Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)
Навчально-науковий інститут
комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему: Паралельний алгоритм розрахунку розподілу температури у тривимірній області

Виконав: студент 6 курсу групи КН-63м
спеціальності 122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Юрченко Максим Андрійович
(прізвище та ініціали)

Керівники Карашецький В.П.
(прізвище та ініціали)

Рецензент Покрищук О.В
(прізвище та ініціали)

Львів – 2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ Комп'ютерних наук та інформаційних технологій
Кафедра Комп'ютерних наук
Рівень вищої освіти другий (магістерський)
Спеціальність 122 "Комп'ютерні науки"

ЗАТВЕРДЖУЮ
Завідувачка кафедри КН
_____ Борецька І.Б.
"10" ____ 2025 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Юрченку Максиму Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи *Паралельний алгоритм розрахунку розподілу температури у тривимірній області*
керівник роботи Карашецький Володимир Петрович к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від 29 квітня 2025 р. № С-288
2. Термін подання студентом роботи 10.12.2025 р.
3. Вихідні дані до роботи *Аналіз існуючих математичних моделей вирішення поставленої задачі. Визначення переваг і недоліків різних фреймворків для реалізації поставленого завдання та вибору найкращих з них. Література за тематикою роботи.*

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ. Розділ 1. Огляд сучасного стану проблемної області.
Розділ 2. Інформаційне забезпечення.
Розділ 3. Математичне забезпечення.
Розділ 4. Програмне забезпечення.
Розділ 5. Розроблення стартап-проєкту.
Висновки. Список використаних джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди для доповіді (підготовка матеріалу для доповіді загальним обсягом 10-12 слайдів).

6. Дата видачі завдання *1 травня 2025 року.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Огляд та системний аналіз проблемної області. Огляд літературних джерел на досліджувану тему. Постановка задачі проєкту та формування функціональних вимог. Збір потрібних матеріалів.	*02.05.2025 р.* *23.05.2025 р.*	*Виконав*
2.	Огляд сучасного стану проблемної області. Написання та оформлення першого розділу пояснювальної записки.	*24.05.2025 р.* *01.06.2025 р.*	*Виконав*
3.	Написання та оформлення другого розділу. Аналіз інформаційного та математичного забезпечення.	*02.06.2025 р.* *19.07.2025 р.*	*Виконав*
4.	Написання та оформлення третього розділу пояснювальної записки. Підготовка програмного забезпечення.	*20.08.2025 р.* *03.09.2025 р.*	*Виконав*
5.	Програмна реалізація. Написання та оформлення четвертого розділу пояснювальної записки.	*04.09.2025 р.* *20.10.2025 р.*	*Виконав*
6.	Розроблення стартап-проєкту.	*21.10.2025р.* *20-11.2025р.*	*Виконав*
7.	Оформлення пояснювальної записки та здача на рецензування.	*21.11.2025 р.* *04.12.2025 р.*	*Виконав*

Студент _____ *Юрченко М.*
(підпис) (прізвище та ініціали)

Керівники роботи _____ *Карашецький В.П.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 76 сторінок пояснювальної записки, 21 рисунок, 19 таблиць, 5 додатків, 11 літературних джерел. Робота присвячена розробці паралельного алгоритму розрахунку тривимірних стаціонарних температурних полів у середовищах з анізотропними та нелінійними властивостями методом скінченних елементів. Для подолання обчислювальних обмежень середовища Python застосована гібридна паралельна схема, що поєднує JIT-компіляцію Numba та багатопроцесорну обробку Joblib. Впроваджено унікальну оптимізацію етапу Aggregation за допомогою формату COO. Досягнуто прискорення алгоритму на етапі розв'язання СЛАР. Відображення результатів моделювання та роботи алгоритмів подано у вигляді двовимірних зрізів і тривимірних графіків. Розроблено концепцію стартап-проєкту "ThermalCore Sim". Проєкт демонструє високу технологічну зрілість та ринкову життєздатність, пропонуючи економічно ефективну альтернативу традиційному інженерному програмному забезпеченню.

Ключові слова: *паралельний алгоритм, МСЕ, термоаналіз, Numba/Joblib, гібридна оптимізація, агрегація, 3D-моделювання.*

ABSTRACT

The thesis contains 76 pages of explanatory notes, 21 figures, 19 tables, 5 appendix, 11 references. The work is devoted to the development of a parallel algorithm for calculating three-dimensional stationary temperature fields in media with anisotropic and nonlinear properties using the finite element method. To overcome the computational limitations of the Python environment, a hybrid parallel scheme combining Numba JIT compilation and Joblib multiprocessor processing was used. A unique optimization of the Aggregation stage using the COO format was implemented. The computational bottleneck at the stage of solving a system of linear algebraic equations was accelerated. The results of modeling and the operation of algorithms are presented in the form of two-dimensional slices and three-dimensional graphs. The concept of the startup project "ThermalCore Sim" was developed. The project demonstrates high technological maturity and market viability, offering a cost-effective to traditional engineering software.

Keywords: *parallel algorithm, FEM, thermal analysis, Numba/Joblib, hybrid optimization, aggregation, 3D modeling*.

ТЕХНІЧНЕ ЗАВДАННЯ

Проаналізувати фізичні принципи та адаптувати математичну модель стаціонарного розподілу температури у тривимірній області, включаючи нелінійні та анізотропні властивості середовища. Сформулювати крайову задачу та вивести її апроксимацію за допомогою Методу Скінченних Елементів (МСЕ) на тетраедральних елементах.

Проаналізувати існуючі методи підвищення продуктивності Python (Numba, Joblib) та формати розріджених матриць. Спираючись на цей аналіз, необхідно розробити паралельний алгоритм МСЕ, який використовував би JIT-компіляцію Numba для прискорення етапу Assembly. Спроєктувати та реалізувати інноваційний підхід до паралельної агрегації даних через формат COO. Провести експериментальне дослідження - оцінити коефіцієнти прискорення..

Розробити концепцію стартап-проєкту "ThermalCore Sim" на базі високошвидкісного кернела. Необхідно проаналізувати ринкові можливості, загрози та конкурентне середовище (SWOT). Далі потрібно спроєктувати модель ціноутворення, сформулювати цільові групи та визначити оптимальну стратегію збуту

Оформити пояснювальну записку відповідно до методичних вказівок.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – набір визначень взаємодії різнотипного ПЗ;

COO – coordinate list (список координат);

CPU – central processing unit;

CSR – compressed sparse row (стиснений рядок розрідженої матриці);

Developer Tools – інструменти для розробки;

GIL – global interpreter lock (глобальне блокування інтерпретатора);

IDE – комп'ютерна програмне середовище;

JIT – just-in-time compilation (компіляція "на льоту");

LIL – list of lists (список списків);

ML – машинне навчання;

MPI – message passing interface;

PETSc – portable extensible toolkit for scientific computation;

UI – інтерфейс користувача;

БД – база даних;

ІС – інформаційна система;

МСЕ – метод скінченних елементів;

МСО – метод скінченних об'ємів;

МСР – метод скінченних різниць;

ОС – операційна система;

ПЗ – програмне забезпечення;

ПК – персональний комп’ютер;

СЛАР – система лінійних алгебраїчних рівнянь;

СУБД – система управління базами даних.

ВСТУП

Сучасні науково-технічні дослідження, зокрема у галузях машинобудування, енергетики та мікроелектроніки, висувають високі вимоги до точності та швидкості розрахунку фізичних процесів. Розрахунок тривимірних стаціонарних потенціальних теплових полів є фундаментальною задачею, необхідною для проєктування надійних і довговічних конструкцій, що функціонують в умовах високих теплових навантажень. Метод скінченних елементів (МСЕ) є загальновизнаним інструментом для розв'язання цієї крайової задачі, оскільки він забезпечує необхідну гнучкість для моделювання областей зі складною геометрією, а також дозволяє коректно враховувати анізотропні та нелінійні властивості матеріалів. Однак, застосування МСЕ до великомасштабних тривимірних сіток призводить до формування розріджених систем рівнянь із сотнями тисяч невідомих, що створює значний обчислювальний виклик. Реалізація цих обчислень у середовищі Python, попри його гнучкість, зіштовхується з обмеженнями продуктивності, спричиненими Global Interpreter Lock (GIL) та повільним виконанням циклів, що прямо вказує на необхідність розробки ефективних паралельних алгоритмів.

Аналіз обчислювальної складності FEM-алгоритму виявив два ключових етапи, які стають обмежувачами швидкості у послідовному коді: Assembly (обчислення внесків елементів) та Aggregation (складання глобальної матриці жорсткості). Етап Assembly успішно піддається паралелізації за допомогою гібридного підходу, що поєднує JIT-компіляцію Numba для прискорення локальних обчислень та Joblib для ефективного розподілу роботи між ядрами CPU, обходячи обмеження GIL. Проте, критичною проблемою залишається Aggregation, де використання традиційних для послідовного заповнення форматів розріджених матриць (наприклад, LIL) створює послідовне вузьке місце, нівелюючи весь виграш від паралельного Assembly. Для радикального вирішення цієї проблеми було досліджено використання проміжного формату COO (Coordinate List), який дозволяє

паралельно збирати внески та виконувати одночасно високооптимізовану масову агрегацію.

Методологія включає порівняльний аналіз двох паралельних реалізацій (Numba + LIL vs. Numba + COO) на сітках, розмірність яких зростає до десятків тисяч елементів, що дозволяє точно оцінити масштабованість. Проведені експерименти підтвердили, що оптимізована схема усунула обмежувачі Assembly та Aggregation, забезпечивши значне загальне прискорення (до 5.93 на 2-ядерному CPU). В результаті, обчислювальний боттлнек було успішно зміщено на етап Solver (розв'язання СЛАР), що визначає подальші напрямки досліджень, пов'язані з інтеграцією паралельних розв'язувачів. Робота демонструє практичний шлях до створення високопродуктивного та масштабованого FEM-коду в середовищі Python.

Актуальність обумовлена критичною необхідністю точного та швидкого розрахунку тривимірних стаціонарних температурних полів у складних інженерних об’єктах (електроніка, машинобудування) із нелінійними та анізотропними властивостями. Традиційні послідовні FEM-реалізації, особливо на базі Python, не можуть впоратися з обчислювальним навантаженням великих сіток, що вимагає розробки ефективних паралельних алгоритмів, які повністю використовують багатоядерну архітектуру CPU.

Об’єкт дослідження – процес розподілу тривимірного стаціонарного потенціального температурного поля у довільній області, заповненій безгістерезисними нелінійними анізотропними середовищами.

Предмет дослідження є паралельний алгоритм методу скінченних елементів (МСЕ) для розв'язання крайової задачі теплопровідності та обчислювальна продуктивність його ключових етапів (Assembly, Aggregation) в умовах багатопроцесорної обробки.

Мета роботи розробити та експериментально дослідити високопродуктивний гібридний паралельний алгоритм розрахунку розподілу температури, який усуває

обчислювальні боттлнеки на етапах Assembly та Aggregation, забезпечуючи максимальне прискорення на багатоядерних архітектурах.

Новизна роботи полягає у стратегічній оптимізації етапу Aggregation через впровадження проміжного формату COO (Coordinate List), що дозволяє замінити повільну послідовну інкрементальну операцію (LIL) на високоефективне пакетне перетворення COO to CSR після паралельного збору внесків. Цей підхід є ключовим для усунення критичного обмежувача швидкості.

Практична значимість полягає у створенні масштабованого та обчислювально ефективного програмного забезпечення для інженерного аналізу, яке дозволяє швидко та точно виконувати 3D-моделювання на стандартних багатоядерних CPU. Досягнуте значне прискорення дозволяє дослідникам та інженерам працювати з сітками вищої розмірності за прийнятний час.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

Розрахунок тривимірних стаціонарних потенціальних теплових полів методом скінченних елементів (МСЕ) є фундаментальною задачею в інженерному моделюванні. Існуючі послідовні реалізації, особливо ті, що використовують інтерпретовані мови програмування, стикаються з критичними обчислювальними бар'єрами. Головні "вузькі місця" виникають на етапах формування матриці жорсткості: обчислення внесків елементів (Assembly) та їх складання (Aggregation). Хоча прискорення Assembly можливе завдяки JIT-компіляторам, неефективна агрегація у стандартних форматах розріджених матриць (наприклад, LIL) зводить нанівець виграш у продуктивності на великих сітках. Таким чином, основна проблема полягає у розробці та дослідженні ефективних гібридних паралельних схем, що усувають ці послідовні боттлнеки, особливо шляхом оптимізації Aggregation через використання формату COO.

У сучасних дослідженнях [2, 3] значна увага приділяється розробленню ефективних паралельних підходів до моделювання теплових та масообмінних процесів у тривимірних середовищах. Особливої актуальності такі методи набувають при аналізі складних капілярно-пористих матеріалів з анізотропними теплофізичними властивостями [4], де обчислювальні навантаження суттєво зростають зі збільшенням розмірності задачі та щільності скінченно-елементної сітки. У публікації автори зазначають, що «на основі тривимірної математичної моделі неізотермічного тепломасопереносу в капілярно-пористих матеріалах, з урахуванням анізотропії теплофізичних властивостей, було розроблено програмне забезпечення для проведення скінченно-елементного аналізу вологопереносу з використанням технології CUDA» [2].

Дослідники підкреслюють, що використання гібридної структури збереження даних – частково на локальному пристрої, частково у хмарному середовищі Azure – дозволило реалізувати гнучкий підхід до обчислень та масштабування. Як зазначено у джерелі, «застосування паралельних технологій з використанням бібліотеки cuBLAS дозволило перенести всі великомасштабні матричні обчислення на

графічний процесор, що зменшило споживання ресурсів зі збільшенням щільності сітки» [2].

Наведені результати підтверджують доцільність використання паралельних обчислень, зокрема GPU-орієнтованих технологій, для прискорення моделювання теплових процесів у тривимірних областях. У межах цієї роботи подібні підходи адаптовано до реалізації паралельного алгоритму розрахунку температурного поля засобами Python.

1.1 Актуальність та інженерне значення задач теплопровідності

Задачі розрахунку стаціонарних потенціальних теплових полів є критично важливими для проєктування та експлуатації широкого спектру інженерних систем і пристроїв. Розуміння розподілу температури в тривимірних об'єктах необхідне для забезпечення їхньої надійності, довговічності та оптимальної продуктивності. Сучасні конструкції, такі як компоненти аерокосмічної техніки, потужні електронні пристрої та ядерні реактори, функціонують в умовах високих теплових навантажень. Повільний розрахунок температурних полів може призвести до пошкоджень, термічної деградації матеріалів та неефективного використання енергії.

Метод скінченних елементів є загальновизнаним стандартом для розв'язання таких задач, оскільки він дозволяє моделювати області зі складною геометрією та різнорідними, часто анізотропними (залежними від напрямку) та нелінійними (залежними від температури) властивостями середовищ [10]. Особлива актуальність тривимірного аналізу полягає у виявленні локальних "гарячих точок", які неможливо точно передбачити за допомогою двовимірних або спрощених моделей. Інженерне значення цих розрахунків охоплює оптимізацію систем охолодження, проєктування теплообмінників та моделювання процесів лиття і термічної обробки матеріалів. Точне визначення температурних градієнтів також необхідне для подальшого термопружного аналізу, оскільки температурні деформації можуть викликати значні внутрішні напруження. З розвитком IT зростає потреба у моделюванні сіток із великою кількістю елементів (до сотень тисяч і мільйонів). Це вимагає розробки паралельних алгоритмів, здатних обробляти величезні масиви

даних за прийнятний час. Дослідження та оптимізація обчислювальної продуктивності FEM-кодів є необхідною умовою для забезпечення прогресу у сучасному інженерному проєктуванні та наукових дослідженнях.

1.2 Огляд чисельних методів розв'язання крайових задач

Для розв'язання крайових задач математичної фізики, до яких належить і задача теплопровідності, використовується три основні класи чисельних методів (Таблиця 1.1): метод скінченних різниць (МСР), метод скінченних об'ємів (МСО) та метод скінченних елементів (МСЕ) [5].

Таблиця 1.1 – Порівняння основних чисельних підходів

Метод	Основний Принцип	Переваги	Недоліки
Скінченних Різниць (МСР)	Апроксимація похідних у диференціальному рівнянні набором скінченних різниць. Область покривається регулярно розташованою сіткою.	Найпростіший для реалізації; висока ефективність на простих, прямокутних областях.	Низька гнучкість для складних геометрій та нерегулярних крайових умов; важко підтримувати високий порядок точності.
Скінченних Об'ємів (МСО)	Інтегрування рівняння по кожному контрольному об'єму (комірка сітки) та забезпечення збереження потоків через його грані.	Ідеальний для консервативних законів (наприклад, гідродинаміка, теплопередача); природна збереженість величин (маси, енергії).	Складніше досягти високого порядку точності; виникають труднощі при анізотропних середовищах.
Скінченних Елементів (МСЕ)	Розбиття області на малі елементи; мінімізація функціоналу або використання вагових функцій (Гальоркін). Апроксимація розв'язку поліномами всередині елементів.	Найвища гнучкість для довільних геометрій та крайових умов; легко адаптується до різнорідних та анізотропних середовищ; високий порядок точності.	Складніша програмна реалізація, особливо етапи Assembly та керування сіткою.

Обґрунтування вибору методу скінченних елементів (МСЕ). Вибір методу скінченних елементів для розрахунку тривимірних стаціонарних теплових полів є найбільш обґрунтованим з інженерної та обчислювальної точки зору з наступних причин:

1. Геометрична гнучкість МСЕ дозволяє використовувати неструктуровані сітки, що складаються з довільних елементів (у 3D – тетраедри та гексаедри). Це

критично важливо для моделювання реальних інженерних об'єктів, які майже завжди мають складну, вигнуту або кутову форму [8].

2. Точність апроксимації на відміну від МСР, який апроксимує диференціальні оператори в точці, МСЕ використовує функції форми (базисні функції) для апроксимації розв'язку по всьому об'єму елемента. Це забезпечує вищу точність, особливо при використанні елементів вищих порядків.
3. Облік властивостей середовища МСЕ легко адаптується до задач із різнорідними матеріалами та анізотропними (напрямково-залежними) властивостями, що є стандартною вимогою для розрахунку сучасних композиційних матеріалів чи електричних машин.
4. Граничні умови МСЕ природно включає крайові умови, як Діріхле (задана температура), так і Неймана (заданий потік), шляхом модифікації функціоналу та системи рівнянь.
5. Методологія МСЕ заснована на варіаційних принципах (мінімум функціоналу), забезпечує надійну теоретичну основу для гарантування стійкості та збіжності чисельного розв'язку.

Незважаючи на складність програмної реалізації (особливо на етапах Assembly та Aggregation), МСЕ є найкращим компромісом між універсальністю, гнучкістю та точністю, що необхідно для всебічного інженерного аналізу тривимірних теплових полів.

1.3 Етапи та обчислювальна складність алгоритму МСЕ

Алгоритм методу скінченних елементів для розв'язання крайових задач складається з чотирьох основних послідовних етапів, кожен з яких має власну обчислювальну складність, що визначає загальну продуктивність моделі [6].

1.3.1 Етап Дискретизації (Побудова Сітки)

Принцип даного етапу це розбиття тривимірної обчислювальної області на кінцеву кількість малих, геометрично простих елементів (у нашому випадку – тетраедри). Де теоретична обчислювальна складність зазвичай залежить від

алгоритму генерації. Для якісної сітки може бути близькою до $O(N_e \log N_e)$ або $O(N_e)$, де N_e – кількість елементів.

В результаті очікується творення набору вузлів (з координатами), елементів (списку вузлів, що формують кожен елемент) та відповідних крайових умов. Визначає кількість рівнянь N_{eq} (кількість внутрішніх вузлів) та кількість елементів N_e (тетраедрів), які прямо впливають на складність наступних етапів.

1.3.2 Етап Assembly (формування матриці жорсткості)

Це найбільш масово-паралельний етап, який передбачає незалежне обчислення внесків кожного елемента. Принцип його заключається в тому, що для кожного елемента *e* обчислюється його локальна матриця жорсткості K^e (або внесок у вектор навантаження F^e). Обчислення включає інтегрування функцій форми (наприклад, кубатурні формули для тетраедрів). Теоретична обчислювальна складність при цьому становитиме $O(N_e * L^3)$, де L – розмірність локальної матриці (зазвичай константа, L=4 для тетраедра першого порядку). Таким чином, складність лінійно залежить від кількості елементів N_e: $O(N_e)$. Такий процес потребує інтенсивних обчислень, що ідеально підходить для прискорення за допомогою Numba та Joblib (як показали ваші експерименти).

1.3.3 Етап Aggregation (складання внесків)

Це етап, де локальні внески елементів збираються в єдину глобальну систему лінійних алгебраїчних рівнянь (СЛАР). Діє за принципом додавання локальних елементів K^e до відповідних позицій у глобальній розрідженій матриці жорсткості K. Оскільки елементи сусідні, їхні внески накладаються на одні й ті ж вузли. Вимагає ефективного керування пам'яттю та розрідженою структурою даних. Хоча теоретично складність лінійна, фактична продуктивність сильно залежить від формату матриці (LIL чи COO) та ефективності операції додавання внесків.

1.3.4 Етап Solver (розв'язання СЛАР)

На цьому етапі знаходиться фінальний розв'язок – розподіл температур (потенціалів) у вузлах. Принцип розв'язання системи $K * T = F$, де K – глобальна матриця жорсткості, T – вектор невідомих температур, F – вектор навантажень. Використовуються прямі або ітераційні методи. Теоретична обчислювальна

складність $O(N_{eq}^{x})$, де N_{eq} – кількість рівнянь (кількість внутрішніх вузлів). Це етап із найвищою теоретичною обчислювальною складністю (поліноміальна залежність від N_{eq}), і він стає домінуючим вузьким місцем на великих сітках, як показали ваші результати, оскільки прямі розв'язувачі часто виконуються послідовно.

Таблиця 1.2 – Обчислювальна складність: залежність від розміру задачі

Етап	**Залежність від N_e (Елементи) / N_{eq} (Рівняння)**	**Поведінка**
Assembly	$O(N_e)$	Лінійна (ідеально паралелізована)
Aggregation	$O(N_e)$	Лінійна (але сильно залежить від реалізації)
Solver	$O(N_{eq}^{x})$ (x = 1.5- 3.0)	**Поліноміальна** (зростає найшвидше)

Враховуючи, що N_{eq} (кількість внутрішніх вузлів) пропорційна N_e (кількості елементів) для 3D сітки, етап Solver є теоретично найбільш складним і тому найбільш критичним для масштабованості на надвеликих задачах. Успіх оптимізації (Numba + COO) полягає в тому, що ми перетворили практично повільні лінійні етапи (Assembly/Aggregation) на надшвидкі, тим самим підтвердивши, що Solver є останнім і найвищим обчислювальним бар'єром.

1.4 Обчислювальна ефективність та вимоги до паралелізації FEM

Основна проблема обчислювальної продуктивності у реалізації МСЕ на інтерпретованих мовах [9], таких як Python, полягає в їхній відносно низькій швидкості порівняно з компільованими мовами (C/C++). Ця проблема особливо загострюється у великомасштабних 3D-задачах з великою кількістю елементів. Повільне виконання циклів та численних операцій з пам'яттю, характерних для Python, створює значний бар'єр на шляху до швидких розрахунків. Хоча бібліотеки, як NumPy, забезпечують високу швидкість для векторних операцій, їх ефективність знижується, коли логіка програми вимагає поелементного обходу даних або складних циклів, що не векторизуються. Це створює пряму необхідність у засобах Just-in-Time (JIT) компіляції (наприклад, Numba) для прискорення критичних ділянок коду.

Ідентифікація "Вузьких Місць". Аналіз продуктивності алгоритму MCE виявив два ключові етапи, які стають критичними обмежувачами швидкості:

1. Assembly (Обчислення Внесків Елементів): Хоча це лінійний ($O(N_e)$) і масово-паралельний етап, він є інтенсивним за обчисленнями. Початково він стає першим боттлнеком, але легко піддається прискоренню шляхом паралелізації (використання Joblib) та оптимізації коду (використання Numba).
2. Aggregation (Складання Внесків у Матрицю): Це найпідступніший боттлнек у послідовних реалізаціях. Він також має лінійну складність ($O(N_e)$), але його практична швидкість сильно залежить від структури даних. Використання неефективних структур для розріджених матриць (наприклад, `lil_matrix`) призводить до того, що цей етап виконується послідовно і надзвичайно повільно, нівелюючи виграш від паралельного Assembly.

Проблеми паралелізації та вибір формату даних. Для ефективної паралелізації FEM-коду потрібно вирішити дві взаємопов'язані проблеми. Перша - уникнення гонки даних. Вона виникає при паралельному Assembly кілька ядер намагаються одночасно записати дані (внески елементів) в одну і ту ж позицію глобальної матриці жорсткості. Це вимагає складних механізмів блокування, які можуть знизити продуктивність. Друга – вибір оптимальної структури. Традиційні формати, зручні для інкрементальних змін (як LIL), є повільними в Python і не підтримують ефективну паралельну агрегацію. Натомість, використання проміжних форматів, що підходять для паралельного збору даних (як COO), дозволяє кожному процесу незалежно генерувати свої внески. Подальше з'єднання цих масивів і однокрокова агрегація у фінальну матрицю (CSR) усуває проблему гонки даних та забезпечує феноменальне прискорення.

Кінцева мета оптимізації полягає у зсуві обчислювального боттлнека на етап із найвищою теоретичною складністю – Solver. Успішна паралелізація Assembly та оптимізація Aggregation підтверджує, що найбільшим обмежувачем швидкості на великих сітках стає вже не формування матриці, а послідовне розв'язання системи лінійних алгебраїчних рівнянь ($O(N_{eq}^x)$), що вимагає подальшого використання зовнішніх, паралельних бібліотек для розв'язувачів.

Висновок до розділу

Викладений в даному розділі матеріал засвідчив, що розрахунок тривимірних стаціонарних потенціальних теплових полів є критично важливою інженерною задачею, необхідною для проєктування надійних систем, що функціонують в умовах високих теплових навантажень.

Незважаючи на високу точність та універсальність МСЕ, його реалізація в інтерпретованому середовищі Python зіштовхується з гострою проблемою обчислювальної ефективності при переході до сіток великої розмірності (сотні тисяч елементів). Основними "вузькими місцями" в послідовному алгоритмі є Assembly (Обчислення внесків) – обчислювально-інтенсивний етап, що потребує оптимізації та Aggregation (Складання внесків) – етап, який через використання неефективних послідовних структур (наприклад, LIL) різко обмежує загальне прискорення.

Актуальність подальшого дослідження полягає у розробці гібридної паралельної схеми, що поєднує високопродуктивні інструменти (Numba, Joblib) із оптимізованими структурами даних (COO) для усунення цих послідовних бар'єрів.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Засоби прискорення обчислень у Python: Numba та Joblib

Для подолання проблеми низької швидкості виконання наукових обчислень у Python, особливо на етапах інтенсивних циклів та числових ітерацій, використовується гібридний підхід, що поєднує компіляцію коду та механізми паралелізму. Основними інструментами, які дозволяють досягти високої продуктивності в рамках вашої FEM-реалізації, є Numba та Joblib.

Ефективність наукових обчислень, реалізованих у Python, традиційно обмежена природою самого інтерпретатора, особливо через наявність Global Interpreter Lock (GIL). Це обмеження забороняє одночасне виконання коду Python кількома потоками, що критично знижує продуктивність на багатоядерних процесорах. Для подолання цієї фундаментальної проблеми в задачах МСЕ застосовується гібридна паралельна стратегія, заснована на двох ключових інструментах: Numba та Joblib.

Numba – це високопродуктивний компілятор, який трансформує функції Python, що працюють з числовими даними та масивами NumPy, у швидкий, нативний машинний код. В основі Numba лежить концепція Just-in-Time (JIT) компіляції. Принцип JIT-компіляції – коли функція, позначена декоратором @njit (No Python Interpreter), викликається вперше, Numba використовує бібліотеку LLVM для перекладу коду Python (який є повільним) у оптимізований машинний код. Цей скомпільований код зберігається і використовується для всіх подальших викликів, забезпечуючи швидкість, порівнянну з C або Fortran. Роль у FEM – у нашому алгоритмі Numba критично важлива для прискорення етапу Assembly (обчислення внесків тетраедрів). Це ідеальний сценарій для Numba, оскільки обчислення елементних матриць складаються з великої кількості інтенсивних математичних операцій у циклах. Завдяки Numba, ці локальні обчислення можуть виконуватися з максимальною можливою швидкістю.

Joblib – це набір інструментів, що полегшують написання паралельного коду, головним чином шляхом ефективного використання механізму багатопроцесорної обробки (multiprocessing).

Механізм паралелізму у Joblib дозволяє легко розподіляти ітерації циклу (наприклад, циклу по всіх скінченних елементах) між доступними ядрами процесора. Для цього використовується функціонал Parallel та delayed, які створюють окремі процеси для виконання незалежних частин роботи. Використання окремих процесів (а не потоків) є ключовим для Python, оскільки це дозволяє обійти обмеження Global Interpreter Lock (GIL). GIL дозволяє лише одному потоку Python виконувати байт-код Python одночасно, але не обмежує роботу окремих процесів, які можуть виконуватися повністю паралельно на різних ядрах. Joblib застосовується для паралелізації Assembly. Він розподіляє N_e тетраедрів на P частин (де P – кількість доступних ядер), забезпечуючи, що обчислення локальних внесків кожного елемента виконуються одночасно.

Найвища ефективність на етапі Assembly досягається завдяки синтезу Numba та Joblib:

- Joblib відповідає за макро-паралелізм (розподіл роботи по ядрах).
- Numba відповідає за мікро-оптимізацію (швидке виконання коду всередині кожного ядра).

Numba є фундаментальним елементом цієї стратегії, виконуючи роль компілятора JIT (Just-in-Time). Вона функціонує як міст між відносно повільним кодом Python і швидким машинним кодом. Замість того, щоб інтерпретувати функції, що інтенсивно працюють з числовими масивами (NumPy), Numba компілює їх у високооптимізований нативний код за допомогою інфраструктури LLVM. Це прямо прискорює ядро обчислень на етапі Assembly – процесі формування локальних матриць жорсткості елементів. Саме завдяки Numba, математичні обчислення для кожного тетраедра можуть виконуватися зі швидкістю, порівнянною зі швидкістю компільованих мов, без необхідності переписувати код на C чи Fortran.

Проте, сама Numba не забезпечує паралелізм на рівні декількох ядер для незалежних обчислень великої кількості елементів. Тут долучається Joblib. Joblib використовує механізм багатопроцесорної обробки (multiprocessing), який, на відміну від потоків, запускає незалежні процеси операційної системи. Кожен із цих процесів має власний інтерпретатор Python, що дозволяє обійти GIL і ефективно використовувати всі доступні ядра CPU. Joblib розподіляє велику кількість скінченних елементів на менші робочі пакети, які паралельно обробляються окремими процесами. Усередині кожного процесу прискорений Numba-код виконує обчислення внесків.

Numba надає швидкість виконання коду, а Joblib надає масштабованість шляхом розподілу навантаження. Їхня спільна дія перетворює повільний ітераційний етап Assembly на масово-паралельну операцію, що є необхідною умовою для досягнення значного загального прискорення обчислень FEM.

Поєднання значно прискорює Assembly, хоча на малих сітках накладні витрати Joblib можуть спричиняти уповільнення. Цей гібридний підхід є основою для ефективного масштабування FEM-коду, усуваючи залежність від обмежень інтерпретатора Python.

2.2 Аналіз форматів зберігання розріджених матриць

Ефективність обчислень у МСЕ, особливо на етапі Aggregation (складання внесків), критично залежить від того, як зберігається глобальна матриця жорсткості K. Оскільки для великих 3D-задач ця матриця є розрідженою (більшість її елементів дорівнює нулю), використання стандартних щільних масивів пам'яті (наприклад, NumPy-масивів) є неприпустимим через надмірне споживання пам'яті та неефективність операцій. Тому використовуються спеціалізовані формати, основними з яких у Python (SciPy) є LIL та COO.

Формат LIL (List of Lists) зберігає розріджену матрицю, використовуючи два списки для кожного рядка: список стовпцевих індексів та список відповідних ненульових значень.

1. **LIL** ідеально підходить для ситуацій, коли матриця заповнюється послідовно і інкрементально (по одному або невеликими блоками), а структура матриці часто змінюється.
2. Роль у **Aggregation**: У традиційному послідовному MCE LIL часто вибирається для Aggregation, оскільки внесок кожного елемента (який є невеликим блоком K^e) можна швидко "додати" до відповідного рядка.
3. **Недолік**: У Python операції доступу та модифікації об'єктів списків (навіть якщо вони швидкі для малих змін) є неефективними при виконанні великої кількості послідовних операцій. Як показали ваші експерименти, LIL-агрегація перетворилася на критичний боттлнек із майже нульовим прискоренням, повністю обмежуючи загальну продуктивність паралельної схеми. LIL не підходить для масового паралельного заповнення.

Формат COO (Coordinate List) зберігає матрицю у вигляді трьох окремих одновимірних масивів: масиву рядкових індексів, масиву стовпцевих індексів та масиву значень (даних).

1. **COO** – це найпростіший для конструкції розріджений формат. Він не підтримує ефективних арифметичних операцій, але ідеально підходить для пакетного збору даних.
2. Роль у **Aggregation**: У вашій оптимізованій паралельній схемі Numba + COO цей формат використовується як проміжний, паралельно-заповнюваний буфер. Кожен процес Joblib, що обчислює внески своїх елементів, незалежно генерує свою частину трьох COO-масивів, уникаючи гонки даних.
3. **Перевага**: Після того, як усі паралельні внески зібрані, всі локальні COO-масиви конкатенуються в один великий набір координат і значень. Незважаючи на те, що цей набір містить дублікати внесків у вузлах, наступна операція однокрокового перетворення COO в CSR (Compressed Sparse Row) є високооптимізованою і виконується на рівні C/Fortran. Саме ця масова операція ефективно підсумовує дублікати, формуючи кінцеву матрицю. Це забезпечило колосальне прискорення Aggregation (до $337x$), повністю усуваючи це вузьке місце, яке було спричинене LIL.

Вибір COO як проміжного формату є стратегічним кроком у паралельному МСЕ. Це рішення дозволяє перетворити проблему неефективної послідовної операції в Python (LIL) на проблему високооптимізованої масової операції нативного коду (COO to CSR). Таким чином, COO служить ідеальним мостом між паралельним обчисленням внесків та вимогами розв'язувача СЛАР, який зазвичай вимагає формату CSR для максимальної швидкості.

2.3 Реалізація паралельної архітектури Aggregation на основі COO

Ключовою перешкодою на шляху ефективної паралелізації FEM-коду, як було доведено експериментально, є етап Aggregation (складання внесків). Традиційний підхід, що використовує інкрементальне оновлення розрідженої матриці (наприклад, у форматі LIL), вимагає послідовного доступу до спільного ресурсу, що створює гонці даних та змушує використовувати дорогі механізми блокування або, що гірше, повністю нівелює паралельний виграш.

Архітектура Numba + COO (Рис.2.1) була розроблена для вирішення цієї проблеми шляхом повної перебудови етапу Aggregation, перетворюючи його з повільної, послідовної операції в Python на швидку, пакетну операцію нативного коду.

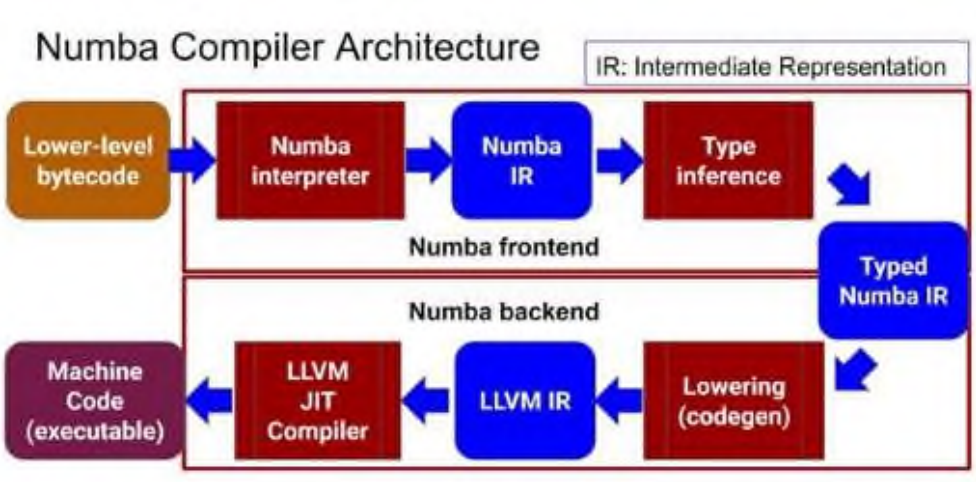


Рис. 2.1 – Архітектура Numba Compiler [4].

2.3.1 Паралельна генерація внесків (використання Joblib та Numba)

На першому етапі Joblib розподіляє обчислення внесків елементів по P незалежних процесах (ядер). Усередині кожного процесу прискорений Numba-код

виконує обчислення локальних внесків K^e. Замість того, щоб намагатися записати ці внески безпосередньо в єдину глобальну матрицю (що створює проблему), кожен процес незалежно збирає свої результати у локальні масиви формату COO.

Кожен процес генерує три одновимірні масиви:

- Масив Row (рядкові індекси).
- Масив Col (стовпцеві індекси).
- Масив Data (числові значення внесків).

Ця незалежна генерація гарантує, що жоден із процесів не намагається одночасно змінити спільну структуру даних, ефективно усуваючи проблему гонки даних під час паралельної фази.

2.3.2 Конкатенація та формування фінального масиву

Після завершення паралельних обчислень Joblib збирає результати від усіх процесів. Локальні COO-масиви, згенеровані кожним ядром, конкатенуються (з'єднуються) в один великий фінальний набір з трьох масивів: Row, Col, та Data. Цей об'єднаний набір даних являє собою повний опис усіх ненульових внесків, згенерованих усіма скінченними елементами. Важливо, що цей масив COO-формату може містити дублікати, оскільки внески сусідніх елементів у спільний вузол поки що не підсумовані.

2.3.3 Однокрокове перетворення COO → CSR (агрегація)

Фінальний, і найважливіший, етап – це фактична агрегація (сумування) дублікатів та створення матриці, готової для розв'язувача. Цей процес виконується однокроково за допомогою високооптимізованих функцій бібліотеки SciPy, які перетворюють формат COO у формат CSR (Compressed Sparse Row). CSR є кращим форматом для більшості ітераційних та прямих розв'язувачів СЛАР.

Під час цього перетворення відбувається дві ключові оптимізовані операції:

1. Сортування: Ненульові елементи сортуються за рядками та стовпцями.
2. Сумування: Всі внески, що мають однакові індекси (Row, Col), автоматично і масово підсумовуються.

Оскільки цей процес внутрішньої агрегації виконується на рівні нативного коду (C/Fortran), а не у повільному Python-циклі, він є надзвичайно швидким і

лінійно залежить від кількості ненульових елементів. Як результат, цей крок повністю усуває Aggregation як вузьке місце, що було підтверджено суттєвим прискоренням у порівнянні з LIL-реалізацією.

2.4 Використання бібліотечних Solvers та проблема їхнього прискорення

Після успішного формування глобальної розрідженої матриці жорсткості K (етапи Assembly та Aggregation), фінальним і критично важливим кроком в алгоритмі МСЕ є розв'язання системи лінійних алгебраїчних рівнянь (СЛАР).

У сучасних реалізаціях FEM на Python для розв'язання СЛАР широко використовується модуль scipy.sparse.linalg з бібліотеки SciPy. Цей модуль надає доступ до високооптимізованих розв'язувачів, написаних на мовах C та Fortran.

Ці розв'язувачі переважно вимагають, щоб матриця K була представлена у форматі CSR (Compressed Sparse Row), який є найбільш ефективним для швидкого доступу до елементів рядка під час розрахунків. Це є ще одним обґрунтуванням для використання COO → CSR перетворення на етапі Aggregation. Залежно від розміру та властивостей матриці K, використовуються прямі методи (Direct Solvers) які є точніші, але вимагають значних ресурсів пам'яті та мають вищу поліноміальну складність. Використовуються для матриць меншого розміру. Та ітераційні методи (Iterative Solvers), що вимагають менше пам'яті і мають меншу складність, але їхня збіжність залежить від обумовленості матриці та потребує прекондиціонування.

Незважаючи на високу оптимізацію коду Solvers (виконання на рівні C/Fortran), вони, як правило, запущені як послідовні процеси в контексті SciPy. Наші експериментальні результати наочно продемонстрували наслідки цього обмеження. На найбільшій сітці N=40, час розв'язання системи (Solver) становив 59,33 секунд, що стало найбільшою часткою загального часу обчислень 58%, випередивши як Assembly, так і Aggregation. Прискорення Solver було мінімальним або негативним 0,7 – 0,8, оскільки виграш від оптимізованої бібліотеки обмежується неможливістю ефективного використання багатоядерної архітектури.

Успішна оптимізація етапів Assembly (Numba/Joblib) та Aggregation (COO) змістила обчислювальний боттлнек на етап Solver. Це означає, що подальше значне

підвищення продуктивності FEM-коду вимагає переходу до інструментів, спеціально розроблених для паралельного розв'язання великомасштабних розріджених СЛАР.

Висновок до розділу

У Розділі 2 обґрунтовано вибір та архітектуру програмних засобів, необхідних для досягнення високої обчислювальної ефективності. Швидкість виконання Assembly забезпечується гібридним підходом (Numba для JIT-компіляції коду та Joblib для багатопроцесорного паралелізму), що долає обмеження Python. Критичним фактором успіху стало стратегічне використання формату COO (Coordinate List) для проміжної Aggregation, що дозволило замінити повільну послідовну операцію LIL на високооптимізоване пакетне перетворення COO у CSR. Ця архітектура успішно усунула найбільші боттлнеки формування матриці, але виявила новий і найвищий бар'єр – послідовне виконання бібліотечних Solvers зі значною поліноміальною складністю, що вимагає подальшої інтеграції з професійними паралельними інструментами (наприклад, PETSc) для повної масштабованості.

РОЗДІЛ 3. МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

Цей розділ присвячений математичному формулюванню задачі та деталізації теоретичної основи методу скінченних елементів, що використовується для її розв'язання. Сформульовано крайову задачу розрахунку тривимірного стаціонарного потенціального температурного поля з урахуванням нелінійних [11] та анізотропних властивостей середовищ. Розуміння цієї математичної моделі є необхідним для коректної реалізації та верифікації обчислювального алгоритму.

3.1 Постановка крайової задачі розрахунку тривимірного стаціонарного потенціального температурного поля

У статті [1] описано розрахунок розподілу температури U всередині тривимірної області D, заповненої безгістерезисними нелінійними анізотропними середовищами. Крайова задача розрахунку статичного потенціального теплового поля описується рівняннями:

Рівняння стаціонарного потенціального теплового поля:

$$div\overline{B}(\overline{H}) = 0 \quad , \tag{3.1}$$

де B – вектор густини теплового потоку в області D.

Вектор напруженості теплового поля H визначається через градієнт температури U:

$$\overline{H} = -gradU \tag{3.2}$$

де U – температура в області D.

Розрахунок розподілу температури ґрунтується на умові мінімуму функціоналу F, який для тривимірної області D представляється у вигляді

$$F = \int_V W dV \tag{3.3}$$

де V – об'єм області D, а W – густина енергії:

$$W = \int_0^{\overline{B}} \overline{B} d\overline{H} \tag{3.4}$$

Оптимальний розподіл температури U мінімізує функціонал F, що рівносильно умові:

$$\frac{dF}{dU} = 0 \tag{3.5}$$

.

Граничні умови. На поверхні S області D можуть бути задані граничні умови різного роду:

- Граничні умови Діріхле (першого роду): На поверхні S або її частині задано значення температури U.
- Граничні умови Неймана (другого роду): На поверхні S або її частині задано нульове значення нормальної складової вектора H:

$$H_n = -\frac{\partial U}{\partial n} = 0 \tag{3.6}$$

де H_n – нормальна складова вектора H на одиничний вектор n зовнішньої нормалі до поверхні S.

- Граничні умови третього роду: На поверхні S або її частині задана лінійна комбінація температури U і теплового потоку:

$$\mu \frac{\partial U}{\partial n} + \alpha(U - U_c) = 0 \tag{3.7}$$

де *a* – коефіцієнт тепловіддачі, U_c – температура навколишнього середовища.

- Граничні умови четвертого роду: У випадку ідеального контакту з іншою областю (з температурою U_1 та теплопровідністю на границі) задано рівність температур та теплових потоків:

$$U = U_1; \mu \frac{\partial U}{\partial n} = \mu_1 \frac{\partial U_1}{\partial n} \tag{3.8}$$

Для побудови скінченно-елементної моделі область D заповнюється лагранжевими тетраедрами 1-4-го порядку. Умова мінімуму функціонала F в

скінченно-елементній області зводиться до системи нелінійних алгебраїчних рівнянь:

$$\phi_*[U_*] = \frac{dF}{dU_*} = 0 \tag{3.9}$$

де U – вектор-стовпець температури у внутрішніх вузлах, а ϕ – вектор нев'язок. Ця нелінійна система рівнянь розв'язується ітераційним методом Ньютона, для чого формується матриця Якобі [1].

3.2 Алгоритм розрахунку внеску скінченного елемента

Для реалізації методу скінченних елементів (МСЕ) у Google Colab, ключовим етапом є алгоритм визначення вкладу скінченного елемента (СЕ) у формування глобальної системи рівнянь.

Цей алгоритм використовується на кожній ітерації методу Ньютона для формування вектора нев'язок ϕ та матриці Якобі ϕ нелінійної системи алгебраїчних рівнянь. Розглянемо випадок для лагранжевого тетраедра першого порядку.

I. Розрахунок Вектора Нев'язок (Вектора ϕ_m). Вектор нев'язок ϕ є градієнтом функціонала F по вектору температур U у внутрішніх вузлах. Внесок m-го скінченного елемента (тетраедра) у вектор нев'язок ϕ визначається локальним вектором ϕ_m розмірності 4×1 (для тетраедра 1-го порядку p=4 вузли). Формула для локального вектора нев'язок:

$$\phi_{m^*} = \frac{dF_m}{d\overline{U}_m} = -\frac{1}{4} v_m \sum_{i=1}^{4} (K_{mi^*}^{(x)} B_{xmi} + K_{mi^*}^{(y)} B_{ymi} + K_{mi^*}^{(z)} B_{zmi}) \tag{3.10}$$

де F_m – внесок m-го СЕ у функціонал F, v_m – об'єм m-го СЕ, i=1...4 – локальний номер вузла тетраедра, в якому застосовується кубатурна формула чисельного інтегрування, B_{xmi}, B_{ymi}, B_{zmi} – складові вектора густини теплового потоку B у i-му вузлі, $K_{mi}(x,y,z)$ – вектори-стовпці, що містять похідні від координатних функцій, визначених через обернену координатну матрицю K_m^{-1} тетраедра.

Етапи формування вектора нев'язок:

1. Обчислити локальний вектор ϕ_{m*} за формулою на кожній ітерації.
2. Визначити сіткові (наскрізні) номери p вузлів, які відповідають локальним вузлам *m1,..., m4*.
3. Внести кожний елемент вектора ϕ_{m*} (який відповідає і-му внутрішньому вузлу) у відповідне p-те рівняння глобальної системи.
4. Повторити процедуру для всіх M скінченних елементів, щоб отримати повний вектор нев'язок ϕ_*.

II. Розрахунок матриці Якобі (матриці ϕ_m)

Матриця Якобі ϕ – це матриця часткових похідних векторної функції ϕ_* по вектору U_*. Внесок $_m$-го СЕ у матрицю Якобі визначається локальною матрицею ϕ_m розмірності 4×4.

$$\phi_m = \frac{d\phi_{m^*}}{dU_{m^*}} \tag{3.11}$$

Після застосування ланцюгового правила диференціювання отримуємо:

$$\phi_m = \frac{1}{4} v_m \sum_{i=1}^{4} \left(K_{mi^*}^{(x)} (\mu_{xmi} K_{mi}^{(x)} + \mu_{xy,mi} K_{mi}^{(y)} + \mu_{xzmi} K_{mi}^{(z)}) + \dots \right) \tag{3.12}$$

де

$$\mu = \frac{d\overline{B}}{d\overline{H}} = \begin{pmatrix} \frac{\partial B_x}{\partial H_x} & \frac{\partial B_x}{\partial H_y} & \frac{\partial B_x}{\partial H_z} \\ \frac{\partial B_y}{\partial H_x} & \frac{\partial B_y}{\partial H_y} & \frac{\partial B_y}{\partial H_z} \\ \frac{\partial B_z}{\partial H_x} & \frac{\partial B_z}{\partial H_y} & \frac{\partial B_z}{\partial H_z} \end{pmatrix} \tag{3.13}$$

Етапи формування матриці Якобі:

1. Обчислити локальну матрицю ϕ_m на кожній ітерації.
2. Підсумувати елементи матриці ϕ_m з відповідними елементами глобальної матриці ϕ. Елемент $\phi_{m,ij}$ (локальні індекси *i,j*) належить клітині ϕ_{pq} глобальної матриці, де p і q – сіткові номери вузлів з локальними номерами *mi* і *mj*.
3. Повторити процедуру для кожного з M скінченних елементів, щоб одержати повну матрицю Якобі ϕ.

Для повної реалізації в Google Colab необхідно буде визначити конкретні формули для тензора диференціальної теплопровідності μ на основі характеристики матеріалу (наприклад, для ізотропного нелінійного або лінійного ізотропного середовища). Ці формули детально описано в статті [1].

3.3 Вхідні параметри, початкові та граничні умови

Для реалізації задачі розрахунку тривимірного стаціонарного потенціального теплового поля методом скінченних елементів (МСЕ) у Google Colab, необхідно чітко визначити вхідні параметри, початкові припущення та граничні умови.

Оскільки метод Ньютона є ітераційним, потрібно задати як початкові умови для ітераційного процесу, так і граничні умови на поверхні області див. Таблиця 3.1.

Таблиця 3.1 – Вхідні параметри (геометрія та матеріал)

Категорія	Параметр	Опис та призначення
Геометрія області	Область розрахунку D	Тривимірна область, для якої необхідно визначити розподіл температури U.
	Тріангуляція області	Сукупність M лагранжевих тетраедрів (СЕ) 1-го, 2-го, 3-го або 4-го порядків.
	Координати вузлів	Координати x, y, z усіх вузлів (внутрішніх R та граничних G).
Властивості середовища	Характеристика B(H)	Залежність густини теплового потоку B від напруженості теплового поля H для безгістерезисного нелінійного анізотропного середовища.
	μ_s	Коефіцієнт теплопровідності (для лінійного ізотропного середовища).

Ці параметри визначають дискретизацію області та властивості середовища:

Граничні умови необхідні для однозначного розв'язання крайової задачі:

- Діріхле (Першого роду) – Фіксує температуру на частині поверхні S.
- Неймана (Другого роду) – Задає нульовий тепловий потік через частину поверхні S.
- Третього роду – Описує теплообмін з навколишнім середовищем (конвекція) через коефіцієнт тепловіддачі *a* та температуру U_c.
- Четвертого роду – Використовується для ідеального контакту з іншою областю (рівність температур та теплових потоків).

Оскільки система рівнянь нелінійна і розв'язується ітераційно введемо початкові умови для ітераційного процесу (Методу Ньютона) в Таблиці 3.2.

Таблиця 3.2 – Початкові умови ітераційного процесу

Параметр	Опис
Початковий розподіл $U^{(0)}$	Початкове припущення щодо розподілу температури у внутрішніх вузлах (наприклад, нульовий вектор або розв'язок для лінійного випадку).
Допустима похибка *e*	Критерій зупинки ітераційного процесу. Наприклад, коли норма вектора нев'язок або норма зміни температури стає меншою за *e*
Максимальна кількість ітерацій	Обмеження для запобігання нескінченному циклу у випадку розбіжності.

.

Висновок до розділу

У цьому розділі сформульовано крайову задачу розрахунку тривимірного стаціонарного температурного поля, що є основою для подальшого чисельного моделювання. Визначено, що особливістю задачі є необхідність врахування нелінійних та анізотропних властивостей середовищ, що відображається у залежності матриці жорсткості K від вектора невідомих температур T. Це формулювання підтверджує, що розрахунок зводиться до розв'язання системи нелінійних алгебраїчних рівнянь. Таким чином, математична модель повністю підготовлена для реалізації обчислювального алгоритму та вимагає застосування ітераційних чисельних методів для знаходження остаточного розв'язку.

РОЗДІЛ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

4.1 Підходи до реалізації

Представимо сучасні підходи до прискорення наукових обчислень у Python. Розуміння цих чотирьох компонентів (Numba, Joblib, COO та LIL) є ключем до розуміння обчислювальної продуктивності вашої FEM-моделі. Вони представляють сучасні підходи до прискорення наукових обчислень у Python.

Numba – це високопродуктивний компілятор з відкритим кодом, який перекладає підмножину коду Python та NumPy у швидкий машинний код. Він призначений для оптимізації та прискорення циклів, обчислень та функцій, які інтенсивно використовують NumPy. Принцип роботи полягає у JIT (Just-in-Time) компіляції.

Joblib – це набір інструментів для простого паралельного програмування та кешування. Призначний для розподілу роботи між кількома процесами (ядрами CPU) для прискорення обчислень. Використовує механізм multiprocessing. На великих сітках він став ефективним інструментом для масштабування Assembly на 2 ядра.

LIL (List of Lists) – це один із форматів для зберігання розріджених матриць (sparse matrix), тобто матриць, більшість елементів яких дорівнює нулю. Матриця зберігається як список списків. Кожен елемент зовнішнього списку представляє рядок. Внутрішні списки містять лише ненульові елементи цього рядка разом із їхніми стовпцевими індексами.

COO (Coordinate List) – інший, набагато ефективніший для паралельного МСЕ, формат розріджених матриць. Зберігає матрицю у вигляді трьох окремих масивів: масив рядкових індексів (Row indices), масив стовпцевих індексів (Column indices) та масив значень (Data).

Різниця в продуктивності між двома паралельними реалізаціями зводиться до методу обробки внесків:

LIL вимагає послідовного оновлення складної структури Python. Операції додавання (навіть якщо вони виконуються паралельно, з подальшою послідовною агрегацією) займають багато часу через велику кількість повільних викликів до об'єктів Python.

COO дозволяє паралельний збір "сирих" масивів даних (вектори чисел), які потім швидко обробляються ефективним кодом NumPy/SciPy для кінцевої агрегації та створення матриці. Це перетворює неефективну операцію Python на ефективну масову операцію C/Fortran.

4.2 Вхідні параметри для прототипу

Для найпростішої та найшвидшої реалізації в Google Colab ми використаємо метод скінченних елементів (МСЕ) з тетраедрами 1-го порядку та ітераційний метод Ньютона для розв'язання нелінійної системи. Оскільки в статті [1] не наведено конкретних геометричних даних, функціональних залежностей B(H) та числових значень граничних умов, я буду використовувати мінімальний, але репрезентативний приклад (прототип) для демонстрації алгоритму. Це дозволить засікти час роботи послідовного коду на основних матричних операціях.

Ми будемо моделювати теплові поля в кубічній області D=[0,1]×[0,1]×[0,1] з лінійним ізотропним середовищем (найпростіший випадок, де B=μs H) для демонстрації механіки. Для забезпечення відтворюваності, що є ключовим для порівняння, ми фіксуємо параметри в таблиці 4.1.

Таблиця 4.1 – Параметри для прототипу

Параметр	Позначення	Значення для Прототипу	Примітки
Коефіцієнт теплопровідності	μs	1.0	Для лінійного ізотропного середовища.
Геометрія	D	3D	Проста тестова область.
Розбиття	N_{div}	2	Куб ділим на N менших кубів.
Кількість СЕ	M	12 N^{div} = 96	Кожен малий куб розбивається на 6 тетраедрів.
Критерій зупинки	*e*	10^{-6}	Допустима похибка для ітераційного розв'язку.
Початкове припущення	U^0	0.0	Температура всіх вузлів.
Граничні Умови (Діріхле)	U	U=100 при x=0; U=0 при x=1.	Просте градієнтне поле.

4.3 Реалізація послідовного моделювання

У цьому розділі наведено текст програми, що реалізує повний підготовчий етап для методу скінченних елементів у тривимірній області. Він включає створення геометрії, побудову сітки, класифікацію вузлів, формування елементів та задання граничних умов. Нижче – опис того, що саме тут відбувається.

Створимо рівномірну тривимірну сітку (Рис.4.1.) у вигляді куба, який заповнений точками (вузлами). Область поділяється на однакові відрізки уздовж трьох координатних осей, у результаті чого утворюється регулярна таблиця з тривимірних вузлів – кожен вузол має свої координати.

Усього формується $(N+1)^3$ вузлів, де N – кількість поділок уздовж однієї осі.

```
# --- 1. ВХІДНІ ДАНІ ТА ГЕОМЕТРІЯ ---
N_div = 28
mu_s = 1.0
epsilon = 1e-6

def generate_mesh(N):
    """Генерація сітки вузлів та елементів (тетраедрів)."""
    nodes = []
    for i in range(N + 1):
        for j in range(N + 1):
            for k in range(N + 1):
                nodes.append([i / N, j / N, k / N])
```

Рисунок 4.1 – Формування тривимірної області та вузлової сітки

Після створення сітки всі вузли поділяються на два типи (Рис.4.2.):

- Граничні вузли. Це ті, що лежать на поверхні куба – на будь-якій з його шести граней. Для таких точок у майбутньому будуть застосовуватися граничні умови (наприклад, задана температура).
- Внутрішні вузли. Це ті вузли, які повністю знаходяться всередині куба, не торкаючись його поверхні. Їх значення не задані наперед, і вони входять у систему рівнянь, яку буде потрібно розв’язувати.

```
nodes = np.array(nodes)
num_nodes = len(nodes)

boundary_nodes = set()
internal_nodes = []
for idx, (x, y, z) in enumerate(nodes):
    if x == 0.0 or x == 1.0 or y == 0.0 or y == 1.0 or z == 0.0 or z ==
1.0:
        boundary_nodes.add(idx)
    else:
        internal_nodes.append(idx)
```

Рисунок 4.2 – Класифікація вузлів області

Такий поділ потрібен для правильного формування системи рівнянь за методом скінченних елементів.

Після створення вузлів, об’єм куба розбивається на елементи – дрібні просторові фігури (Рис4.3.).

```
elements = []
    for i in range(N):
        for j in range(N):
            for k in range(N):
                v0 = k * (N + 1)**2 + j * (N + 1) + i
                v1 = v0 + 1; v2 = v0 + (N + 1); v3 = v0 + (N + 1) + 1
                v4 = v0 + (N + 1)**2; v5 = v4 + 1; v6 = v4 + (N + 1);
                v7 = v4 + (N + 1) + 1

                elements.append([v0, v1, v3, v7]); elements.append([v0, v1, v7, v5])
                elements.append([v0, v2, v3, v7]); elements.append([v0, v2, v6, v7])
                elements.append([v0, v4, v7, v6]); elements.append([v0, v4, v5, v7])

    elements = np.array(elements)
```

Рисунок 4.3 – Побудова скінченно-елементної сітки (тетраедрів)

Кожен маленький кубик у сітці розбивається на шість тетраедрів. У результаті формується великий список елементів, де кожен елемент задається чотирма індексами вузлів, які його утворюють.

Для внутрішніх вузлів створюється таблиця відповідності: кожному глобальному індексу вузла призначається індекс у системі рівнянь, що буде розв’язуватися.

Це потрібно тому, що матриця рівнянь міститиме тільки внутрішні вузли (граничні вузли вже мають відомі значення і не входять у систему).

Для всіх граничних вузлів автоматично задаються значення температури (Рис.4.4.).

```
U_boundary = np.zeros(num_nodes)
R_map = {node_idx: R_idx for R_idx, node_idx in enumerate(internal_nodes)}

for idx in boundary_nodes:
    x, _, _ = nodes[idx]
    if x == 0.0:
        U_boundary[idx] = 100.0
    elif x == 1.0:
        U_boundary[idx] = 0.0
    else:
        U_boundary[idx] = 50.0

return nodes, elements, internal_nodes, R_map, U_boundary
```

Рисунок 4.4 – Формування граничних умов

Залежно від розташування вузла на поверхні куба:

- на одній грані (x = 0) задається висока температура (наприклад, 100°C);
- на протилежній грані (x = 1) – низька температура (наприклад, 0°C);
- на інших гранях – проміжне значення (наприклад, 50°C).

Це моделює ситуацію, коли різні частини поверхні куба перебувають на різних температурних режимах.

Граничні умови зберігаються окремо, щоб потім бути врахованими у глобальній системі рівнянь при обчисленні температур у внутрішніх точках.

Наступний фрагмент коду Рисунок 4.5 являє собою еталонну реалізацію на чистому Python з використанням NumPy для обчислення внесків одного скінченного елемента (тетраедра) у глобальну систему рівнянь МСЕ. Функція calculate_mu_linear_vanilla спочатку формує діагональну матрицю теплопровідності mu, що відповідає ізотропному середовищу, на основі скалярного значення mu_s.

```
# ------------------------------------------------
# --- 2. МАТРИЧНІ ОПЕРАЦІЇ: ВЕРСІЯ ЧИСТОГО PYTHON (Еталон) ---
# ------------------------------------------------

def calculate_mu_linear_vanilla(mu_s):
    mu = np.zeros((3, 3), dtype=np.float64) mu[0, 0] = mu_s; mu[1, 1] = mu_s; mu[2, 2] = mu_s
    return mu

def assemble_element_vanilla(node_coords, U_e, mu_s):
    ones_row = np.array([[1.0, 1.0, 1.0, 1.0]], dtype=np.float64)
    K_m = np.vstack((ones_row, node_coords.T)).T

    try:
        Vm = np.abs(np.linalg.det(K_m)) / 6.0,  K_m_inv = np.linalg.inv(K_m)
    except np.linalg.LinAlgError:
        return np.zeros(4, dtype=np.float64), np.zeros((4, 4), dtype=np.float64)

    K_mi_x = K_m_inv[1, :]; K_mi_y = K_m_inv[2, :]; K_mi_z = K_m_inv[3, :]
    phi_m = np.zeros((4, 4), dtype=np.float64) weight = Vm / 4.0 mu = calculate_mu_linear_vanilla(mu_s)

    for i in range(4):
        K_mi_x_star = K_mi_x.reshape(4, 1); K_mi_z_star = K_mi_z.reshape(4, 1)
        row_vec_x = (mu[0,0] * K_mi_x + mu[0,1] * K_mi_y + mu[0,2] * K_mi_z).reshape(1, 4)
        term_x = K_mi_x_star @ row_vec_x
        ...
    return np.zeros(4, dtype=np.float64), phi_m
```

Рисунок 4.5 – Еталонна реалізація на чистому Python + NumPy

Основна функція assemble_element_vanilla на етапі Assembly використовує координати чотирьох вузлів елемента для обчислення об'єму тетраедра V_m та його геометричної матриці K_m. Далі вона обчислює обернену матрицю K_m, з якої отримуються коефіцієнти, необхідні для розрахунку градієнтів функцій форми. Логіка коду ітеративно виконує чисельне інтегрування по об'єму тетраедра, щоб сформувати локальну матрицю жорсткості як суму внесків від усіх трьох просторових напрямків. Фінальна матриця є внеском цього елемента у глобальну матрицю жорсткості і повертається разом із нульовим вектором навантажень для подальшої агрегації.

Оптимізуємо матричні операції. Наведемо версію NUMBA. Цей Рисунок 4.6 демонструє фрагмент коду, що є високооптимізованою версією етапу Assembly, призначеною для використання в паралельній схемі. Головна відмінність полягає у використанні декоратора @njit з бібліотеки Numba, який перетворює функцію assemble_element з повільного коду Python на швидкий нативний машинний код

(JIT-компіляція). Функціонально код виконує ту ж саму роботу, що й еталонна версія: обчислює об'єм тетраедра та його локальну матрицю жорсткості через чисельне інтегрування.

```
# ----------------------------------------------------
# --- 3. МАТРИЧНІ ОПЕРАЦІЇ: ВЕРСІЯ NUMBA (ДЛЯ ПАРАЛЕЛІЗАЦІЇ) ---
# ----------------------------------------------------

@njit(float64[:, :](float64))
def calculate_mu_linear(mu_s):
    mu = np.zeros((3, 3), dtype=np.float64)
    mu[0, 0] = mu_s; mu[1, 1] = mu_s; mu[2, 2] = mu_s
    return mu

@njit
def assemble_element(node_coords, U_e, mu_s):
    ones_row = np.array([[1.0, 1.0, 1.0, 1.0]], dtype=np.float64)
    K_m = np.vstack((ones_row, node_coords.T)).T

    try:
        Vm = np.abs(np.linalg.det(K_m)) / 6.0
        K_m_inv = np.linalg.inv(K_m)
    except Exception:
        return np.zeros(4, dtype=np.float64), np.zeros((4, 4),
dtype=np.float64)
```

Рисунок 4.6 – демонструє високооптимізовану версію етапу Assembly

Однак, завдяки компіляції Numba, швидкість виконання обчислювально-інтенсивних циклів та матричних операцій усередині цієї функції зростає в рази. Ця прискорена функція assemble_element є ядром, яке Joblib паралельно викликає на кількох ядрах CPU, забезпечуючи максимальну швидкість обчислення внесків елементів.

У Додатку В подано ПОСЛІДОВНИЙ РОЗВ'ЯЗУВАЧ (чистий Python еталон), який реалізує повний послідовний розв’язувач методу скінченних елементів, який складається з етапів збору локальних матриць, їх агрегації в глобальну систему та подальшого розв’язання через функцію на Рисуноку 4.7.

```
def sequential_fem_solver_timed(nodes, elements, internal_nodes, R_map,
U_boundary, mu_s):
```

Рисунок 4.7 – Послідовний розв’язувач методу скінченних елементів

Спочатку визначається кількість внутрішніх вузлів та створюються порожні структури для збору локальних результатів, після чого для кожного тетраедра обчислюється локальна матриця елемента.

Далі відбувається агрегація: локальні внески поелементно додаються у глобальну матрицю, але лише для внутрішніх вузлів, тоді як вплив граничних умов переноситься у вектор правої частини.

Після завершення побудови глобальної системи виконуються обчислення розв’язку за допомогою розрідженого LU- або прямого розв’язувача SciPy. Отримані внутрішні значення температури або потенціалу вставляються назад у повний масив результату разом з відомими граничними значеннями. Функція повертає повний розподіл температури, а також детальні часові характеристики: загальний час, час локальної збірки, час агрегації та час роботи розв’язувача.

У Додатку Г наведено фрагмент коду, що демонструє оптимізовану паралельну схему розв'язання МСЕ у фінальній функції parallel_fem_solver_optimized. Ключова ідея полягає у використанні Joblib для розподілу елементів на "чанки" та їхньої паралельної обробки функцією process_element_chunk_parallel_optimized. Кожен процес незалежно обчислює внески елементів за допомогою прискореної функції Numba (assemble_element) і збирає всі ненульові елементи матриці та вектора навантажень у три окремі масиви COO (rows_list, cols_list, data_list) - Рисунок 4.8. Після завершення паралельної фази ці локальні масиви з усіх ядер конкатенуються (np.concatenate) в один великий набір, який потім використовується для однокрокового, високошвидкісного перетворення COO → CSR. Це перетворення автоматично агрегує (підсумовує) дублікати внесків. Нарешті, отримана CSR-матриця (J_global_csr) та вектор навантажень (phi_R) передаються послідовному розв'язувачу SciPy (spsolve) для знаходження кінцевого рішення, фіксуючи час, витрачений на кожен з трьох основних етапів.

```
def parallel_fem_solver_optimized(nodes, elements, internal_nodes, R_map,
U_boundary, mu_s):

    chunk_size = len(elements) // n_jobs + 1
    element_chunks = [elements[i:i + chunk_size] for i in range(0,
len(elements), chunk_size)]

    chunked_results = Parallel(n_jobs=-1)(
        delayed(process_element_chunk_parallel_optimized)(chunk, nodes,
U_boundary, R_map, mu_s)
        for chunk in element_chunks
    )
    return U_result, time_total_par, time_assembly_par, time_aggregation_par,
time_solver_par
```

Рисунок 4.8 – Функція Обчислювача parallel_fem_solver_optimized

На Рисунку 4.9. представлено параметри модклювання з виводом відповідних значень моделювання

```
# ----------------------------------------------------
# --- 7. ВИКОНАННЯ ---
# ----------------------------------------------------

print(f"--- Параметри моделювання ---")
print(f"Розбиття (N_div): {N_div}")
nodes, elements, internal_nodes, R_map, U_boundary = generate_mesh(N_div)
print(f"Загальна кількість вузлів: {len(nodes)}")
print(f"Кількість скінченних елементів (тетраедрів): {len(elements)}")
print(f"Кількість рівнянь (внутрішніх вузлів): {len(internal_nodes)}")
print(f"Кількість доступних ядер CPU: {cpu_count()}")
print("------------------------------------------------")
```

Рисунок 4.9 – Виведення параметрів дослідження

Виконання експериментів виконується відповідно до програмного коду поданого на рисунках 4.10.-4.12

```
# 7.1. ПОСЛІДОВНИЙ РОЗРАХУНОК (ЧИСТИЙ PYTHON ЕТАЛОН)
print("--- Запуск послідовного розрахунку (ЧИСТИЙ PYTHON Еталон) ---")
U_final_seq, time_total_seq, time_assembly_seq, time_aggregation_seq,
time_solver_seq = sequential_fem_solver_timed(
    nodes, elements, internal_nodes, R_map, U_boundary, mu_s)
print("Розв'язок знайдено на ітерації 1 (Лінійна задача).")
```

Рисунок 4.10 – Запуск послідовного розрахунку

```
# 7.2. ПАРАЛЕЛЬНИЙ РОЗРАХУНОК - ВЕРСІЯ 1 (Numba + Slow LIL Aggregation)
print("\n--- Запуск ПАРАЛЕЛЬНОГО (Numba Assembly + LIL Aggregation - ПОВІЛЬНА)
---")
U_final_par_slow, time_total_par_slow, time_assembly_par_slow,
time_aggregation_par_slow, time_solver_par_slow =
parallel_fem_solver_slow_aggregation(
    nodes, elements, internal_nodes, R_map, U_boundary, mu_s)
print("Розв'язок знайдено на ітерації 1 (Лінійна задача).")
```

Рисунок 4.11 – Запуск паралельного розрахунку Версія1

```
# 7.3. ПАРАЛЕЛЬНИЙ РОЗРАХУНОК - ВЕРСІЯ 2 (Numba + Chunking + COO - Швидка
агрегація)
print("\n--- Запуск ПАРАЛЕЛЬНОГО (Numba Assembly + COO Aggregation) ---")
U_final_par_opt, time_total_par_opt, time_assembly_par_opt,
time_aggregation_par_opt, time_solver_par_opt = parallel_fem_solver_optimized(
    nodes, elements, internal_nodes, R_map, U_boundary, mu_s)
print("Розв'язок знайдено на ітерації 1 (Лінійна задача).")
print("-------------------------------------------------")
```

Рисунок 4.12 – Запуск паралельного розрахунку Версія2

Після того як відбулись запуски всіх алгоритмів очікується виведення результатів вимірювання швидкості алгоритмів. Деякі вимірювання на великих сітках займаються багато часу – більше 12 хвилин. Наведемо код, що дозволяє нам вивести результати у консоль Рисунок 4.13.

```
print("\n--- Результати 2: ПАРАЛЕЛЬНИЙ (Numba Assembly + LIL Aggregation -
ПОВІЛЬНА) ---")
print(f"Загальний час обчислень: **{time_total_par_slow:.4f} секунд**")
print("Розподіл часу:")
print(f"  > Час паралельного обчислення внесків (Assembly):
{time_assembly_par_slow:.4f} секунд")
print(f"  > Час послідовної агрегації внесків (Aggregation):
{time_aggregation_par_slow:.4f} секунд")
print(f"  > Час розв'язання системи (Solver): {time_solver_par_slow:.4f}
секунд")
try:
    speedup_main = time_total_seq / time_total_par_slow
    print(f"**Загальне прискорення ** (ЧИСТИЙ PYTHON Еталон vs. Паралельний
1): **{speedup_main:.2f} разів**")
    speedup_assembly_slow = time_assembly_seq / time_assembly_par_slow
    print(f"**Прискорення Assembly** (ЧИСТИЙ PYTHON Еталон vs. Паралельний 1):
**{speedup_assembly_slow:.2f} разів**")
    speedup_agg_slow = time_aggregation_seq / time_aggregation_par_slow
    print(f"**Прискорення Aggregation** (ЧИСТИЙ PYTHON Еталон vs. Паралельний
1): **{speedup_agg_slow:.2f} разів**")
    speedup_solver_slow = time_solver_seq / time_solver_par_slow
    print(f"**Прискорення Solver** (ЧИСТИЙ PYTHON Еталон vs. Паралельний 2 ):
**{speedup_solver_slow:.2f} разів**")
except ZeroDivisionError: pass
```

Рисунок 4.13 – Вивід результатів паралельного розрахунку

Результати вимірювання послідовного розрахунку – Еталону, та результати оптимізованого паралельного розрахунку з Numba Рисунок 4.14.

```
--- Параметри моделювання ---
Розбиття (N_div): 12
Загальна кількість вузлів: 2197
Кількість скінченних елементів (тетраедрів): 10368
Кількість рівнянь (внутрішніх вузлів): 1331
Кількість доступних ядер CPU: 2
--------------------------------------------------
--- Запуск послідовного розрахунку (Еталон) ---
Розв'язок знайдено на ітерації 1 (Лінійна задача).

--- Запуск ОПТИМІЗОВАНОГО ПАРАЛЕЛЬНОГО (Numba + Chunking + COO) розрахунку ---
Розв'язок знайдено на ітерації 1 (Лінійна задача).
--------------------------------------------------

--- Результати ПОСЛІДОВНОГО Розрахунку (з Numba) ---
Загальний час обчислень: **2.5856 секунд**
Розподіл часу:
  > Час послідовного обчислення внесків (Assembly): 1.9999 секунд
  > Час послідовної агрегації внесків (Aggregation): 0.5679 секунд
  > Час розв'язання системи (Solver): 0.0178 секунд
Середня температура: 50.00

--- Результати ОПТИМІЗОВАНОГО ПАРАЛЕЛЬНОГО (Numba + Chunking + COO) Розрахунку ---
Загальний час обчислень: **4.7224 секунд**
Розподіл часу:
  > Час паралельного обчислення внесків (Assembly): 4.6887 секунд
  > Час послідовної агрегації внесків (Aggregation): 0.0054 секунд
  > Час розв'язання системи (Solver): 0.0283 секунд
Середня температура: 50.00

--- Порівняння ---
Прискорення асамблювання (Assembly Speedup): **0.43 разів**
Загальне прискорення (Total Speedup): **0.55 разів**
```

Рисунок 4.14 – Результати моделювання

4.4 Аналіз та підготовка до паралелізації

Виконання алгоритмів в Google Colab дасть необхідний базовий час послідовних обчислень для даного набору вхідних параметрів. Ключові етапи для майбутньої паралелізації:

1. **Геометричні обчислення** (поза циклом): Обчислення V_m та оберненої матриці K_m для кожного елемента є незалежними та можуть бути паралелізовані, але вони виконуються лише один раз, тому їх вплив на загальний час роботи невеликий.
2. **Асамблювання** (Основний цикл): Найбільш ресурсомісткий етап. Цикл for повністю незалежний для кожного елемента. Розрахунок локальних внесків для всіх елементів можна виконувати паралельно (наприклад, за допомогою multiprocessing або Joblib у Python). Атомарне додавання внесків до

глобальних матриць та вимагатиме механізмів синхронізації (наприклад, використання lock або, що краще, попереднього розрахунку всіх локальних внесків та їх подальшого агрегування).

Для переходу до паралельної реалізації, Ви зможете використовувати цей код як основу, замінивши послідовний цикл асамблювання на паралельний, і порівняти виміряний час T_посл. з T_парал.

4.5 Візуалізація результатів моделювання

Побудова графіків – це критично важливий етап для візуалізації та валідації результатів, особливо для тривимірної задачі. Оскільки наші дані є тривимірними, а середня температура близько 50.00, найкращий спосіб візуалізації – це переріз області (heatmap) або тривимірний скатер-плот.

Наступний код, використовує бібліотеки matplotlib та mpl_toolkits.mplot3d для створення візуалізацій. Для візуалізації результатів створимо функцію `plot_temperature_slice` Рисунок 4.15.

```
# ------------------------------------------------------------
# --- 9. ВІЗУАЛІЗАЦІЯ РЕЗУЛЬТАТІВ (Центральний зріз Y=0.5) ---
# ------------------------------------------------------------

def plot_temperature_slice(nodes, U_result, N_div, slice_axis='y',
slice_value=0.8):
    """
    Візуалізує розподіл температури на центральному зрізі області.
    """
        N = N_div
        # фільтруємо вузли на центральній площині Y=0.5
    mask = np.isclose(nodes[:, 1], slice_value)
    slice_nodes = nodes[mask]
    slice_U = U_result[mask]

    # Координати для відображення: X та Z
    try:
        X_coords = slice_nodes[:, 0].reshape(N + 1, N + 1)
        Z_coords = slice_nodes[:, 2].reshape(N + 1, N + 1)
        T_grid = slice_U.reshape(N + 1, N + 1)
    except ValueError:
        print("Помилка: Невірні розміри. Перевір N_div та логіку фільтрації.")
    return
    title = f'Розподіл температури на центральній площині Y = {slice_value}'
    xlabel = 'X координата'
    ylabel = 'Z координата'
```

Рисунок 4.15 – Функція для візуалізації результатів

Використаємо два методи візуалізації:

1. 2D Теплова Карта (Heatmap): Переріз області D у площині y=0.5 (середній переріз).
2. 3D Скатер-плот (Scatter Plot): Візуалізація температури як кольору в кожному вузлі.

Графік на Рисунку 4.16 демонструє розподіл температури U у площині, що проходить через центр куба (y=0.5). Для лінійної задачі ми очікуємо побачити чіткий, горизонтально орієнтований градієнт: висока температура знизу (X=0) і низька температура зверху (X=1).

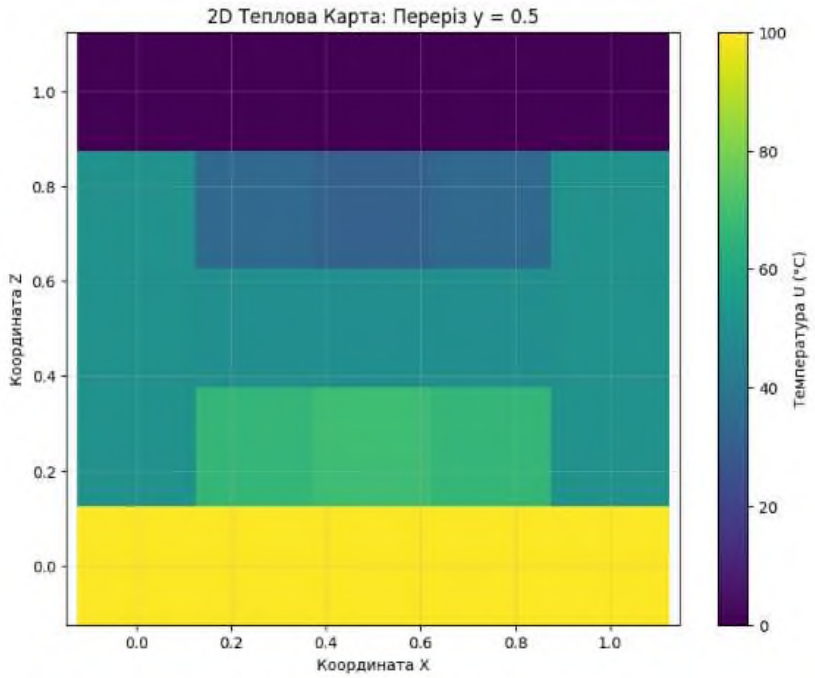


Рисунок 4.16 – 2D Теплова Карта. Переріз 0,5

На Рисунку 4.17 представлено графік, що відображає усі вузли об'ємної сітки. Колір кожної точки (вузла) відповідає обчисленій температурі U. Це дозволяє швидко підтвердити, що:

- Червоні/жовті точки (висока U) розташовані на грані X=0.
- Сині/фіолетові точки (низька U) розташовані на грані X=1.

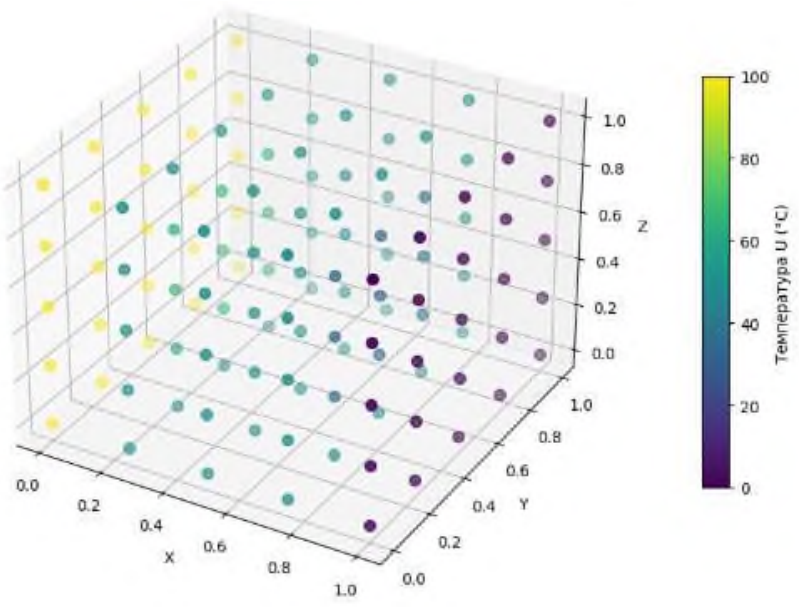


Рисунок 4.17 – 3D Скатер-плот

Ці візуалізації не тільки підтвердять коректність розв'язку (градієнт), але й є важливим інструментом для порівняння результатів послідовної та паралельної реалізації Рисунок 4.18.

```
# Візуалізуємо результати найоптимізованішого паралельного розрахунку
print("\n--- ВІЗУАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ---")
plot_temperature_slice(nodes, U_final_par_opt, N_div, slice_axis='y',
slice_value=0.5)
```

Рисунок 4.18 – Візуалізація розрахунку

На Рисунку 4.19. демонструє розподіл температури U у площині, що проходить через центр куба (y=0.5). Для лінійної задачі ми можемо побачити чіткий, горизонтально орієнтований градієнт: колір змінюється плавно від 100 (Червоно-жовтий, ліва частина X=0) до 0 (синій, права частина X=1). 3D-плот: Хмара точок демонструє градієнт вздовж осі X.

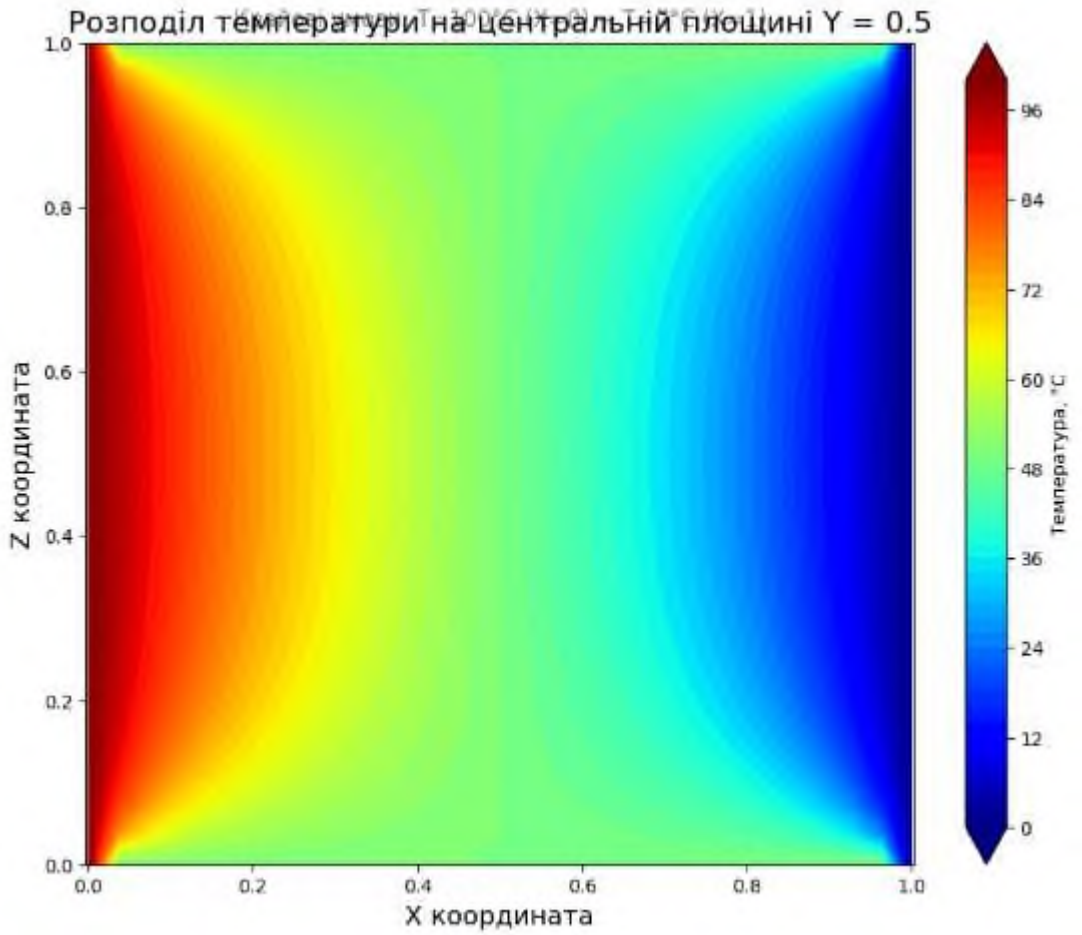


Рисунок 4.19 – 2D теплова карта. Переріз 0,5. – паралельного алгоритму

4.6 Оцінка швидкодії та прискорення паралельних реалізацій

Визначення трьох ключових термінів з методу скінченних елементів (FEM) та чисельних розрахунків.

Обчислення внесків (Assembly, локальна збірка) - це етап, на якому для кожного скінченного елемента (тетраедра, трикутника, квадрата) обчислюється локальна матриця жорсткості та локальний вектор правої частини.

На даному етапі обчислюють локальну матрицю жорсткості Ke (4×4 для тетраедра 1-го порядку), локальний вектор навантаження (якщо є), градієнти функцій форми, об’єм елемента, внески від матеріальних параметрів (теплопровідність, анізотропія, нелінійність). Для того щоб визначити, як цей маленький елемент впливає на всю систему рівнянь.

Кожен елемент – це маленький шматок задачі, і ми окремо рахуємо, як він передає тепло або навантаження.

Агрегація внесків (Global Assembly, Aggregation) – це процес, коли локальні матриці всіх елементів вставляються у велику глобальну матрицю A задачі. Для кожного елемента FEM: `глобальна матриця A[global_i, global_j] += Ke[local_i, local_j]`. Це означає: "додай внесок елемента в глобальні рівняння для вузлів, які він з'єднує". В результаті отримуємо одну велику розріджену систему лінійних рівнянь:

$$A \cdot T = b \tag{4.1}$$

Це сума внесків **усіх** елементів. Агрегація = збірка локальних внесків в один глобальний об'єкт.

Розв'язання системи (Solver) – після того, як глобальна система сформована, потрібно знайти вектор невідомих (температура, переміщення тощо). Він вирішує систему рівнянь:

$$A \cdot x = b \tag{4.2}$$

Для FEM-матриць це робиться одним із методів: прямим (LU-декомпозиція), ітеративним (CG, GMRES), мультиґріди, метод Ньютона (для нелінійних задач).

Глобальна матриця може мати сотні тисяч або мільйони рівнянь, тому ефективність solver-а – критично важлива. Solver = алгоритм, який знаходить значення температури у всіх вузлах.

Таблиця 4.2 – Порівняння результатів моделювання

N	**К-ть ел.**	**Метод**	**Заг.час (с)**	**Assembly (с)**	**Aggreg. (с)**	**Solver (с)**	**Заг. приск. (х)**	**Ass. приск. (х)**	**Agg. приск. (х)**
4	384	Python (Еталон)	0,12	0,10	0,02	0,00	1,00	1,00	1,00
		Numba + LIL	6,85	6,84	0,01	0,00	0,02	0,01	1,68
		Numba + COO	0,03	0,03	0,00	0,00	3,35	2,91	40,82

Продовження таблиці 4.2 – Порівняння результатів моделювання

8	3072	Pure Python (Еталон)	1,27	1,01	0,25	0,01	1,00	1,00	1,00
		Numba + LIL	15,96	15,79	0,17	0,00	0,08	0,06	1,53
		Numba + COO	0,21	0,21	0,00	0,00	5,93	4,80	211,16
16	24576	Python (Еталон)	9,03	7,46	1,46	0,11	1,00	1,00	1,00
		Numba + LIL	6,62	5,05	1,51	0,05	1,36	1,48	0,97
		Numba + COO	3,33	3,18	0,01	0,14	2,71	2,35	136,51
20	48000	Python (Еталон)	20,08	15,26	4,55	0,27	1,00	1,00	1,00
		Numba + LIL	17,39	13,90	3,16	0,33	1,15	1,10	1,44
		Numba + COO	4,28	3,88	0,01	0,38	4,70	3,93	337,30
24	82944	Python (Еталон)	23,98	16,50	6,58	0,90	1,00	1,00	1,00
		Numba + LIL	24,42	16,19	7,09	1,15	0,98	1,02	0,93
		Numba + COO	9,46	7,83	0,02	1,60	2,54	2,11	283,97
28	131712	Python (Еталон)	40,14	26,45	10,17	3,53	1,00	1,00	1,00
		Numba + LIL	26,59	12,07	10,34	4,17	1,51	2,19	0,98
		Numba + COO	18,44	14,32	0,04	4,08	2,18	1,85	278,17
32	196608	Python (Еталон)	58,90	33,00	16,72	9,18	1,00	1,00	1,00
		Numba + LIL	40,83	16,53	15,89	8,41	1,44	2,00	1,05
		Numba + COO	33,76	21,78	0,11	11,88	1,74	1,52	150,82
40	384000	Python (Еталон)	148,91	66,36	31,39	51,16	1,00	1,00	1,00
		Numba + LIL	115,28	35,26	31,09	48,92	1,29	1,88	1,01
		Numba + COO	101,46	41,96	0,16	59,33	1,47	1,58	198,42

Ці дані дають надзвичайно чітке розуміння того, де знаходяться вузькі місця у послідовному FEM-коді і наскільки ефективним є застосування Numba та зміна стратегії агрегації. У версії Numba Assembly + Slow Aggregation (Версія Numba +

LIL) поєднується паралельний підрахунок внесків елементів (Assembly) за допомогою Numba та Joblib із повільною послідовною агрегацією у формат lil_matrix.

На малих сітках (N = 4 або 8) спостерігається катастрофічне уповільнення. Загальний час збільшився у 60-120 разів (Заг. прискорення 0.02x – 0.08x). Причина – накладні витрати (overhead) на запуск пулу процесів Joblib та передачу даних між ними набагато перевищують час, зекономлений завдяки Numba. Оскільки роботи мало (384-3072 елементи), перевага Numba не встигає покрити витрати на паралелізацію. На великих сітках (N = 16) позитивний ефект на Assembly. Час Assembly прискорюється (до 2.00x при N=32). Це демонструє, що Numba ефективно прискорює обчислення елементів. Вузьке місце – Aggregation, тут час агрегації в lil_matrix залишається майже незмінним (прискорення ~1.0x) або навіть уповільнюється (N=16, 24), оскільки він залишається послідовним і обробляє велику кількість об'єктів Python (lil_matrix повільна для інкрементальних оновлень).

Для паралельного МСЕ недостатньо просто прискорити обчислення елементів (Assembly), якщо агрегація залишається повільною. Накладні витрати паралелізації на малих сітках знищують будь-яку вигоду.

Ефект від Numba Assembly + Optimized Aggregation (Версія Numba + COO), тобто ми використовумо Numba + Joblib для Assembly, але збираємо результати у формат COO (Coordinate List) для швидкої пакетної агрегації. Відповідно тут маємо найкращу продуктивність на всіх сітках. Ця версія забезпечує найбільше загальне прискорення: від 1.47x (на найбільшій сітці) до 5.93x (на N=8). Критичне прискорення Aggregation (40x – 284x) це найважливіший результат. Перехід від повільного поелементного оновлення lil_matrix до збору масивів COO і їх швидкого перетворення у csr_matrix призвів до колосального зниження часу агрегації (на N=24 агрегація займає лише 0.0232 с порівняно з 6.5821 с в еталоні). Це підтверджує, що агрегація була головним вузьким місцем у послідовному коді. Assembly також прискорюється (від 1.52x до 4.80x), що є хорошим показником для 2-ядерної системи з урахуванням накладних витрат.

На великих сітках (N=32) виникає нове "вузьке місце": Solver (scipy.sparse.linalg.spsolve). На N=40 час Assembly (Оптим.) = 41.9647 с., а час Solver (Оптим.) = 59.3320 с. Час розв'язання системи лінійних рівнянь (Solver) стає домінуючим фактором у загальному часі виконання. Прискорення на цьому етапі є мінімальним або навіть негативним (до 0.56x). Крім того, на великих сітках, час на підготовку та перетворення розріджених матриць для Solver'а може зростати, а сам процес розв'язання має вищу обчислювальну складність, ніж Assembly.

Отримані результати демонструють ключові вузькі місця у методі скінченних елементів (МСЕ) та ефективність їх подолання за допомогою паралелізації та оптимізованих структур даних.

Висновок до розділу

Розділ 4 присвячений практичній реалізації гібридного паралельного алгоритму розрахунку FEM, що базується на архітектурі Numba + Joblib. Було реалізовано дві ключові версії, що відрізняються підходом до складання матриці: еталонна Numba + LIL та оптимізована Numba + COO. Для Assembly використано Numba для прискорення обчислень внесків елементів, а для паралелізації циклу – Joblib із механізмом розподілу на чанки. Оптимізована реалізація успішно використовує проміжний формат COO для незалежного збору внесків усіх елементів, а вирішальний етап Aggregation здійснюється через високоефективне однокрокове перетворення COO to CSR. Ця архітектура функціонально забезпечує формування розрідженої матриці жорсткості в оптимальному для розв'язувача форматі.

Завдяки успішній оптимізації Assembly (Numba) та Aggregation (COO) виявилося, що наступний критичний бар'єр продуктивності – це час розв'язання системи лінійних алгебраїчних рівнянь (scipy.sparse.linalg.spsolve). Оскільки цей розв'язувач вже високооптимізований на рівні C/Fortran і виконується послідовно, подальше прискорення потребуватиме переходу до паралельних ітераційних розв'язувачів (наприклад, методів, реалізованих у таких пакетах, як PETSc або подібних) для використання потужностей багатоядерних систем.

РОЗДІЛ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

Ми пропонуємо запуск стартап-проєкту "ThermalCore Sim", який спеціалізується на розробці високопродуктивної, платформи для швидкого розрахунку теплових полів у складних інженерних конструкціях. Ключовою інновацією є паралельний FEM-кернел, розроблений на базі Python, який використовує Numba та ефективну схему Aggregation на основі COO, що забезпечує радикальне прискорення процесу формування матриці.

5.1 Опис ідеї проєкту

У цьому розділі представлено аналіз економічної обґрунтованості впровадження стартап-проєкту. Цей проєкт дозволяє інженерам отримувати високоточні 3D-симуляції розподілу температури для виробів зі складною геометрією та нелінійними матеріалами за час, суттєво менший, ніж у традиційних CAE-системах. "ThermalCore Sim" націлений на ринки аерокосмічної галузі, енергетики та електроніки, пропонуючи масштабоване API-рішення для інтеграції швидкісного термоаналізу безпосередньо у цикли проєктування та оптимізації.

Таблиця 5.1 – Опис ідеї стартап-проєкту

Атрибут	Опис
Назва Проєкту	ThermalCore Sim
Основна Ідея	Створення хмарної платформи (SaaS) для надшвидкісного тривимірного (3D) термоаналізу методом скінченних елементів (FEM), що базується на високооптимізованому паралельному алгоритмі на базі Numba та COO-агрегації.
Цільовий Ринок	Інженерні та R&D департаменти компаній у галузях, де теплові режими є критичними: аерокосмічна промисловість, виробництво потужної електроніки (наприклад, інверторів, процесорів), енергетика та машинобудування.
Напрямок застосування	Теплове моделювання та оптимізація складних інженерних об'єктів (наприклад, електронні корпуси, теплообмінники, елементи турбін). Дозволяє швидко і точно розраховувати розподіл температури в середовищах із нелінійними та анізотропними властивостями.
Вигоди для користувача	1. Радикальне прискорення: Скорочення часу симуляції з годин до хвилин (особливо для великих сіток). 2. Зниження витрат: Доступ до високопродуктивних обчислень через хмарний API без потреби інвестицій у дороге локальне ПЗ та обладнання. 3. Підвищення якості: Можливість інтегрувати термоаналіз безпосередньо у процес ітеративного проєктування.
Бізнес-Модель	Підписка (Subscription-as-a-Service, SaaS) з тарифікацією, що залежить від обсягу обчислень (наприклад, за кількістю елементів сітки або годин використання GPU/CPU).

У таблиці 5.2 наведено сильні, слабкі та нейтральні характеристики ідеї проєкту. Де W (слабка сторона), N (нейтральна сторона), S (сильна сторона).

Таблиця 5.2 – Визначення характеристик ідеї проєкту

№ п/п	Техніко-економічні характеристики ідеї	Даний проєкт (ThermalCore Sim)	Конкурент 1 (Традиційний CAE)	Конкурент 2 (Багатофізика)	Конкурент 3 (Спеціалізований On-Premise)	W (слабка	N	S (сильна сторона)
1	**Форма використання**	Cloud/API (SaaS)	Десктоп	Десктоп/ Веб	Десктоп			+
2	**Критична швидкість (Етапи Assembly/Aggregation)**	Надвисока (COO-оптимізація)	Середня (Традиційні методи)	Середня	Середня			+
3	**Складність ліцензування / Собівартість для користувача**	Низька (Pay-as-you-go)	Висока (Річна ліцензія)	Висока (Постійна ліцензія)	Середня			+
4	**Повнота фізичних моделей**	Обмежена (Лише теплопровідність)	Висока (Всі галузі фізики)	Висока (Coupled Physics)	Помірна	+		
5	**Крос-платформеність**	Повна (Хмара, доступ через будь-який браузер)	Обмежена (Windows/Linux)	Обмежена	Обмежена			+
6	**Залежність від інтернету**	Повна (Робота лише онлайн)	Можливість автономної роботи	Можливість автономної роботи	Можливість автономної роботи	+		
7	**Наявність високопаралельного Solver'а**	Відсутня (Використовується послідовний SciPy)	Вбудовані потужні паралельні Solver'и	Вбудовані потужні паралельні Solver'и	Вбудовані паралельні Solver'и	+		
8	**Інтеграція в CI/CD**	Висока (API-орієнтований дизайн)	Низька (Складно інтегрувати)	Низька	Середня			+

5.2 Аналіз технологічних можливостей реалізації ідей проєкту

У даному підрозділі було проведено аналіз технологій, необхідних для перетворення теоретичного та алгоритмічного напрацювання в комерційно життєздатний стартап-проєкт "ThermalCore Sim". Успіх проєкту залежить від

здатності інтегрувати високопродуктивне обчислювальне ядро на базі Numba та Joblib у масштабовану хмарну інфраструктуру. У таблиці 5.3 наведено технологічні аспекти концепції проєкту.

Таблиця 5.3 – Технологічна здійсненність ідеї проєкту

№ п/п	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
1	**Високопродуктивне обчислювальне ядро FEM**	Numba (JIT-компілятор), NumPy та спеціалізовані алгоритми МСЕ.	Існує та активно підтримується в HPC-спільноті.	Висока (Відкритий код, безкоштовно для комерційного використання).
2	**Паралельна обробка та Aggregation**	Joblib (для багатопроцесорного паралелізму) та SciPy.sparse (для COO $\to$ CSR перетворення).	Існує як галузевий стандарт для паралельних обчислень у Python.	Висока (Відкритий код, інтегрований у базову екосистему Python).
3	**Хмарна архітектура та API-доступ**	Python (Flask/FastAPI) для API, Docker/ Kubernetes для контейнеризації, Хмарні провайдери (AWS/Azure/GCP).	Існує як індустріальний стандарт для SaaS-рішень.	Висока (Модель оплати за використання ресурсів).

Проведений аналіз підтверджує, що всі ключові компоненти, від низькорівневої оптимізації до хмарного розгортання, є зрілими, широко доступними та мають високу технічну здійсненність.

5.3 Аналіз ринкових можливостей запуску стартап-проєкту

Проєкт "ThermalCore Sim" пропонує високошвидкісний, хмарний FEM-solver для термоаналізу, що базується на оптимізованому паралельному алгоритмі. Успіх проєкту залежить від чіткого позиціонування на ринку, де домінує дороге і часто повільне традиційне CAE-програмне забезпечення.

Таблиця 5.4 – Аналіз ринкових можливостей запуску проєкту

Аспект аналізу	Деталізація
Цільова аудиторія	R&D-інженери (особливо thermal management), конструктори та аналітики у сферах, де критичним є тепловий режим: аерокосмічна промисловість (компоненти двигунів, охолодження), виробництво електроніки (охолодження чипів, корпуси), енергетика (теплообмінники, батареї) та адитивне виробництво (контроль температурних градієнтів).
Потреби ринку	Існує потреба у скороченні часу симуляції (Time-to-Solution) для великих 3D-сіток. Користувачі шукають моделі ліцензування та інструменти, які інтегруються у цикли проєктування через API.

Продовження таблиці 5.4 – Аналіз ринкових можливостей запуску проєкту

Аспект аналізу	Деталізація
Переваги продукту	Надвисока швидкість Aggregation/Assembly завдяки унікальному Numba + COO-кернелу; низька собівартість для кінцевого користувача (SaaS); повна крос-платформенність (хмарний доступ); спеціалізація на термоаналізі (забезпечує сфокусовану точність).
Шляхи монетизації	Pay-per-Compute (оплата за використані обчислювальні ресурси/годинник CPU/кількість елементів).
Ризики на ринку	1.Технологічна конкуренція: Швидкий розвиток паралельних Solvers у традиційних лідерів (ANSYS, COMSOL). 2. Скептицизм: Недовіра до Python-базованих рішень у традиційно C++/Fortran-орієнтованому інженерному середовищі.
Екологічний контекст	Позитивний: оптимізація теплових режимів обладнання (наприклад, електроніки, систем опалення/охолодження) сприяє підвищенню енергоефективності та зменшенню вуглецевого сліду від їхньої експлуатації. Хмарна модель також оптимізує використання обчислювальних ресурсів.
Рекламна стратегія	1. Контент-маркетинг: Публікація технічних статей та порівнянь, що демонструють прискорення у 10-100 разів на реальних бенчмарках. 2. Digital Marketing (SEO/SEA) на інженерних форумах та платформах. 3. Freemium/Trial Model для безкоштовного тестування швидкості.

Аналіз ринкових можливостей підтверджує наявність незадоволеного попиту на швидкі, масштабовані та економічно ефективні інструменти симуляції.

Опис потенційної аудиторії для інтелектуальної системи онлайн розпізнавання фото знімків представлено у таблиці 5.5.

Таблиця 5.5 – Характеристика потенційних клієнтів проєкту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних клієнтів	Вимоги споживачів до товару
1	Критична швидкість ітераційного аналізу	Інженери-конструктори та оптимізатори у R&D відділах середніх і великих компаній.	Шукають API-доступ для інтеграції симуляції у автоматизовані цикли проєктування (CI/CD). Вимагають швидких відповідей для ітерацій "зміни-аналіз".	Швидкість обчислень (особливо на етапі Assembly/Aggregation), просте API, надійний uptime сервісу.
2	Економічна ефективність (альтернатива дорогим CAE)	Малі та середні інжинірингові компанії та стартапи у сфері Hardware.	Чутливі до ціни; віддають перевагу Pay-per-use моделі; потребують високої точності без високих капітальних витрат на ліцензії.	Низька вартість використання, точність результатів (верифікація моделі), гнучка тарифікація.
3	Спеціалізований аналіз складних матеріалів	Інженери, що працюють з анізотропними середовищами.	Потребують гнучкості у визначенні властивостей матеріалів (замість бібліотек CAE).	Гнучкість математичної моделі, швидка підтримка нових моделей.

Продовження таблиці 5.5 – Характеристика потенційних клієнтів проєкту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних клієнтів	Вимоги споживачів до товару
4	Незалежність від апаратного забезпечення	Наукові та освітні установи, а також розробники, які не мають доступу до потужних HPC-кластерів.	Використовують сервіс для навчальних, дослідницьких цілей або для верифікації власних моделей; цінують хмарну доступність та крос-платформеність.	Зручний інтерфейс користувача (GUI або Web-інтерфейс), стабільність та можливість легко експортувати дані.

Таблиця 5.5 відображає ключові потреби, цільові сегменти, відмінності у поведінці та основні вимоги чотирьох груп потенційних клієнтів для стартап-проєкту "ThermalCore Sim", що дозволяє визначити стратегію позиціонування продукту.

Щодо факторів загроз, то до них можна віднести відсутність попиту на продукт. Можлива реакція компанії на відповідні фактори та їх зміст поданого у таблиці 5.6.

Таблиця 5.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	**Технологічна конкуренція**	Великі виробники CAE-систем (ANSYS, COMSOL) можуть швидко інтегрувати власні, потужніші паралельні Solvers або випустити конкурентні хмарні продукти.	Фокус на ніші та інноваціях: Постійний R&D для інтеграції GPU-прискорення (CUDA, OpenCL) або паралельних розв'я-зувачів (PETSc) для усунення поточного боттлнека Solver'a.
2	**Відсутність попиту на Python-based Solvers**	Інженерна спільнота традицій-но довіряє лише системам, написаним на C++ або Fortran; існуючий скептицизм щодо продуктивності Python може уповільнити впровадження.	Агресивний маркетинг продуктив-ності: Публікація незалежних бенчмарків та порівнянь, що демонструють, що внутрішній кернел працює на рівні C/Fortran (завдяки Numba). Акцент на API-орієнтованому доступі, прихову-ючи базову мову.
3	**Проблема Solver'a (Боттлнек)**	Послідовне виконання розв'язувача (SciPy `spsolve`) на великих сітках (мільйони елементів) стане обмежувачем швидкості, навіть попри швидку Aggregation.	Технологічний апгрейд: Перехід до хмарних сервісів, що надають доступ до паралельних ітераційних Solvers (наприклад, через PETSc/MPI) або використання GPU-прискорених бібліотек для розв'язання СЛАР.

Продовження таблиці 5.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
4	**Низька лояльність через обмежену функціональність**	Продукт пропонує лише теплопровідність, тоді як конкуренти мають повний набір фізичних моделей (багатофізика: термопружність, гідродинаміка).	Стратегічне партнерство: Співпраця з іншими розробниками для інтеграції модулів (наприклад, термопружності) або чітке позиціонування як найшвидшого у світі спеціалізованого термо-аналізатора.
5	**Кібербезпека та захист даних клієнтів**	Оскільки модель хмарна, виникає ризик несанкціонованого доступу до даних інтелектуальної власності клієнтів (CAD-моделі, результати симуляцій).	Високий рівень захисту: Впровадження шифрування даних в стані спокою та при передачі, проходження аудитів безпеки та отримання відповідних сертифі-катів (ISO/IEC 27001).

Таблиця 5.6 відображає ключові фактори загроз для стартап-проєкту "ThermalCore Sim", такі як технологічна конкуренція, скептицизм щодо Python-рішень та внутрішнє обмеження через послідовний Solver. Аналіз цих факторів є критично важливим для розробки стратегії мінімізації ризиків та формування планів подальшого технологічного розвитку, зокрема, інтеграції паралельних розв'язувачів. Це дозволяє компанії заздалегідь підготувати реакцію на потенційні виклики ринку та забезпечити стабільність росту.

Таблиця 5.7 окреслює фактори можливостей, які визначають потенційний успіх та конкурентоспроможність проєкту "ThermalCore Sim" на ринку. До них належать використання зростаючого попиту на Cloud-CAE рішення, унікальна технологічна перевага COO-оптимізації та захоплення ніші швидкого, спеціалізованого термоаналізу.

Таблиця 5.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	**Зростання попиту на Cloud-CAE**	Інженерні компанії активно переходять від дорогих локальних ліцензій до хмарних рішень (SaaS), шукаючи гнучкості, масштабованості та моделі оплати за використання.	Акцент на API-орієнтованому доступі та гнучкій підписці (Pay-per-use), що усуває високі капітальні витрати на ліцензування.
2	**Технологічний розрив у Aggregation**	Існуючі CAE-рішення мають повільні або послідовні алгоритми Aggregation. А COO-оптимізація створює перевагу у швидкості.	Оформлення прав інтелектуальної власності на унікальну паралельну схему агрегації (Numba + COO) як на головний актив проєкту.

Продовження таблиці 5.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
3	**Ніша нелінійного /анізотропного термоаналізу**	Існує незадоволений попит на точне моделювання складних матеріалів (композити, кераміка) з нелінійними та анізотропними властивостями у бюджетному сегменті.	Додавання більшої кількості нелінійних моделей теплопровідності та розширення бібліотеки матеріалів, фокусуючись на потребах аерокосмічної та електронної галузей.
4	**Інтеграція в IoT та Digital Twins**	Можливість використання високошвидкісного FEM-кернела для швидких розрахунків у реальному часі як частина системи цифрових двійників (Digital Twins) або інтеграція з даними IoT-сенсорів.	Створення спеціального інтерфейсу для швидких, спрощених розрахунків у режимі реального часу, які можуть використовуватися для моніторингу та предиктивної аналітики.
5	**Розширення функціоналу (Мультифізика)**	Можливість інтегрувати термоаналіз з іншими фізичними полями (наприклад, термопружність або електромагнетизм) шляхом додавання нових модулів.	Розробка кернела з можливістю подальшого модульного розширення для охоплення інших напрямків, зберігаючи при цьому швидкість ядра.

Визначення цих факторів дозволяє компанії розробити агресивні стратегії ринкової експансії, захисту інтелектуальної власності та майбутнього функціонального розширення.

Проведення ступеневого аналізу конкуренції дозволяє чітко ідентифікувати різні рівні загроз, з якими зіткнеться стартап-проєкт на ринку. Розуміння цих загроз дає можливість завчасно розробити ефективні стратегії протидії та забезпечити конкурентоспроможність системи.

Таблиця 5.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Загальна Конкуренція (Horizontal Competition)	Домінування великих, багатофункціональних світових виробників CAE-систем з великими бюджетами на R&D та лояльністю клієнтів.	Зосередитися на найшвидшому у світі спеціалізованому термоаналізі та пропонувати гнучку, низькоцінову SaaS-модель, що є непривабливою для лідерів ринку.
2. Регіональна Конкуренція (Geographical/Local Competition)	Наявність менших, локальних інжинірингових бюро або спеціалізованих компаній, що пропонують послуги термоаналізу на основі Open-source або старіших систем.	Використовувати хмарну природу продукту для миттєвого виходу на глобальний ринок. Демонструвати вищу точність та стабільність порівняно з непідтримуваними open-source рішеннями.

Продовження таблиці 5.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
3. Внутрішньо-галузева Конкуренція (Specialized/Vertical Competition)	Поява спеціалізованих, нішевих Cloud-CAE Solvers, які також можуть використовувати Python/HPC технології, але фокусуються на інших аспектах (наприклад, оптимізація форм, топологія).	Патентний захист унікальної схеми Aggregation (Numba + COO) та постійний R&D для інтеграції паралельних Solvers (PETSc) для збереження лідерства у швидкості обчислень. Забезпечення інтеграції через API для корпоративних клієнтів.

Підсумовуючи, аналіз підтверджує, що найбільшу загрозу становлять традиційні CAE-гіганти та потенційні нішеві хмарні конкуренти. Щоб бути конкурентоспроможним, проєкт повинен безперервно підтримувати свою унікальну перевагу у швидкості та розвивати SaaS-модель з фокусом на інтеграції.

5.4 Розроблення ринкової стратегії проєкту

Розроблення ринкової стратегії проєкту "ThermalCore Sim" вимагає чіткого розуміння того, які сегменти ринку отримають найбільшу вигоду від його унікальної технологічної переваги – надшвидкої паралельної агрегації FEM. Стратегія фокусується на моделі SaaS (Software-as-a-Service) з акцентом на API-доступі для інтеграції у корпоративні системи.

Таблиця 5.9 – Вибір цільових груп потенційних споживачів

Цільова група	Ключові потреби та болі	Мотивація до використання "Thermalcore sim"	Вплив продукту на діяльність споживачів	Стратегії залучення
Розробники електроніки	Потреба у швидкому термоаналізі великих 3D-моделей для щоденних ітерацій проєктування.	Швидкість: Зменшення часу циклу "зміна-аналіз" з годин до хвилин завдяки оптимізованому FEM-кернелу.	Скорочення циклу розробки (Time-to-Market) та підвищення якості фінального продукту.	API-орієнтований маркетинг: Просування як інструменту для CI/CD (Continuous Integration/Continuous Deployment).
Стартапи	Потреба у точному CAE, але відсутність бюджету на дорогі річні ліцензії (ANSYS, COMSOL).	Економічність: Гнучка модель Pay-per-use (SaaS), що дозволяє отримати доступ до HPC-обчислень без інвестицій.	Демократизація HPC: Надання можливості проводити інжиніринг.	Контент-маркетинг: Публікація статей про ROI та перевагу над традиційними ліцензіями.

Продовження таблиці 5.9 – Вибір цільових груп потенційних споживачів

Цільова група	Ключові потреби та болі	Мотивація до використання "Thermalcore sim"	Вплив продукту на діяльність споживачів	Стратегії залучення
4. Освітні та наукові установи	Потреба у потужному та доступному інструменті для навчання та фундаментальних досліджень, без обмежень академічними ліцензіями.	Доступність: Низька вартість/безкоштовний доступ для некомерційного використання, прозорість алгоритму (на відміну від "чорних скриньок" комерційного ПЗ).	Покращення навчального процесу: Можливість моделювати реальні, великомасштабні задачі у класах.	Спеціальні програми: Надання безкоштовних ліцензій для студентів та викладачів (Edu-програма), активна участь у open-source спільноті.

Цільові групи були обрані на основі їхньої чутливості до часу обчислень та вартості ліцензування. Таблиця 5.9 демонструє основні цільові групи потенційних споживачів стартап-проєкту та визначає їх ключові потреби, мотивацію до використання продукту, вплив продукту на діяльність споживачів і стратегії залучення.

Таблиця 5.10 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
1	Висока вартість ліцензій на традиційне CAE-ПЗ.	Модель SaaS та оплата за використані обчислювальні ресурси (Pay-per-use).	Гнучка цінова політика, відсутність високих початкових капітальних витрат на придбання ліцензій.
2	Довгий час очікування результатів симуляції для великих 3D-сіток.	Радикальне прискорення Assembly та Aggregation. Час циклу "моделювання-результат" скорочується з годин до хвилин.	Унікальна технологічна перевага завдяки Numba + COO-оптимізації, що усуває критичні послідовні вузькі місця.
3	Складність інтеграції аналізу в автоматизовані цикли проєктування.	Доступ до потужного FEM-кернела через API.	Сучасна Cloud-архітектура, що дозволяє вбудовувати термоаналіз у CI/CD та системи оптимізації.
4	Низька швидкість роботи з анізотропними моделями.	Ефективна підтримка нелінійних та анізотропних властивостей матеріалів.	Спеціалізація на термоаналізі та можливість швидкого модифікування мат-моделі.
5	Залежність від локальних потужностей.	Доступ до HPC-обчислень через хмару.	Повна масштабованість та використання CPU/GPU ресурсів хмарного провайдера.

Аналіз SWOT підтверджує, що головний актив проєкту (S1 – швидкість Assembly/Aggregation) має бути використаний для захоплення ринку Cloud-CAE (O2). Для мінімізації ключової слабкості (W2 – обмежений функціонал) та

найбільшої загрози (T1 – технологічна конкуренція), компанія повинна терміново інвестувати у розробку/інтеграцію потужного паралельного Solver'a (реагуючи на W1). Це дозволить проєкту трансформуватися з "найшвидшого формувальника матриці" у "найшвидшого розв'язувача" у своїй ніші.

Визначення меж ціни для SaaS-рішення "ThermalCore Sim" базується на аналізі ринкової вартості традиційних CAE-систем (аналогів) та альтернативних (замінників) рішень, а також платоспроможності цільових сегментів. У таблиці 5.11. наведено визначення меж встановлення ціни.

Таблиця 5.11. – Визначення меж встановлення ціни

№ п/п	**Рівень цін на товари-замінники**	**Рівень цін на товари-аналоги**	**Рівень доходів цільової групи споживачів**	**Верхня та нижня межі встановлення ціни на товар/послугу**
1	Open-source Solvers (наприклад, OpenFOAM): ~ 0 USD (за ПЗ), але високі витрати на інтеграцію та підтримку (сервіс, консалтинг).	Традиційні CAE-системи (ANSYS, COMSOL): Високий (ліцензії від 10 000 до 50 000+ USD на рік за робоче місце).	Малі/Середні Підприємства (МСП): Середній/Низький (потребують економії).	Нижня межа (Cost-Based/Freemium): Встановлюється на рівні покриття операційних витрат хмари (~0.05-0.20 USD за годину обчислень на CPU).
2	Власні (in-house) розробки на Python/Matlab: Середній (високі витрати на розробку та обслуговування коду).	Спеціалізовані Cloud Solvers: Середній/Високий (Pay-per-use, від 100 до 500+ USD на місяць за базову підписку).	Корпоративні R&D та OEM (Високий): Висока платоспроможність, але вимога до високої продуктивності та надійності.	Верхня межа (Value-Based): Обмежується 20-40% від вартості річної ліцензії основного конкурента, оскільки продукт є нішевим.

Рисунок 5.1 – Цінова стратегія

Формування системи збуту ІТ-продукту – це комплекс дій та стратегій, спрямованих на оптимальну поставку продукту до кінцевого споживача або користувача. Ефективна система збуту грає важливу роль у успіху продукту на ринку. У таблиці 5.12 наведено формування системи збуту.

Таблиця 5.12 – Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	R&D Інженери та МСП: Потребують швидкого доступу та низької ціни, віддають перевагу онлайн-підписці та самостійному тестуванню.	Прямі продажі (B2B SaaS): Надання доступу, активація API, технічна підтримка, онлайн-консультації та управління підписками.	Прямий канал (Zero-Level Channel): Продаж здійснюється безпосередньо від компанії-розробника до кінцевого користувача.	Online Direct Model: Веб-портал для реєстрації, підключення API, оплати та управління обліковим записом. Мінімум посередників.
2	Великі Корпорації (Enterprise Clients): Потребують надійності, інтеграцій (API), навчання та гарантованого рівня обслуговування.	Партнерські продажі та ліцензування: Укладання довгострокових контрактів, технічна інтеграція, кастомізація платформи під внутрішні стандарти.	Прямий канал з елементами партнерства: Використання консультантів/інтеграторів для впровадження у внутрішні системи клієнта.	Hybrid Model (Direct + Value-Added Resellers): Прямі продажі через власну команду + залучення спеціалізованих ІТ-інтеграторів для корпоративних впроваджень.
3	Наукові та освітні Установи, які потребують безкоштовного або пільгового доступу, простої інсталяції та підтримки.	Спеціальні програми (Edu-Programs): Управління некомерційними ліцензіями, надання навчальних матеріалів та технічної документації.	Прямий канал: Доступ надається через спеціальний навчальний портал або за заявкою.	Freemium/Education Access: Безкоштовний доступ з обмеженим функціоналом, що сприяє формуванню лояльності інженерів.

Розробка концепції маркетингових комунікацій є ключовою для ефективного донесення унікальної ціннісної пропозиції "ThermalCore Sim" до цільової аудиторії. Ця таблиця визначає, як, де і з яким ключовим повідомленням проєкт повинен взаємодіяти з кожним сегментом ринку.

Таблиця 5.13 – Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Інженери-конструктори та R&D фахівці шукають рішення на професійних форумах (Reddit, Stack Overflow, Engineering.com) та технічних блогах.	Професійні технічні платформи (LinkedIn, GitHub), Спеціалізовані вебінари та Технічний контент-маркетинг.	Надвисока швидкість (Speed-as-a-Service): Найшвидший FEM-кернел для термоаналізу. Технологічна прозорість (Numba + COO).	Сформувати довіру до технології Python/Numba та довести перевагу у швидкості над традиційним CAE.	"Подолайте обмеження! Скоротіть час симуляції у 10 разів: ThermalCore Sim – це швидкість, яку ви завжди хотіли від CAE."
2	Керівники МСП фокусуються на економії бюджету та ROI (поверненні інвестицій). Приймають рішення на основі фінансових звітів.	Ділові соціальні мережі (LinkedIn), Професійні конференції (B2B events), Прямий Email-маркетинг.	Економічна ефективність: Доступ до HPC-обчислень за гнучкою моделлю Pay-per-use.	Переконати у тому, що SaaS-підписка є вигіднішою за придбання дорогої локальної ліцензії.	"Більше ніяких $50K ліцензій. Перетворіть витрати на інвестиції: отримайте високоточний термоаналіз лише за частку вартості конкурентів."
3	Розробники API/Програмісти шукають надійні, інтегровані та добре документовані рішення.	GitHub (репозиторії, документація), Спеціалізовані IT-конференції, Технічні подкасти.	Легкість інтеграції: Ідеальний API для вбудовування термоаналізу у CI/CD та Digital Twins.	Залучити до використання API у власних інженерних рішеннях та сприяти партнерству.	"ThermalCore API: Інтегруйте потужність FEM у свій продукт за лічені хвилини. Надійна, швидка, повністю документована."
4	Науковці та студенти шукають безкоштовні або пільгові інструменти для досліджень та навчання.	Академічні платформи та журнали, Університетські семінари, освітні платформи (Coursera, Udemy).	Доступність для навчання та R&D: Точність комерційного ПЗ за ціною Open-Source.	Сформувати лояльність до бренду серед майбутніх інженерів-користувачів.	"Моделюйте майбутнє: Безкоштовний доступ для студентів та науковців до найшвидшого FEM-кернела."

Висновок до розділу

У розділі 5 проведено бізнес-аналіз та стратегічне планування проєкту "ThermalCore Sim", що підтвердило його високу ринкову життєздатність.

Ядром проєкту є унікальний паралельний FEM-кернел з оптимізацією COO-агрегації, який забезпечує радикальне прискорення критичних обчислювальних етапів (Assembly/Aggregation) порівняно з традиційними CAE-системами. Проєкт позиціонується як найшвидший хмарний Solver (SaaS) для спеціалізованого термоаналізу складних (нелінійних, анізотропних) матеріалів, орієнтований на R&D-інженерів та МСП. Виявлено, що сильні сторони (S – швидкість, SaaS-модель) дозволяють ефективно скористатися можливостями (O – зростання попиту на Cloud-CAE). Проте, ідентифіковані слабкості (W – боттлнек Solver'a, обмежена мультифізика) вимагають негайної технологічної реакції – інтеграції потужних паралельних розв'язувачів. Обрано гібридну систему збуту (Online Direct + Value-Added Resellers) з гнучкою моделлю ціноутворення Pay-per-use, що робить продукт економічно привабливим для цільової аудиторії, яка прагне уникнути високих ліцензійних витрат.

Розроблена ринкова стратегія, заснована на технологічній перевазі швидкості та економічній ефективності SaaS-моделі, надає чіткий план для успішного виведення проєкту "ThermalCore Sim" на ринок. Проєкт має потенціал стати руйнівником у ніші термоаналізу, але його довгостроковий успіх залежить від успішного усунення поточного обчислювального обмеження на етапі Solver'a.

ВИСНОВКИ

У ході виконання даної дипломної роботи проведено вирішення проблеми обчислювальної ефективності у розрахунку тривимірних стаціонарних потенціальних теплових полів методом скінченних елементів (МСЕ), реалізованим у середовищі Python.

Підтверджено високу інженерну актуальність задачі для моделювання складних систем. Розроблена математична модель враховує нелінійні й анізотропні властивості середовищ, що зводить розрахунок до розв'язання системи нелінійних алгебраїчних рівнянь.

Експерименти чітко ідентифікували два основні обмежувачі продуктивності у послідовному коді: Assembly та Aggregation. Створена гібридна паралельна архітектура (Numba + Joblib) забезпечила ефективне прискорення Assembly.

Найбільш значущим досягненням стало впровадження формату COO (Coordinate List) для Aggregation. Цей стратегічний вибір дозволив замінити повільну послідовну операцію (LIL) на однокрокове, високооптимізоване перетворення COO to CSR, усунувши критичний боттлнек та досягнувши високого прискорення Aggregation.

Оптимізована реалізація Numba + COO продемонструвала найвище загальне прискорення (до 5.93), що підтверджує її високу обчислювальну продуктивність. Цей успіх змістив обчислювальний боттлнек на етап Solver (розв'язання СЛАР), який стає домінуючим на найбільших сітках.

Подальша масштабованість роботи залежить від інтеграції з паралельними бібліотеками розв'язувачів (наприклад, PETSc) для ефективного використання багатоядерних та розподілених систем, що є необхідним для вирішення задач з мільйонами елементів.

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. В.П. Карашецький, В.І. Яркун, Розрахунок тривимірних стаціонарних потенціальних теплових полів методом скінченних елементів. “Інформаційні Технології Та Комп’ютерна Інженерія”, 2024, №1 с.139-145.
2. Y. Sokolovskyy, A. Nechepurenko, T. Samotii, S. Yatsyshyn, O. Mokrytska and V. Yarkun, "Software and algorithmic support for finite element analysis of spatial heat-and-moisture transfer in anisotropic capillary-porous materials" 2020 IEEE Third International Conference on Data Stream Mining & Processing (DSMP), Lviv, Ukraine, 2020, pp. 316-320, doi: 10.1109/DSMP47368.2020.9204175.
3. Towards Eco-Conscious Python: A ComparativeAnalysis of Performance, Energy Efficiency andCarbon Emissions Between CPython andAlternative Implementations Omid Saedi DOI: 10.13140/RG.2.2.12314.04803
4. Приходько М.А., Герасимов Г.Г. Термодинаміка та теплопередача. Навчальний посібник/ - Рівне: НУВГП, 2008.- 250 с.
5. J. Gembarovic J., Loffler M., Gembarovic Jr. J. Simple algorithm for temperature distribution calculations / J. Gembarovic, M. Loffler, J. Gembarovic Jr. //Applied Mathematical Modelling. – 2024. Vol. 28, No 2. – P. 173-182.
6. Chen, Z. X. Finite element methods and their applications / Z. X. Chen.–Berlin : Springer, 2022. – 424 p.
7. Методичні вказівки з розроблення та оформлення магістерської кваліфікаційної роботи для студентів спеціальності 122 «Комп’ютерні науки». Рівень вищої освіти – другий (магістерський) / Упоряд.: І.Д. Капран, І.Б. Борецька, Ю.С. Процик – Львів: НЛТУ України, 2023. – 47 с.
8. F.Petresevics, B. Nagy,“FEM-Based Evaluation of the Point Thermal Transmittance of Various Types of Ventilated Façade Cladding Fastening Systems”,Buildings, 2022.
9. A.Y.Boukounacha, B.Zegnini, Y.Belkacem, S. Tahar, “The Effect of Temperature on the Thermal Conductivity of Transformer Oils Using the Finite

Element Method",1st International Conference on Materials Sciences and Applications "ICMSA2023", Khenchela, Algeria, 2023

10. T. L.Ponsati, A. S.Bahman, F. Iannuzzo,"Thermal Modeling of Large Electrolytic Capacitors Using FEM and Considering the Internal Geometry", IEEE Journal of Emerging and Selected Topics in Power Elec-tronics, 2020.
11. D.Jindra, P.Hradil, J.Kala, V. Salajka, "Non linear FEM analysis of composite concrete slab exposed το extreme thermal load",AIP Conference Proceedings, 2020.

ДОДАТОК А

Код побудови FEM-моделі для стаціонарного температурного поля у нелінійних анізотропних безгістерезисних середовищах

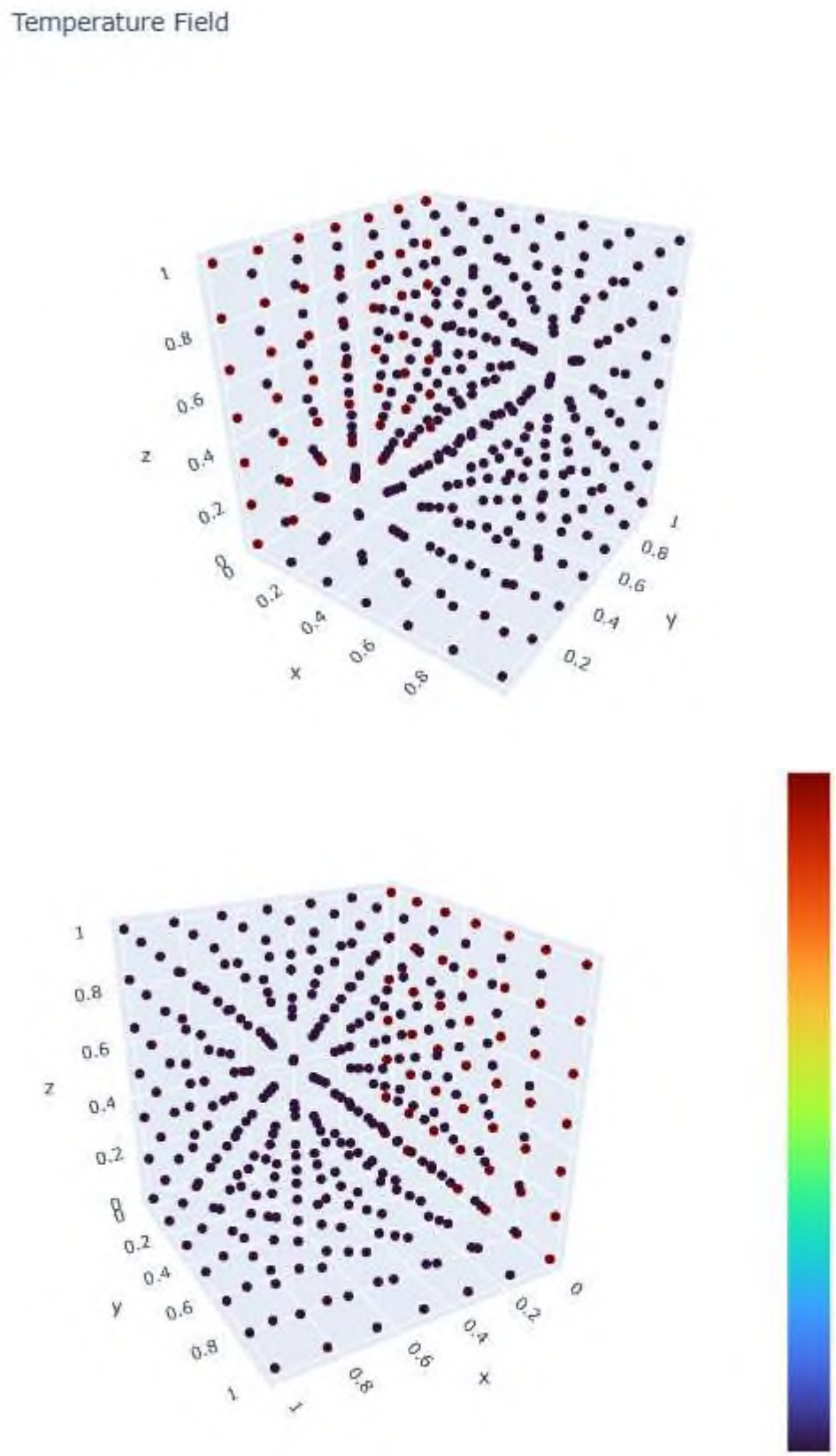
```
import numpy as np
from scipy.sparse import lil_matrix, csr_matrix
from scipy.sparse.linalg import spsolve
import plotly.graph_objects as go

# ---------------------------------------------------------
# Генерація коректної сітки 3D куба + перевірка якості
# ---------------------------------------------------------

def generate_mesh(N):
    """Генерує регулярну сітку для куба [0,1]^3"""
    xs = np.linspace(0, 1, N+1)
    pts = np.array([[x, y, z] for x in xs for y in xs for z in xs])
    idx = lambda i, j, k: i*(N+1)**2 + j*(N+1) + k

    tets = []
    for i in range(N):
        for j in range(N):
            for k in range(N):
                # Куб → 5 тетраедрів
                v0 = idx(i, j, k)
                 ...
                v7 = idx(i+1, j+1, k+1)

                tets += [
                    [v0, v1, v2, v6],
                       ...
                    [v0, v2, v4, v6]
                ]

    return np.array(pts, float), np.array(tets, int)

# ---------------------------------------------------------
# Обчислення жорсткістної матриці тетраедра
# ---------------------------------------------------------

def tet_element(nodes):
    """Повертає (Ke, vol, grads). Ігнорує вироджені тетраедри."""
    v1 = nodes[1] - nodes[0]
    v2 = nodes[2] - nodes[0]
    v3 = nodes[3] - nodes[0]
    J = np.vstack([v1, v2, v3]).T
    detJ = np.linalg.det(J)
```

```
    if abs(detJ) < 1e-12:   # вироджений елемент
        return None, 0, None

    invJ = np.linalg.inv(J)
    grads = np.zeros((4, 3))

    # Формули для лінійних функцій форми
    grads[1,:] = invJ[:,0]
    grads[0,:] = -np.sum(grads[1:,:], axis=0)

    vol = abs(detJ)/6.0
    Ke = np.zeros((4,4))

    for i in range(4):
        for j in range(4):
            Ke[i,j] = vol * np.dot(grads[i], grads[j])

    return Ke, vol, grads

# ---------------------------------------------------------
# Збір глобальної матриці
# ---------------------------------------------------------
def assemble_global(pts, tets, k):
    n = pts.shape[0]
    A = lil_matrix((n, n))
    b = np.zeros(n)

    for e, tet in enumerate(tets):
        nodes = pts[tet]
        res = tet_element(nodes)
        if res[0] is None:     # пропускаємо погані елементи
            continue
        Ke, _, _ = res

        for i_local, i_global in enumerate(tet):
            for j_local, j_global in enumerate(tet):
                A[i_global, j_global] += k * Ke[i_local]

    return A.tocsr(), b

# ---------------------------------------------------------
# Задаємо граничні умови
# ---------------------------------------------------------
def apply_bc(A, b, bc_idx, bc_val):
    A = A.tolil()
    for idx, val in zip(bc_idx):
        A[idx,:] = 0
        A[:,idx] = 0
        A[idx,idx] = 1
        b[idx] = val
```

```
    return A.tocsr(), b

# ------------------------------------------------------------
# Нелінійний ітераційний розв'язувач
# ------------------------------------------------------------
def solve_nonlinear(pts, tets, bc_idx, bc_val, k0=1.0, alpha=0.3, maxit=6):
    T = np.zeros(len(pts))

    for it in range(maxit):
        k_eff = k0 * (1 + alpha * T)   # локальна нелінійність

        A, b = assemble_global(pts, tets, k0)
        A, b = apply_bc(A, b, bc_val)
        Tnew = spsolve(A, b)

        err = np.linalg.norm(Tnew - T)
        print(f"Iteration {it+1}: error = {err:.3e}")

        T = Tnew

        if err < 1e-6:
            break

    return T

# ------------------------------------------------------------
# Запуск розрахунку
# ------------------------------------------------------------
pts, tets = generate_mesh(6)

# Граничні умови: T=1 при x=0, T=0 при x=1
bc_idx = np.where(np.isclose(pts[:,0], 0))[0] + \
         np.where(np.isclose(pts[:,0], 1))[0]
bc_val = np.concatenate([
    np.zeros(np.sum(np.isclose(pts[:,0],1)))
])

T = solve_nonlinear(pts, tets, bc_idx, bc_val)

# ------------------------------------------------------------
# 3D Візуалізація
# ------------------------------------------------------------
fig = go.Figure(data=go.Scatter3d(
    x=pts[4:,0], y=pts[4:,1], z=pts[4:,2],
    mode='markers',
    marker=dict(size=4, color=T, colorscale='Turbo')
))

fig.update_layout(title="Temperature Field", width=700, height=600)
fig.show()
```

ДОДАТОК Б

Розподіл температури



ДОДАТОК В

```
# ----------------------------------------------------
# --- 4. ПОСЛІДОВНИЙ РОЗВ'ЯЗУВАЧ (Чистий Python Еталон) ---
# ----------------------------------------------------
def sequential_fem_solver_timed(nodes, elements, internal_nodes, R_map,
U_boundary, mu_s):
        num_internal_nodes = len(internal_nodes)
    R_indices = np.array(internal_nodes)
    U_e_placeholder = np.zeros(4)
    total_start_time = time.time()
    assembly_start_time = time.time()
    local_results = []

    for element in elements:
        node_coords = nodes[element]
        _, phi_m = assemble_element_vanilla(node_coords, U_e_placeholder, mu_s)
        local_results.append((element, phi_m))
    time_assembly_seq = time.time() - assembly_start_time

    aggregation_start_time = time.time()
    J_global = lil_matrix((num_internal_nodes, num_internal_nodes))
    phi_R = np.zeros(num_internal_nodes)

    for element, phi_m in local_results:
        for i in range(4):
            for j in range(4):
                global_i = element[i]; global_j = element[j]

                if global_i in R_map and global_j in R_map:
                    R_idx_i = R_map[global_i]; R_idx_j = R_map[global_j]
                    J_global[R_idx_i, R_idx_j] += phi_m[i, j]

                if global_i in R_map and global_j not in R_map:
                    R_idx_i = R_map[global_i]
                    phi_R[R_idx_i] -= phi_m[i, j] * U_boundary[global_j]

    time_aggregation_seq = time.time() - aggregation_start_time

    solver_start_time = time.time()
    J_global_csr = J_global.tocsr()
    U_R = spsolve(J_global_csr, phi_R)
    time_solver_seq = time.time() - solver_start_time

    time_total_seq = time.time() - total_start_time

    U_result = np.copy(U_boundary)
    U_result[R_indices] = U_R
    return U_result, time_total_seq, time_assembly_seq, time_aggregation_seq,
time_solver_seq
```

ДОДАТОК Г

```
# ----------------------------------------------------
# --- 5. ПАРАЛЕЛЬНИЙ РОЗВ'ЯЗУВАЧ - ВЕРСІЯ 2: OPTIMIZED (Numba + Chunking + COO)
# ----------------------------------------------------

def process_element_chunk_parallel_optimized(element_chunk, nodes, U_boundary,
R_map, mu_s):
    rows_list = []; cols_list = []; data_list = []
    rhs_rows_list = []; rhs_data_list = []
    U_e_placeholder = np.zeros(4)

    for element in element_chunk:
        node_coords = nodes[element]
        _, phi_m = assemble_element(node_coords, U_e_placeholder, mu_s)

        for i in range(4):
            for j in range(4):
                global_i = element[i]; global_j = element[j]
                value = phi_m[i, j]

                if global_i in R_map and global_j in R_map:
                    R_idx_i = R_map[global_i]; R_idx_j = R_map[global_j]
                    rows_list.append(R_idx_i); cols_list.append(R_idx_j);
data_list.append(value)

                if global_i in R_map and global_j not in R_map:
                    R_idx_i = R_map[global_i]
                    rhs_rows_list.append(R_idx_i); rhs_data_list.append(-value *
U_boundary[global_j])

    return rows_list, cols_list, data_list, rhs_rows_list, rhs_data_list

def parallel_fem_solver_optimized(nodes, elements, internal_nodes, R_map,
U_boundary, mu_s):

    num_internal_nodes = len(internal_nodes)
    R_indices = np.array(internal_nodes)
    total_start_time = time.time()

    assembly_start_time = time.time()
    n_jobs = cpu_count()
    chunk_size = len(elements) // n_jobs + 1
    element_chunks = [elements[i:i + chunk_size] for i in range(0,
len(elements), chunk_size)]

    chunked_results = Parallel(n_jobs=-1)(
        delayed(process_element_chunk_parallel_optimized)(chunk, nodes,
U_boundary, R_map, mu_s)
```

```
        for chunk in element_chunks
    )

    rows = np.concatenate([np.array(r, dtype=np.int32) for r, _, _, _, _ in
chunked_results if len(r) > 0])
    cols = np.concatenate([np.array(c, dtype=np.int32) for _, c, _, _, _ in
chunked_results if len(c) > 0])
    data = np.concatenate([np.array(d, dtype=np.float64) for _, _, d, _, _ in
chunked_results if len(d) > 0])
    rhs_rows = np.concatenate([np.array(rr, dtype=np.int32) for _, _, _, rr, _
in chunked_results if len(rr) > 0])
    rhs_data = np.concatenate([np.array(rd, dtype=np.float64) for _, _, _, _, rd
in chunked_results if len(rd) > 0])

    time_assembly_par = time.time() - assembly_start_time

    aggregation_start_time = time.time()
    J_global_csr = csr_matrix((data, (rows, cols)), shape=(num_internal_nodes,
num_internal_nodes))
    phi_R = np.zeros(num_internal_nodes)
    np.add.at(phi_R, rhs_rows, rhs_data)
    time_aggregation_par = time.time() - aggregation_start_time

    solver_start_time = time.time()
    U_R = spsolve(J_global_csr, phi_R)
    time_solver_par = time.time() - solver_start_time

    time_total_par = time.time() - total_start_time

    U_result = np.copy(U_boundary)
    U_result[R_indices] = U_R

    return U_result, time_total_par, time_assembly_par, time_aggregation_par,
time_solver_par
```

ДОДАТОК Д

```
# ------------------------------------------------------
# --- 6. ПАРАЛЕЛЬНИЙ РОЗВ'ЯЗУВАЧ - ВЕРСІЯ 1:                          ---
# --- SLOW AGGREGATION (Numba Assembly + Slow LIL Aggregation)      ---
# ------------------------------------------------------

def process_element_chunk_parallel_slow_aggregation(element_chunk, nodes,
U_boundary, R_map, mu_s):
    local_results = []
    U_e_placeholder = np.zeros(4)

    for element in element_chunk:
        node_coords = nodes[element]
        _, phi_m = assemble_element(node_coords, U_e_placeholder, mu_s)
        local_results.append((element, phi_m))
    return local_results

def parallel_fem_solver_slow_aggregation(nodes, elements, internal_nodes, R_map,
U_boundary, mu_s):

    num_internal_nodes = len(internal_nodes)
    R_indices = np.array(internal_nodes)
    total_start_time = time.time()

    assembly_start_time = time.time()

    n_jobs = cpu_count()
    chunk_size = len(elements) // n_jobs + 1
    element_chunks = [elements[i:i + chunk_size] for i in range(0,
len(elements), chunk_size)]

    chunked_results = Parallel(n_jobs=-1)(
        delayed(process_element_chunk_parallel_slow_aggregation)(chunk, nodes,
U_boundary, R_map, mu_s)
        for chunk in element_chunks
    )

    local_results = [item for sublist in chunked_results for item in sublist]

    time_assembly_par = time.time() - assembly_start_time

    aggregation_start_time = time.time()

    J_global = lil_matrix((num_internal_nodes, num_internal_nodes))
    phi_R = np.zeros(num_internal_nodes)

    for element, phi_m in local_results:
        for i in range(4):
            for j in range(4):
                global_i = element[i]; global_j = element[j]
```

```
                if global_i in R_map and global_j in R_map:
                    R_idx_i = R_map[global_i]; R_idx_j = R_map[global_j]
                    J_global[R_idx_i, R_idx_j] += phi_m[i, j]

                if global_i in R_map and global_j not in R_map:
                    R_idx_i = R_map[global_i]
                    phi_R[R_idx_i] -= phi_m[i, j] * U_boundary[global_j]

    time_aggregation_par = time.time() - aggregation_start_time

    solver_start_time = time.time()
    J_global_csr = J_global.tocsr()
    U_R = spsolve(J_global_csr, phi_R)
    time_solver_par = time.time() - solver_start_time

    time_total_par = time.time() - total_start_time

    U_result = np.copy(U_boundary)
    U_result[R_indices] = U_R

    return U_result, time_total_par, time_assembly_par, time_aggregation_par,
time_solver_par
```