$ do
        $R_k^Z \leftarrow R_k^Z + R_{v0}^{t0}$;
    end
end
```

Therefore, the time complexity of this step is

$$T_{dp23}(M,N,P,q) = M\left(\frac{N}{Pq}C_{\text{add}}\right). \tag{34}$$

Step 4 can be implemented as in Algorithm 12.

Algorithm 12: Dependent Phase Part 2 Step 4

```
if $N$ is on current processing element then
    $R^{t1}[0]^{11} \leftarrow \texttt{inv}(R^{34}[K_r - 1], R^{t1}[0]^{21})$;
    $R_{v1}^{t0} \leftarrow -R^{t1}[0]^{11} R_{K_r-1}^{v2}$;
end
Broadcast $R_{v1}^{t0}$, receiving in $R_{v1}^{t0}$;
```

The time complexity of this step is

$$T_{dp24}(M,N,P,q) = C_{\text{inv}}M^3 + C_{\text{mul}}M^2 + (\tau + \mu M)\log_2 P. \tag{35}$$

The total time complexity of the second part of the dependent phase is therefore

$$T_{dp}(M,N,P,q) = \tau\log_2(P) + M\left(2(C_{\text{add}} + \mu)\log_2(P) + \begin{cases} \frac{N}{Pq}(5C_{\text{add}}) + 2C_{\text{add}}\log_2(q)) & q > 1 \\ \frac{N}{P}(3C_{\text{add}}) & q = 1 \end{cases}\right). \tag{36}$$

5.6 Dependent Phase Part 3

In the third part of the Dependent Phase, we have Z'_i, x_1, and Q''_i, and we must calculate x_i. For any given k, the calculation of x_k only relies on Z'_k, x_1, and Q''_{k-1}. Therefore, we can implement this part using an embarrassingly parallel algorithm as shown in Algorithm 13.

Algorithm 13: Dependent Phase Part 3

```
for $t \leftarrow 0$ to $q-1$ in parallel do
    for $k \leftarrow ts_t$ to $te_t - 1$ do
        Execute each instruction in Table 4;
    end
end
```

Table 4. Instructions for Dependent Phase Part 3

Instruction	R^Z_k	R^{v2}_k	Complexity
—	Z'_i	F^{13}_i	—
$R^Z_k \leftarrow R^Z_k + \begin{bmatrix} x_1 & 0 \end{bmatrix}^T$	$Z'_i + \begin{bmatrix} x_1 & 0 \end{bmatrix}^T$	F^{13}_i	$C_{\text{add}}M$
$R^{v2}_k \leftarrow R^{v2}_k + R^{34}_{K-i+1}R^Z_{k-1}$	$Z'_i + \begin{bmatrix} x_1 & 0 \end{bmatrix}^T$	x_i	$2C_{\text{mul}}M^2$

The time complexity of this part of the algorithm is

$$T_{dp3}(M,N,P,q) = \frac{N}{Pq}\left(M^2(2C_{\text{mul}}) + MC_{\text{add}}\right). \tag{37}$$

6. Time Complexity Analysis and Comparison

6.1 Naïve implementation

The naïve implementation is considered to use $M \times M$ matrices to represent L, D and U matrices, $2M \times 2M$ matrices to represent C and Q matrices, while using $2M \times 1$ matrices to represent F and Z matrices.

To determine the time complexity of the dependent phase of the naïve implementation, we list the time-complexities of the naïve implementations of each step, and then sum the complexities.

6.1.1 Independent phase

Step	Complexity
Invert U's	$\frac{N}{P}(C_{\text{inv}}M^3)$
Calculate C's using L, D, and U^{-1}	$\frac{N}{P}(2C_{\text{mul}}M^3)$
Prefix C's locally to find local Q's	$\frac{N}{P}(8C_{\text{mul}}M^3)$
Cross-rank prefix scan on Q	$(\tau + 4\mu M^2 + 8C_{\text{mul}}M^3)\log_2(P)$
Update Q's locally to find global Q's	$\frac{N}{P}(8C_{\text{mul}}M^3)$
Invert Q's	$\frac{N}{P}(8C_{\text{inv}}M^3)$

The total time complexity for the independent phase is

$$T_{ni}(M, N, P) = M^3\left(\frac{N}{P}(9C_{\text{inv}} + 18C_{\text{mul}})) + 8C_{\text{mul}}\log_2(P)\right) + \Theta(M^2\log_2 P) \tag{38}$$

6.1.2 Dependent Phase

Step	Complexity
Calculate F's	$\frac{N}{P}(C_{\text{mul}}M^2)$
Calculate V's	$\frac{N}{P}(2C_{\text{mul}}M^2)$
Prefix V's locally to find local Z's	$\frac{N}{P}(2C_{\text{add}}M)$
Cross-rank prefix scan on Z	$(\tau + 2\mu M + 2C_{\text{add}}M)\log_2 P$
Update Z's locally to find global Z's	$\frac{N}{P}(2C_{\text{add}}M)$
Calculate E's using Z, Q and F	$\frac{N}{P}(2C_{\text{mul}}M^2 + 2C_{\text{add}}M)$
Calculate x_1	$C_{\text{inv}}M^3 + C_{\text{mul}}M^2$
Broadcast x_1	$(\tau + \mu M)\log_2 P$
Calculate x_i using x_0, Q, and E	$\frac{N}{P}C_{\text{mul}}(4M^2 + 2M)$

The total time complexity for the dependent phase is

$$T_{nd}(M, N, P) = M^3C_{\text{inv}} + M^2\left(\frac{N}{P}(9C_{\text{mul}}) + C_{\text{mul}}\right) + \Theta\left(\frac{MN}{P} + M\log_2 P\right) \tag{39}$$

6.2 Optimized implementation

6.2.1 Independent phase

Summing up the time complexities given in Equations (24), (29), and (30), we get:

$$\begin{aligned}
T_{oi}(M,N,P,q) =& M^3\left(8C_{\text{mul}}\log_2 P + \frac{N}{Pq}(3C_{\text{inv}}) + \begin{cases}\frac{N}{Pq}19C_{\text{mul}} + 8C_{\text{mul}}\log_2 q & q>1\\ \frac{N}{P}15C_{\text{mul}} & q=1\end{cases}\right)+\\
& M^2\left(\frac{N}{Pq}(2C_{\text{copy}}) + (4C_{\text{copy}}+4\mu)\log_2 P\right) + M\frac{N}{Pq}C_{\text{add}} \qquad (40)\\
=& M^3\left(8C_{\text{mul}}\log_2 P + \frac{N}{Pq}(3C_{\text{inv}}) + \begin{cases}\frac{N}{Pq}19C_{\text{mul}} + 8C_{\text{mul}}\log_2 q & q>1\\ \frac{N}{P}15C_{\text{mul}} & q=1\end{cases}\right)+\\
& \Theta\left(\frac{M^2N}{Pq} + M^2\log_2 P\right). \qquad (41)
\end{aligned}$$

Setting $q = 1$, we get

$$T_{oi}(M,N,P) = M^3\left(\frac{N}{P}(15C_{\text{mul}} + 3C_{\text{inv}}) + 8C_{\text{mul}}\log_2 P\right) + \Theta\left(\frac{M^2N}{P} + M^2\log_2 P\right) \qquad (42)$$

6.2.2 Dependent phase

Summing up the time complexities given in Equations (31), (36), and (31), we get:

$$\begin{aligned}
T_{od}(M,N,P,q) =& M^3C_{\text{inv}} + M^2\left(\frac{N}{Pq}(5C_{\text{mul}}) + C_{\text{mul}}\right)+\\
& 2\tau\log_2 P + M\left((2C_{\text{add}}+3\mu)\log_2 P + 2C_{\text{add}}\begin{cases}3\frac{N}{Pq} + \log_2 q & q>1\\ 2\frac{N}{P} & q=1\end{cases}\right) \qquad (43)\\
=& M^3C_{\text{inv}} + M^2\left(\frac{N}{Pq}(5C_{\text{mul}}) + C_{\text{mul}}\right) + \Theta\left(\frac{MN}{Pq} + M\log_2 P + M\log_2 q\right) \qquad (44)
\end{aligned}$$

Setting $q = 1$, we get

$$T_{od}(M,N,P) = M^3C_{\text{inv}} + M^2\left(\frac{N}{P}(5C_{\text{mul}}) + C_{\text{mul}}\right) + \Theta\left(\frac{MN}{P} + M\log_2 P\right) \qquad (45)$$

6.3 Comparison

In order to compare the two algorithms, we assume that the number of shared-memory threads is $q = 1$, and $C_{\text{inv}} = C_{\text{mul}} = C$. For the two problem parameters M and N, we choose the dominantly growing term that grows along with that parameter.

We then measure the asymptotic speedup defined as the limit

$$\lim_{\text{parameter}\to\infty} \frac{\text{Dominant term of the naïve implementation}}{\text{Dominant term of the optimized implementation}}. \qquad (46)$$

6.3.1 Independent Phase

Parameter	Dominant term (Optimized)	Dominant term (Naïve)	Asymptotic Speedup
M	$18CM^3\frac{N}{P} + 8CM^3\log_2 P$	$27CM^3\frac{N}{P} + 8CM^3\log_2 P$	$\frac{27N+8P\log_2 P}{18N+8P\log_2 P} = \frac{3}{2}(N >> P)$
N	$18CM^3\frac{N}{P}$	$27CM^3\frac{N}{P}$	$\frac{3}{2}$

6.3.2 Dependent Phase

Parameter	Dominant term (Optimized)	Dominant term (Naïve)	Asymptotic Speedup
M	CM^3	CM^3	1
N	$5CM^2\frac{N}{P}$	$9CM^2\frac{N}{p}$	$\frac{9}{5}$

7. Memory Complexity Analysis and Comparison

7.1 Naïve implementation

The memory complexity of the naïve implementation was estimated by summing up the sizes of the allocations relevant to the phase of the algorithm over all processing elements in the implementation of the ARDA algorithm used in [2].

For the independent phase, the memory complexity was found to be $M_{ni}(M, N, P) \geq (14M^2N + 8M^2P)C_{\text{cell}}$. For the dependent phase, the memory complexity was found to be $M_{nd}(M, N, P) \geq (6MN + 4MP)C_{\text{cell}}$.

7.2 Optimized implementation

The memory complexity of the optimized implementation can be measured simply by summing up the sizes of the registers relevant to the phase of the algorithm.

For the independent phase, the memory complexity was found to be $M_{oi}(M, N, P, q) = (5M^2N + 8M^2Pq + 4M^2P)C_{\text{cell}}$. For the dependent phase, the memory complexity was found to be $M_{od}(M, N, P, q) = (3MN + 3MPq)C_{\text{cell}}$.

7.3 Comparison

In order to compare the two algorithms, we assume that the number of shared-memory threads is $q = 1$. For the two problem parameters M, N, we choose the dominantly growing term that grows along with that parameter.

We then measure the asymptotic memory savings defined as the limit

$$\lim_{\text{parameter}\to\infty} \frac{\text{Dominant term of the optimized implementation}}{\text{Dominant term of the naïve implementation}}. \tag{47}$$

7.3.1 Independent Phase

Parameter	Dominant term (Optimized)	Dominant term (Naïve)	Asymptotic Memory Savings
M	$M^2(5N + 12P)$	$M^2(14N + 8P)$	$\frac{5N+12P}{14N+8P} = \frac{5}{14}(N >> P)$
N	$N(5M^2)$	$N(14M^2)$	$\frac{5}{14}$

7.3.2 Dependent Phase

Parameter	Dominant term (Optimized)	Dominant term (Naïve)	Asymptotic Memory Savings
M	$M(3N + 3P)$	$M(6N + 4P)$	$3\frac{\frac{N}{P}+1}{6\frac{N}{P}+4}$
N	$N(3N)$	$N(6M)$	$\frac{1}{2}$

8. Shared-memory threading speedups

We determine the effect of adding shared-memory threading to the asymptotic time-complexity of the Algorithm.

8.1 Independent phase

The asymptotic time complexity of the independent phase is

$$T_{oi}(M, N, P, q) = \Theta\left(\frac{M^3 N}{Pq} + M^3 \log_2(Pq)\right). \tag{48}$$

Therefore, the asymptotic speedup with increasing q is

$$\begin{aligned}\text{Speedup} = \frac{\text{Complexity for num threads} = 1}{\text{Complexity for num threads} = q} &= \Theta\left(\frac{\frac{M^3N}{P} + M^3\log_2(P)}{\frac{M^3N}{Pq} + M^3\log_2(Pq)}\right)\\ &= \Theta\left(\frac{q\left(\frac{N}{P} + \log_2(P)\right)}{\frac{N}{P} + q\log_2(Pq)}\right).\end{aligned}$$

For small block-row granularities $\frac{N}{P}$, we can plug in $\frac{N}{P} \approx 0$ in the above equation to obtain an asymptotic speedup of $\Theta(1)$. For large block-row granularities, we can plug in $\frac{\frac{N}{P}+\log_2 P}{\frac{N}{P}} \approx \frac{\frac{N}{P}+q\log_2(Pq)}{\frac{N}{P}} \approx 1$ to obtain an asymptotic speedup of $\Theta(q)$.

8.2 Dependent phase

The asymptotic time complexity of the dependent phase is

$$T_{od}(M, N, P, q) = \Theta\left(M^3 + \frac{M^2 N}{Pq} + M\log_2(Pq)\right). \tag{49}$$

Therefore, the asymptotic speedup with increasing q is

$$\begin{aligned}\text{Speedup} = \frac{\text{Complexity for num threads} = 1}{\text{Complexity for num threads} = q} &= \Theta\left(\frac{M^3 + \frac{M^2N}{P} + M\log_2 P}{M^3 + \frac{M^2N}{Pq} + M\log_2(Pq)}\right)\\ &= \Theta\left(\frac{q\left(\frac{MN}{P} + M^2 + \log_2 P\right)}{\frac{MN}{P} + q(M^2 + \log_2(Pq))}\right).\end{aligned}$$

For small block-cell granularities $\frac{MN}{P}$, we can plug in $\frac{MN}{P} \approx 0$ in the above equation to obtain an asymptotic speedup of $\Theta(1)$. For large block-cell granularities, we can plug in $\frac{\frac{MN}{P}+M^2+\log_2 P}{\frac{MN}{P}} \approx \frac{\frac{MN}{P}+q(M^2+\log_2(Pq))}{\frac{MN}{P}} \approx 1$ to obtain an asymptotic speedup of $\Theta(q)$.

9. Experiments and Results

The optimized algorithm described was implemented in C, using BLAS and LAPACK calls for matrix and vector operations. The solutions generated by the algorithm were measured for accuracy by the error function $E(x) = \log_2 \frac{\|Ax^* - b\|}{MN}$ where x^* is the solution given by the algorithm. The smaller the value of $E(x^*)$, the more accurate the solution is.

The results were as follows:

Table 5. Mean error for $M = 2, N = 20$

P	Mean $E(x)$
6	−22.177
8	−24.389
10	−21.317
12	−21.318

Table 6. Mean error for $M = 3, N = 9$

P	Mean $E(x)$
6	−21.518
8	−22.116
10	−21.232
12	−25.232

The optimized implementation works correctly, with the magnitude of the error $= 2^{E(x^8)} \leq 10^{-7}$ for the above problem sizes.

However, for larger problem sizes, we get

Table 7. Mean error for $M = 3, N = 22$

P	Mean $E(x)$
6	9.020
8	9.740
10	9.740
12	9.087

The error continues to rise rapidly with increases in M and N, and $E(x)$ approaches 225 for $N = 20$, $M = 80$. The rapidly increasing error poses a significant challenge to the practical use of this algorithm. The possible causes have been identified as

1. The blockwise inverting algorithm used in Optimization 2 to efficiently calculate only the first block column
2. The limited precision to which we can store large numbers resulting from prefix products of large numbers of matrices.

New techniques need to be developed to keep the error within practical limits.

10. Conclusions

We presented alternative mathematical formulations of intermediary matrices in the algorithm. We also made the assumptions made by the algorithm explicit. This helped us determine that a particular algorithm cited in [2] will not run.

We have shown methods by which the Accelerated Recursive Doubling Algorithm can be optimized, and shown how the speedup and memory savings is affected by the value of the problem parameters M, N and the new parameter q. We can achieve up to $\frac{3}{2}$ speedup on the independent phase and up to $\frac{9}{5}$ speedup on the dependent phase. Since the dependent phase is expected to run 10^2–10^4 times, the overall speedup is close to $\frac{9}{5}$.

We implemented the optimized algorithm and verified that runs with acceptable errors for small matrices. We also noted the numerical instability of the algorithm. Resolving this numerical stability is the direction of our research into this algorithm.

References

[1] S. K. Seal, K. P. Perumalla, and S. P. Hirshman, "Revisiting Parallel Cyclic Reduction and Parallel Prefix-based Algorithms for Block Tridiagonal Systems of Equations," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 273–280, 2013.

[2] S. K. Seal, "An accelerated recursive doubling algorithm for block tridiagonal systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1019–1028.