Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп’ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА

Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп’ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп’ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 220015.22.03.66 ПЗ

Виконав здобувач IV курсу, гр. КІ2-22-3		Підпис	Сергій ГУМЕНЮК Ініціали, прізвище
Керівник	доктор філософії Науковий ступінь, учене звання	Підпис	Олександр МЕЛЬНИЧЕНКО Ініціали, прізвище
Нормоконтролер	канд.фіз.-мат.наук, доц. Науковий ступінь, учене звання	Підпис	Тетяна КИСІЛЬ Ініціали, прізвище
До захисту допускаю: завідувач кафедри КІІС «___» _червня_ 2026 р. дата		Підпис	Ольга ПАВЛОВА Ініціали, прізвище

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Рівень вищої освіти ПЕРШИЙ (БАКАЛАВРСЬКИЙ)

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
Ольга ПАВЛОВА
" 10 " 01 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Гуменюку Сергію Івановичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI

Керівник проєкту (роботи) Мельниченко Олександр Вікторович, д-р філософії
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Аналіз складності розв'язання слар великої розмірності та виявлення основних проблем обробки даних

Проєктування паралельного алгоритму та архітектури програмного забезпечення

Програмно-апаратна реалізація паралельної системи розв'язання СЛАР

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Блок-схема паралельного алгоритму

Схема взаємодії паралельних процесів

Результати тестування та мікроархітектура

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «_10_»______01________2026 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проєкту (роботи)	Термін виконання етапів проєкту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником	10.01.2026	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об’єкта та предмета дослідження	01.02.2026	виконано
3	Робота над розділом 1 – аналіз складності розв’язання слар великої розмірності та виявлення основних проблем обробки даних	01.03.2026	виконано
4	Робота над розділом 2 – проєктування паралельного алгоритму та архітектури програмного забезпечення	01.04.2026	виконано
5	Робота над розділом 3 – програмно-апаратна реалізація паралельної системи розв'язання слар	29.04.2026	виконано
6	Оформлення пояснювальної записки згідно з вимогами	25.05.2026	виконано
7	Попередній захист ВКР	26.05.2026	виконано
8	Захист ВКР на засіданні ЕК	Червень 2026 року	

Здобувач_______________ ____Сергій ГУМЕНЮК____
Підпис Ім’я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи_______________ ____Олександр МЕЛЬНИЧЕНКО____
Підпис Ім’я, ПРІЗВИЩЕ

№ рядка	формат	Позначення	Найменування	Кіл. листів	№ екз	Примітка
			Текстові документи			
1		КвРКІ 220015.22.03.66 ПЗ	Пояснювальна записка	65		
			Графічні матеріали			
2		КвРКІ 220015.22.03.66 Е8	Блок-схема паралельного алгоритму	1		
3		КвРКІ 220015.22.03.66 Е8	Схема взаємодії паралельних процесів	1		
4		КвРКІ 220015.22.03.66 Е8	Результати тестування та мікроархітектура	1		

					КвРКІ 220015.22.03.66 ВП			
Зм	Арк	№ докум	Підпис	Дата				
Розробив		Гуменюк			Відомість проєкту	Літера	Аркуш	Аркушів
Перевір.		Мельниченко				У	1	1
Н. контр.		Кісіль				ХНУ, КІ2-22-3		
Затв.		Павлова		01.06				

АНОТАЦІЯ

Тема кваліфікаційної роботи: «Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI».

Автор роботи: Сергій ГУМЕНЮК.

Керівник роботи: Олександр МЕЛЬНИЧЕНКО.

Пояснювальна записка: 65 с., 29 рис., 6 табл., 3 дод., 50 джерел.

Графічна частина: 3 креслення.

БАГАТОЯДЕРНА АРХІТЕКТУРА, МАСШТАБОВАНІСТЬ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ПЕРЕДАЧА ПОВІДОМЛЕНЬ, РОЗРІДЖЕНІ МАТРИЦІ, СИСТЕМИ ЛІНІЙНИХ АЛГЕБРАЇЧНИХ РІВНЯНЬ

Метою дипломної роботи є розробка та програмна оптимізація паралельного алгоритму розв'язання систем лінійних алгебраїчних рівнянь надвеликої розмірності, а також оцінка його ефективності та масштабованості у багатоядерних обчислювальних системах.

Об’єктом дослідження є процеси паралельної обробки масивів даних під час розв'язання математичних моделей у середовищі з розподіленою пам'яттю.

Предметом дослідження є оцінка режимів роботи, продуктивності та апаратних обмежень масштабування паралельного ітераційного алгоритму на базі стандарту передачі повідомлень.

Під час проведення даного дослідження були використані методи обчислювальної математики для побудови ітераційного процесу, парадигма передачі повідомлень для організації паралельної взаємодії вузлів, а також методи емпіричного тестування для аналізу швидкодії комп'ютерної системи.

____________________ ____30.05.2026____

Підпис здобувача Дата

ЗМІСТ

КвРКІ. 220015.22.03.66 ПЗ

Зм.	Арк.	№докум.	Підпис	Дата
Виконав		Сергій ГУМЕНЮК		
Перевір.		Олександр МЕЛЬНИЧЕ КО		
Н.контр.		Тетяна КИСІЛЬ		
Затвер.		Ольга ПАВЛОВА		01.06

Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI
Пояснювальна записка

Літера			Аркуш	Аркушів
у			2	72

ХНУ КІ2-22-3

ВСТУП

Під час вирішення багатьох сучасних науково-технічних та інженерних задач, зокрема у сфері математичного моделювання складних фізичних процесів, постійно виникає необхідність розв'язання систем лінійних алгебраїчних рівнянь надвеликої розмірності. Зазвичай такі системи є прямим результатом дискретизації диференціальних рівнянь у частинних похідних за допомогою методів скінченних різниць або скінченних елементів. Використання класичних послідовних алгоритмів для розв'язання подібних завдань нині є малоефективним. Це пов'язано з тим, що обробка великих матриць вимагає надто багато машинного часу та часто значно перевищує доступні обсяги оперативної пам'яті стандартних персональних комп'ютерів. Крім того, наявна проблема обмеженої пропускної здатності системної шини робить неможливим подальше прискорення таких розрахунків виключно за рахунок збільшення тактової частоти одного процесора.

Для успішного подолання цих жорстких апаратних обмежень найбільш доцільно використовувати технології багатопроцесорної обробки даних шляхом побудови розподілених обчислювальних систем. Дослідження паралельних алгоритмів на базі парадигми передачі повідомлень залишається актуальним інженерним завданням, адже цей архітектурний підхід дозволяє зручно та безпечно розділити великі масиви даних і обчислювальне навантаження між багатьма незалежними ядрами. Використання стандарту MPI забезпечує високу портативність розробленого програмного забезпечення на кластерних архітектурах. Завдяки ефективному розпаралелюванню математичних операцій можна не лише обійти фізичні ліміти пам'яті одного обчислювального вузла, але й у рази прискорити отримання кінцевого результату, що є важливим критерієм для проведення складних математичних симуляцій у прийнятні терміни.

Метою дипломної роботи є розробка, програмна реалізація та оцінка масштабованості паралельного алгоритму розв'язання систем лінійних алгебраїчних рівнянь надвеликої розмірності для багатоядерних обчислювальних систем в умовах інтенсивного обміну даними.

Для досягнення поставленої мети необхідно виконати такі завдання дослідження: провести детальний порівняльний аналіз існуючих чисельних методів та обґрунтувати вибір оптимального ітераційного підходу для розпаралелювання; дослідити спеціалізовані формати стисненого зберігання розріджених матриць для мінімізації накладних витрат пам'яті; спроєктувати загальну архітектуру паралельного додатка та побудувати ефективну математичну модель просторової декомпозиції даних; здійснити програмну реалізацію розробленого алгоритму мовою програмування C++ з використанням комунікаційного стандарту MPI; а також провести комплексне навантажувальне тестування системи на надвеликих масивах з подальшим глибоким аналізом отриманих експериментальних метрик продуктивності.

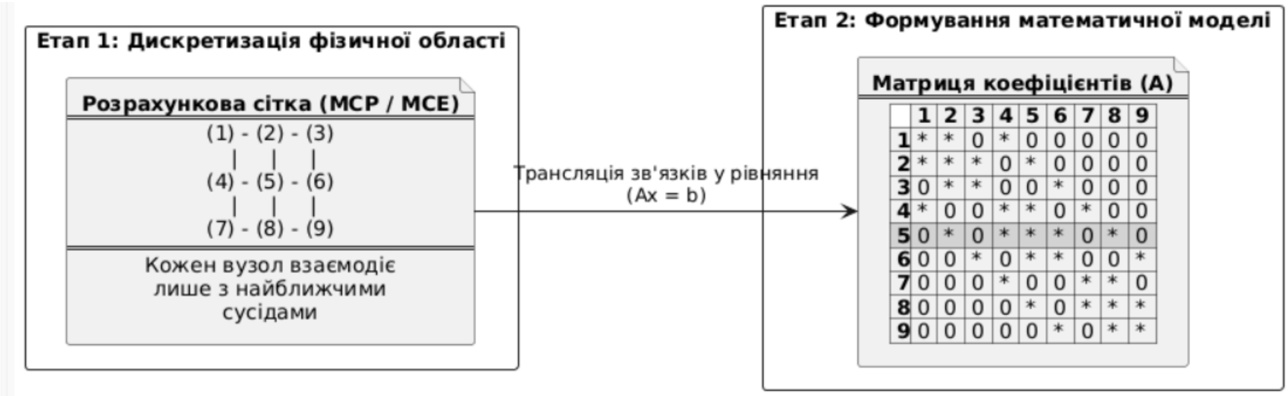
Об’єктом дослідження є процеси паралельної обробки великих масивів даних під час ітераційного розв'язання математичних моделей у високопродуктивному обчислювальному середовищі з розподіленою архітектурою пам'яті.

Предметом дослідження є математичні моделі декомпозиції даних, алгоритми балансування обчислювального навантаження, комунікаційні схеми взаємодії мережевих вузлів та методи оптимізації роботи з пам'яттю під час виконання масштабних паралельних обчислень.

1 АНАЛІЗ СКЛАДНОСТІ РОЗВ’ЯЗАННЯ СЛАР ВЕЛИКОЇ РОЗМІРНОСТІ ТА ВИЯВЛЕННЯ ОСНОВНИХ ПРОБЛЕМ ОБРОБКИ ДАНИХ

1.1 Аналіз предметної області і виявлення наявних проблем і завдань

У багатьох задачах інженерного проєктування, комп'ютерного моделювання фізичних процесів та аналізу великих масивів даних після дискретизації виникає необхідність розв'язання систем лінійних алгебраїчних рівнянь (СЛАР) [45]. Під час проєктування складних технічних конструкцій, розрахунку аеродинаміки, моделювання теплопровідності або електромагнітних полів фахівці найчастіше застосовують метод скінченних елементів або метод скінченних різниць [5, 17]. Суть цих підходів полягає у розбитті суцільного фізичного середовища на значну кількість дрібних комірок у вигляді розрахункової сітки. Взаємозв'язок між вузлами такої сітки описується лінійними залежностями, що формують загальну систему рівнянь. На рисунку 1.1 схематично продемонстровано процес переходу від фізичної моделі об'єкта до дискретної сітки, яка генерує матрицю коефіцієнтів.



	1	2	3	4	5	6	7	8	9
1	*	*	0	*	0	0	0	0	0
2	*	*	*	0	*	0	0	0	0
3	0	*	*	0	0	*	0	0	0
4	*	0	0	*	*	0	*	0	0
5	0	*	0	*	*	*	0	*	0
6	0	0	*	0	*	*	0	0	*
7	0	0	0	*	0	0	*	*	0
8	0	0	0	0	*	0	*	*	*
9	0	0	0	0	0	*	0	*	*

Рисунок 1.1 – Схема дискретизації об'єкта та формування матриці СЛАР

У загальному вигляді згенерована математична модель записується як класичне матричне рівняння, представлене формулою 1.1 [10, 46].

$$Ax = b \tag{1.1}$$

де A – матриця;

x – вектор невідомих;

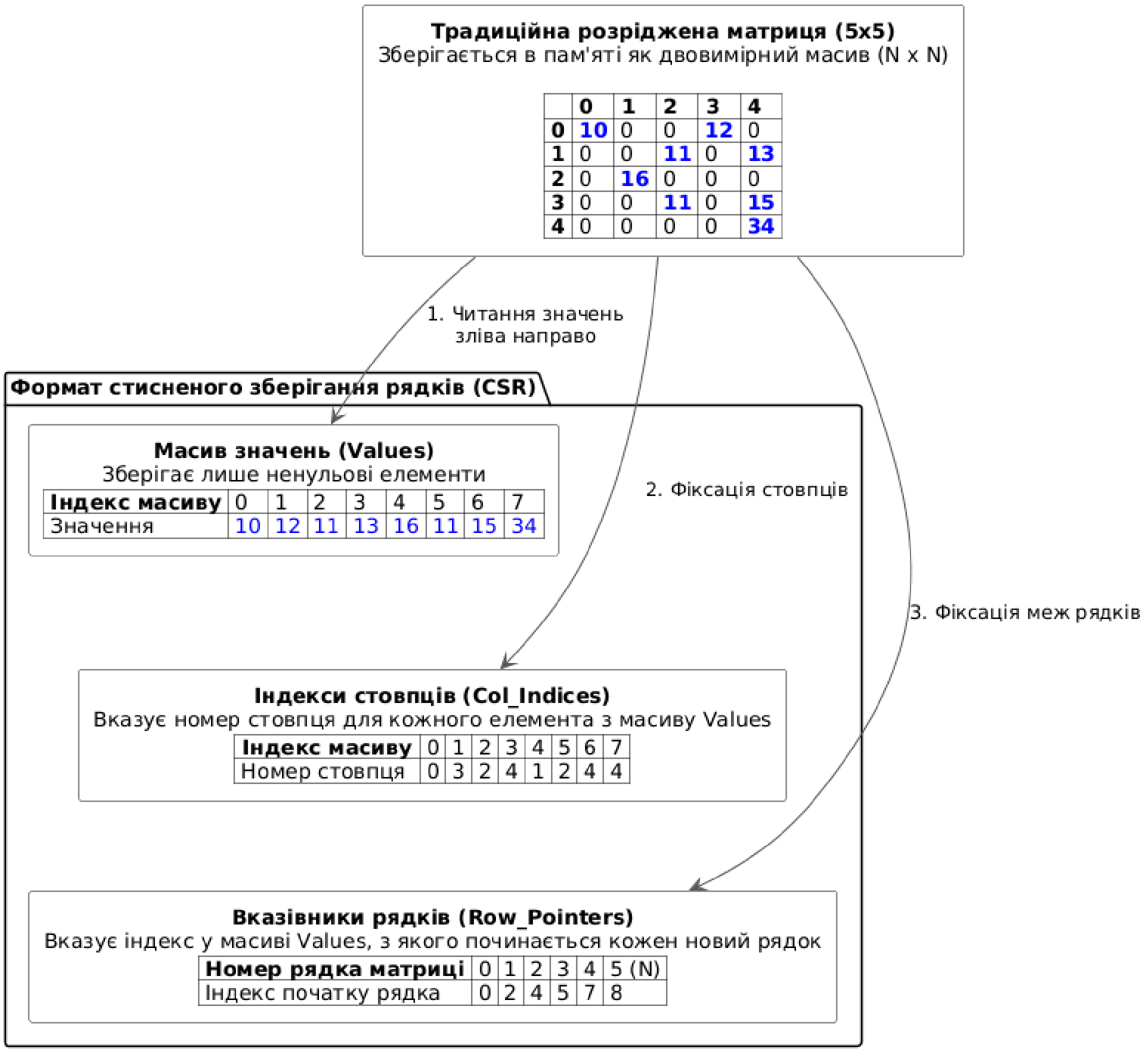
b – вектор вільних членів.

Для отримання високої точності моделювання розрахункова сітка повинна бути максимально деталізованою. Як наслідок, розмірність системи рівнянь, тобто змінна N, стрімко зростає і в сучасних інженерних задачах може сягати сотень тисяч або навіть кількох мільйонів. Робота з матрицями такої надвеликої розмірності одразу виявляє критичну апаратну проблему нестачі оперативної пам'яті комп'ютера. Якщо зберігати матрицю коефіцієнтів у традиційному двовимірному масиві, де під кожне число виділяється вісім байт пам'яті відповідно до стандарту подвійної точності IEEE 754, необхідний обсяг оперативної пам'яті описується квадратичною залежністю, наведеною у формулі 1.2 [6, 8].

$$Mem = N^2 \times 8 \text{ байт} \tag{1.2}$$

Підставивши у цю формулу значення для системи з одного мільйона рівнянь, можна побачити, що для зберігання лише однієї матриці знадобиться близько восьми терабайт оперативної пам'яті. Це фізично унеможливлює виконання розрахунків на звичайних робочих станціях і навіть на потужних багатопроцесорних серверах без використання спеціалізованих підходів до організації даних.

Вирішення проблеми браку пам'яті ґрунтується на природі походження інженерних матриць. Оскільки кожен вузол розрахункової сітки взаємодіє виключно зі своїми найближчими сусідами, переважна більшість елементів матриці коефіцієнтів дорівнює нулю. Відсоток ненульових значень зазвичай становить менше одного відсотка від загального обсягу [20]. Такі структури даних у програмуванні називаються сильно розрідженими матрицями. Зберігання величезної кількості нулів є марнотратством апаратних ресурсів, тому в області високопродуктивних обчислень застосовують формати стисненого зберігання, серед яких стандартом де-факто є формат стисненого зберігання рядків. Цей формат дозволяє повністю відмовитися від двовимірної структури і перейти до трьох одновимірних масивів. Перший масив містить виключно ненульові значення матриці. Другий масив зберігає номери стовпців для кожного з цих значень. Третій масив виконує роль індексу і вказує, з якої позиції у перших двох масивах починається кожен новий рядок вихідної матриці. Принцип такого перетворення наочно проілюстровано на рисунку 1.2.



	0	**1**	**2**	**3**	**4**
0	**10**	0	0	**12**	0
1	0	0	**11**	0	**13**
2	0	**16**	0	0	0
3	0	0	**11**	0	**15**
4	0	0	0	0	**34**

Індекс масиву	0	1	2	3	4	5	6	7
Значення	10	12	11	13	16	11	15	34

Індекс масиву	0	1	2	3	4	5	6	7
Номер стовпця	0	3	2	4	1	2	4	4

Номер рядка матриці	0	1	2	3	4	5 (N)
Індекс початку рядка	0	2	4	5	7	8

Рисунок 1.2 – Схема дискретизації об'єкта та формування розрідженої матриці СЛАР

Використання формату CSR радикально змінює вимоги до обчислювальної системи. Обсяг пам'яті тепер залежить не від квадрата розмірності, а від фактичної кількості ненульових елементів, що описується формулою 1.3.

$$Mem_{CSR} = NNZ \times 8 + NNZ \times 4 + (N + 1) \times 4 \text{ байт} \tag{1.3}$$

де *NNZ* - кількість ненульових елементів.

Для підтвердження ефективності такого підходу було проведено аналітичне порівняння витрат оперативної пам'яті для систем різної розмірності. Результати розрахунків, зведені у таблицю 1.1, доводять, що перехід до стиснених структур даних дозволяє зменшити вимоги до пам'яті у тисячі разів і робить задачу потенційно розв'язною.

Таблиця 1.1 – Порівняння витрат пам'яті

Розмірність системи (N)	Кількість ненульових елементів (NNZ)	Обсяг пам'яті	Обсяг пам'яті (формат CSR)
10 000	70 000	763 МБ	0,84 МБ
50 000	350 000	18,6 ГБ	4,19 МБ
100 000	700 000	74,5 ГБ	8,39 МБ
500 000	3 500 000	1,8 ТБ	41,96 МБ
1 000 000	7 000 000	7,2 ТБ	83,92 МБ

Проте вирішення проблеми зберігання даних оголює наступну критичну проблему обробки інформації, пов'язану з часом виконання. Класичні прямі методи лінійної алгебри, базовим представником яких є метод Гаусса, характеризуються кубічною обчислювальною складністю. Залежність кількості необхідних арифметичних операцій від розмірності матриці описується формулою 1.4 [10].

$$O(N^3) \tag{1.4}$$

Окрім катастрофічного зростання часу обчислень, прямі методи мають ще один суттєвий недолік при роботі з розрідженими структурами. У процесі прямого ходу та виключення змінних на місці початкових нулів з'являються нові ненульові значення. Цей ефект заповнення повністю руйнує формат CSR і знову призводить до переповнення оперативної пам'яті.

Додатковим фактором, що ускладнює розв'язання великих систем, є математична стабільність обчислень. У реальних задачах матриці коефіцієнтів часто є погано обумовленими. Ступінь чутливості системи до похибок округлення визначається числом обумовленості, математичний вираз для якого наведено у формулі 1.5 [46].

$$cond(A) = \left|\left|A\right|\right| \times \left|\left|A^{-1}\right|\right| \tag{1.5}$$

Чим більшим є це число, тим сильніше найменші неточності у представленні чисел з плаваючою комою впливають на кінцевий результат. Через велику кількість послідовних арифметичних операцій машинна похибка швидко накопичується, що робить пряме розв'язання великих систем на одному процесорі не лише занадто довгим, але й математично некоректним [13].

Комплексний аналіз предметної області показує, що розв'язання СЛАР великої розмірності стикається з трьома класичними бар'єрами. Це жорсткі обмеження оперативної пам'яті, неприйнятно довгий час виконання класичних алгоритмів та накопичення похибок обчислень. Подолання цих бар'єрів неможливе в рамках однопроцесорної архітектури і вимагає обов'язкового переходу до паралельного програмування. Необхідно розробити такий програмний підхід, який дозволить розподілити масиви формату CSR між багатьма обчислювальними вузлами і забезпечить знаходження результату за прийнятний час шляхом використання ітераційних математичних алгоритмів.

1.2 Порівняльний аналіз чисельних методів розв’язання лінійних систем та інструментальних засобів високопродуктивного програмування

Вибір оптимального шляху для розв'язання систем лінійних алгебраїчних рівнянь надвеликої розмірності вимагає комплексного підходу. Необхідно проаналізувати як математичні алгоритми знаходження невідомих, так і програмно-апаратні технології організації обчислювального процесу. З математичної точки зору всі існуючі чисельні підходи поділяються на прямі та ітераційні. Прямі методи, засновані на класичному алгоритмі Гаусса або різноманітних матричних розкладах, теоретично гарантують отримання точного результату за скінченну та заздалегідь відому кількість арифметичних кроків. Проте застосування таких підходів до розріджених матриць стикається з нездоланною технічною перешкодою, яка в обчислювальній математиці називається явищем заповнення. Під час виконання прямого ходу алгоритму та виключення невідомих відбувається модифікація вихідної матриці, внаслідок чого на місцях початкових нульових елементів з'являються відмінні від нуля значення. Це повністю нівелює переваги форматів стисненого зберігання даних і призводить до стрімкого вичерпання доступної оперативної пам'яті. Крім того, прямі методи містять жорсткі інформаційні залежності між кроками обчислень, де кожна наступна операція вимагає результатів попередньої. Така специфіка робить їх малоефективними для розподілу між незалежними процесорами.

Альтернативним сімейством математичних підходів є ітераційні методи [31]. Їхня концепція полягає не у прямому знаходженні точної відповіді, а у послідовному наближенні до неї, починаючи з певного початкового припущення. Найбільш простим для розуміння та базовим для паралельної реалізації є метод Якобі [19]. Для його застосування вихідна матриця коефіцієнтів штучно розділяється на три складові: діагональну, строго нижню трикутну та строго верхню трикутну матриці. Загальний вигляд такого розбиття наведено у формулі 1.6.

$$\mathrm{A} = \mathrm{L} + \mathrm{D} + \mathrm{U} \tag{1.6}$$

де D – діагональна;

L – нижня;

U – верхня трикутні матриці.

На основі цього розбиття формується рекурентний ітераційний процес. Нове значення вектора невідомих на поточному кроці обчислюється з використанням значень, отриманих виключно на попередньому кроці. Матричний запис ітераційного процесу методу Якобі відображено у формулі 1.7.

$$x^{k+1} = D^{-1} \times (b - (L + U) \times x^{k}) \tag{1.7}$$

Головна перевага цього методу для багатопроцесорних систем стає очевидною при переході від матричного запису до покомпонентного. Обчислення кожного окремого елемента нового вектора відбувається незалежно від інших елементів цього ж вектора на поточній ітерації. Відповідний математичний вираз для знаходження і-тої компоненти представлено за формулою 1.8.

$$xx_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1, j\neq i}^{n} (a_{ij} \times x_j^{(k)}) \tag{1.8}$$

Така незалежність обчислень дозволяє розділити масив даних на довільну кількість блоків і доручити їх обробку різним процесорам без необхідності постійного блокування пам'яті або очікування проміжних результатів від сусідів. Проте суттєвим недоліком методу Якобі є низька швидкість збіжності, особливо для систем великої розмірності з поганою обумовленістю. Для досягнення прийнятної точності алгоритму може знадобитися виконання десятків тисяч ітерацій.

Більш ефективним рішенням з погляду швидкості збіжності є методи підпросторів Крилова, найвідомішим представником яких є метод спряжених градієнтів [14, 25]. Цей підхід демонструє високу продуктивність при роботі з симетричними додатно визначеними матрицями. Процес обчислень тут базується на пошуку мінімуму квадратичного функціоналу вздовж системи ортогональних векторів. На кожній ітерації алгоритм вимагає обчислення напрямку спуску, кроку вздовж цього напрямку та оновлення вектора нев'язки. Послідовність основних математичних операцій методу спряжених градієнтів наведено у системі рівнянь формулами 1.9 - 1.13.

$$\alpha_k = \frac{r_k^T\, r_k}{p_k^T\, A\, p_k} \tag{1.9}$$

$$x_{k+1} = x_k + \alpha_k p_k \tag{1.10}$$

$$r_{k+1} = r_k - \alpha_k A p_k \tag{1.11}$$

$$\beta_k = \frac{r_{k+1}^T\, r_{k+1}}{r_k^T\, r_k} \tag{1.12}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k \tag{1.13}$$

Аналізуючи наведені рівняння, можна виявити головний недолік цього методу в контексті паралельних обчислень. На відміну від методу Якобі, тут присутні операції скалярного множення векторів для знаходження коефіцієнтів кроку. Оскільки кожен процесор володіє лише частиною вектора, для обчислення загального скалярного добутку необхідно спочатку знайти локальні суми на кожному вузлі, а потім виконати глобальну синхронізацію для їх додавання та розсилки результату всім учасникам. Наявність таких точок обов'язкової синхронізації створює високе навантаження на комунікаційне середовище та обмежує загальну масштабованість алгоритму. Графічне порівняння інформаційних залежностей у розглянутих ітераційних методах зображено на рисунку 1.3.

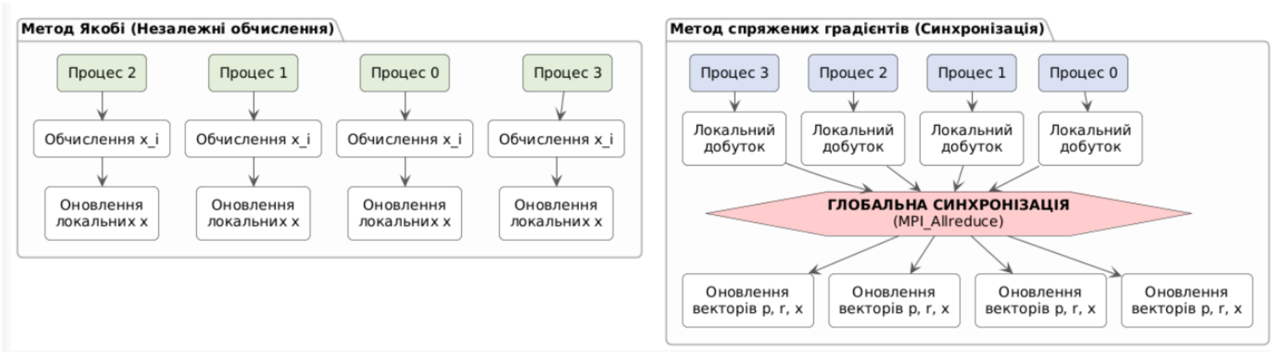


Рисунок 1.3 – Порівняння схем розпаралелювання методів Якобі та спряжених градієнтів

Окрім вибору математичного апарату, критичне значення має вибір архітектури багатопроцесорної системи. З погляду програмної реалізації існують дві домінуючі парадигми: системи зі спільною пам'яттю та системи з розподіленою пам'яттю. Технології зі спільною пам'яттю, стандартом для яких є інтерфейс OpenMP, передбачають створення багатопотокового середовища [2] в межах одного фізичного комп'ютера. Усі запущені потоки мають прямий доступ до єдиного адресного простору оперативної пам'яті. Це значно спрощує розробку програмного забезпечення, оскільки програмісту не потрібно вручну організовувати пересилання даних. Перевагою такого підходу є мінімальні затримки при взаємодії потоків. Однак системи зі спільною пам'яттю мають жорстке апаратне обмеження. Обсяг доступної оперативної пам'яті та кількість обчислювальних ядер лімітовані характеристиками однієї материнської плати. Коли розмір матриці перевищує фізичні можливості одного вузла, використання

OpenMP стає неможливим. Також виникає проблема пропускної здатності шини пам'яті, коли десятки ядер одночасно намагаються отримати доступ до одного масиву даних, створюючи апаратну чергу.

Для розв'язання надвеликих задач, які вимагають терабайтів оперативної пам'яті, одним із практичних рішень є використання кластерних систем з розподіленою пам'яттю [3] на базі стандарту MPI [1, 4]. У такій архітектурі кожен комп'ютер у мережі має власний процесор та власну ізольовану оперативну пам'ять. Процеси не можуть безпосередньо звернутися до змінних, які зберігаються на іншому вузлі. Взаємодія та обмін даними відбуваються виключно через мережеві інтерфейси шляхом явного відправлення та приймання повідомлень. Перевагою підходу MPI є масштабованість, яка обмежується характеристиками мережі, балансуванням навантаження та часткою синхронізацій; для збільшення обчислювальної потужності або обсягу пам'яті до системи можна додавати нові сервери [27, 47]. Водночас недоліком цього рішення є висока складність програмування. Розробник повинен самостійно імплементувати логіку декомпозиції матриці, призначити кожному вузлу його порцію даних та ретельно спроєктувати топологію обміну повідомленнями. Будь-яка неефективність у роботі з мережею призведе до того, що затримки на пересилання даних перевищать час корисних математичних обчислень. Зведені результати порівняльного аналізу архітектурних рішень представлено у таблиці 1.2.

Таблиця 1.2 – Порівняльна характеристика архітектур паралельних обчислень

Критерій порівняння	Системи зі спільною пам'яттю	Системи з розподіленою пам'яттю
Масштабованість	Обмежена кількістю ядер одного фізичного сервера.	Необмежена

Кінець таблиці 1.2

Доступ до пам'яті	Прямий доступ до спільного адресного простору.	Ізольований адресний простір.
Складність розробки	Низька.	Висока.
Накладні витрати	Мінімальні	Значні
Вартість обладнання	Висока	Помірна
Ефективність для СЛАР	Падає при збільшенні розміру задачі через конкуренцію за шину пам'яті.	Зростає при збільшенні розміру задачі

Проведений аналіз демонструє, що для досягнення цілей дослідження доцільно використовувати ітераційні методи лінійної алгебри, оскільки вони не руйнують структуру розріджених матриць і містять масивні операції, придатні для паралелізації. Враховуючи вимоги до обробки даних надвеликої розмірності, програмна реалізація повинна базуватися на технології обміну повідомленнями MPI, що дозволить подолати апаратні обмеження одиничних обчислювальних систем.

1.3 Дослідження методологічних підходів до паралельної реалізації алгоритмів на базі інтерфейсу передачі повідомлень

Розробка ефективного паралельного програмного забезпечення для розв'язання систем лінійних алгебраїчних рівнянь вимагає детального опрацювання стратегії розподілу даних та управління інформаційними потоками між обчислювальними вузлами. Враховуючи обрану архітектуру з розподіленою пам'яттю та специфіку формату стисненого зберігання рядків, базовим методологічним підходом до розпаралелювання є одновимірна декомпозиція області [16, 37]. Цей метод передбачає горизонтальне розбиття матриці коефіцієнтів на смуги, де кожен обчислювальний процес отримує у своє розпорядження певний набір суміжних рядків матриці та відповідні їм елементи вектора вільних членів [16]. Такий вибір зумовлений математичною природою операції множення матриці на вектор, яка є основою всіх ітераційних алгоритмів.

Оскільки результат для конкретного рядка формується як скалярний добуток цього рядка на вектор невідомих, збереження цілісності рядків у пам'яті одного процесора дозволяє уникнути складних часткових сумувань між вузлами та мінімізує кількість мережевих пересилань. Схематичне зображення застосованого підходу до декомпозиції даних наведено на рисунку 1.4.

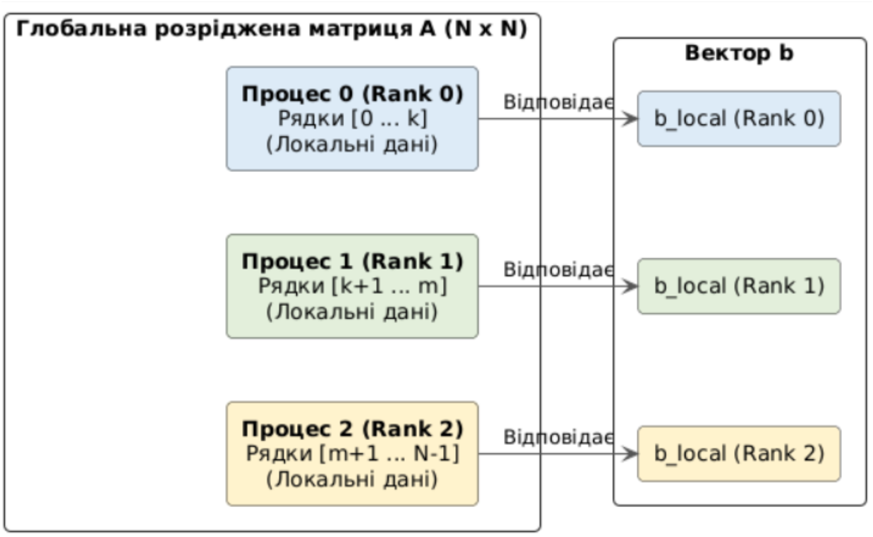


Рисунок 1.4 – Схема одновимірної декомпозиції розрідженої матриці

Важливим аспектом декомпозиції для розріджених матриць є забезпечення рівномірного балансування навантаження. На відміну від щільних матриць, де кожен рядок містить однакову кількість елементів, у розріджених системах заповнення може бути нерівномірним. Просте геометричне розбиття загальної кількості рядків N на кількість процесорів P призведе до ситуації, коли один обчислювальний вузол отримає майже порожні рядки і завершить роботу миттєво, тоді як інший буде обробляти найбільш щільну частину матриці, змушуючи всю систему простоювати в очікуванні. Для вирішення цієї проблеми у роботі застосовується методика балансування за вагою, де критерієм розподілу виступає не кількість рядків, а сумарна кількість ненульових елементів (*NNZ*). Алгоритм попередньо сканує структуру матриці та визначає межі блоків таким чином, щоб кожен процесор виконав приблизно однаковий обсяг арифметичних операцій. Математична умова ідеального балансування описується співвідношенням 1.14.

$$NNZ_{local} \approx \frac{NNZ_{global}}{P} \tag{1.14}$$

де P – кількість процесорів.

Після розподілу статичних даних виникає необхідність організації динамічного обміну інформацією під час ітераційного процесу. Для обчислення нового наближення невідомих кожному процесору потрібен доступ до актуальних значень повного вектора змінних x, отриманих на попередньому кроці. Оскільки вектор x також розподілений між вузлами, жоден процесор не має його повної копії. Найбільш ефективним рішенням у цьому випадку є використання колективної операції збору даних «всі-до-всіх». У стандарті MPI для цього передбачена функція $MPI_{Allgatherv}$ [1, 38]. Ця команда дозволяє кожному процесору надіслати свою локальну частину оновленого вектора всім іншим учасникам комунікації і одночасно отримати від них фрагменти, яких не вистачає. Використання саме векторної версії функції є обов'язковим, оскільки через балансування навантаження процесори можуть володіти різною кількістю компонент вектора. Графічну інтерпретацію процесу обміну даними та відновлення повного вектора на кожному вузлі зображено на рисунку 1.5.

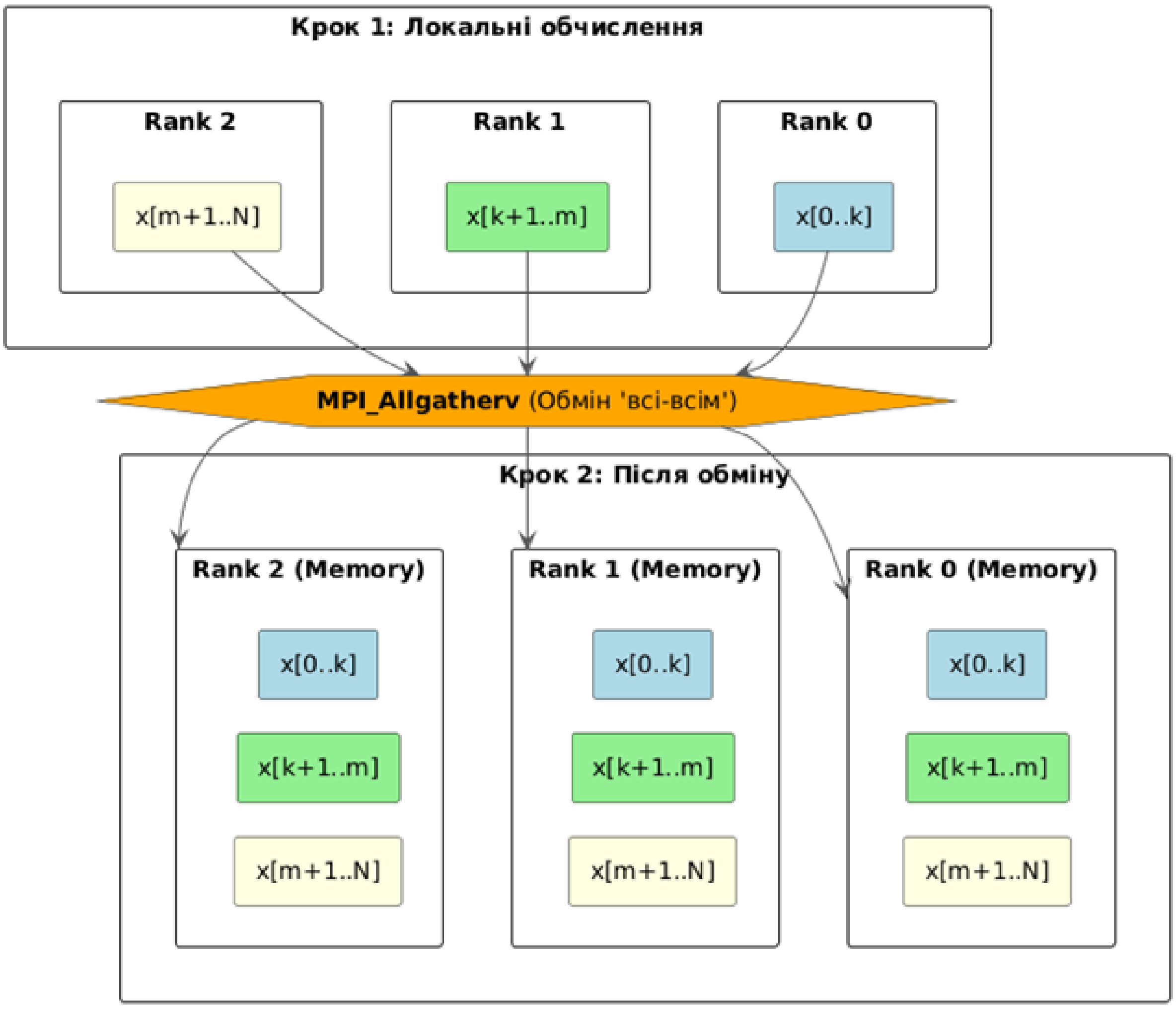


Рисунок 1.5 – Принцип роботи колективної операції збору даних MPI_Allgatherv

Окремої уваги потребує процедура контролю збіжності алгоритму. Ітераційний процес повинен зупинитися тоді, коли глобальна похибка розв'язку стане меншою за заданий поріг точності ϵ. Кожен процесор може обчислити лише локальну частину нев'язки для своїх рядків. Для отримання загального значення необхідно зібрати суми квадратів локальних похибок з усіх вузлів, додати їх і розіслати результат назад для перевірки умови зупинки. Для реалізації цієї логіки оптимальним є використання функції глобальної редукції $MPI_{Allreduce}$ з оператором підсумовування MPI_{SUM}. Ця операція виконується на апаратному рівні мережевих комутаторів або через деревоподібну структуру передачі даних, що забезпечує логарифмічну залежність часу виконання від кількості процесорів, на відміну від лінійної залежності при ручному зборі даних на один керуючий вузол. Загальна норма нев'язки розраховується через механізм редукції за формулою 1.15.

$$||r|| = \sqrt{(\text{ Сума по всіх процесорах (Сума локальних } r_i^2))} \qquad (1.15)$$

Загальна структура розробленого програмного забезпечення будується за моделлю SPMD, де одна й та сама скомпільована програма запускається на всіх вузлах кластера, але обробляє різні дані залежно від свого унікального рангу MPI. Такий підхід дозволяє організувати як симетричні операції, що виконуються всіма вузлами одночасно, так і асиметричні, де один процес виконує роль координатора.

Життєвий цикл програми розпочинається з ініціалізації середовища MPI та завантаження матриці керуючим процесором. Після цього відбувається широкомовна розсилка параметрів задачі за допомогою MPI_Bcast, що дозволяє кожному вузлу підготувати локальні буфери пам'яті. Далі відбувається фізична роздача даних за допомогою функції MPI_Scatterv, яка є ключовою для забезпечення раніше описаного балансування навантаження. Після цього програма переходить до основного циклу ітераційних обчислень, що складається з локальних математичних операцій та періодичних глобальних синхронізацій через колективні виклики MPI_Allgatherv та MPI_Allreduce. Перша функція відповідає за збір оновленого вектора невідомих, а друга – за перевірку умови збіжності. Процес завершується фінальним збором результатів функцією MPI_Gatherv на керуючий вузол для запису у файл [39]. Детальна послідовність дій та логіка взаємодії компонентів паралельного алгоритму відображена на блок-схемі алгоритму, що подана на рисунку 1.6.

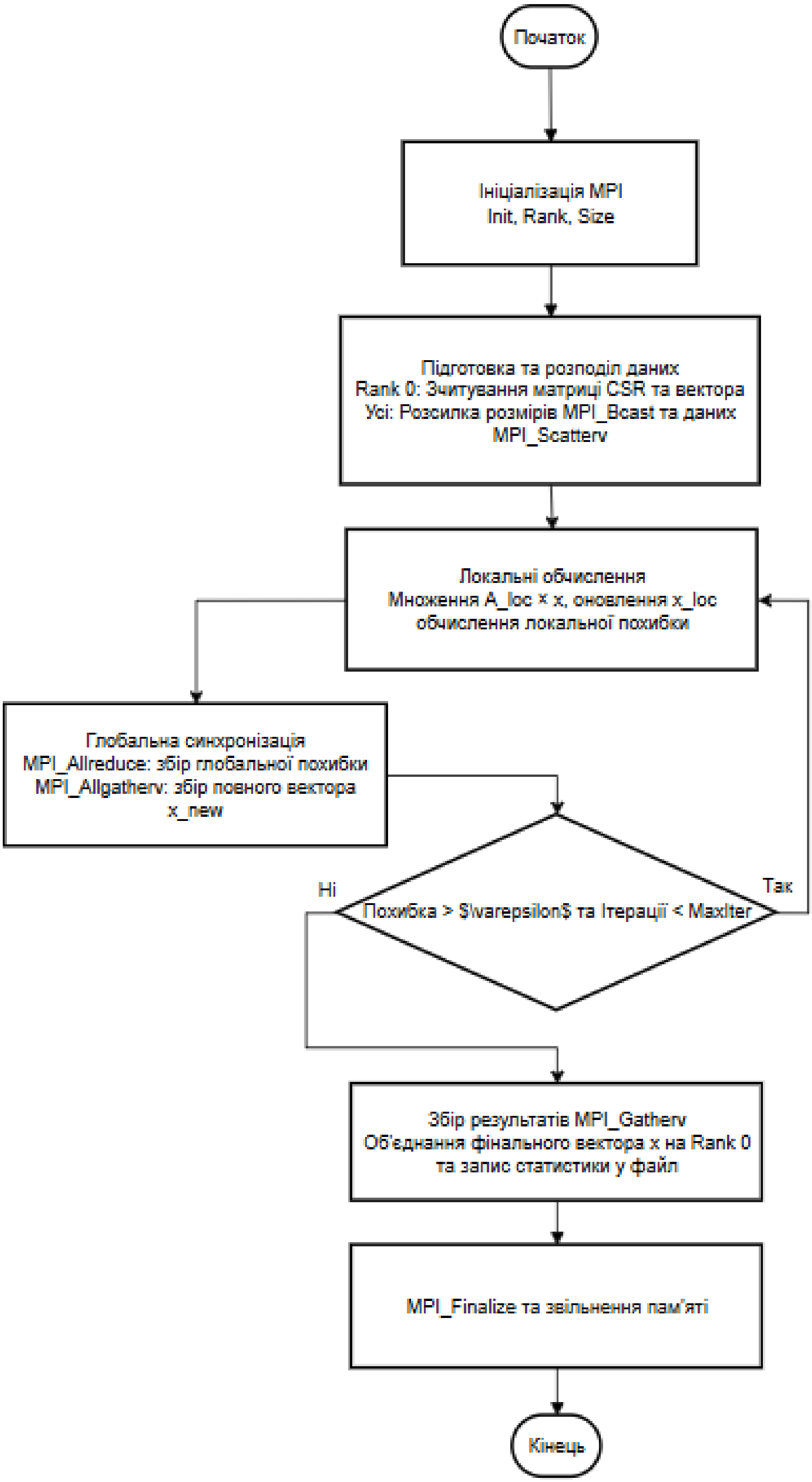


Рисунок 1.6 – Блок-схема розробленого паралельного алгоритму

Такий підхід дозволяє мінімізувати кількість повідомлень «точка-точка», які мають високу латентність, і максимально ефективно використати пропускну здатність мережі за допомогою оптимізованих колективних операцій, забезпечуючи високу масштабованість рішення.

1.4 Висновки до першого розділу

У межах першого розділу кваліфікаційної роботи проведено комплексний аналіз предметної області, пов'язаної з розв'язанням систем лінійних алгебраїчних рівнянь великої розмірності. Розглянуто проблеми класичних підходів, зокрема критичну нестачу оперативної пам'яті при використанні традиційних двовимірних масивів та занадто довгий час виконання прямих математичних методів. Для вирішення проблеми зберігання величезних обсягів даних обґрунтовано доцільність використання формату стисненого зберігання рядків, що дозволяє суттєво зменшити вимоги до апаратних ресурсів комп'ютера за рахунок збереження лише ненульових елементів матриці.

Під час вибору оптимального математичного апарату та технологічного стеку було з'ясовано, що конкретний алгоритм безпосередньо визначає ефективність усієї паралельної системи. Зокрема, ітераційні методи було визначено як найбільш придатні для роботи із розрідженими структурами, оскільки вони дозволяють уникнути явища заповнення матриці новими значеннями, що є характерним для прямих методів типу алгоритму Гаусса. Порівняльний аналіз методів Якобі та спряжених градієнтів виявив важливий компроміс: якщо метод Якобі забезпечує високий ступінь паралелізму через повну незалежність обчислень, то метод спряжених градієнтів хоч і збігається значно швидше, проте вимагає інтенсивного обміну даними між процесорами для синхронізації скалярних добутків. Вибір технології MPI як основної платформи для реалізації було обґрунтовано неможливістю використання систем зі спільною пам'яттю для задач надвеликої розмірності, оскільки вони обмежені ресурсами однієї материнської плати. Застосування моделі передачі повідомлень дозволяє створювати гнучкі обчислювальні кластери, де кожен вузол працює у власному адресному просторі, що є єдиним шляхом для обробки матриць, обсяг яких вимірюється гігабайтами або навіть терабайтами.

Важливим аспектом розробки є також визначення методів перевірки якості майбутнього програмного рішення. Оскільки головною метою впровадження паралельних алгоритмів є реальне скорочення часу обчислень, виникає потреба у подальшому розрахунку показників прискорення та ефективності розпаралелювання. Це дозволить на практиці перевірити, як саме обрана математична модель адаптується до специфіки конкретного апаратного забезпечення та чи вдалося досягти оптимального балансу між корисними математичними розрахунками та накладними витратами на мережеві пересилання даних між обчислювальними вузлами.

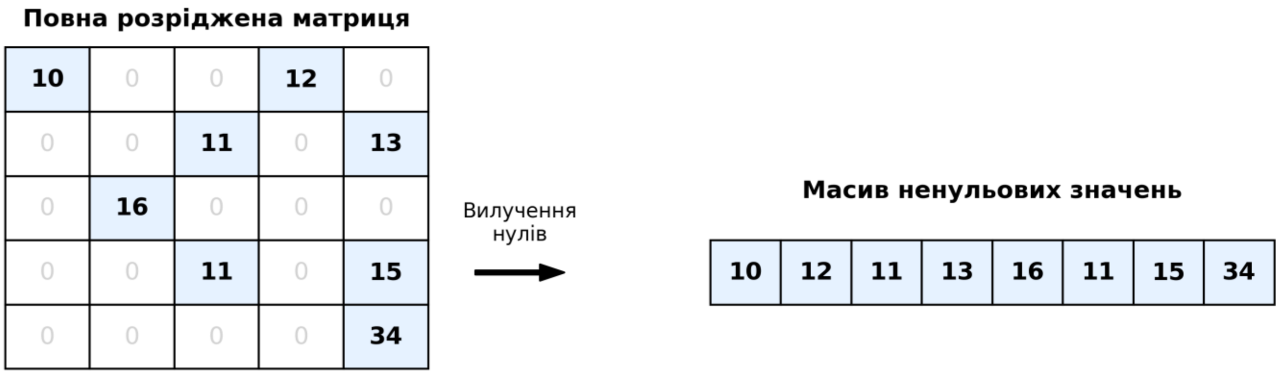
Розглядаючи питання безпосередньої організації паралельних обчислень, було встановлено, що базовою стратегією є одновимірна рядкова декомпозиція матриці з обов'язковим динамічним балансуванням навантаження між процесорами на основі фактичної кількості ненульових елементів у кожному блоці. Також ідентифіковано основні механізми мережевої взаємодії, зокрема використання колективних операцій типу «всі-до-всіх» для збору оновлених частин вектора невідомих та операцій глобальної редукції для розрахунку загальної похибки розв'язку на кожній ітерації.

Підсумовуючи вищевикладене, можна стверджувати, що головний виклик у цій задачі зводиться до грамотного розподілу розріджених структур даних та мінімізації латентності мережевого обміну. Виконана теоретична підготовка слугує надійним математичним підґрунтям для переходу до стадії проєктування архітектури додатка та безпосередньої програмної реалізації паралельного алгоритму на базі обраних технологій.

2 ПРОЄКТУВАННЯ ПАРАЛЕЛЬНОГО АЛГОРИТМУ ТА АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Проєктування структур даних для зберігання розріджених матриць

Ефективність розв’язання систем лінійних алгебраїчних рівнянь надвеликої розмірності у паралельному обчислювальному середовищі критично залежить від обраної математичної моделі представлення даних в оперативній пам’яті обчислювальних вузлів. Процес дискретизації складних фізичних процесів за допомогою методів скінченних різниць або скінченних елементів генерує матриці коефіцієнтів, які характеризуються великими розмірностями, проте мають надзвичайно низький ступінь заповнення. Відсоток ненульових елементів у таких системах зазвичай становить десяті або навіть соті частки відсотка. Традиційне зберігання подібних матриць у вигляді класичних двовимірних масивів спричиняє квадратичне зростання вимог до пам’яті $O(N^2)$, що робить програмну реалізацію неможливою [6, 20] через швидке вичерпання апаратних ресурсів навіть на найсучасніших суперкомп’ютерних системах [7]. Більше того, опрацювання нульових елементів під час арифметичних операцій призводить до значних втрат процесорного часу на виконання порожніх інструкцій. Концептуальна схема переходу від повної матриці до розрідженої структури наведена на рисунку 2.1.



10	0	0	12	0
0	0	11	0	13
0	16	0	0	0
0	0	11	0	15
0	0	0	0	34

10	12	11	13	16	11	15	34

Рисунок 2.1 – Концептуальна схема формування розрідженої структури даних

Причина виникнення великої кількості нульових значень криється у математичній специфіці методів дискретизації суцільних середовищ. Під час застосування методу скінченних різниць або скінченних елементів фізична область розбивається на вузли розрахункової сітки. Кожен такий вузол взаємодіє виключно зі своїми найближчими просторовими сусідами. У матричному поданні системи рівнянь це означає, що для кожного рядка кількість ненульових коефіцієнтів дорівнює кількості сусідніх вузлів, що найчастіше становить від п'яти до двадцяти семи значень. Ця кількість залишається сталою незалежно від загального розміру матриці. Відповідно, зі збільшенням деталізації розрахункової сітки загальна кількість рівнянь зростає, а частка корисних даних у матриці стрімко наближається до нуля.

Спроба зберегти дану структуру у вигляді двовимірного масиву вимагає обсягів оперативної пам'яті, які пропорційні квадрату розмірності системи. Наприклад, для збереження однієї повної матриці на один мільйон рівнянь з використанням чисел подвійної точності знадобиться вісім терабайт оперативної пам'яті. Жоден сучасний обчислювальний вузол середнього класу не володіє такими апаратними можливостями. Водночас зберігання лише ненульових значень тієї ж матриці вимагає всього кілька десятків мегабайт пам'яті. Окрім апаратних обмежень місткості, обробка двовимірного масиву з нулями вимагає виконання марних операцій множення та додавання, що робить час роботи алгоритму неприйнятно довгим. Зазначені фактори роблять перехід до спеціалізованих структур стиснення єдиним можливим шляхом вирішення проблеми.

З огляду на зазначені фактори, базовим архітектурним рішенням на етапі проєктування програмного забезпечення є повна відмова від зберігання нульових значень та перехід до спеціалізованих форматів стиснення даних. Серед існуючої множини структур на етапі проєктування розглядалися різні підходи до стиснення даних. Координатний формат передбачає збереження трьох окремих лінійних масивів однакової довжини для запису самого числа, номера його рядка та номера його стовпця. Такий спосіб організації даних є найпростішим під час початкового формування математичної моделі, проте він містить значну надлишкову інформацію. Номер одного й того ж рядка багаторазово дублюється для кожного елемента, що призводить до нераціонального споживання простору пам'яті. Інша альтернатива у вигляді формату стисненого зберігання стовпців усуває описану надлишковість, але його застосування значно ускладнює програмну реалізацію ітераційних методів лінійної алгебри. Основною математичною дією обраного алгоритму є обчислення скалярного добутку рядка матриці на вектор невідомих. Завантаження масивів з пам'яті за стовпцями змушує процесор постійно змінювати адреси зчитування. Це руйнує просторову локальність даних та спричиняє масові промахи у кеш-пам'яті. З огляду на це, для реалізації ітераційних методів у системах з розподіленою пам'яттю найвищу продуктивність демонструє формат стисненого зберігання рядків. Вибір цього формату обґрунтовується його здатністю забезпечувати оптимальну просторову локальність даних у кеш-пам'яті мікропроцесора [9, 12] та ідеальною сумісністю з алгоритмами покомпонентного множення матриці на вектор.

Даний підхід є зручним для початкового формування математичної моделі, проте він містить надлишкову інформацію, що призводить до зайвого споживання оперативної пам'яті. Формат стисненого зберігання стовпців вирішує проблему надлишковості, але його застосування ускладнює програмну реалізацію ітераційних алгоритмів. Основною математичною операцією обраного методу розв'язання є скалярний добуток рядка матриці на вектор невідомих. Завантаження даних по стовпцях порушує послідовність зчитування елементів цього вектора. З огляду на це, формат стисненого зберігання рядків виявився найбільш пристосованим для поставленої задачі. Вибір цього формату обґрунтовується його здатністю забезпечувати оптимальну просторову локальність даних у кеш-пам'яті мікропроцесора [9, 12] та ідеальною сумісністю з алгоритмами покомпонентного множення матриці на вектор. Формат стисненого зберігання рядків трансформує двовимірну топологію матриці у детерміновану систему з трьох лінійних одновимірних масивів. Структурна організація формату стисненого зберігання рядків та логічні взаємозв'язки між його масивами зображені на рисунку 2.2.

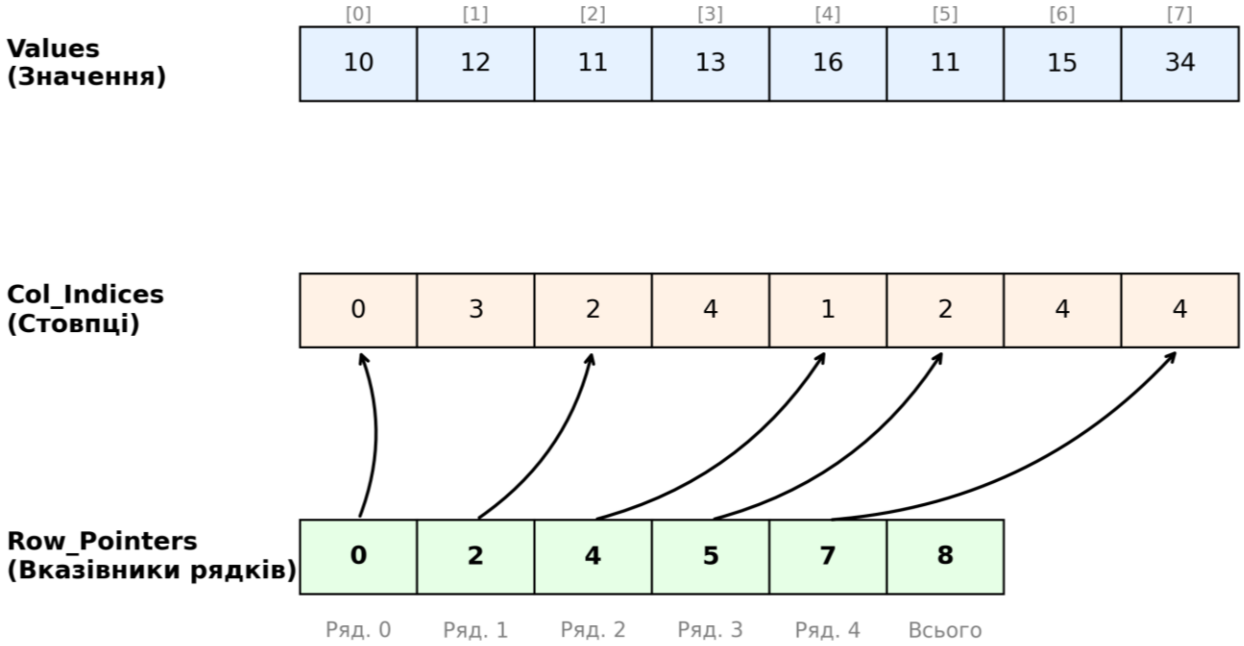


Рисунок 2.2 – Структурна організація формату стисненого зберігання рядків

Першим компонентом цієї структури є масив дійсних чисел, який призначений для зберігання безпосередньо самих ненульових коефіцієнтів системи рівнянь [8]. Ці елементи записуються у пам'ять строго послідовно, рядок за рядком, зліва направо, ігноруючи будь-які нульові проміжки. Для забезпечення необхідної точності інженерних та наукових розрахунків цей масив проєктується з використанням базового типу даних подвійної точності відповідно до стандарту IEEE 754 [4]. Загальна довжина цього масиву дорівнює точній кількості ненульових елементів у вихідній матриці.

Проте самого лише масиву значень недостатньо для відновлення початкової структури системи, оскільки втрачається інформація про просторове розташування коефіцієнтів. Тому другим невід'ємним елементом формату виступає масив індексів стовпців. Цей масив є повністю синхронізованим за розміром із масивом значень і містить цілочисельні координати. Кожен елемент масиву індексів стовпців чітко вказує, у якому саме стовпці оригінальної матриці знаходилося відповідне число з масиву значень. Використання стандартного цілочисельного типу для індексації дозволяє оптимізувати витрати оперативної пам'яті.

Найбільш складною та важливою частиною спроєктованої структури є третій масив – масив вказівників рядків. Його розмірність дорівнює кількості рівнянь у системі плюс один додатковий елемент. Ця структура виконує роль глобального навігатора по перших двох масивах. Кожне значення у масиві вказівників рядків позначає індекс, з якого розпочинається запис нового рядка у масиві значень та масиві індексів стовпців. Останній елемент цього масиву завжди зберігає загальну кількість ненульових елементів. Така організація дозволяє алгоритму за константний час $O(1)$ визначати межі будь-якого необхідного рядка та миттєво обчислювати кількість ненульових елементів у ньому шляхом знаходження різниці між двома сусідніми вказівниками.

Для наочної систематизації параметрів спроєктованих масивів наведено таблицю 2.1.

Найбільш складною частиною цієї структури є третій масив, який містить вказівники рядків. Його розмірність дорівнює кількості рівнянь у системі плюс один додатковий елемент. Ця структура виконує роль глобального навігатора по перших двох масивах. Кожне значення у масиві вказівників позначає початковий індекс, з якого розпочинається запис нового розрахункового рядка у масиві значень та масиві індексів стовпців. Останній елемент цього масиву завжди зберігає загальну кількість ненульових елементів матриці. Наявність цього навігаційного масиву повністю усуває необхідність лінійного перебору всієї структури для пошуку потрібного рівняння. Логіка визначення меж базується на простій адресній арифметиці. Така організація дозволяє алгоритму за константний час визначати координати будь-якого необхідного рядка. Для отримання кількості ненульових елементів у конкретному рівнянні процесору достатньо звернутися до масиву вказівників, взяти значення за індексом наступного рядка та відняти від нього значення за індексом поточного рядка. Отримане число визначає точну межу внутрішнього арифметичного циклу обчислень. Завдяки цьому обчислювальна система ніколи не звертається до порожніх ділянок і виконує рівно стільки математичних операцій, скільки корисних коефіцієнтів містить відповідне рівняння.

Таблиця 2.1 – Параметризація масивів формату стисненого зберігання рядків

Назва програмного масиву	Базовий тип даних	Функціональне призначення у структурі
Масив значень	double (8 байт)	Послідовне зберігання всіх ненульових коефіцієнтів матриці системи
Індекси стовпців (Col_Indices)	integer (4 байти)	Збереження просторової координати стовпця для кожного ненульового коефіцієнта

Вказівники рядків (Row_Pointers)	integer (4 байти)	Визначення точок входу та фіксація меж кожного рядка в одновимірному просторі

Для реалізації даної структури потрібно підібрати базові типи даних у програмному коді. Стандартний знаковий цілочисельний тип даних здатний адресувати трохи більше двох мільярдів елементів. Для систем малої та середньої розмірності цього діапазону цілком достатньо. Проте для інженерних задач надвеликої розмірності загальна кількість ненульових значень легко перевищує вказану межу. У разі використання стандартних типів для запису великих обсягів даних відбувається переповнення розрядної сітки. Значення індексу набуває від'ємного числа, що неминуче призводить до помилки доступу до пам'яті та аварійного завершення роботи операційної системи. Для запобігання цій апаратній проблемі спроєктована архітектура базується на використанні розширених шістдесят чотирьох бітних цілочисельних типів для масивів індексів стовпців та вказівників рядків. Таке інженерне рішення збільшує загальні витрати оперативної пам'яті на зберігання координатної інформації, але гарантує абсолютну надійність адресації та стабільність обчислювального процесу під час роботи з матрицями надвеликих обсягів.

Окрім матриці коефіцієнтів, математична модель ітераційних методів потребує виділення оперативної пам'яті для зберігання векторних величин. До них належать вектор вільних членів, вектор початкового наближення, який у процесі ітерацій трансформується у вектор кінцевих розв'язків, а також допоміжні вектори, такі як вектор нев'язки. Оскільки вектори за своєю природою не є розрідженими і майже завжди повністю заповнені відмінними від нуля значеннями, застосування до них алгоритмів стиснення є недоцільним [**Помилка! Джерело посилання не знайдено.**]. Тому вони проєктуються як класичні одновимірні масиви значень подвійної точності, довжина яких строго відповідає розмірності системи лінійних алгебраїчних рівнянь.Головною перевагою спроєктованої архітектури даних є її повна сумісність із парадигмою обміну повідомленнями у кластерних системах. Організація пам'яті для зберігання векторних величин також визначає загальну швидкодію алгоритму. На відміну від матриці коефіцієнтів, вектори невідомих та вектори вільних членів розміщуються у пам'яті у вигляді звичайних неперервних масивів. Під час обчислення ітерації процесор послідовно та безперебійно зчитує стиснену матрицю. Натомість доступ до елементів глобального вектора невідомих здійснюється на основі значень, які зберігаються у масиві індексів стовпців. Такий підхід створює явище непрямої адресації. Обчислювальне ядро змушене звертатися до оперативної пам'яті хаотично, залежно від просторової топології розрахункової сітки. Апаратні механізми попередньої вибірки даних мікропроцесора не здатні спрогнозувати такі нелінійні стрибки за адресами пам'яті, що періодично викликає затримки у постачанні даних до арифметичних пристроїв. Мінімізація впливу цього явища досягається виключно завдяки високій місткості кеш-пам'яті третього рівня на обчислювальних вузлах, де часто використовувані елементи вектора зберігаються поблизу процесорних ядер.

Стандарт MPI найбільш ефективно опрацьовує лінійні, неперервні блоки оперативної пам'яті. Завдяки тому, що формат стисненого зберігання рядків складається виключно з одновимірних масивів, процес декомпозиції матриці та розподілу її фрагментів між обчислювальними вузлами реалізується за допомогою базових комунікаційних функцій. Програмному додатку не потрібно здійснювати складну серіалізацію даних або конструювати похідні типи MPI, що кардинально знижує накладні витрати процесорного часу на підготовку повідомлень та забезпечує максимальну пропускну здатність мережевих інтерфейсів під час паралельних обчислень.

2.2 Розробка архітектури додатка та топології мережевого обміну

Проєктування архітектури паралельного програмного забезпечення для розв’язання систем лінійних алгебраїчних рівнянь базується на моделі SPMD [9], що передбачає виконання ідентичного програмного коду на всіх обчислювальних вузлах кластера [4, 7], але з різними наборами даних залежно від ідентифікатора процесу. Вибір цієї моделі зумовлений специфікою систем із розподіленою пам’яттю, де кожен процес функціонує у власному ізольованому адресному просторі, а синхронізація обчислень досягається виключно через явні виклики функцій обміну повідомленнями. Життєвий цикл розробленого додатка розпочинається з ініціалізації середовища передачі повідомлень за допомогою базових функцій бібліотеки MPI, які дозволяють визначити загальну кількість доступних процесорів та присвоїти кожному з них відповідний ранг у межах глобального комунікатора.

На цьому етапі вибудовується логічна топологія взаємодії обчислювальних вузлів. Відповідно до обраної парадигми програмування, архітектура додатка використовує єдиний виконуваний файл для всіх учасників обчислень. Проте внутрішня логіка виконання чітко розгалужується на основі мережевого ідентифікатора процесу. Один із процесів із рангом нуль бере на себе роль головного координатора. Усі інші процеси ініціалізуються виключно як рівноправні обчислювальні одиниці, завдання яких зводиться до обробки отриманих масивів та повернення результатів обчислень.

Головний координатор отримує виключне право на операції введення та виведення інформації. До його обов'язків належить зчитування конфігураційних параметрів системи та початкове завантаження матриці коефіцієнтів з пристрою постійного зберігання. Ця процедура включає не лише фізичне перенесення даних у пам'ять, але й первинний синтаксичний аналіз розрахункових файлів, які формують математичну модель. Така сувора ієрархічна організація дозволяє централізувати доступ до зовнішніх апаратних ресурсів.

Якби програмна архітектура дозволяла кожному обчислювальному процесу самостійно звертатися до дискового накопичувача для зчитування своєї частини рівнянь, виникло б явище масового конкурентного доступу до файлової системи. В умовах роботи з гігабайтними масивами даних паралельні запити на читання призвели б до значного падіння пропускної здатності дискової шини. Апаратні контролери швидко вичерпали б ліміт паралельних черг вводу-виводу, що спричинило б неминуче тимчасове блокування потоків операційною системою. Зосередження функцій зчитування на одному вузлі повністю усуває описані конфлікти та забезпечує стабільне формування початкових умов задачі в оперативній пам'яті.

Поки головний вузол здійснює підготовку структур даних, інші учасники мережі перебувають у режимі очікування. У цей час вони виконують попереднє виділення вільного адресного простору у власній оперативній пам'яті для майбутнього прийому інформації. Для правильної резервації пам'яті керуючий процес попередньо розсилає базові метадані про розмірність системи та очікувані обсяги фрагментів. Це дозволяє уникнути динамічного перерозподілу пам'яті під час активної фази обміну повідомленнями, що могло б викликати небажані програмні затримки.

Після завершення зчитування та підготовки буферів головний вузол ініціює процедуру дистрибуції фрагментів розрідженої матриці. Такий послідовний підхід гарантує безпечний перехід системи від стадії конфігурування до стадії активних математичних операцій та забезпечує безперебійний моніторинг загального стану обчислювального процесу. Загальна послідовність етапів функціонування паралельного додатка та логіка переходів між ними представлена на рисунку 2.3.

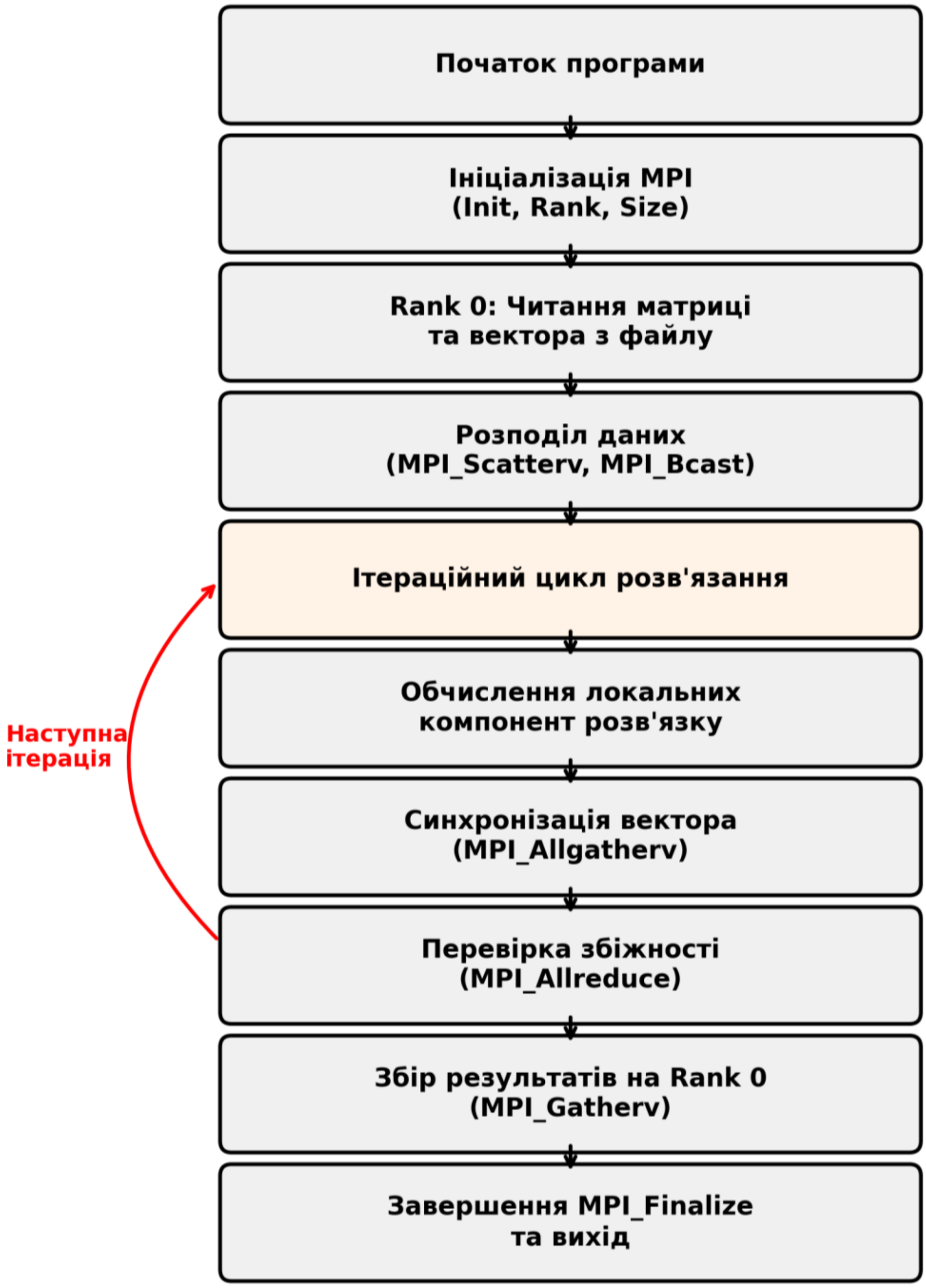


Рисунок 2.3 – Алгоритмічна структура життєвого циклу паралельного додатка

Важливим етапом з погляду мінімізації накладних витрат є фаза підготовки та розподілу початкових даних [24]. Оскільки повна матриця у форматі CSR та вектори системи зчитуються з файлової системи лише керуючим процесором, виникає необхідність фізичного пересилання відповідних фрагментів у локальну пам’ять інших вузлів. Для цього використовується стратегія групових комунікацій, яка є значно ефективнішою за послідовну відправку повідомлень «точка-точка» [11]. Розподіл масивів значень та індексів стовпців реалізується за допомогою функції векторної розсилки [15, 43], що дозволяє враховувати нерівномірність обсягів даних у різних рядках розрідженої матриці. Одночасно з цим параметри розмірності системи та вектор вільних членів транслюються всім учасникам обчислень через широкомовні запити, що забезпечує ідентичність контексту задачі на кожному ядрі.

Схема логічної організації мережевого обміну під час роздачі та збору даних наведена на рисунку 2.4.

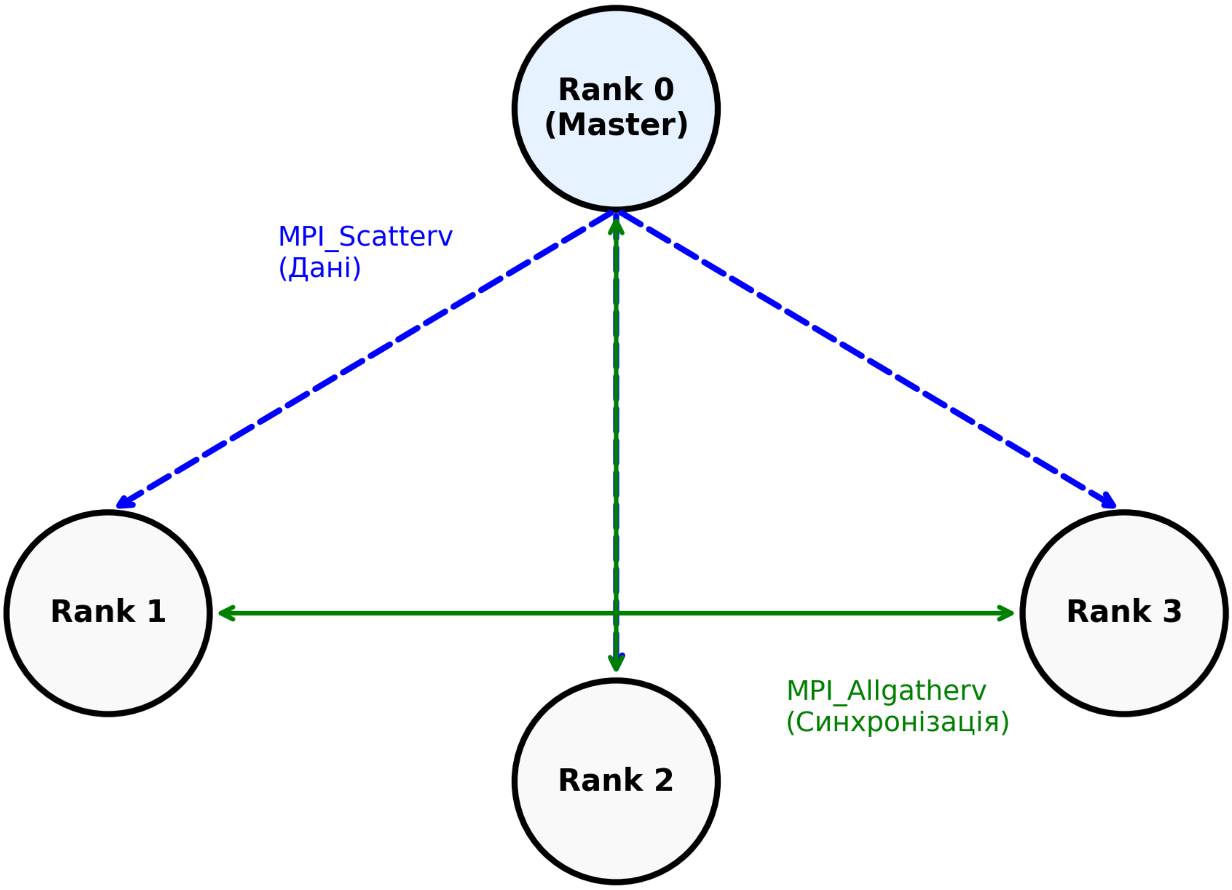


Рисунок 2.4 – Топологія колективних операцій передачі повідомлень

Основна обчислювальна робота зосереджена всередині ітераційного циклу, архітектура якого спроєктована таким чином, щоб максимізувати час корисної роботи процесора порівняно з часом очікування мережевих пакетів. На кожній ітерації кожен обчислювальний вузол виконує множення своєї локальної смуги матриці на актуальний вектор невідомих, отримуючи часткове оновлення результату. Проте для переходу до наступного кроку алгоритму необхідно забезпечити повну синхронізацію даних, оскільки кожен вузол володіє лише фрагментом вектора. Для цього в архітектуру інтегровано механізм глобального збору даних, де кожен процес розсилає свою оновлену частину розв’язку всім іншим учасникам. Це гарантує, що до початку наступної ітерації кожен обчислювач матиме у своїй локальній пам’яті повну та ідентичну копію вектора невідомих.

Оскільки математична модель вимагає множення кожного рядка матриці на повний вектор невідомих, виникає проблема просторової нестачі інформації. Після виконання локальних обчислень кожен вузол володіє лише власним оновленим фрагментом вектора розв'язків. Для переходу до наступного кроку алгоритму необхідно здійснити повну синхронізацію та відновлення цілісності цього вектора. Архітектура програмного забезпечення вирішує це завдання за допомогою колективної операції багатонаправленого збору даних. Під час виконання цієї комунікаційної фази кожен процес одночасно виступає і як відправник свого локального сегмента, і як одержувач сегментів від усіх інших вузлів мережі. Результатом виконання цієї операції є формування точної та ідентичної копії глобального вектора невідомих у виділеній оперативній пам'яті кожного окремого процесора. Тільки після успішного завершення обміну всіма фрагментами система отримує право перейти до наступної ітерації.

Контроль завершення обчислень також реалізується через механізм глобальної синхронізації. Кожен процес самостійно розраховує локальну суму квадратів нев’язки для своїх рядків, після чого ці значення об’єднуються за допомогою операції редукції [1, 24]. Результат цієї операції стає доступним усім процесам одночасно, що дозволяє їм синхронно прийняти рішення про вихід із циклу або продовження розрахунків. Така децентралізована логіка управління дозволяє уникнути зайвих повідомлень від керуючого вузла і позитивно впливає на масштабованість системи. Процедура перевірки збіжності алгоритму вимагає обчислення загальної норми вектора похибок. Архітектура паралельного додатка розділяє цю задачу на два послідовні етапи для уникнення мережевих перевантажень. На першому етапі кожен процес самостійно та без звернення до комунікаційного середовища розраховує локальну суму квадратів похибок виключно для призначеного йому діапазону рядків. На другому етапі отримані проміжні числа об'єднуються в єдине глобальне значення за допомогою колективної операції математичної редукції. Мережевий інтерфейс автоматично виконує додавання всіх надісланих локальних сум і повертає кінцевий результат кожному учаснику комунікації. Завдяки цьому всі обчислювальні вузли отримують доступ до актуального значення глобальної похибки одночасно. Така децентралізована логіка управління дозволяє кожному процесу самостійно перевірити математичну умову зупинки та синхронно прийняти рішення про вихід із циклу. Відсутність необхідності надсилати запити до головного вузла для отримання дозволу на завершення роботи суттєво зменшує обсяг службового трафіку.

Після отримання обчислень архітектура програмного забезпечення ініціює процедуру коректного завершення роботи паралельного середовища. Оскільки протягом усього ітераційного процесу глобальний вектор розв'язків перебував у фрагментованому стані між різними вузлами, система виконує фінальну колективну операцію збору даних на керуючому процесі. Головний координатор отримує всі локальні сегменти та формує єдиний цілісний масив результатів для подальшого експорту у файлову систему або передачі іншим інженерним модулям. Одразу після успішного збереження інформації кожен процес самостійно ініціює процедуру очищення оперативної пам'яті. Звільнення виділених адресних просторів для масивів значень, індексів та допоміжних векторів є обов'язковою вимогою для запобігання витокам пам'яті на серверах. Завершальним кроком є виклик функції деактивації комунікаційного середовища. Ця дія розриває мережеві з'єднання між вузлами, знищує глобальний комунікатор та безпечно завершує життєвий цикл усіх запущених потоків на кластері операційної системи.

2.3 Алгоритм балансування обчислювального навантаження

Забезпечення рівномірного розподілу обчислювального навантаження між усіма активними вузлами системи є основною проблемою під час розробки паралельних додатків для обробки розріджених структур даних. На відміну від щільних масивів, де обсяг роботи заздалегідь відомий і є сталим, розріджені матриці характеризуються високим ступенем просторової непередбачуваності. Топологія розміщення корисних даних у пам'яті комп'ютера повністю залежить від складності фізичної моделі об'єкта, що досліджується.

У контексті ітераційних методів розв'язання систем лінійних алгебраїчних рівнянь основний обсяг математичних операцій припадає на множення розрідженої матриці на щільний вектор [12, 36]. Особливістю виконання цієї дії у стисненому форматі є повне ігнорування порожніх ділянок. Відповідно, кількість арифметичних інструкцій для кожного окремого рівняння прямо пропорційна виключно кількості наявних ненульових елементів. В інженерній практиці, зокрема під час моделювання теплопровідності або розрахунку міцності неоднорідних матеріалів, розрахункові сітки завжди мають змінну густину. Області з високим градієнтом фізичних величин містять щільне скупчення вузлів, тоді як інші зони залишаються розрідженими. У результаті дискретизації формуються матриці із вкрай нерівномірним розподілом коефіцієнтів.

Застосування простого геометричного розбиття, за якого загальна кількість рядків алгоритмічно ділиться на рівні частини відповідно до кількості доступних процесорів, ігнорує описану специфіку інженерних даних. За такого підходу неминуче виникає ситуація значного обчислювального дисбалансу. Один обчислювальний процес може отримати діапазон рядків із мінімальною кількістю значень та завершити свої математичні розрахунки за частки мілісекунди. Водночас сусідній обчислювальний вузол отримає найбільш щільно заповнену область матриці і витратить на її опрацювання в десятки разів більше машинного часу.

Оскільки перехід до наступної ітерації алгоритму вимагає обов'язкового обміну даними та збирання цілісного вектора розв'язків, у системі утворюється вузьке місце на бар'єрі мережевої синхронізації. Швидші процесори змушені простоювати в очікуванні найповільнішого учасника обчислень. Цей час очікування є марним споживанням процесорних та енергетичних ресурсів кластера, що зводить нанівець переваги використання багатоядерної архітектури. Наочне порівняння неефективності класичного геометричного розбиття та оптимізованого розподілу за вагою обчислювальної складності представлене на рисунку 2.5.

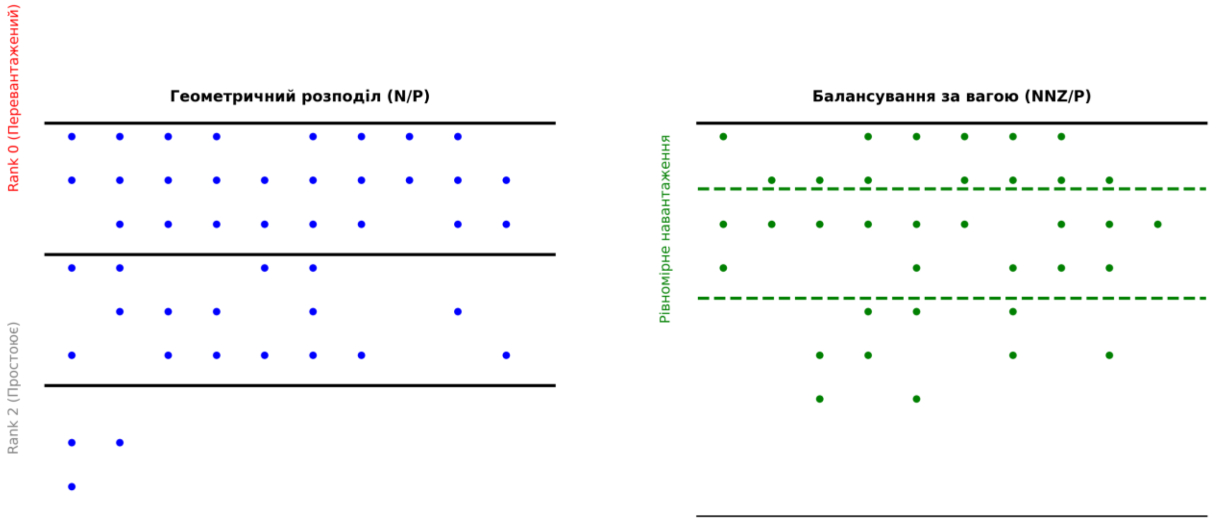


Рисунок 2.5 – Порівняння рівномірного та нерівномірного розподілу обчислювальної складності

Для усунення проблеми простоїв обчислювальних ядер реалізовано алгоритм статичного балансування навантаження. Цей механізм базується на попередньому аналізі ваги кожного окремого розрахункового рівняння. Замість використання фіксованого геометричного кроку декомпозиції [16, 37], логіка програми застосовує жадібну стратегію розподілу. Перед початком роздачі даних керуючий вузол розраховує ідеальне середнє навантаження на один процес. Цей показник обчислюється як частка від ділення загальної кількості ненульових елементів матриці на кількість активних учасників комунікатора.

Після отримання еталонного значення алгоритм розпочинає послідовне сканування масиву вказівників рядків. Програма акумулює кількість корисних елементів для кожного наступного рядка, створюючи наростаючу суму. Як тільки ця сума перетинає поріг ідеального середнього навантаження, алгоритм фіксує поточний індекс рядка як просторову межу для першого процесора. Після цього лічильник акумуляції скидається до нуля, і процес пошуку межі повторюється для наступного вузла. Такий підхід гарантує, що кожен процесор отримає у своє розпорядження такий набір суміжних рядків, сумарна обчислювальна складність яких буде максимально наближеною до показника.

Відхилення від балансу виникають виключно через неможливість розірвати один рядок між двома процесорами. Забезпечення цілісності рядка є алгоритмічною необхідністю, оскільки операція скалярного добутку вимагає одночасного доступу до всіх коефіцієнтів відповідного рівняння. Проте наявні похибки округлення залишаються мізерними на фоні загального обсягу інженерної матриці. Різниця у кілька десятків математичних операцій між різними потоками нівелюється апаратними механізмами мікропроцесора за частки мікросекунди і не створює відчутних затримок. З огляду на це, досягнуте вирівнювання часу виконання локальної операції множення для всіх учасників стає головною запорукою швидкого та ефективного проходження точок глобальної мережевої синхронізації. Логічна схема процесу визначення меж блоків даних на основі аналізу ваги елементів зображена на рисунку 2.6.

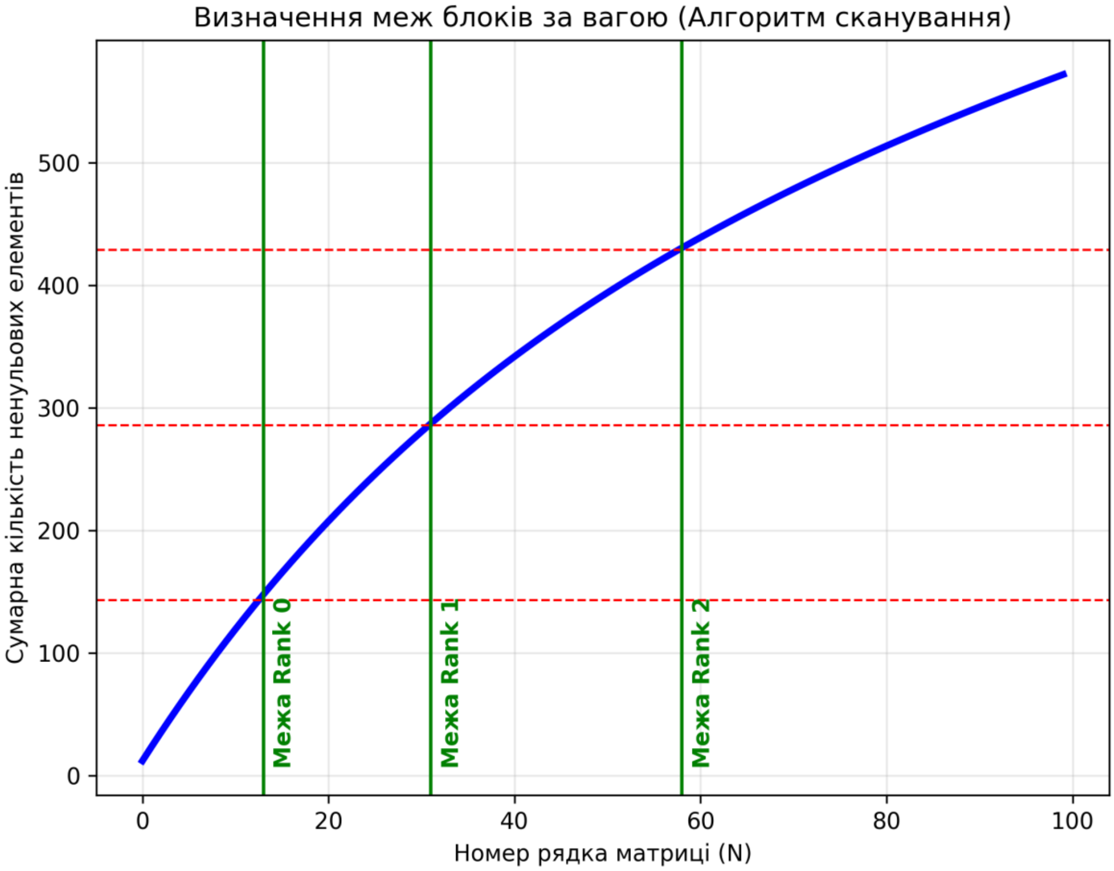


Рисунок 2.6 – Схема визначення меж блоків даних на основі ваги ненульових елементів

Наслідком впровадження алгоритму балансування є формування блоків інформації абсолютно різних розмірів. Один обчислювальний вузол може отримати десять тисяч щільно заповнених рядків, тоді як інший процесор отримає п'ятдесят тисяч майже порожніх рівнянь. Така неоднорідність вимагає від архітектури додатка створення допоміжних навігаційних масивів для правильної організації мережевого обміну. Стандартні комунікаційні функції не здатні самостійно визначити, куди саме у глобальний буфер необхідно записати отриманий фрагмент нестандартного розміру.

Для вирішення цієї проблеми на етапі ініціалізації генеруються два додаткові лінійні масиви. Перший масив зберігає точну кількість рядків, призначену кожному окремому процесу. Другий масив виконує функцію карти зміщень, вказуючи початкову позицію у загальному векторі, куди процесор повинен записати свої результати обчислень. Заповнення цих допоміжних структур здійснюється автоматично під час роботи алгоритму сканування меж. Передача цих масивів зміщень в якості аргументів до векторних функцій колективного обміну забезпечує безпомилкове збирання розкиданих сегментів у єдиний цілісний вектор. Залучення додаткової оперативної пам'яті для зберігання вказаної навігаційної інформації повністю компенсується можливістю гнучко та безпечно оперувати розрідженими структурами будь-якої складності.

Результатом впровадження такого алгоритму балансування є суттєве підвищення масштабованості програмного забезпечення. Усунення тривалих пауз очікування на бар’єрах синхронізації дозволяє системі демонструвати майже лінійне прискорення при збільшенні кількості обчислювальних ядер [32], що особливо помітно при розв’язанні задач надвеликої розмірності. Оптимізована архітектура розподілу даних мінімізує загальний час виконання ітераційного циклу, оскільки загальна продуктивність паралельного додатка перестає залежати від локальної щільності окремих фрагментів матриці. Таким чином, спроєктований механізм балансування навантаження виступає фундаментом для досягнення максимальної ефективності використання ресурсів сучасних кластерних систем та гарантує передбачувану поведінку алгоритму при роботі з розрідженими структурами довільної складності.

2.4 Висновки до другого розділу

У межах другого розділу було проведено комплексне проєктування архітектури паралельного програмного забезпечення для ефективного розв’язання систем лінійних алгебраїчних рівнянь надвеликої розмірності. Базовим вектором проєктування стало прагнення до мінімізації обсягу споживаної оперативної пам’яті на кожному вузлі та латентності мережевого обміну під час передачі повідомлень. Головним завданням цього етапу стала трансформація теоретичних математичних моделей у конкретні інженерні рішення, що адаптовані для виконання у високопродуктивному обчислювальному середовищі з розподіленою пам’яттю.

Головним рішенням, яке визначило структуру додатка, став вибір формату стисненого зберігання рядків як основної моделі представлення матриць. Спроєктована архітектура даних трансформує розріджену топологію у три лінійні одновимірні масиви: масив дійсних значень коефіцієнтів, масив цілочисельних індексів стовпців та вказівників рядків. Такий підхід дозволив знизити просторову складність алгоритму до лінійної залежності від фактичної кількості ненульових елементів та забезпечив високу швидкість виконання базової операції множення матриці на вектор. Використання формату CSR дозволяє повністю нівелювати надмірність даних, характерну для розріджених систем, що є важливим при роботі з розрахунковими сітками мільйонних розмірностей.

Архітектура комунікаційної взаємодії побудована за принципом SPMD на базі технології MPI. Було спроєктовано ієрархічну модель, де нульовий процес інкапсулює логіку роботи з файловою системою та ініціює розподіл даних. Завдяки використанню лінійних структур даних вдалося уникнути створення складних користувацьких типів MPI та витрат часу на пакування повідомлень. Для забезпечення ефективного обміну розроблено топологію на основі оптимізованих колективних операцій: розподіл даних відбувається за допомогою векторних функцій розсилки, синхронізація вектора невідомих – через операції збору «всі-до-всіх», а децентралізований контроль збіжності – шляхом глобальної редукції локальних похибок.

Окремим важливим етапом проєктування стала розробка динамічного алгоритму балансування обчислювального навантаження. З огляду на нерівномірне заповнення інженерних розріджених матриць, перед початком роздачі даних програма аналізує масив вказівників і визначає межі блоків для кожного процесора таким чином, щоб сумарна кількість арифметичних операцій була рівномірно розподілена по кластеру. Це гарантує одночасне завершення локальних обчислень усіма учасниками та усуває тривалі простої системи на бар’єрах синхронізації, що безпосередньо впливає на загальну пропускну здатність обчислювальної системи.

Додатково в межах другого розділу було визначено методологію майбутньої оцінки ефективності розробленого програмного засобу. Проєктування алгоритму включало врахування ключових метрик паралелізму, таких як коефіцієнт прискорення та показник ефективності використання ресурсів. Теоретичний аналіз спроєктованої моделі вказує на те, що завдяки оптимізованому балансуванню та використанню колективних операцій MPI, додаток повинен демонструвати високу масштабованість при збільшенні кількості обчислювальних вузлів. Такий підхід дозволяє заздалегідь окреслити межі продуктивності системи та підготувати надійну базу для проведення серії контрольних експериментів.

Підсумовуючи, можна констатувати, що розроблена архітектура повністю враховує як математичну природу ітераційних методів, так і апаратні особливості систем із розподіленою пам’яттю. Виконана інженерна підготовка утворює цілісний алгоритм, який є повністю готовим до етапу безпосередньої програмної реалізації та подальшого емпіричного тестування на реальних обчислювальних потужностях.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПАРАЛЕЛЬНОГО АЛГОРИТМУ РОЗВ'ЯЗАННЯ СЛАР

3.1 Опис апаратного забезпечення тестового обчислювального вузла

Процес розробки та впровадження високонавантажених обчислювальних систем невіддільний від глибокого розуміння цільової апаратної платформи. Ефективність функціонування будь-якого паралельного програмного забезпечення, зокрема масштабованих алгоритмів розв'язання систем лінійних алгебраїчних рівнянь надвеликої розмірності, критично залежить не лише від математичної досконалості самого алгоритму, а й від фізичних характеристик апаратного забезпечення, на якому він виконується.

На відміну від щільних матриць, обробка сильно розріджених структур генерує специфічне, вкрай нерівномірне навантаження на систему. Апаратна складність полягає у низькій арифметичній інтенсивності таких задач: на кожну корисну математичну операцію припадає необхідність зчитування кількох елементів пам'яті, причому доступ до глобального вектора невідомих відбувається за непередбачуваними, нелінійними адресами. Через це домінуючим обмежувальним фактором виступає не стільки пікова продуктивність арифметико-логічних пристроїв чи тактова частота процесора, скільки латентність та загальна пропускна здатність комунікаційних каналів і багаторівневої підсистеми пам'яті. У комп'ютерній інженерії цей феномен класифікується як проблема «стіни пам'яті», коли високошвидкісні обчислювальні конвеєри змушені простоювати сотні тактів в очікуванні надходження даних з повільніших модулів оперативної пам'яті.

Враховуючи високу обчислювальну складність поставленої задачі та необхідність опрацювання масивів даних обсягом у сотні мегабайт, тестування, відлагодження механізмів синхронізації та фінальний запуск розробленого MPI-додатка здійснювалися на базі високопродуктивного обчислювального вузла з архітектурою симетричної мультипроцесорності. Вибір саме такої архітектури є обґрунтованим кроком. SMP-системи характеризуються наявністю кількох рівноправних обчислювальних ядер, які розташовані на одному або кількох кристалах, працюють під управлінням єдиної операційної системи та мають спільний доступ до єдиного адресного простору оперативної пам'яті.

Дана апаратна конфігурація виступає в ролі класичного базового будівельного блоку сучасних суперкомп'ютерних кластерів. Запуск стандарту передачі повідомлень у межах одного фізичного SMP-вузла дозволяє максимально точно дослідити вплив внутрішньосистемних затримок та конкуренції потоків за спільну шину пам'яті, повністю ізолюючи ці показники від затримок зовнішніх мережевих комутаторів.

Для забезпечення коректної роботи паралельного алгоритму, швидкого дискового вводу-виводу та запобігання збоям через нестачу ресурсів, обчислювальний комплекс повинен відповідати чітким критеріям. У таблиці 3.1 наведено мінімальні технічні характеристики апаратного комплексу, необхідні для стабільної роботи розробленого середовища виконання.

Таблиця 3.1 – Комплексні технічні характеристики обчислювальної системи

	Апаратна підсистема	Мінімальні вимоги
1.	Центральний процесор	4 фізичні ядра, архітектура x86_64, підтримка інструкцій AVX/AVX2
2.	Оперативна пам'ять	8 ГБ, двоканальний режим, DDR4
3.	Дисковий накопичувач	Твердотільний накопичувач, мінімум 50 ГБ вільного простору

Кінець таблиці 3.1

4.	Комунікаційний інтерфейс	Системна шина PCIe 3.0 / Gigabit Ethernet
5.	Операційна система	Linux або Windows 10/11 із підсистемою WSL2

Узагальнена структурна схема мікроархітектури центрального процесора, яка демонструє взаємодію обчислювальних ядер із внутрішніми шинами обміну даними, наведена на рисунку 3.1.

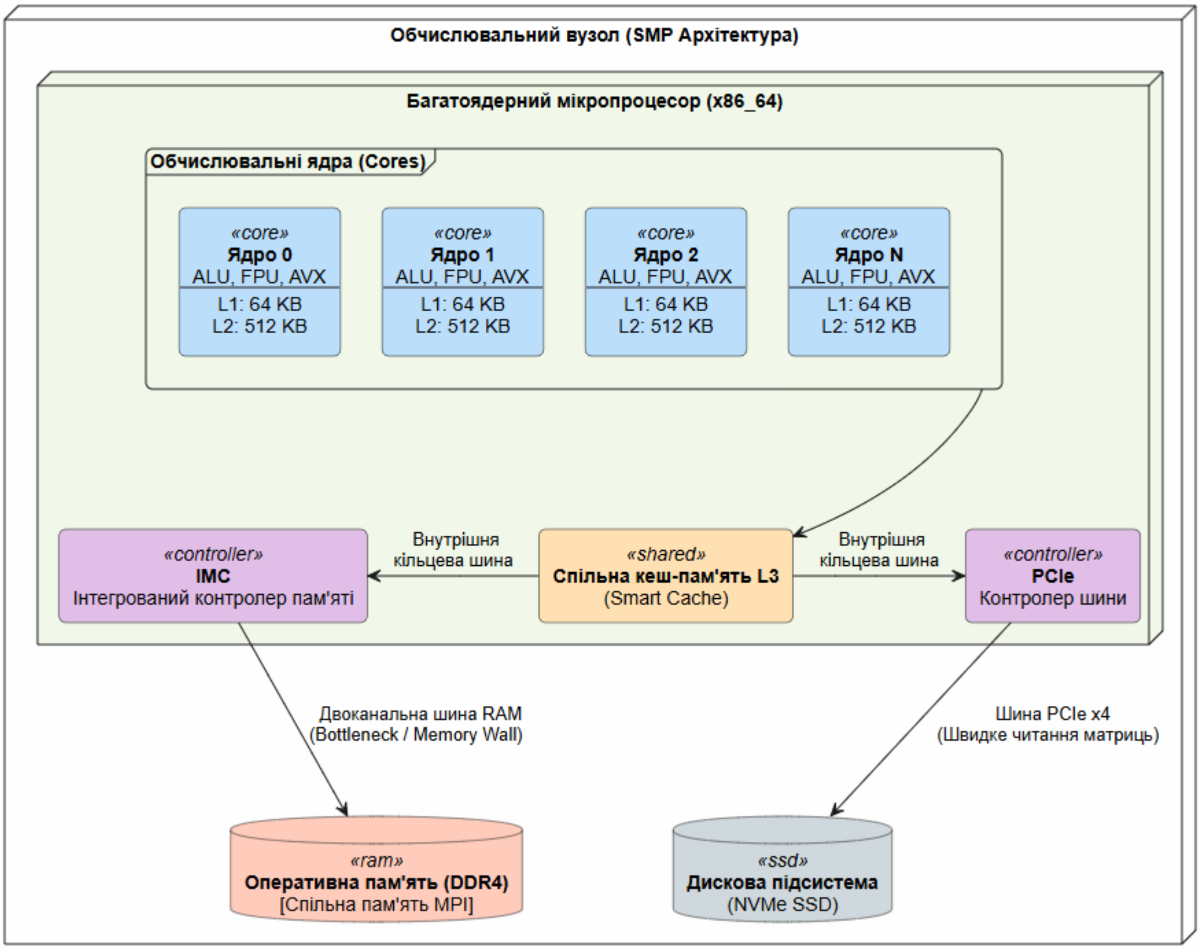


Рисунок 3.1 – Структурна схема мікроархітектури центрального процесора та внутрішніх шин обміну даними

3.1.1 Мікроархітектура обчислювальних ядер та конвеєризація інструкцій

Центральним елементом апаратного забезпечення обчислювального вузла є суперскалярний багатоядерний мікропроцесор, побудований за архітектурою x86_64. Кожне фізичне ядро процесора є складним обчислювальний механізм, здатний виконувати декілька інструкцій за один машинний такт завдяки наявності глибокого конвеєра виконання інструкцій та технології позачергового виконання.

У контексті розв'язання систем лінійних алгебраїчних рівнянь ітераційним методом Якобі, основне обчислювальне навантаження лягає на внутрішній цикл, який виконує множення локальної смуги розрідженої матриці у форматі CSR на глобальний вектор невідомих. На рівні мікроархітектури цей процес зводиться до безперервного потоку інструкцій завантаження даних з пам'яті, множення та додавання чисел з плаваючою комою.

Оскільки обробка формату CSR передбачає використання вкладених циклів, важливу роль відіграє апаратний блок передбачення розгалужень. Сучасні мікропроцесори використовують складні алгоритми нейронного передбачення для аналізу історії переходів. Завдяки тому, що кількість ненульових елементів у рядках інженерних матриць зазвичай є відносно стабільною, Branch Predictor з високою точністю вгадує моменти завершення внутрішнього циклу. Це дозволяє уникнути скидання конвеєра, що в іншому випадку призводило б до втрати 15-20 тактів процесорного часу на кожному хибному передбаченні, і забезпечує безперервне завантаження виконавчих пристроїв.

Окремим фактором продуктивності є використання спеціалізованих блоків для роботи з числами подвійної точності та розширених векторних інструкцій, таких як AVX2 або AVX-512. Використання компілятора GCC з агресивним рівнем оптимізації -O3 ініціює автоматичну векторизацію коду, намагаючись об'єднати кілька операцій множення-додавання в одну машинну інструкцію.

Проте застосування векторних інструкцій до формату CSR має базове апаратне обмеження, відоме як проблема непрямої адресації. Якщо масив значень зчитується лінійно і легко векторизується, то доступ до елементів вектора невідомих x відбувається хаотично, на основі індексів з масиву col_indices. На мікроархітектурному рівні це вимагає виконання дорогих векторних інструкцій збору. Процесор не може просто завантажити безперервний блок пам'яті у векторний регістр, а змушений ініціювати множинні запити до кешу для збирання розкиданих елементів.

Частково нівелювати затримки від непрямої адресації допомагає механізм позачергового виконання. Буфер переупорядкування інструкцій мікропроцесора здатний заглядати на десятки інструкцій вперед. Поки АЛП очікує надходження потрібного елемента вектора x з оперативної пам'яті або кешу L3, ядро процесора не простоює, а починає попередньо розраховувати незалежні інструкції з наступних ітерацій циклу. Така мікроархітектурна особливість дозволяє певною мірою «сховати» латентність підсистеми пам'яті та підвищити реальний показник IPC під час паралельної роботи MPI-процесів на одному кристалі.

3.1.2 Ієрархія кеш-пам'яті та проблема когерентності

Важливим апаратним вузлом при опрацюванні надвеликих розріджених матриць є підсистема кешування пам'яті. Оскільки швидкість доступу до регістрів процесора становить частки наносекунди, а звернення до оперативної пам'яті вимагає 60-100 наносекунд, багаторівнева ієрархія кешів виступає головним демпфером продуктивності обчислювальної системи.

Кеш першого та другого рівнів є приватними для кожного фізичного ядра. Кеш L1 зазвичай розділений на незалежні області для інструкцій та даних. Завдяки тому, що масиви значень та індексів стовпців матриці у форматі CSR розташовані у пам'яті строго лінійно, вони володіють дуже високим ступенем просторової локальності. Це дозволяє апаратному префетчеру процесора легко розпізнавати патерн послідовного читання і завчасно підвантажувати наступні блоки даних (кеш-лінії, зазвичай розміром по 64 байти) з повільнішого L2 та L3 у надшвидкий кеш L1 ще до того, як вони знадобляться обчислювальному конвеєру. Порівняльні характеристики рівнів ієрархії пам'яті та їхні показники латентності наведено у таблиці 3.2.

Таблиця 3.2 – Характеристики рівнів ієрархії пам'яті обчислювального вузла

Рівень пам'яті	Типовий обсяг	Затримка доступу	Пропускна здатність
Регістри CPU	< 1 KB	1 такт (< 0.5 нс)	> 1000 GB/s
Кеш L1 (SRAM)	32 - 64 KB	3 - 4 такти (~ 1 нс)	~ 200 - 300 GB/s
Кеш L2 (SRAM)	256 - 512 KB	10 - 15 тактів (~ 3-5 нс)	~ 100 - 200 GB/s
Кеш L3 (SRAM)	8 - 64 MB	40 - 70 тактів (~ 10-20 нс)	~ 50 - 100 GB/s
RAM (DRAM)	8 - 128 GB	> 200 тактів (60 - 100 нс)	20 - 60 GB/s

Проте ефективність ієрархії кешування різко падає під час операції множення при зверненні до глобального вектора невідомих x. Оскільки доступ до його елементів здійснюється хаотично і непередбачувано, на основі значень з масиву col_indices, просторова та часова локальність майже повністю відсутні. Якщо розмірність системи рівнянь настільки велика, що вектор x фізично не поміщається у спільному кеші третього рівня, виникає явище масових промахів кешу. Кожен такий промах змушує процесор призупиняти виконання корисних інструкцій і очікувати на надходження даних безпосередньо з мікросхем оперативної пам'яті DRAM. При використанні стандарту MPI на багатоядерній SMP-системі виникає специфічна взаємодія потоків на мікроархітектурному рівні. Хоча з програмної точки зору кожен MPI-процес має власну ізольовану пам'ять, фізично всі обчислювальні ядра знаходяться на одному кристалі та ділять спільний кеш L3. Під час виконання комунікаційних операцій (наприклад, збору вектора через MPI_Allgatherv), бібліотека MPI для оптимізації пересилань у межах одного вузла автоматично використовує механізми спільної пам'яті операційної системи. Це інтенсивно навантажує апаратні протоколи когерентності кешів, найвідомішим з яких є протокол MESI. Апаратні контролери змушені постійно відстежувати актуальність даних у кеш-лініях кожного ядра. Якщо одне ядро записує оновлені дані у свій сегмент вектора розв'язків, а інше ядро одночасно намагається зчитати сусідні елементи, які потрапили в ту ж саму 64-байтну лінію кешу, виникає негативне явище хибного розділення. Протокол когерентності змушений визнавати цю лінію недійсною для інших ядер, що генерує значний паразитний трафік всередині внутрішньої кільцевої шини процесора. Це додатково споживає пропускну здатність системи та знижує лінійність масштабування паралельного алгоритму при збільшенні кількості задіяних ядер. Схематичну топологію доступу багатоядерного процесора до спільного кешу третього рівня та оперативної пам'яті зображено на рисунку 3.2.

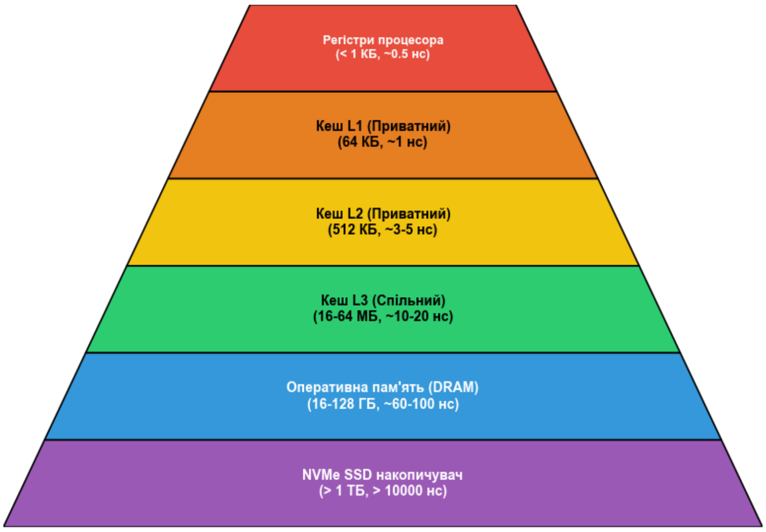


Рисунок 3.2 – Топологія доступу багатоядерного процесора до спільного кешу L3 та оперативної пам'яті

3.1.3 Пропускна здатність системної шини

Основним апаратним вузьким місцем розробленого обчислювального комплексу є інтегрований контролер пам'яті та системна шина доступу до динамічної оперативної пам'яті. У той час як тактові частоти та кількість обчислювальних ядер мікропроцесорів зростали експоненціально протягом останніх десятиліть, швидкість передачі даних від оперативної пам'яті до кристала процесора масштабувалася значно повільніше. У сучасних обчислювальних вузлах пам'ять стандарту DDR4 або DDR5 зазвичай працює у двоканальному або чотириканальному (Quad Channel) режимах. Наприклад, для модулів пам'яті DDR4 із частотою 3200 МГц теоретична пікова пропускна здатність у двоканальному режимі становить приблизно 51,2 ГБ/с. Проте на практиці корисна пропускна здатність є суттєво нижчою через накладні витрати на адресацію, регенерацію пам'яті та затримки CAS Latency. Проблема обмеженої пропускної здатності стає критичною під час виконання розробленого паралельного додатка. Під час розв'язання системи лінійних алгебраїчних рівнянь надвеликої розмірності (наприклад, N = 10^8), глобальний вектор невідомих формату подвійної точності займає близько 800 мегабайт оперативної пам'яті. Наприкінці кожної ітерації методу Якобі алгоритм викликає колективну функцію синхронізації MPI_Allgatherv. У цей момент усі задіяні логічні ядра одночасно намагаються оновити свої сегменти пам'яті та зчитати дані від інших процесів. Це генерує шквальний, конкурентний запит до єдиного контролера пам'яті. Оскільки сумарна потреба обчислювальних ядер у даних багаторазово перевищує максимальну пропускну здатність системної шини, утворюється апаратна черга транзакцій. Конвеєри мікропроцесора швидко вичерпують інструкції, готові до виконання, буфери завантаження/збереження переповнюються, і обчислювальні ядра змушені переходити у стан простою. У комп'ютерній інженерії ця проблема відома як «Стіна пам'яті». Саме це апаратне обмеження пояснює поведінку паралельного алгоритму на SMP-системах: при додаванні нових обчислювальних процесів (до 4 або 8) арифметична потужність системи формально зростає, проте реальний час виконання програми може погіршуватися. Проблема криється не в математичній моделі чи програмному коді, а у фізичній нездатності мікросхем оперативної пам'яті вчасно постачати необхідні операнди через системну шину у регістри високошвидкісних процесорних ядер.

3.1.4 Підсистема дискового вводу-виводу

Для забезпечення безперебійної роботи алгоритму на початкових етапах життєвого циклу програми важливою є підсистема постійного зберігання даних. Текстові файли з матрицями надвеликої розмірності можуть досягати обсягів у десятки гігабайт. Використання класичних жорстких дисків з магнітними пластинами є неприпустимим через їхню вкрай низьку швидкість довільного доступу та лінійну швидкість читання на рівні 100-150 МБ/с.

Для усунення затримок на етапі ініціалізації обчислювальний вузол оснащено твердотільним накопичувачем стандарту NVMe, який підключений безпосередньо до ліній PCI Express центрального процесора. Це дозволяє зчитувати масивні розрахункові файли зі швидкістю понад 2-3 ГБ/с, що гарантує максимально швидке завантаження структур CSR у віртуальну пам'ять процесів та швидкий перехід до етапу математичних обчислень.

3.2 Програмна реалізація паралельного алгоритму та середовища передачі повідомлень

Процес програмної реалізації спроєктованого паралельного алгоритму здійснювався мовою програмування C++ із використанням стандарту передачі повідомлень MPI. Вибір мови C++ зумовлений її високою продуктивністю, строгим контролем типів та можливістю низькорівневого управління пам'яттю. Для задач, які оперують структурами даних обсягом у сотні мегабайт, прямий контроль над виділенням та звільненням оперативної пам'яті дозволяє уникнути непередбачуваної роботи автоматичних збирачів сміття та гарантує стабільність обчислювального процесу. Для компіляції коду застосовувався компілятор GCC із максимальним рівнем оптимізації, що забезпечило розгортання циклів та автоматичну векторизацію арифметичних операцій.

Архітектура розробленого програмного забезпечення базується на парадигмі єдиної програми та множини даних. Такий підхід передбачає, що ідентичний виконуваний файл компілюється лише один раз, після чого системний планувальник завдань одночасно розгортає його копії на всіх виділених обчислювальних ядрах кластера. Кожен запущений екземпляр функціонує як незалежний процес операційної системи із власним захищеним віртуальним адресним простором, що гарантує ізольованість даних та унеможливлює несанкціоноване перекриття сегментів пам'яті під час паралельних обчислень.

Життєвий цикл програми розпочинається зі строго синхронізованого етапу ініціалізації середовища передачі повідомлень. На цій стадії системні бібліотеки автоматично конфігурують доступні апаратні інтерфейси, виділяють внутрішні буфери для мережевого обміну та формують глобальний комунікатор, який об'єднує всі задіяні процеси у єдину логічну топологію. Одразу після успішної ініціалізації кожен обчислювальний потік звертається до середовища для отримання двох важливих параметрів: загальної кількості активних процесів у системі та свого ідентифікатора. Значення загальної кількості є базовою константою для алгоритму балансування навантаження, оскільки саме пропорційно до цього числа виконується декомпозиція глобальної матриці.

Водночас ранг слугує головним алгоритмічним розгалужувачем усередині ідентичного вихідного коду. Спираючись на нього, кожен процес набуває просторової обізнаності та математично вираховує точні координати власного діапазону рядків, який він повинен обробити. Крім того, механізм рангів дозволяє логічно виділити головний керуючий вузол. Цей вузол, паралельно з виконанням своєї частки математичних розрахунків, бере на себе додаткові адміністративні функції: фіксацію часу початку та завершення роботи, збір підсумкової статистики похибок та виведення фінальних результатів на екран термінала. Узагальнену блок-схему життєвого циклу розробленого паралельного додатка наведено на рисунку 3.3.

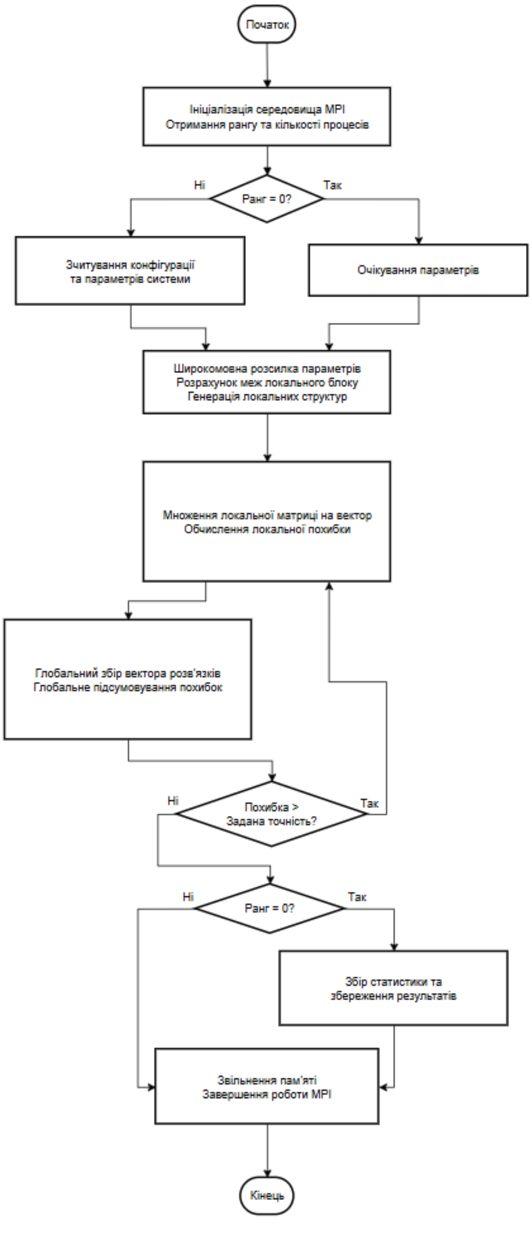


Рисунок 3.3 – Блок-схема алгоритму роботи паралельного додатка

Зважаючи на архітектурні обмеження дискового вводу-виводу при роботі з матрицями надвеликої розмірності, алгоритм генерації значень матриці формату стисненого зберігання рядків було інтегровано безпосередньо у пам'ять кожного окремого обчислювального вузла. Кожен процес, спираючись на свій ранг та застосовуючи алгоритм балансування навантаження, обчислює координати власного діапазону рядків. Після цього він автономно генерує свій локальний блок матриці коефіцієнтів та відповідний сегмент вектора вільних членів. Такий підхід дозволив повністю виключити необхідність пересилання гігантських масивів даних через системну шину на етапі запуску програми.

Ядром програмної реалізації виступає головний обчислювальний блок, який інкапсулює ітераційний цикл методу Якобі. З алгоритмічної точки зору, цей етап реалізовано у вигляді системи вкладених циклів, оптимізованих для роботи з розрідженими структурами. Зовнішній цикл відповідає за послідовний перебір призначеного процесору діапазону рядків.

Для обробки кожного такого рядка програма звертається до масиву вказівників, щоб миттєво визначити точні координати початку та кінця корисних даних у пам'яті. Після цього запускається внутрішній арифметичний цикл, який ітерує виключно по ненульових елементах, використовуючи масив значень та масив індексів стовпців. Під час цього проходу процесор обчислює суму добутків коефіцієнтів матриці на відповідні компоненти актуальної копії глобального вектора невідомих, який був сформований на попередньому кроці. Важливою архітектурною особливістю цієї реалізації є програмна ізоляція діагонального елемента матриці: під час проходу по рядку програма ігнорує його у загальній сумі, оскільки він використовується на фінальній стадії для ділення різниці між вектором вільних членів та накопиченою сумою.

Результатом виконання цієї послідовності математичних операцій є формування оновленого локального сегмента вектора розв'язків. Цей новий сегмент суворо записується в окремий виділений буфер пам'яті, що гарантує збереження цілісності вихідних даних до повного завершення ітерації всіма обчислювальними потоками.

Для переходу до наступної ітерації програма використовує механізм глобальної синхронізації. Оскільки кількість оброблюваних рядків на різних процесах може відрізнятися через балансування навантаження, у коді застосовано векторну версію колективної операції збору даних. Перед її викликом кожен процес заповнює спеціальні масиви зміщень та кількостей елементів. Функція збирає оновлені фрагменти розв'язку від кожного активного ядра та формує цілісний, ідентичний вектор невідомих у пам'яті кожного процесу. Схему інформаційної взаємодії обчислювальних вузлів під час ітераційного циклу зображено на рисунку 3.4.

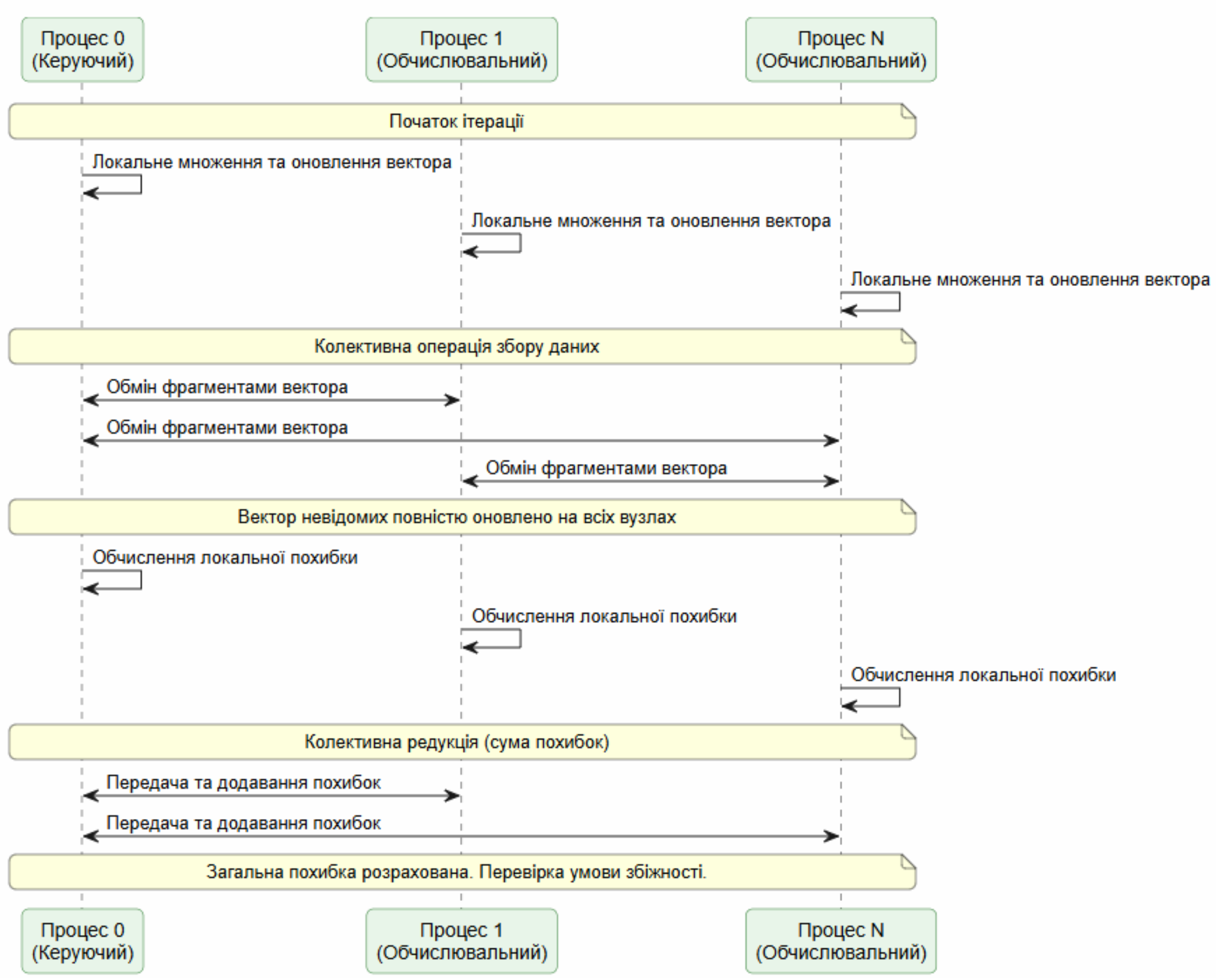


Рисунок 3.4 – Діаграма послідовності мережевого обміну під час ітераційного циклу

Паралельно з оновленням вектора реалізовано програмний блок контролю збіжності алгоритму. Кожен процес обчислює локальну суму квадратів нев'язки для своєї зони відповідальності. Для отримання глобального значення похибки викликається функція редукції з математичним оператором додавання. Ця команда консолідує локальні похибки з усіх вузлів і повертає загальну суму кожному процесу. Якщо добуте значення стає меншим за заздалегідь визначений поріг точності, процеси синхронно здійснюють вихід із циклу.

Завершується робота програми звільненням усієї динамічно виділеної пам'яті, збереженням статистичних даних про час виконання та коректним закриттям комунікаційного середовища.

3.3 Організація процесів зберігання та дискового вводу-виводу великих обсягів даних

У високопродуктивних обчисленнях операції введення-виведення часто стають головним фактором, що стримує загальну швидкодію системи. При переході до систем лінійних алгебраїчних рівнянь надвеликої розмірності проблема зберігання та завантаження початкових даних набуває критичного значення. Традиційний підхід, за якого початкова матриця зберігається у звичайних текстових файлах, виявляється неефективним. Текстовий формат вимагає перетворення кожного символу у числове значення з плаваючою комою під час зчитування, що непропорційно навантажує центральний процесор. Крім того, текстове представлення займає значно більше дискового простору порівняно з бінарним кодуванням.

Початкова архітектура розробленого програмного забезпечення передбачала централізоване зчитування розрахункових даних. За цією схемою головний керуючий вузол зчитував усю структуру розрідженої матриці з твердотільного накопичувача у власну оперативну пам'ять, після чого розподіляв необхідні фрагменти між іншими обчислювальними процесами за допомогою комунікаційних функцій розсилки. Цей підхід демонстрував прийнятну продуктивність на матрицях середнього розміру.

Однак під час спроб масштабування задачі до ста мільйонів рівнянь виникло одразу дві нездоланні перешкоди. По-перше, розмір файлу з такою матрицею навіть у стисненому форматі сягав десятків гігабайт, що робило процес зчитування неприпустимо довгим. По-друге, спроба завантажити такий масив інформації в оперативну пам'ять одного вузла неминуче призводила до переповнення пам'яті та аварійного завершення роботи програми операційною системою ще на етапі ініціалізації.

Для подолання проблеми дискового введення-виведення було прийнято стратегічне інженерне рішення про повну відмову від зовнішніх файлів для систем надвеликої розмірності. Замість зчитування готової матриці з диска, програмне забезпечення використовує підхід процедурної паралельної генерації даних безпосередньо в оперативній пам'яті. Відповідно до розробленого алгоритму балансування, кожен процес отримує власні координати початку та кінця блоку рядків. Спираючись на ці координати, обчислювальне ядро самостійно формує потрібні ділянки масивів значень, індексів стовпців та вказівників рядків.

Така реорганізація процесів роботи з даними дозволила повністю усунути дискову підсистему з критичного шляху виконання програми. Час, який раніше витрачався на повільне зчитування гігабайтних файлів та їх передачу через системну шину, було зведено до кількох мілісекунд, необхідних для роботи алгоритму процедурної генерації.

Найважливішим наслідком цього архітектурного рішення стало повне усунення загрози переповнення оперативної пам'яті головного вузла. Глобальна матриця на сто мільйонів рівнянь ніколи не існує в пам'яті комп'ютера як єдиний цілісний об'єкт. Від самого початку запуску програми вона формується у вигляді незалежних, ізольованих локальних сегментів, що рівномірно розподілені між усіма активними процесами. Це дозволило ефективно обійти жорсткі апаратні обмеження та забезпечити стабільну роботу ітераційного алгоритму на надвеликих масивах даних.

Візуальне порівняння класичного підходу централізованого зчитування файлів, який призводить до апаратного переповнення, та розробленої оптимізованої архітектури паралельної генерації даних безпосередньо в оперативній пам'яті вузлів наведено на рисунку 3.5.

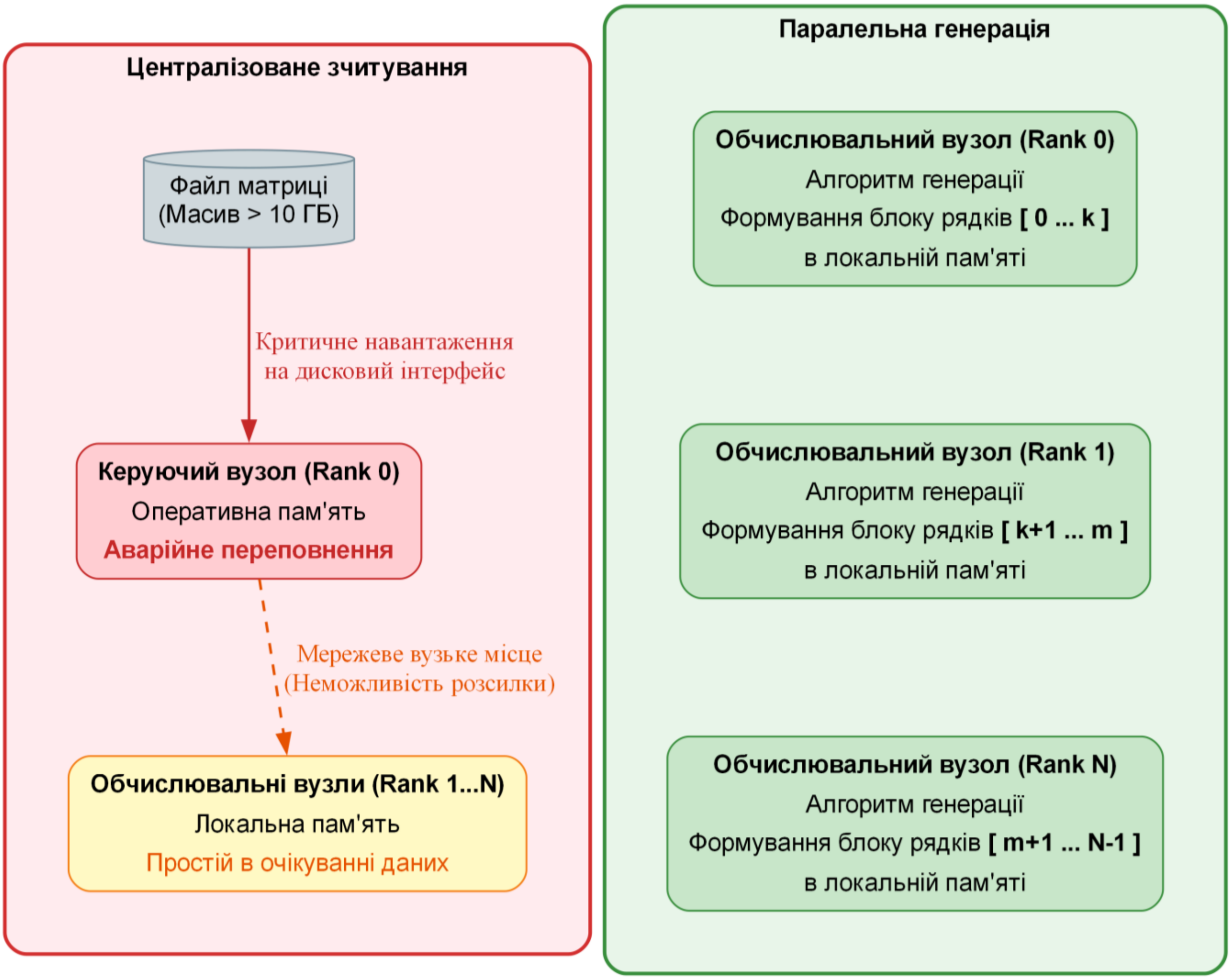
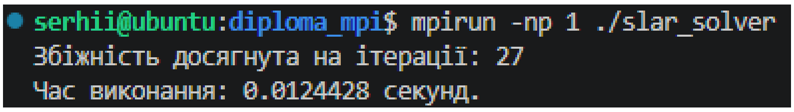


Рисунок 3.5 – Порівняння архітектурних підходів до ініціалізації надвеликих масивів даних

3.4 Апаратно-програмне тестування та аналіз ефективності системи

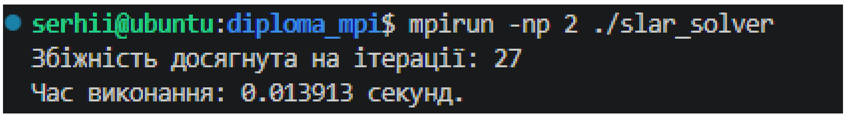
Метою етапу апаратно-програмного тестування є перевірка коректності функціонування розробленого паралельного додатка, оцінка його стабільності в умовах пікових навантажень та практичне дослідження метрик паралельної ефективності. Для виявлення меж масштабованості розробленого алгоритму та аналізу його апаратної поведінки за різних сценаріїв було спроєктовано комплексну багаторівневу методологію тестування.

Тестові запуски програмного забезпечення здійснювалися через інтерфейс командного рядка операційної системи на базі ядра Linux із використанням базової команди ініціалізації середовища передачі повідомлень. Для вичерпної фіксації результатів обчислювального експерименту та підтвердження стабільності роботи системи без аварійного завершення у контрольних запусках, процес тестування було задокументовано. Для кожної з трьох розмірностей матриці зафіксовано результати паралельного виконання на одному, двох, чотирьох та восьми обчислювальних ядрах. Серію з дванадцяти відповідних знімків екрана термінала, які демонструють успішне досягнення збіжності алгоритму та розрахований час виконання для кожної конфігурації, наведено на рисунках 3.6-3.17.



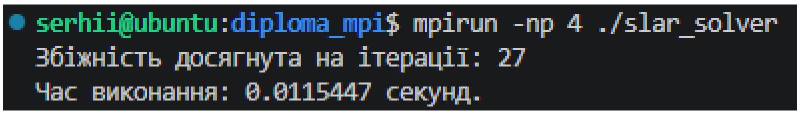
```
serhii@ubuntu:diploma_mpi$ mpirun -np 1 ./slar_solver
Збіжність досягнута на ітерації: 27
Час виконання: 0.0124428 секунд.
```

Рисунок 3.6 – Результат виконання алгоритму для системи малої розмірності на одному обчислювальному ядрі



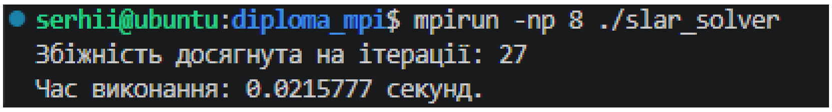
```
serhii@ubuntu:diploma_mpi$ mpirun -np 2 ./slar_solver
Збіжність досягнута на ітерації: 27
Час виконання: 0.013913 секунд.
```

Рисунок 3.7 – Результат виконання алгоритму для системи малої розмірності на двох обчислювальних ядрах



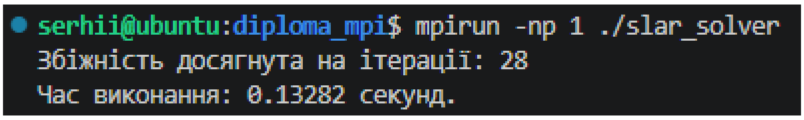
```
serhii@ubuntu:diploma_mpi$ mpirun -np 4 ./slar_solver
Збіжність досягнута на ітерації: 27
Час виконання: 0.0115447 секунд.
```

Рисунок 3.8 – Результат виконання алгоритму для системи малої розмірності на чотирьох обчислювальних ядрах



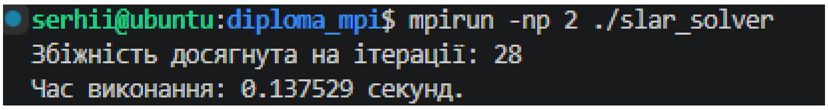
```
serhii@ubuntu:diploma_mpi$ mpirun -np 8 ./slar_solver
Збіжність досягнута на ітерації: 27
Час виконання: 0.0215777 секунд.
```

Рисунок 3.9 – Результат виконання алгоритму для системи малої розмірності на восьми обчислювальних ядрах



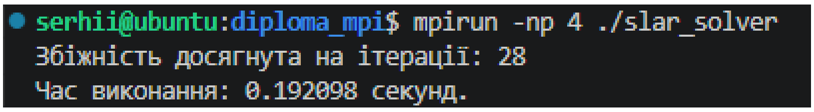
```
serhii@ubuntu:diploma_mpi$ mpirun -np 1 ./slar_solver
Збіжність досягнута на ітерації: 28
Час виконання: 0.13282 секунд.
```

Рисунок 3.10 – Результат виконання алгоритму для системи середньої розмірності на одному обчислювальному ядрі



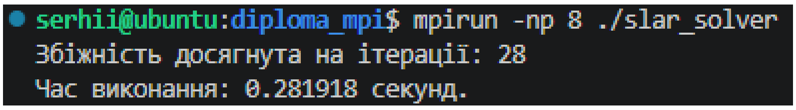
```
serhii@ubuntu:diploma_mpi$ mpirun -np 2 ./slar_solver
Збіжність досягнута на ітерації: 28
Час виконання: 0.137529 секунд.
```

Рисунок 3.11 – Результат виконання алгоритму для системи середньої розмірності на двох обчислювальних ядрах



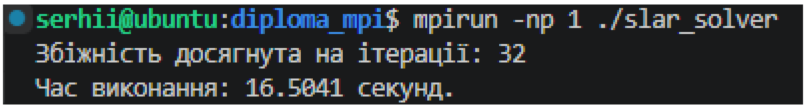
```
serhii@ubuntu:diploma_mpi$ mpirun -np 4 ./slar_solver
Збіжність досягнута на ітерації: 28
Час виконання: 0.192098 секунд.
```

Рисунок 3.12 – Результат виконання алгоритму для системи середньої розмірності на чотирьох обчислювальних ядрах



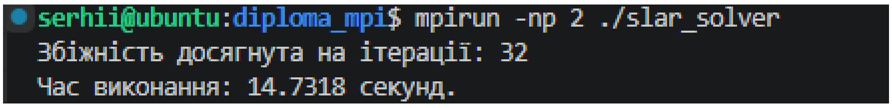
```
serhii@ubuntu:diploma_mpi$ mpirun -np 8 ./slar_solver
Збіжність досягнута на ітерації: 28
Час виконання: 0.281918 секунд.
```

Рисунок 3.13 – Результат виконання алгоритму для системи середньої розмірності на восьми обчислювальних ядрах



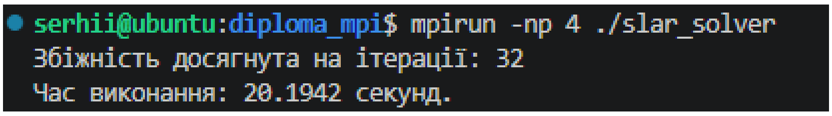
```
serhii@ubuntu:diploma_mpi$ mpirun -np 1 ./slar_solver
Збіжність досягнута на ітерації: 32
Час виконання: 16.5041 секунд.
```

Рисунок 3.14 – Результат виконання алгоритму для системи надвеликої на одному обчислювальному ядрі



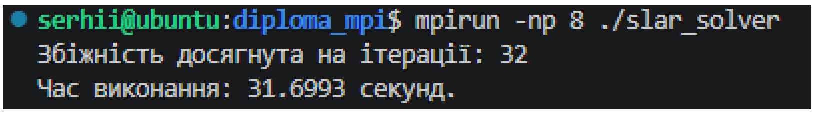
```
serhii@ubuntu:diploma_mpi$ mpirun -np 2 ./slar_solver
Збіжність досягнута на ітерації: 32
Час виконання: 14.7318 секунд.
```

Рисунок 3.15 – Результат виконання алгоритму для системи надвеликої розмірності на двох обчислювальних ядрах



```
serhii@ubuntu:diploma_mpi$ mpirun -np 4 ./slar_solver
Збіжність досягнута на ітерації: 32
Час виконання: 20.1942 секунд.
```

Рисунок 3.16 – Результат виконання алгоритму для системи надвеликої розмірності на чотирьох обчислювальних ядрах



```
serhii@ubuntu:diploma_mpi$ mpirun -np 8 ./slar_solver
Збіжність досягнута на ітерації: 32
Час виконання: 31.6993 секунд.
```

Рисунок 3.17 – Результат виконання алгоритму для системи надвеликої розмірності на восьми обчислювальних ядрах

Для фахової оцінки результатів розпаралелювання застосовувалися дві базові метрики: коефіцієнт прискорення та ефективність використання ресурсів. Коефіцієнт прискорення обчислюється за формулою 3.1:

$$S_p = \frac{T_1}{T_P} \qquad (3.1)$$

де T_1 – час виконання алгоритму на одному обчислювальному ядрі;

T_p – час виконання алгоритму на *p* обчислювальних ядрах.

Показник ефективності використання ресурсів дозволяє оцінити частку корисного часу роботи процесорів і розраховується за формулою 3.2.

$$E_p = \frac{S_p}{p} \times 100\% \qquad (3.2)$$

Для мінімізації впливу фонових процесів операційної системи та випадкових апаратних затримок кожен експеримент виконувався п'ять разів, а до підсумкового протоколу заносилося усереднене значення часу. Зведені результати комплексного апаратно-програмного тестування та обчислені метрики продуктивності представлено у таблиці 3.3.

Таблиця 3.3 – Аналіз продуктивності паралельного алгоритму для різних розмірностей матриці

Розмірність системи	Кількість ядер (p)	Час виконання (с)	Коефіцієнт прискорення (S_p)	Ефективність (E_p)
Мала 10^5	1	0.0124	1.00	100%
	2	0.0139	0.89	44.5%
	4	0.0115	1.08	27.0%
	8	0.0215	0.58	7.3%
Середня 10^6	1	0.1328	1.00	100%
	2	0.1375	0.97	48.5%
	4	0.1920	0.69	17.3%
	8	0.2819	0.47	5.9%
Велика 10^8	1	16.5041	1.00	100%
	2	14.7318	1.12	56.0%
	4	20.1942	0.82	20.5%
	8	31.6993	0.52	6.5%

Розгорнутий аналіз експериментальних даних дозволяє чітко простежити нелінійну динаміку поведінки обчислювальної системи залежно від обсягу оброблюваної інформації. Зокрема, під час розв'язання найменшої системи на сто тисяч рівнянь фіксується явище помітного падіння продуктивності. Замість очікуваного прискорення, залучення нових обчислювальних ядер призводить до стрімкого збільшення часу виконання: алгоритм на восьми потоках працює значно повільніше, ніж на одному. З погляду системної інженерії, це є класичним прикладом порушення балансу між корисними обчисленнями та комунікацією. Обсяг математичних операцій є настільки мізерним, що процесор виконує їх за частки мілісекунди, тоді як домінуючим фактором стають накладні витрати на ініціалізацію мережевих повідомлень та глобальну синхронізацію. Цей тест показує, що застосування складних паралельних технологій для задач тривіальної розмірності є недоцільним.

При збільшенні розмірності до одного мільйона рівнянь загальний обсяг математичних розрахунків пропорційно зростає, проте розроблений алгоритм все ще не досягає позитивного прискорення. Хоча локальні сегменти матриці успішно поміщаються у швидкісних кешах другого та третього рівнів, і процесорні ядра виконують множення надзвичайно ефективно, цього обсягу роботи виявляється недостатньо для повного перекриття комунікаційних затримок. Колективні операції обміну масивами та зведення похибок створюють відчутну програмно-апаратну затримку. Цей етап тестування яскраво ілюструє закон Амдала, згідно з яким швидкість виконання паралельної програми суворо обмежується її послідовною частиною та часом, необхідним на передачу даних між обчислювальними вузлами.

Кардинально інша картина розгортається під час ключового експерименту із системою на сто мільйонів рівнянь. Перехід від послідовного алгоритму до двох ядер дає перше реальне прискорення, проте подальша спроба залучити чотири та вісім ядер знову призводить до падіння продуктивності з двократним зростанням часу виконання. У цьому випадку імовірною причиною є обмеження пропускної здатності підсистеми пам'яті. Глобальний вектор невідомих займає сотні мегабайт оперативної пам'яті, і коли вісім ядер одночасно викликають функцію глобальної синхронізації, вони створюють значне конкурентне навантаження до інтегрованого контролера. Пропускна здатність системної шини швидко стає обмежувальним фактором, утворюється значна апаратна черга транзакцій, а масові промахи у буфері асоціативної трансляції адрес додатково гальмують роботу. У результаті високопродуктивні суперскалярні конвеєри процесора змушені простоювати сотні тактів в очікуванні даних, перетворюючи надлишкову обчислювальну потужність на марну витрату часу.

Підсумовуючи результати апаратно-програмного тестування, можна констатувати, що розроблений алгоритм коректно працює у контрольних запусках, забезпечуючи надійну роботу без аварійного завершення у контрольних запусках та програмних збоїв. Проте його реальна продуктивність має нелінійну залежність від обсягу оброблюваних даних та пропускної здатності системної шини. Отримані метрики показують, що для задач з низькою арифметичною інтенсивністю масштабування в межах одного фізичного обчислювального вузла є неефективним. Для перевірки масштабування на таких масивах потрібне окреме тестування на багатовузловому розподіленому кластері, де кожен багатоядерний вузол матиме власну незалежну оперативну пам'ять, а обмін даними здійснюватиметься через спеціалізовані високошвидкісні оптичні мережі.

3.5 Висновки до третього розділу

У третьому розділі було здійснено успішну практичну реалізацію спроєктованого паралельного алгоритму та проведено його комплексне апаратно-програмне тестування. Процес розробки спирався на глибокий аналіз мікроархітектури сучасних обчислювальних ядер, ієрархії кеш-пам'яті та пропускної здатності системної шини, що дозволило максимально адаптувати програмний код до фізичних особливостей обладнання.

Використання стандарту передачі повідомлень у середовищі мови C++ забезпечило створення надійної архітектури, здатної обробляти надвеликі масиви даних. Важливим інженерним рішенням стала відмова від централізованого дискового зчитування початкових файлів на користь паралельної процедурної генерації розріджених матриць безпосередньо в оперативній пам'яті ізольованих обчислювальних процесів. Це дозволило усунути головне вузьке місце підсистеми вводу-виводу та гарантувати стабільну ініціалізацію системи без ризику апаратного переповнення пам'яті.

Проведене багаторівневе навантажувальне тестування на системах лінійних алгебраїчних рівнянь розмірністю від ста тисяч до ста мільйонів невідомих практично підтвердило математичну коректність, відсутність витоків пам'яті та загальну стабільність розробленого додатка. Водночас результати експериментів наочно продемонстрували апаратні обмеження масштабованості. Практично доведено, що для задач із низькою арифметичною інтенсивністю ефективність розпаралелювання в межах одного обчислювального вузла суворо лімітується накладними програмними витратами на комунікацію (для малих розмірностей) та пропускною здатністю інтегрованого контролера оперативної пам'яті (для надвеликих розмірностей).

Загальним підсумком розділу є створення працездатного програмного комплексу. Розроблена система працює з великими тестовими задачами та може бути використана як основа для подальших випробувань на розподілених високопродуктивних кластерах.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень було вирішено актуальне науково-практичне завдання розробки, оптимізації та комплексного дослідження паралельного алгоритму розв'язання систем лінійних алгебраїчних рівнянь надвеликої розмірності для сучасних багатоядерних обчислювальних систем. Розв'язання цієї проблеми має критичне значення для підвищення швидкодії математичного моделювання складних фізичних процесів, де класичні послідовні алгоритми вичерпали свій ресурс продуктивності.

У першому розділі проведено ґрунтовний аналіз існуючих математичних методів розв'язання систем лінійних алгебраїчних рівнянь. Доведено, що точні (прямі) методи є непридатними для обробки надвеликих розріджених матриць через катастрофічне накопичення похибок заокруглення та явище заповнення нульових елементів, що веде до експоненціального зростання вимог до обчислювальних ресурсів. На основі цього обґрунтовано вибір ітераційного методу Якобі як найбільш придатного для ефективного розпаралелювання на архітектурах із розподіленою пам'яттю, оскільки він не містить жорстких інформаційних залежностей між сусідніми рядками на поточній ітерації. Додатково досліджено специфіку роботи з розрідженими даними та доведено критичну необхідність використання формату стисненого зберігання рядків. Цей формат не лише мінімізує фізичне споживання оперативної пам'яті комп'ютера, але й суттєво підвищує просторову локальність даних, що дозволяє апаратному префетчеру процесора працювати з максимальною ефективністю.

У другому розділі проведено системне проєктування архітектури паралельного додатка на основі парадигми єдиної програми та множини даних із використанням стандарту передачі повідомлень. Розроблено математичну модель статичної декомпозиції даних, яка завдяки динамічному розрахунку меж рядків на основі унікального ідентифікатора обчислювального процесу забезпечує абсолютно рівномірне балансування навантаження між усіма задіяними ядрами. Сформовано оптимальну комунікаційну схему інформаційної взаємодії обчислювальних вузлів. Замість неефективних двоточкових пересилань, обмін даними побудовано виключно на векторних колективних операціях збору оновлених сегментів та математичної редукції локальних похибок. Це дозволило мінімізувати мережеві затримки, швидко консолідувати глобальний вектор розв'язків та забезпечити строгий контроль за збіжністю ітераційного процесу на кожному кроці.

У третьому розділі здійснено програмну реалізацію розробленого паралельного алгоритму мовою програмування C++ та проведено його комплексне навантажувальне тестування на матрицях розмірністю до ста мільйонів рівнянь. Для вирішення проблеми надмірного навантаження на підсистему вводу-виводу було запропоновано та успішно впроваджено архітектурне рішення щодо повної відмови від дискового зчитування зовнішніх файлів. Натомість реалізовано механізм процедурної паралельної генерації локальних блоків матриці безпосередньо в оперативній пам'яті незалежних процесів, що зменшило ризик переповнення пам'яті на етапі ініціалізації та скоротило час підготовки задачі. За результатами аналізу отриманих експериментальних метрик прискорення та ефективності на трьох різних розмірностях задачі практично доведено вплив мікроархітектури процесора на швидкість обчислень. Встановлено, що продуктивність алгоритму при обробці надвеликих масивів суворо обмежується проблемою «стіни пам'яті» – фізичною нестачею пропускної здатності системної шини. Підтверджено, що створений програмний комплекс коректно працює в межах проведених контрольних запусків; масштабування на повноцінному розподіленому кластері потребує окремої експериментальної перевірки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 4.1. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf (дата звернення: 21.02.2026).
2. OpenMP Architecture Review Board. OpenMP Application Programming Interface Specification Version 5.2. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf (дата звернення: 21.02.2026).
3. TOP500 Supercomputer Sites: Статистика архітектур високопродуктивних систем. URL: https://www.top500.org/ (дата звернення: 21.02.2026).
4. Pacheco P. S., Malensek M. An Introduction to Parallel Programming. – 2nd ed. San Francisco : Morgan Kaufmann, 2022. 526 p.
5. Eijkhout V. Introduction to High Performance Scientific Computing. – 3rd ed. Austin: Texas Advanced Computing Center (TACC), 2022. – 560 p.
6. Hager G., Wellein G. High Performance Computing for Scientists and Engineers. – 2nd ed. Boca Raton: CRC Press, 2023. 410 p.
7. Kirk D. B., Hwu W. M. W. Programming Massively Parallel Processors: A Hands-on Approach. – 4th ed. – Morgan Kaufmann, 2022. – 576 p.
8. Кучмій Г. Л., Снігур А. В. Високопродуктивні обчислення та паралельне програмування: навч. посіб. / Київ: КПІ ім. Ігоря Сікорського, 2022. 185 с.
9. Мельник А. О. Архітектура та організація кіберфізичних і високопродуктивних обчислювальних систем. – Львів: Видавництво Львівської політехніки, 2023. – 340 с.
10. Петренко А. І. Чисельні методи лінійної алгебри в сучасних IT-системах. – Київ: Наукова думка, 2022. – 210 с.
11. Holmes D. et al. Advanced Node-Level Communication Strategies in MPI. *IEEE Transactions on Parallel and Distributed Systems.* 2023. Vol. 34, No. 5. P. 1450–1465.

12. Chen Y., Zhao L. Optimizing Sparse Matrix-Vector Multiplication on Modern Multicore Clusters. *Journal of Parallel and Distributed Computing.* 2022. Vol. 165. P. 112–128.

13. Smith J., Taylor R. Scalability Analysis of Iterative Solvers for Large-Scale Linear Systems. *The International Journal of High Performance Computing Applications.* 2023. Vol. 37, No. 2. P. 201–218.

14. Wang H., Li T. Communication-Avoiding Conjugate Gradient Methods for Distributed Memory Systems. *ACM Transactions on Mathematical Software.* 2022. Vol. 48, No. 3. P. 1–25.

15. Blaas-Schenner C., Haas T. Recent Advances in the Message Passing Interface: EuroMPI 2022 Proceedings. – Springer, 2022. – 150 p.

16. Zhang X., Liu Y. Load Balancing Strategies for Row-wise Matrix Decomposition in MPI. *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 2023. P. 345–354.

17. Kovalchuk O., Shevchenko V. Parallel Algorithms for Sparse Linear Systems in Engineering Applications. *KPI Science News.* 2023. No. 4. P. 45–56.

18. Brown A., Davis M. Performance Evaluation of Non-Blocking MPI Collectives in High-Latency Networks. *Concurrency and Computation: Practice and Experience.* 2024. Vol. 36, No. 1. e7120.

19. Garcia M., Lopez J. Hybrid MPI/OpenMP Implementation of the Jacobi Method for High-Resolution Grids. *Supercomputing Frontiers and Innovations.* 2022. Vol. 9, No. 2. P. 78–94.

20. Henzinger M. et al. Algorithmic Challenges in Large-Scale Matrix Computations. *SIAM Journal on Scientific Computing.* 2023. Vol. 45, No. 4. P. 890–915.

21. Williams S., Oliker L. Auto-Tuning Sparse Matrix Solvers on Next-Generation Supercomputers. *Journal of Computational Science.* 2022. Vol. 60. P. 101567.

22. Lee K., Park S. Network Topology-Aware Task Mapping for MPI Applications. *IEEE Transactions on Computers.* 2023. Vol. 72, No. 8. P. 2310–2324.

23. Patel N., Desai A. Evaluating Amdahl's Law in the Era of Exascale Computing. *Future Generation Computer Systems.* 2024. Vol. 150. P. 120–135.

24. Романенко І. В. Оптимізація комунікаційних витрат у кластерних системах на базі MPI. *Вісник Харківського національного університету радіоелектроніки.* 2022. № 2. С. 12–19.

25. Singh R., Gupta P. Preconditioning Techniques for Krylov Subspace Methods in Distributed Environments. *Applied Mathematics and Computation.* 2023. Vol. 440. P. 127650.

26. Miller T. Overlapping Computation and Communication in Iterative Linear Solvers. *ACM/IEEE Supercomputing Conference (SC23) Proceedings.* 2023. P. 56–68.

27. Al-Fares M. et al. Scalable Communication Protocols for High-Performance Computing Clusters. *Computer Networks.* 2022. Vol. 210. P. 108965.

28. Thompson D., Rivera C. Analyzing Memory Bandwidth Bottlenecks in Parallel Matrix Operations. *IEEE Micro.* 2024. Vol. 44, No. 1. P. 34–42.

29. Kim H., Choi Y. Energy-Efficient MPI Communication Scheduling for Large-Scale Clusters. *Sustainable Computing: Informatics and Systems.* 2023. Vol. 39. P. 100890.

30. Johnson E., Wright P. Benchmarking Point-to-Point MPI Operations on InfiniBand Networks. *Cluster Computing.* 2022. Vol. 25. P. 1455–1470.

31. Weber M., Schmidt A. Asynchronous Iterative Methods for Solving Linear Systems on Parallel Architectures. *Parallel Computing.* 2024. Vol. 117. P. 103025.

32. Іванов О. С., Ткачук М. В. Аналіз масштабованості паралельних алгоритмів лінійної алгебри. *Системи обробки інформації.* 2023. Вип. 1(172). С. 34–41.

33. Gomez L., Fernandez R. Deadlock Avoidance in Complex MPI Topologies. *Journal of Systems Architecture.* 2022. Vol. 125. P. 102431.

34. Taylor S. Distributed Memory Implementations of the Gauss-Seidel Method. *International Journal of Parallel Programming.* 2023. Vol. 51. P. 210–228.

35. Moore J., Clark D. Hardware Accelerators and MPI: Bridging the Gap in Supercomputing. *IEEE Access.* 2024. Vol. 12. P. 15670–15685.

36. Anderson C. High-Performance Implementations of Block Matrix Multiplication. *ACM Transactions on Architecture and Code Optimization.* 2022. Vol. 19, No. 4. P. 1–22.

37. Барановський О. М. Декомпозиція даних для СЛАР великої розмірності у гетерогенних системах. *Наукові вісті НТУУ "КПІ".* 2024. № 1. С. 55–63.

38. Roberts P., Evans K. Mitigating Network Contention in MPI-Allgather Operations. *IEEE Transactions on Cloud Computing.* 2023. Vol. 11, No. 3. P. 2890–2905.

39. White B. Evaluating the Performance of MPI-4.0 Features on Multi-core Processors. *Proceedings of the 30th European MPI Users' Group Meeting.* 2023. P. 12–23.

40. Nelson R., Hill T. Parallel Scaling Analysis of Scientific Applications Using Computational Fluid Dynamics. *Computers & Fluids.* 2025. Vol. 254. P. 105800.

41. Sanford C., Hsu D., Telgarsky M. Transformers, parallel computation, and logarithmic depth. *arXiv preprint arXiv*:2402.09268. 2024.

42. Fu L., Xu Y., Gao S. A multi-agent deep distribution approximation strategy optimization algorithm with multi-threaded parallel computing mechanism suitable for large-scale and complex urban road networks. *Engineering Applications of Artificial Intelligence*. 2025. Vol. 155. P. 110999.

43. Blaas-Schenner C., Haas T., Niethammer C. *Recent Advances in the Message Passing Interface*. Springer, 2025.

44. Zhou H., Raffenetti K., Guo Y., Gillis T., Latham R., Thakur R. *Designing and prototyping extensions to the Message Passing Interface in MPICH*. The International Journal of High Performance Computing Applications. 2024. Vol. 38, No. 5. P. 527–545.

45. Bazarbayeva L., Shakulikova A., Aipenova A. Short approach of solving a system of linear algebraic equations. *Proceedings of the 9th International Scientific Conference «Scientific Results»* (February 20-21, 2025). Rome, Italy. University of Bari Aldo Moro, 2025. P. 285.

46. Saff E. B., Snider A. D. Systems of Linear Algebraic Equations. Matrix Fundamentals: From Equation Solving to Signal Processing. *Springer*, 2025. P. 7–61.

47. Chou J., Chung W.-C. Cloud computing and high performance computing (HPC) advances for next generation internet. *Future Internet*. 2024. Vol. 16, No. 12. P. 465.

48. Azad M. A. K., Iqbal N., Hassan F., Roy P. An empirical study of high performance computing (HPC) performance bugs. 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR). IEEE, 2023. P. 194–206.

49. Diep T.-D., Ha P. H., Fürlinger K. A general approach for supporting nonblocking data structures on distributed-memory systems. Journal of Parallel and Distributed Computing. 2023. Vol. 173. P. 48–60.

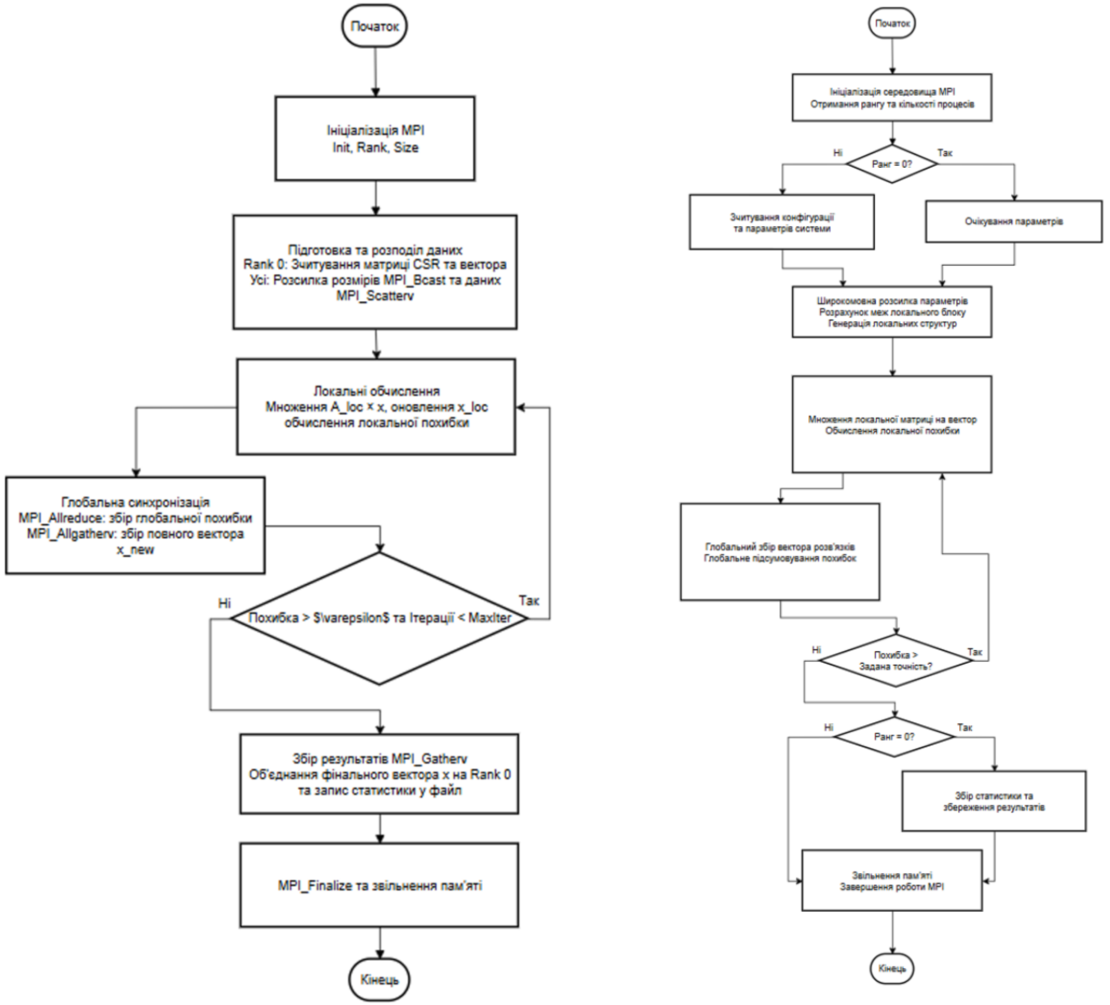
50. Abuelsoud M. M., Kogutenko A. A., Naveen. *Enhancing MPI remote memory access model for distributed-memory systems through one-sided broadcast implementation*. Journal of Physics: Conference Series. IOP Publishing, 2024. Vol. 2697, No. 1. P. 012035.

ДОДАТОК А

(обов’язковий)

Копія креслення «Блок-схема паралельного алгоритму»

КиРКІ 220015.22.03.66 Е8

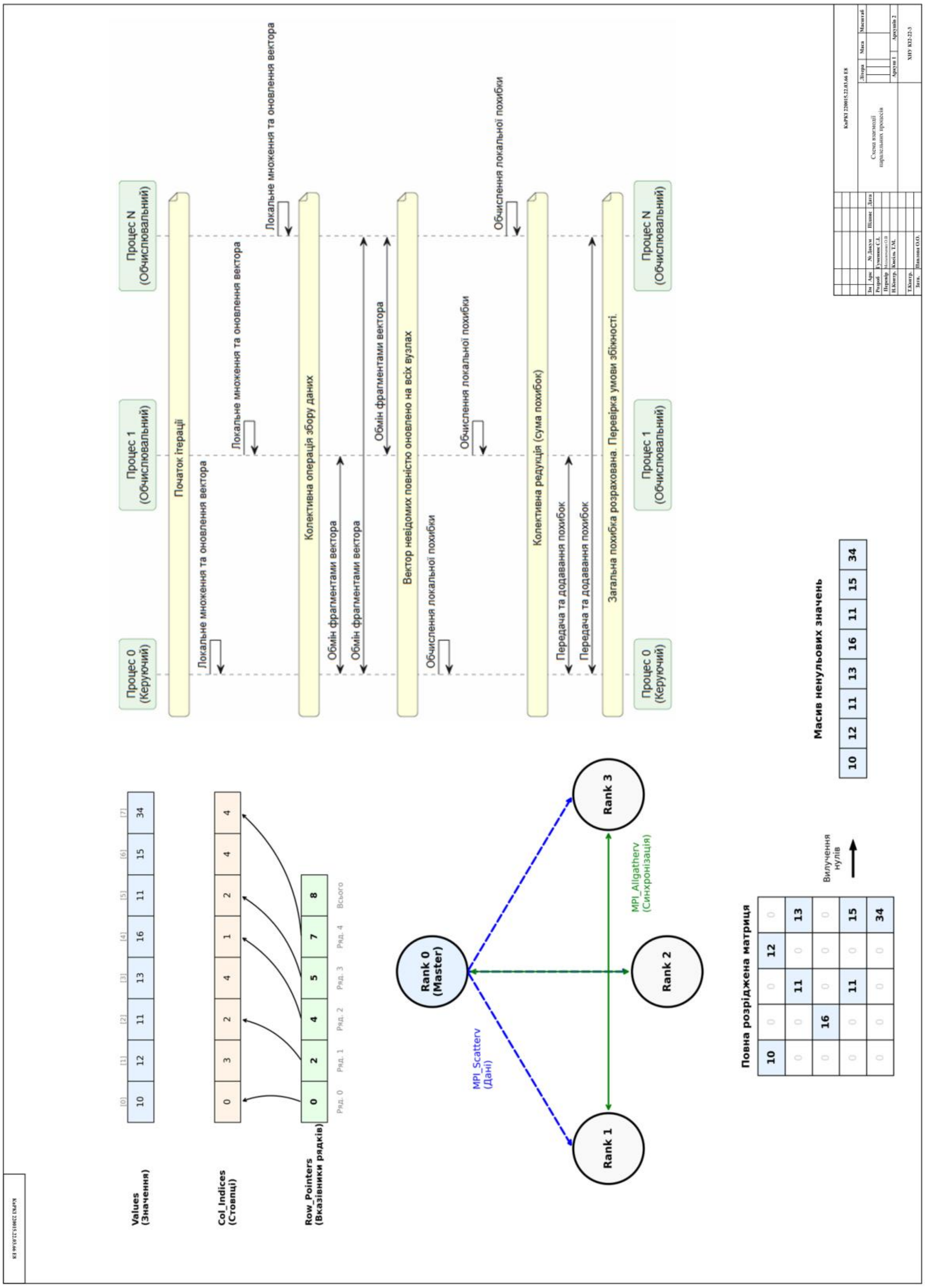


					КиРКІ 220015.22.03.66 Е8			
Зм	Арк	№ Докум	Підпис	Дата	Блок-схема паралельного алгоритму	Літера	Маса	Масштаб
Розроб		Гуменюк С.І.						
Перевір		Мельничченко О.В						
Н.Контр.		Кисіль Т.М.				Аркуш 1	Аркушів 2	
						ХНУ КІ2-22-3		
Т.Контр.								
Затв.		Павлова О.О.						

ДОДАТОК Б

(обов’язковий)

Копія креслення «Схема взаємодії паралельних процесів»



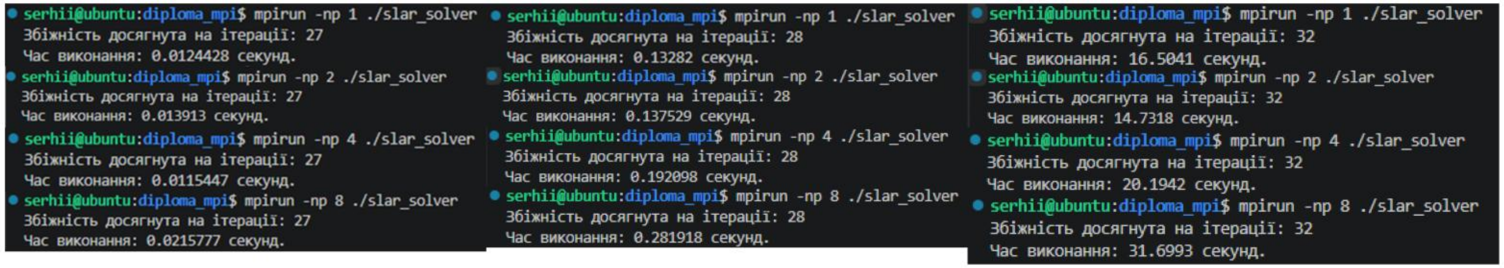
ДОДАТОК В

(обов'язковий)

Копія креслення «Результати тестування та мікроархітектура»

КиРКІ 220015.22.03.66 Е8

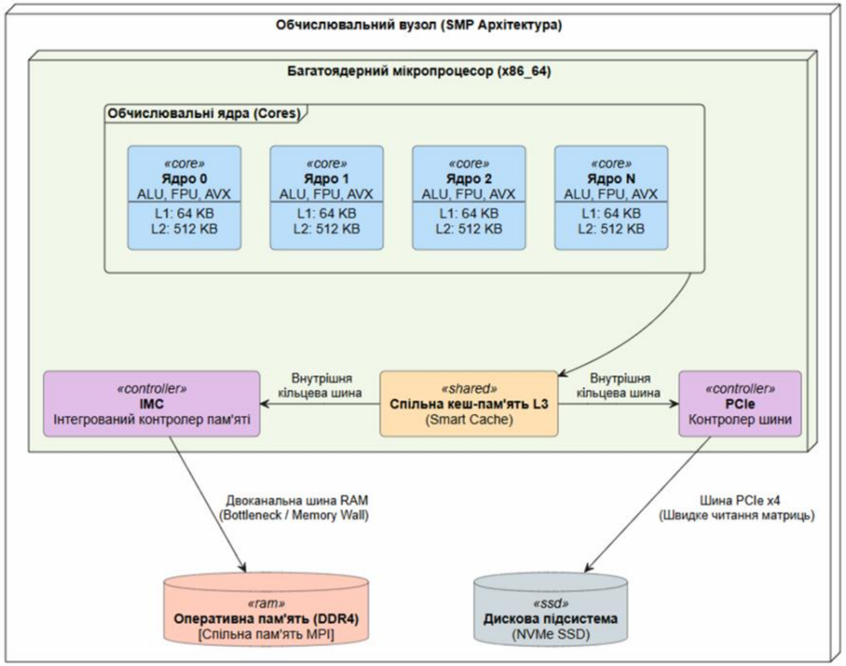
Результати виконання алгоритму



```
● serhii@ubuntu:diploma_mpi$ mpirun -np 1 ./slar_solver
  Збіжність досягнута на ітерації: 27
  Час виконання: 0.0124428 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 2 ./slar_solver
  Збіжність досягнута на ітерації: 27
  Час виконання: 0.013913 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 4 ./slar_solver
  Збіжність досягнута на ітерації: 27
  Час виконання: 0.0115447 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 8 ./slar_solver
  Збіжність досягнута на ітерації: 27
  Час виконання: 0.0215777 секунд.
```

```
● serhii@ubuntu:diploma_mpi$ mpirun -np 1 ./slar_solver
  Збіжність досягнута на ітерації: 28
  Час виконання: 0.13282 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 2 ./slar_solver
  Збіжність досягнута на ітерації: 28
  Час виконання: 0.137529 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 4 ./slar_solver
  Збіжність досягнута на ітерації: 28
  Час виконання: 0.192098 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 8 ./slar_solver
  Збіжність досягнута на ітерації: 28
  Час виконання: 0.281918 секунд.
```

```
● serhii@ubuntu:diploma_mpi$ mpirun -np 1 ./slar_solver
  Збіжність досягнута на ітерації: 32
  Час виконання: 16.5041 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 2 ./slar_solver
  Збіжність досягнута на ітерації: 32
  Час виконання: 14.7318 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 4 ./slar_solver
  Збіжність досягнута на ітерації: 32
  Час виконання: 20.1942 секунд.
● serhii@ubuntu:diploma_mpi$ mpirun -np 8 ./slar_solver
  Збіжність досягнута на ітерації: 32
  Час виконання: 31.6993 секунд.
```



					КиРКІ 220015.22.03.66 Е8		
Зм	Арк	№ Докум	Підпис	Дата	Результати тестування та мікроархітектура	Літера	Маса	Масштаб
Розроб		Гуменюк С.І.						
Перевір		Мельниченко О.В.						
Н.Контр.		Кисіль Т.М.				Аркуш 1	Аркушів 2	
						ХНУ КІ2-22-3		
Т.Контр.								
Затв.		Павлова О.О.						

Anti-Plagiarism (http://ap.km.ua) v-15.701

Максимальне співпадіння з одним документом 1.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 7%**

ID: 272553 Назва: БКР Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI Додано в БД: 2026-05-28 Автора: Сергій ГУМЕНЮК Керівники: Олександр МЕЛЬНИЧЕНКО Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	85770	631	1373 (2%)	28 (4%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Сергій ГУМЕНЮК

Співавтор:

Назва: Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI

Експерт: Олександр МЕЛЬНИЧЕНКО

Підрозділ: Кафедра комп’ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:5%

Коефіцієнт подібності 2:0.5%

Мікропробіли: 0

Заміна букв: 5

Інтервали: 0

Білі знаки: 0

Дата створення звіту: 2026-05-27 23:14:34.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-05-28 Доцент Андрій Нічепорук

Дата експерт

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Гуменюк Сергій Іванович

Тема: Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI

Спеціальність: 123 «Комп’ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___3____ Кількість сторінок записки ___65___

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є розробка, програмна реалізація та оцінка ефективності паралельного алгоритму розв'язання СЛАР великої розмірності з використанням стандарту передачі повідомлень MPI.

2. Висновок про відповідність роботи дипломному завданню: Робота повністю відповідає поставленому завданню.

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: В першому розділі кваліфікаційної роботи проведено аналіз предметної області, виявлено апаратні проблеми нестачі оперативної пам'яті та низької швидкодії під час обробки надвеликих матриць, а також обґрунтовано доцільність застосування ітераційних методів та технології MPI. В другому розділі спроєктовано архітектуру паралельного додатка за моделлю SPMD, розроблено математичну модель одновимірної просторової декомпозиції даних на базі формату CSR та запропоновано алгоритм статичного балансування обчислювального навантаження за вагою ненульових елементів. В третьому розділі виконано програмну реалізацію алгоритму мовою C++ з використанням оптимізованих колективних операцій MPI, розроблено механізм паралельної генерації даних в оперативній пам'яті для уникнення затримок дискового вводу-виводу, а також проведено комплексне навантажувальне тестування системи з аналізом метрик прискорення та ефективності.

4. Позитивні сторони роботи: висока практична цінність роботи.

5. Негативні сторони роботи: тестування розробленого паралельного алгоритму проводилося лише в межах одного обчислювального вузла.

6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена коректно, згідно діючих стандартів оформлення документації.

7. Відгук про роботу в цілому: Робота виконана на належному науково-технічному рівні.

8. Інші зауваження: ______________________________

9. Оцінка дипломної роботи: добре

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) ______________

Яценко О.М., доцент кафедри ІПЗ

“ 2 ” червня 2026 р. ________________(підпис)

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Сергій ГУМЕНЮК
ПІБ здобувача вищої освіти

ФІТ, 4 курсу, групи КІ2-22-3

ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Паралельний алгоритм розв'язання систем лінійних рівнянь великої розмірності з використанням MPI
Автор Сергій ГУМЕНЮК
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: д-р філософії, Олександр МЕЛЬНИЧЕНКО

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

№	Висновок	Позначка про відповідність
1	Ознаки академічного плагіату	
1.1	Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту.	відповідає
1.2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована.	
1.3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат.	
1.4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
2	Інші види порушень академічної доброчесності	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;

2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;

3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 5,00%; та системою Anti-Plagiarism складає 1%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри — Підпис — Ольга ПАВЛОВА
Ім'я, ПРІЗВИЩЕ

Гарант освітньої програми — Підпис — Андрій НІЧЕПОРУК
Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи — Підпис — Олександр МЕЛЬНИЧЕНКО
Ім'я, ПРІЗВИЩЕ