NATURAL SCIENCE REVIEW 3 100903 (2026)
nsr.jinr.int

Parallel partition algorithms for solving pentadiagonal and heptadiagonal systems of linear equations

Milena Veneva*[1]

[1] *RIKEN Center for Computational Science, R-CCS, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan*

Abstract

This paper presents parallel partition algorithms for solving systems of linear algebraic equations with pentadiagonal and heptadiagonal coefficient matrices. Theorems for the sparsity pattern of the partitioned matrices are formulated and proven. It is proven that both the pentadiagonal and the heptadiagonal partition algorithms in general do not preserve the diagonal dominance. The pentadiagonal algorithm is implemented with the help of CUDA. A survey of its performance on one and two GPUs is made. Configuration programs for the maximum size of an SLAE that can be solved with the pentadiagonal partition algorithm and cuSPARSE pentadiagonal solver are written. The computational times from the pentadiagonal partition method are compared with those from the serial pentadiagonal method, the cuSPARSE pentadiagonal solver, and cuDSS. The results are analyzed and discussed.

Keywords: parallel partition method, CUDA, GPU programming

DOI: 10.54546/NaturalSciRev.100903

1. Introduction

Systems of linear algebraic equations (SLAEs) with banded coefficient matrices frequently arise in scientific engineering, real-time interactive applications, and computational linear algebra. For instance, matrices generated by finite element or finite difference methods are typically banded. As solving these band SLAEs often dominates the total execution time, modern algorithms must prioritize computational speed. Consequently, developing high-performance parallel algorithms is essential to meet these efficiency demands.

In particular, fast solutions to pentadiagonal SLAEs are critical in numerical solution of ordinary and partial differential equations [1], interpolation problems [2], and boundary value problems [3].

Solving heptadiagonal SLAEs is a basic computational kernel of models of multiphase fluid flow in porous media [4], simulation of incompressible blood flow in arteries with a structured mesh domain [5], spline approximations [6], etc.

*Corresponding author e-mail address: milena.p.veneva@gmail.com

Received: 2 July 2026; Revised: 30 October 2026; Accepted: 12 November 2026; Published: 23 November 2026
Copyright: © by the author(s), 2026, under license to Joint Institute for Nuclear Research. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution-NonCommercial 4.0 International license (CC BY-NC License 4.0).

The development of parallel solvers for banded SLAEs has evolved from traditional CPU partition methods into highly accelerated GPU algorithms. Early partition approaches, such as the one introduced in [7], laid the groundwork for scattering the linear equations across independent processors but required tracing boundary coupling rows that yielded large interface systems. To optimize memory usage in these setups, the authors of [8] introduced a highly memory-efficient tridiagonal partition algorithm specifically tailored for diagonally dominant matrices, minimizing the local allocation overhead. Building on similar ideas to minimize inter-processor communication latency, the authors of [9] introduced a two-way parallel partition method tailored specifically for pentadiagonal matrices, halving communication startups by condensing the global interface down to a reduced block system. Finally, in [10] the focus shifts toward the modern graphics processing architectures and a CUDA implementation of a pentadiagonal parallel cyclic reduction solver.

The parallel partition algorithm introduced in [8] provides an efficient numerical method for solving tridiagonal systems of linear algebraic equations by decomposing the coefficient matrix for parallel computation. Designed for large-scale processor arrays, this method was originally implemented using Message Passing Interface technology. Here, we are going to formulate analogous parallel partition algorithms for solving SLAEs with pentadiagonal and heptadiagonal coefficient matrices.

2. Pentadiagonal partition algorithm

Following the logic of the tridiagonal partition method [8], we now describe the memory-efficient algorithm for the parallel solution of pentadiagonal systems of linear equations.

2.1. Notation

Let us define the notation and the algorithm. The pentadiagonal partition algorithm consists of three stages (see Figure 1 and Figure 2 for a detailed scheme of all the steps in the case of a system with 30 rows and size of the sub-system equal to 10). Starting with an SLAE $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$, where $\boldsymbol{A} = \text{pentadiag}(\boldsymbol{d}_{-2}, \boldsymbol{d}_{-1}, \boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2)$, $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is an $N \times N$ pentadiagonal matrix with non-zero elements structure as shown in Figure 1; $\boldsymbol{x}$ and $\boldsymbol{y}$ are real column vectors with N elements. Let $\boldsymbol{a}_i$ denote the ith row vector of $\boldsymbol{A}$. Each row $\boldsymbol{a}_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,N-1})$ satisfies $a_{i,j} = 0$ for $j \neq i-2, i-1, i, i+1, i+2$. We assume a set of $p < N$ processors, where each processor is assigned a unique index from 1 to p. Let j_k be the smallest row index owned by processor k, with boundary conditions $j_1 = 0$ and $j_{p+1} = N$, such that $j_k < j_{k+1}$ for $k = 1, 2, \ldots, p$. Consequently, processor k is allocated $N_k = j_{k+1} - j_k$ rows, specifically spanning rows j_k through $j_{k+1} - 1$ for $k = 1, 2, \ldots, p$.

2.2. The algorithm

Stage 1. First, each processor computes linear combinations of its rows to generate the four upper and lower interface equations. We store these rows in $\boldsymbol{f}_k$, $\boldsymbol{s}_k$, $\boldsymbol{p}_k$, $\boldsymbol{l}_k$ (**f**irst, **s**econd, **p**enultimate, and **l**ast, respectively) and the corresponding right-hand side (RHS) in y_{x_k}. The purpose of Stage 1 is to decouple the SLAE into independent smaller SLAEs by finding the dependencies on the boundaries between the partitions. To do this, on each step we eliminate the penultimate non-zero element (for the two upper equations) or the second non-zero element (for the two lower equations). We perform the following steps simultaneously on all processors:

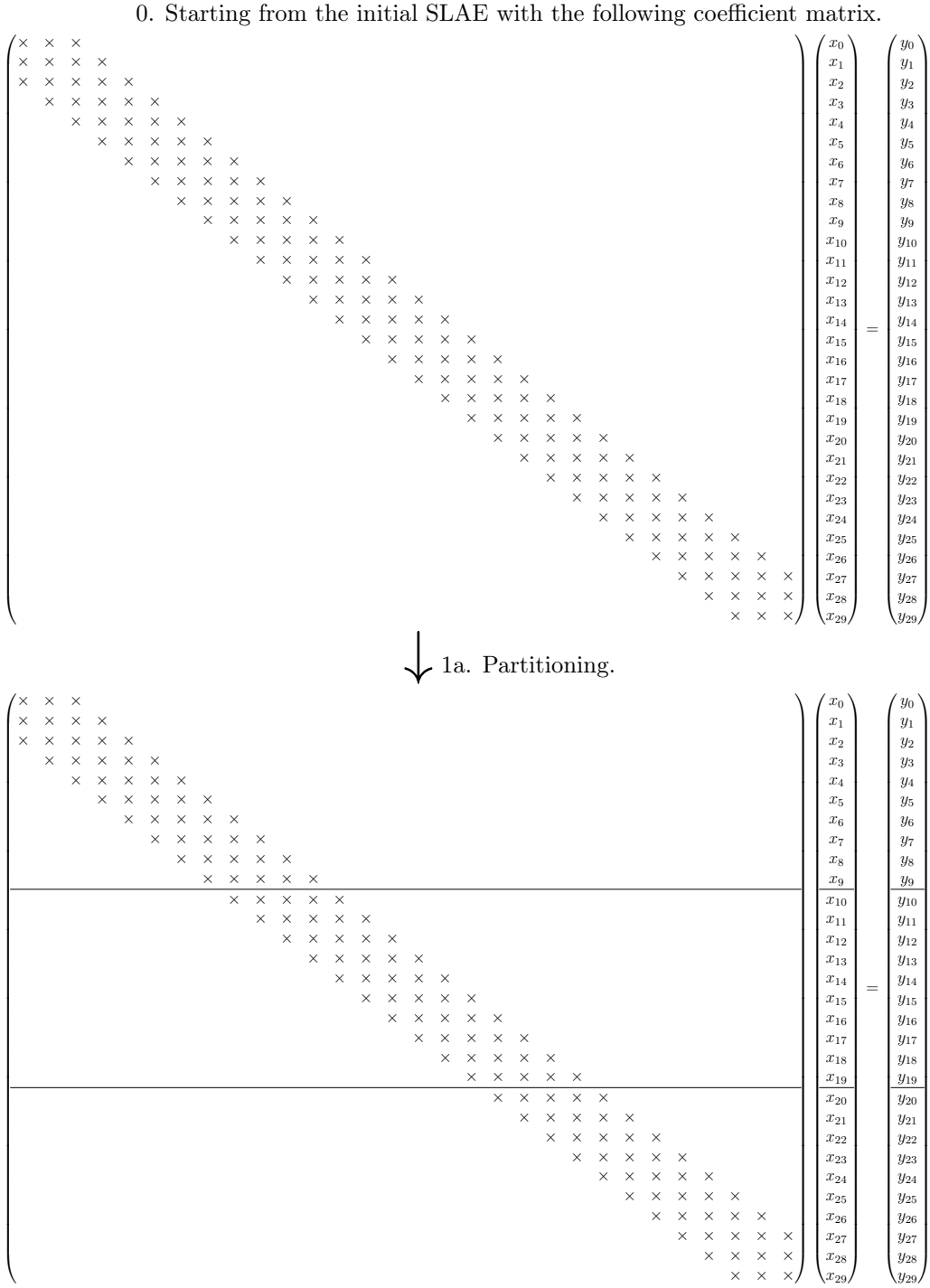


Figure 1. The stages of the pentadiagonal partition algorithm (fig. 1 out of 2).

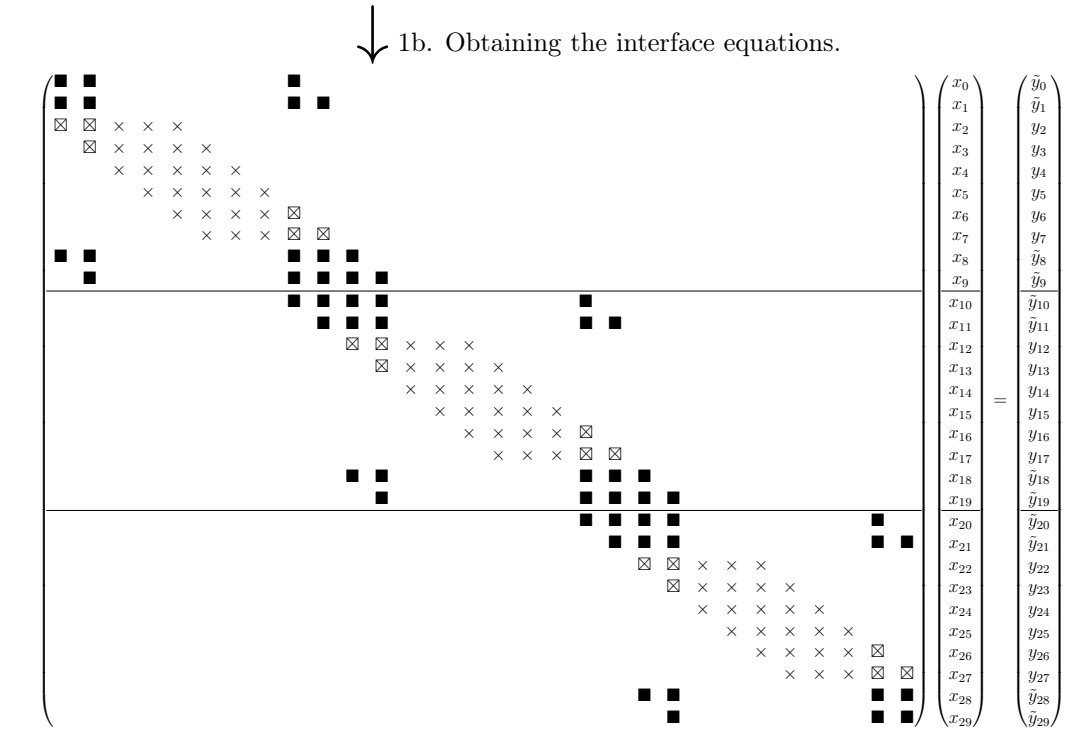


Figure 2. The stages of the pentadiagonal partition algorithm (fig. 2 out of 2).

For $\boldsymbol{l}_k$ (last lower):
1) initialization:

$$\begin{aligned}\boldsymbol{l}_k^{(j_k+3)} &= \boldsymbol{a}_{j_k+3},\\ y_{l_k}^{(j_k+3)} &= y_{j_k+3},\end{aligned}$$

2) for $i = j_k+4, \ldots, j_k+N_k-1$:

$$\begin{aligned}\beta_i &= \frac{a_{i,i-2}}{l_{k,i-2}^{(i-1)}},\\ \boldsymbol{l}_k^{(i)} &= \boldsymbol{a}_i - \beta_i \boldsymbol{l}_k^{(i-1)},\\ y_{l_k}^{(i)} &= y_i - \beta_i y_{l_k}^{(i-1)}.\end{aligned}$$

For $\boldsymbol{p}_k$ (penultimate lower):
1) initialization:

$$\begin{aligned}\boldsymbol{p}_k^{(j_k+2)} &= \boldsymbol{a}_{j_k+2},\\ y_{p_k}^{(j_k+2)} &= y_{j_k+2},\end{aligned}$$

2) for $i = j_k+3, \ldots, j_k+N_k-1$:

$$\begin{aligned}\beta_i &= \frac{a_{i,i-2}}{p_{k,i-2}^{(i-1)}},\\ \boldsymbol{p}_k^{(i)} &= \boldsymbol{a}_i - \beta_i \boldsymbol{p}_k^{(i-1)},\\ y_{p_k}^{(i)} &= y_i - \beta_i y_{p_k}^{(i-1)}.\end{aligned}$$

We store the interface equations:

$$\begin{aligned}\boldsymbol{l}_k &= \boldsymbol{l}_k^{(j_k+N_k-1)},\\ \boldsymbol{p}_k &= \boldsymbol{p}_k^{(j_k+N_k-1)},\\ y_{l_k} &= y_{l_k}^{(j_k+N_k-1)},\\ y_{p_k} &= y_{p_k}^{(j_k+N_k-1)}.\end{aligned}$$

Additional processing is needed for the penultimate row. So as to obtain the needed structure, we need to take a linear combination of the already computed last lower interface equation and the penultimate lower interface equation. If we are computing the interface equations for the last sub-system, $\beta := -1$.

$$\begin{aligned}\beta &= \frac{l_{k,j_k+N_k+1}}{p_{k,j_k+N_k+1}},\\ \boldsymbol{p}_k &= \boldsymbol{l}_k - \beta \boldsymbol{p}_k,\\ y_{p_k} &= y_{l_k} - \beta y_{p_k}.\end{aligned}$$

The result is the penultimate lower interface equation.

Simultaneously with the steps required for obtaining the two lower interface equations we perform the following steps concurrently on all processors:

For $\boldsymbol{f}_k$ (first upper):
1) initialization:

$$\begin{aligned}\boldsymbol{f}^{(j_{k+1}-4)} &= \boldsymbol{a}_{j_{k+1}-4},\\ y_{f_k}^{(j_{k+1}-4)} &= y_{j_{k+1}-4},\end{aligned}$$

2) for $i = j_{k+1}-5, j_{k+1}-6, \ldots, j_k$:

$$\begin{aligned}\beta_i &= \frac{a_{i,i+2}}{f_{k,i+2}^{(i+1)}},\\ \boldsymbol{f}_k^{(i)} &= \boldsymbol{a}_i - \beta_i \boldsymbol{f}_k^{(i+1)},\\ y_{f_k}^{(i)} &= y_i - \beta_i y_{f_k}^{(i+1)}.\end{aligned}$$

For $\boldsymbol{s}_k$ (second upper):
1) initialization:

$$\begin{aligned}\boldsymbol{s}_k^{(j_{k+1}-3)} &= \boldsymbol{a}_{j_{k+1}-3},\\ y_{s_k}^{(j_{k+1}-3)} &= y_{j_{k+1}-3},\end{aligned}$$

2) for $i = j_{k+1}-4, j_{k+1}-5, \ldots, j_k$:

$$\begin{aligned}\beta_i &= \frac{a_{i,i+2}}{s_{k,i+2}^{(i+1)}},\\ \boldsymbol{s}_k^{(i)} &= \boldsymbol{a}_i - \beta_i \boldsymbol{s}_k^{(i+1)},\\ y_{s_k}^{(i)} &= y_i - \beta_i y_{s_k}^{(i+1)}.\end{aligned}$$

We store the interface equations:

$$\begin{aligned}\boldsymbol{f}_k &= \boldsymbol{f}_k^{(j_k)},\\ \boldsymbol{s}_k &= \boldsymbol{s}_k^{(j_k)},\\ y_{f_k} &= y_{f_k}^{(j_k)},\\ y_{s_k} &= y_{s_k}^{(j_k)}.\end{aligned}$$

Additional processing is needed for the second row. So as to obtain the needed structure, we need to take a linear combination of the already computed first upper interface equation and the second upper interface equation. If we are computing the interface equations for the first sub-system, $\beta := -1$.

$$\begin{aligned}\beta &= \frac{f_{k,j_k-2}}{s_{k,j_k-2}},\\ \boldsymbol{s}_k &= \boldsymbol{f}_k - \beta \boldsymbol{s}_k,\\ y_{s_k} &= y_{f_k} - \beta y_{s_k}.\end{aligned}$$

The result is the second upper interface equation.

All the interface equations are assembled in order (that is, the first upper interface equation on the first partition becomes the first row, the second upper interface equation on the first partition becomes the second row, the penultimate lower interface equation on the first partition becomes the third row, the last lower interface equation from the same partition becomes the fourth row, and so on for the other partitions) into an interface SLAE.

Stage 2. Having found the interface equations, the initial matrix $\boldsymbol{A}$ is transformed into $\tilde{\boldsymbol{A}}$ which has the form shown in Figure 2. There, the parameters are $N = 30, p = 3, N_k = 10, k = \{1, 2, 3\}$; the matrix entries marked with $\times$ are equal to the corresponding elements of the original coefficient matrix, while the matrix items marked with ■ correspond to the entries from the interface equations obtained in Stage 1.

Thus, the interface system has the form:

$$\begin{bmatrix}\boldsymbol{f}_1\\ \boldsymbol{s}_1\\ \boldsymbol{p}_1\\ \boldsymbol{l}_1\\ \vdots\\ \boldsymbol{f}_p\\ \boldsymbol{s}_p\\ \boldsymbol{p}_p\\ \boldsymbol{l}_p\end{bmatrix}\begin{bmatrix}x_{j_1}\\ x_{j_1+1}\\ x_{j_2-2}\\ x_{j_2-1}\\ \vdots\\ x_{j_p}\\ x_{j_p+1}\\ x_{j_{p+1}-2}\\ x_{j_{p+1}-1}\end{bmatrix} = \begin{bmatrix}y_{f_1}\\ y_{s_1}\\ y_{p_1}\\ y_{l_1}\\ \vdots\\ y_{f_p}\\ y_{s_p}\\ y_{p_p}\\ y_{l_p}\end{bmatrix} \tag{1}$$

Before proceeding with the algorithm, we need to prove a theorem.

Theorem 1. Each interface equation has at most five non-zero entries, and the interface system is pentadiagonal.

Proof: the proof was done by induction. Having made the proof, we can list the positions of the non-zero items as follows:

- first upper interface equation: $i-2, i-1, i, i+1, j_{k+1}-2$ for $i = j_{k+1}-4, j_{k+1}-5, \ldots, j_k$;
- second upper interface equation: $i-2, i-1, i, i+1, j_{k+1}-1$ for $i = j_{k+1}-3, j_{k+1}-4, \ldots, j_k$;

- penultimate lower interface equation: $j_k, i-1, i, i+1, i+2$ for $i = j_k+2, \ldots, j_k+N_k-1$;
- last lower interface equation: $j_k+1, i-1, i, i+1, i+2$ for $i = j_k+3, \ldots, j_k+N_k-1$.

□

The result of this theorem is very important, because it ensures that on Stage 2 (and Stage 3) of the pentadiagonal partition method we have pentadiagonal systems which can be solved with the help of a pentadiagonal solver.

Stage 3. Having solved the interface SLAE from Eq. (1), we have obtained the values of some of the unknowns from each of the sub-systems (beside the interface equations). In Figure 2, the matrix coefficients which correspond to the already found unknowns are marked with ⊠. Therefore, these values are substituted (that is, respective transformations are made for the second two and the penultimate two rows of each sub-system), and, as a result, from each sub-system with size m remains a pentadiagonal SLAE with $m-4$ rows (denoted with $\times$ in Figure 2). These remaining pentadiagonal SLAEs are independent of each other and can be solved in parallel applying a pentadiagonal solver for each of them.

Remark: the pentadiagonal partition method could be used in combination with any (direct or iterative) serial or parallel solver for SLAEs with pentadiagonal coefficient matrices, for instance, a solver from the cuSPARSE Library [11], a solver from the Intel oneAPI [12] Math Kernel Library (oneMKL) [13], etc.

Theorem 2. The pentadiagonal partition method in general does not preserve the diagonal dominance.

Proof: the proof is by counterexample. Let us assume that the pentadiagonal partition method preserves the diagonal dominance, that is, if the initial coefficient matrix is pentadiagonal and diagonally dominant, so is the coefficient matrix of the interface SLAE. The first part regarding the pentadiagonality was proven in **Theorem 1**. We are going to construct an example in which the interface system is not diagonally dominant. Let us have an SLAE with $N = 10^5$, and let the sub-system size be $m = 8$. If we have an SLAE with matrix, whose diagonals are constant vectors $\boldsymbol{A} = \text{pentadiag}(\mathbf{1.0}, \mathbf{2.0}, \mathbf{21.0}, \mathbf{4.0}, \mathbf{3.0})$, this matrix is clearly diagonally dominant, then, following the algorithm shown above, we obtain that the first upper interface equation is

$$\boldsymbol{f}_1 = (2.260015 \times 10^1, 2.344765 \times 10^1, 0, 0, 0, 0, 1.843703 \times 10^{-1}).$$

However, this equation is not diagonally dominant. □

Remark: as the diagonal dominance is not preserved, we can apply the QR factorization method [14] which stands behind the cuSPARSE [11] pentadiagonal solver, because this factorization does not require the matrix to be diagonally dominant.

2.3. Configuration

2.3.1. Maximum size of an SLAE that can be solved with the pentadiagonal partition algorithm

A configuration program that evaluates the maximum size of an SLAE that can be solved with the pentadiagonal partition algorithm on a single GPU of the target architecture was written. The program takes the following input parameters (defined as pre-processor macros):

number_of_lines	size of the SLAE to be checked
m	size of the sub-system
USE_FLOAT USE_DOUBLE	data types to choose from
OVERWRITE_RHS	in-place solver; out-of-place solver otherwise (that is, the right-hand side will not be overwritten)

The configuration program checks if the preset size of the SLAE number_of_lines can be solved on the target architecture, and prints a message to confirm it or to deny it. It also prints the maximum size of an SLAE that can be handled by the GPU, taking into account the input parameters. The implementation relies on calling the `cudaMemGetInfo` API so as to find the amount of available GPU memory according to the OS. One should bear in mind that not all the GPU global memory can be used. Some memory is used by the CUDA context for the source code, heap buffer, device symbols, printf buffer, constant memory buffer, textures, local memory buffer, dynamic parallelism buffer, memory required by the CUDA driver, the CUDA runtime, etc. Therefore, predicting the memory requirements of a kernel is a difficult task [15, 16].

2.3.2. Memory consumption of the cuSPARSE pentadiagonal solver

The cuSPARSE pentadiagonal solver requires the programmer to allocate an additional buffer. The size of this buffer was found for different sizes of an SLAE experimentally as shown in Table 1.

Table 1. Size (in bytes) of the cuSPARSE pentadiagonal buffer for different SLAE sizes, single precision.

SLAE size N	5	6	7	128	2×10^3	8192	10^4	10^8	134 354 543
bytes	40	48	56	1024	16×10^3	65 536	8×10^4	8×10^8	1 074 836 344

Thus, we can derive a formula for the buffer size:

$$2 \times \text{sizeof(data)} \times \text{number_of_rows}.$$

Logically, the size of the buffer for double precision floating point data is twice the size of the buffer for single precision data.

2.3.3. Maximum size of an SLAE that can be solved with the cuSPARSE pentadiagonal solver

We used the obtained formula to write a configuration program so as to find the maximum SLAE size that can be solved on the target architecture with the help of the cuSPARSE pentadiagonal solver. The program takes the following input parameters:

number_of_lines	size of the SLAE to be checked
USE_FLOAT USE_DOUBLE	data types to choose from

The configuration program checks if the preset size of the SLAE number_of_lines can be solved on the target architecture with the cuSPARSE solver, and prints the buffer size the programmer needs to allocate, or a message to say that the system is too big for the target

GPU. The program also prints the maximum size of an SLAE that can be handled by the GPU, taking into account the input parameters. This program queries the amount of the available global memory by calling the `cudaMemGetInfo` API. Just like above, one should bear in mind that not all the GPU global memory can be used. Some memory is needed for the program, the heap, etc.

3. Heptadiagonal partition algorithm

Now following the already established procedure, let us describe the partition method in the case of heptadiagonal systems of linear algebraic equations.

3.1. Notation

Let us define the notation and the algorithm. The heptadiagonal partition algorithm consists of three stages (see Figure 3 and Figure 4 for a detailed scheme of all the steps in the case of a system with 39 rows and size of the sub-system equal to 13). Starting with an SLAE $\boldsymbol{Ax} = \boldsymbol{y}$, where $\boldsymbol{A} = \text{heptadiag}(\boldsymbol{d}_{-3}, \boldsymbol{d}_{-2}, \boldsymbol{d}_{-1}, \boldsymbol{d}_0, \boldsymbol{d}_1, \boldsymbol{d}_2, \boldsymbol{d}_3)$, $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ is an $N \times N$ heptadiagonal matrix with non-zero elements structure as shown in Figure 3; $\boldsymbol{x}$ and $\boldsymbol{y}$ are real column vectors with N elements. Let $\boldsymbol{a}_i$ denote the ith row vector of $\boldsymbol{A}$. Each row $\boldsymbol{a}_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,N-1})$ satisfies $a_{i,j} = 0$ for $j \neq i-3, i-2, i-1, i, i+1, i+2, i+3$. We assume a set of $p < N$ processors, where each processor is assigned a unique index from 1 to p. Let j_k be the smallest row index owned by processor k, with boundary conditions $j_1 = 0$ and $j_{p+1} = N$, such that $j_k < j_{k+1}$ for $k = 1, 2, \ldots, p$. Consequently, processor k is allocated $N_k = j_{k+1} - j_k$ rows, specifically spanning rows j_k through $j_{k+1} - 1$ for $k = 1, 2, \ldots, p$.

3.2. The algorithm

Stage 1. First, each processor computes a linear combination of its rows to generate the six upper and lower interface equations. We store these rows in $\boldsymbol{f}_k$, $\boldsymbol{s}_k$, $\boldsymbol{t}_k$, $\boldsymbol{b}_k$, $\boldsymbol{p}_k$, $\boldsymbol{l}_k$ (**f**irst, **s**econd, **t**hird, **b**efore the penultimate — antepenultimate, **p**enultimate, and **l**ast, respectively) and the corresponding right-hand side in y_{x_k}. The purpose of Stage 1 is to decouple the SLAE into independent smaller SLAEs by finding the dependencies on the boundaries between the partitions. To do this, on each step we eliminate the penultimate non-zero element (for the three upper equations) or the second non-zero element (for the three lower equations). We perform the following steps simultaneously on all processors:

For $\boldsymbol{l}_k$ (last lower):

1) initialization:

$$\boldsymbol{l}_k^{(j_k+5)} = \boldsymbol{a}_{j_k+5},$$
$$y_{l_k}^{(j_k+5)} = y_{j_k+5},$$

2) for $i = j_k + 6, \ldots, j_k + N_k - 1$:

$$\beta_i = \frac{a_{i,i-3}}{l_{k,i-3}^{(i-1)}},$$
$$\boldsymbol{l}_k^{(i)} = \boldsymbol{a}_i - \beta_i \boldsymbol{l}_k^{(i-1)},$$
$$y_{l_k}^{(i)} = y_i - \beta_i y_{l_k}^{(i-1)}.$$

For $\boldsymbol{p}_k$ (penultimate lower):

1) initialization:

$$\boldsymbol{p}_k^{(j_k+4)} = \boldsymbol{a}_{j_k+4},$$
$$y_{p_k}^{(j_k+4)} = y_{j_k+4},$$

2) for $i = j_k + 5, \ldots, j_k + N_k - 1$:

$$\beta_i = \frac{a_{i,i-3}}{p_{k,i-3}^{(i-1)}},$$
$$\boldsymbol{p}_k^{(i)} = \boldsymbol{a}_i - \beta_i \boldsymbol{p}_k^{(i-1)},$$
$$y_{p_k}^{(i)} = y_i - \beta_i y_{p_k}^{(i-1)}.$$

For $\boldsymbol{b}_k$ (antepenultimate lower):

1) initialization:

$$\boldsymbol{b}_k^{(j_k+3)} = \boldsymbol{a}_{j_k+3},$$
$$y_{b_k}^{(j_k+3)} = y_{j_k+3},$$

2) for $i = j_k + 4, \ldots, j_k + N_k - 1$:

$$\beta_i = \frac{a_{i,i-3}}{b_{k,i-3}^{(i-1)}},$$
$$\boldsymbol{b}_k^{(i)} = \boldsymbol{a}_i - \beta_i \boldsymbol{b}_k^{(i-1)},$$
$$y_{b_k}^{(i)} = y_i - \beta_i y_{b_k}^{(i-1)}.$$

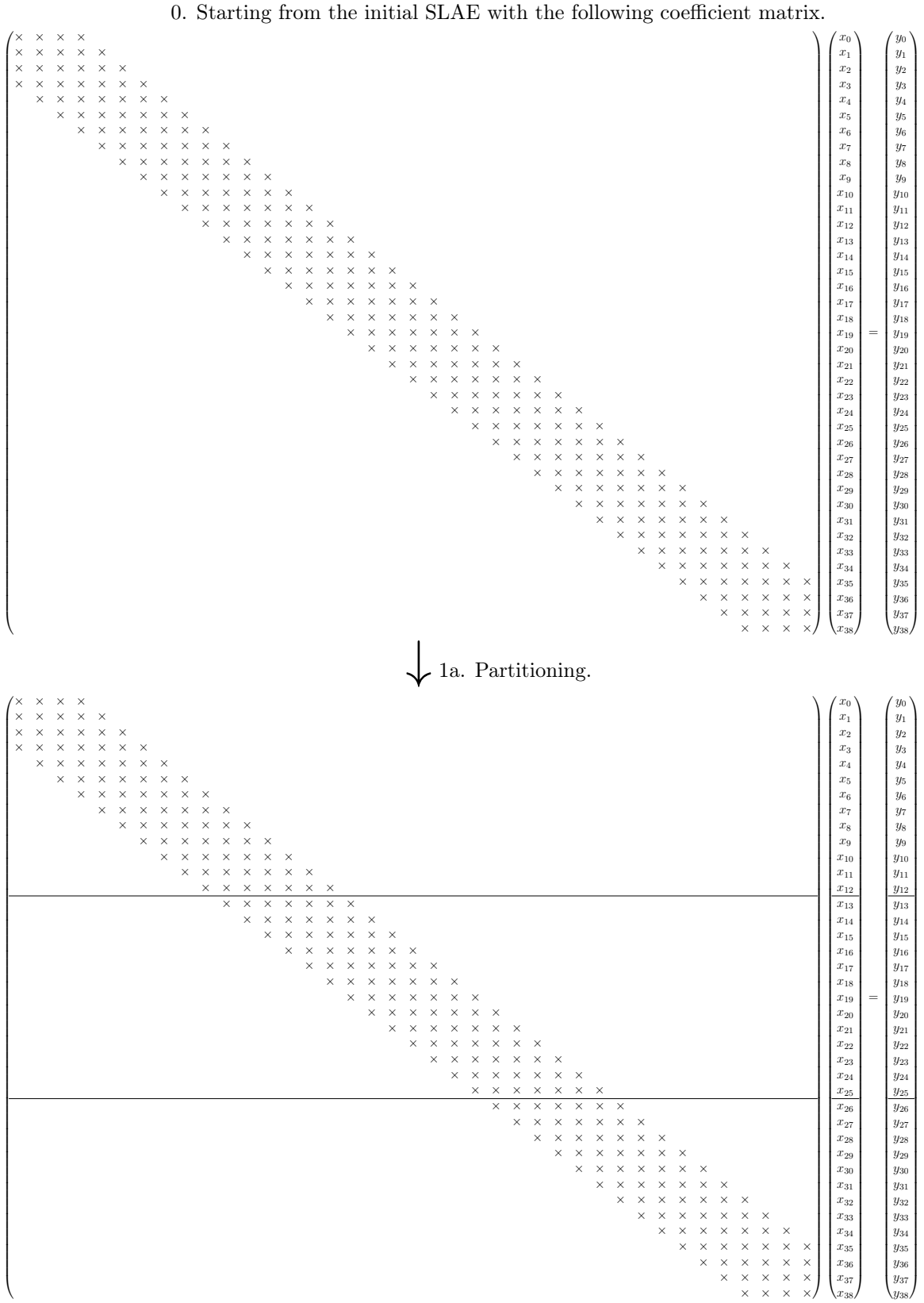
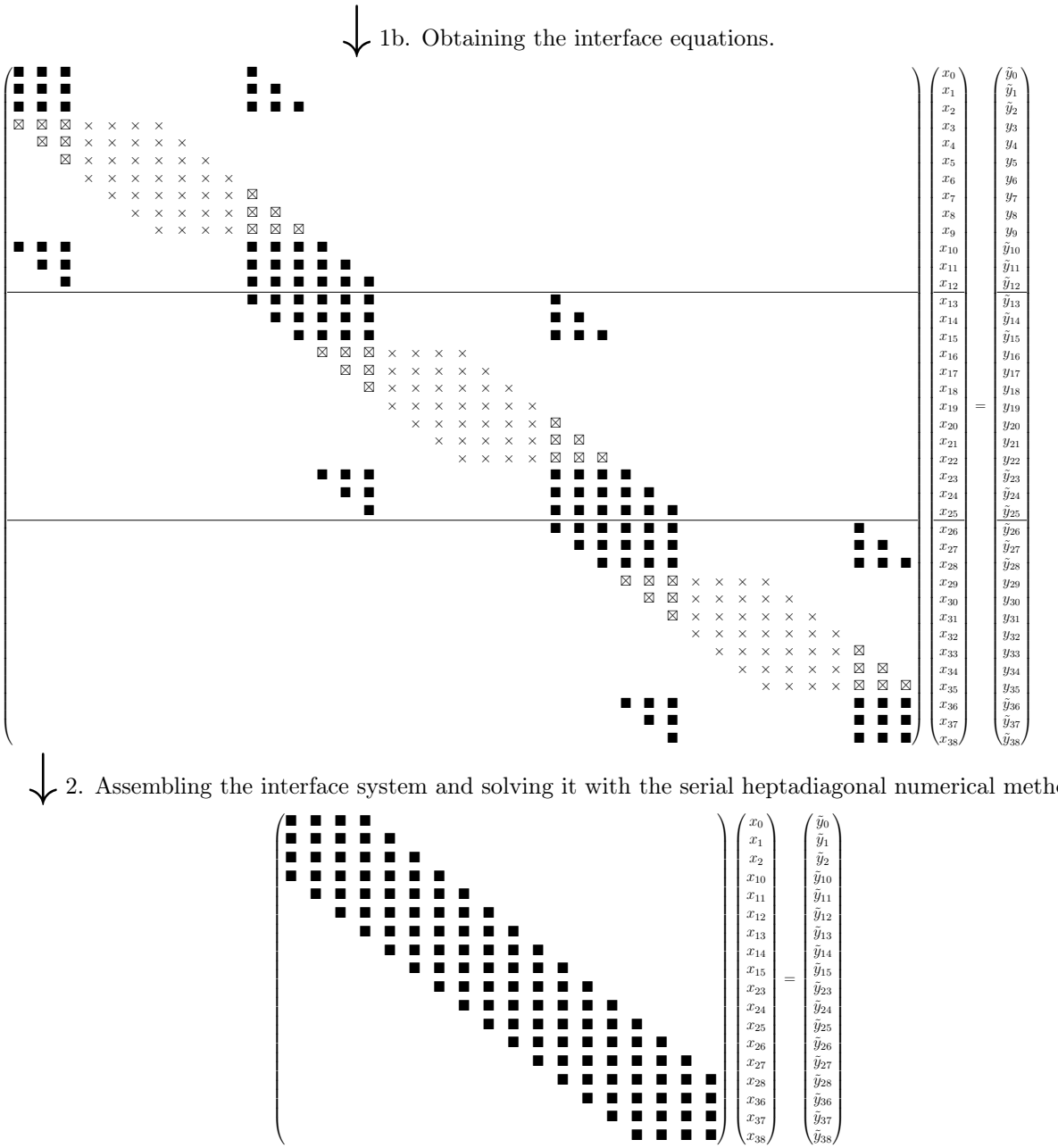


Figure 3. The stages of the heptadiagonal partition algorithm (fig. 1 out of 2).



3. Substituting the already found unknowns corresponding to the ☒ coefficients, and solving the rest of the heptadiagonal sub-systems.

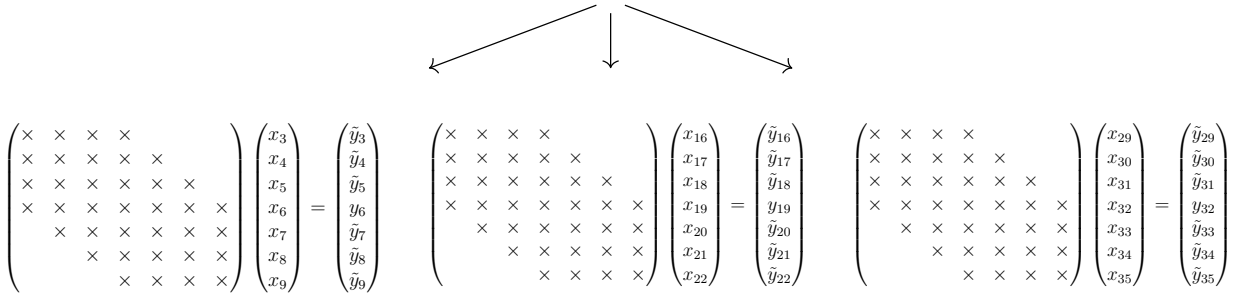


Figure 4. The stages of the heptadiagonal partition algorithm (fig. 2 out of 2).

We store the interface equations:

$$\begin{aligned}
\boldsymbol{b}_k &= \boldsymbol{b}_k^{(j_k+N_k-1)},\\
\boldsymbol{p}_k &= \boldsymbol{p}_k^{(j_k+N_k-1)},\\
\boldsymbol{l}_k &= \boldsymbol{l}_k^{(j_k+N_k-1)},\\
y_{b_k} &= y_{b_k}^{(j_k+N_k-1)},\\
y_{p_k} &= y_{p_k}^{(j_k+N_k-1)},\\
y_{l_k} &= y_{l_k}^{(j_k+N_k-1)}.
\end{aligned}$$

Additional processing is needed for the penultimate row. So as to obtain the needed structure, we need to take a linear combination of the already computed last lower interface equation and the penultimate lower interface equation. In the case we are computing the interface equations for the last sub-system, $\beta := -1$.

$$\begin{aligned}
\beta &= \frac{l_{k,j_k+N_k+2}}{p_{k,j_k+N_k+2}},\\
\boldsymbol{p}_k &= \boldsymbol{l}_k - \beta \boldsymbol{p}_k,\\
y_{p_k} &= y_{l_k} - \beta y_{p_k}.
\end{aligned}$$

The result is the penultimate lower interface equation.

Additional processing is also needed for the antepenultimate row. Subsequently, we take linear combinations of the antepenultimate and the last equations, and the antepenultimate and the penultimate equations, respectively. In the case we are computing the interface equations for the last sub-system, $\beta := -1$.

$$\begin{aligned}
\beta &= \frac{l_{k,j_k+N_k+2}}{b_{k,j_k+N_k+2}},\\
\boldsymbol{b}_k &= \boldsymbol{l}_k - \beta \boldsymbol{b}_k,\\
y_{b_k} &= y_{l_k} - \beta y_{b_k}.
\end{aligned}$$

$$\begin{aligned}
\beta &= \frac{p_{k,j_k+N_k+1}}{b_{k,j_k+N_k+1}},\\
\boldsymbol{b}_k &= \boldsymbol{p}_k - \beta \boldsymbol{b}_k,\\
y_{b_k} &= y_{p_k} - \beta y_{b_k}.
\end{aligned}$$

The result is the antepenultimate lower interface equation.

Simultaneously with the steps required for obtaining the three lower interface equations we perform the following steps concurrently on all processors:

For $\boldsymbol{f}_k$ (first upper):
1) initialization:

$$\begin{aligned}
\boldsymbol{f}_k^{(j_{k+1}-6)} &= \boldsymbol{a}_{j_{k+1}-6},\\
y_{f_k}^{(j_{k+1}-6)} &= y_{j_{k+1}-6},
\end{aligned}$$

For $\boldsymbol{s}_k$ (second upper):
1) initialization:

$$\begin{aligned}
\boldsymbol{s}_k^{(j_{k+1}-5)} &= \boldsymbol{a}_{j_{k+1}-5},\\
y_{s_k}^{(j_{k+1}-5)} &= y_{j_{k+1}-5},
\end{aligned}$$

For $\boldsymbol{t}_k$ (third upper):
1) initialization:

$$\begin{aligned}
\boldsymbol{t}_k^{(j_{k+1}-4)} &= \boldsymbol{a}_{j_{k+1}-4},\\
y_{t_k}^{(j_{k+1}-4)} &= y_{j_{k+1}-4},
\end{aligned}$$

2) for $i = j_{k+1} - 7, j_{k+1} - 8, \ldots, j_k$:

$$\beta_i = \frac{a_{i,i+3}}{f_{k,i+3}^{(i+1)}},$$
$$\boldsymbol{f}_k^{(i)} = \boldsymbol{a}_i - \beta_i \boldsymbol{f}_k^{(i+1)},$$
$$y_{f_k}^{(i)} = y_i - \beta_i y_{f_k}^{(i+1)}.$$

2) for $i = j_{k+1} - 6, j_{k+1} - 7, \ldots, j_k$:

$$\beta_i = \frac{a_{i,i+3}}{s_{k,i+3}^{(i+1)}},$$
$$\boldsymbol{s}_k^{(i)} = \boldsymbol{a}_i - \beta_i \boldsymbol{s}_k^{(i+1)},$$
$$y_{s_k}^{(i)} = y_i - \beta_i y_{s_k}^{(i+1)}.$$

2) for $i = j_{k+1} - 5, j_{k+1} - 6, \ldots, j_k$:

$$\beta_i = \frac{a_{i,i+3}}{t_{k,i+3}^{(i+1)}},$$
$$\boldsymbol{t}_k^{(i)} = \boldsymbol{a}_i - \beta_i \boldsymbol{t}_k^{(i+1)},$$
$$y_{t_k}^{(i)} = y_i - \beta_i y_{t_k}^{(i+1)}.$$

We store the interface equations:

$$\boldsymbol{f}_k = \boldsymbol{f}_k^{(j_k)},$$
$$\boldsymbol{s}_k = \boldsymbol{s}_k^{(j_k)},$$
$$\boldsymbol{t}_k = \boldsymbol{t}_k^{(j_k)},$$
$$y_{f_k} = y_{f_k}^{(j_k)},$$
$$y_{s_k} = y_{s_k}^{(j_k)},$$
$$y_{t_k} = y_{t_k}^{(j_k)}.$$

Additional processing is needed for the second row. So as to obtain the needed structure, we need to take a linear combination of the already computed first upper interface equation and the second upper interface equation. In the case we are computing the interface equations for the first sub-system, $\beta := -1$.

$$\beta = \frac{f_{k,j_k-3}}{s_{k,j_k-3}},$$
$$\boldsymbol{s}_k = \boldsymbol{f}_k - \beta \boldsymbol{s}_k,$$
$$y_{s_k} = y_{f_k} - \beta y_{s_k}.$$

The result is the second upper interface equation.

Additional processing is also needed for the third row. Subsequently, we take linear combinations of the third and the first equations, and the third and the second equations, respectively. In the case we are computing the interface equations for the first sub-system, $\beta := -1$.

$$\beta = \frac{f_{k,j_k-3}}{t_{k,j_k-3}},$$
$$\boldsymbol{t}_k = \boldsymbol{f}_k - \beta \boldsymbol{t}_k,$$
$$y_{t_k} = y_{f_k} - \beta y_{t_k}.$$

$$\beta = \frac{s_{k,j_k-2}}{t_{k,j_k-2}},$$
$$\boldsymbol{t}_k = \boldsymbol{s}_k - \beta \boldsymbol{t}_k,$$
$$y_{t_k} = y_{s_k} - \beta y_{t_k}.$$

The result is the third upper interface equation.

All the interface equations are assembled in order (that is, the first upper interface equation on the first partition becomes the first row, the second upper interface equation on the first partition becomes the second row, the third upper interface equation on the first partition becomes the third row, the antepenultimate lower interface equation on the first partition becomes the fourth row, the penultimate lower interface equation on the first partition becomes the fifth row, the last lower interface equation from the same partition becomes the sixth row, and so on for the other partitions) into an interface SLAE.

Stage 2. Having found the interface equations, the initial matrix $\boldsymbol{A}$ is transformed into $\tilde{\boldsymbol{A}}$ which has the form shown in Figure 4. There, the parameters are $N = 39, p = 3, N_k = 13, k = \{1, 2, 3\}$; the matrix entries marked with $\times$ are equal to the corresponding elements of

the original coefficient matrix, while the matrix items marked with ■ correspond to the entries from the interface equations obtained in Stage 1.

Thus, the interface system has the form:

$$\begin{bmatrix} \boldsymbol{f}_1 \\ \boldsymbol{s}_1 \\ \boldsymbol{t}_1 \\ \boldsymbol{b}_1 \\ \boldsymbol{p}_1 \\ \boldsymbol{l}_1 \\ \vdots \\ \boldsymbol{f}_p \\ \boldsymbol{s}_p \\ \boldsymbol{t}_p \\ \boldsymbol{b}_p \\ \boldsymbol{p}_p \\ \boldsymbol{l}_p \end{bmatrix} \begin{bmatrix} x_{j_1} \\ x_{j_1+1} \\ x_{j_1+2} \\ x_{j_2-3} \\ x_{j_2-2} \\ x_{j_2-1} \\ \vdots \\ x_{j_p} \\ x_{j_p+1} \\ x_{j_p+2} \\ x_{j_{p+1}-3} \\ x_{j_{p+1}-2} \\ x_{j_{p+1}-1} \end{bmatrix} = \begin{bmatrix} y_{f_1} \\ y_{s_1} \\ y_{t_1} \\ y_{b_1} \\ y_{p_1} \\ y_{l_1} \\ \vdots \\ y_{f_p} \\ y_{s_p} \\ y_{t_p} \\ y_{b_p} \\ y_{p_p} \\ y_{l_p} \end{bmatrix} \tag{2}$$

Following the example set by the pentadiagonal partition algorithm, we can formulate two theorems.

Theorem 3. Each interface equation has at most seven non-zero entries, and the interface system is heptadiagonal.

Proof: the proof was done by induction. Having made the proof, we can list the positions of the non-zero items as follows:

- first upper interface equation: $j_k-3, j_k-2, j_k-1, j_k, j_k+1, j_k+2, j_{k+1}-3$;
- second upper interface equation: $j_k-2, j_k-1, j_k, j_k+1, j_k+2, j_{k+1}-3, j_{k+1}-2$;
- third upper interface equation: $j_k-1, j_k, j_k+1, j_k+2, j_{k+1}-3, j_{k+1}-2, j_{k+1}-1$;
- antepenultimate lower interface equation: $j_k, j_k+1, j_k+2, j_{k+1}-3, j_{k+1}-2, j_{k+1}-1, j_{k+1}$;
- penultimate lower interface equation: $j_k+1, j_k+2, j_{k+1}-3, j_{k+1}-2, j_{k+1}-1, j_{k+1}, j_{k+1}+1$;
- last lower interface equation: $j_k+2, j_{k+1}-3, j_{k+1}-2, j_{k+1}-1, j_{k+1}, j_{k+1}+1, j_{k+1}+2$.

□

The result of this theorem is very important, because it ensures that on Stage 2 (and Stage 3) of the heptadiagonal partition method we have heptadiagonal systems which can be solved with the help of a heptadiagonal solver.

Stage 3. Having solved the interface SLAE from Eq. (2), we have obtained the values of some of the unknowns from each of the sub-systems (beside the interface equations). Therefore, these values are substituted (that is, respective transformations are made for the second three and the penultimate three rows of each sub-system), and, as a result, from each sub-system with size m remains a heptadiagonal SLAE with $m-6$ rows. These remaining heptadiagonal SLAEs are independent of each other and can be solved in parallel applying a heptadiagonal solver for each of them.

Remark: the heptadiagonal partition method could be used in combination with any (direct or iterative) serial or parallel solver for SLAEs with heptadiagonal coefficient matrices, for instance, a solver from the cuSOLVER Library [17], a solver from the Intel oneAPI [12] Math Kernel Library (oneMKL) [13], etc.

Similarly to what was shown for the pentadiagonal partition method, we can formulate the following theorem.

Theorem 4. The heptadiagonal partition method in general does not preserve the diagonal dominance.

The proof of this theorem is analogous to that of **Theorem 2** for the pentadiagonal case.

4. A survey of performance for the pentadiagonal partition algorithm

The pentadiagonal partition algorithm was implemented with the help of CUDA [18]. The code is available upon request. In particular, Stage 2 can use the serial CPU pentadiagonal solver [19], the cuSPARSE pentadiagonal solver, or a QR-based CPU solver. Computational experiments were made on the basis of NVIDIA GPU RTX A5000 [20, 21], for different SLAE sizes, both FP32 and FP64 precision; out-of-place solution and no recursions for the partition algorithm. Table 2 summarizes the times from these tests. There, the 2nd and the 4th columns show the computational times from the pentadiagonal partition run on a single GPU, and the 5th and the 7th columns — from the pentadiagonal partition run on 2 GPUs. The (m, #str) notation stands for the size of the sub-system and the number of CUDA streams that were used. For the sake of fairness of the comparison, the results shown below are based on Stage 2 using the serial CPU pentadiagonal solver. Table 3 sums up the times from different pentadiagonal solvers. There, the 2nd and the 3rd columns show the computational times from the CPU serial pentadiagonal solver [19], the 4th and the 5th columns — from the cuSPARSE pentadiagonal solver (which is based on QR factorization [14]), the 6th and the 7th columns — from the cuDSS solver [22]. All times of GPU solvers include the times for the memory transfer of the SLAE to the device and the obtained solution from the device to the host. Thus, we compare the times of applications, not just solvers. Figure 5 and Figure 6 compare the results between the five different pentadiagonal solvers for FP32 and FP64 precision, respectively.

Table 2. Computational times (in [ms]) for the pentadiagonal partition on 1 GPU and the pentadiagonal partition on 2 GPUs, NVIDIA GPU RTX A5000, for different SLAE sizes, FP32 and FP64 precision.

SLAE size	Partition (0 rec), 1 GPU			Partition (0 rec), 2 GPUs		
	FP32	(m, #str)	FP64	FP32	(m, #str)	FP64
10^3	0.305811	(10, 10)	0.321347	0.562117	(10, 10)	0.571242
10^4	0.379163	(10, 10)	0.410391	0.653523	(10, 10)	0.668628
10^5	0.616439	(50, 10)	3.028722	0.893094	(50, 10)	4.286951
10^6	2.795257	(50, 10)	8.630578	3.122965	(50, 10)	8.507976
10^7	22.13688	(50, 10)	60.46829	21.12160	(50, 10)	48.37534
10^8	220.4968	(50, 10)	581.8253	205.3438	(50, 10)	449.4791
2.5×10^8	555.5591	(50, 10)	1447.758	507.3357	(50, 10)	1107.439
5.0×10^8	1112.557	(50, 10)	out of mem	1019.489	(50, 10)	2236.859
7.5×10^8	1801.362	(50, 10)	out of mem	1694.875	(50, 10)	3330.489
10^9	out of mem	(50, 10)	out of mem	out of mem	(50, 10)	out of mem

Table 3. Computational times (in [ms]) for the pentadiagonal solvers — serial pentadiagonal, cuSPARSE, cuDSS, on 1 GPU, NVIDIA GPU RTX A5000, for different SLAE sizes, FP32 and FP64 precision.

SLAE size	Serial pentadiagonal		cuSPARSE		cuDSS	
	FP32	FP64	FP32	FP64	FP32	FP64
10^3	0.007404	0.010560	0.477184	6.516896	29.100033	25.288704
10^4	0.079310	0.095742	4.352320	64.67373	40.190975	36.240417
10^5	0.797817	0.923294	42.95008	591.4767	149.883804	148.831238
10^6	7.701258	8.848475	430.2635	5844.986	1581.151245	1585.057739
10^7	75.05075	86.08222	4155.377	58 411.43	22 072.404297	21 890.316406
10^8	748.4604	854.7202	41 592.66	587 952.9	out of mem	out of mem
2.5×10^8	1879.850	2150.669	104 217.1	1 470 736.0	out of mem	out of mem
5.0×10^8	3782.922	4298.387	208 639.5	out of mem	out of mem	out of mem
7.5×10^8	5622.153	6418.177	312 644.3	out of mem	out of mem	out of mem
10^9	out of mem	out of mem	out of mem	out of mem	out of mem	out of mem

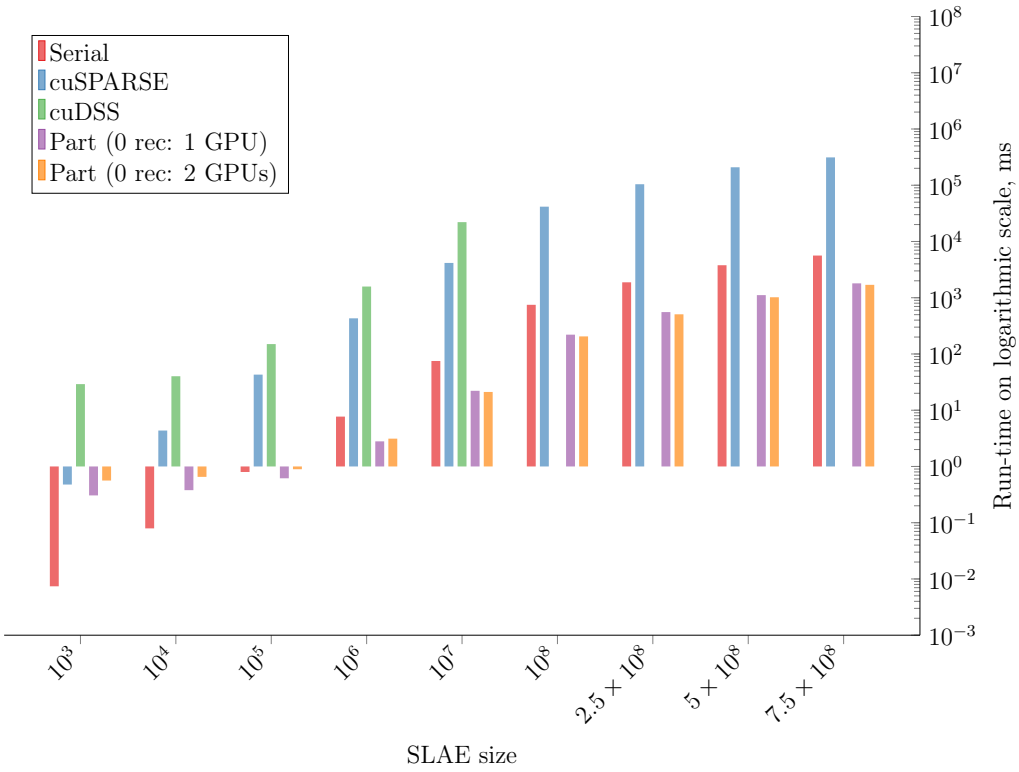


Figure 5. Comparison of the results from the pentadiagonal solvers — CPU serial pentadiagonal, cuSPARSE, cuDSS, pentadiagonal partition on 1 GPU, and pentadiagonal partition on 2 GPUs, NVIDIA GPU RTX A5000, for different SLAE sizes, FP32 precision.

4.1. Serial pentadiagonal solver

The pentadiagonal numerical solver is a generalization of the Thomas method for pentadiagonal SLAEs [19]. The algorithm is based on LU decomposition and requires forward reduction for reducing the initial matrix into a lower triangular one, and a backward substitution for finding the unknowns in a reverse order. The algorithm was implemented with the help of

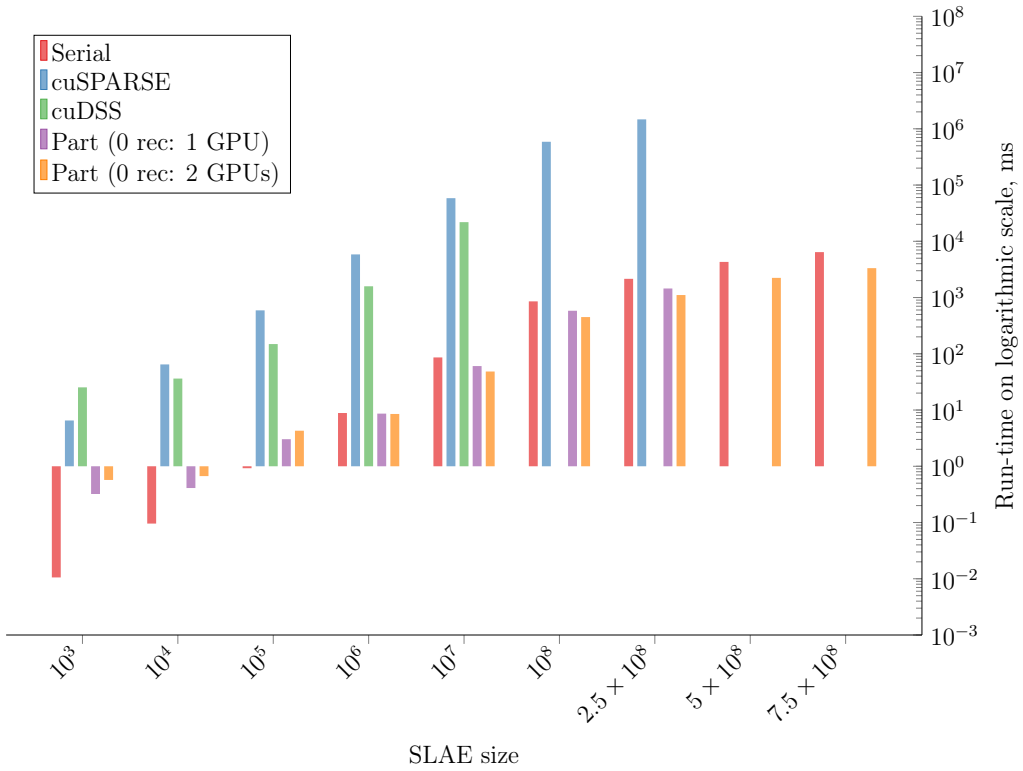


Figure 6. Comparison of the results from the pentadiagonal solvers — CPU serial pentadiagonal, cuSPARSE, cuDSS, pentadiagonal partition on 1 GPU, and pentadiagonal partition on 2 GPUs, NVIDIA GPU RTX A5000, for different SLAE sizes, FP64 precision.

C++. The full pentadiagonal numerical method in pseudocode is given in Algorithm 1. Both the out-of-place and in-place solution options are presented. The described algorithm reuses the $\boldsymbol{d}$, $\boldsymbol{u}$, and $\boldsymbol{w}$ arrays so as to keep the intermediate results.

5. Error analysis

To evaluate the robustness, accuracy, and scaling behavior of the proposed pentadiagonal partition solver, we conduct a numerical error analysis. The performance is benchmarked against the serial pentadiagonal solver. All experiments are performed in double-precision floating-point arithmetic. The test suite consists of three distinct classes of diagonally dominant pentadiagonal matrices:

- class A: this class represents a modified high-order 1D Laplacian finite difference matrix, where the main diagonal is shifted to guarantee the diagonal dominance. The elements are defined as
$$a_{ii} = 10, \quad a_{i,i\pm1} = -4, \quad a_{i,i\pm2} = 1. \tag{3}$$
- class B: to evaluate backward stability under severe precision loss without violating the diagonal dominance, this class utilizes an exponentially decaying inverse-distance structure for off-diagonals, with a main diagonal adaptively shifted to sit precisely near the edge of singularity:
$$a_{i,i\pm1} = -\frac{1}{i+1}, \quad a_{i,i\pm2} = -\frac{1}{i+2}, \tag{4}$$

Algorithm 1: Pentadiagonal algorithm for solving an SLAE $\boldsymbol{Ax} = \boldsymbol{y}$, $\boldsymbol{A} = \text{pentadiag}(\boldsymbol{s}, \boldsymbol{l}, \boldsymbol{d}, \boldsymbol{u}, \boldsymbol{w})$.

Input: N, $\boldsymbol{s}, \boldsymbol{l}, \boldsymbol{d}, \boldsymbol{u}, \boldsymbol{w}, \boldsymbol{y}$
Output: $\boldsymbol{x}$ (out-of-place); $\boldsymbol{y}$ (in-place)

// forward reduction:
$\mu = 1.0/d_0$;
$d_0 = u_0 \times \mu$;
$u_0 = w_0 \times \mu$;
$w_0 = y_0 \times \mu$;
$\gamma = l_1$;
$\mu = 1.0/(d_1 - d_0 \times \gamma)$;
$d_1 = (u_1 - u_0 \times \gamma) \times \mu$;
$u_1 = w_1 \times \mu$;
$w_1 = (y_1 - w_0 \times \gamma) \times \mu$;
$temp = 0.0$;
for $i = 2;\ i \leqslant N - 3;\ i = i + 1$ **do**
  $temp = s_i$;
  $\gamma = l_i - d_{i-2} \times temp$;
  $\mu = 1.0/(d_i - u_{i-2} \times temp - d_{i-1} \times \gamma)$;
  $d_i = (u_i - u_{i-1} \times \gamma) \times \mu$;
  $u_i = w_i \times \mu$;
  $w_i = (y_i - w_{i-2} \times temp - w_{i-1} \times \gamma) \times \mu$;
end for
$temp = s_{N-2}$;
$\gamma = l_{N-2} - d_{N-4} \times temp$;
$\mu = 1.0/(d_{N-2} - u_{N-4} \times temp - d_{N-3} \times \gamma)$;
$d_{N-2} = (u_{N-2} - u_{N-3} \times \gamma) \times \mu$;
$w_{N-2} = (y_{N-2} - w_{N-4} \times temp - w_{N-3} \times \gamma) \times \mu$;
$temp = s_{N-1}$;
$\gamma = l_{N-1} - d_{N-3} \times temp$;
$\mu = 1.0/(d_{N-1} - u_{N-3} \times temp - d_{N-2} \times \gamma)$;
$w_{N-1} = (y_{N-1} - w_{N-3} \times temp - w_{N-2} \times \gamma) \times \mu$;
// backward substitution:
$x_{N-1} = w_{N-1}$; /* for in-place solution replace x with y */
$x_{N-2} = w_{N-2} - d_{N-2} \times x_{N-1}$;
for $i = N - 3;\ i \geqslant 0;\ i = i - 1$ **do**
  $x_i = w_i - d_i \times x_{i+1} - u_i \times x_{i+2}$;
end for

$$a_{ii} = \left(\sum_{j \neq i,\, |i-j| \leqslant 2} |a_{ij}| \right) + \epsilon, \tag{5}$$

where $\epsilon = 10^{-12}$ is a fixed regularization parameter.

- class C: this class simulates a heavily-damped interface equation. It enforces a heavy, growing diagonal to ensure that the condition number remains low:

$$a_{ii} = 20 + \ln(i+1), \quad a_{i,i\pm1} = -2, \quad a_{i,i\pm2} = 0.5. \tag{6}$$

The true target solution vectors $\boldsymbol{x}_{\text{exact}}$ are explicitly declared prior to the construction of the right-hand side vectors. Two distinct configurations are defined:

1) smooth solution: to assess the optimal convergence and baseline accuracy under continuous fields, a low-frequency harmonic solution is mapped onto a uniform spatial grid, where $x_i = i/N$:

$$x_{\text{exact},i} = \sin(\pi x_i) \cdot \cos(2\pi x_i), \quad \forall i = 0, 1, \ldots, N-1, \tag{7}$$

2) noisy solution: a non-smooth solution is generated by random perturbations of a uniform field:

$$x_{\text{exact},i} = 1.0 + \delta \cdot U_i(-1, 1), \quad \forall i = 0, 1, \ldots, N-1, \tag{8}$$

where $U_i(-1,1)$ represents a uniformly distributed random variable $\in [-1; 1]$ (seed 42), and $\delta = 10^{-2}$ dictates the constant noise scaling amplitude.

Once these target solution vectors are initialized in memory, the mathematically consistent right-hand side vector is constructed via explicit pentadiagonal matrix-vector multiplication, ensuring $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x}_{\text{exact}}$.

Upon obtaining the computed solution vector $\boldsymbol{x}_{\text{comp}}$, the forward error and backward residual metrics are evaluated using the following formulae:

$$\begin{aligned} &\text{absolute error:} && \|e\|_\infty = \|\boldsymbol{x}_{\text{exact}} - \boldsymbol{x}_{\text{comp}}\|_\infty, \\ &\text{relative forward error:} && e_{\text{rel}} = \frac{\|\boldsymbol{x}_{\text{exact}} - \boldsymbol{x}_{\text{comp}}\|_2}{\|\boldsymbol{x}_{\text{exact}}\|_2}, \\ &\text{relative residual:} && r_{\text{rel}} = \frac{\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_{\text{comp}}\|_2}{\|\boldsymbol{b}\|_2}. \end{aligned} \tag{9}$$

Theoretically, the relative forward error is bounded by the product of the matrix condition number and the relative residual:

$$e_{\text{rel}} \leqslant \kappa(A) r_{\text{rel}}. \tag{10}$$

5.1. Numerical results

The results from different matrix sizes N, condition numbers $\kappa(A)$, and structural features are summarized in Tables 4 and 5. The error analysis confirms that the proposed pentadiagonal partition solver achieves high forward accuracy and small residuals and forward errors for the tested cases. For the well-conditioned systems with coefficient matrices from class A ($\kappa \approx 5.54$) and class C ($\kappa \leqslant 2.17$), both the relative residuals and the relative forward errors remain clustered near the machine precision, empirically demonstrating perfect stability.

Table 4. Structural specifications, RHS configurations, and condition number scaling across the test suite.

Matrix class	Size N	Exact sol	Measured $\kappa(A)$	Theoretical scaling	Expected digit loss	Measured κ(interface)
class A	10^3	smooth/noisy	5.54×10^0	$\mathcal{O}(1)$	< 1 digit	2.20×10^1
	10^4	smooth/noisy	5.54×10^0	$\mathcal{O}(1)$	< 1 digit	2.20×10^1
	10^5	smooth/noisy	5.54×10^0	$\mathcal{O}(1)$	< 1 digit	2.20×10^1
class B	10^2	smooth/noisy	1.83×10^{12}	$\mathcal{O}(1/\epsilon)$	≈ 12 digits	9.15×10^{13}
	5×10^2	smooth/noisy	1.83×10^{12}	$\mathcal{O}(1/\epsilon)$	≈ 12 digits	7.57×10^{13}
	10^3	smooth/noisy	1.83×10^{12}	$\mathcal{O}(1/\epsilon)$	≈ 12 digits	7.38×10^{13}
	10^4	smooth/noisy	1.83×10^{12}	$\mathcal{O}(1/\epsilon)$	≈ 12 digits	7.21×10^{13}
	10^5	smooth/noisy	1.83×10^{12}	$\mathcal{O}(1/\epsilon)$	≈ 12 digits	—
	10^6	smooth/noisy	1.83×10^{12}	$\mathcal{O}(1/\epsilon)$	≈ 12 digits	—
class C	10^3	smooth	1.90×10^0	$O(\log N)$ or $O(1)$	< 1 digit	2.10×10^1
	10^4	smooth	2.03×10^0	$O(\log N)$ or $O(1)$	< 1 digit	2.41×10^1
	10^5	smooth	2.17×10^0	$O(\log N)$ or $O(1)$	< 1 digit	—

Table 5. Numerical verification against the exact solution and the serial pentadiagonal solver.

Matrix	Size N	Partition vs. exact solution $\|e\|_\infty$	e_{rel}	r_{rel}	Cross-solver discrepancy $\|x_{\text{partition}} - x_{\text{serial}}\|_\infty$
class A (smooth solution)	10^3	1.55×10^{-15}	4.26×10^{-16}	6.00×10^{-16}	1.55×10^{-15}
	10^4	1.55×10^{-15}	3.97×10^{-16}	5.73×10^{-16}	1.55×10^{-15}
	10^5	2.00×10^{-15}	3.97×10^{-16}	5.76×10^{-16}	2.00×10^{-15}
class A (noisy solution)	10^3	2.00×10^{-15}	3.84×10^{-16}	5.97×10^{-16}	2.33×10^{-15}
	10^4	2.00×10^{-15}	3.93×10^{-16}	5.75×10^{-16}	2.33×10^{-15}
	10^5	2.44×10^{-15}	3.95×10^{-16}	5.76×10^{-16}	2.33×10^{-15}
class B (smooth solution)	10^2	5.56×10^{-5}	1.11×10^{-4}	8.84×10^{-14}	5.51×10^{-5}
	5×10^2	4.36×10^{-6}	8.73×10^{-6}	1.89×10^{-13}	4.32×10^{-6}
	10^3	5.37×10^{-6}	1.07×10^{-5}	3.12×10^{-13}	5.36×10^{-6}
	10^4	9.77×10^{-7}	1.94×10^{-6}	9.74×10^{-13}	9.78×10^{-7}
	10^5	1.46×10^{-7}	1.61×10^{-7}	3.03×10^{-12}	1.46×10^{-7}
	10^6	7.38×10^{-8}	3.94×10^{-8}	1.19×10^{-11}	7.38×10^{-8}
class B (noisy solution)	10^2	2.39×10^{-4}	2.39×10^{-4}	1.62×10^{-12}	2.38×10^{-4}
	5×10^2	1.98×10^{-5}	1.98×10^{-5}	1.61×10^{-12}	1.96×10^{-5}
	10^3	5.80×10^{-6}	5.80×10^{-6}	1.59×10^{-12}	5.75×10^{-6}
	10^4	9.75×10^{-7}	9.65×10^{-7}	1.48×10^{-12}	9.76×10^{-7}
	10^5	1.19×10^{-6}	6.13×10^{-7}	1.69×10^{-12}	1.19×10^{-6}
	10^6	1.19×10^{-6}	1.98×10^{-7}	1.67×10^{-12}	1.19×10^{-6}
class C (smooth solution)	10^3	1.22×10^{-15}	3.30×10^{-16}	3.89×10^{-16}	1.33×10^{-15}
	10^4	1.78×10^{-15}	3.13×10^{-16}	3.51×10^{-16}	1.67×10^{-15}
	10^5	1.89×10^{-15}	3.09×10^{-16}	3.47×10^{-16}	1.89×10^{-15}

Conversely, class B introduces severe, regularized numerical ill-conditioning ($\kappa \approx 1.83 \times 10^{12}$) via explicit negative off-diagonal interactions, causing a planned forward precision drop due to localized digit cancellation. A precision recovery trend emerges as N scales from 10^2 to

10^3, where the relative error for the noisy RHS drops from 2.39×10^{-4} to 5.80×10^{-6}, while the relative residual remains near 1.6×10^{-12}. Upon scaling the problem size up to 10^6, the absolute error plateaus at a ceiling of $\approx 1.19 \times 10^{-6}$, while the relative residual remains strictly bounded at $\approx 1.6 \times 10^{-12}$. The smooth RHS configuration allows the parallel solver to attenuate the absolute error down to 7.38×10^{-8} at $N = 10^6$. The relative error follows a strictly monotonic decay, reaching 3.94×10^{-8}. Although the relative residual exhibits a minor scaling growth from 10^{-14} up to 10^{-11} due to standard floating-point aggregation across millions of elements, the results preserve 7 to 8 digits of pure precision. Furthermore, the cross-solver discrepancy mirrors the absolute error perfectly across all dimensions. This behavior conclusively demonstrates that both the serial and parallel algorithms converge to identical numerical states, proving that the parallel implementation introduces no architectural instability or floating-point degradation even at massive scales.

Across all test suites, the cross-solver discrepancies validate the partition algorithm against the serial pentadiagonal solver. Every experimental metric strictly satisfies the classical numerical linear algebra error bound (10).

6. Discussion and conclusions

We have presented parallel partition algorithms for solving systems of linear algebraic equations with pentadiagonal and heptadiagonal coefficient matrices. Theorems for the sparsity pattern of the partitioned matrices have been formulated and proven. It has been proven that both the pentadiagonal and the heptadiagonal partition algorithms in general do not preserve the diagonal dominance. The pentadiagonal algorithm has been implemented with the help of CUDA.

For SLAEs with size up to 10^4 (for FP32 precision) and up to 10^5 (for FP64 precision), the CPU serial pentadiagonal solver is faster than the rest. The speedup in comparison with the partition method is up to 41.3 times for FP32 precision (SLAE size $N = 10^3$) and 30.4 times for FP64 precision (SLAE size $N = 10^3$). Most probably the reason is that for these relatively small sizes of the SLAEs, the overheads of using the GPU take longer than the computations, and thus cannot be amortized. Effectively, for these SLAE sizes, the GPU remains under-utilized and it spends most of its time waiting for warps to be scheduled, completely negating the compute power benefit it has in comparison with the CPU. Additionally, all times of GPU solvers include the times for the memory transfer of the SLAE to the device and the obtained solution from the device to the host. For bigger than that SLAE sizes, the speedup of the partition algorithm being run on 1 GPU in comparison with the CPU serial pentadiagonal method is up to 3.40 times for FP32 precision (SLAE size $N = 5 \times 10^8$) and up to 1.48 times for FP64 precision (SLAE size $N = 2.5 \times 10^8$). As it can be seen, the cuSPARSE pentadiagonal solver seems to be very slow — it is between 1 and 3 orders of magnitude slower than the pentadiagonal partition method (see Figure 7). The cuSPARSE pentadiagonal solver is in fact a batch solver. According to its documentation, it relies on QR decomposition (in the past there was an opportunity for the developer to choose among three algorithms (cuThomas, LU with pivoting, and QR), now only QR is supported). We believe that what they do is to solve multiple systems in parallel, but if only one system is to be solved, the algorithm uses only one CUDA block. The Nsight Compute profiler [23, 24] shows that the cuSPARSE solver is launched with configuration of 1 block and 256 threads. Therefore, it is slow in comparison with the pentadiagonal partition method. The pentadiagonal partition method is faster than the cuSPARSE pentadiagonal solver for all SLAE sizes. The speedup that we achieve when using the pentadiagonal partition solver is up to 188.6 for FP32 precision (SLAE size $N = 10^8$) and 1015.9 for FP64 precision

(SLAE size $N = 2.5 \times 10^8$). Another observation is that both cuSPARSE and the pentadiagonal partition method (run on 1 GPU) run out of memory simultaneously. However, cuSPARSE overwrites the right-hand side, and the partition method does not. The cuDSS library does not seem to provide very fast solving process of pentadiagonal SLAEs either. For SLAE sizes between 10^4 and 10^7 (for FP64 precision), cuDSS is faster than cuSPARSE, but not faster than the partition algorithm. cuDSS requires a sequential, three-stage pipeline: (1) analysis (performs a purely symbolic evaluation of the matrix sparsity pattern; it applies advanced reordering algorithms to minimize structural fill-in, constructs the elimination tree, and pre-allocates optimal internal device memory buffers), (2) factorization (handles the core floating-point computation by executing numerical decomposition based on the pre-computed symbolic framework), (3) solve stage (computes the target solution). Therefore, for SLAE sizes 10^8 or larger, it runs out of memory. Interestingly, we have achieved comparable computational times for both FP32 and FP64 precision. Most likely the reason is that the computational time is dominated by the analysis stage. The multi-GPU (MGPU) pentadiagonal partition method gives the opportunity for large SLAEs that cuSPARSE cannot handle to be solved on the GPUs, and even bigger speedup — up to 205.4 for FP32 precision (SLAE size $N = 2.5 \times 10^8$) and 1328.1 for FP64 precision (SLAE size $N = 2.5 \times 10^8$). The MGPU partition method becomes more beneficial than the partition method on 1 GPU starting with SLAE size 10^7 (for FP32 precision) and 10^6 (for FP64 precision).

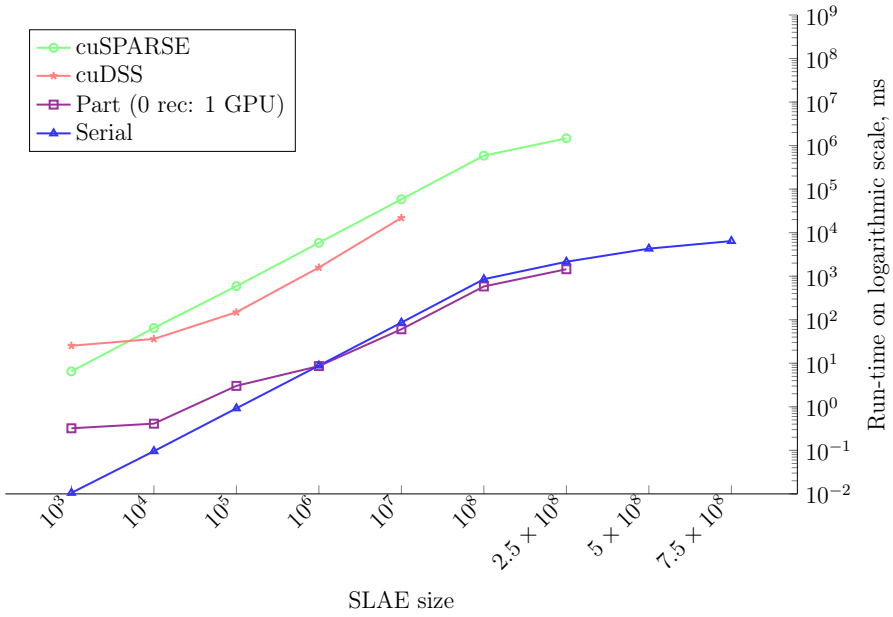


Figure 7. Comparison of the results from the pentadiagonal solvers — cuSPARSE, cuDSS, pentadiagonal partition on 1 GPU, NVIDIA GPU RTX A5000, and the CPU serial solver for different SLAE sizes, FP64 precision.

Remark: the CUDA implementation of the pentadiagonal partition method also allows for overwriting of the right-hand side. If we overwrite the RHS of the numerical method and the kernel responsible for Stage 3, the amount of needed memory decreases with 14% on the host and 14% on the device. When the RHS is not overwritten, 14 arrays are needed on the host and 14 — on the device (this does not count the shared memory and local kernel arrays in the kernel responsible for Stage 3). When the RHS is overwritten, the arrays decrease to 12 on both the host and the device.

Acknowledgements

I would like to thank Dr. Toshiyuki Imamura (RIKEN) and Dr. Alexander Ayriyan (JINR) for their precious comments.

Funding

This work was supported by the Scientific program of the Bulgarian Plenipotentiary Representative at JINR.

Conflicts of interest

The author declares no conflicts of interest.

References

[1] W. Briley, H. McDonald, Solution of the multidimensional compressible Navier–Stokes equations by a generalized implicit method, J. Comput. Phys. 24 (4) (1977) 372–397. doi:10.1016/0021-9991(77)90029-8.

[2] M. Van Daele, G. Berghe, H. De Meyer, Five-diagonal finite difference methods based on mixed-type interpolation for a certain fourth-order two-point boundary-value problem, Comput. Math. Appl. 24 (10) (1992) 55–76. doi:10.1016/0898-1221(92)90019-E.

[3] S. S. Nemani, L. E. Garey, An efficient method for second order boundary value problems with two point boundary conditions, Int. J. Comput. Math. 79 (9) (2002) 1001–1008. doi:10.1080/00207160212122.

[4] J. G. Kim, H. W. Park, Advanced simulation technique for modeling multiphase fluid flow in porous media, in: A. Laganá, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, O. Gervasi (Eds.), Computational Science and Its Applications – ICCSA 2004, Springer Berlin Heidelberg, 2004, pp. 1–9. doi:10.1007/978-3-540-24709-8_1.

[5] A. Duran, M. S. Celebi, S. Piskin, M. Tuncel, Scalability of OpenFOAM for bio-medical flow simulations, J. Supercomput. 71 (3) (2015) 938–951. doi:10.1007/s11227-014-1344-1.

[6] M. Zamani, Approximation by a kind of B-spline method, Indian J. Sci. Technol. 4 (2) (2011) 147–150. doi:10.17485/ijst/2011/v4i2.12.

[7] H. H. Wang, A parallel method for tridiagonal equations, ACM Trans. Math. Softw. 7 (2) (1981) 170–183. doi:10.1145/355945.355947.

[8] T. M. Austin, M. Berndt, J. D. Moulton, A memory efficient parallel tridiagonal solver, Preprint LA-VR-03-4149 (2004).

[9] I. G. Ivanov, C. Walshaw, A parallel method for solving pentadiagonal systems of linear equations, Tech. rep. (1998).

[10] A. Ghosh, C. Mishra, A parallel cyclic reduction algorithm for pentadiagonal systems with application to a convection-dominated Heston PDE, SIAM J. Sci. Comput. 43 (2) (2021). doi:10.1137/20M1311053.

[11] NVIDIA, NVIDIA cuSPARSE Library, NVIDIA (2024).
URL https://docs.nvidia.com/cuda/cusparse/

[12] Intel, Intel oneAPI Programming Guide (2024).
URL https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2024-2/overview.html

[13] Intel, Intel oneAPI Math Kernel Library (oneMKL) – Data Parallel C++ Developer Reference (2022).

[14] R. A. Horn, C. R. Johnson, Matrix Analysis, 2nd Edition, Cambridge University Press, 2012.

[15] G. Smith, Memory required for kernel launch, Tech. rep., NVIDIA Developer Forums (2015). URL https://forums.developer.nvidia.com/t/memory-required-for-kernel-launch/39558/2

[16] G. Smith, What is the memory requirement for a cuda kernel execution?, Tech. rep. (2014). URL https://stackoverflow.com/a/22865553

[17] NVIDIA, NVIDIA cuSOLVER Library, NVIDIA (2026). URL https://docs.nvidia.com/cuda/cusolver/

[18] NVIDIA, NVIDIA CUDA C++ Programming Guide. Design Guide, PG-02829-001_v11.7, NVIDIA Corporation (2022). URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[19] A. A. Samarskii, E. S. Nikolaev, Metody Resheniya Setochnykh Uravnenii, M.: Nauka, 1978.

[20] NVIDIA, NVIDIA RTX A5000 datasheet, Tech. rep. (2021). URL https://nvdam.widen.net/s/wrqrqt75vh/nvidia-rtx-a5000-datasheet

[21] NVIDIA, Introducing NVIDIA RTX A5000 graphics card, Tech. rep. (2021). URL https://www.nvidia.com/en-us/design-visualization/rtx-a5000/

[22] NVIDIA, NVIDIA cuDSS Library, NVIDIA (2026). URL https://docs.nvidia.com/cuda/cudss/

[23] NVIDIA, Kernel Profiling Guide. User Manual (2023).

[24] NVIDIA, Nsight Compute Command Line Interface. User Manual (2022).