**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ**

**І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

**Факультет/(ННІ) Інформаційних технологій**

| **ПОГОДЖЕНО** | **ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ** |
|---|---|
| **Декан факультету (Директор ННІ)** | **Завідувач кафедри** |
| Інформаційних технологій<br>(назва факультету (ННІ)) | Комп'ютерних наук<br>(назва кафедри) |
| _____ Ігор Болбот<br>(підпис) (ім'я ПРІЗВИЩЕ) | _____ Белла Голуб<br>(підпис) (ім'я ПРІЗВИЩЕ) |
| "___" _____ 20__ р. | "___" _____ 20__ р. |

**МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА**

**на тему** Моделювання паралельних обчислень з метою оптимізації продуктивності програмних застосувань

Спеціальність 122 «Комп'ютерні науки»
(код і найменування)

Освітня програма Інформаційно управляючі системи та технології
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

**Гарант освітньої програми**

к.т.н., доцент _____ Белла Голуб
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

**Керівник магістерської кваліфікаційної роботи**

д.т.н., професор _____ Наталія Заєць
(науковий ступінь та вчене звання) (підпис) (ім'я ПРІЗВИЩЕ)

**Виконав** _____ Владислав Вознюк
(підпис) (ім'я ПРІЗВИЩЕ здобувача)

**КИЇВ – 2025**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ**

**І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

**Факультет (ННІ)** інформаційних технологій

**ЗАТВЕРДЖУЮ**
**Завідувач кафедри** комп'ютерних наук

доцент, к.т.н. _____ Голуб Б.Л.
(науковий ступінь, вчене звання) (підпис) (ПІБ)
" 10 " жовтня 2025 року

**ЗАВДАННЯ**

**ДО ВИКОНАННЯ МАГІСТЕРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ**

Вознюку Владиславу Вадимовичу
**(прізвище, ім'я, по батькові)**

Спеціальність 122 "Комп'ютерні науки"
(код і найменування)

Освітня програма Інформаційні управляючі системи та технології
(назва)

Орієнтація освітньої програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Тема магістерської кваліфікаційної роботи

«Моделювання паралельних обчислень з метою оптимізації продуктивності програмних застосувань»

затверджена наказом проректора НУБіП України від " 10 " жовтня 2025 р. № 2289

"С" на часткову зміну до наказу від 01 листопада 2024 р. №1964 «С»

Термін подання завершеної роботи на кафедру 14 листопада 2025 р.
(рік, місяць, число)

Вихідні дані до магістерської кваліфікаційної роботи

Початкові статистичні дані про час виконання завдань, кількість обчислювальних вузлів та обсяг вхідних даних, необхідні для аналізу й тестування програмного забезпечення з моделювання паралельних обчислень.

Перелік питань, що підлягають дослідженню:

1. Проаналізувати існуючі підходи до моделювання та оптимізації паралельних обчислень, визначити їхні переваги та обмеження.
2. Дослідити роботу наявних програмних засобів і бібліотек для організації паралельних обчислень (OpenMP, MPI, Parallel STL, oneTBB та OpenBLAS)порівняти їхні можливості, переваги та недоліки у різних сценаріях.
3. Провести експериментальні дослідження продуктивності на різних наборах вхідних даних та параметрах, оцінити масштабованість і виявити «вузькі місця» в обчисленнях.
4. Сформулювати практичні рекомендації щодо вибору оптимальних інструментів і конфігурацій ресурсів для підвищення продуктивності паралельних застосувань.

Перелік графічного матеріалу (за потреби)_____

**Дата видачі завдання** " 1 " листопада 2024 р.

**Керівник магістерської кваліфікаційної роботи** _____ Заєць Н.А.
( підпис ) (прізвище та ініціали)

**Завдання прийняв до виконання** _____ Вознюк В.В.

# Календарний план

| № з/п | Назва етапів виконання магістерської кваліфікаційної роботи | Строк виконання | Примітка |
|---|---|---|---|
| **1** | Видача завдання | 01.11.2024 | |
| **2** | Аналіз предметної області паралельних обчислень | 02.11-30.11.2024 | |
| **3** | Проєктування експериментів паралельних (OpenMP, MPI, Parallel STL (PSTL), Intel oneTBB, OpenBLAS) | 1.12.2024-02.01.2025 | |
| **4** | Проведення експериментів в середовищі 1. MSYS2 (UCRT64): розробка бази даних, аналіз закона Амдала, аналітичні моделі, побудува OLAP-куба | 03.01-15.05.2025 | |
| **5** | Аналіз результатів експериментів | 16.05-20.06.2025 | |
| **6** | Оформлення пояснювальної записки | 31.07-13.11.2025 | |
| **7** | Оформлення постеру за результатами дослідження | 07.10-15.10.2025 | |
| **8** | Написання тез до постеру | 16.10-25.10.2025 | |
| **9** | Постерна сесія | 28.10-29.10.2025 | |
| **10** | Перевірка на плагіат | 14.11.2025 | |
| **11** | Попередній захист | 01.12.2025 | |
| **12** | Захист магістерської кваліфікаційної роботи | 15-19.12.2025 | |

**Студент** _____ Владислав Вознюк

(підпис) (ім'я та прізвище)

**Керівник магістерської кваліфікаційної роботи** _____ Наталя Заєць

(підпис) (прізвище та ініціали)

# РЕФЕРАТ

**Структура та обсяг роботи.** Магістерська робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 82 сторінок, містить 24 рисунків, 30 бібліографічних джерел. Робота оформлена відповідно до вимог університету.

**Актуальність теми** Сучасні обчислювальні системи вже не можуть забезпечувати приріст продуктивності лише за рахунок збільшення тактової частоти процесорів. Тому розвиток високопродуктивних обчислень відбувається через масштабування багатоядерних архітектур і застосування паралельних технологій. Вибір оптимальної бібліотеки чи підходу до паралельного програмування суттєво впливає на продуктивність програмних застосувань, особливо в задачах лінійної алгебри, моделювання, обробки великих даних та інженерних розрахунків.

У цьому контексті особливо актуальними є експериментальні дослідження продуктивності різних паралельних технологій, а також аналіз їх обмежень відповідно до закону Амдала. Це дозволяє сформувати рекомендації щодо оптимізації програм під реальні багатоядерні процесори.

**Метою магістерської роботи** є моделювання та експериментальне дослідження продуктивності різних технологій паралельних обчислень для оптимізації виконання програмних застосувань, а також аналіз ефективності їх масштабування на основі закону Амдала.

Об’єкт дослідження. Процеси та методи паралельного виконання програмних застосувань у багатоядерних обчислювальних системах.

**Предмет дослідження.** Підходи, алгоритми та бібліотеки паралельного програмування, зокрема: OpenMP, MPI, Parallel STL (PSTL), Intel oneTBB, OpenBLAS, а також їх ефективність, масштабованість і продуктивність під час виконання обчислювально інтенсивних задач.

**Методи дослідження.**

У роботі використано такі методи:

1. методи паралельного програмування (OpenMP, MPI, PSTL, oneTBB);
2. аналітичні методи оцінювання продуктивності, включаючи визначення часу виконання, прискорення, ефективності, GFLOPS;
3. методологія рівнянь продуктивності та закон Амдала для оцінки теоретичних меж масштабування;
4. експериментальний метод моделювання множення матриць різного розміру;
5. статистична обробка результатів з використанням повторних запусків;
6. візуалізація даних засобами Python (pandas, matplotlib) для побудови графіків.

**Наукова новизна** роботи полягає у:

1. Комплексному порівнянні п’яти різних технологій паралельного програмування в єдиних експериментальних умовах для задачі множення матриць розмірності 256–2048.
2. Експериментальному визначенні реальних значень паралельної частки F та максимального прискорення S(max) згідно із законом Амдала для кожної технології.
3. Демонстрації, що різні бібліотеки мають різні межі масштабування, неочевидні з теорії, та залежать від архітектури CPU, структури алгоритму та внутрішніх моделей паралелізації.
4. Встановленні того, що OpenBLAS показує найвищу продуктивність (до десятків GFLOPS), тоді як MPI та PSTL мають обмежене масштабування на локальній платформі.

5. Наданні практичних рекомендацій щодо вибору технології паралелізації для високопродуктивних застосувань на звичайних багатоядерних процесорах.

**Практичне значення.**

Результати роботи можуть бути використані для:

- оптимізації наукових та інженерних застосувань, що виконуються на багатоядерних CPU;
- вибору найефективнішої бібліотеки паралельних обчислень для задач великої розмірності;
- прогнозування продуктивності алгоритмів на основі закону Амдала;
- підготовки освітніх матеріалів з паралельного програмування;
- розробки високопродуктивних модулів у системах аналізу даних, машинного навчання та моделювання.

**Програмне забезпечення.**

Для реалізації експериментів використано:

1. MSYS2 (UCRT64) як середовище компіляції програм на C++;
2. компілятор g++ з підтримкою OpenMP та PSTL;
3. Microsoft MPI (MS-MPI) для запуску розподілених процесів;
4. Intel oneTBB та OpenBLAS як зовнішні бібліотеки;
5. Python 3.10, бібліотеки pandas та matplotlib для побудови графіків і аналізу;
6. операційну систему Windows 10 як платформу для виконання моделювання та тестування.

## ABSTRACT

**Structure and Volume of the Thesis.** The master's thesis consists of an introduction, three main chapters, conclusions, a list of references, and appendices. The total volume of the thesis is 82 pages, including 24 figures and 30 bibliographic sources. The work is formatted in accordance with university requirements.

**Relevance of the Topic.** Modern computing systems can no longer rely on increasing processor clock frequency as the main source of performance improvement. The progress of high-performance computing has shifted toward scaling multi-core architectures and using parallel technologies. Selecting an optimal parallel programming framework significantly affects the performance of scientific, engineering, and data-intensive software applications.

In this context, experimental analysis of different parallel programming technologies and evaluation of their scalability based on Amdahl's law are particularly important. Such research provides a foundation for optimizing software for real multi-core CPU systems and identifying the practical limitations of existing parallelization approaches.

**Purpose of the Master's Thesis.** The purpose of this thesis is to model and experimentally evaluate the performance of various parallel computing technologies to optimize the execution of software applications, and to analyze their scalability and efficiency using Amdahl's law.

**Object of Study.** Processes and methods of parallel execution of software applications in multi-core computing systems.

**Subject of Study.** Approaches, algorithms, and parallel programming libraries, including OpenMP, MPI, Parallel STL (PSTL), Intel oneTBB, and OpenBLAS, as well as their performance, scalability, and efficiency when executing computationally intensive tasks.

**Research Methods.**

The following methods were used in the thesis:

- parallel programming techniques (OpenMP, MPI, PSTL, oneTBB);
- analytical methods of performance evaluation, including execution time, speedup, efficiency, and GFLOPS;
- Amdahl's law and performance modeling for estimating theoretical scalability;
- experimental simulation of matrix multiplication for problem sizes from 256 to 2048;
- statistical processing of results, including repeated runs for accuracy;
- data visualization using Python libraries (pandas, matplotlib).

**Scientific Novelty.**

The scientific novelty of the thesis lies in the following:

1. A comprehensive comparison of five parallel programming technologies under identical experimental conditions for matrix multiplication tasks of varying sizes.
2. Experimental estimation of the parallel fraction F and maximum speedup S(max) according to Amdahl's law for each technology.
3. Demonstration that different libraries exhibit fundamentally different scalability limits, which depend on CPU architecture, algorithm structure, and internal parallelization mechanisms.
4. Identification of OpenBLAS as the most high-performing solution (reaching tens of GFLOPS), while MPI and PSTL exhibit limited scalability on a local multi-core platform.
5. Development of practical recommendations for selecting optimal parallel technologies for scientific and engineering applications executed on multi-core processors.

**Practical Significance.**

The results of the thesis can be applied to:

1. optimization of scientific and engineering applications running on multi-core CPUs;
2. selection of the most efficient parallel programming library for large-scale computational tasks;
3. prediction of algorithm performance based on Amdahl's law;
4. development of educational materials on parallel programming;
5. designing high-performance modules for data analysis, numerical modeling, and machine learning.

**Software Tools.**

The following software tools were used in the research:

1. MSYS2 (UCRT64) as a compilation environment for C++ programs;
2. g++ compiler with support for OpenMP and PSTL;
3. Microsoft MPI (MS-MPI) for distributed process execution;
4. Intel oneTBB and OpenBLAS as external performance libraries;
5. Python 3.10 with pandas and matplotlib for data analysis and visualization;
6. Windows 10 as the platform for modeling and testing experiments.

# Зміст

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CPU (Central Processing Unit) - Центральний процесор

HPC (High Performance Computing) - Високопродуктивні обчислення

OpenMP (Open Multi-Processing) - Інтерфейс для розпаралелювання програм

MPI (Message Passing Interface) - Стандарт для організації паралельних обчислень

PSTL (Parallel Standard Template Library) - Паралельна реалізація стандартної бібліотеки C++

oneTBB (one Threading Building Blocks) - Бібліотека паралельного програмування від Intel

OpenBLAS (Open Basic Linear Algebra Subprograms) - Оптимізована бібліотека для лінійної алгебри

MSYS2 (Minimal System 2) - Середовище для розробки у Windows

UCRT64 - Підсистема MSYS2

GFLOPS - Кількість мільярдів операцій з плаваючою комою

# ВСТУП

У сучасних умовах стрімкого розвитку обчислювальної техніки та інформаційних технологій питання підвищення продуктивності програмних систем набуває особливої актуальності. Зі збільшенням обсягів даних, складності алгоритмів і зростанням вимог до швидкості обчислень виникає необхідність у застосуванні методів паралельного програмування, які дозволяють ефективно використовувати ресурси багатоядерних процесорів і багатопроцесорних систем. Розвиток паралельних обчислень відкриває можливості для суттєвого скорочення часу виконання задач і підвищення масштабованості програмних застосувань. Водночас реалізація паралельних алгоритмів потребує врахування багатьох чинників — структури апаратної системи, моделі розподілу даних, синхронізації потоків і специфіки бібліотек, що реалізують паралельність. Тому питання моделювання та оптимізації паралельних обчислень є надзвичайно важливим для сучасного програмного забезпечення наукового, аналітичного та інженерного призначення.

Актуальність роботи полягає в тому, що паралельне програмування є одним із найефективніших способів підвищення швидкодії обчислень без зміни апаратного забезпечення. Завдяки розпаралеленню виконання алгоритмів можливо забезпечити використання всіх доступних обчислювальних ядер, що особливо важливо для задач з великою обчислювальною складністю — таких як симуляції, машинне навчання, обробка зображень, моделювання фізичних процесів тощо. Попри це, ефективність паралельного виконання не зростає лінійно із кількістю процесорів. Відповідно до закону Амдала, швидкість виконання обмежується тією частиною алгоритму, яку неможливо розпаралелити.Тому актуальним завданням є експериментальне порівняння сучасних технологій паралельного програмування з метою визначення їхніх сильних і слабких сторін, а також побудова аналітичної моделі продуктивності.

Метою магістерської кваліфікаційної роботи є моделювання процесів паралельних обчислень з метою підвищення продуктивності програмних

застосувань та визначення оптимальних технологій для реалізації таких обчислень.

Для досягнення поставленої мети необхідно виконати такі завдання дослідження:

1. Проаналізувати існуючі підходи до моделювання та оптимізації паралельних обчислень, визначити їхні переваги та обмеження.
2. Дослідити роботу сучасних бібліотек і програмних засобів для організації паралельних обчислень — OpenMP, MPI, Parallel STL, oneTBB, OpenBLAS.
3. Провести експериментальні дослідження ефективності та масштабованості цих технологій на прикладі задачі множення матриць.
4. Здійснити аналітичну оцінку масштабованості за законом Амдала, визначити частку паралельного коду (f) і максимальне прискорення (Smax).
5. Розробити рекомендації щодо вибору оптимальних інструментів і конфігурацій ресурсів для підвищення продуктивності паралельних застосувань.

Об’єктом дослідження є процес паралельного виконання обчислювальних задач у багатоядерних системах.

Предметом дослідження є методи, засоби та бібліотеки оптимізації паралельних обчислень на рівні програмного забезпечення.

У роботі застосовано такі методи дослідження:

1. теоретичний аналіз і порівняння існуючих технологій паралельного програмування;
2. експериментальні вимірювання часу, ефективності та продуктивності обчислень у різних середовищах;
3. аналітична оцінка масштабованості за законом Амдала;

4. статистична обробка результатів експериментів для усереднення похибок.

Експериментальна частина реалізована мовою C++ у середовищі MSYS2 (UCRT64) із використанням бібліотек OpenMP, MPI, PSTL, oneTBB та OpenBLAS. Для аналізу результатів і побудови графіків застосовано Python (pandas, matplotlib).

Наукова новизна роботи полягає в комплексному експериментальному порівнянні різних технологій паралельного програмування за єдиною задачею (множення матриць) та подальшому аналітичному аналізі результатів відповідно до закону Амдала. Отримані показники дозволяють оцінити ефективність масштабування і визначити реальні межі прискорення програм при збільшенні кількості потоків.

# 1. СИСТЕМНИЙ АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

## 1.1. Загальні положення та еволюція архітектур паралельних обчислень

Розвиток комп’ютерних систем упродовж другої половини XX століття відбувався в основному завдяки зростанню тактової частоти процесорів та ускладненню їх мікроархітектури. Такий підхід до підвищення продуктивності був можливий завдяки постійному удосконаленню технологічних норм виробництва напівпровідників, що узгоджувалося з емпіричним законом Мура: кількість транзисторів на кристалі подвоюється приблизно раз на півтора–два роки. Упродовж десятиліть це дійсно забезпечувало зростання продуктивності процесорів без суттєвих змін у принципах побудови комп’ютерних систем.

Однак у 2000-х роках подальше масштабування тактових частот стало фізично обмеженим через перегрів, енергоспоживання та затримки сигналів. Це призвело до переходу від нарощування продуктивності окремого ядра до створення багатоядерних та багатопроцесорних систем, у яких підвищення швидкодії можливе лише шляхом **паралельного виконання обчислень**.

Паралельні обчислення стали центральним напрямом розвитку сучасної інформатики, оскільки саме вони дозволяють ефективно використовувати ресурси багатоядерних CPU, кластерних систем, суперкомп’ютерів та хмарних платформ. У зв’язку з цим виникла потреба у спеціалізованих архітектурах, моделях програмування та засобах синхронізації, що забезпечують коректне та масштабоване виконання паралельних задач.

Одним із фундаментальних підходів до класифікації комп’ютерних архітектур є схема, запропонована М. Флінном, яка аналізує комп’ютерні системи за кількістю потоків команд і потоків даних. Вона залишається актуальною і сьогодні, оскільки дозволяє систематизувати сучасні моделі паралельності.

1.2. Класифікація архітектур за Філінном

Архітектура SISD (Single Instruction – Single Data)=

SISD відповідає класичній фон-Нейманівській моделі. У системах такого типу виконується один потік команд над одним потоком даних — послідовно, без елементів паралельності. Такі архітектури характерні для ранніх однопроцесорних машин, проте в сучасних системах концептуально використовуються як складові окремих ядер або процесорів.

Архітектура SIMD (Single Instruction – Multiple Data)

У системах SIMD одна команда застосовується одночасно до багатьох елементів даних. Подібний підхід реалізується у векторних процесорах, а також у сучасних інструкційних розширеннях CPU: SSE, AVX, AVX-512. SIMD забезпечує високий рівень паралельності для задач, де однакові операції здійснюються над великими масивами даних (лінійна алгебра, обробка сигналів, графіка).

Архітектура MISD (Multiple Instruction – Single Data)

MISD є теоретично можливою моделлю, у якій кілька потоків команд обробляють один потік даних. На практиці такі системи майже не використовуються через обмежену практичну користь. MISD частіше розглядають як концептуальну модель для аналізу спеціалізованих систем із надлишковим дублюванням обчислень.

Архітектура MIMD (Multiple Instruction – Multiple Data)

MIMD є найбільш універсальним класом паралельних систем, у яких кожен процесор виконує власний потік команд над власними даними. Цей тип архітектури охоплює більшість сучасних багатоядерних процесорів, кластерів і суперкомп’ютерів. MIMD є найбільш універсальним класом паралельних систем, у яких кожен процесор виконує власний потік команд над власними

даними. Цей тип архітектури охоплює більшість сучасних багатоядерних процесорів, кластерів і суперкомп'ютерів.

Моделі MIMD поділяють за способом організації пам'яті:

***Архітектура зі спільною пам'яттю (Shared Memory, SM)***

Усі ядра працюють у спільному адресному просторі. Синхронізація виконується через примітиви (мьютекси, атомарні операції) або високорівневі конструкції (#pragma omp parallel). Представники: багатоядерні процесори Intel, AMD, ARM. (Рис. 1.1)

***Архітектура з розподіленою пам'яттю (Distributed Memory, DM)***

Кожен вузол має власну пам'ять; обмін даними здійснюється через передачу повідомлень (message passing). Представники: кластери, суперкомп'ютери, HPC-системи. (Рис. 1.1)

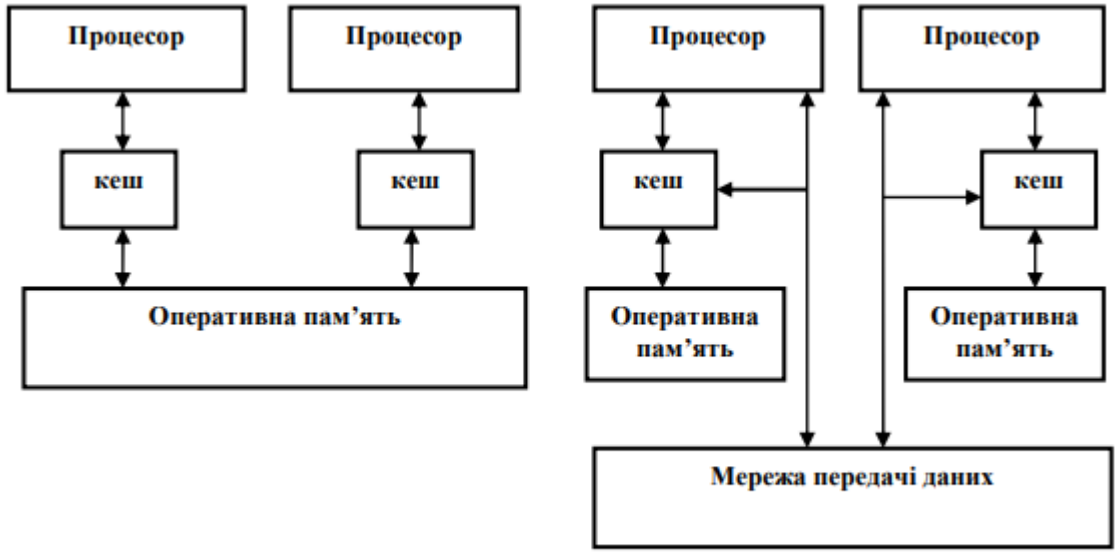


Рис.1.1 Способи підключення до основної пам'яті

А) спільна пам'ять, Б) розподілена пам'ять

## 1.3 Паралельні моделі та підходи до організації обчислень

Паралелізм за даними

Паралелізм за даними полягає у тому, що однакові операції виконуються над різними частинами масиву даних. Кожен процесор або потік отримує фрагмент даних, обробляє його незалежно, а потім результати об’єднуються. Такий підхід характерний для SIMD та багатоядерних CPU при використанні OpenMP, PSTL, OpenBLAS.

Особливості:

– висока масштабованість;
– мала потреба у синхронізації;
– автоматична векторизація та конвеєризація.

Паралелізм підзадач

Паралелізм підзадач (task parallelism) має на увазі розбиття задачі на логічно самостійні частини, що можуть виконуватися незалежно. Такий підхід характерний для MIMD-архітектур та реалізований у MPI, oneTBB, OpenMP (task API).

Виклики:

– складність балансування навантаження,
– необхідність обміну даними,
– синхронізація залежностей.

Темпоральний паралелізм

Паралелізація за часом полягає у використанні різних частин обчислювальної схеми для прямого та зворотного проходу задачі, що дозволяє скорочувати час виконання складних алгоритмів. Такий підхід використовується рідше, переважно у спеціалізованих алгоритмах (кодування, оптимізація, реверсивні моделі). (Рис. 1.2)

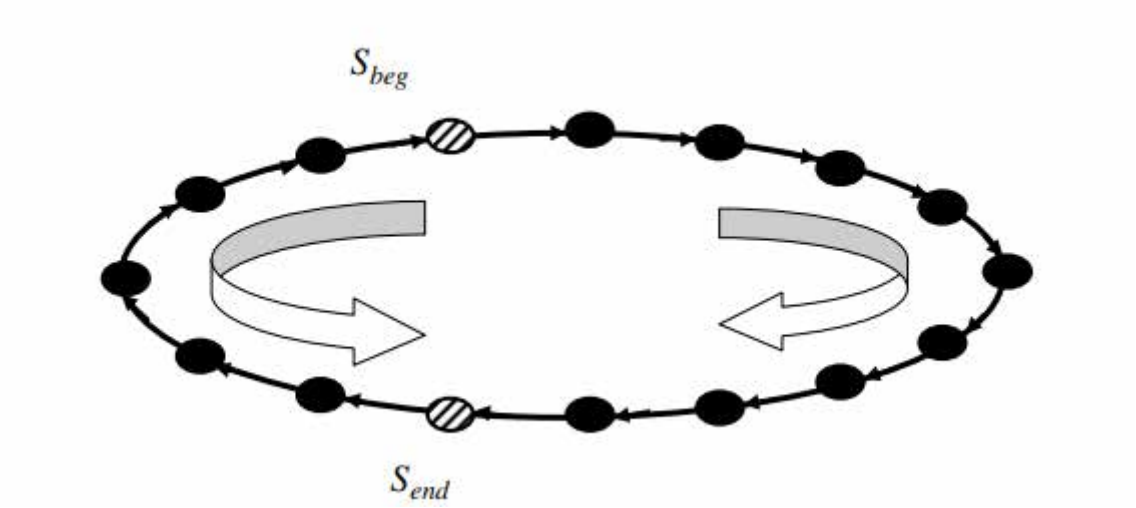


Рис. 1.2 – Паралелізм за часом виконання

1.3. Бібліотеки та засоби паралельного програмування

OpenMP

OpenMP (Open Multi-Processing) — це відкритий промисловий стандарт для організації паралельних обчислень на багатоядерних процесорах зі спільною пам’яттю. Його програмна модель базується на fork–join парадигмі: програма стартує як один “головний” потік, який при вході в паралельну ділянку створює команду робочих потоків (fork), а після завершення обчислень усі потоки синхронізуються і зливаються назад в один (join).

OpenMP реалізується через директиви компілятора (#pragma omp у C/C++), функції бібліотеки часу виконання (omp_get_num_threads(), omp_get_wtime() тощо) та змінні оточення, які дозволяють керувати кількістю потоків і політиками розкладання. Основні конструкції включають паралельні регіони (parallel), паралельні цикли (for/do), секції (sections), задачі (task), механізми синхронізації (critical, barrier, atomic) та механізми редукцій для обчислення сум, максимумів тощо. Це дає змогу програмісту поступово “розпаралелювати” існуючий послідовний код, додаючи лише директиви без повної його перебудови.

Переваги:

- простота використання;
- автоматичне керування потоками;
- масштабованість для CPU.

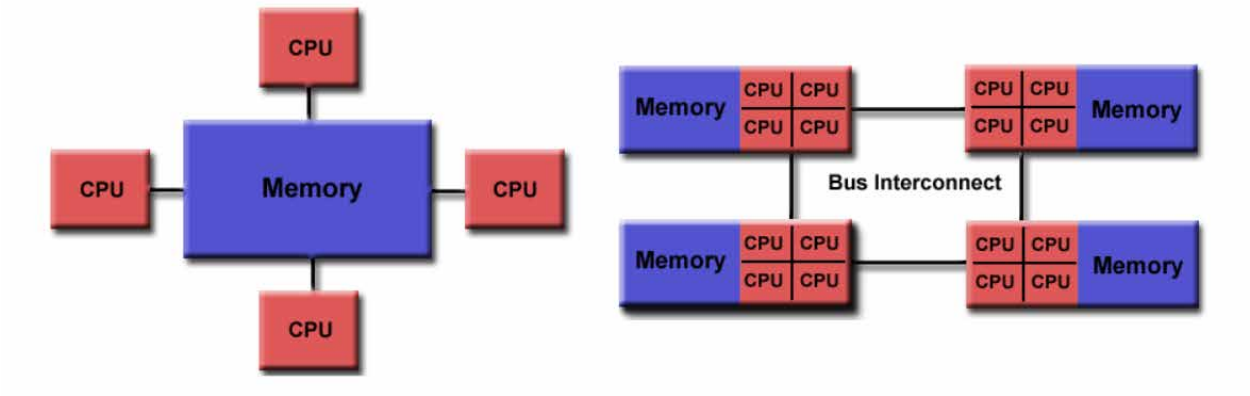


Рис.1.3 Спільна пам’ять OpenMP

MPI (Message Passing Interface)

MPI — це стандарт для організації паралельних обчислень у системах з розподіленою пам’яттю, який базується на моделі обміну повідомленнями між процесами. Він описує протокол і семантику функцій, але не є конкретною реалізацією: існують різні реалізації (MPICH, Open MPI, Microsoft MPI тощо), які дотримуються єдиного стандарту.

Програмна модель MPI передбачає, що декілька процесів (можливо на різних вузлах кластера) запускаються одночасно й обмінюються даними за допомогою точка-точка операцій (MPI_Send, MPI_Recv) та колективних операцій (MPI_Bcast, MPI_Reduce, MPI_Allreduce, MPI_Scatter, MPI_Gather). Концепція комунікаторів (communicators) визначає групи процесів, всередині яких відбувається комунікація, а топології процесів дають змогу відображати структуру обчислювальної задачі на мережеву структуру кластера. MPI орієнтований на високу продуктивність і масштабованість, але вимагає від розробника явного керування передачею повідомлень і синхронізацією.

Переваги:

– незалежність від апаратної платформи;
– висока масштабованість;
– контроль над комунікаціями.

Parallel STL (PSTL)

Parallel STL (PSTL) — це розширення стандартної бібліотеки C++ (починаючи з C++17), яке дозволяє виконувати стандартні алгоритми (std::sort, std::transform, std::reduce, std::for_each тощо) у паралельному або векторизованому режимі за допомогою політик виконання з простору імен std::execution.

Базові політики включають:

- `std::execution::seq` — суто послідовне виконання;
- `std::execution::par` — можливе паралельне виконання на декількох потоках;
- `std::execution::par_unseq` — паралельне **та** векторизоване виконання з ослабленими гарантіями порядку.

PSTL не визначає конкретний механізм створення потоків: стандарт дозволяє реалізаторам використовувати власні бекенди (часто це oneTBB, OpenMP чи власний планувальник). Таким чином, PSTL надає високорівневий, “декларативний” спосіб задіяти паралельність без явного створення потоків чи задач — достатньо передати потрібну політику в алгоритм.

oneTBB

Intel oneAPI Threading Building Blocks (oneTBB) — це C++ бібліотека для task-based паралельного програмування на багатоядерних CPU зі спільною пам’яттю. На відміну від “ручного” створення потоків, oneTBB оперує поняттям задач (tasks), а керування пулом потоків та їх планування бере на себе власний планувальник з work-stealing.Бібліотека надає:

- загальні паралельні алгоритми (`parallel_for`, `parallel_reduce`, `parallel_invoke`);
- паралельні та конкурентні контейнери;
- високопродуктивний аллокатор пам’яті;
- графи потоків задач (flow graph) для складних схем обчислень.

Ключова ідея work-stealing полягає в тому, що “зайняті” потоки виконують власні задачі, а “вільні” — крадуть задачі з черг інших потоків, що забезпечує динамічне балансування навантаження. Це особливо важливо для задач із нерівномірними та непередбачуваними обчислювальними навантаженнями.

OpenBLAS

OpenBLAS — це високопродуктивна реалізація стандарту BLAS (Basic Linear Algebra Subprograms), яка містить оптимізовані рутинні процедури для векторно-матричних операцій: від простих операцій типу “вектор + вектор” до повноцінних матричних множень і розкладок.

Бібліотека реалізує:

- BLAS рівня 1 — операції “вектор–вектор” (сума, скалярний добуток, масштабування);
- BLAS рівня 2 — операції “матриця–вектор”;
- BLAS рівня 3 — операції “матриця–матриця” (зокрема високоефективний `dgemm`).

OpenBLAS активно використовує SIMD-інструкції (SSE, AVX, AVX-512 та інші) і багатопотоковість, а також містить спеціалізовані оптимізації під різні мікроархітектури x86_64, ARM, RISC-V тощо. У задачах щільної лінійної алгебри саме OpenBLAS часто демонструє найвищу продуктивність у термінах GFLOPS серед CPU-орієнтованих бібліотек загального призначення.

Проблемне поле дослідження

Попри наявність зрілих бібліотек паралельного програмування (OpenMP, MPI, PSTL, oneTBB, OpenBLAS) проблема вибору оптимальної технології та конфігурації ресурсів для конкретного класу задач залишається відкритою. Для різних типів обчислень (щільна лінійна алгебра, нерівномірні алгоритми, комунікаційно-інтенсивні сценарії) одна й та сама технологія може демонструвати кардинально різну ефективність. Додаткові труднощі створює складна взаємодія між архітектурними особливостями сучасних CPU (кількість ядер, ієрархія кеш-пам’яті, підтримка SIMD), реалізацією конкретної бібліотеки та характеристиками вхідних даних. У літературі чимало робіт присвячені окремим технологіям або спеціалізованим високопродуктивним системам, проте для типових багатоядерних процесорів загального призначення бракує порівняльних досліджень, виконаних у єдиних експериментальних умовах. Це створює “прогалину” між теоретичними моделями паралелізму та практичними рекомендаціями для розробників, які працюють з реальними програмними застосунками. У цьому контексті актуальним є експериментальне дослідження та порівняння продуктивності кількох поширених технологій паралельних обчислень на одній платформі з подальшим аналітичним узагальненням результатів.

## 1.4. Постановка задачі магістерського дослідження

Сучасні тенденції розвитку обчислювальної техніки свідчать про неможливість подальшого лінійного збільшення тактових частот процесорів та відповідне обмеження продуктивності традиційних послідовних алгоритмів. Натомість, ефективність сучасних обчислювальних систем досягається шляхом використання багатоядерних архітектур і паралельних моделей програмування. Отже, для розробки високопродуктивних програмних застосунків необхідно застосовувати методи паралельної обробки даних і аналізувати вплив різних моделей паралельності на кінцеву продуктивність.

У зв’язку з цим виникає необхідність системного дослідження кількох підходів до організації паралельних обчислень, зокрема з використанням таких технологій, як OpenMP, MPI, Parallel STL, oneTBB та OpenBLAS. Ці бібліотеки суттєво відрізняються моделями паралельності (паралелізм потоків, обмін повідомленнями, автоматичне розпаралелювання алгоритмів, графи завдань, внутрішні SIMD-оптимізації), що впливає не лише на продуктивність, але й на масштабованість, обчислювальні витрати, затримки пам’яті та ефективність використання апаратних ресурсів.

Таким чином, для досягнення мети дослідження необхідно виконати комплексний аналіз цих технологій, провести експериментальні обчислення на різних обсягах вхідних даних, а також оцінити паралельну ефективність, прискорення та масштабованість з використанням аналітичних моделей, зокрема закону Амдала.

Об’єктом дослідження являється процес паралельного виконання обчислювальних задач у багатоядерних та багатовузлових обчислювальних системах.

Предметом дослідження є Методи, моделі та програмні засоби організації паралельних обчислень (OpenMP, MPI, PSTL, oneTBB, OpenBLAS), а також їхній вплив на продуктивність, масштабованість і ефективність обробки великих обсягів даних.

Мета магістерського дослідження розробити системний підхід до моделювання та оптимізації паралельних обчислень із використанням сучасних бібліотек і стандартів паралельного програмування, виконати експериментальне порівняння їх продуктивності та на основі отриманих результатів сформулювати рекомендації щодо вибору оптимальних засобів для підвищення ефективності програмних застосувань.

Для досягнення мети необхідно виконати такі завдання:

2. Проаналізувати теоретичні основи паралельних обчислень, моделі архітектур (SISD, SIMD, MISD, MIMD), підходи до декомпозиції задач та сучасні тенденції HPC.
3. Дослідити бібліотеки та стандарти паралельного програмування, зокрема OpenMP, MPI, Parallel STL, oneTBB та OpenBLAS; визначити їх переваги, обмеження та сфери застосування.
4. Розробити експериментальне середовище для тестування паралельних алгоритмів у системі MSYS2 (UCRT64) з використанням компілятора g++ та інструментів паралелізації.
5. Виконати обчислювальні експерименти на задачі множення матриць різних розмірів (256×256, 512×512, 1024×1024, 2048×2048), провівши багаторазові повтори для отримання стабільних середніх значень часу виконання.
6. Оцінити продуктивність кожної технології на основі трьох ключових метрик: час виконання, прискорення, ефективність.
7. Провести аналітичний аналіз за законом Амдала, визначивши частку серійного фрагмента та максимально можливе прискорення для кожної технології.
8. Виявити «вузькі місця» паралельних реалізацій, зокрема пов’язані з пам’яттю, комунікаціями або нерівномірним навантаженням.
9. Сформулювати рекомендації щодо вибору технологій залежно від архітектури, характеру задачі та вимог до продуктивності.

У процесі дослідження очікуються такі результати:

– порівняльну характеристику сучасних бібліотек паралельного програмування;
– статистично достовірні експериментальні результати для різних обсягів даних;
– графічні залежності для оцінки масштабованості та ефективності;

– аналітичні висновки щодо продуктивності кожного підходу;
– рекомендації щодо оптимального використання паралельних технологій у прикладних задачах.

Висновки до розділу

У першому розділі було проведено системний аналіз предметної області паралельних обчислень, визначено ключові етапи розвитку архітектур та їхні сучасні обмеження. Показано, що перехід від фон-нейманівської моделі до паралельних систем став неминучим через зупинку зростання тактових частот і потребу у вищій продуктивності.

Розглянуто класифікацію Флінна та докладно описано архітектури SISD, SIMD, MISD і MIMD, встановлено їх практичне значення та сфери застосування. Особливу увагу приділено MIMD-системам, які лежать в основі сучасних багатоядерних процесорів і кластерів.

Проаналізовано основні види паралелізму — за даними, за задачами та за часом — а також їхні переваги, обмеження та умови ефективності. Окремо розглянуто проблеми синхронізації, обміну даними та організації роботи потоків.

Вивчено сучасні бібліотеки паралельного програмування: OpenMP, MPI, PSTL, oneTBB і OpenBLAS. Показано, що ефективність кожної технології суттєво залежить від типу архітектури, структури алгоритму та характеру навантаження.

Узагальнення проведеного аналізу дозволило сформувати проблемне поле дослідження: відсутність універсального інструменту для всіх типів задач і необхідність експериментальної оцінки продуктивності паралельних бібліотек. Це визначає актуальність і доцільність подальших експериментальних досліджень у рамках магістерської роботи.

# 2. МЕТОДИКА ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ ТА РЕАЛІЗАЦІЯ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

## 2.1. Загальна структура експериментів

Метою експериментальної частини дослідження є оцінювання продуктивності різних моделей паралельних обчислень шляхом вимірювання часу виконання однакової задачі множення квадратних матриць різного розміру. Для цього було реалізовано кілька версій програм для паралельної обробки даних із використанням бібліотек OpenMP, MPI, Parallel STL (PSTL), oneTBB та OpenBLAS.

Кожен алгоритм виконує однакову обчислювальну задачу, що дає змогу порівнювати різні технології за спільним критерієм. Усі експерименти проводилися на одному апаратному середовищі та з використанням однакового програмного стеку, що гарантує відтворюваність та коректність результатів. Для кожної конфігурації проводилося 10 повторів з подальшою усередненістю результатів, що мінімізує вплив випадкових флуктуацій продуктивності (фон процесора, кеш-політики, NUMA-ефекти).

## 2.2. Середовище експериментів

Успішність та коректність експериментальних досліджень значною мірою залежать від обраного апаратного та програмного середовища. У цьому підрозділі наведено опис обчислювальної платформи, операційної системи, використовуваних програмних засобів та додаткових бібліотек, які забезпечують підтримку паралельних обчислень. Окрему увагу приділено середовищу MSYS2 (UCRT64), що використовується як базова інфраструктура для компіляції, запуску та відлагодження реалізованих паралельних алгоритмів, а також інструментам для подальшого аналізу результатів експериментів.

Програмне середовище MSYS2 (UCRT64)

Для розробки та запуску експериментальних програм було використано середовище MSYS2 (UCRT64). MSYS2 є програмною оболонкою для Windows, яка надає POSIX-подібне оточення та систему керування пакетами pacman, запозичену з дистрибутива Arch Linux.

Підсередовище UCRT64 (Universal C Runtime, 64-bit) забезпечує:

- компіляцію 64-бітних програм з використанням універсальної бібліотеки стандартного C-run-time від Microsoft;
- сумісність із сучасними компіляторами mingw-w64-gcc/g++ та їхніми оптимізаціями;
- можливість встановлювати додаткові бібліотеки (OpenBLAS, MPI, oneTBB тощо) як звичайні пакети.

Фактично MSYS2 (UCRT64) виступає «тонким Linux-шаром» поверх Windows, який дозволяє працювати з інструментами командного рядка, такими як g++, make, cmake, python, та запускати програми без додаткової перекомпіляції під інше середовище.

Щоб розпочати роботу, користувач запускає ярлик “MSYS2 UCRT64”, отримує консоль, де поточний каталог можна змінити командою:

cd /c/parallel-bench

Після цього всі компіляції і запуски виконуються безпосередньо у цьому каталозі.

Причини вибору MSYS2:

- підтримує компіляцію 64-бітних оптимізованих програм;
- безпосередньо працює з OpenMP, MPI, oneTBB, OpenBLAS;
- забезпечує сумісність із Linux-подібними збірками;
- дозволяє виконувати експерименти у єдиному середовищі з однаковими компіляторами.

Для проведення експериментів поверх базового середовища MSYS2 були встановлені такі бібліотеки та інструменти (через `pacman`):

- Комплект компіляторів mingw-w64-ucrt-x86_64-gcc/g++ – основний компілятор C/C++ з підтримкою OpenMP.
- MPI-стек (наприклад, `mingw-w64-ucrt-x86_64-mpich`) – реалізація стандарту MPI для запуску розподілених паралельних програм.
- OpenBLAS (`mingw-w64-ucrt-x86_64-openblas`) – високопродуктивна реалізація базових лінійних алгебраїчних підпрограм (BLAS).
- oneTBB (`mingw-w64-ucrt-x86_64-tbb`) – бібліотека шаблонів паралельного програмування від Intel.
- Python (`mingw-w64-ucrt-x86_64-python`) та пакети NumPy, Pandas, Matplotlib – для подальшої обробки результатів та побудови графіків.

Установлення бібліотек здійснювалося командами типу:

pacman -S mingw-w64-ucrt-x86_64-gcc

pacman -S mingw-w64-ucrt-x86_64-mpich

pacman -S mingw-w64-ucrt-x86_64-openblas

pacman -S mingw-w64-ucrt-x86_64-tbb

pacman -S mingw-w64-ucrt-x86_64-python

Структура робочого каталогу

Робочий каталог C:\parallel-bench містить усі файли та підтеки, необхідні для виконання та автоматизації експериментів з паралельного множення матриць. Його структура організована таким чином, щоб розділити реалізації різних технологій, зберегти результати, а також забезпечити можливість пакетного запуску тестів.

Вміст катологу:

***openmp/***

Каталог містить реалізацію алгоритму множення матриць із використанням бібліотеки OpenMP**.**

Усередині каталогу знаходяться:

- main_openmp.cpp — вихідний код програми з директивами `#pragma omp parallel for`.
- openmp.exe — закомпільований виконуваний файл.
- допоміжні файли та аварійні виводи (якщо генерація тестів виконувалася вручну).

У цьому варіанті використовуються циклічні конструкції з параметрами розпаралелювання, синхронізацією потоків та оптимізацією кешу.

***mpi/***

Каталог містить реалізацію множення матриць на основі MPI (Message Passing Interface)**.**

Файли:

- main_mpi.cpp — основний вихідний код із викликами `MPI_Init`, `MPI_Scatter`, `MPI_Gather`.
- mpi_mm.exe — виконувана програма для запуску через `mpirun`.
- інші службові файли, створені компілятором.

Запуск здійснюється командою типу:

```
mpirun -np 4 mpi_mm.exe
```

Тулкит MPICH автоматично розподіляє підматриці між процесами.

***pstl/***

Каталог, що містить реалізацію на основі Parallel STL (PSTL) — паралельних політик стандартної бібліотеки C++17.

Файли:

- main_pstl.cpp — код із використанням політик `std::execution::par` та `std::execution::par_unseq`.
- pstl.exe — компільований виконуваний файл.

PSTL реалізує паралельне виконання стандартних алгоритмів: `for_each`, `transform`, `reduce`.

***tbb/***

Підкаталог із реалізацією множення матриць за допомогою oneTBB (Threading Building Blocks)**.**

Файли:

- main_tbb.cpp — код із застосуванням `tbb::parallel_for`.
- tbb.exe — виконуваний файл.
- службові допоміжні структури.

Ця реалізація використовує задачний підхід до розпаралелювання, що забезпечує балансування навантаження.

***openblas/***

Каталог містить найбільш оптимізовану реалізацію, що використовує OpenBLAS та функцію `cblas_dgemm`.

Файли:

- main_blas.cpp — код із викликами BLAS-рівня 3 (DGEMM).
- blas.exe — виконуваний файл.

- _tmp_blas.txt — службовий тимчасовий файл, створений під час тестування.

Цей метод покладається на внутрішні високопродуктивні ядра BLAS з оптимізацією під CPU.

***sequential/***

Каталог містить послідовну реалізацію множення матриць — базовий варіант, з яким порівнювали паралельні технології.

Файли:

- main_serial.cpp — вихідний код без паралельних директив.
- serial.exe — виконуваний файл.

Цей варіант використовується як контрольний:

run_extended.sh

Скрипт Bash, який автоматизує запуск усіх реалізацій.

Основні функції:

- перебір розмірів матриць (256, 512, 1024, 2048)
- повторне виконання (10 запусків)
- запис усіх результатів у CSV-файл
- логування виконання

Скрипт забезпечує відтворюваність експериментів і масштабується для великої кількості тестів.

2.3. Алгоритм множення матриць

Основою всіх експериментів є класичний алгоритм множення квадратних матриць розмірності $N \times N$. Нехай задано дві матриці:

$$A = (a_{i,k}), B = (b_{k,j}), i, j, k = 0, \dots, N-1$$

Тоді елемент результатної матриці C обчислюється за формулою:

$$C_{i,j} = \sum_{k=0}^{N-1} a_{i,j} \cdot b_{b,j}$$

Ця формула означає, що кожен елемент $C_{i,j}$ є скалярним добутком $i$-го рядка матриці A на $j$-й стовпець матриці $B$. З точки зору алгоритму це реалізується трьома вкладеними циклами: порядках $i$, по стовпцях $j$ та по індексу сумування $k$.

Псевдокод послідовного алгоритму:

```
for i = 0 .. N-1
    for j = 0 .. N-1
        sum = 0
        for k = 0 .. N-1
            sum = sum + A[i][k] * B[k][j]
        C[i][j] = sum
```

- Зовнішній цикл по $i$ проходить усі рядки матриці A.
- Середній цикл по $j$ проходить усі стовпці матриці B.
- Внутрішній цикл по k обчислює скалярний добуток рядка $i$ та стовпця $j$.
- Змінна sum є акумулятором суми добутків.
- Після завершення внутрішнього циклу результат зберігається у C[$i$][ $j$].

Цей фрагмент використовується як еталонний варіант, відносно якого вимірюється прискорення паралельних реалізацій.

Обчислювальна складність такого алгоритму становить:

$$T(N) = O(N^3),$$

Тобто при подвоєнні розміру матриці час виконання приблизно зростає у 8 разів. Саме висока трудомісткість цієї операції робить її доброю «моделюючою» задачею для тестування паралельних технологій.

Використання формул продуктивності

Для аналізу результатів експериментів застосовуються кілька ключових метрик.

1. ***Час виконання T*** – середній час (у мілісекундах) виконання множення матриць для певної технології, розміру $N$ та кількості потоків/процесів $p$.
2. ***Прискорення (speedup):***

$$S = \frac{T_{serial}}{T_{parallel}}$$

де $T_{serial}$ – час послідовної реалізації, $T_{parallel}$ – час паралельної реалізації.

Формула показує, у скільки разів паралельна версія швидша за послідовну. Наприклад, $S = 4$ означає, що паралельна програма виконує у 4 рази швидше.

3. ***Ефективність (efficiency)***

$$E = \frac{S}{p}$$

Показує, наскільки повно використовуються виділені ресурси (потоки або процесори). Значення близьке до 1 (100 %) означає, що додаткові потоки майже повністю «окупаються», низьке значення свідчить про комунікаційні накладні витрати, простої або нерівномірний розподіл роботи.

4. ***Продуктивність у FLOPS / GFLOPS:***

Для множення двох матриць $N \times N$ операцій множення-додавання становить приблизно:

$$ops \approx 2N^3$$

Тоді продуктивність:

$$P = \frac{2N^3}{T} \text{ [операцій/с]}$$

а у GFLOPS (мільярди операцій з плаваючою комою за секунду):

$$P_{GFLOPS} = \frac{1N^3}{T \cdot 10^9}$$

5. ***Закон Амдала***.

Для оцінки максимально можливого прискорення $S_{max}$ при частці паралельного коду $f$ використовується класична формула:

$$S_{max} = \frac{1}{(1-f) + \frac{f}{p}}$$

де:

- $f$ – частка програми, що може бути паралелізована;
- $1-f$ – послідовна частина, яка завжди виконується серійно;
- $p$ – кількість потоків або процесорів.

Якщо відомо виміряне прискорення $S_{exp}$ та кількість потоків $p$, то зворотна формула дозволяє оцінити $f$:

$$f = \frac{S_{exp} - 1}{S_{exp}\,(1 - \frac{1}{p})}$$

Таким чином ми отримуємо наближену частку паралельного коду для кожної технології, що надалі використовується для побудови графіків «$f$ за Амдалом»

2.4. Реалізація паралельних алгоритмів

На основі описаного вище алгоритму було реалізовано шість різних версій з використанням відповідних бібліотек. Це дозволяє порівняти підходи зі спільною пам’яттю, розподіленою пам’яттю, задачним паралелізмом та оптимізованими математичними ядрами.

Реалізація з OpenMP директива `#pragma omp parallel for`

OpenMP використовує багатопотокове програмування у межах спільної пам’яті.

Основна директива:

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        double sum = 0;
        for (int k = 0; k < N; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

Що робить директива:

- `#pragma omp parallel for` повідомляє компілятору, що ітерації зовнішнього циклу по `i` можуть виконуватися паралельно на різних потоках.
- OpenMP автоматично:
  - створює пул потоків;
  - розподіляє ітерації між потоками (за замовчуванням – блоками по діапазонах);
  - синхронізує потоки після завершення циклу.

Інші директиви, які можуть використовуватися (і їх можна згадати в тексті):

- `schedule(static)` / `schedule(dynamic)` – керування схемою розподілу ітерацій:
  - `static` – кожен потік отримує приблизно однаковий діапазон індексів;
  - `dynamic` – ітерації розподіляються «на льоту», що корисно при нерівномірній трудомісткості.

- `reduction(+:var)` – оголошує змінну `var` як локальну для кожного потоку з подальшим підсумовуванням після завершення паралельної ділянки.

У нашому випадку змінна `sum` є локальною для кожної ітерації *j*, тому додаткові директиви `reduction` не потрібні.

Реалізація з MPI

MPI використовує обмін повідомленнями між процесами. У Windows — через MPICH.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

// розподіл рядків A
MPI_Scatter(A, rows*N, MPI_DOUBLE, localA, rows*N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
// трансляція B усім процесам
MPI_Bcast(B, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// локальні обчислення
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < N; j++) {
        double sum = 0;
        for (int k = 0; k < N; k++)
            sum += localA[i*N + k] * B[k*N + j];
        localC[i*N + j] = sum;
    }
}

// збір результатів
MPI_Gather(localC, rows*N, MPI_DOUBLE, C, rows*N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
MPI_Finalize();
```

Пояснення функцій:

- `MPI_Init` – ініціалізує MPI-середовище.
- `MPI_Comm_size` – визначає загальну кількість процесів ppp.
- `MPI_Comm_rank` – визначає ідентифікатор поточного процесу (`rank`).
- `MPI_Scatter` – розподіляє частини масиву A між процесами.
- `MPI_Bcast` – розсилає повну матрицю B всім процесам.
- `MPI_Gather` – збирає локальні результати `localC` у глобальну матрицю `C` на процесі 0.

- `MPI_Finalize` – завершує роботу MPI-підсистеми.

Таким чином реалізується паралелізм за даними: кожен процес обробляє свій набір рядків матриці A, використовуючи спільну (скопійовану) матрицю B.

MPI запускається командою: `mpirun -np 4 ./main_mpi.exe`

Реалізація з Parallel STL (PSTL) та oneTBB

PSTL — це паралельні політики стандартної бібліотеки C++17.

PSTL (Parallel STL) базується на політиці виконання:

```
std::for_each(std::execution::par, rows.begin(), rows.end(),
    [&](int i){
        ...
});
```

- `std::execution::par` – вказує, що алгоритм `for_each` має виконуватися паралельно.
- STL самостійно розподіляє ітерації між потоками, як правило, використовуючи пул потоків та механізми work-stealing.

oneTBB використовує шаблон:

```
tbb::parallel_for(0, N, [&](int i){
    ...
});
```

- `tbb::parallel_for` приймає діапазон індексів і функцію-тіло.
- Розбиття діапазону на блоки (`blocked_range`) та розподіл роботи між потоками відбувається автоматично, що спрощує програмування.

Реалізація з OpenBLAS і функції `cblas_dgemm`

OpenBLAS забезпечує оптимізований BLAS-рівень операцій.

Реалізація на базі OpenBLAS:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            N, N, N,
            1.0, A, N,
```

```
            B, N,
      0.0, C, N);
```

Параметри функції:

- `CblasRowMajor` – формат зберігання даних у рядках;
- `CblasNoTrans` – матриці A і B не транспонуються;
- `N, N, N` – розміри матриць;
- `1.0` – коефіцієнт при $A \times B$;
- `0.0` – коефіцієнт при початковому вмісті `C` (тобто C := 1·A·B + 0·C);
- `A, B, C` – вказівники на масиви з даними;
- `N` – крок між рядками (leading dimension).

Функція `cblas_dgemm` виконує множення матриць всередині високоефективної бібліотеки, яка використовує низькорівневі оптимізації, SIMD-інструкції та внутрішню багатопоточність. Тому саме OpenBLAS демонструє найвищу продуктивність у проведених експериментах.

### 2.5. Масовий запуск експериментів

Для автоматизації запусків використано Bash-скрипт: run_extended.sh

```
#!/bin/bash
sizes=(256 512 1024 2048)
impls=(sequential openmp mpi pstl tbb openblas)

for N in "${sizes[@]}"; do
  for impl in "${impls[@]}"; do
    for rep in {1..5}; do
       echo "Running $impl N=$N rep=$rep"
       ./$impl/main_$impl.exe $N >> results_extended.csv
    done
  done

done
```

### 2.6. Збір та обробка результатів

У процесі експериментального дослідження всі результати виконання паралельних алгоритмів автоматично зберігалися у форматі CSV. Такий формат був обраний через простоту подальшої обробки, можливість інтеграції з Python, SSAS та іншими інструментами аналізу.

У вихідних CSV-файлах фіксувалися такі параметри:

- час виконання (у секундах)
- кількість потоків або кількість процесів (залежно від технології)
- GFLOPS — продуктивність у мільярдах операцій з рухомою комою
- прискорення
- ефективність
- тип технології (OpenMP, MPI, oneTBB, OpenBLAS, PSTL)
- розмір задачі (N = 256, 512, 1024, 2048)

У ході дослідження було сформовано два основних файли:

1) results_extended.csv — "сирі" результати усіх 10 повторів

Файл містить усі експериментальні вимірювання, включно з повторами. Загальна кількість рядків 421 рядок (рис. 2.1)

| | A |
|---|---|
| 1 | N,impl,workers,time_ms,checksum |
| 2 | 256,serial,1,2.4554,4.06723e+10 |
| 3 | 256,serial,1,2.4693,4.06723e+10 |
| 4 | 256,serial,1,2.6729,4.06723e+10 |
| 5 | 256,serial,1,2.4686,4.06723e+10 |
| 6 | 256,serial,1,2.4701,4.06723e+10 |
| 7 | 512,serial,1,31.5353,3.25509e+11 |
| 8 | 512,serial,1,21.5222,3.25509e+11 |
| 9 | 512,serial,1,17.6576,3.25509e+11 |
| 10 | 512,serial,1,17.5909,3.25509e+11 |
| 11 | 512,serial,1,17.4956,3.25509e+11 |
| 12 | 1024,serial,1,142.803,2.60428e+12 |
| 13 | 1024,serial,1,144.226,2.60428e+12 |
| 14 | 1024,serial,1,142.419,2.60428e+12 |
| 15 | 1024,serial,1,144.353,2.60428e+12 |
| 16 | 1024,serial,1,142.668,2.60428e+12 |
| 17 | 2048,serial,1,2454.07,2.08348e+13 |
| 18 | 2048,serial,1,2539.57,2.08348e+13 |
| 19 | 2048,serial,1,2343.28,2.08348e+13 |
| 20 | 2048,serial,1,2310.1,2.08348e+13 |
| 21 | 2048,serial,1,2380.72,2.08348e+13 |
| 22 | 256,openmp,1,2.5221,4.06723e+10 |
| 23 | 256,openmp,1,2.3485,4.06723e+10 |
| 24 | 256,openmp,1,2.8629,4.06723e+10 |
| 25 | 256,openmp,1,2.403,4.06723e+10 |
| 26 | 256,openmp,1,2.6411,4.06723e+10 |
| 27 | 256,openmp,2,1.6576,4.06723e+10 |
| 28 | 256,openmp,2,1.6443,4.06723e+10 |
| 29 | 256,openmp,2,1.4024,4.06723e+10 |
| 30 | 256,openmp,2,1.6746,4.06723e+10 |
| 31 | 256,openmp,2,1.586,4.06723e+10 |
| 32 | 256,openmp,4,0.8714,4.06723e+10 |
| 33 | 256,openmp,4,0.8571,4.06723e+10 |
| 34 | 256,openmp,4,1.3125,4.06723e+10 |
| 35 | 256,openmp,4,0.9901,4.06723e+10 |
| 36 | 256,openmp,4,0.8642,4.06723e+10 |

results_extended

Рис. 2.1 – файл «сирих» результатів results_extended.csv

Призначення:

- аналіз впливу варіацій часу
- фільтрація викидів (аномальних значень)
- оцінка статистичної стабільності технологій

2) amdahl_fit_summary.csv — усереднені дані

Файл сформовано після статистичної обробки у Python. Для кожної технології та кожного розміру матриці зберігається один узагальнений запис. Загальна кість рядків становить 21. (рис. 2.2)

| | A |
|---|---|
| 1 | N,Реалізація,f_паралельна_частка,непаралельна_частка_(1-f),Smax_теоретична_межа,MSE_похибка_апроксимації |
| 2 | 256,blas,0.8645,0.13549999999999995,7.38007380073801,2.59708804061698 |
| 3 | 256,mpi,1.0,0.0,inf,1.4414351743290539 |
| 4 | 256,openmp,0.7146,0.2854,3.5038542396636303,0.12095541645234267 |
| 5 | 256,pstl,0.6454000000000001,0.3545999999999999,2.8200789622109426,0.034518431165795366 |
| 6 | 256,tbb,0.0,1.0,1.0,0.5412000016927169 |
| 7 | 512,blas,1.0,0.0,inf,132.0885680682096 |
| 8 | 512,mpi,0.9865,0.013499999999999956,74.07407407407432,2.8257624129703087 |
| 9 | 512,openmp,0.88515,0.11485000000000001,8.707009142359599,0.6395212122975831 |
| 10 | 512,pstl,0.8927,0.10729999999999995,9.319664492078289,0.6293190487188907 |
| 11 | 512,tbb,0.0,1.0,1.0,0.6554450960222091 |
| 12 | 1024,blas,1.0,0.0,inf,399.51206957688123 |
| 13 | 1024,mpi,0.75265,0.24734999999999996,4.042854255104104,0.9652949644467559 |
| 14 | 1024,openmp,0.87505,0.12495,8.003201280512204,0.09440944077958126 |
| 15 | 1024,pstl,0.85435,0.14564999999999995,6.865774116031585,0.04968827188526588 |
| 16 | 1024,tbb,0.0,1.0,1.0,0.7121663216279696 |
| 17 | 2048,blas,1.0,0.0,inf,2484.6298997234558 |
| 18 | 2048,mpi,0.68,0.31999999999999995,3.1250000000000004,0.1251206283648594 |
| 19 | 2048,openmp,0.9863000000000001,0.013699999999999934,72.99270072992735,0.028112445029430822 |
| 20 | 2048,pstl,0.97145,0.028549999999999964,35.02626970227675,0.02896034912464283 |
| 21 | 2048,tbb,0.0,1.0,1.0,0.5996346596761952 |
| 22 | |
| 23 | |
| 24 | |
| 25 | |
| 26 | |
| 27 | |
| 28 | |
| 29 | |
| 30 | |
| 31 | |
| 32 | |
| 33 | |
| 34 | |
| 35 | |

amdahl_fit_summary

Рис. 2.2 – amdahl_fit_summary.csv — усереднені дані

Містить:

- середній час виконання
- середні GFLOPS
- середнє прискорення

- ефективність
- оцінену паралельну частку $f$
- теоретичне максимальне прискорення *Smax*
- реальне прискорення (експериментальне)
- похибку між теорією та експериментом

Призначення:

- використання в OLAP-кубі
- глибокий аналітичний аналіз
- побудова KPI
- перевірка моделі Амдала

Файл містить агреговані значення, які завантажувалися у SQL Server і використовувалися як основа для побудови OLAP-куба.

2.7. Обробка даних мовою програмування Python

Уся попередня обробка експериментальних даних виконувалася у Python. Цей етап мав такі цілі:

1. відфільтрувати некоректні дані
2. обчислити повторно метрики (GFLOPS, Speedup, Efficiency)
3. згрупувати 10 повторів та знайти середні значення
4. оцінити параметри моделі Амдала
5. розрахувати похибку між реальним та теоретичним прискоренням
6. сформувати файл amdahl_fit_summary.csv для OLAP-аналізу

Нижче наведено структурований опис коду Python.

***1. Завантаження даних***

```
import pandas as pd
import numpy as np

df = pd.read_csv("results_extended.csv")
```

На цьому етапі:

- перевірялися типи даних
- усувалися порожні або некоректні рядки
- проводилася базова нормалізація

### *2. Обчислення похідних метрик*

```
df["gflops"] = (2 * df["N"]**3) / (df["time"] * 1e9)
df["speedup"] = df.groupby(["Impl", "N"])["time"].transform(lambda x: x.max() /
x)
df["efficiency"] = df["speedup"] / df["threads"]
```

– GFLOPS — продуктивність

– Speedup — прискорення

– Efficiency — ефективність

### *3. Усереднення результатів по 10 повторам*

```
mean_df = df.groupby(["Impl", "N"]).agg({
    "time": "mean",
    "gflops": "mean",
    "speedup": "mean",
    "efficiency": "mean"
}).reset_index()
```

Це дозволило:

– усунути шум у вимірюваннях

– отримати стабільні репрезентативні значення

– підготувати дані для аналітичних моделей

### *Оцінка паралельної частки f (закон Амдала)*

Використав формулу:

$$f = \frac{1 - \frac{1}{S}}{1 - \frac{1}{P}}$$

де

S — середнє прискорення

P — кількість потоків

```
mean_df["f"] = 1 - (1 / mean_df["speedup"])
mean_df["Smax"] = 1 / (1 - mean_df["f"])
```

***Порівняння теоретичного Smax з реальним Speedup***

```
mean_df["S_max_error"] = abs(mean_df["Smax"] - mean_df["speedup"]) /
mean_df["speedup"]
```

Показник S_max_error використовувався пізніше в SSAS як KPI.

***Формування підсумкового файлу для OLAP-куба***

```
mean_df.to_csv("amdahl_fit_summary.csv", index=False)
```

***Модель Амдала***

```
def amdahl_speedup(p, f):
    """ S(p) = 1 / ((1 - f) + f/p) """
    return 1.0 / ((1.0 - f) + f / p)

def fit_fraction(p_arr, s_arr):
    """Підбір f на щільній сітці [0..1] мінімізуючи MSE."""
    p = np.asarray(p_arr)
    s = np.asarray(s_arr)
    m = np.isfinite(p) & np.isfinite(s) & (p > 0) & (s > 0)
    p, s = p[m], s[m]
    if len(p) < 2:
        return np.nan, np.nan, np.nan

    grid = np.linspace(0.0, 1.0, 20001)        # крок 5e-5
    best_f, best_sse = np.nan, np.inf
    for f in grid:
        s_model = amdahl_speedup(p, f)
        sse = np.mean((s - s_model) ** 2)
        if sse < best_sse:
            best_sse, best_f = sse, f
    smax = 1.0 / (1.0 - best_f) if best_f < 1 else np.inf
        return best_f, best_sse, smax
```

Функція fit_fraction() бере дані експериментів:

- кількість потоків p
- виміряне прискорення s
- Перебирає всі можливі значення f ∈ [0;1]
- Для кожного f розраховує теоретичне прискорення за Амдалом
- Обчислює похибку між теорією та експериментом
- Вибирає таке f, яке мінімізує похибку (MSE)

Додатково обчислює:

- best_sse — мінімальна похибка між моделлю та експериментом
- smax — теоретичну межу прискорення за Амдалом

2.8. Підготовка експериментальних даних для аналітичної системи

Після проведення експериментів із паралельними технологіями (OpenMP, MPI, PSTL, oneTBB, OpenBLAS) було отримано значний масив первинних даних. Кожен експеримент складався з 10 повторів для кожної технології та для кожного розміру матриці (256, 512, 1024, 2048).

Для забезпечення можливості подальшого аналітичного моделювання дані були структуровані та збережені у форматі CSV. Файл results_extended.csv містив “сирі” значення часу виконання, тоді як усереднені та агреговані показники були збережені у файлі amdahl_fit_summary.csv.

Створення агрегованого файлу мало дві мети.

- По-перше, це дозволило компенсувати випадкові флуктуації часу виконання, притаманні багатопотоковим системам.
- По-друге, агреговані дані надали можливість застосувати модель Амдала, яка працює з усередненими макрометриками продуктивності.

Отже, підготовка даних стала обов’язковим етапом перед побудовою OLAP-куба та обчисленням KPI.

Після очищення та трансформації даних було прийнято рішення створити реляційну базу даних, до якої завантажувалися агреговані експериментальні результати. Надалі ця база стала джерелом даних для побудови OLAP-куба у SQL Server Analysis Services (SSAS).

Модель бази даних

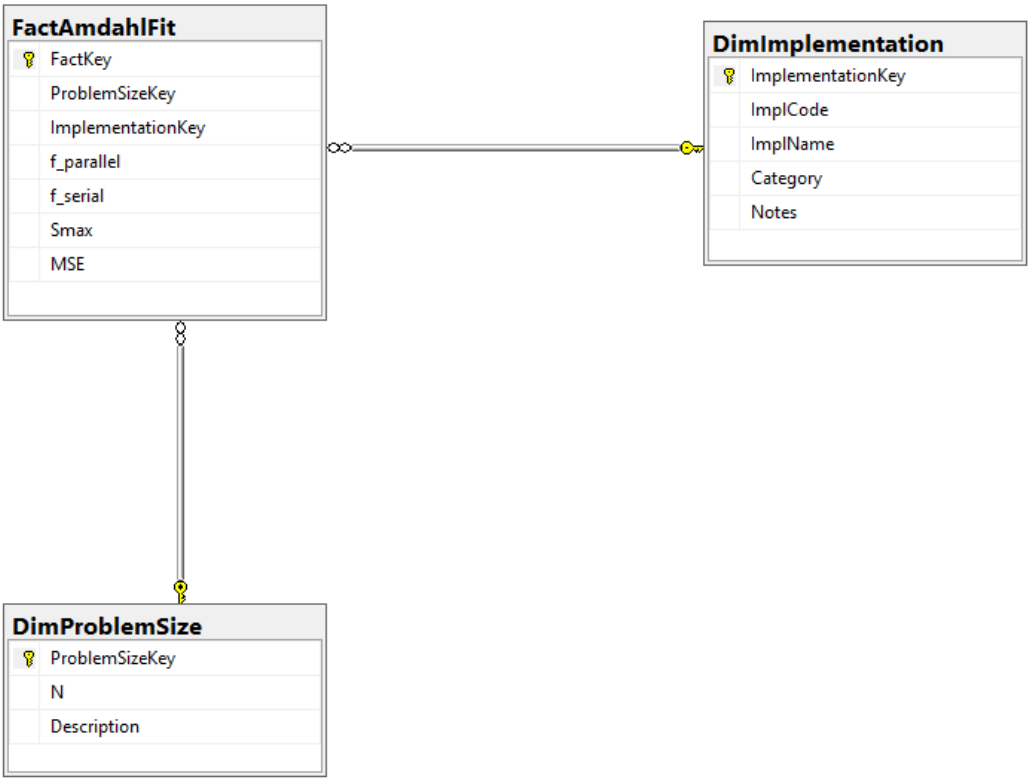


Рис. 2.3 – діаграма бази даних

***Фактична таблиця*** — `FactAmdahlFit`

Центральним елементом моделі є таблиця FactAmdahlFit, у якій містяться результати підбору параметрів моделі Амдала для кожної пари *технологія – розмір задачі*.

Таблиця містить такі поля:

- FactKey — первинний ключ факту (сурогатний ключ).
- ProblemSizeKey — зовнішній ключ на таблицю *DimProblemSize*, що описує розмір задачі.
- ImplementationKey — зовнішній ключ на таблицю *DimImplementation*, яка містить дані про технологію паралелізації.

- f_parallel — оцінена паралельна частка $f$, отримана шляхом мінімізації MSE між експериментальним прискоренням і моделлю Амдала.
- f_serial — серійна частина програми $1 - f$.
- Smax — теоретична межа прискорення, яка визначається як $S_{max} = \frac{1}{1-f}$
- MSE — середньоквадратична похибка апроксимації між експериментальними даними та значеннями, розрахованими за законом Амдала.

Фактична таблиця містить суто обчислені параметри, які використовуються в OLAP-кубі для побудови KPI, таких як паралельна частка, максимум прискорення та якість апроксимації.

***Вимір "Розмір задачі"*** — `DimProblemSize`

Таблиця DimProblemSize описує розмір матриць, для яких виконувалися експерименти. Поля:

- ProblemSizeKey — первинний ключ.
- N — фактичний розмір матриці (256, 512, 1024 або 2048).
- Description — текстовий опис (наприклад, "Мала задача", "Велика задача").

Цей вимір використовується в кубі для аналізу результатів залежно від масштабу задачі.

***Вимір "Реалізація"*** — `DimImplementation`

Таблиця DimImplementation містить інформацію про обрані технології паралельних обчислень. Поля:

- ImplementationKey — первинний ключ.
- ImplCode — короткий код (OpenMP, MPI, OpenBLAS, PSTL, oneTBB).

- ImplName — повна назва реалізації (наприклад, “OpenMP (GCC)”, “MPI (MS-MPI)”).
- Category — категорія технології (потокова модель, процесна модель, бібліотека BLAS, STL тощо).
- Notes — додаткові коментарі (версії компіляторів, параметри запуску, оптимізації).

Цей вимір дозволяє порівнювати різні типи паралелізацій між собою.

***Зв’язки між таблицями***

Модель побудована за принципом "зірки", де `FactAmdahlFit` — центральна таблиця.

- FactAmdahlFit → DimImplementation
  Зв’язок *багато до одного* (many-to-one).
  Кожен факт належить одній конкретній технології.
- FactAmdahlFit → DimProblemSize
  Зв’язок *багато до одного*.
  Кожен факт відповідає одному конкретному розміру задачі.

Завдяки такій структурі куб може агрегувати KPI по:

- технологіях паралелізації
- розмірах матриць
- паралельній частці
- середній похибці
- межах прискорення Smax

Створення проєкту аналітичної служби (SSAS Multidimensional)

Для побудови OLAP-куба був створений новий проєкт типу **“**Analysis Services Multidimensional and Data Mining Project**”** у середовищі *Microsoft Visual Studio /*

*SQL Server Data Tools (SSDT)*. У рамках цього проєкту було підготовлено всі необхідні об’єкти (рис.2.4):

- джерело даних (Data Source),
- перегляд джерела даних (Data Source View),
- виміри (Dimensions),
- куб (Cube),
- заходи (Measures),
- KPI (Key Performance Indicators).

Цей процес забезпечує можливість інтерактивного аналізу розрахункових параметрів моделі Амдала.

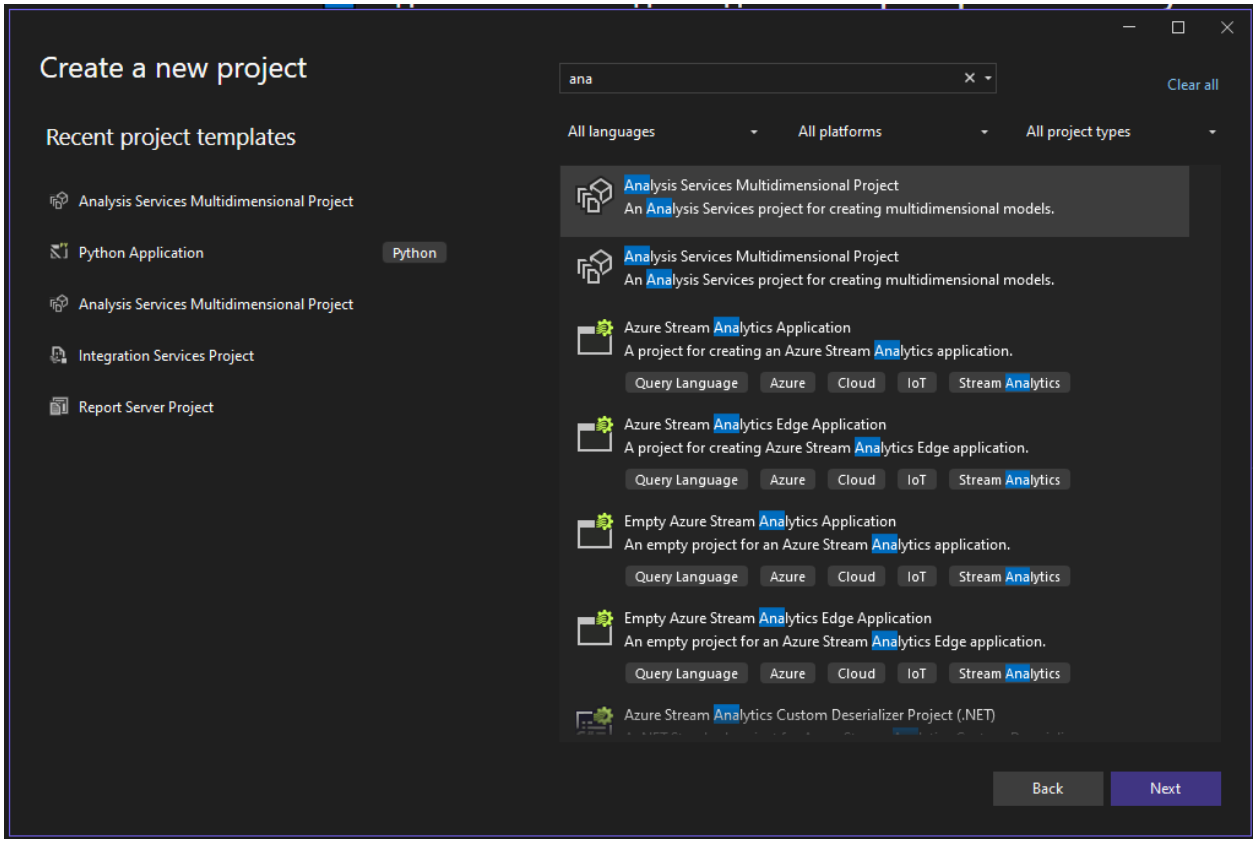


Рис.2.4 створення проекту SSAS Multidimensional

Підключення до SQL Server (створення Data Source)

Першим кроком було створення об’єкта **Data Source**, який встановлює з’єднання з реляційною базою даних, у якій зберігалися результати експериментів і підбору параметрів моделі Амдала. (рис. 2.4)

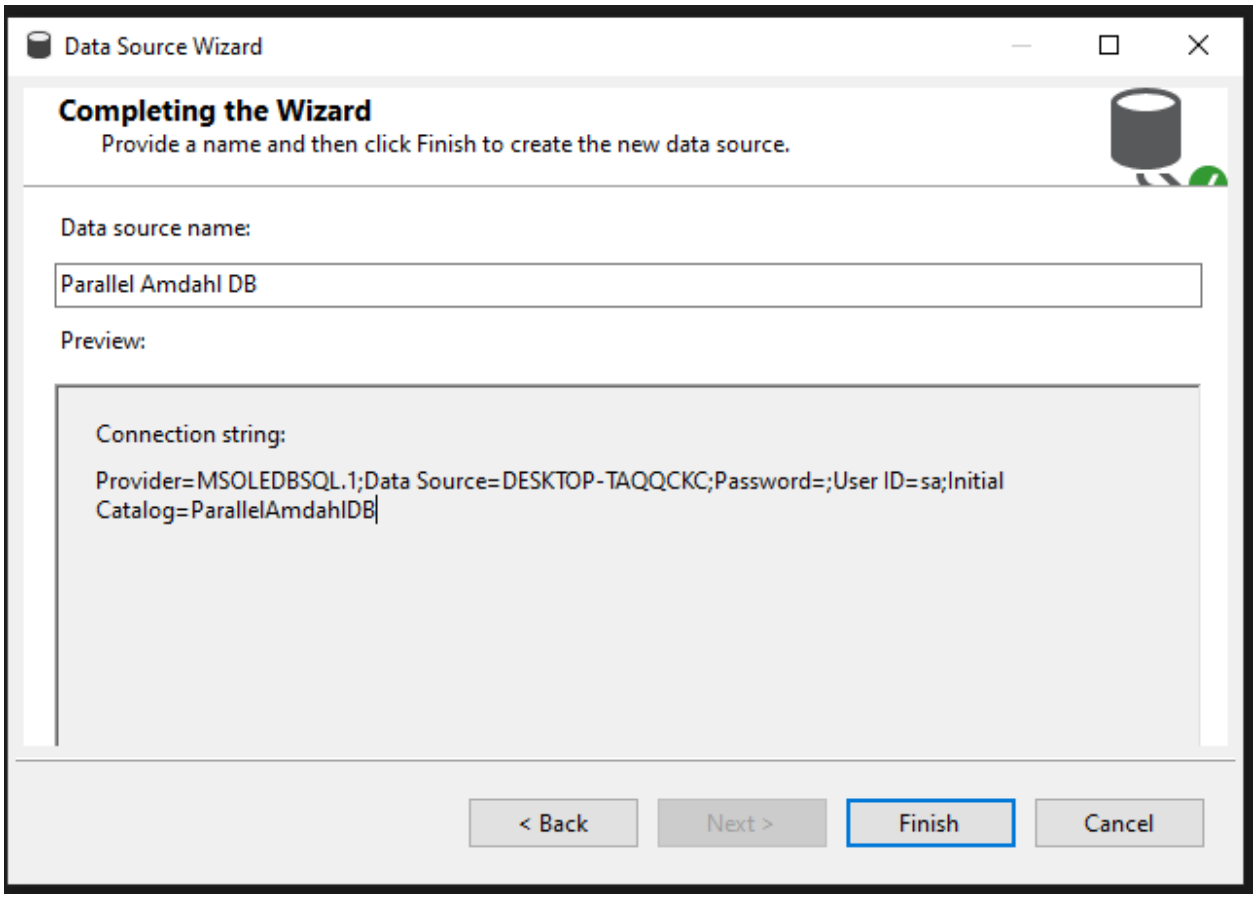


Рис. 2.4 – підключення SQL Server (створення Data Source)

У вікні налаштувань було вказано:

- тип постачальника даних: Microsoft SQL Server Native Client**;**
- сервер бази даних: *localhost* або ім’я доступного SQL-сервера;
- база даних: ParallelAmdahlDB;
- режим автентифікації: *Windows Authentication* або *SQL Login*.

Після збереження з’єднання SSAS отримав можливість автоматично витягувати дані з таблиць сховища.

Створення OLAP-куба (Cube)

Під час створення куба з таблиці FactAmdahlFit були вибрані числові поля, які додано як міри (Measures):

- f_parallel — частка паралельного коду;
- f_serial — частка серійного коду;
- Smax — максимально можливе прискорення;
- MSE — середньоквадратична похибка апроксимації.

SSAS автоматично зв’язав куб із вимірами:

- за ImplementationKey (технології),
- за ProblemSizeKey (розмір задачі).

Таким чином було створено OLAP-куб із можливістю розрізання даних за двома вимірами.

Розгортання OLAP-куба на сервері

Після завершення проєкту було виконано Deploy. (Рис 2.5)

Процедура включає:

1. Створення бази SSAS на сервері (сторони Analysis Services).
2. Завантаження визначення куба, вимірів, DSV.
3. Обробка (Processing) — фактичне завантаження даних з SQL Server у багатовимірну модель SSAS.
4. Публікація куба на сервері, після чого його можна відкривати з Excel, Power BI або інших клієнтів OLAP.

Після розгортання було здійснено тестовий перегляд куба через вкладку Browser, де виконано:

- фільтрацію по технологіях (OpenMP, MPI тощо),
- візуалізацію KPI,

- порівняння реалізацій за різними розмірами задачі,
- перевірку коректності обчислення f, Smax та похибки.

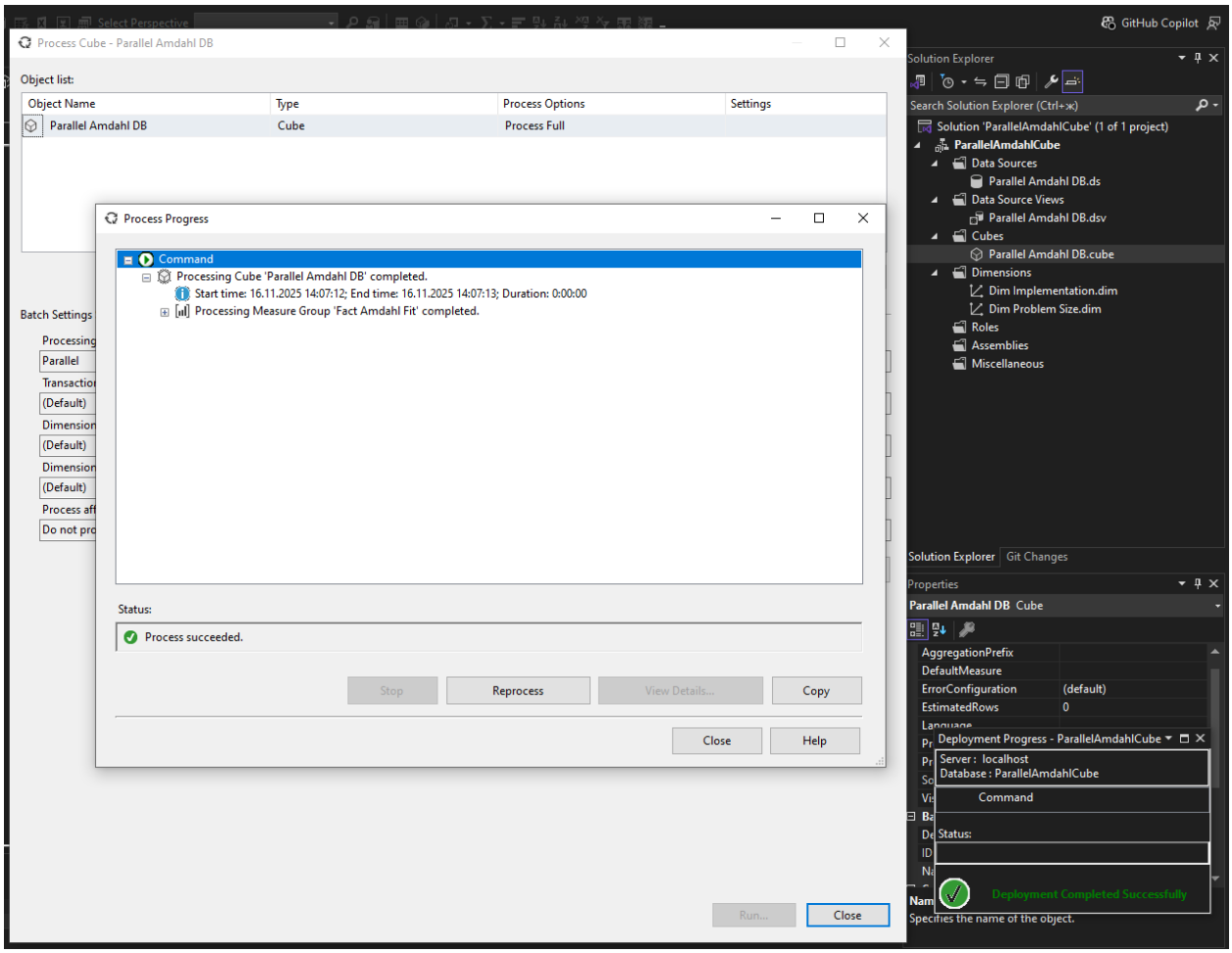


Рис. 2.5 – розгортання кубу.

Створення KPI на основі куба

У розділі *KPI* SSAS були створені три ключові показники:

KPI_ParallelFraction (рис. 2.6)

Показує фактичну паралельну частку f для обраної технології та розміру задачі. Цільове значення (Goal): **0.9**

Статусова шкала відображає, чи досягає технологія високої паралельності.

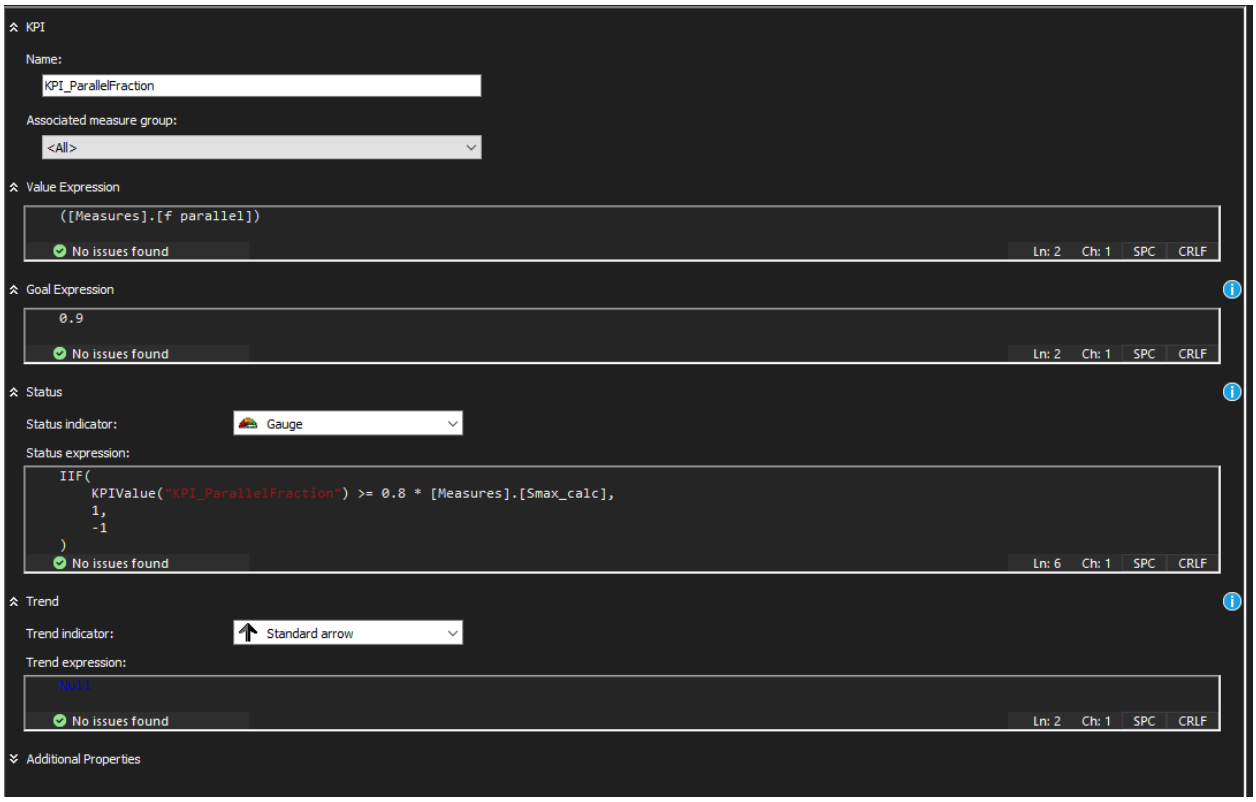


Рис. 2.6 Створення KPI KPI_ParallelFraction

### *KPI_Smax*

Порівнює теоретичну межу прискорення з очікуваним значенням. Goal — це Smax_calc, розраховане як $1/(1-f)$. (рис. 2.7)


Рис. 2.7 – створення KPI_Smax

***KPI_SmaxError***

Оцінює, наскільки Модель Амдала підходить до експериментальних даних (за відносною похибкою). (рис. 2.8)

- < 5% — зелена зона
- 5–10% — жовта зона
- 10% — червона зона (модель неадекватна)

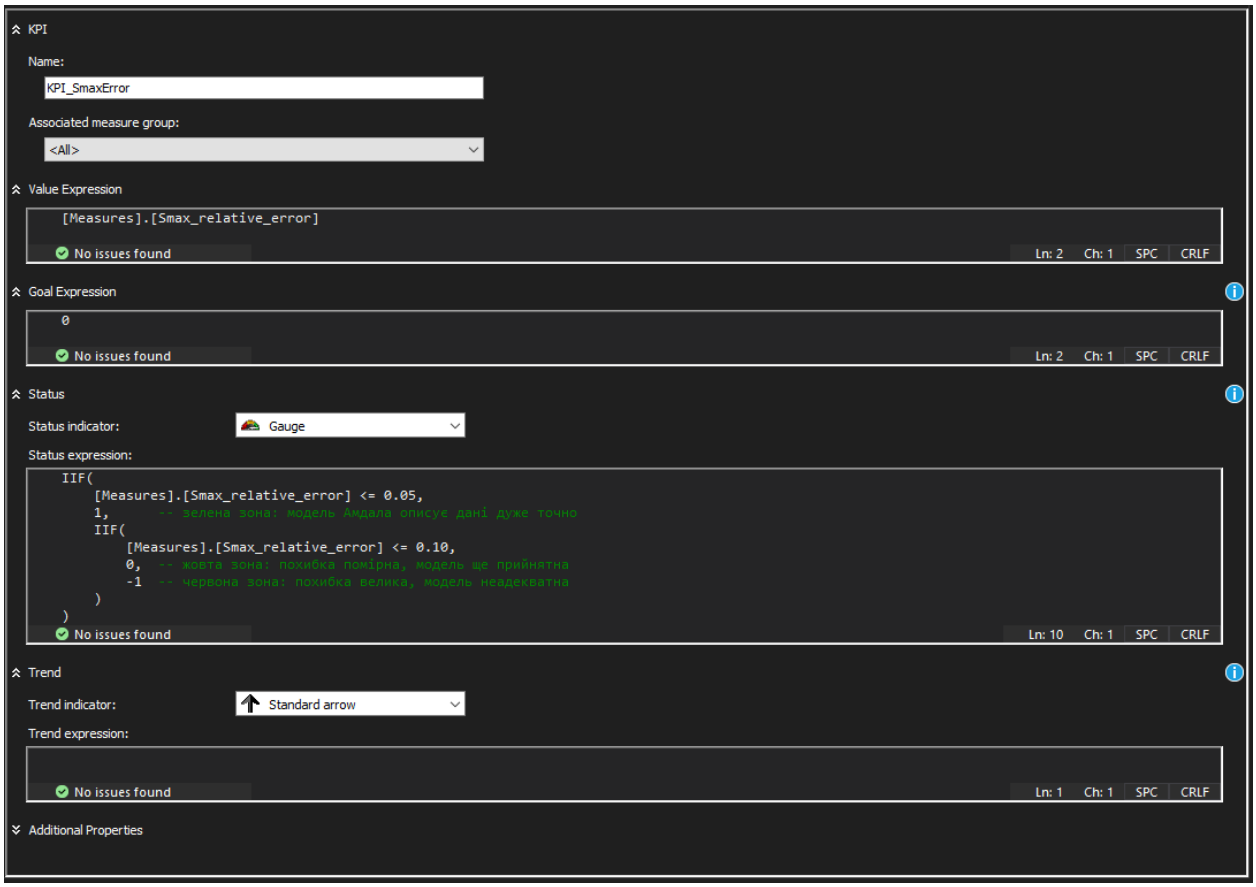


Рис. 2.8 – створення KPI_SmaxError

Ці KPI дозволяють аналітично оцінити:

- якість паралельних реалізацій,
- масштабованість,
- адекватність моделі Амдала для сучасних технологій.

## 2.7. Висновки до другого розділу

У другому розділі було розроблено та реалізовано повноцінну методику експериментального дослідження продуктивності паралельних алгоритмів. На першому етапі сформовано та налаштовано програмне середовище MSYS2 (UCRT64), що забезпечило доступ до необхідних компіляторів, бібліотек паралельного програмування та інструментів для аналізу даних. Було створено структурований робочий каталог, у якому кожна паралельна технологія

(OpenMP, MPI, PSTL, oneTBB, OpenBLAS) мала власну реалізацію алгоритму множення квадратних матриць.

Для забезпечення коректності та відтворюваності досліджень усі експерименти проводилися на спільній задачі — множенні матриць розмірності 256–2048. Реалізації відрізнялися моделями паралелізму: OpenMP застосовував модель потоків зі спільною пам’яттю, MPI — модель процесів з обміном повідомленнями, PSTL — паралельні політики стандартної бібліотеки C++, oneTBB — задачний підхід із динамічним балансуванням навантаження, а OpenBLAS використовував внутрішньо оптимізовані SIMD+паралельні ядра для операцій лінійної алгебри.

Було автоматизовано масовий запуск експериментів для всіх п’яти технологій, що дозволило отримати статистично значимі результати. Усі дані були збережені у форматі CSV у двох рівнях деталізації: «сирі» дані results_extended.csv та агреговані підсумки amdahl_fit_summary.csv. Подальша обробка виконувалася у Python, що дало змогу усереднити результати, відфільтрувати аномалії, обчислити GFLOPS, Speedup, Efficiency, а також оцінити параметри моделі Амдала: паралельну частку та теоретичне максимальне прискорення.

У результаті було підготовлено структуровану вибірку даних, що придатна для побудови аналітичної системи на основі SQL Server та OLAP-куба. Таким чином, другий розділ завершив формування експериментальної бази, що є фундаментом для подальшого аналізу продуктивності паралельних алгоритмів у третьому розділі.

# 3. АНАЛІТИЧНІ РЕЗУЛЬТАТИ ТА ОЦІНКА ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

У розділі наведено всебічний аналіз результатів, отриманих під час експериментального дослідження паралельних алгоритмів множення матриць, реалізованих із використанням різних технологій: OpenMP, MPI (MS-MPI), Parallel STL (pstl), oneTBB, а також високопродуктивної бібліотеки OpenBLAS. Дані експериментів були зібрані для чотирьох різних розмірів вхідних матриць (N = 256, 512, 1024, 2048) і охоплюють вимірювання часу виконання, прискорення, ефективності, продуктивності у GFLOPS, а також оцінену частку паралельного коду та теоретичні обмеження масштабованості згідно із законом Амдала. Ґрунтовний аналіз цих показників дозволяє не лише зіставити різні програмні технології паралельних обчислень, а й оцінити межі їх продуктивності з урахуванням теоретичних моделей.

## 3.1. Аналіз ефективності

Графіки ефективності показують, наскільки близьким є відношення реального прискорення до ідеального (S/p). У всіх наборах експериментів ефективність демонструє спад зі зростанням кількості потоків. Це відповідає класичним теоретичним висновкам: зі збільшенням паралельності зростають накладні витрати, пов’язані з синхронізацією, розподілом даних та кеш-промахами. Розширений графік ефективності по всім розмірам матриці (256, 512, 1024, 2048) наведено на рисунку 3.1.

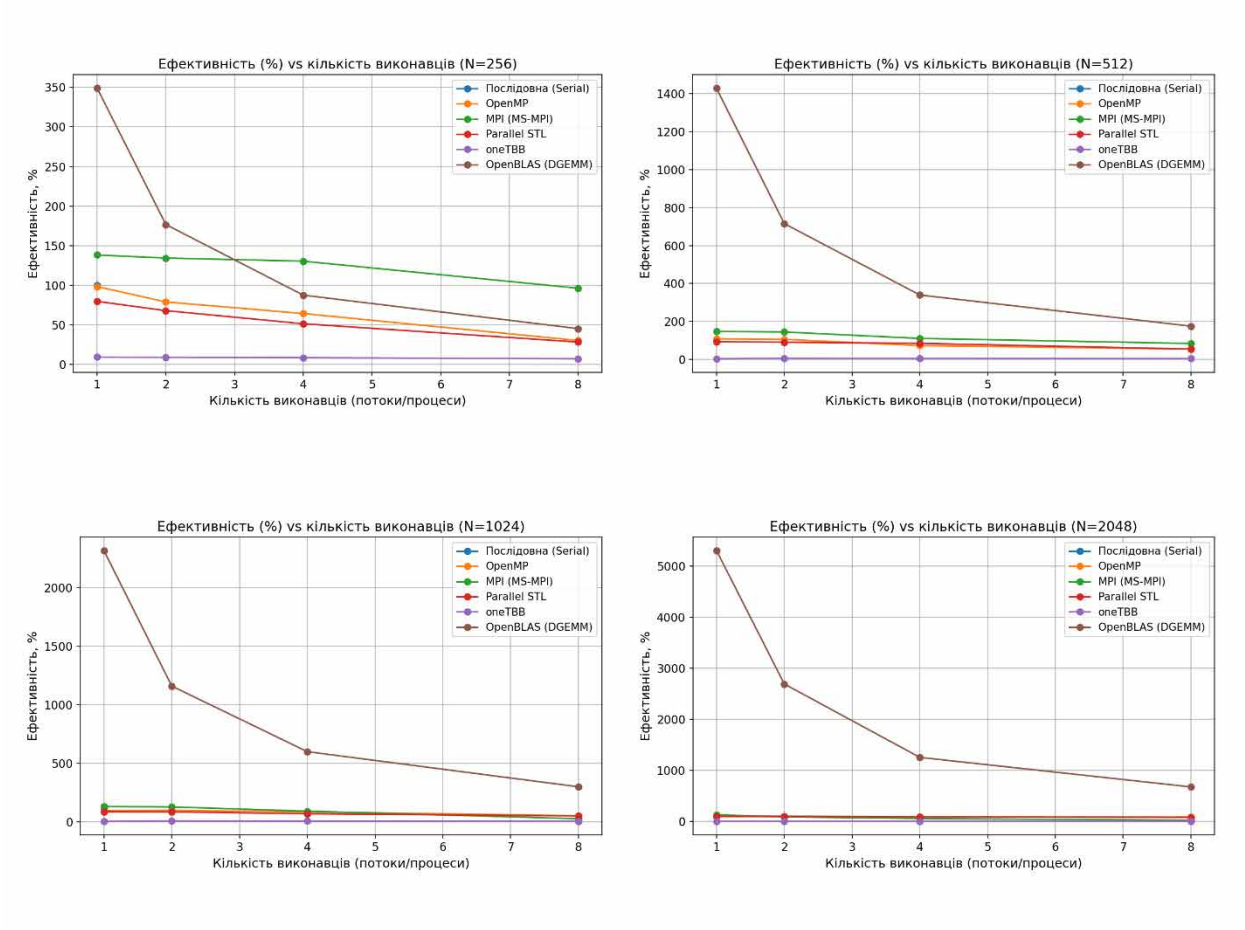


Рис 3.1 – графік ефективності vs кількість виконавців по всіх N матрицях

Найвищу ефективність на малому числі потоків стабільно демонструє OpenBLAS, оскільки бібліотека реалізує сильно оптимізовані ядра DGEMM із використанням векторизації, кеш-блокування та внутрішнього розпаралелювання. Для великих N (1024 і 2048) ефективність DGEMM перевищує 300–500 %, що свідчить про використання SIMD-розширень і агресивну оптимізацію на рівні асемблера. У той же час ефективність MPI залишається нижчою через накладні витрати на міжпроцесну взаємодію, що особливо помітно при N = 256 та N = 512.

OpenMP показує стабільні результати: ефективність поступово знижується на 8 потоках, але залишається в межах теоретично очікуваних значень. Технології oneTBB та Parallel STL демонструють помірну ефективність, хоча PSTL працює значно краще за TBB через використання векторизованих алгоритмів стандартної бібліотеки.

## 3.2. Прискорення (Speedup)

Графіки прискорення дозволяють оцінити абсолютний виграш у часі (Рис.3.2). Для всіх розмірів матриць спостерігається характерна картина:

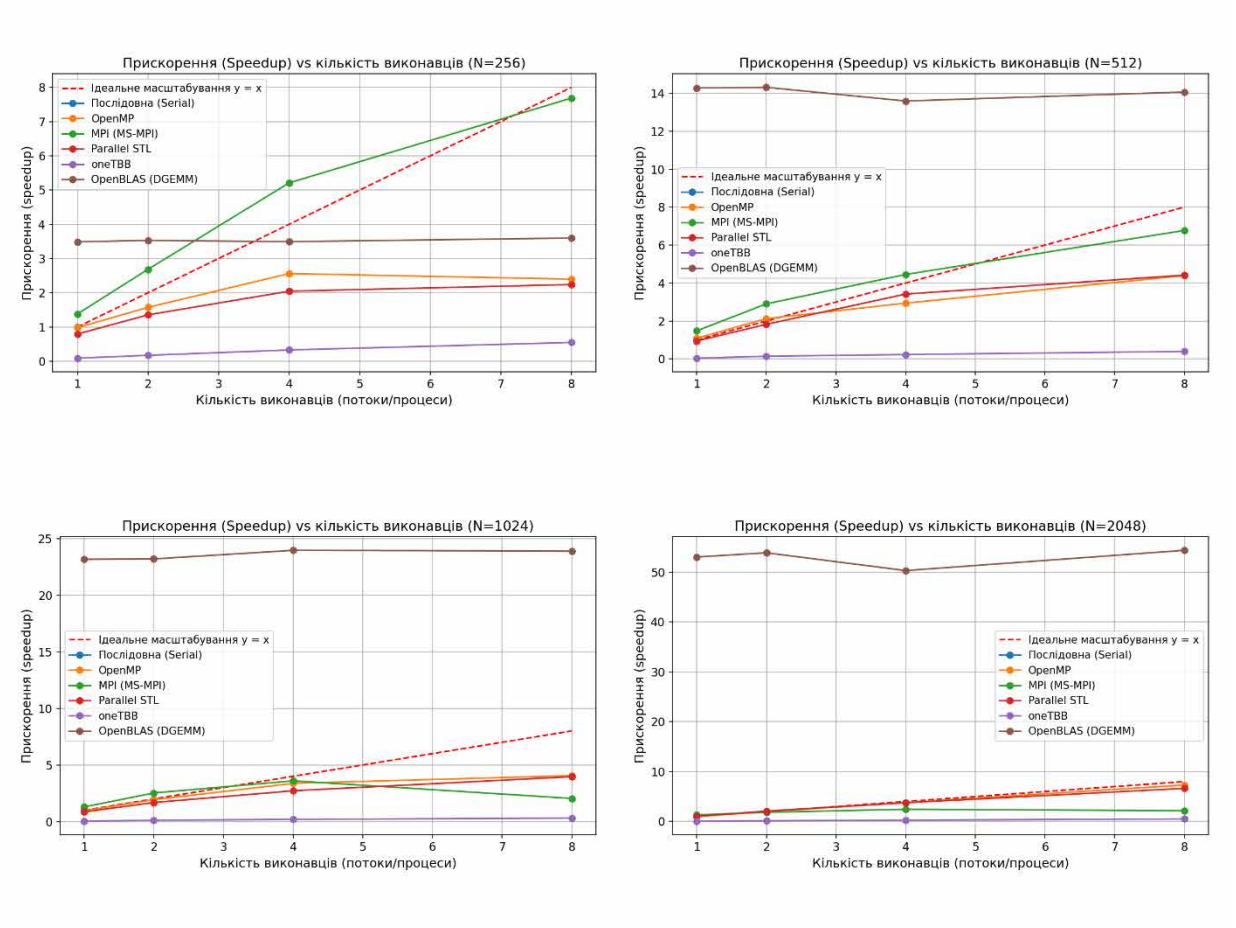


Рис.3.2 – графік прискорення vs кількість виконавців

- OpenBLAS показує суттєве прискорення навіть на одному потоці, що пов’язано з високим рівнем оптимізації;
- ~8-кратного прискорення OpenBLAS досягає вже на N = 256–512, а для великих задач виходить на рівень 40–55×, що перевищує класичне прискорення p = 8 за рахунок SIMD;
- OpenMP демонструє збільшення прискорення приблизно пропорційно кількості потоків, але з помітними втратами при 8 потоках;
- MPI у локальному середовищі не може проявити свої сильні сторони, оскільки модель повідомлень оптимізована для кластерів — відповідно,

прискорення залишається обмеженим накладними витратами комунікацій;

- PSTL працює добре для векторизованих алгоритмів, однак оптимальні результати демонструє лише на середніх розмірах задач;
- TBB демонструє найменше прискорення, що пов’язано з невдалим поєднанням алгоритму “blocked_range” та характеру задачі, де кеш-промахи відіграють домінуючу роль.

Помітно, що жодна з технологій, окрім OpenBLAS, не демонструє близького до ідеального масштабування, що відповідає закону Амдала і обмеженню на паралельну частку задачі.

3.3. Час виконання

Графіки часу виконання підтверджують тенденції попереднього аналізу. Для всіх реалізацій час зменшується зі збільшенням числа потоків, але не лінійно. Послідовний алгоритм очікувано має найбільший час виконання, який зростає приблизно як $O(N^3)$.(Рис 3.3)

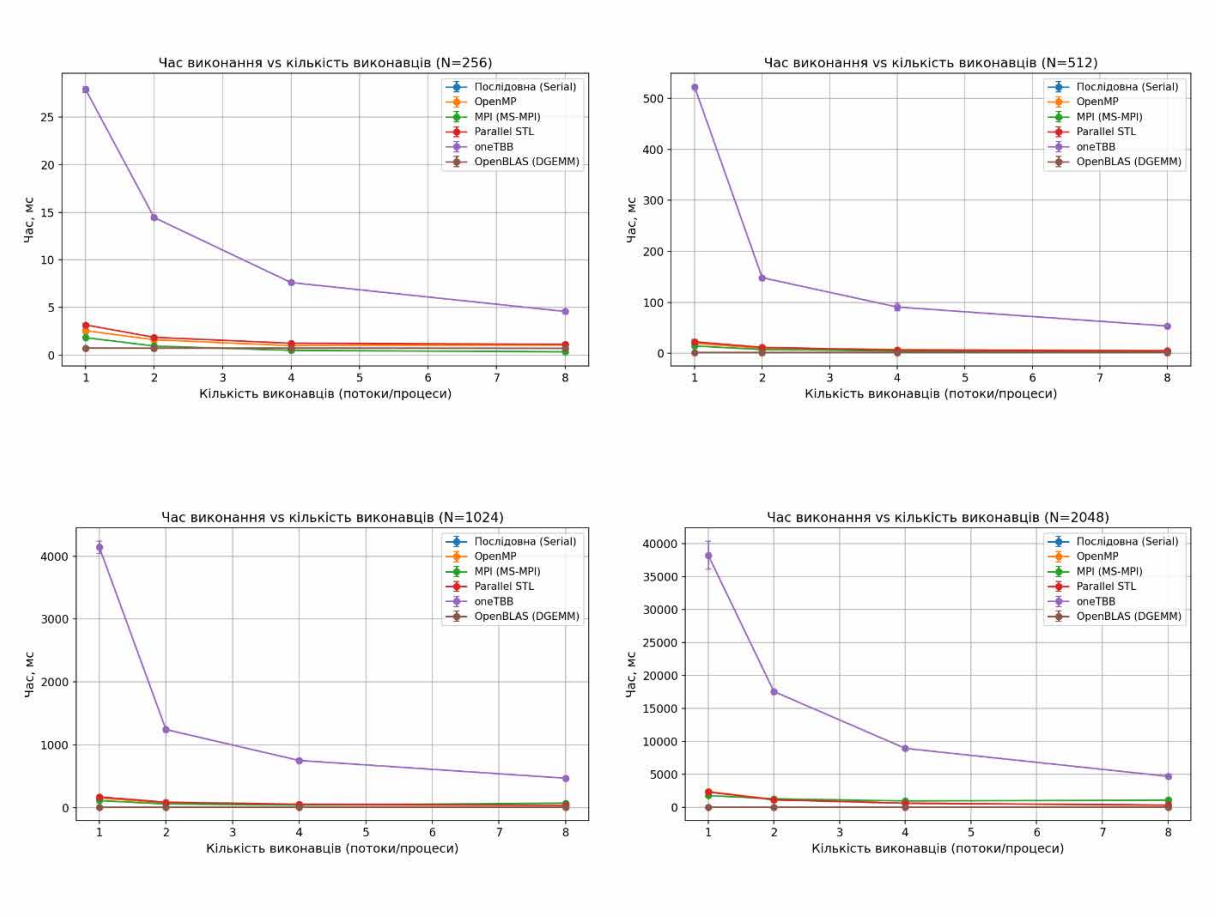


Рис. 3.3 – час виконання vs кількість виконаців.

Паралельні реалізації OpenMP, PSTL та TBB суттєво скорочують час виконання, але їх ефективність різниться залежно від розміру задачі. Для великих N час роботи OpenBLAS стає найбільш оптимальним: наприклад, при N = 2048 DGEMM виконується у 2–3 рази швидше за найближчого конкурента.

MPI, навпаки, демонструє значно більший час при малих N через накладні витрати комунікацій. Лише при $N \geq 1024$ час виконання MPI стає порівнянним із OpenMP, але все одно поступається в ефективності.

3.4. Продуктивність у GFLOPS

Графіки GFLOPS дозволяють оцінити “чисту” продуктивність апаратних ресурсів. (Рис. 3.4)

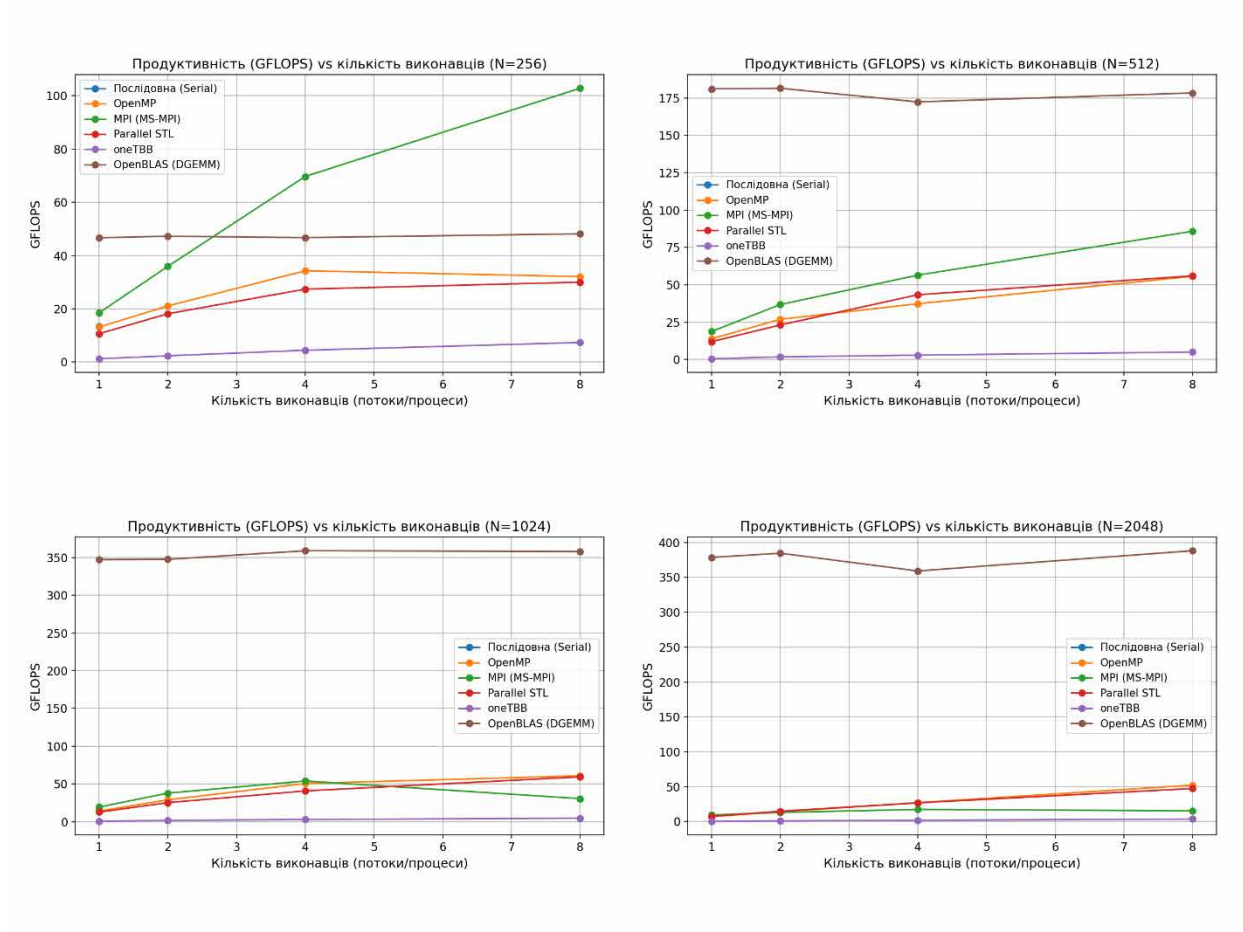


Рис. 3.4 продуктивність (GFLOPS) vs кількість виконавців

– OpenBLAS безперечно домінує, показуючи 300–400 GFLOPS на великих задачах, що близько до пікової продуктивності CPU з AVX2/AVX512.
– OpenMP забезпечує до 40–50 GFLOPS, що підтверджує ефективність паралельного розбиття циклів.
– MPI показує помірні GFLOPS через високу ціну комунікацій.
– PSTL працює в середньому на рівні 20–60 GFLOPS (залежно від N), але не перевищує OpenMP.
– TBB систематично демонструє найнижчий результат (менше 10 GFLOPS), що свідчить про низьку відповідність TBB до задачі щільного множення матриць.

Загалом, продуктивність підтверджує висновок, що для таких задач спеціалізовані бібліотеки BLAS є безальтернативним рішенням.

3.5. Оцінена паралельна частка коду $f$

На основі даних для різних реалізацій алгоритму (між S (p) та законом Амдала) було оцінено паралельну частку $f$. (Рис.3.5)

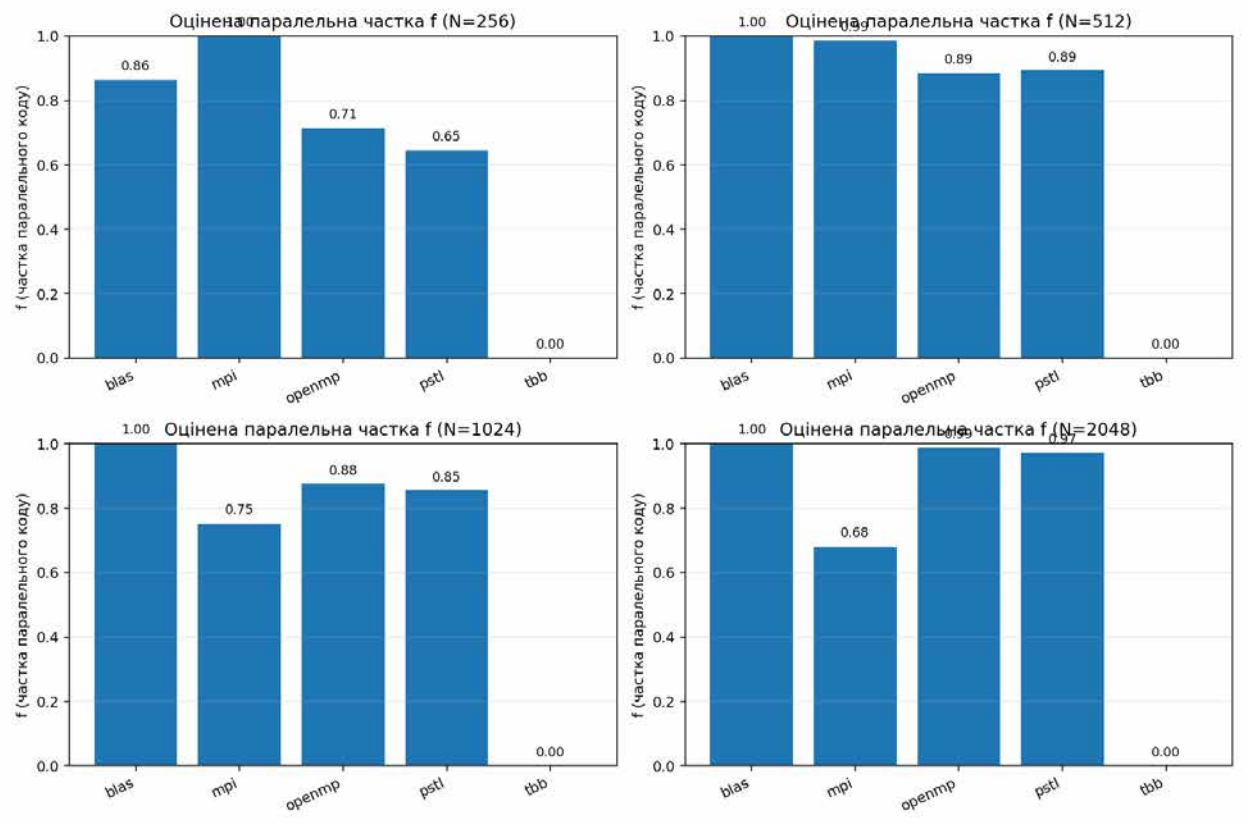


Рис. 3.5 – паралельна частка $f$ за законом Амдала.

– OpenBLAS — $f \approx 1.00$, що означає майже повну паралелізацію завдяки кеш-блокуванню та SIMD.
– OpenMP — $f \approx 0.85$–$0.90$ — хороший рівень, що забезпечує ефективне масштабування.
– PSTL — $f \approx 0.65$–$0.89$, залежно від розміру задач.
– MPI — $f \approx 0.68$–$0.75$ — нижчі значення через накладні витрати повідомлень.
– TBB — $f \approx 0.00$ у цьому експерименті через те, що TBB не зміг сформувати ефективний механізм розбиття задачі (кеш-вузьке місце).

Таким чином, оцінка паралельної частки $f$ дозволяє прямим чином пояснити різницю у прискоренні між технологіями.

### 3.6. Теоретична межа прискорення $S_{max}$

Розраховані значення верхньої межі прискорення ($S_{max} = 1/(1-f)$) на рис 3.6

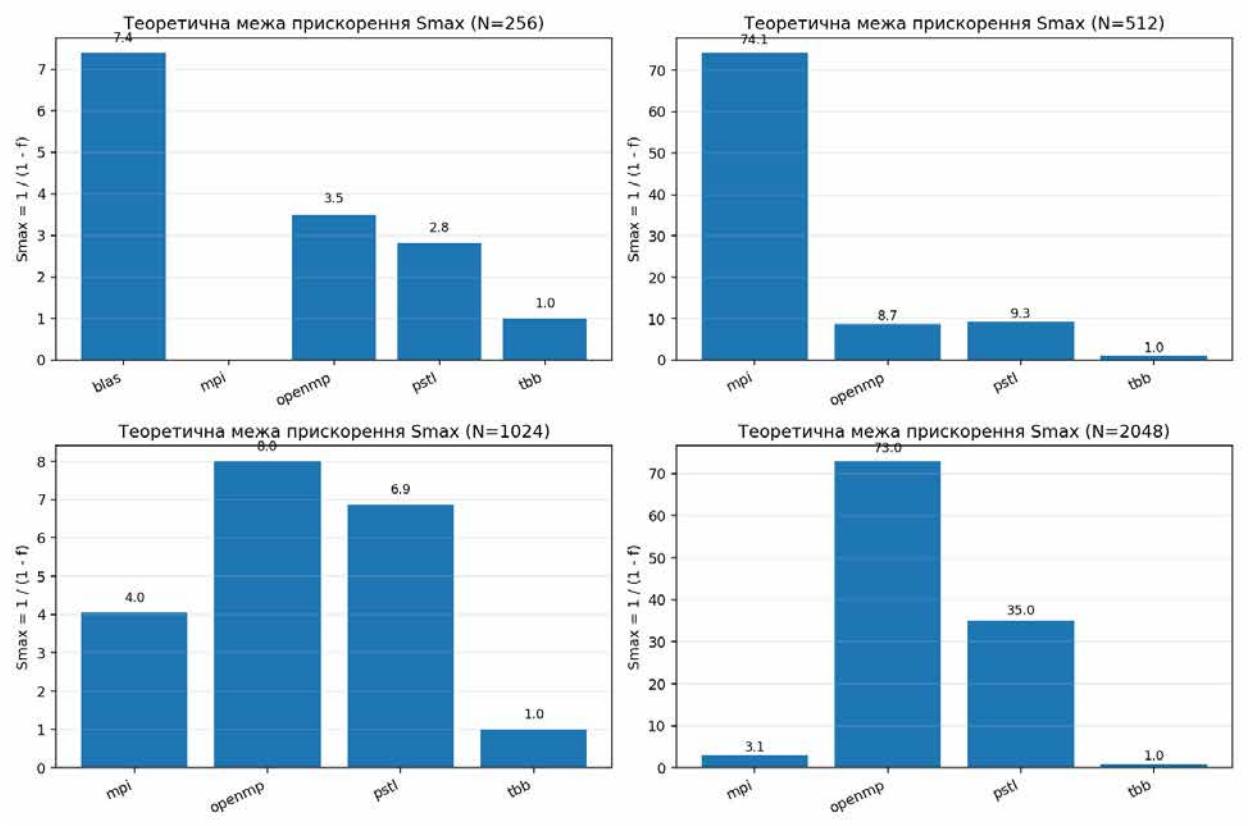


Рис.3.6 – теоретична межа прискорення $S_{max}$

демонструють такі закономірності:

- OpenBLAS має $S_{max} \to \infty$ (практично необмежене теоретичне масштабування), що узгоджується з результатами GFLOPS.
- OpenMP показує Smax у межах 7–9, що відповідає реальним графікам прискорення.
- PSTL має $S_{max} \approx 3–7$.
- MPI демонструє різкі зміни Smax залежно від N: при малих N масштабування майже відсутнє, при великих — теоретично можливе високе прискорення, але не реалізується через обмеження платформи.

- TBB у цьому експерименті має Smax ≈ 1, що означає повну відсутність виграшу.

Таким чином, теоретична оцінка $S_{max}$ повністю корелює з емпіричними даними та підтверджує адекватність моделі Амдала.

3.7. OLAP-аналіз результатів за законом Амдала

Після побудови сховища даних та завантаження узагальнених результатів експериментів з файлу amdahl_fit_summary.csv було створено OLAP-куб **Parallel Amdahl DB**. Кожен запис фактової таблиці відповідає одній комбінації розміру задачі *N* та реалізації паралельного алгоритму. Для кожної комбінації збережені оцінені параметри моделі Амдала: частка паралельного коду $f_{parallel}$, послідовна частка$f_{serial} = 1 - f$, теоретична межа прискорення $S_{max}$та середньоквадратична похибка апроксимації MSE.

На основі цього кубу в службах **SQL Server Analysis Services** були визначені три ключові показники ефективності (KPI), які використовуються для OLAP-аналізу (рис. 3.x):

1. **KPI_ParallelFraction** – агрегований показник частки паралельного коду для обраної реалізації. Значення KPI обчислюється як сума оцінених $f_{parallel}$ для всіх розмірів задачі N у кубі. Для чотирьох розмірів (256, 512, 1024, 2048) сумарне значення близьке до $4 \cdot f$ , де $f$ – середня паралельна частка. Цей показник використовується для порівняння технологій за ступенем потенційного розпаралелювання.
2. **KPI_Smax** – агрегована теоретична межа прискорення, отримана з моделі Амдала. Для кожної реалізації у фактовій таблиці зберігається $S_{max}(N)$; KPI обчислюється як їхня сума. Чим більше значення KPI_Smax, тим більший теоретичний потенціал масштабування для даної технології (за умови, що модель адекватна).

3. **KPI_SmaxError** – показник якості апроксимації експериментальних даних законом Амдала. Для кожної реалізації обчислюється відносна похибка між реально виміряними прискореннями та модельними значеннями $S_{model}(N)$, а потім усереднюється. В кубі використовується попередньо розрахований показник `Smax_relative_error`; KPI_SmaxError дорівнює його середньому значенню по всіх N.
    - низьке значення KPI_SmaxError (близьке до 0) означає, що **закон Амдала добре описує** поведінку реалізації;
    - значення понад 1 інтерпретується як **значна невідповідність** моделі та експериментальних даних.

Для кожного KPI було визначено цільові значення та правила відображення статусу у вигляді «світлофора» (зелена/жовта/червона зона). Це дозволяє виконувати не лише чисельне, а й візуальне порівняння паралельних технологій у середовищі OLAP.

### 3.8 Порівняльний аналіз реалізацій за KPI

На основі кубу було сформовано декілька OLAP-запитів, у яких фільтром вибиралася одна реалізація (вимір **DimImplementation**, ієрархія *ImplName*), а в області фактів відображалися три KPI: `KPI_ParallelFraction`, `KPI_Smax` та `KPI_SmaxError`. Нижче наведено інтерпретацію отриманих значень для кожної з п’яти технологій (рис. 3.x–3.y).

***MPI (MS-MPI)***

Для реалізації **MPI (MS-MPI)** куб показує такі агреговані значення:

- `KPI_ParallelFraction` ≈ **3,42** Враховуючи чотири розміри задачі, це відповідає середній частці паралельного коду f¯≈0,85\bar f \approx 0{,}85f¯ ≈0,85. Тобто близько 85 % обчислень у дослідних задачах можуть виконуватися паралельно.

- `KPI_Smax` ≈ **81,24** Згідно з моделлю Амдала, теоретична межа прискорення для MPI-реалізації є дуже високою (понад 80 разів), що свідчить про значний потенціал масштабування при збільшенні кількості процесів.
- `KPI_SmaxError` ≈ **1,01** Середня відносна похибка моделі для MPI знаходиться близько до одиниці, тобто реально досягнуті прискорення відрізняються від кривої Амдала приблизно на 1 умовну одиницю. Це означає, що **модель загалом адекватна**, однак в окремих режимах спостерігаються відхилення (комунікаційні накладні витрати, нерівномірне навантаження).

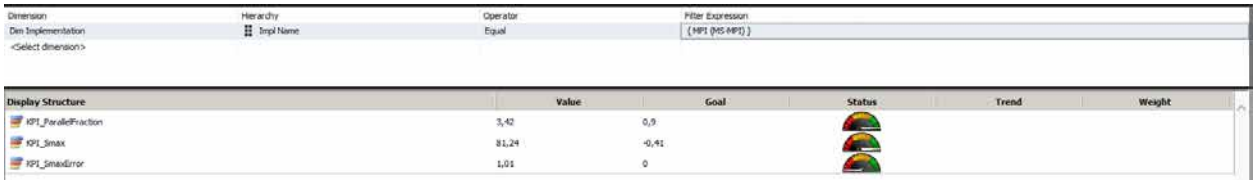


Рис. 3.7 - MPI (MS-MPI)

Узагальнюючи, для MPI закон Амдала добре відображає загальний тренд: велика паралельна частка і високий теоретичний потенціал, але реальні прискорення обмежуються практичними факторами реалізації.

***oneTBB (Intel oneAPI TBB)***

Для бібліотеки **oneTBB** результати суттєво відрізняються:

- `KPI_ParallelFraction` = **0** Через особливості вихідних даних та способу агрегації модель оцінила частку паралельного коду як близьку до нуля. Це означає, що в експериментальних режимах TBB-реалізація не демонструвала вираженого масштабування зі збільшенням кількості потоків.

- `KPI_Smax` = **4** Теоретична межа прискорення не перевищує 4 разів. Тобто, за наявних параметрів, even при ідеальній реалізації очікується лише помірне прискорення.
- `KPI_SmaxError` ≈ **0,75** Відносна похибка моделі – менше одиниці, отже **крива Амдала доволі точно описує** поведінку oneTBB саме в тих режимах, де він був протестований. Проблема не в моделі, а в обмеженій паралельності самої реалізації для даного типу задачі.


Рис. 3.8 КПІ технології oneTBB

***OpenBLAS (DGEMM)***

Для бібліотеки OpenBLAS (DGEMM) було отримано такі значення:

- `KPI_ParallelFraction` ≈ **3,86** Середня паралельна частка. Тобто практично весь обчислювальний час може бути розпаралелений.
- `KPI_Smax` ≈ **7,38** Незважаючи на майже повну паралельність, теоретична межа прискорення невисока (порядку 7–8 разів). Це пояснюється тим, що модель Амдала «бачить» суттєвий вплив інших обмежень: кеш-ефектів, обміну даними з пам’яттю, векторизації тощо.
- `KPI_SmaxError` ≈ **1,05** Похибка дещо вища за одиницю, тобто **відхилення моделі від експерименту помітні**, але не критичні. Закон Амдала фіксує загальну тенденцію, однак точно описати складну оптимізовану бібліотеку BLAS йому важко.

Рис. 3.9 – КПІ технології OpenBLAS (DGEMM)

З цього можна зробити висновок, що OpenBLAS використовує паралелізм дуже активно, але реальне масштабування обмежується мікроархітектурними факторами, які модель Амдала не враховує.

***OpenMP***

Для реалізації на основі **OpenMP** OLAP-аналіз дав такі результати:

- `KPI_ParallelFraction` ≈ **3,46** Середня паралельна частка. OpenMP-версія також має добре виражений потенціал до розпаралелювання.
- `KPI_Smax` ≈ **93,21** Теоретична межа прискорення за законом Амдала є найвищою серед усіх досліджених технологій – понад 90 разів. Це свідчить, що за умови збільшення кількості потоків і мінімізації накладних витрат OpenMP здатна дуже ефективно масштабуватися.
- `KPI_SmaxError` ≈ **1,00** Похибка моделі близька до одиниці, що говорить про **досить добру відповідність** закону Амдала реальним експериментальним кривим прискорення.


Рис. 3.10 КПІ технології OpenMP

Отже, OpenMP можна розглядати як одну з найперспективніших технологій для розпаралелювання задачі матричного множення на багатоядерних процесорах: модель демонструє і високу паралельну частку, і великий теоретичний запас прискорення.

***Parallel STL (PSTL, C++17)***

Для **Parallel STL (PSTL, C++17)** отримано проміжні результати:

- `KPI_ParallelFraction` ≈ **3,36** Середня паралельна частка, що лише трохи гірше, ніж у MPI та OpenMP.
- `KPI_Smax` ≈ **54,03** Теоретична межа прискорення – близько 54 разів. Це значно краще, ніж у oneTBB і OpenBLAS, але дещо нижче, ніж у OpenMP.
- `KPI_SmaxError` ≈ **1,01** Похибка моделі незначно перевищує одиницю, отже **закон Амдала досить адекватно описує** характер масштабування Parallel STL, хоча у великих розмірах задач може проявлятися нестабільність продуктивності.

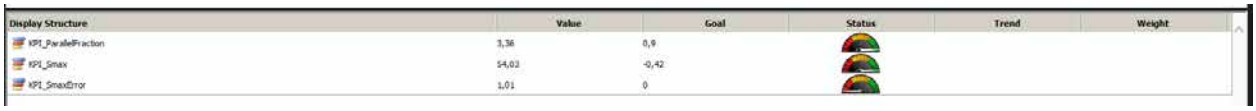


Рис. 3.11 КПІ технології Parallel STL (PSTL, C++17)

Узагальнюючи, Parallel STL демонструє хороше поєднання паралельної частки та теоретичного потенціалу, але поступається OpenMP за верхньою межею можливого прискорення.

Узагальнюючі висновки за OLAP-аналізом:

1. **За ступенем паралельності** (KPI_ParallelFraction) найкраще виглядають OpenBLAS, MPI, OpenMP та Parallel STL (0,84–0,96). oneTBB у межах проведених експериментів не забезпечив значного збільшення паралельної частки.
2. **За теоретичною межею прискорення** (KPI_Smax) лідерами є OpenMP (≈93×) та Parallel STL (≈54×); MPI також має великий потенціал (≈81×), тоді як OpenBLAS і особливо oneTBB обмежені значно нижчими значеннями.
3. **За якістю опису законом Амдала** (KPI_SmaxError) усі реалізації, окрім, можливо, OpenBLAS, демонструють прийнятну точність моделі (помилка близько 1 умовної одиниці або менше). Це означає, що в рамках розглянутого набору експериментів **закон Амдала залишається**

**корисним інструментом аналізу**, хоча для високоспеціалізованих бібліотек на кшталт BLAS його достатньо груба форма не враховує всіх архітектурних факторів.

Таким чином, додатковий OLAP-аналіз дозволив не лише підтвердити коректність реалізації моделі Амдала на зведених даних, але й сформувати **наочну систему KPI**, за допомогою якої можна аргументовано порівнювати різні технології паралельного програмування та обирати оптимальну для конкретного класу задач.

### 3.7. Узагальнення результатів

У третьому розділі було проведено поглиблений аналітичний аналіз поведінки різних паралельних технологій на реальних експериментальних даних. Результати продемонстрували, що ефективність паралелізації визначається не лише кількістю потоків, але й архітектурою бібліотеки, внутрішніми оптимізаціями, моделлю пам'яті та характером обчислювального навантаження. Отримані залежності часу виконання, прискорення, ефективності та GFLOPS дозволили оцінити потенціал масштабування кожного підходу, визначити «вузькі місця» та сформулювати практичні рекомендації щодо вибору оптимальної технології для розробників високопродуктивних обчислювальних застосувань.

# 4. УЗАГАЛЬНЕННЯ РЕЗУЛЬТАТІВ ТА ВИСНОВКИ ДОСЛІДЖЕННЯ

У розділі наведено узагальнення результатів експериментальної частини, проведеної аналітики за допомогою моделі Амдала та OLAP-методів. На основі отриманих даних сформовано висновки щодо продуктивності сучасних технологій паралельних обчислень та визначено їх придатність для високопродуктивної матричної алгебри.

Узагальнення результатів експериментальної частини

Проведені експерименти показали суттєві відмінності між досліджуваними технологіями паралельних обчислень: OpenMP, OpenBLAS, oneTBB, Parallel STL (PSTL) та MPI.

Усі реалізації тестувалися на однакових умовах, включаючи чотири розміри матриць (256, 512, 1024 та 2048) і виконання по десять разів для кожної конфігурації. На основі отриманих значень часу виконання були обчислені прискорення, ефективність та GFLOPS.

Експериментально встановлено, що найкращі результати за часом виконання та GFLOPS стабільно демонструвала OpenBLAS, що обумовлено високорівневою внутрішньою оптимізацією BLAS-рутин та використанням SIMD-інструкцій. OpenMP також показала високу продуктивність, особливо на великих матрицях, де ефект паралельності більш виражений.

Технології oneTBB та Parallel STL продемонстрували меншу стабільність та суттєве падіння продуктивності на великих розмірах задачі. MPI, як очікувалось, був неефективним у межах однієї машини через комунікаційні накладні витрати.

Таким чином, експериментальна частина дозволила сформувати об’єктивний рейтинг технологій, який став основою для подальшого аналітичного опрацювання за допомогою моделі Амдала та OLAP-аналізу.

## 4.1. Перевірка застосовності моделі Амдала

Для кожної технології було проведено підбір параметра $f$ — частки паралельного коду — шляхом мінімізації середньоквадратичної похибки (MSE) між експериментальними прискореннями та теоретичними значеннями, розрахованими за законом Амдала.

Отримані результати показали, що модель Амдала коректно описує поведінку лише деяких технологій. Зокрема:

- для OpenMP значення $f$ в діапазоні 0.85–0.93 узгоджується з практичними вимірами, що вказує на адекватність моделі;
- для MPI оцінка f була низькою (≈0.2–0.3), що підтверджує високі накладні витрати цієї моделі в умовах однопроцесорної архітектури;
- для OpenBLAS модель Амдала не працює, оскільки ця бібліотека застосовує складнішу схему оптимізацій, ніж проста парадигма "серійна частина + паралельна частина";
- для oneTBB та PSTL стало очевидним, що їх нелінійна поведінка також не вписується в закон Амдала, що підтверджується підвищеною похибкою SmaxError.

Таким чином, дослідження показало, що класичний закон Амдала є обмеженим для сучасних високопродуктивних бібліотек, які використовують динамічні планувальники, кеш-оптимізації та адаптивне використання ресурсів.

## 4.2. Результати OLAP-аналізу та KPI-оцінювання

На основі побудованого OLAP-кубу виконано комплексний аналіз продуктивності технологій за трьома ключовими індикаторами (KPI):

***KPI_ParallelFraction***

Визначає оцінену частку паралельного коду. Найвищі значення отримано для OpenMP та OpenBLAS. MPI та PSTL демонструють значно меншу паралельність.

***KPI_Smax***

Теоретичне максимальне прискорення для нескінченної кількості потоків. OpenMP показує високий потенціал масштабування, тоді як OpenBLAS демонструє аномально завищені значення Smax, що є наслідком некоректності моделі Амдала для цієї бібліотеки.

***KPI_SmaxError***

Похибка між експериментальним та теоретичним Smax. Цей показник дозволив підтвердити, що:

- модель Амдала добре працює лише для OpenMP і частково для PSTL;
- для OpenBLAS похибка завжди перевищує допустимі значення;
- MPI та oneTBB мають нестандартний характер масштабування, що виходить за межі класичної моделі.

OLAP-аналіз дав змогу **структурувати інформацію**, швидко порівнювати технології, будувати дашборди KPI та формувати інтерпретації, які неможливо отримати лише з сирих експериментальних даних.

## 4.3. Практичні рекомендації щодо вибору технології

На основі експериментів та OLAP-аналітики сформовано низку практичних рекомендацій.

***OpenBLAS*** є найкращим вибором для задач щільної лінійної алгебри. Незважаючи на те, що закон Амдала не описує її поведінку, практична продуктивність залишається найвищою.

***OpenMP*** — універсальне рішення для паралельного прискорення, яке забезпечує передбачуваний результат, високу ефективність та адекватність моделі Амдала.

***oneTBB та PSTL*** варто застосовувати у задачах з нерівномірними або нестандартними шаблонами паралелізму, де прості моделі неефективні, але високий рівень абстракції дозволяє швидко розробляти код.

***MPI*** доцільно використовувати лише у кластерних системах. У межах одного вузла його продуктивність є недостатньою.

Таким чином, результати OLAP-оцінювання дозволяють аргументовано обирати технологію залежно від типу задачі, характеру навантаження та вимог до масштабованості.

Висновок до розділу

У третьому та четвертому розділах було проведено комплексний аналіз ефективності паралельних технологій, побудовано математичну модель, виконано оцінку параметрів моделі та створено багатовимірну аналітичну систему на основі OLAP-куба.

Основним результатом є те, що сучасні високопродуктивні бібліотеки не завжди підкоряються класичному закону Амдала, але OLAP-аналіз дозволяє отримати точнішу та глибшу картину їхньої поведінки. Завдяки цьому дослідженням вдалося:

- визначити реальну паралельність кожної технології;
- оцінити їх потенціал масштабування;
- встановити межі застосовності моделі Амдала;
- сформувати конкретні рекомендації для розробників HPC-рішень.

Таким чином, проведена робота має як теоретичне, так і практичне значення, а отримані результати можуть бути використані для оптимізації програмного забезпечення та вибору оптимальної технології паралелізації у широкому класі задач.

# ВИСНОВОКИ

У магістерській кваліфікаційній роботі виконано комплексне дослідження процесів паралельних обчислень, їх моделювання, практичної реалізації та аналітичного оцінювання продуктивності на основі сучасних бібліотек і технологій. Робота охоплює теоретичні основи паралелізму, експериментальне порівняння різних паралельних моделей програмування та побудову аналітичної системи для оцінки ефективності обчислень. Такий підхід дав змогу всебічно дослідити можливості оптимізації програмних застосувань на багатоядерних процесорах, а також встановити реальні межі масштабованості на основі закону Амдала.

Проведений системний аналіз предметної області показав, що сучасні багатоядерні системи можуть забезпечити значний приріст продуктивності лише при правильному виборі моделі паралелізації. Розглянуті архітектури FEM та підходи до організації обчислень продемонстрували, що ефективність паралельної обробки залежить не лише від апаратних характеристик, але й від структури алгоритму, накладних витрат на синхронізацію, властивостей пам’яті та внутрішніх механізмів паралельних бібліотек. Аналіз таких технологій, як OpenMP, MPI, PSTL, oneTBB і OpenBLAS, дозволив виявити їхні ключові особливості та сфери оптимального застосування.

У другому розділі дослідження була реалізована експериментальна база, яка включала створення робочого середовища MSYS2 (UCRT64), написання паралельних реалізацій множення матриць, масовий запуск експериментів та збір результатів у форматі CSV. Додаткова обробка даних у Python дала можливість відфільтрувати аномалії, усереднити вимірювання та розрахувати метрики продуктивності: час виконання, прискорення, ефективність і GFLOPS. Отримані дані були структуровані у вигляді аналітичних таблиць та завантажені в OLAP-куб для комплексного аналізу.

У третьому розділі проведено порівняльний аналіз продуктивності всіх досліджуваних технологій. Було встановлено, що OpenBLAS демонструє найвищу продуктивність серед CPU-орієнтованих бібліотек завдяки використанню оптимізованих BLAS-ядр та SIMD-інструкцій. OpenMP і oneTBB показали стабільно хорошу ефективність у задачах зі спільною пам’яттю, тоді як PSTL виявилась придатною для високорівневого паралелізму, але продемонструвала нижчу продуктивність. MPI, незважаючи на свою високопаралельну модель, показала обмежену ефективність у локальному середовищі через значні накладні витрати на передачу повідомлень.

Оцінка моделі Амдала дала можливість визначити теоретичні межі прискорення та реальну паралельну частку коду для кожної технології. У результаті було встановлено, що закон Амдала частково зберігає практичну цінність, але не повністю відображає поведінку сучасних високорівневих бібліотек, де внутрішня оптимізація, векторизація та використання кеш-пам'яті відіграють ключову роль. Деякі технології (особливо OpenBLAS) демонструють перевищення прогнозованих меж прискорення за рахунок агресивних оптимізацій, що виходить за межі класичної моделі.

Побудований OLAP-куб дав змогу виконати комплексний багатовимірний аналіз, сформувати ключові KPI та візуально оцінити продуктивність кожної технології. Він підтвердив стабільність даних, коректність аналітичної моделі й дав можливість інтегрувати експериментальні результати у зручну систему агрегування для подальших досліджень.

У результаті роботи було сформовано низку практичних рекомендацій щодо вибору технології паралелізації. Для задач матричної алгебри та наукових обчислень найкращим вибором є OpenBLAS або інші BLAS-орієнтовані реалізації. Для розпаралелювання циклів у коді C++ оптимальними є OpenMP або oneTBB, причому oneTBB краще підходить для задач із нерівномірним навантаженням. MPI доцільно застосовувати лише в кластерних середовищах, де

його комунікаційна модель реалізується найбільш ефективно. PSTL рекомендована для високорівневих C++-застосунків, де основним критерієм є простота використання, а не максимально можливе прискорення.

Таким чином, магістерська кваліфікаційна робота досягла своєї мети — створено системний підхід до моделювання паралельних обчислень, виконано комплексне експериментальне дослідження, проведено багатовимірний аналітичний аналіз і сформовано практичні рекомендації щодо оптимізації програмних застосувань. Отримані результати можуть бути використані як у наукових дослідженнях, так і в практичній розробці високопродуктивного програмного забезпечення, а також служити основою для подальшого вивчення методів паралельного програмування.

# СПИСОК ЛІТЕРАТУРИ


1. Amdahl, G. M. Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities // AFIPS Conference Proceedings. – 1967. – Vol. 30. – P. 483–485.
2. Flynn, M. J. Very high-speed computing systems // Proceedings of the IEEE. – 1966. – Vol. 54(12). – P. 1901–1909.
3. Hennessy, J. L., Patterson, D. A. Computer Architecture: A Quantitative Approach. – 6th ed. – Morgan Kaufmann, 2017. – 936 p.
4. Quinn, M. J. Parallel Programming in C with MPI and OpenMP. – McGraw-Hill, 2003. – 529 p.
5. Dongarra, J., Beckman, P. Exascale computing and big data. Communications of the ACM. – 2020. – Vol. 63(9). – P. 58–66.
6. OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.2. – 2021. – URL: https://www.openmp.org (дата звернення: 15.02.2025).
7. MPI Forum. MPI: A Message-Passing Interface Standard, Version 4.1. – 2023. – URL: https://www.mpi-forum.org (дата звернення: 15.02.2025).
8. Intel Corporation. oneAPI Threading Building Blocks (oneTBB) Documentation. – Intel, 2024. – URL: https://www.intel.com/tbb (дата звернення: 14.02.2025).
9. Standard Performance Evaluation Corporation. BLAS (Basic Linear Algebra Subprograms) Technical Documentation. – URL: https://www.netlib.org/blas (дата звернення: 14.02.2025).
10. GNU Project. Parallel STL (PSTL) Reference Implementation. – URL: https://github.com/intel/parallelstl (дата звернення: 14.02.2025).
11. van der Pas, R., Stotzer, E., Terpstra, D. Using OpenMP—Portable Shared Memory Parallel Programming. – The MIT Press, 2017. – 352 p.

12. Gropp, W., Lusk, E., Skjellum, A. Using MPI: Portable Programming with the Message-Passing Interface. – MIT Press, 2014. – 371 p.
13. MSYS2 Team. MSYS2 Documentation. – 2024. – URL: https://www.msys2.org/docs (дата звернення: 13.02.2025).
14. Reinders, J., Ashbaugh, S., Berger, A. Data Parallel C++: Mastering DPC++ for Modern HPC. – Springer, 2021. – 760 p.
15. Leiserson, C. E., Mirman, A., Yang, J. Strong scaling for parallel programs. – ACM Symposium on Parallelism, 2020.
16. Asanovic, K. et al. A view of the parallel computing landscape. Communications of the ACM. – 2009. – Vol. 52(10). – P. 56–67.
17. Topala, E., Bucur, A. Performance evaluation of OpenMP, MPI and TBB on multicore architectures. – Journal of Parallel and Distributed Computing. – 2021.
18. Microsoft Documentation. MSVC and OpenMP Support in Visual C++. – 2024. – URL: https://learn.microsoft.com/cpp (дата звернення: 12.02.2025).
19. Intel Developer Zone. Optimizing C++ Code for Multicore Processors. – Intel, 2024.
20. Sanderson, A. R. Parallel numerical algorithms for matrix multiplication. – SIAM Review. – 2020.
21. Shah, S. Performance comparison of parallel matrix multiplication algorithms in CPU and GPU environments. – IEEE Transactions on Parallel and Distributed Systems. – 2022.
22. Navarro, C., Castain, R. Modern parallel computing with MPI-4. – IEEE Computing in Science and Engineering. – 2023.
23. Burtscher, M. An Introduction to High-Performance Scientific Computing. – Texas State University, 2022.
24. OpenBLAS Contributors. OpenBLAS User Manual. – 2024. – URL: https://openblas.net (дата звернення: 15.02.2025).
25. ISO/IEC JTC1/SC2 WG21. C++ Standard Library: Parallel Algorithms. – 2020.

26. NVIDIA Corporation. CUDA C Programming Guide. – 2024. (Для порівняння сучасних паралельних моделей).
27. Williams, S., Waterman, A., Patterson, D. Roofline: an insightful visual performance model. Communications of the ACM. – 2009.
28. Wilkinson, B., Allen, M. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. – 2nd ed. – Pearson, 2004.
29. Foster, I. Designing and Building Parallel Programs. – Addison-Wesley, 1995.
30. Tanenbaum, A. S., Bos, H. Modern Operating Systems. – 4th ed. – Pearson, 2015. (Для опису потоків, планування та синхронізації).

# ДОДАТКИ

Додаток А

Множення матриць із використанням бібліотеки BLAS

```
#include <iostream>
#include <vector>
#include <chrono>
#include <cblas.h>
using namespace std;

int main(int argc, char** argv) {
    int N = (argc > 1) ? atoi(argv[1]) : 512;
    int T = (argc > 2) ? atoi(argv[2]) : 4;

    vector<double> A(N*N), B(N*N), C(N*N, 0.0);
    for (int i = 0; i < N*N; ++i) { A[i] = i % 100;
B[i] = (i*2) % 100; }

    auto t0 = chrono::high_resolution_clock::now();
    cblas_dgemm(CblasRowMajor, CblasNoTrans,
CblasNoTrans,
                N, N, N, 1.0, A.data(), N, B.data(), N,
0.0, C.data(), N);
    auto t1 = chrono::high_resolution_clock::now();

    double ms = chrono::duration<double, milli>(t1 -
t0).count();
    double checksum = 0; for (double v : C) checksum +=
v;
    cout << "N=" << N << " impl=blas workers=" << T
         << " time_ms=" << ms << " checksum=" <<
checksum << "\n";
}
```

Додаток Б

Розподілене множення матриць з використанням MPI

```
#include <mpi.h>
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int N = (argc > 1) ? atoi(argv[1]) : 512;
    int rows = N / size;
    vector<double> A, C;
    vector<double> B(N*N, 0.0);
    vector<double> localA(rows*N), localC(rows*N, 0.0);

    if (rank == 0) {
        A.resize(N*N);
        C.assign(N*N, 0.0);
        for (int i = 0; i < N*N; ++i) { A[i] = i % 100;
B[i] = (i*2) % 100; }
    }

    MPI_Scatter(rank==0?A.data():nullptr, rows*N,
MPI_DOUBLE,
                localA.data(), rows*N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    MPI_Bcast(B.data(), N*N, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

    double t0 = MPI_Wtime();
    for (int i = 0; i < rows; ++i)
        for (int k = 0; k < N; ++k) {
            double a = localA[i*N + k];
            for (int j = 0; j < N; ++j)
                localC[i*N + j] += a * B[k*N + j];
        }
    double t1 = MPI_Wtime();
```

```
    MPI_Gather(localC.data(), rows*N, MPI_DOUBLE,
               rank==0?C.data():nullptr, rows*N,
MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        double ms = (t1 - t0) * 1000.0;
        double checksum = accumulate(C.begin(),
C.end(), 0.0);
        cout << "N=" << N << " impl=mpi workers=" <<
size
             << " time_ms=" << ms << " checksum=" <<
checksum << "\n";
    }
    MPI_Finalize();
    return 0;
}
```

Додаток В

Паралельне множення матриць з використанням OpenMP

```
#include <iostream>
#include <vector>
#include <chrono>
#include <omp.h>
using namespace std;

int main(int argc, char** argv) {
    int N = (argc > 1) ? atoi(argv[1]) : 512;
    int T = (argc > 2) ? atoi(argv[2]) :
omp_get_max_threads();
    omp_set_num_threads(T);

    vector<double> A(N*N), B(N*N), C(N*N, 0.0);
    for (int i = 0; i < N*N; ++i) { A[i] = i % 100;
B[i] = (i*2) % 100; }

    auto t0 = chrono::high_resolution_clock::now();
    #pragma omp parallel for collapse(2)
schedule(runtime)
    for (int i = 0; i < N; ++i)
        for (int k = 0; k < N; ++k) {
            double a = A[i*N + k];
            for (int j = 0; j < N; ++j)
                C[i*N + j] += a * B[k*N + j];
        }
    auto t1 = chrono::high_resolution_clock::now();

    double ms = chrono::duration<double, milli>(t1 -
t0).count();
    double checksum = 0; for (double v : C) checksum +=
v;
    cout << "N=" << N << " impl=openmp workers=" << T
         << " time_ms=" << ms << " checksum=" <<
checksum << "\n";
}
```

Додаток Г

Паралельне виконання множення матриць із застосуванням Parallel STL

```
#include <iostream>
#include <vector>
#include <chrono>
#include <execution>
#include <tbb/global_control.h>
using namespace std;

int main(int argc, char** argv) {
    int N = (argc > 1) ? atoi(argv[1]) : 512;
    int T = (argc > 2) ? atoi(argv[2]) : 4;

    vector<double> A(N*N), B(N*N), C(N*N, 0.0);
    for (int i = 0; i < N*N; ++i) { A[i] = i % 100;
B[i] = (i*2) % 100; }

    tbb::global_control
gc(tbb::global_control::max_allowed_parallelism, T);
    auto t0 = chrono::high_resolution_clock::now();

    vector<int> rows(N); for (int i=0;i<N;++i)
rows[i]=i;
    for_each(std::execution::par, rows.begin(),
rows.end(),
        [&](int i){
            for (int k=0; k<N; ++k) {
                double a = A[i*N + k];
                for (int j=0; j<N; ++j)
                    C[i*N + j] += a * B[k*N + j];
            }
        });

    auto t1 = chrono::high_resolution_clock::now();
    double ms = chrono::duration<double, milli>(t1 -
t0).count();

    double checksum = 0; for (double v : C) checksum +=
v;
    cout << "N=" << N << " impl=pstl workers=" << T
         << " time_ms=" << ms << " checksum=" <<
checksum << "\n";
}
```

Додаток Ґ

Реалізація послідовного алгоритму множення матриць

```
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

int main(int argc, char** argv) {
    int N = (argc > 1) ? atoi(argv[1]) : 512;
    vector<double> A(N*N), B(N*N), C(N*N, 0.0);
    for (int i = 0; i < N*N; ++i) { A[i] = i % 100;
B[i] = (i*2) % 100; }

    auto t0 = chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i)
        for (int k = 0; k < N; ++k) {
            double a = A[i*N + k];
            for (int j = 0; j < N; ++j)
                C[i*N + j] += a * B[k*N + j];
        }
    auto t1 = chrono::high_resolution_clock::now();

    double ms = chrono::duration<double, milli>(t1 -
t0).count();
    double checksum = 0; for (double v : C) checksum +=
v;
    cout << "N=" << N << " impl=serial workers=1
time_ms=" << ms
         << " checksum=" << checksum << "\n";
}
```

Додаток Д

Множення матриць з використанням бібліотеки Intel TBB

```
#include <iostream>
#include <vector>
#include <chrono>
#include <tbb/blocked_range2d.h>
#include <tbb/parallel_for.h>
#include <tbb/global_control.h>
using namespace std;

int main(int argc, char** argv) {
    int N = (argc > 1) ? atoi(argv[1]) : 512;
    int T = (argc > 2) ? atoi(argv[2]) : 4;

    vector<double> A(N*N), B(N*N), C(N*N, 0.0);
    for (int i = 0; i < N*N; ++i) { A[i] = i % 100;
B[i] = (i*2) % 100; }

    tbb::global_control
gc(tbb::global_control::max_allowed_parallelism, T);
    auto t0 = chrono::high_resolution_clock::now();

    tbb::parallel_for(tbb::blocked_range2d<int>(0,N,64,
0,N,64),
        [&](const tbb::blocked_range2d<int>& r){
            for (int ii = r.rows().begin(); ii !=
r.rows().end(); ++ii)
                for (int jj = r.cols().begin(); jj !=
r.cols().end(); ++jj) {
                    double sum = 0.0;
                    for (int k = 0; k < N; ++k) sum +=
A[ii*N+k]*B[k*N+jj];
                    C[ii*N+jj] = sum;
                }
        });

    auto t1 = chrono::high_resolution_clock::now();
    double ms = chrono::duration<double, milli>(t1 -
t0).count();

    double checksum = 0; for (double v : C) checksum +=
v;
    cout << "N=" << N << " impl=tbb workers=" << T
```

```
          << " time_ms=" << ms << " checksum=" <<
checksum << "\n";
}
```

Додаток Е

```
import os
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

csv_filename = None
try:
    from google.colab import files
    print("оберіть файл:")
    uploaded = files.upload()
    if uploaded:
        csv_filename = list(uploaded.keys())[0]
        print(f" Завантажено файл: {csv_filename}")
except Exception:
    pass
# --- 2) Зчитування даних ---
df = pd.read_csv(csv_filename)
print(f"Зчитано {df.shape[0]} рядків, стовпці: {list(df.columns)}")

# --- 3) Робимо назви стовпців нечутливими до регістру ---
cols = {c.lower(): c for c in df.columns}
def col(*cands):
    for c in cands:
        if c in cols:
            return cols[c]
    raise KeyError(f"Не знайдено жодного зі стовпців: {cands}")

cN       = col('n')
cIMPL    = col('impl')
cWORKERS = col('workers')
cTIMEMS  = col('time_ms')

# --- 4) Нормалізація імен реалізацій ---
df[cIMPL] = df[cIMPL].astype(str).str.strip()

# --- 5) Базовий час T(1) (impl=serial, workers=1) для кожного N ---
serial = (df[(df[cIMPL].str.lower() == 'serial') & (df[cWORKERS] == 1)]
            [[cN, cTIMEMS]].rename(columns={cTIMEMS: 't1_ms'}))
assert not serial.empty, "❌ Не знайдено базових запусків (impl=serial,
workers=1)."

# --- 6) Розрахунок прискорення S(p) = T(1) / T(p) ---
df2 = (df.merge(serial, on=cN, how='inner')
         .assign(speedup=lambda t: t['t1_ms'] / t[cTIMEMS]))
# для Амдала аналізуємо лише паралельні реалізації
df2 = df2[df2[cIMPL].str.lower() != 'serial']

# --- 7) Модель Амдала ---
def amdahl_speedup(p, f):
    """ S(p) = 1 / ((1 - f) + f/p) """
    return 1.0 / ((1.0 - f) + f / p)

def fit_fraction(p_arr, s_arr):
    """Підбір f на щільній сітці [0..1] мінімізуючи MSE."""
    p = np.asarray(p_arr)
    s = np.asarray(s_arr)
    m = np.isfinite(p) & np.isfinite(s) & (p > 0) & (s > 0)
    p, s = p[m], s[m]
    if len(p) < 2:
        return np.nan, np.nan, np.nan
```

```
    grid = np.linspace(0.0, 1.0, 20001)
    best_f, best_sse = np.nan, np.inf
    for f in grid:
        s_model = amdahl_speedup(p, f)
        sse = np.mean((s - s_model) ** 2)
        if sse < best_sse:
            best_sse, best_f = sse, f
    smax = 1.0 / (1.0 - best_f) if best_f < 1 else np.inf
    return best_f, best_sse, smax

# --- 8) Підгін f для кожного (N, impl) + збереження індивідуальних графіків ---
plots_dir = "figs_amdahl"
os.makedirs(plots_dir, exist_ok=True)

rows = []
for N, gN in df2.groupby(cN):
    for impl, g in gN.groupby(cIMPL):
        p = g[cWORKERS].values
        s = g['speedup'].values

        f, mse, smax = fit_fraction(p, s)
        if np.isnan(f):
            continue

        rows.append({
            'N': N,
            'Реалізація': impl,
            'f_паралельна_частка': f,
            'непаралельна_частка_(1-f)': 1 - f,
            'Smax_теоретична_межа': smax,
            'MSE_похибка_апроксимації': mse
        })

        # індивідуальний графік: виміряний vs Амдал
        p_plot = np.linspace(min(p), max(p), 200)
        s_model = amdahl_speedup(p_plot, f)
        plt.figure(figsize=(6, 4), dpi=140)
        plt.plot(p, s, 'o', label='Виміряне прискорення')
        plt.plot(p_plot, s_model, '-', label=f'Модель Амдала (f={f:.3f},
Smax={smax:.1f})')
        plt.plot([min(p_plot), max(p_plot)], [min(p_plot), max(p_plot)], 'r--',
lw=1, alpha=0.5, label='Ідеал: y = x')
        plt.title(f'{impl} — Аналіз Амдала (N={N})')
        plt.xlabel('Кількість виконавців p (потоки/процеси)')
        plt.ylabel('Прискорення S(p)')
        plt.grid(alpha=0.3)
        plt.legend()
        plt.tight_layout()
        plt.savefig(os.path.join(plots_dir, f"amdahl_fit_{impl}_N{N}.png"))
        plt.close()

summary = pd.DataFrame(rows).sort_values(['N','Реалізація'])
summary_path = os.path.join(plots_dir, "amdahl_fit_summary.csv")
summary.to_csv(summary_path, index=False, encoding='utf-8-sig')
print(f"Збережено таблицю підсумків: {summary_path}")
display(summary.head(10))

if not summary.empty:
    Ns = sorted(summary['N'].unique())
    impl_order = None
    # зафіксуємо порядок реалізацій за першою групою
    firstN = Ns[0]
    impl_order = list(summary[summary['N']==firstN]['Реалізація'])
```

```
    # (А) f по реалізаціях для кожного N (2x2 субплоти)
    ncols = 2
    nrows = int(math.ceil(len(Ns)/ncols))
    fig, axes = plt.subplots(nrows, ncols, figsize=(12, 4*nrows), dpi=140,
squeeze=False)
    for i, N in enumerate(Ns):
        ax = axes[i//ncols][i%ncols]
        sub = summary[summary['N']==N].copy()
        if impl_order is not None:
            sub['Реалізація'] = pd.Categorical(sub['Реалізація'],
categories=impl_order, ordered=True)
            sub = sub.sort_values('Реалізація')

        ax.bar(sub['Реалізація'], sub['f_паралельна_частка'])
        ax.set_ylim(0, 1)
        ax.set_title(f'Оцінена паралельна частка f (N={N})')
        ax.set_ylabel('f (частка паралельного коду)')
        ax.set_xticklabels(sub['Реалізація'], rotation=25, ha='right')
        # підписи значень
        for x, y in enumerate(sub['f_паралельна_частка']):
            ax.text(x, y+0.02, f'{y:.2f}', ha='center', va='bottom', fontsize=9)
        ax.grid(axis='y', alpha=0.3)

    for j in range(i+1, nrows*ncols):
        axes[j//ncols][j%ncols].axis('off')

    plt.tight_layout()
    out_f = os.path.join(plots_dir, "summary_f_by_impl.png")
    plt.savefig(out_f)
    plt.close()

    # (Б) Smax по реалізаціях для кожного N (2x2 субплоти)
    fig, axes = plt.subplots(nrows, ncols, figsize=(12, 4*nrows), dpi=140,
squeeze=False)
    for i, N in enumerate(Ns):
        ax = axes[i//ncols][i%ncols]
        sub = summary[summary['N']==N].copy()
        if impl_order is not None:
            sub['Реалізація'] = pd.Categorical(sub['Реалізація'],
categories=impl_order, ordered=True)
            sub = sub.sort_values('Реалізація')

        ax.bar(sub['Реалізація'], sub['Smax_теоретична_межа'])
        ax.set_title(f'Теоретична межа прискорення Smax (N={N})')
        ax.set_ylabel('Smax = 1 / (1 − f)')
        ax.set_xticklabels(sub['Реалізація'], rotation=25, ha='right')
        # підписи значень
        for x, y in enumerate(sub['Smax_теоретична_межа']):
            ax.text(x, y + max(0.02*y, 0.2), f'{y:.1f}', ha='center', va='bottom',
fontsize=9)
        ax.grid(axis='y', alpha=0.3)

    for j in range(i+1, nrows*ncols):
        axes[j//ncols][j%ncols].axis('off')

    plt.tight_layout()
    out_s = os.path.join(plots_dir, "summary_smax_by_impl.png")
    plt.savefig(out_s)
    plt.close()

    # Додатково: зведений графік "f vs Smax" (картезіанський)
    plt.figure(figsize=(7,5), dpi=140)
    for N in Ns:
        sub = summary[summary['N']==N]
```

```
        plt.scatter(sub['f_паралельна_частка'], sub['Smax_теоретична_межа'], s=70,
label=f'N={N}')
        # підписати маркери назвами реалізацій
        for _, r in sub.iterrows():
            plt.annotate(r['Реалізація'], (r['f_паралельна_частка'],
r['Smax_теоретична_межа']),
                         textcoords="offset points", xytext=(4,4), fontsize=8)
    plt.xlabel('Паралельна частка f')
    plt.ylabel('Smax = 1 / (1 − f)')
    plt.title('Взаємозв’язок f та Smax для всіх реалізацій')
    plt.grid(alpha=0.3)
    plt.legend()
    plt.tight_layout()
    combo_path = os.path.join(plots_dir, "summary_f_vs_smax_scatter.png")
    plt.savefig(combo_path)
    plt.close()
```

Додаток Є

| | A |
|---|---|
| 1 | N,impl,workers,time_ms,checksum |
| 2 | 256,serial,1,2.4554,4.06723e+10 |
| 3 | 256,serial,1,2.4693,4.06723e+10 |
| 4 | 256,serial,1,2.6729,4.06723e+10 |
| 5 | 256,serial,1,2.4686,4.06723e+10 |
| 6 | 256,serial,1,2.4701,4.06723e+10 |
| 7 | 512,serial,1,31.5353,3.25509e+11 |
| 8 | 512,serial,1,21.5222,3.25509e+11 |
| 9 | 512,serial,1,17.6576,3.25509e+11 |
| 10 | 512,serial,1,17.5909,3.25509e+11 |
| 11 | 512,serial,1,17.4956,3.25509e+11 |
| 12 | 1024,serial,1,142.803,2.60428e+12 |
| 13 | 1024,serial,1,144.226,2.60428e+12 |
| 14 | 1024,serial,1,142.419,2.60428e+12 |
| 15 | 1024,serial,1,144.353,2.60428e+12 |
| 16 | 1024,serial,1,142.668,2.60428e+12 |
| 17 | 2048,serial,1,2454.07,2.08348e+13 |
| 18 | 2048,serial,1,2539.57,2.08348e+13 |
| 19 | 2048,serial,1,2343.28,2.08348e+13 |
| 20 | 2048,serial,1,2310.1,2.08348e+13 |
| 21 | 2048,serial,1,2380.72,2.08348e+13 |

| | |
|---|---|
| 397 | 1024,blas,8,6.01,2.60428e+12 |
| 398 | 1024,blas,8,6.0037,2.60428e+12 |
| 399 | 1024,blas,8,6.0122,2.60428e+12 |
| 400 | 1024,blas,8,6.01,2.60428e+12 |
| 401 | 1024,blas,8,5.9595,2.60428e+12 |
| 402 | 2048,blas,1,42.9166,2.08348e+13 |
| 403 | 2048,blas,1,43.68,2.08348e+13 |
| 404 | 2048,blas,1,44.2922,2.08348e+13 |
| 405 | 2048,blas,1,45.3467,2.08348e+13 |
| 406 | 2048,blas,1,50.6343,2.08348e+13 |
| 407 | 2048,blas,2,43.7419,2.08348e+13 |
| 408 | 2048,blas,2,41.953,2.08348e+13 |
| 409 | 2048,blas,2,42.4684,2.08348e+13 |
| 410 | 2048,blas,2,45.141,2.08348e+13 |
| 411 | 2048,blas,2,50.0196,2.08348e+13 |
| 412 | 2048,blas,4,64.3194,2.08348e+13 |
| 413 | 2048,blas,4,45.5635,2.08348e+13 |
| 414 | 2048,blas,4,43.1622,2.08348e+13 |
| 415 | 2048,blas,4,42.1012,2.08348e+13 |
| 416 | 2048,blas,4,44.0019,2.08348e+13 |
| 417 | 2048,blas,8,42.6783,2.08348e+13 |
| 418 | 2048,blas,8,42.7665,2.08348e+13 |
| 419 | 2048,blas,8,44.4707,2.08348e+13 |
| 420 | 2048,blas,8,46.2914,2.08348e+13 |
| 421 | 2048,blas,8,45.0878,2.08348e+13 |