Parallel Cyclic Reduction

Rachit Kumar
IIT Madras

Rudra Panch
IIT Madras

Raul Om Deepak
IIT Madras

August 19, 2024

Abstract

The Parallel Cyclic Reduction (PCR) algorithm is a powerful numerical technique employed in scientific and engineering domains for solving **tridiagonal linear systems** efficiently.

Our project focuses on implementing, optimizing, and benchmarking the **Parallel Cyclic Reduction (PCR)** algorithm using OpenACC, a parallel programming model for GPUs. PCR efficiently solves **tridiagonal linear systems** by recursively reducing them until direct solution is feasible. We aim to understand PCR deeply and translate this understanding into practical code.

OpenACC enables **high-level parallel programming**, leveraging GPU computational power.

Accurate assessment of PCR's efficiency is crucial. We compare its performance with **NVIDIA's PCR routines**. Beyond basic implementation, we extend PCR to compact schemes like the Four ordered Pade scheme for 1D problems and finding partial derivatives of 3D curves for 2D problems, showcasing its versatility across numerical methods.

PCR is applied to various tridiagonal systems, including Eigenvalue Problems, Boundary Value Problems, and Finite Difference Equations, demonstrating its relevance in scientific and engineering domains.

1 Introduction

The Parallel Cyclic Reduction (PCR) algorithm is a cornerstone in numerical techniques, renowned for efficiently solving tridiagonal linear systems—a ubiquitous task in scientific and engineering domains. Our project embarks on dissecting, implementing, optimizing, and benchmarking PCR using OpenACC, a parallel programming model tailored for accelerators like GPUs.

PCR operates through recursive reduction, partitioning tridiagonal systems into smaller subproblems. Each step combines adjacent equations, streamlining them for direct solution. Leveraging OpenACC's high-level parallel programming, we tailor strategies for 1D and 2D problems, focusing on efficient data partitioning, parallel forward reduction, concurrent back substitution, and boundary condition handling.

Rigorous benchmarking against NVIDIA's PCR routines gauges our implementation's efficiency. We extend PCR's application to diverse compact schemes, showcasing its versatility in numerical methodologies. Spanning Eigenvalue Problems, Boundary Value Problems, and Finite Difference Equations, PCR's relevance across scientific domains becomes evident.

Our report navigates PCR's implementation intricacies, optimization strategies, and efficacy across varied applications, unveiling its profound impact on computational methodologies.

2 Cyclic Reduction (CR)

Cyclic reduction is a numerical method for solving large linear systems by repeatedly splitting the problem. Each step eliminates even or odd rows and columns of a matrix and remains in a similar form. The elimination step is relatively expensive but splitting the problem allows parallel computation.

The method only applies to matrices that can be represented as a (block) Toeplitz matrix, such problems often arise in implicit solutions for partial differential equations on a lattice.

Let's rewrite the system for x instead of x_i, where $1 \leq i \leq n$:

$$\begin{aligned} b_1x + c_1y &= d_1 \\ a_2x + b_2y + c_2z &= d_2 \\ a_3y + b_3z + c_3w &= d_3 \\ &\vdots \\ a_{n-1}v + b_nw &= d_n \end{aligned}$$

In matrix form:

$$\begin{pmatrix} b_1 & c_1 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & \cdots & 0 \\ 0 & a_3 & b_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n & b_n \end{pmatrix} \begin{pmatrix} x \\ y \\ \vdots \\ v \\ w \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

2.1 Forward Reduction

In Forward Reduction we reduced number of equation from n to n/2 in each step untill it converges. Eliminate variables $x_1, y_1, z_1 \ldots, x_n, y_n, z_n$ by combining equations.

$$\alpha_i = -a_i^{k-1}/b_{i-2^{k-1}}^{k-1}$$

$$\beta_i = -c_i^{k-1}/b_{i+2^{k-1}}^{k-1}$$

$$a_i^k = \alpha_i a_{i-2^{k-1}}^{k-1}$$

$$b_i^k = b_i^{k-1} + \alpha_i c_{i-2^{k-1}}^{k-1} + \beta_i a_{i+2^{k-1}}^{k-1}$$

$$c_i^k = \beta_i c_{i+2^{k-1}}^{k-1}$$

$$d_i^k = d_i^{k-1} + \alpha_i d_{i-2^{k-1}}^{k-1} + \beta_i d_{i+2^{k-1}}^{k-1}$$

Considering boundary conditions separately and accordingly.

2.2 Back Substitution

Solve for x from the equations obtained in the forward reduction step. Substitute it back in the equations;

$$\alpha_i = -a_i^{k-1}/b_{i-2^k}^{k-1}$$

$$\beta_i = -c_i^{k-1}/b_{i+2^k}^{k-1}$$

$$d_i^k = d_i^{k-1} + \alpha_i a_{i-2^k}^{k-1} + \beta_i c_{i+2^k}^{k-1}$$

for boundary conditions:

- (a) when $i - 2^k < 0$:

$$d_i^k = d_i^{k-1} + \beta_i c_{i+2^k}^{k-1}$$

- (a) when $i + 2^k > n$:

$$d_i^k = d_i^{k-1} + \alpha_i a_{i-2^k}^{k-1}$$

3 Serial Code

In serial code we use simple approach, using the formulation mentioned above.

3.1 Butterfly Structure

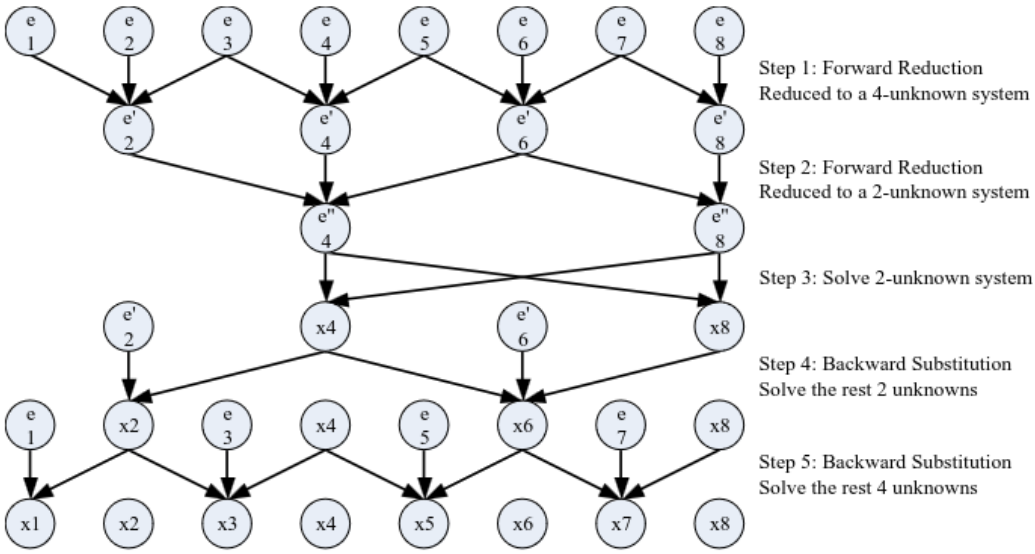


Figure 1: Cyclic Reduction

3.2 Code and Explanation

Our C code implements the cyclic reduction algorithm for solving a tridiagonal system of equations represented by the coefficients arrays a, b, c, and d, with size n.

```
void cyclic_reduction(double a[], double b[], double c[], double d[], int n)
{
    int logn;
    logn = log(n) / log(2);

    // Cyclic reduction
    for (int k = 1; k <= logn; k++)
    {
        for (int i = pow(2, k) - 1; i < n; i += pow(2, k))
        {
            // Update coefficients
            double alpha_i, beta_i;
            ....
            ....
        }
    }
```

```
    // Back substitution
    for (int k = logn - 1; k >= 0; k--)
    {
        for (int i = pow(2, k) - 1; i < n; i += pow(2, k + 1))
        {
            // Update solution using back substitution
            double sub_alpha, sub_beta;
            ....
            ....
        }
    }

    // Normalize solution
    for (int i = 0; i < n; i++)
    {
        d[i] /= b[i];
    }
}
```

4 Parallelisation Strategy

To parallelize the serial cyclic reduction program , following steps can be followed :

- **Identify Parallelizable Loops:**
  Identify the loops that can be parallelized. In this code, both the cyclic reduction loop and the back substitution loop can be parallelized. We have three Parallelizable Loops:

```
    // Cyclic reduction
    for (int k = 1; k <= logn; k++)
    {
```

Our First Loop with no loop carried dependency.

```
      #pragma acc parallel num_gangs(NUM) present(a[0:n], b[0:n], c[0:n], d[0:n])
        #pragma acc loop independent
        for (int i = pow(2, k) - 1; i < n; i += pow(2, k))
        {
            double alpha_i, beta_i;
            ....
            ....
        }
    }

// Back substitution
    for (int k = logn - 1; k >= 0; k--)
    {
```

Our Second Loop (loop for backward substitution) for specific k can be parallelised since it also doesn't have any loop carried dependency.

```
    #pragma acc parallel num_gangs(NUM) reduction(+:eqtn) present(a[0:n], b[0:n], c[0:n], d[0:n]
        #pragma acc loop independent
        for (int i = pow(2, k) - 1; i < n; i += pow(2, k + 1))
        {
            double sub_alpha, sub_beta;
```

```
        ....
        ....
    }
}
```

Our Third Loop (loop for final solution) can be parallelised since it also doesn't have any loop carried dependency.

```
// Normalize solution
#pragma acc parallel num_gangs(NUM)
#pragma acc loop gang
for (int i = 0; i < n; i++)
{
    d[i] /= b[i];
}
```

- **Directive Placement:** Placing OpenACC directive in the starting of each loop which can be parallelized.

```
#pragma acc data copyin(a[0:n], b[0:n], c[0:n], d[0:n]) copyout(d[0:n])
{
    cyclic_reduction(a, b, c, d, n, NUM);
}
```

- **Time Complexity:**
  In Reduction step we reduce number of equation by a factor of 2 for each step and total no. of steps are $log_2 n$ and similar to it Back Substitution also have same number of iterations.,
  Final solution loop iterates n time .
  Time complexity for serial cyclic reduction is $2 * (n/2 + n/4 + n/8 + ...n/2^{log_2 n}) + n$
  In **parallel code** we reduce number of equation by a factor of 2 for each step but this work get distributed between P Gangs.
  Time complexity for Parallel cyclic reduction is $(2*(n/2+n/4+n/8+....n/2^{log_2 n})+n)/P$

- **Problem:**
  In this parallel code when **number of equation become lower than number of Gangs**. Some gangs stays idle and decreases efficiency.

5 Optimised Parallel Cyclic Reduction

Our optimised code solve the problem of gangs staying ideal by breaking reduction and back substitution in three parts :

- **Forward Elimination**
  In forward elimination we keep reducing number of equations till it reaches the point where it should not be reduced further for better efficiency of code. By using following condition in loop: $eqtn/2 >= NUM$

$$\begin{aligned} eqtn/2 >= NUM \\ here, \\ eqtn = number - of - equations, \\ NUM = number - of - Gangs. \end{aligned}$$

  We had to use Reduction clause for counting number of equation per iteration.

```
#pragma acc parallel num_gangs(NUM) reduction(+:eqtn) present(a[0:n], b[0:n], c[0:n], d[0:n])
    #pragma acc loop independent
    for(int i=pow(2,k)-1;i<n;i+=pow(2,k)) { eqtn++; .... }
```

- **Recursive Doubling**
  The Recursive Doubling algorithm operates by recursively subdividing the participating processors into pairs and then doubling the distance between processors at each recursive step. This process continues until each processor has communicated with all other processors.

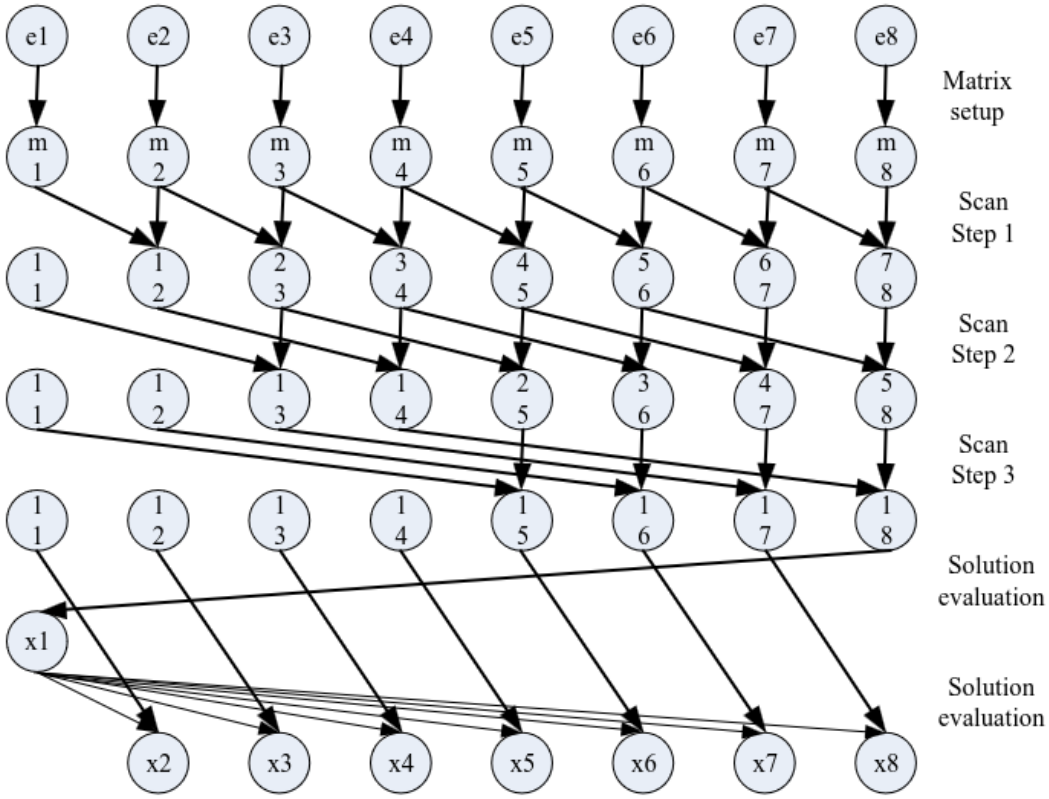


Figure 2: Recursive Doubling

```
for(int j=0;j<=log_eqtn;j++)
{
    #pragma acc parallel num_gangs(NUM) present(a[0:n], b[0:n], c[0:n], d[0:n])
    #pragma acc loop independent
    for(int i=pow(2,k)-1;i<n;i+=pow(2,k))
    {
        double alpha_i,beta_i;
        int up,dwn;
        up = i-pow(2,k+j);
        dwn= i+pow(2,k+j);
        alpha_i= (-1)*a[i]/b[up];
        beta_i= (-1)*c[i]/b[dwn];
        ....
                a[i]=a[up]*alpha_i;
                b[i]=b[i]+a[dwn]*beta_i+c[up]*alpha_i;
                c[i]=c[dwn]*beta_i;
                d[i]=d[i]+alpha_i*d[up]+beta_i*d[dwn];
```

```
            ....
            }
        }
    }
```

- **Back Substitution**
  Back substitution is same as Serial part but it has some number of equation (eqtn) already solved completely. So it has less iteration to perform for getting complete solution due to which **efficiency of PCR increases**.

- **Problem Faced in Parallelising :** The main problem was that "auto" directive was considering that loops has loop carried complex dependency due to

```
alpha_i= (-1)*a[i]/b[up];
   beta_i= (-1)*c[i]/b[dwn];
```

  which was not a loop carried dependency..We were not updating those value which are used to update values in our current Iteration.
  Also the solution we were getting was too noisy or had much error

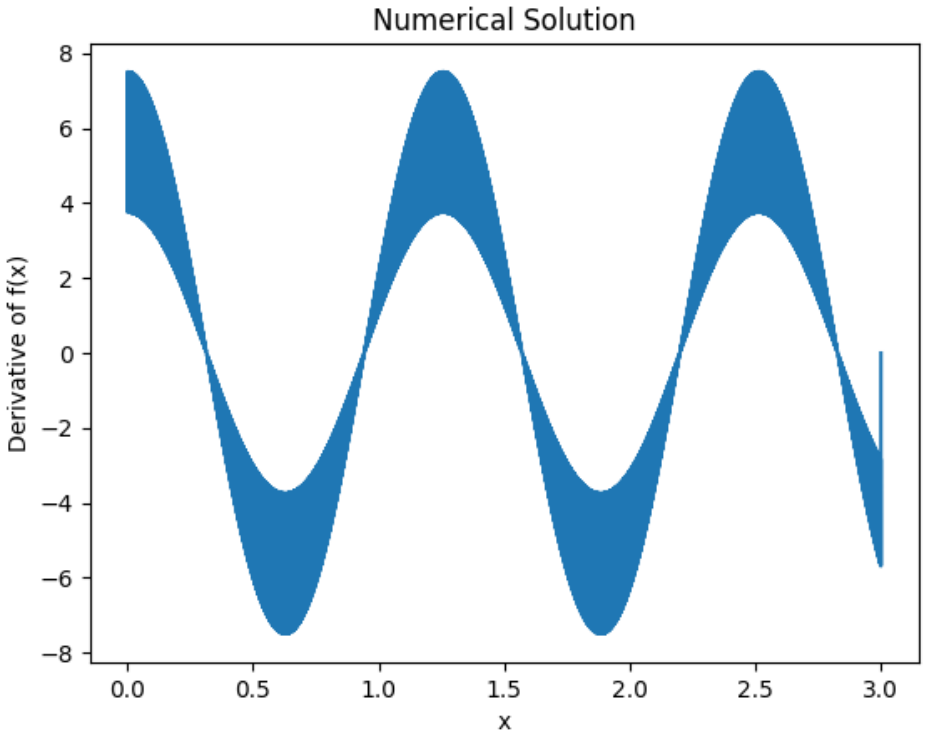


Figure 3: Numerical solution using "#pragma auto" High error

- **Solution :**
  In place of "auto" we used "independent" ;

```
#pragma acc loop independent
```

  directive to tell compiler that this loop do not have complex/simple or any kind of loop carried dependency.

6 Implementation

PCR for tri-diagonal system can be implemented in numerous ways and at many places. To implement our algorithm we only need following:

$$Ax = B$$
$$where,$$
$$A_{(n*n)} = Tridiagonal - matrix$$
$$x = Required - soltion - matrix$$
$$B = Function - Matrix$$

6.1 Pade Scheme

The Pade scheme is a type of numerical method used for solving PDEs. It's based on constructing a rational approximation to the solution of a differential equation. This scheme often involves representing the solution as a ratio of polynomials, where the numerator and denominator polynomials are constructed to approximate the behavior of the solution.
Let's take,

$$f(x) = \sin(5x)$$
$$for, 0 \leq x \leq 3.$$

We will calculate the derivative of the function using Fourth order Padé scheme for the interior and third-order accurate one-sided Padé scheme near the boundaries is given as follows:

$$f'_{j+1} + 4f'_j + f'_{j-1} = 3h(f_{j+1} - f_{j-1})$$

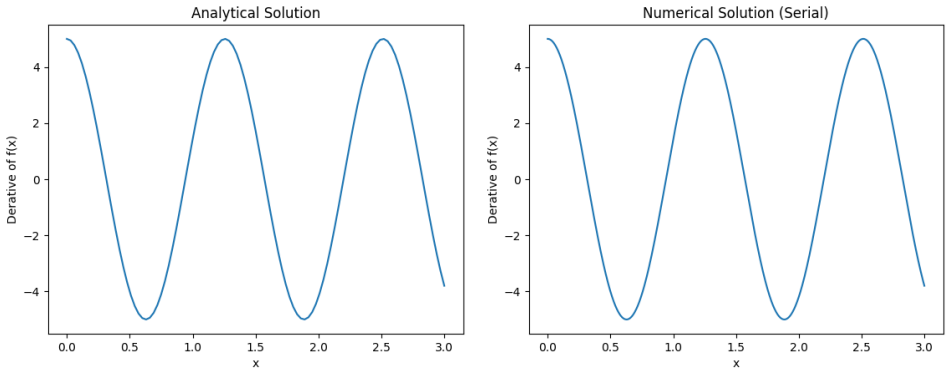
where j is any interior point ($j = 1, 2, \ldots, n-1$), and

$$f'_0 + 2f'_1 = \frac{1}{h}\left(-\frac{5}{2}f_0 + 2f_1 + \frac{1}{2}f_2\right)$$

$$f'_n + 2f'_{n-1} = \frac{1}{h}\left(\frac{5}{2}f_n - 2f_{n-1} - \frac{1}{2}f_{n-2}\right)$$

where h is the grid spacing and n is the number of grid points in the x direction.

- Here is the **graphical representation of solution** provided by the code which shows that the Parallel Cyclic Reduction Provides similar solution to Analytical solution. Both Serial and Analytical solution are similar to parallel one.



(a) Analytical Solution (b) Numerical solution for Serial Code

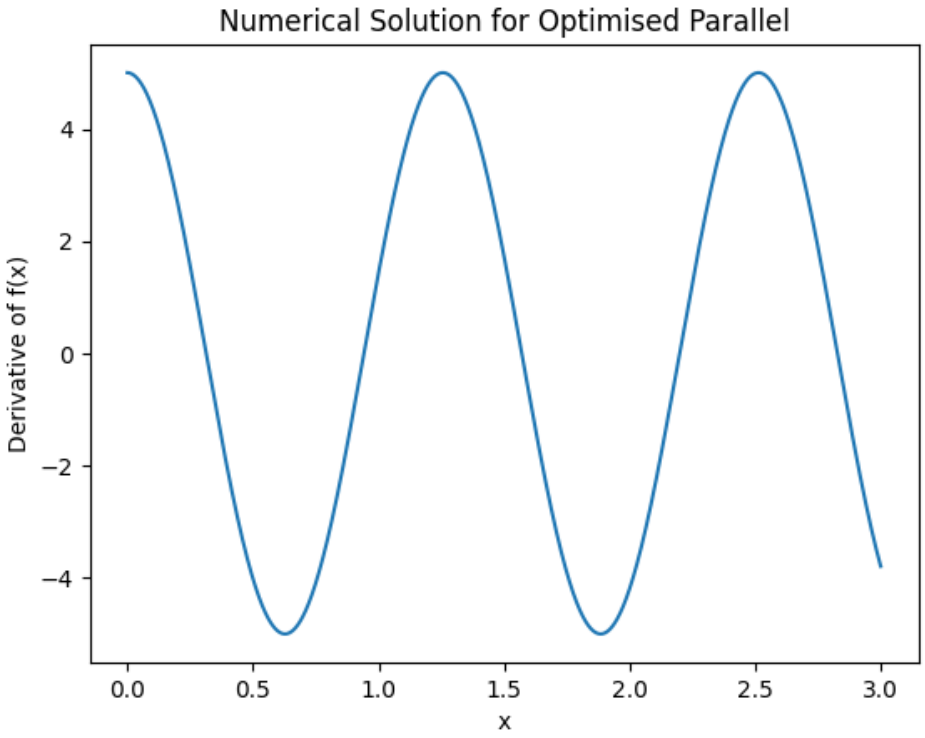


Figure 5: Numerical solution using optimised parallel code

The grid size for all the graphs is N=100.

- **Execution Time Analysis:**

Table 1: Execution Time Comparison

Input Size	Available PCR Algorithm on Nvidia	Optimized Parallel Code	Serial Code
26	1009 microseconds	942 microseconds	148 microseconds
101	851 microseconds	1188 microseconds	258 microseconds
1001	2304 microseconds	2442 microseconds	1777 microseconds
10001	8139 microseconds	4104 microseconds	8170 microseconds

Available PCR Algorithm on NVIDIA:

Input size 26: 1009 μs

Input size 101: 851 μs

Decreasing trend with larger input sizes: 2304 μs (1001), 8139 μs (10001)

Optimized Parallel Code:

Input size 26: 942 μs (lower than NVIDIA)

Outperforms NVIDIA across various input sizes: 1188 μs (101), 2442 μs (1001), 4104 μs (10001)

Serial Code:

Highest execution times:

Input size 26: 148 μs

Increasing trend with larger input sizes: 8170 μs (10001)

Overall, data indicates that the optimized parallel code performs the best in terms of execution time across all input sizes, followed by the available PCR algorithm on NVIDIA. The serial code,

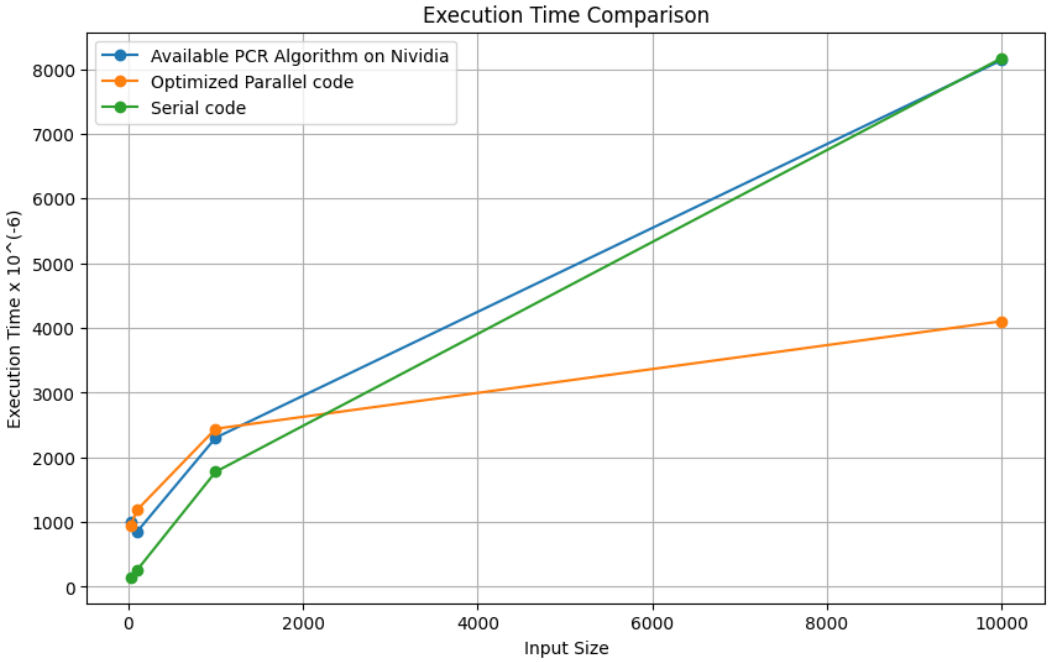


Figure 6: Numerical solution using optimised parallel code

being a non-parallel implementation, exhibits the highest execution times for all input sizes, making it the least efficient among the three implementations.

References

1. The cyclic reduction algorithm: from Poisson equation to stochastic processes and beyond
Dario A. Bini · Beatrice Meini

2. Parallel accelerated cyclic reduction preconditioner for three-dimensional elliptic PDEs with variable coefficients
Gustavo Chávez a,*, George Turkiyyah b, Stefano Zampini a, David Keyes, Journal of Computational and Applied Mathematics

3. Review of Cyclic Reduction for Parallel Solution of Hermitian Positive Definite Block-Tridiagonal Linear Systems
Martin Neuenhofen

4. A Parallel Method for Tridiagonal Equations
H. H. WANG, IBM Scientific Center