Міністерство освіти і науки України
Криворізький національний університет
Кафедра комп’ютерних систем та мереж

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

за спеціальністю 123 - «Комп’ютерна інженерія»

Тема наукової роботи: МЕТОДИ І ЗАСОБИ ОРГАНІЗАЦІЇ ВИСОКОРЕЗУЛЬТАТИВНИХ ОБЧИСЛЕНЬ ДЛЯ ПАРАЛЕЛЬНОЇ АРХІТЕКТУРИ КОМП’ЮТЕРА

Виконав _____ В. Р. Кабанов

Керівник роботи _____ А. І. Купін

Нормоконтроль _____ Д. І. Кузнєцов

Завідувач кафедри _____ А. І. Купін

Кривий Ріг
2022

Криворізький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерних систем та мереж

Ступінь вищої освіти магістр
Спеціальність 123 - «Комп'ютерна інженерія»

**ЗАТВЕРДЖУЮ**
Завідувач кафедри

_____ А. І. Купін

"____" _____ 20__ року

# З А В Д А Н Н Я
## НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

_____
(прізвище, ім'я, по батькові)

1. Тема роботи _____
_____
_____

керівник роботи _____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від "___"_____ 20__року №__

2. Строк подання студентом роботи _____

3. Вихідні дані до роботи _____
_____
_____
_____
_____
_____

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

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
_____
_____
_____
_____

6. Консультанти розділів роботи

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

7. Дата видачі завдання_____

## КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів магістерської роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

**Студент** _____ _____
(підпис) (прізвище та ініціали)

**Керівник роботи** _____ _____
(підпис) (прізвище та ініціали)

# РЕФЕРАТ

Пояснювальна записка: 100 сторінок, 34 рисунка, 15 таблиць, 1 додаток, 40 використаних джерел.

Об’єкт дослідження – процеси паралельних високопродуктивних обчислень у комп’ютерних системах з багатоядерною архітектурою на основі процесорів родини AMD.

Робота складається з чотирьох розділів.

Перший розділ присвячено основним поняттям та методам оцінки продуктивності паралельних систем. Наведено класифікацію паралельних та розподілених обчислень. Розглянуто методи оцінки продуктивності паралельних алгоритмів і систем. Зроблено огляд сучасних процесорів типу AMD Ryzen Threadripper 1950X і 1920X. Виконано постановку завдання.

Другий розділ розкриває питання технічних вимог та організації обчислень у кластерних системах з багатоядерною архітектурою. Розглянуто моделі паралельних та розподілених обчислень.

У третьому розділі наведено схеми паралельних алгоритмів та запропоновано критерії ефективності розв’язання задачі. Представлено алгоритми перемноження матриці на матрицю і їх реалізація на структурах різного типу.

У четвертому розділі виконано розробку програм, проведено експерименти та проаналізовано вихідні результати. Наведено програму, моделюючу повний цикл роботи системи, яка складається з чотирьох потоків. Доведено, шляхом тестування, що потоки працюють з спільним ресурсом без конфліктів.

Ключові слова: ВИСОКОПРОДУКТИВНІ ОБЧИСЛЕННЯ, БАГАТОЯДЕРНІ ПРОЦЕСОРИ, AMD, КЛАСТЕРНА СИСТЕМА, КОМП’ЮТЕРНА СИСТЕМА, БАГАТОЯДЕРНА АРХІТЕКТУРА.

Master’s work: 100 pages, 34 figures, 15 tables, 1 addition, 40 used sources.

Object of research – the processes of parallel high-performance computing in computer systems with multi-core architecture based on base AMD processors.

The paper consists of four parts.

The first chapter is devoted to the basic concepts and methods for evaluating the performance of parallel systems. Classification of parallel and distributed computations is presented. The methods of estimating the performance of parallel algorithms and systems are considered. An overview of today's AMD Ryzen Threadripper 1950X type and 1920X processors was made. The task is executed.

The second chapter covers the issues of technical requirements and organization of computations in cluster systems with multi-core architecture. Models of parallel and distributed computing are considered.

In the third chapter, the schemes of parallel algorithms are presented and the criteria for the solution of the problem are proposed. The algorithms of multiplication of the matrix on the matrix and their realization on structures of different type are presented.

In the fourth chapter, the development of programs, experiments and the initial results were analyzed. The program that simulates the full cycle of the system, which consists of four streams, is given. It is proved by testing that the threads work with a shared resource without conflicts.

Keywords: HIGH-PRODUCTING CALCULATIONS, MULTIPLE PROCESSORS, AMD, CLUSTER SYSTEM, COMPUTER SYSTEM, MULTIPLE ARCHITECTURE.

ЗМІСТ

# ПЕРЕЛІК СКОРОЧЕНЬ

CSP – Communicating Sequential Processes;
HEDT – High-End Desktop;
HPC – High Performance Computing;
MPP – Massively Parallel Processing;
SMP – Symmetric Multi-Processing;
SNA – Systems Network Architecture;
ЕОМ – електронна обчислювальна машина;
КС – кластерна система;
КСБА – комп’ютерна система з багатоядерною архітектурою;
ПК – персональний комп’ютер.

# ВСТУП

Прогрес у розвитку обчислювальної техніки, зниження вартості обчислень, та широке розповсюдження обчислювальних систем, що мають високу продуктивність обчислень, стимулювали поширення використання обчислювально складних чисельних методів, моделювання за допомогою комп'ютерів у різноманітних областях наукових досліджень та на виробництві. Не є новиною той факт, що обчислювальна потужність високопродуктивних кластерів зростає за експоненційним законом, а також те, що на сьогоднішній день немає тенденції до зниження цих темпів зростання [1].

Зараз спостерігається стійка тенденція стосовно поширення та застосування кластерних обчислювальних систем. Для побудови кластерів використовують компоненти, які серійно випускаються, забезпечують високу продуктивність обчислень та масштабованість обчислювальної системи. Очевидно, що зміни в елементній базі призводять до певних змін у підходах до побудови паралельних алгоритмів. Зауважимо, що кластери є дешевою альтернативою до дорогих суперкомп'ютерів. Недоліком останніх є те, що вони покращення ефективності обчислювальної системи завдяки збільшенню тактової частоти процесора практично вичерпані або є економічно не вигідними.

Тому перспективною є ідея підвищення продуктивності архітектури обчислювальних систем унаслідок збільшення кількості процесорів. Багатопроцесорні та багатоядерні обчислювальні системи все ширше використовуються у різних предметних областях і поступово витісняють однопроцесорні. Зараз триває процес модернізації кластерів шляхом переходу на багатоядерні процесори виробництва компаній AMD, Intel та IBM [2].

**Актуальність роботи.** Для розв'язання багатьох задач (прогноз погоди, задачі гідро- і газодинаміки, квантової хімії, астрономії, спектроскопії, біології, ядерної фізики) необхідна висока продуктивність та висока швидкість передачі інформації по каналах зв'язку, великі об'єми оперативної і постійної пам'яті, які не можуть забезпечити типові обчислювальні засоби. Одним з шляхів забезпечення таких вимог є організація паралельних та розподілених високопродуктивних обчислень і відповідних моделей та засобів їх реалізації на основі багатоядерних процесорів родини AMD.

Причому, ефективність паралельної обробки залежить як від продуктивності комп'ютерів, так і від розмірів і структури пам'яті, пропускної здатності каналів зв'язку, використаних мов програмування, компіляторів, операційних систем, чисельних методів та інших математичних досліджень. Такий широкий обсяг параметрів вимагає проведення досліджень на різних рівнях: на рівні розпаралелення алгоритмів, створення спеціальних мов програмування, компіляторів, багатопроцесорних систем, неоднорідних систем, кластерів і систем, що розподілені на великих територіях.

**Мета і завдання дослідження.** Метою магістерської роботи є вдосконалення основних методів та алгоритмів організації паралельних та розподілених обчислень, принципів побудови відповідних структур, набуття початкових практичних навиків проектування таких засобів.

Для досягнення поставленої мети необхідно виконати наступні основні завдання дослідження:

1) проаналізувати апаратне забезпечення кластерних систем та визначити основні чинники впливу на організацію обчислень;

2) дослідити моделі паралельних обчислень;

3) розробити модель обчислень, що дозволяє описати поведінку і взаємодію процесів на рівні вузлів системи при різній організації системи зв'язків вузлів;

4) реалізувати модель обчислень у кластерних системах з багатоядерною архітектурою на основі родини AMD процесорів;

5) дослідити тупикові ситуації при використанні об'єктів комунікації і синхронізації.

**Об'єкт дослідження** – процеси паралельних високопродуктивних обчислень у комп'ютерних системах з багатоядерною архітектурою на основі процесорів родини AMD.

**Предмет дослідження** – моделі та засоби реалізації паралельних високопродуктивних обчислень у кластерних системах з багатоядерною архітектурою на основі процесорів родини AMD.

**Методи досліджень.** Для дослідження та розробки методів організації паралельних обчислень і архітектури обчислювальних систем на основі процесорів родини AMD використовуються положення системного аналізу, загальної теорії обчислювальних систем, елементи теорії інформації, ймовірності, графів, алгоритмів та множин.

**Наукова новизна одержаних результатів.** Вдосконалено основний метод організації паралельних та розподілених обчислень, що дозволяє оптимізувати взаємодію процесів з врахуванням наявності різних форм архітектури на основі процесорів родини AMD.

**Практичне значення отриманих результатів.** Запропоновані засоби організації обчислювальних процесів дозволяють підвищити ефективність обчислень у кластерах з різною структурною організацією. Запропонована математична модель доведена до практичної реалізації у вигляді програми і може бути використана при організації обчислювальних процесів у кластерних системах з багатоядерною архітектурою на основі процесорів родини AMD.

# 1 ОСНОВНІ ПОНЯТТЯ ТА МЕТОДИ ОЦІНКИ ПРОДУКТИВНОСТІ ПАРАЛЕЛЬНИХ СИСТЕМ

## 1.1 Поняття та класифікація паралельних та розподілених обчислень

У залежності від предметної області застосування є багато визначень термінів, які характеризують паралельні та розподілені обчислення. На основі аналізу літературних джерел і варіантів практичної реалізації можна так визначити ці терміни:

Паралельні обчислення – обчислення, що підтримуються на математичному, алгоритмічному, програмному чи апаратному рівні (на всіх або декількох) і забезпечують можливість паралельного виконання задачі [3].

Під терміном «паралельні обчислення» розуміється сукупність питань, які відносяться до створення ресурсів паралелізму в процесах розв'язання задач і гнучкому керуванню реалізацій цього паралелізму з метою досягнення найбільшої ефективності обчислювальної техніки.

Розподілені обчислення – обчислення, які підтримуються стандартними чи закритими протоколами обміну та незалежними апаратними засобами (комп'ютери, сервери), що представляються користувачу єдиним обчислювачем, придатним для вирішення складної задачі.

Стосовно використаних ресурсів можна стверджувати: здебільшого паралельні структури реалізовуються на спеціалізованих процесорах, розподілені структури – на універсальних (стандартних) комп'ютерах (серверах), які об'єднані в мережі різного типу.

Є коло обчислювальних задач, для розв'язку яких необхідні потужніші обчислювальні ресурси, ніж ті, які можуть забезпечити типові комп'ютери чи системи. У таких задачах необхідно забезпечити: надвисоку швидкодію, великий об'єм оперативної пам'яті, велику кількість інформації, що передається, обробку і зберігання великого об'єму інформації. При наявності хоча б однієї з наведених вимог використання паралельної обробки оправдано.

Конвеєризація – метод, що забезпечує сукупність різних дій за рахунок їх розбиття на підфункції з зміщенням у часі виконанням. Конвеєрний пристрій розробляють з окремих ступенів, час спрацювання яких в ідеальному випадку повинен бути однаковим.

Паралелізм – метод, що забезпечує виконання різних функцій шляхом їх розбиття на підфункції з одночасним виконанням в часі.

Розпаралелити кожну задачу можна не єдиним способом. Тому доцільно розглянути алгоритми можливого розпаралелення типових задач незалежно від конкретної програмної і платформенної реалізації. В загальному випадку процедура розпаралелення розіб'ється на 3 етапи:

- розбиття задачі на незалежні під задачі;
- призначення конкретних процесорів для виконання кожної під задачі;
- збирання результатів роботи окремих процесорів.

Є різні системи класифікації паралельних систем, які базуються на різнотипних головних ознаках до класифікації. Багато з них базується на класифікації М. Флінна (1966 р.), де паралельна обробка виконується на SIMD і MIMD архітектурах.

У сучасних класифікаційних системах архітектури, які попадають в один клас, відрізняються за кількістю процесорів, природі і топології зв'язку між ними, за способом організації пам'яті, за технологією програмування та іншими ознаками.

Наприклад, А. Базу (A. Basu) вважає, що будь-яку паралельну обчислювальну систему можна однозначно описати послідовністю рішень, прийнятих на етапі її проектування, а сам процес проектування представити у виді дерева. Класифікація за Базу наведена на рис.1.1.

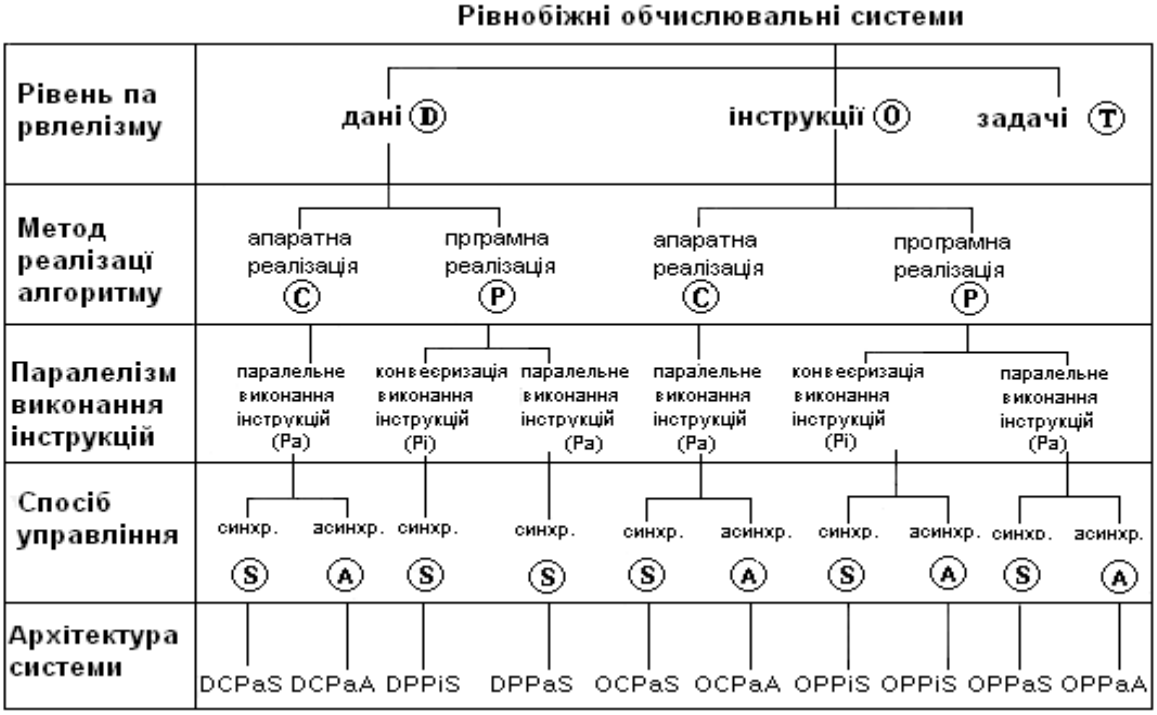



Рисунок 1.1 – Класифікація за Базу

Р. Дункан визначає такий набір вимог на який може спиратися класифікація (див. рис. 1.2):

З класу паралельних машин повинні бути виключені ті, у яких паралелізм закладений лише на найнижчому рівні, включаючи:

- конвеєризацію на етапі підготовки і виконання команди (instruction pipelining), тобто часткове перекриття таких етапів, як дешифрація команди, обчислення адрес операндів, вибірка операндів, виконання команди і збереження результату;

- наявність в архітектурі декількох функціональних пристроїв, що працюють незалежно, зокрема, можливість паралельного виконання логічних і арифметичних операцій;

- наявність окремих процесорів вводу/виводу, що працюють незалежно і паралельно з основними процесорами.

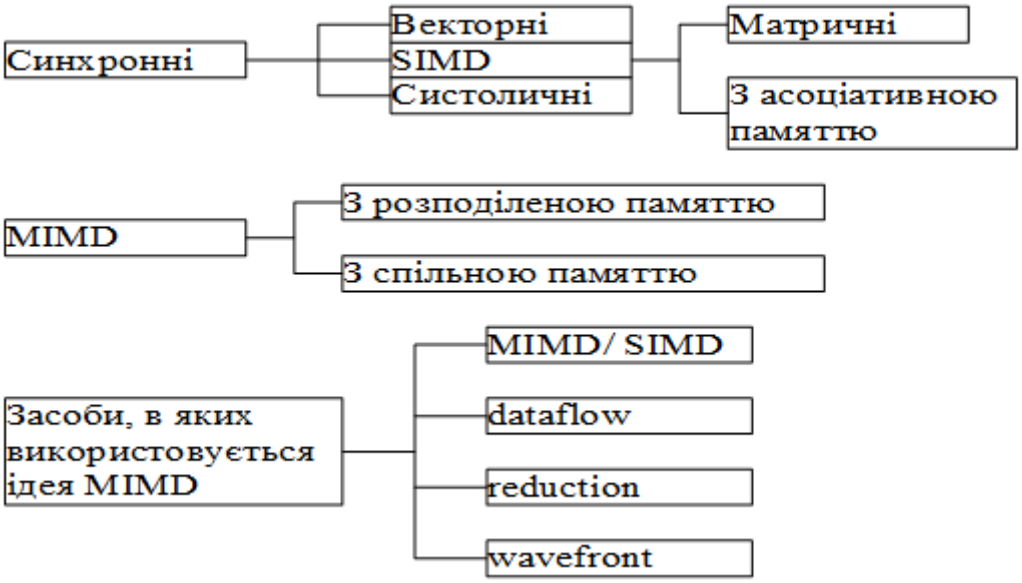



Рисунок 1.2 – Вимоги до класифікації за Р. Дунканом

результат її роботи потрібно іншій, доступній для виконання, команді як операнд.

Wavefront array архітектура поєднує в собі ідею систолічної обробки даних і модель обчислень, що використовується в dataflow. У даній архітектурі процесори об’єднуються в модулі і фіксуються зв'язки, по яких процесори можуть взаємодіяти один з одним. Однак, на противагу ритмічній роботі систоличних масивів, дана архітектура використовує асинхронний механізм зв'язку з підтвердженням (handshaking), через що «фронт хвилі» обчислень може змінювати свою форму в міру проходження по всіх процесорах.

Е. Кришнамарфі для класифікації паралельних обчислювальних систем пропонує використовувати чотири характеристики, які подібні до характеристик класифікації А.Базу: ступінь гранулярності, спосіб реалізації паралелізму, топологія і природа зв'язку, спосіб керування процесорами.

Незважаючи на те, що класифікація Е. Кришнамарфі побудована лише на чотирьох ознаках, вона дозволяє виділити і описати такі «нетрадиційні» паралельні системи, як систоличні масиви, машини типу dataflow і wavefront. Також існують класифікації Хандлера (ерлангерська схема) та Р. Хокні [3].

## 1.2 Методи оцінки продуктивності паралельних алгоритмів і систем

Паралельні системи характеризуються: різноплановістю задач, типом опрацювання (векторне, скалярне), різними операційними системами, різними конфігураціями систем, одночасністю виконання операцій, складністю взаємодії між елементами системи, що не дозволяє використовувати стандартні підходи до визначення їх продуктивності.

А продуктивність окремого процесора залежить від: частоти синхронізації, середньої кількості тактів на команду, кількості команд, що виконується.

Тому час виконання деякої програми в процесорі може бути виражений двома способами: кількістю тактів синхронізації для даної програми, які перемножуються на тривалість такту синхронізації, або кількістю тактів синхронізації для даної програми поділеними на частоту синхронізації.

У процесі пошуку стандартної одиниці вимірювання продуктивності комп'ютерів було прийнято декілька популярних одиниць вимірювання, а для оцінки продуктивності паралельних вузлів деякі терміни були штучно вирвані з їх контексту і використані там, для чого вони ніколи не призначалися. Насправді єдиною і надійною одиницею вимірювання продуктивності є час виконання реальних програм, і всі пропоновані заміни цього часу як одиниці вимірювання або заміни реальних програм як об'єктів вимірювання на синтетичні програми тільки вводять в оману [3].

Небезпеки використання популярних альтернативних одиниць вимірювання (MIPS і MFLOPS) для оцінки продуктивності паралельних систем.

MIPS – швидкість виконання операцій за одиницю часу, тобто - відношення кількості команд в програмі до часу її виконання. MIPS:

- залежить від набору команд процесора, що затрудняє порівняння за показниками MIPS комп'ютерів, що мають різні системи команд;
- навіть на тому ж комп'ютері змінюється від програми до програми;
- змінюватися по відношенню до продуктивності в протилежний бік (при більшому значенні MIPS реальна продуктивність менша).

Приклад для останнього випадку.

До складу процесора входять вузли обчислення елементарних функцій. За відсутності вузлів операції обчислення елементарних функцій виконуються за допомогою підпрограм. Тоді, такі процесори мають вищий рейтинг MIPS, але

виконують більшу кількість команд, що приводить до збільшення часу виконання програми.

MFLOPS - мільйон операцій з рухомою крапкою за секунду. Як одиниця вимірювання, MFLOPS, призначена для оцінки продуктивності тільки операцій з рухомою крапкою, і тому не застосовується поза цією обмеженою областю.

Наприклад, програми компіляторів мають рейтинг MFLOPS близький до нуля не залежно від того, наскільки швидка машина, оскільки компілятори рідко використовують арифметику з рухомою крапкою.

Рейтинг MFLOPS залежить і від характеристик процесора і від програми, і є об'єктивнішим ніж параметр MIPS, оскільки базується на кількості виконаних операцій, а не на кількості виконаних команд.

Проте і з використанням MFLOPS для оцінки продуктивності виникають певні проблеми. Перш за все, це пов'язане з тим, що набори операцій з рухомою крапкою не сумісні на різних комп'ютерах. Наприклад, в суперкомп'ютерах фірми Cray Research відсутня команда ділення (є, правда, операція обчислення оберненої величини числа з рухомою крапкою, а операція ділення може бути реалізована за допомогою множення діленого на зворотну величину дільника). В той же час сучасні мікропроцесори мають команди ділення, обчислення квадратного кореня, синуса і косинуса, тощо.

Інша проблема полягає в тому, що рейтинг MFLOPS міняється не тільки на суміші цілочисельних операцій і операцій з рухомою крапкою, але і на суміші швидких і повільних операцій з рухомою крапкою. Наприклад, програма з 100% операцій додавання матиме вищий рейтинг, ніж програма з 100% операцій ділення.

Поява векторних і паралельних процесорів і систем, не зменшила важливості Ліверморських циклів, проте змінилися значення продуктивності і величини розкиду між різними циклами. На векторних структурах продуктивність залежить не тільки від елементної бази, але і від характеру самого алгоритму, тобто коефіцієнта векторизації.

На паралельній машині продуктивність істотно залежить від відповідності між структурою апаратних засобів і структурою обчислень в алгоритмі. Важливо, щоб тестовий пакет представляв алгоритми різних структур. Тому в Ліверморських циклах зустрічаються послідовні, сіткові, конвеєрні, хвильові обчислювальні алгоритми, що підтверджує їх придатність і для паралельних машин.

При оцінці продуктивності необхідно враховувати: тип алгоритму, тип програмного забезпечення, параметри протоколів каналів передачі даних, структуру окремого процесора.

Тип алгоритму. Алгоритм, що ідеально пристосований для роботи на одній архітектурі, на іншій (з цією ж кількістю процесорів) може працювати

Розрахунковий метод. Базується на обчисленні продуктивності. Є трудомістким і недосконалим.

Практичний метод. Розв'язання конкретної задачі на конкретній структурі. Дає об'єктивну оцінку продуктивності. Недолік – продуктивність можна оцінити тільки після виготовлення системи. При негативному результаті – висока ціна помилки.

Характеристиками продуктивності паралельних алгоритмів є: фактор прискорення, максимальне прискорення (закон Амдала), ефективність паралельного алгоритму, ціна, масштабність, загальний час виконання паралельного алгоритму, повний час виконання паралельного алгоритму, теоретичний час комунікацій.

Оцінюючи продуктивність безпосередньо на паралельних системах, відзначають позитивний ефект від розпаралелювання (Speedup – прискорення) і вигоди від збільшення масштабності розв'язаних задач (Scaleup).

Показник Speedup визначає, у скільки разів швидше може бути вирішена одна й та ж задача на N процесорах порівняно з її вирішенням на одному процесорі.

Показник Scaleup визначає, у скільки разів більшу за розмірами проблему можна вирішити за той же час N процесорами порівняно з проблемою, що вирішується одним процесором.

Амдалом сформульований «закон Амдала», де єдиним параметром є поділ програми на послідовну і паралельну частини; масштаб вирішуваної задачі залишається постійним. Розгляньмо дещо спрощену формулу цього закону.

Фактор прискорення. Прискоренням (speedup factor) паралельного алгоритму в N – процесорній системі називається величина S(N) = T1/TN, де

- T1 – час виконання алгоритму на одному процесорі чи однопроцесорній системі;

- TN – час виконання алгоритму на багатопроцесорній системі з N - процесорами.

Закон Амдала. Позначимо:

- Pc – максимальний ступінь розпаралелення (максимальна кількість процесорів, які можуть працювати паралельно в будь-який момент часу в період виконання програми). Тут вирішальне значення має також тип застосовуваної моделі паралельності (наприклад, MIMD чи SIMD);

- Tk – тривалість виконання програм з максимальним показником розпаралелення на системі з k процесорами;

- N – кількість процесорів у паралельній системі;

- f – відсоток послідовних операцій програми (не можуть бути виконані паралельно на N процесорах).

Тривалість виконання програми на паралельній системі з N процесорами оцінюється формулою:

$$T_N = f * T_1 + (1 - f) * \frac{T_1}{N} \quad (1.1)$$

звідки дістанемо показник прискорення (Speedup) в системі з $N$ процесорами

$$S_N = \frac{T_1}{T_N} = \frac{N}{1 + f * (1 - N)}. \quad (1.2)$$

Оскільки, $0 < f < 1$, справедливе таке співвідношення:

$$1 \le S_N \le N,$$

тобто показник прискорення не може бути більшим ніж кількість процесорів $N$.

Як міра досягнутого прискорення, відносно максимального визначається ефективність системи з N процесорами

$$E_N = S_N / N \quad (1.3)$$

Область межі ефективності : $1/N \le E_N \le 1$

На практиці використовують значення $E_N$ у відсотках. При $E_N$ = 0,9. наприклад, могла б бути досягнута ефективність, що дорівнює 90% максимально можливої.

З збільшенням послідовної частини програми падає завантаження процесорів, а з ним і показник прискорення порівняно з послідовною обчислювальною системою. Для кожної послідовної частини програми може бути обчислений максимально можливий показник прискорення незалежно від кількості застосовуваних процесорів:

$$\lim_{N \to \infty} S_N(f) = \frac{N}{1 + f * (N - 1)} = \frac{1}{f} \quad (1.4)$$

Це означає, що програма із скалярною частиною, яка становить 1%, ніколи не зможе досягти показника прискорення (Speedup), що був би більшим 100 – незалежно від того, застосовується 100,1000 або 1000000 процесорів.

Ефективність паралельного алгоритму E показує у скільки разів більший час виконання завдання одним процесором $T_1$, ніж час виконання цього ж завдання багатопроцесорною системою $T_N$, помножений на кількість процесорів N. $E = T_1/(T_N*N) = S_N/N$. Ефективність характеризує ту частину часу, яку процесори використовують на обчислення.

Ціна (робота) обчислення визначається як Cost = (час виконання) * (повне число використаних процесорів). Ціна послідовних обчислень рівна часу їх виконання $T_1$. Ціна паралельних обчислень рівна $T_N*N$. З іншого боку $T_N = T_1/S_N$. Звідси ціна паралельних обчислень: $Cost = T_1*N/S_N = T_1/E$.

Ціноптимальні паралельні алгоритми – алгоритми, в яких ціна для вирішення проблеми на багатопроцесорній системі є пропорційною ціні (часу виконання) на однопроцесорній системі [3].

## 1.3 Огляд апаратного забезпечення багатоядерних систем AMD

Для реалізації процесу паралельного виконання задач більш ефективно інтегрувати два ядра або більше в одному мікропроцесорі [4]. Така багатоядерна конфігурація на одному кристалі забезпечує більш високу швидкість обміну між ядрами, чим використання зовнішніх шин, комутаторів тощо у багатопрцесорних системах.

Багатоядерна архітектура надає два або більше повофункціональних набори ресурсів для підвищення продуктивності процесора. Багатоядерність й 65нм техпроцес дозволили домогтися значної економії енергоспоживання й підвищення продуктивності на 1 Вт споживаній потужності.

Разом із багатоядерними процесорами, в архітектуру нових платформ впроваджуються такі нові технології, як незалежність відповідних програмних компонентів (Intel Virtualization Technology – VM), прискорення механізму обміну даними (Intel I/O Acceleration Technology - I/O AT), віддалена керованість (Intel Active Management Technology - iAMT).

Комплекс нових технологій спрямований на підвищення ефективності обчислювальної платформи в цілому віртуалізація (забезпечення віртуальної незалежності роботи кожного ядра), безпека й удосконалена технологія пам'яті. Багатоядерні рішення від AMD

Корпорація AMD розробила технологію віртуализації Pacifica і технологію безпеки Presidio для серійних багатоядерних процесорів [5]. Технологія віртуализації Pacifica дозволяє запускати на одному комп'ютері кілька незалежних операційних систем і додатків, а технологія безпеки Presidio підвищує безпека роботи з даними за допомогою спеціальної захищеної області в процесорі.

Багатоядерні чіпи, спроектовані інженерами AMD, відрізняються більш низьким енергоспоживанням, оскільки виконані на єдиній підкладці, на відміну від кристалів Intel, де механічно з'єднуються кілька окремих ядер. Система Cool'n'Quiet автоматично регулює тактову частоту й напругу живлення процесора залежно від реального навантаження, а також швидкість обертання процесорного кулера залежно від температури процесора. Конструкція процесорів AMD забезпечує найбільш тісні зв'язки між всіма ядрами, що дозволяє не тільки підвищити продуктивність, але й оптимізувати споживання електроенергії. Напрямок на випуск багатоядерних процесорів змушує відмовитися від схем, оптимальних лише для одноядерних мікросхем. Можливе створення вузькоспеціалізованих моделей процесорів, чітко орієнтованих на рішення тих або інших завдань. Приміром, для серверів продуктивність процесора у векторних операціях не потрібна, тому за рахунок оптимізації конструкції можна домогтися підвищеної потужності в пошукових обчисленнях і зниження енергоспоживання. При цьому важливо зберегти взаємну сумісність

набору інструкцій різних модифікацій – це не тільки здешевить виробництво, але й унеможливить проблеми при написанні програмного забезпечення. Для об'єднання обчислювальних ядер процесорів AMD використає архітектуру Direct Connect (прямого з'єднання), а для зв'язку з набором системної логіки використовується шина HyperTransport. Третя версія цієї шини в три рази перевищує пропускну здатність HyperTransport 2.0. Сукупна максимальна пропускна здатність HyperTransport 2.0. становить 22,4 Гб/с. У новій версії шини зберігається їїтрадиційна перевага - прямий зв'язок між процесором і системами уведення/виводу, тільки тепер ці прямі лінії зв'язку будуть установлюватися здекількома ядрами.

Компанія влітку 2004 р. продемонструвала оснащений двома ядрами 64-розрядний процесор типу AMD Opteron (з підтримкою системи команд x86) [6]. На демонстрації був представлений сервер HP ProLiant DL585 із чотирма двуядерними процесорами Opteron, виготовленими за технологією «кремній на ізоляторі» (SOI) з дотриманням проектних норм 90 нм. Таким чином, компанія AMD заклала фундамент для створення процесорів із двома ядрами, забезпечивши свої одноядерні процесори типу AMD 64 вбудованою інфраструктурою для підтримки другого ядра на тієї же мікросхемі. Двуядерні процесори являють собою природне розширення технології AMD 64 з архітектурою прямих з'єднань. Компанія AMD усунула вузькі місця в роботі зовнішньої шини, характерні для архітектури x86, вона об'єднала два ядра на одному кристалі разом з контролером пам'яті й підсистемою вводу-виводу.

Це дозволило поліпшити загальносистемну продуктивність і підвищити ефективність обробки даних. Архітектура прямих з'єднань - Direct Connect дозволяє підключити кілька процесорів, контролер пам'яті й модулі уведення-виводу прямо. Вона нейтралізує недоліки сучасних системних архітектур й усуває вузькі місця при обміні даними. Уважається, що дана архітектура скорочує затримки при звертанні до пам'яті, а оскільки засобу вводу-виводу безпосередньо підключаються до центрального процесора, поліпшується баланс між продуктивністю процесора й пропускною здатністю підсистеми уведення-виводу.

У свою чергу, всі процесори з'єднуються безпосередньо один з одним, що забезпечує практично лінійне підвищення продуктивності для багатопроцесорних систем. З зростанням числа процесорів пропускна здатність пам'яті росте лінійно.

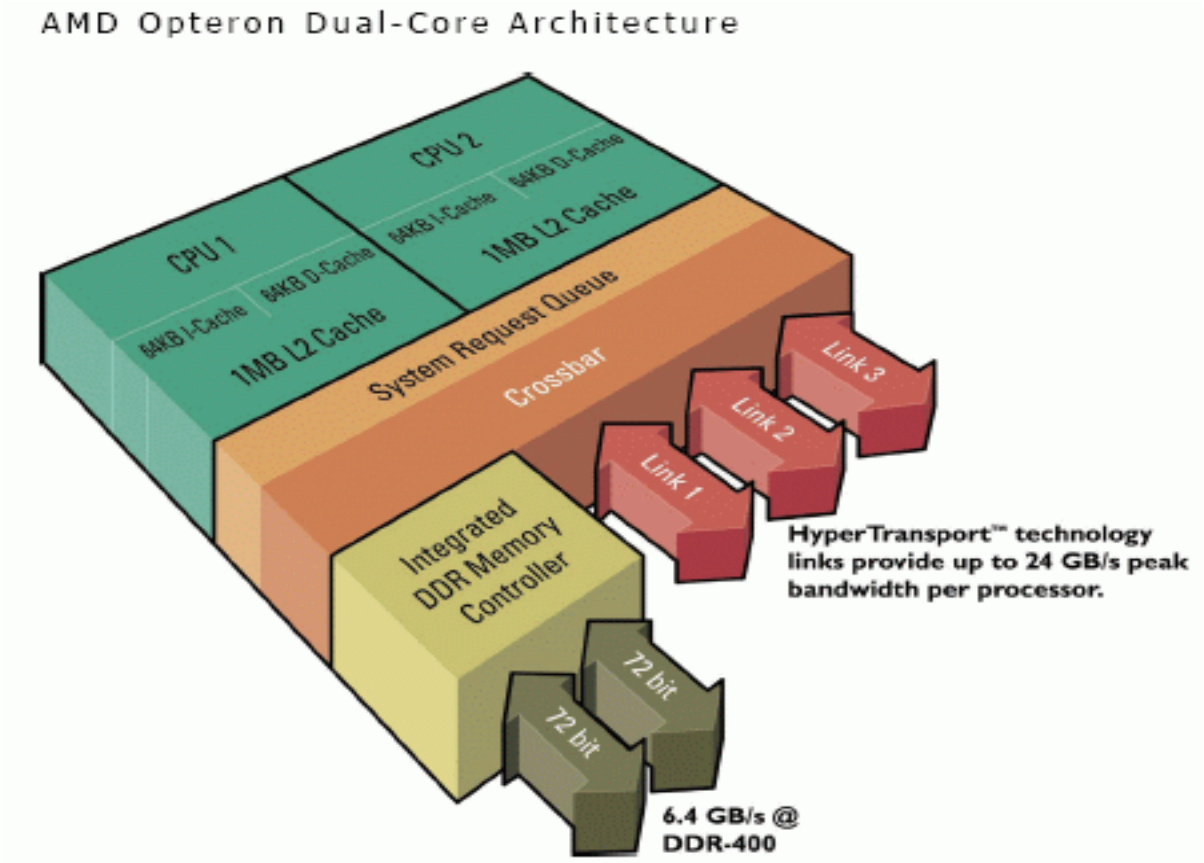



Рисунок 1.3 – Схема двуядерного процесора Opteron

Двуядерні процесори AMD 64 підтримують сумісність із додатками AMD 64 для платформи x86, що спрощує їх масове поширення. Слід зазначити, що реалізація двуядерності в процесорах AMD трохи відрізняється від реалізації Intel. AMD пропонує трохи інший спосіб взаємодії ядер між собою. Підхід Intel полягає в простому розташуванні на один кристал двох ядер. При такій організації двуядерності процесор не має ніяких спеціальних механізмів для здійснення взаємодії між ядрами. Як й у звичайних двупроцесорних системах, ядра спілкуються за допомогою системної шини. Відповідно, системна шина розділяється між ядрами процесора й при роботі з пам'яттю, що приводить до збільшення затримок при звертанні до пам'яті двох ядер одночасно.

У процесорах AMD дубльовані деякі ресурси. Хоча кожне з ядер Athlon 64 X2 має власний набір виконавчих пристроїв і виділеної кеш- пам'яттю другого рівня, контролер пам'яті й контролер шини Hyper-Transport на обидва ядра загальний. Взаємодія кожного з ядер з поділюваними ресурсами здійснюється за допомогою спеціального Crossbar-перемикача й черги системних запитів (System Request Queue). На цьому ж рівні організоване й взаємодія ядер між собою, завдяки чому питання когерентності кеш пам'яті вирішуються без додаткового навантаження на системну шину й шину пам'яті

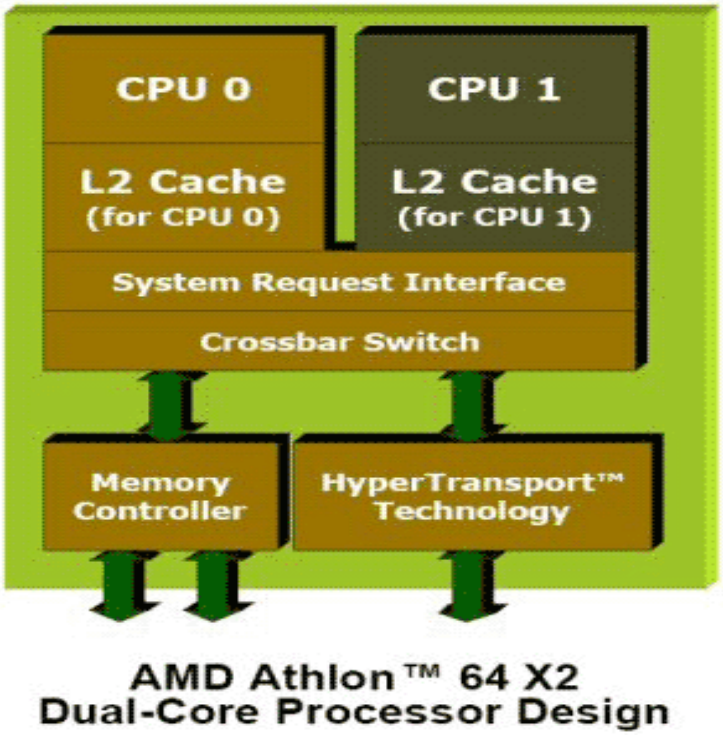



Рисунок 1.4 – Схема двуядерного процесора Athlon 64 X2

Таким чином, єдине вузьке місце, наявне в архітектурі Athlon 64X2 - це пропускна здатність підсистеми пам'яті (6.4 Гбайт у секунду), що ділиться між процесорними ядрами. AMD прагнула зберегти сумісність Athlon 64 X2 з існуючими платформами.

У результаті, ці процесори стало можливо використати на тих же самих материнських платах, що й одноядерні Athlon 64. Тому, Athlon 64 X2 мають такий же корпус, двухканальний контролер пам'яті з підтримкою DDR400 SDRAM й працюють із шиною HyperTransport із частотою до 1 Ггц. Завдяки цьому єдине, що треба було для підтримки двухядерних CPU від AMD материнськими платами, – це відновлення BIOS. Інженерам AMD удалося вписати в раніше встановлені рамки й енергоспоживання процесора Athlon 64 X2. Athlon 64 X2 підтримують набір інструкцій SSE3, а також мають удосконалений контролер пам'яті. Серед особливостей контролера пам'яті Athlon 64 X2 варто згадати можливість використання різномастих модулів DIMM у різних каналах (аж до установки в обидва канали пам'яті модулів різного розміру) і можливість роботи із чотирма двосторонніми модулями DIMM у режимі DDR400.

Процесори Athlon 64 X2 (Toledo), що містять два ядра з кеш-пам'яттю другого рівня по 1 Мбайту на кожне ядро, складаються із приблизно 233.2 млн. транзисторів й мають площа близько 199 кв. мм. Кристал і складність двуядерного процесора виявляється приблизно вдвічі більше кристала відповідного одноядерного. Наявність посередника між процесором і пам'яттю в особі північного моста для багатоядерних процесорів Intel збільшує затримки.

При цьому пропускна здатність пам'яті не росте із числом процесорів, а навпаки, процесори розділяють пропускну здатність системи процесор – пам'ять (до 6,4 Гбайт/с) між собою. Для компенсації проблеми пропускної здатності оперативної пам'яті в Intel пропонують збільшити ємність кеш пам'яті.

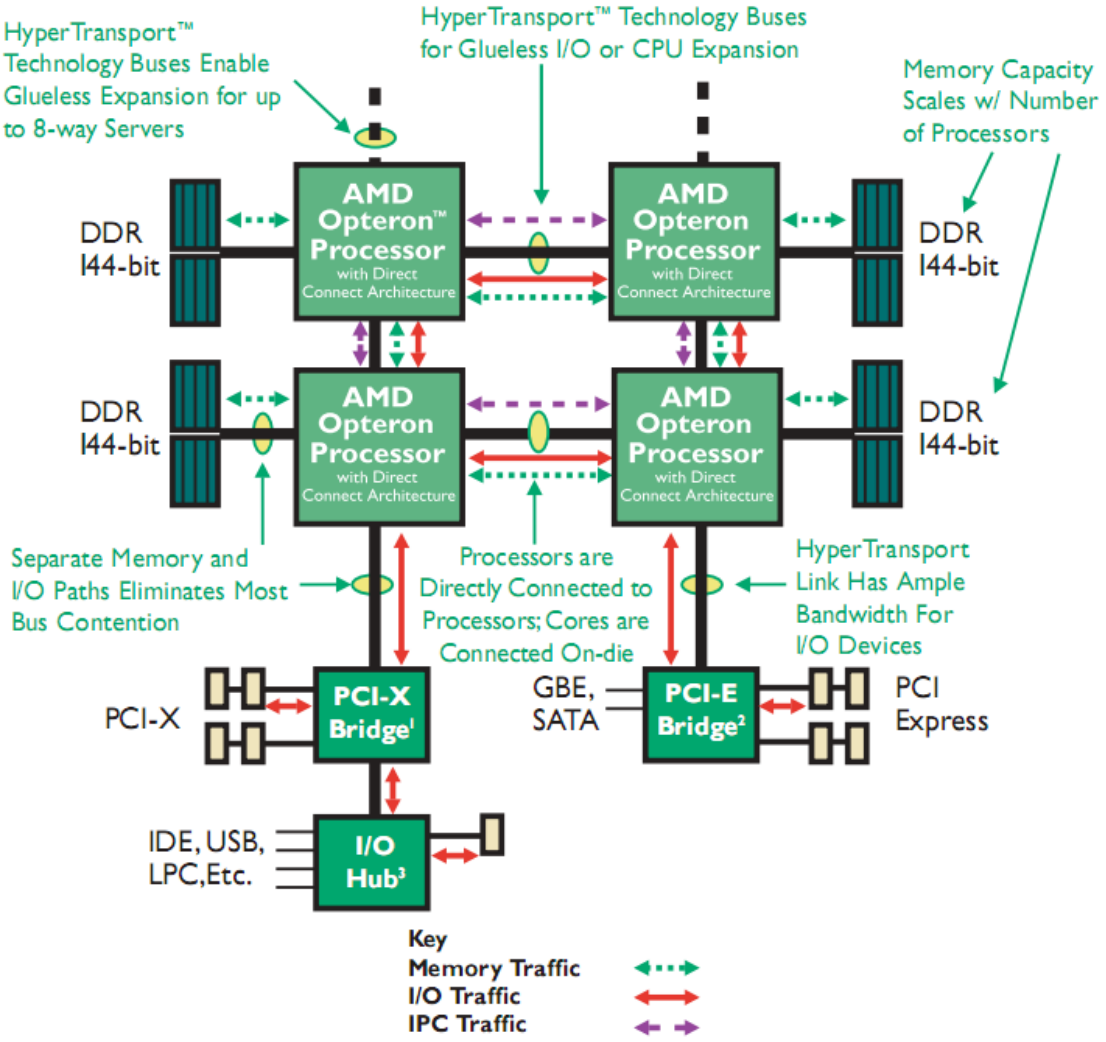



Рисунок 1.5 – Схема чотирьохядерного процесора типу AMD Opteron

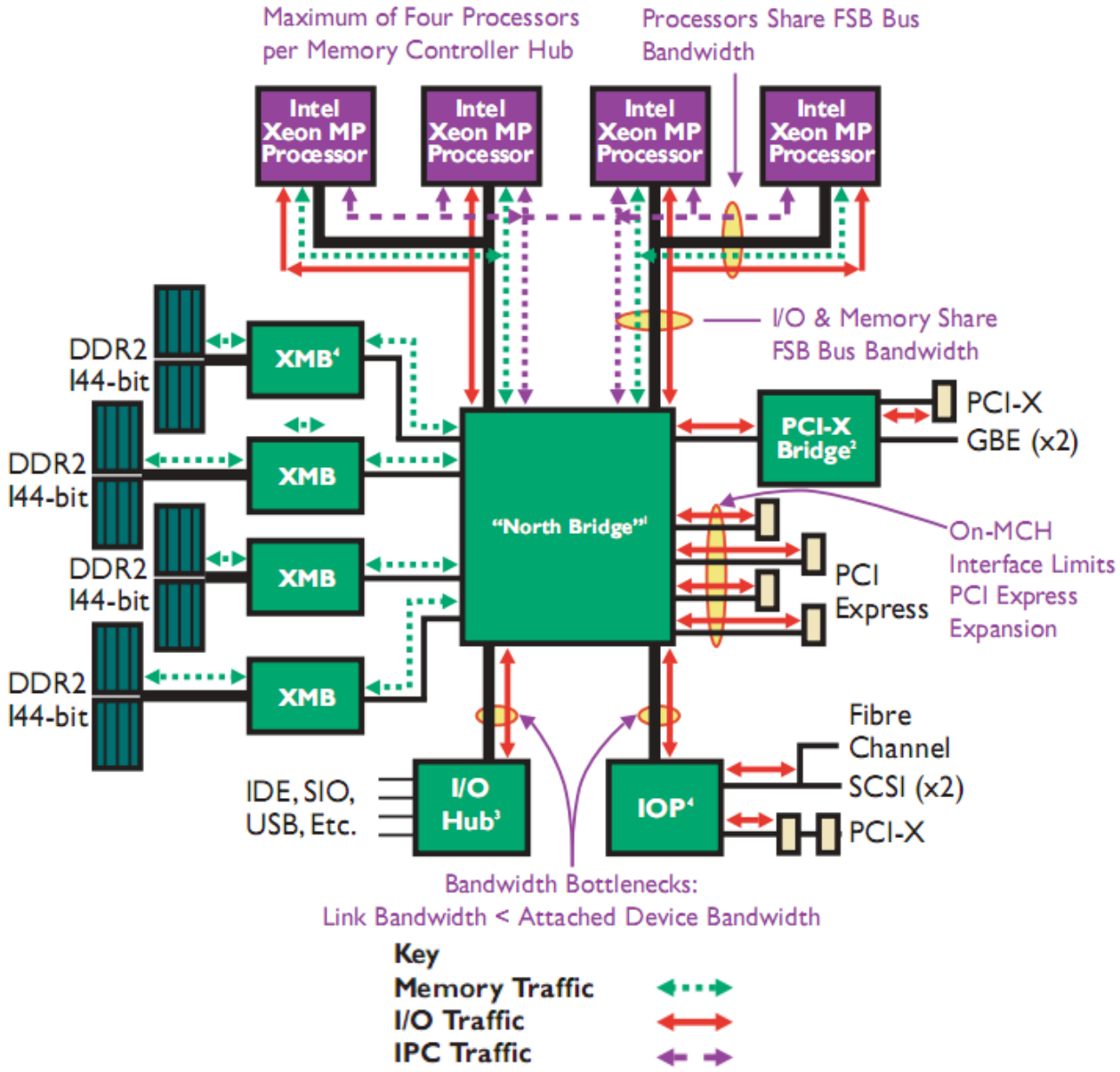



Рисунок 1.6 – Схема чотирьохядерного процесора Intel Xeon MP

Використання високошвидкісного інтерфейсу HyperTransport в Opteron/Athlon64 припускає, як з'єднання процесорів через HyperTransport з підтримкою когерентності кеша в багатопроцесорних серверах, так і пряме – без північного й південного мостів - приєднання через HyperTransport, мости для шин PCI-X/PCI-Express, що також підвищує продуктивність. Сумарна пропускна здатність уведення/виводу для 8-процесорних систем на базі Opteron 8xx досягає 30,4 Гбайт/с. Побудова серверів на базі Opteron відповідає архітектурі кеш когерентної неоднорідної пам'яті ( ccNUMA), поки з невеликою кількістю (до 8) процесорів. Можна згадати й інші переваги, наприклад, низькі величини затримок при роботі з ієрархією пам'яті. Так, затримка Opteron при вибірці з кеш пам'яті даних першого рівня дорівнює трьом тактам. Два порти читання дозволяють виконувати одночасно дві таких операції. Пропускна здатність пам'яті лімітує продуктивність багатьох додатків. Пропускна здатність важлива, наприклад, при рішенні завдань гідро- і аеродинаміки, де

використаються сіткові методи рішення рівнянь у частках похідних. Класичний приклад - короткостроковий прогноз погоди.

Розглянемо список найпродуктивніших самих швидких комп'ютерів світу за листопад 2017 на рисунку 1.7. На п'ятій сходинці Titan – Cray XK7 на AMD процесорах типу Opteron 6274.

| Rank | System | Cores | Rmax (TFlop/s) | Rpeak (TFlop/s) | Power (kW) |
|---|---|---|---|---|---|
| 1 | **Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC<br>National Supercomputing Center in Wuxi<br>China | 10,649,600 | 93,014.6 | 125,435.9 | 15,371 |
| 2 | **Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT<br>National Super Computer Center in Guangzhou<br>China | 3,120,000 | 33,862.7 | 54,902.4 | 17,808 |
| 3 | **Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc.<br>Swiss National Supercomputing Centre (CSCS)<br>Switzerland | 361,760 | 19,590.0 | 25,326.3 | 2,272 |
| 4 | **Gyoukou** - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler<br>Japan Agency for Marine-Earth Science and Technology<br>Japan | 19,860,000 | 19,135.8 | 28,192.0 | 1,350 |
| 5 | **Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc.<br>DOE/SC/Oak Ridge National Laboratory<br>United States | 560,640 | 17,590.0 | 27,112.5 | 8,209 |

Рисунок 1.7 – Список TOP 500 (листопад 2017)

Розглянемо більш детально технічні характеристики цього процесора на рисунках 1.8 – 1.10.

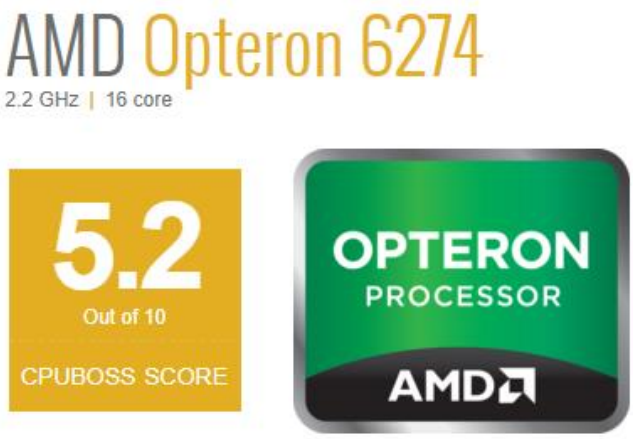



Рисунок 1.8 – Повна назва процесора

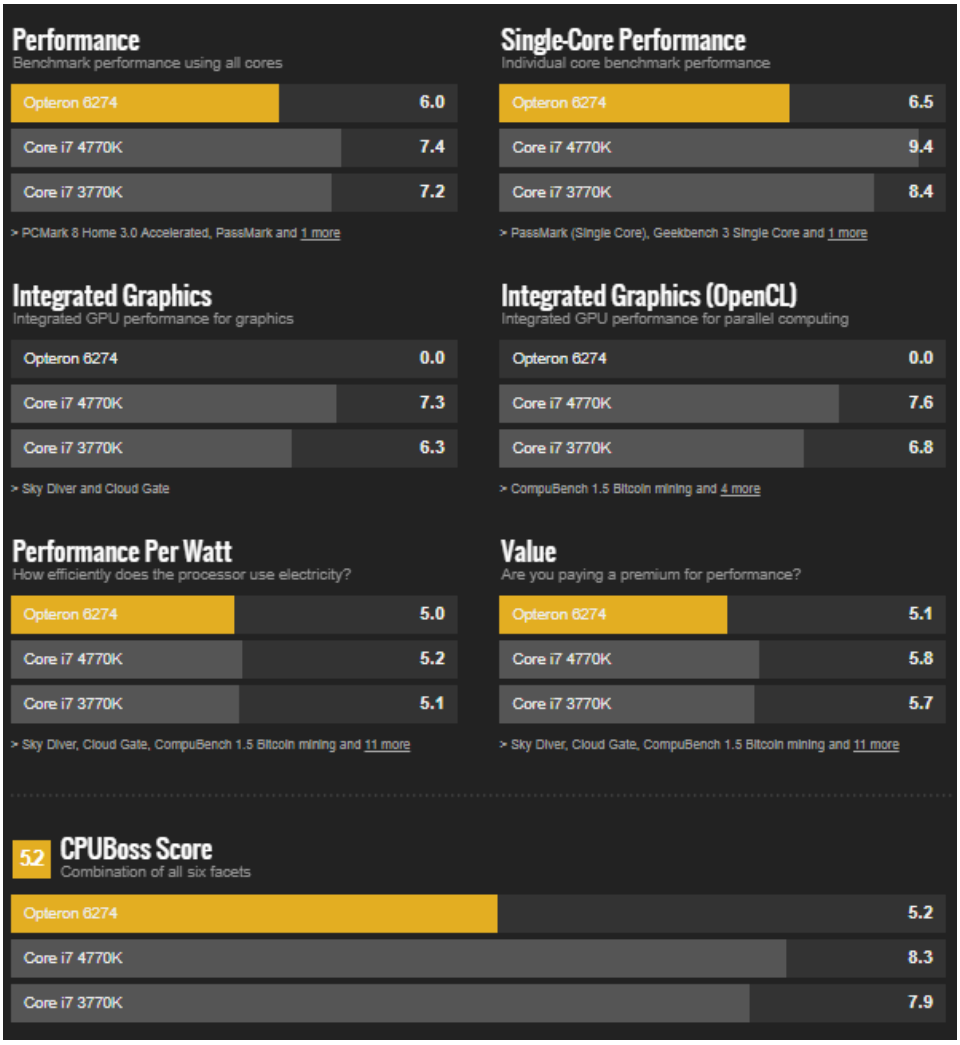



Рисунок 1.9 – Порівняння з Intel процесорами

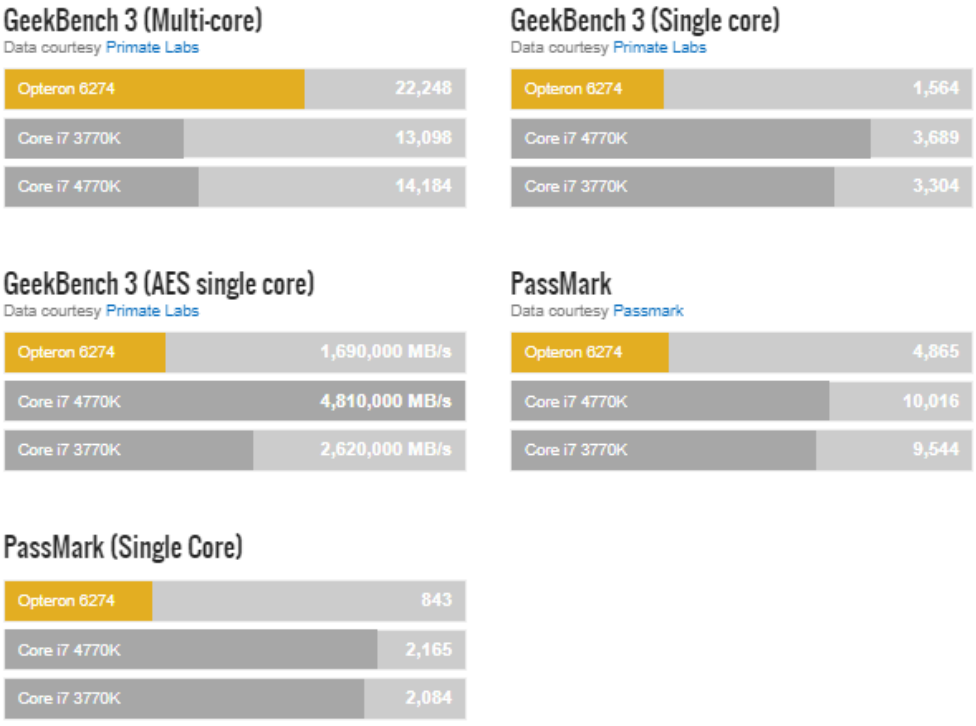



Рисунок 1.10 – Тести продуктивності

## 1.4 Процесори AMD Ryzen Threadripper 1950X і 1920X

Платформа Socket AM4 перетворилася з експериментальної продукції з неясними перспективами в дуже привабливий варіант для тих споживачів, які шукають вигідні багатоядерні рішення з хорошим співвідношенням ціни та продуктивності. І пропозиція вже встигли вдало оцінити багатьох ентузіастів. Популярність процесорів Ryzen стала швидко зростати. Але AMD не зупиняється на досягнутому. Запропоновано рішення для масового ринку, процесори для робочих столів преміального рівня – HEDT (High-End Desktop). Як виявилося, нова мікроархітектура Zen може чудово підійти і для таких продуктів, адже вона пропонує продуктивність, яка добре розкривається в додатках для створення та обробки цифрового контенту. І, більше того, ядра Zen легко збираються в крупні кластери, дозволяючи без обтяжливих витрат на створення процесорів з великим числом обчислювальних ядер.

Використання двох кристалів Zeppelin означає, що в Ryzen Threadripper використовуються чотири CCX-комплексу, у кожному з яких є 8 Мбайт розділеної кеш-пам'яті. У сумі це дає 32-мегабайтний L3-кеш, що рівно вдвічі більше, ніж пропонується в звичайних Ryzen (рис. 1.11).


Рисунок 1.11 – Чотири CCX-комплексу у двох кристалів Zeppelin

Загальний обсяг пам’яті, який теоретично може адресувати Threadripper, досягає 2 Тбайта. Однак через брак у даний момент на ринку небуферізованих модулів DDR4 SDRAM об’ємом більше 16 Гбайт, отримати в системі на базі Ryzen Threadripper понад 128 Гбайт не вийде. Крім того, не слід забувати і про обмеження з боку операційної системи. Більше 128 Гбайт пам’яті, наприклад, не підтримується в Windows 10 Home, а версії Pro і Enterprise не можуть працювати з обсягами пам’яті більше 512 Гбайт.

Для прикладу нижче наводяться результати тестів Ryzen Threadripper 1950X у додатках при роботі з чотирьохканальним масивом DDR4-3200 SDRAM (14-14-14-34) в розподіленому (UMA) і локальному (NUMA) режимах (рис. 1.12).

| | Local (NUMA) | Distributed (UMA) | Лучший режим |
|---|---|---|---|
| | Рендеринг | | |
| Blender 2.78c, сек | 162.5 | 161.9 | UMA |
| Corona 1.3, сек | 71 | 71 | Любой |
| V-Ray 3.57.01, сек | 45 | 46 | NUMA |
| | Обработка фото | | |
| Photoshop CC 2017, сек | 147 | 153.5 | NUMA |
| Lightroom 6, сек | 250.9 | 181.4 | UMA |
| | Обработка видео | | |
| Premiere Pro CC 2017, сек | 241.4 | 210.2 | UMA |
| After Effects CC 2017, сек | 275 | 280 | NUMA |
| | Перекодирование видео | | |
| x264 (r2851), fps | 94.94 | 100.81 | UMA |
| x265 (2.4), fps | 34.78 | 37.07 | UMA |
| | Архивация | | |
| WinRAR 5.50, сек | 138.5 | 125.7 | UMA |
| | Компиляция | | |
| Visual Studio 2017, сек | 208.6 | 210 | NUMA |
| | Шахматы | | |
| Stockfish 8, kNodes/s | 32651 | 33080 | UMA |

Рисунок 1.12 – Результати тестів Ryzen Threadripper 1950X у додатках

Ryzen Threadripper використовують процесорне гніздо Socket sTR4 з 4096 контактами, яке до сих пір ніде і ніколи не зустрічалося. Однак повністю новим його назвати все ж не можна – сам роз’єм у сенсі його конструкційного виконання запозичений у процесорної платформи EPYC і має точно такий же форм-фактор і конструкцію, як і серверний Socket SP3. Швидше за все, між цими роз’ємами є і часткова електрична сумісність, а відмінність полягає лише в тому, що в Socket sTR4 залишається незадіяною частина контактів, яка

відповідає за п’ятий восьмий канали пам’яті, за міжсокетне з’єднання Infinity Fabric і за додаткові лінії PCI Express, які є у EPYC (рис. 1.13).

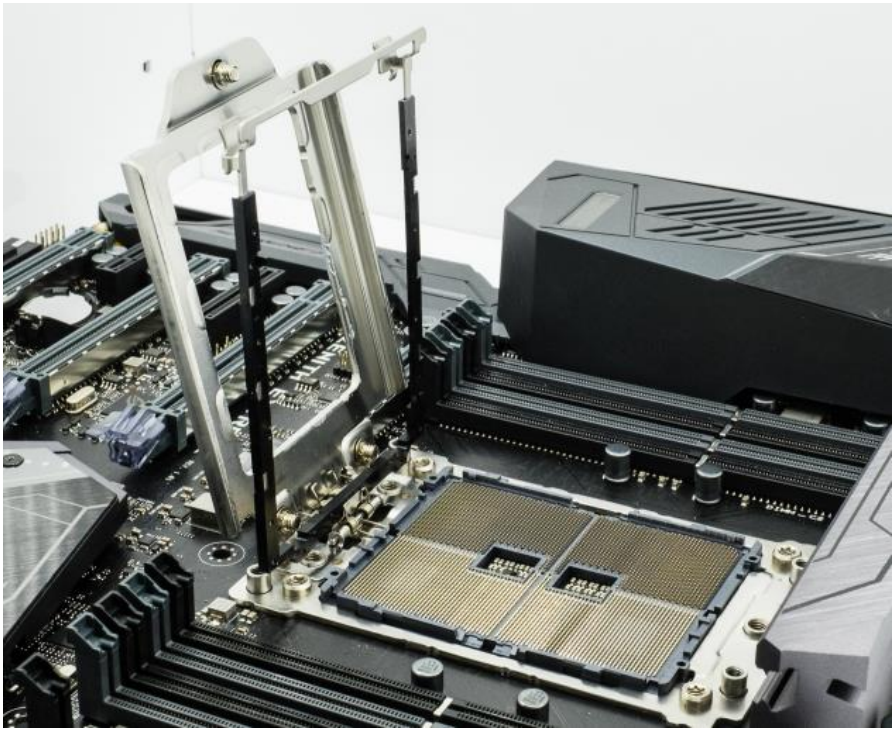


Рисунок 1.13 – Процесорне гніздо Socket sTR4

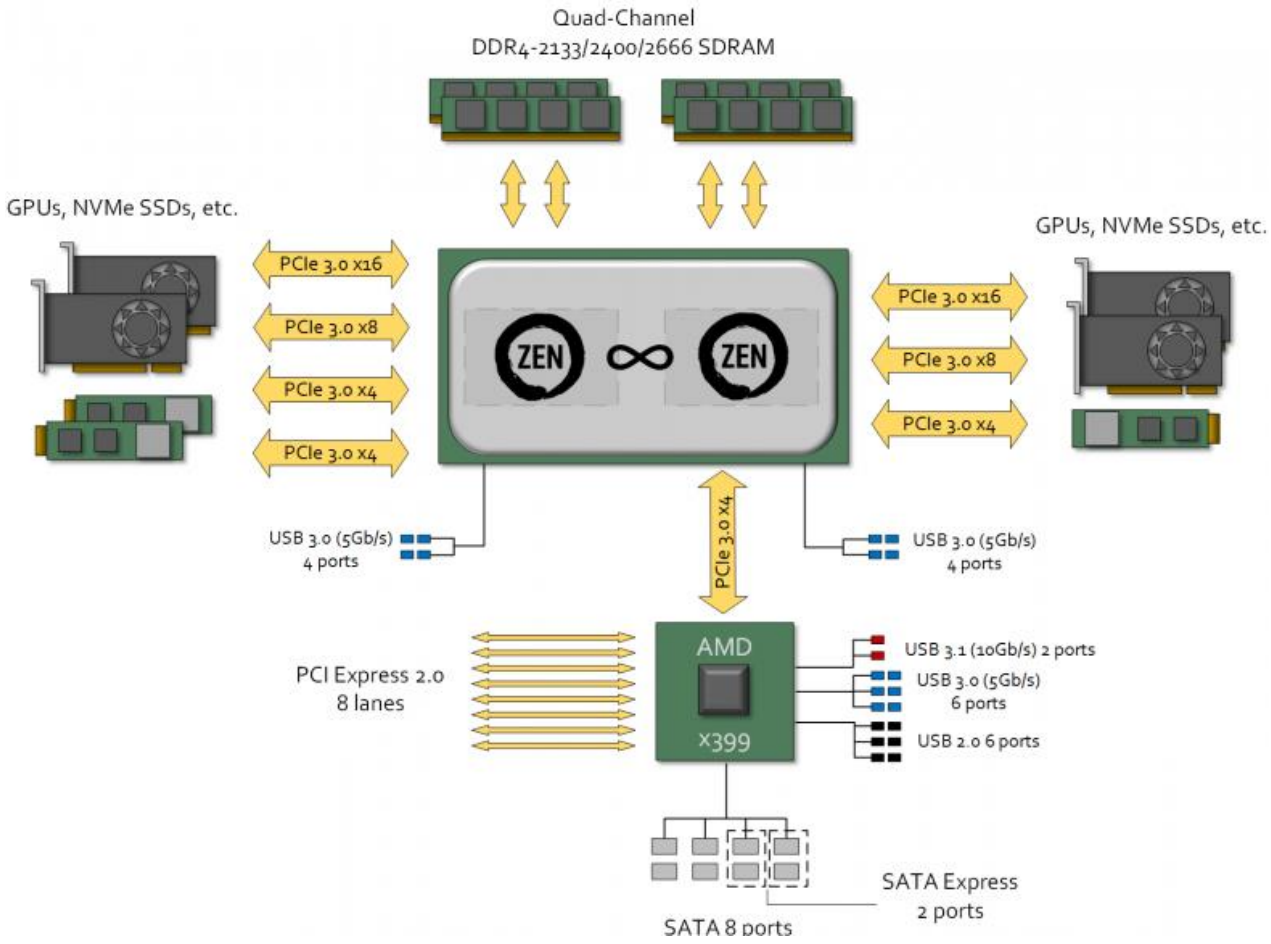



Рисунок 1.14 – Системна логіка X399

Для платформи Socket sTR4 компанія AMD пропонує і власний набір системної логіки, який отримав назву X399 (рис. 1.14).

## 1.5 Постановка завдання

Головні завдання магістерської роботи:

1) проаналізувати апаратне забезпечення кластерних систем та визначити основні чинники впливу на організацію обчислень;

2) дослідити моделі паралельних обчислень;

3) розробити модель обчислень, що дозволяє описати поведінку і взаємодію процесів на рівні вузлів системи при різній організації системи зв'язків вузлів;

4) реалізувати модель обчислень у кластерних системах з багатоядерною архітектурою на основі родини AMD процесорів;

5) дослідити тупикові ситуації при використанні об'єктів комунікації і синхронізації;

6) написати на мові C# пакет багатопоточних програм, які вирішують наступні задачі. Векторно-матричні задачі для реализації:

- MA=MB*MC;
- MA=(MB*MC)*MX;
- A=B*(MC*MX).

де A - вектор, MA,MB,MX - матриці розмірності N (NxN). Елементи матриць можуть бути цілими, з фіксованою точкою, з плаваючою точкою, комплексні.

## Висновки з розділу 1

Усі без винятку дані, що характеризують продуктивність паралельних обчислювальних систем, мають розглядатися критично. Для цього є цілий ряд причин.

1. Характеристики продуктивності, як показник прискорення і показник масштабності завжди пов'язані з конкретним застосуванням паралельних систем - показники справедливі тільки для даного класу задач і дуже умовно можуть бути поширені на інші задач.

2. Показник прискорення деякої програми стосується тільки одного окремого процесора паралельної системи.

3. У SIMD – системах процесорні елементи, як правило, мають значно меншу потужність в порівнянні з MIMD-процесорами, що може дати на порядок, а то й більше, різницю в оцінках швидкості.

4. Завантаження паралельних процесорів – головна складність в MIMD-системах. У SIMD – системах це не так важливо, бо неактивні процесори не можуть бути використані іншими задачами. Незважаючи на це, показник прискорення обчислюється залежно від характеристик завантаження. Якщо SIMD-програма спробує «включити в роботу» непотрібні ПЕ, то дані завантаження, а з ними і показники прискорення, стануть недійсними. Паралельна програма в цьому випадку буде менш ефективною, ніж за результатами тестування.

5. Порівнюючи паралельну систему (наприклад, векторну) з послідовною (скалярною), не доцільно застосовувати два рази один і той же алгоритм (у паралельній і в послідовній версіях).

Також у даному розділі розглянуті апаратні засоби сучасних паралельних комп'ютерних систем (ПКС). Показано, що в них широко використовуються багатоядерні процесори. Виконано аналіз структурної організації багатоядерних процесорів, що показав, що вони відрізняються кількістю ядер (2-16), різними системами зв'язку ядер, використанням кеш-пам'яті різних рівнів і різного обсягу.

# 2 ТЕХНІЧНІ ВИМОГИ ТА ОРГАНІЗАЦІЯ ОБЧИСЛЕНЬ У КЛАСТЕРНИХ СИСТЕМАХ З БАГАТОЯДЕРНОЮ АРХІТЕКТУРОЮ

## 2.1 Організація обчислень у кластерних системах

Ріст продуктивності КС традиційно пов’язаний зі збільшенням кількості вузлів системи. Проте нарощування числа вузлів обмежене із-за підвищеної вимоги до системи зв'язків вузлів при збільшенні вузлів і навантаженні на систему зв'язків. Є другий шлях підвищення продуктивності КС, заснований на збільшенні обчислювальної потужності безпосередньо самого вузла. Такий підхід забезпечується використанням у вузлах потужних процесорів, також реалізацією у вузлах КС принципів паралельної обробки за рахунок використання багатопроцесорних систем. Використання у вузлах багатопроцесорних систем дозволяє перейти в КС до дворівневої паралельної обробки. Перший рівень забезпечується паралельною обробкою на множині вузлів КС, другий - на множині процесорів всередині вузла. Проте, паралелізм на рівні вузла на сьогодні використовується не повною мірою, оскільки вузли існуючих КС, як правило, побудовані на дорогих СМП системах, що включають 2 (рідше 4) процесори.

Поява багатоядерних процесорів відкриває перспективи збільшення потужності вузлів за рахунок ефективнішої реалізації другого рівня паралельної обробки в КС. Проста заміна у вузлі звичайного процесора на двохядерний процесор дозволяє подвоїти обчислювальну потужність КС без зміни її структури. Поява на ринку чотириядерних процесорів і анонсоване в найближчому майбутньому появи процесорів з вісьмома і більше ядрами, відкриває можливості різкого збільшення продуктивності КС при побудові на їх основі кластерних систем з багатоядерною архітектурою (КСБА).

Як видно з рейтингу TOP100, на сьогодні багатоядерні процесори використовуються вже в 62 кластерних системах. Серед них основну масу складають процесори Інтел. Також використовуються процесори від АМД і IBM.

Таким чином, розвиток кластерної архітектури пов'язаний з використанням багатоядерних процесорів.

Це у свою чергу вимагає вдосконалення принципів організації обчислювальних процесів, завданням яких стане ефективне використання обох рівнів паралелізму в КСБА.

Збільшення кількості ядер ставить проблему організації зв'язків між ядрами. Шинна архітектура, яка використовувалася в 2-4 ядрах стане гальмом для ефективного зв'язку ядер і неминучим буде перехід на розподілену архітектуру. Так, наприклад, зв'язування ядер за допомогою топології типу грати дозволять відмовитися від централізованих ресурсів, а також розв'язати проблему надійності при виході з ладу ядер або зв'язків.

Випуск багатоядерних процесорів сьогодні виконується декількома компаніями. Лідерами є компанії Intel і AMD. Разом з ними випуском багатоядерних процесорів для систем серверного рівня займаються також компанії Sun і IBM. Їх особливість - реалізація RISC архітектури.

Багатоядерні AMD процесори. Особливості архітектури чотириядерних процесорів компанії AMD розглянемо на прикладі процесора типу AMD Phenom II 940 (рисунок 2.1).

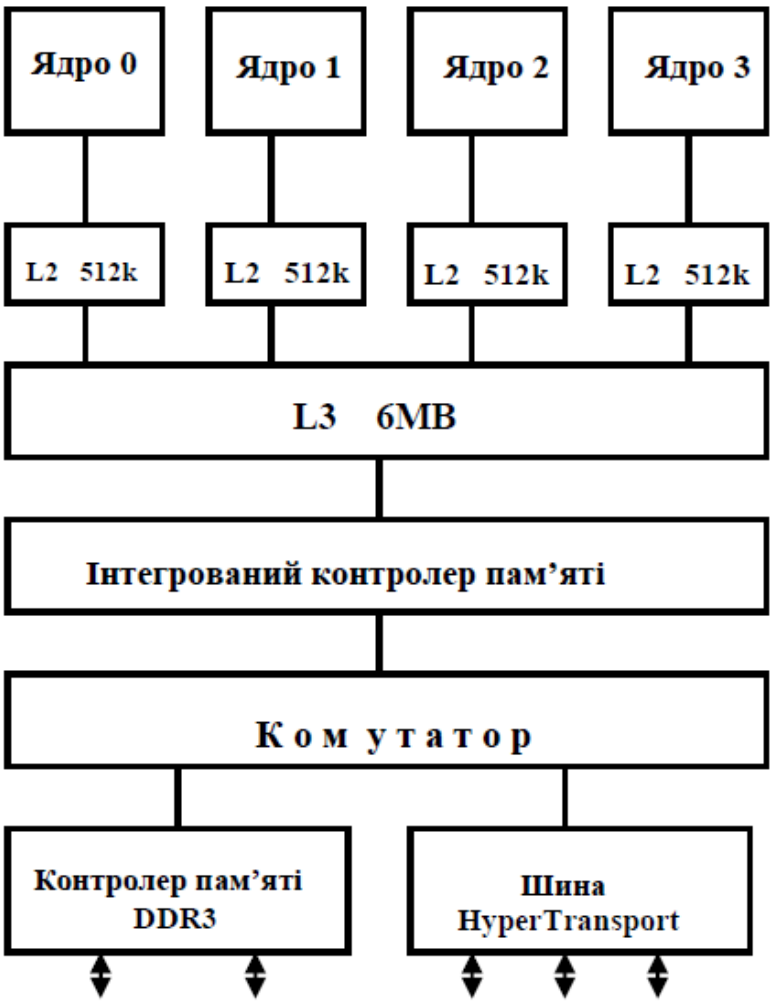



Рисунок 2.1 – Архітектура процесорів типу AMD Phenom II 940

Процесор Phenom побудований на архітектурі Stars, її нова реалізація містить численні покращення, що дозволяє збільшити число інструкцій, що виконуються за такт. В результаті переходу з 65 на 45-нм техпроцес, напруга живлення, яка потрібна для роботи Phenom II, істотно впала. Разом з покращеннями базової мікроархітектури, це дозволило AMD збільшити тактові частоти. Якщо Phenom першого покоління працювали на частоті 2,6 ГГц, то останні Phenom II починаються від 2,8 ГГц і збільшують частоту до 3,0 ГГц.

Phenom першого покоління не могли бути оснащені великим об'ємом кеш-пам'яті L3 із-за великого теплового пакету при 65-нм технологічному процесі (рисунок 2.2). Струми витоку були дуже сильно понижені при переході на 45-нм техпроцес, що дозволило AMD збільшити розмір кеш-пам’яті L3 з 2 до 6 Мбайт. Кеш-пам'ять L2 512 кбайт, індивідуальні для кожного ядра, в новому дизайні не змінилися, те ж саме торкається кеш-пам'яті інструкцій і даних рівня L1 по 64 кбайт.

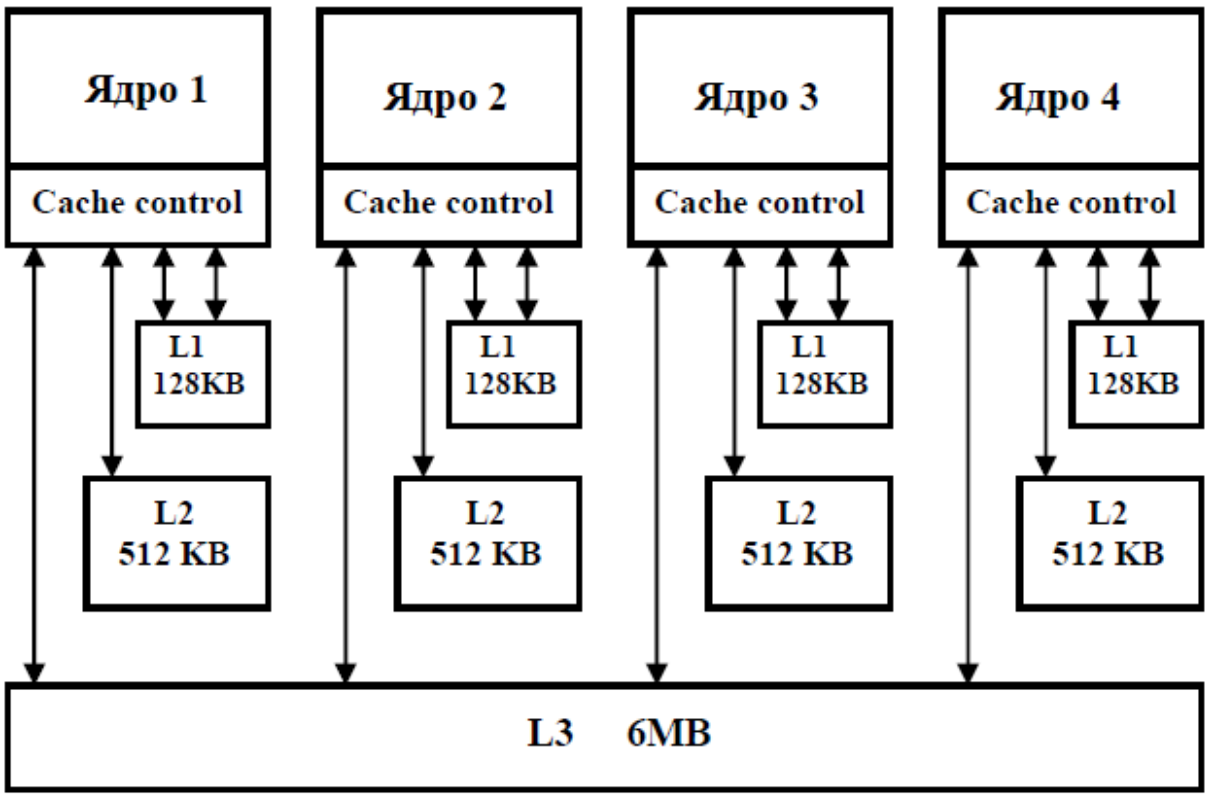



Рисунок 2.2 – Структура кеш пам’яті в процесорі типу AMD Phenom II 940

Ще одне покращення продуктивності Phenom II - підтримка пам'яті DDR3.

Організація обчислювальних процесів в КСБА повинна забезпечити реалізацію паралелізму на обох рівнях кластерної систем. На першому рівні - забезпечити ефективну взаємодію вузлів за рахунок мінімальних втрат часу, пов'язаних з передачею даних між вузлами. На другому рівні – організацію

обчислювальних процесів у вузлі, пов'язану з обчисленнями в кожному ядрі процесора і взаємодією процесів безпосередньо у вузлі системи.

Для розробки програмного забезпечення для КСБА необхідно використовувати спеціальні програмні засоби, які сьогодні вбудовані безпосередньо в мови програмування або представлені спеціальними бібліотеками [6-12].

Основою проблемою при розробці програмного забезпечення, заснованого на використанні механізму процесів, являється організація взаємодії процесів. Взаємодія процесів (process interaction) має різноманітні форми, але може бути зведене до двох видів взаємодії :

- передача даних між процесами (process communication)
- синхронізації процесів (process synchronization).

Коректна поведінка паралельної програми критично залежить від організації взаємодії процесів.

Передача даних пов'язана з пересилкою інформації між процесами. При цьому процеси можуть передавати дані з одного процесу в іншій або обмінюватися даними. Синхронізація - узгодження процесами своєї поведінки, яка зводиться до того, що процеси обмінюються сигналами для блокування і розблокування свого стану. Обидва види взаємодії пов'язано між собою, оскільки деякі форми передачі даних вимагають синхронізації, а синхронізація може бути пов'язаною з передачею даних.

Взаємодія процесів ґрунтується на двох моделях паралельного програмування [12]:

- модель, що базується на загальних змінних;
- модель, що базується на передачі повідомлень [22].

## 2.2 Вимоги до кластерної системи

Вимоги до системи загалом:

1) вимоги до структури і функціонування системи:
   розглядається паралельна комп’ютерна система (ПКМ) з багатоядерним процесором. Кількість ядер – 4. Структура ПКМ приведена на рисунку 2.3.
2) вимоги до чисельності і кваліфікації персоналу, що обслуговує систему і режиму його роботи:
   у більшості випадків користувачу не потрібно втручатись у процес розподілення системних ресурсів для роботи програми, але можливі випадки, в яких було б доречно, наприклад, чітко зазначити номера ядер, на яких буде виконуватись програма, для явного розподілення

навантаження, якщо система пріорітетів не дає потрібних результатів. У загальному випадку вимоги до кваліфікації персоналу не відрізняються від вимог до користувачів використовуваної операційної системи Windows 7-10.

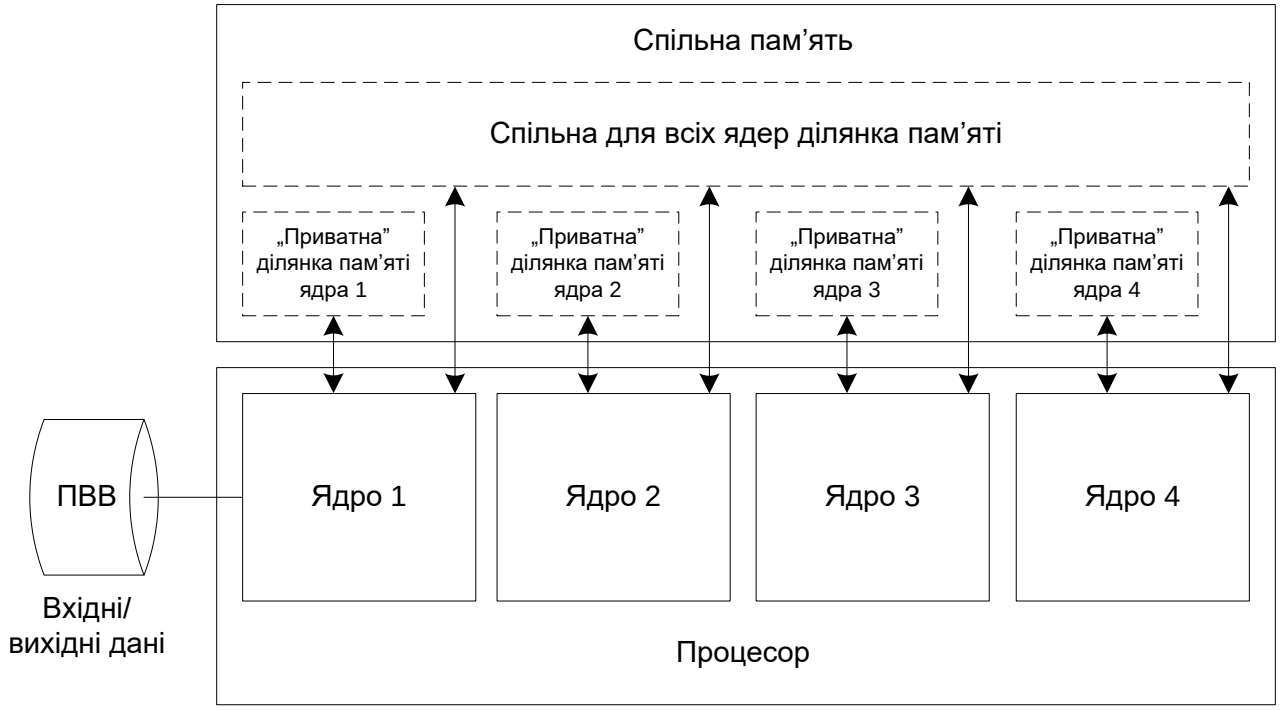



Рисунок 2.3 – Структурна схема багатоядерної системи

3) вимоги до захисту інформації від несанкціонованого доступу:

усі вхідні/вихідні дані є незахищеними, шифрування і підписування не відбувається.

4) вимоги до збереження інформації при аваріях:

при аварії можливо відновити частину корисної інформації з повного дампу оперативної пам'яті при умові, що він виконався. Проміжних записів інформації на протязі роботи програми не виконується.

5) вимоги до стандартизації й уніфікації:

структура програми дозволяє легко додавати задачі.

Вимоги до функцій (задач), що виконує система:

1) вимоги до режимів функціонування системи:

забезпечений вибір задачі, типу даних, розмірності вхідних матриць, типу виведення результатів, типу заповнення матриць (діапазон випадкових чисел).

2) вимоги до діагностування системи:

можливе використання програмних відлагоджувачів для спостереження за роботою програми.

3) перспективи розвитку, модернізації системи:

об’єднання багатоядерних систем в кластерну систему з мережевою комутацією дозволить збільшити продуктивність та маштабованість системи.

Вимоги для програмного забезпечення системи:

1) перелік покупних програмних засобів:
   Windows 7-10 або Server.
2) до якості програмних засобів, а також до способів її забезпечення і контролю:

роботу програми можна перервати без втрати стабільності роботи системи, але з втратою поточних результатів.

## 2.3 Моделі паралельних та розподілених обчислень

Розробка програмного забезпечення для паралельних і розподілених систем має ряд особливостей, головна з яких пов'язана з необхідністю врахування архітектури системи, яка ускладнюється при використанні великого числа процесорів в паралельних системах і рості вузлів в розподілених системах. Крім того, на розробку програми робить істотний вплив і вибрана модель програмування, яка у свою чергу пов'язана з архітектурою системи.

Інтерфейсом між архітектурою і моделлю паралельного програмування може служити модель паралельних обчислень (рисунок 2.4). Її призначення - відображення особливостей архітектури комп'ютерної системи і облік вибраної моделі паралельного програмування. Модель обчислень для розподілених систем повинна відображати взаємодію процесів, яка представляється в моделях програмування [18, 19]. Також модель обчислень дозволяє оцінити наскільки ефективна реалізація програми на вибраній архітектурі. При цьому ефективність визначається за такими критеріями як час виконання програми, завантаження процесорів, передача даних між процесами, наявність тупикових ситуацій та ін.

Крім того, математична модель паралельних обчислень повинна забезпечити ефективну реалізацію програми, що базується на моделі обчислень і на моделі паралельного програмування. Враховуючи складність програм для паралельних і розподілених систем, модель паралельних обчислень повинна дозволити виконати попередній аналіз коректності проведення паралельних процесів і в першу чергу – організації їх взаємодії, що є причиною тупикових ситуацій.

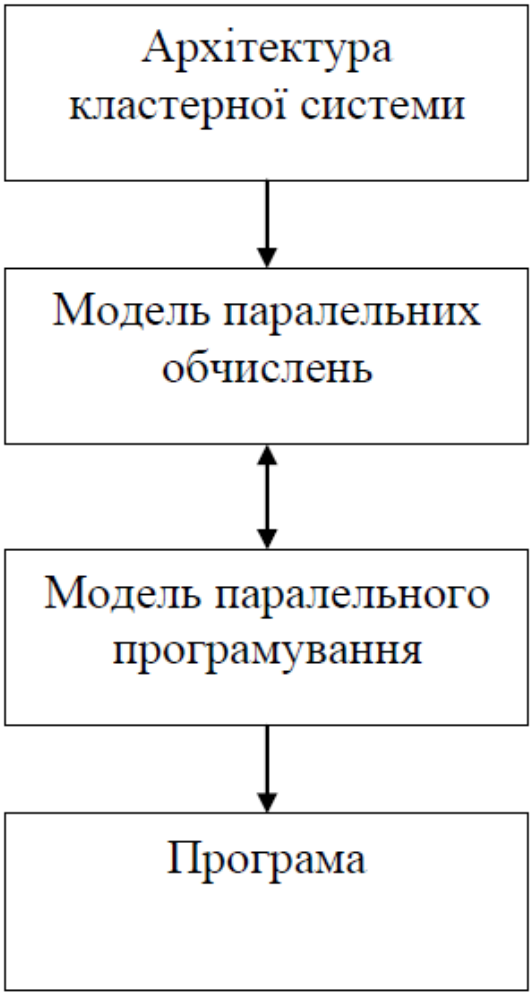



Рисунок 2.4 – Структура взаємозв’язків моделей

Існує досить велика кількість моделей, що дозволяють описати обчислення в паралельних системах (моделі паралельних обчислень ). Це моделі Р. Мильнера (CSS модель), Ч.Хоара (CSP модель), мережі Петрі, мережі на основі кінцевих автоматів та ін. [20]. Різноманітність модель пов'язана з різною спрямованістю цих моделей.

Можливим способом моделювання паралельних обчислень є використання математичних моделей і методів дослідження дискретних систем, розроблених у рамках теорії мереж Петрі [20]. При такому підході в якості моделі програми може бути використана мережа Петрі, що представляється розміченим орієнтованим графом.

При використанні мереж Петрі для опису паралельних програм переходи зазвичай відповідають діям (процесам), а місця - умовам (виділенню або звільненню ресурсів). Розмітка місць в цьому випадку інтерпретується як кількість наявних (нерозподілених) одиниць ресурсу.

У рамках теорії мереж Петрі розроблені методи, що дозволяють для довільної мережі визначити, чи є мережа обмеженою або живою, перевірити досяжність будь-якого переходу або розмітки мережі.

Як результат, ці методи дозволяють визначити наявність безвиході в мережі.

Мережі Петрі в першу чергу спрямовані на моделювання процесів, а не на реалізацію алгоритмів. При цьому у мережах Петрі відсутнє строге поняття процесу для виконання на процесорі. Крім того, реалізація моделі обчислень, побудованої на мережах Петрі, може викликати певні складнощі для користувача.

Останнім часом для опису паралельних обчислень отримали розвиток автоматні моделі [21]. Операції алгебри на множині автоматів дозволяють впровадити нові методи аналізу і синтезу паралельних алгоритмів. Ці методи відпрацьовані в практиці проектування цифрових систем. Крім того, проектування на базі автоматів паралельних систем аж до етапу реалізації може бути єдиним як для програмних, так і для апаратних систем.

Автоматні моделі відрізняються від мереж Петрі більшою потужністю, простотою розуміння, структурною ясністю, великою і перевіреною часом теорією. Їх реалізація зажадає від моделі природності в описі і високій ефективності наступного виконання. Потрібна також наявність формальних процедур аналізу створених алгоритмів.

Алгебра процесів Ч.Хоара. Модель CSP (Communicating Sequential Processes) була запропонована Ч.Хоаром. У теорії Ч.Хоара поведінку будь-якого об'єкту можна описати через процес - математичну абстракцію взаємодії системи і її оточення. Кожен об'єкт характеризується набором станів (подій), який задається через алфавіт об'єкту.

Алфавіт – множина імен подій, вибраних для опису конкретного об'єкту. Для кожного об'єкту існує свій, заздалегідь вибраний алфавіт, який залежить від властивостей об'єкту. Запропонована нотація процесу ґрунтується на тому, що конкретна подія відбувається миттєво (елементарна дія), тобто не має протяжності в часі. Протяжність розглядається як пара подій, які визначають початок і завершення дії. Тривалість дії - інтервал часу між настанням події - початок і настанням події - завершення. Також характерною особливістю даної нотації процесу є абстрагування від прив'язки подій до часу.

Процес означає поведінку об'єкту і може бути описаний в термінах обмеженого набору подій, вибраних в якості його алфавіту.

Якщо x – деяка подія, а P - процес, то описати об'єкт, який спочатку бере участь в події x, а потім поводиться точно як процес P можна таким чином: $a(x \rightarrow P)=aP$, якщо x aP.

Будь-який процес може бути представлений у вигляді: ( x: B → F(x) ), де функція ставить у відповідність множині символів x множину процесів B.

Процес, поведінка якого задається рекурсивним співвідношенням, може бути представлений як: ( x: B → F(x, yX.( x: B → F(x, X)) ) ).

Запропонована нотація дозволяє розглядати будь-який процес як функцію F з областю визначення B, яка виділяє множину подій, які служать як початкові події процесу.

Поведінка об'єкту (процес) – це послідовна зміна станів, яка задається операцією префіксації (→). При описі процесу допускаються також рекурсивність і вибір. Обмежений набір операцій зміни станів не дозволяє описати поведінку складних об'єктів. Проте якщо виходити з того, що КСБА орієнтовані на складні обчислювальні задачі, засновані на крупнозернистому паралелізмі, то моделі Хоара досить для того, щоб промоделювати обчислення в таких системах. Хоча саме недостатність засобів для опису поведінки об'єкту виділяють як основний недолік CSP теорії .

Таким чином, моделлю паралельних обчислень в теорії Ч.Хоара є паралельна поведінка набору об'єктів: ( P1 || P2 || . . .|| Pn ), а поведінка і-го об'єкту системи як процес Pi з послідовною зміною станів Ci: αPi = ( C1 → C2 →C4 → C3 →C5 → C6 ) на підставі визначеного для об'єкту Pi алфавіту: αPi={C1, C2, C3, C4, C5, C6 }.

Взаємодія процесів в CSP теорії ґрунтується на моделі передачі повідомлень. При цьому в алгебрі процесів Хоара визначено поняття логічного каналу, за допомогою якого один процес може переслати дані іншому процесу.

Наявність розвинених засобів для опису взаємодії процесів виділяється як важлива перевага CSP теорії. При цьому алгебра процесів Хоара представляє розвинений математичний апарат не лише для опису поведінки процесів і їх взаємодії, але дозволяє виконати з його допомогою аналіз коректності самої моделі і, зокрема, представляє інструмент для виявлення тупикових ситуацій ще на етапі проектування програмного забезпечення.

Ще один аргумент на користь застосування CSP теорії для опису моделей паралельних обчислень для КСБА базується на тому, що концепція послідовних взаємодіючих процесів лежить в основі більшості сучасних мов паралельного програмування. Це дозволить спростити реалізацію математичної моделі за допомогою вибору відповідної мови паралельного програмування [22].

**Висновки з розділу 2**

1. Розвиток кластерної архітектури пов'язаний з використанням багатоядерних процесорів AMD та Intel.

2. За допомогою CSP теорії можлива побудова математичної моделі обчислень для КСБА з різною структурною організацією, то розробникові програмного забезпечення для КСБА стане доступним ефективний механізм

створення програмного забезпечення, що дозволяє скоротити час проектування програмного продукту і підвищити його надійність.

3. Система володіє двома рівнями паралельної обробки – між вузлами і на рівні кожного вузла і вимагає розробки моделей паралельних обчислень, що забезпечують опис поведінки процесів на обох рівнях.

# 3 СХЕМИ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ ЗАДАЧ ТА КРИТЕРІЇ ЕФЕКТИВНОСТІ РОЗВ'ЯЗАННЯ ЗАДАЧ

## 3.1 Схеми алгоритмів

Паралельні алгоритми [23-40] використовуються для розв'язання таких задач: множення матриці на матрицю, задача Діріхле, розв'язання систем лінійних алгебраїчних рівнянь (СЛАР) методом Гауса і методом простої ітерації та інші. В простому варіанті сіткової задачі (задача Діріхле) крок сітки в просторі обчислень однаковий і не змінюється в процесі обчислень. При динамічній зміні кроку сітки треба було б розв'язувати задачу паралельного програмування, як перебалансування обчислювального простору між комп'ютерами, для вирівнювання обчислювального навантаження комп'ютерів.

Особливості алгоритмів:

1. Задачі відносяться до задач, що розпаралелюються грубозернистими методами.

2. Для представлення алгоритмів використовується SPMD-модель обчислень (розпаралелення за даними).

3. Однорідне розподілення даних по комп'ютерах - основа для гарного балансу часу обчислення і часу, затрачуваного на взаємодії віток паралельної програми. Такий розподіл використовується з метою забезпечення рівності обсягів частин даних, що розподіляються, і відповідності нумерації частин даних, що розподіляються, нумерації комп'ютерів у системі.

4. Вихідними даними розглянутих алгоритмів є матриці, вектори і 2D (двовимірний) простір обчислень.

5 У розглянутих алгоритмах застосовуються такі способи однорідного розподілу даних: горизонтальними смугами, вертикальними смугами і циклічними горизонтальними смугами.

При розподілі горизонтальними смугами матриця, вектор або 2D простір "розрізається" на смуги по рядках. Нехай M - кількість рядків матриці, кількість елементів вектора або кількість рядків вузлів 2D простору, P - кількість віртуальних комп'ютерів у системі, C1 = M /P - ціла частина від ділення, C2 = M /P - дробова частина від ділення. Дані розрізаються на P смуг. Перші (P-C2) смуг мають по C1 рядків, а інші C2 смуги мають по C1+1 рядків.

Смуги даних розподіляються по комп'ютерах таким чином. Перша смуга поміщається в комп'ютер з номером 0, друга смуга - у комп'ютер 1, і т.д. Такий розподіл смуг по комп'ютерах враховується в паралельному алгоритмі. Розподіл вертикальними смугами аналогічний попередньому, тільки в розподілі беруть участь стовпці матриці або стовпці вузлів 2D простору. При розподілі циклічними горизонтальними смугами дані розрізаються на кількість смуг значно більшу, від кількості комп'ютерів. І найчастіше смуга складається з одного рядка. Перша смуга завантажується в комп'ютер 0, друга - у комп'ютер 1, і т.д., потім, P-1-а смуга знову в комп'ютер 0, P-а смуга в комп'ютер 1, і т.д.

Проте, тільки однорідність розподілу даних ще недостатня для ефективного виконання алгоритму. Ефективність алгоритмів залежить і від способу розподілу даних. Різний спосіб представлення даних веде, відповідно, і до різної організації алгоритмів, що обробляють ці дані.

Точні значення ефективності конкретного паралельного алгоритму можуть бути визначені на конкретній обчислювальній системі на деякому наборі даних. Тобто, ефективність паралельних алгоритмів залежить, по-перше, від обчислювальної системи, на якій виконується задача, а, по-друге, від структури самих алгоритмів. Вона визначається як відношення часу реалізації паралельного алгоритму задачі до часу реалізації послідовного алгоритму цієї ж задачі. Ефективність можна вимірювати і співвідношенням між часом, витраченим на обмін даними між процесами, і загальним часом обчислень. Зауважимо, що ефективність алгоритмів, що використовують глобальний обмін даними, знижується з збільшенням кількості паралельних віток. Тобто з збільшенням кількості комп'ютерів у системі, швидкість виконання глобальної операції обміну буде падати. До таких задач можна віднести, наприклад, задачу розв’язання СЛАР ітераційними методами. Ефективність алгоритмів, у яких обмін даними здійснюється тільки локально, буде незмінною з збільшенням кількості паралельних віток.

## 3.2 Алгоритми перемноження матриці на матрицю і їх реалізація на структурах типу: кільцева, 2D (решітка), 3D (куб)

Множення матриці на вектор і матриці на матрицю є базовими макрооперaціями для багатьох задач лінійної алгебри, наприклад ітераційних методів розв’язання систем лінійних рівнянь і т.п. Тому приведені алгоритми тут можна розглядати як фрагменти в алгоритмах цих методів. Розглянемо три алгоритми множення матриці на матрицю. Розмаїтість варіантів алгоритмів виникає із-за розмаїтості обчислювальних систем і розмаїтості розмірів задач. Розглядаються і різні варіанти завантаження даних у систему: завантаження

даних через один комп'ютер; і завантаження даних безпосередньо кожним комп'ютером з дискової пам'яті. Якщо завантаження даних здійснюється через один комп'ютер, то дані зчитуються цим комп'ютером з дискової пам'яті, розрізаються на частини, які розсилаються іншим комп'ютерам. Але дані можуть бути підготовлені і заздалегідь, тобто заздалегідь розрізані вроздріб і кожна частина записана на диск у виді окремого файлу зі своїм ім'ям; потім кожен комп'ютер безпосередньо зчитує з диска, призначений для нього файл.

Алгоритм 1 – Перемноження матриці на матрицю на кільцевій структурі

Задано дві вихідні матриці A і B. Обчислюється добуток C = A x B, де A - матриця n1 x n2, і B - матриця n2 x n3. Матриця результатів C має розмір n1 x n3. Вихідні матриці попередньо розрізані на смуги, смуги записані на дискову пам'ять окремими файлами зі своїми іменами і доступні всім комп'ютерам. Матриця результатів повертається в нульовий процес.

Реалізація алгоритму виконується на кільці з p1 комп'ютерів. Матриці розрізані як показане на рис. 3.1: матриця A розрізана на p1 горизонтальних смуг, матриця B розрізана на p1 вертикальних смуг, і матриця результату C розрізана на p1 смуги. Тут передбачається, що в пам'ять кожного комп'ютера завантажується і може знаходитися тільки одна смуга матриці A і одна смуга матриці B.

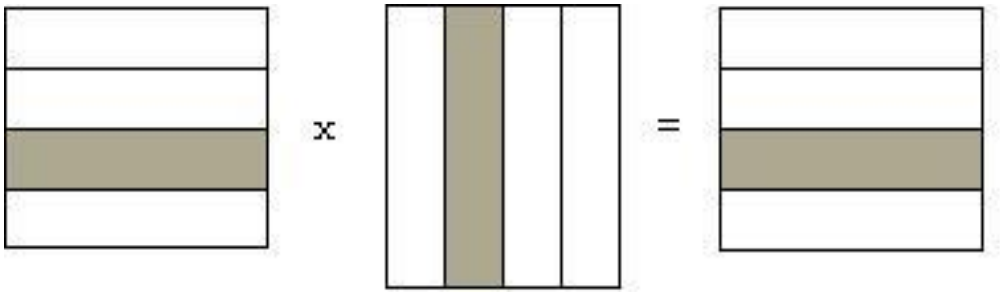



Рисунок 3.1 – Розрізування даних для паралельного алгоритму добутку двох матриць при обчисленні на кільці комп'ютерів. Виділені смуги розташовані в одному комп'ютері

Оскільки за умовою в комп'ютерах знаходиться по одній смузі матриць, то смуги матриці B (або смуги матриці A) необхідно "прокрутити" по кільцю комп'ютерів повз смуги матриці A (матриці B). Кожний зсув смуг уздовж кільця і відповідна операція множення наведена на рис.3.2 у виді окремого кроку. На кожному з таких кроків обчислюється тільки частина смуги.

Процес i обчислює на j-м кроці добуток i-й горизонтальної смуги матриці A j-ї вертикальної смуги матриці B, добуток отриманий у під матриці (i,j) матриці C.

Обчислення відбувається в такій послідовності.

1. Кожен комп'ютер зчитує з дискової пам’яті відповідну йому смугу матриці A. Нульова смуга повинна зчитуватися нульовим комп'ютером, перша смуга - першим комп'ютером і т.д., остання смуга - зчитується останнім комп'ютером. На рис. 7.2 смуги матриці A і B пронумеровані.

2. Кожен комп'ютер зчитує з дискової пам’яті відповідну йому смугу матриці B. У даному випадку нульова смуга повинна зчитуватися нульовим комп'ютером, перша смуга - першим комп'ютером і т.д., остання смуга - зчитується останнім комп'ютером.

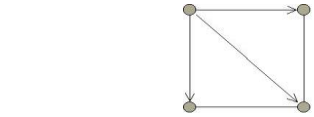
3. Обчислювальний крок 1. Кожен процес обчислює одну підматрицю добутку. Вертикальні смуги матриці B зсуваються уздовж кільця комп'ютерів.

4. Обчислювальний крок 2. Кожен процес обчислює одну підматрицю добутку. Вертикальні смуги матриці B зсуваються уздовж кільця комп'ютерів. І т.д.

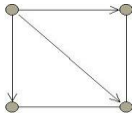
5. Обчислювальний крок p1-1. Кожен процес обчислює одну підматрицю добутку. Вертикальні смуги матриці B зсуваються уздовж кільця комп'ютерів.

6. Обчислювальний крок p1. Кожен процес обчислює одну підматрицю добутку. Вертикальні смуги матриці B зсуваються уздовж кільця комп'ютерів.

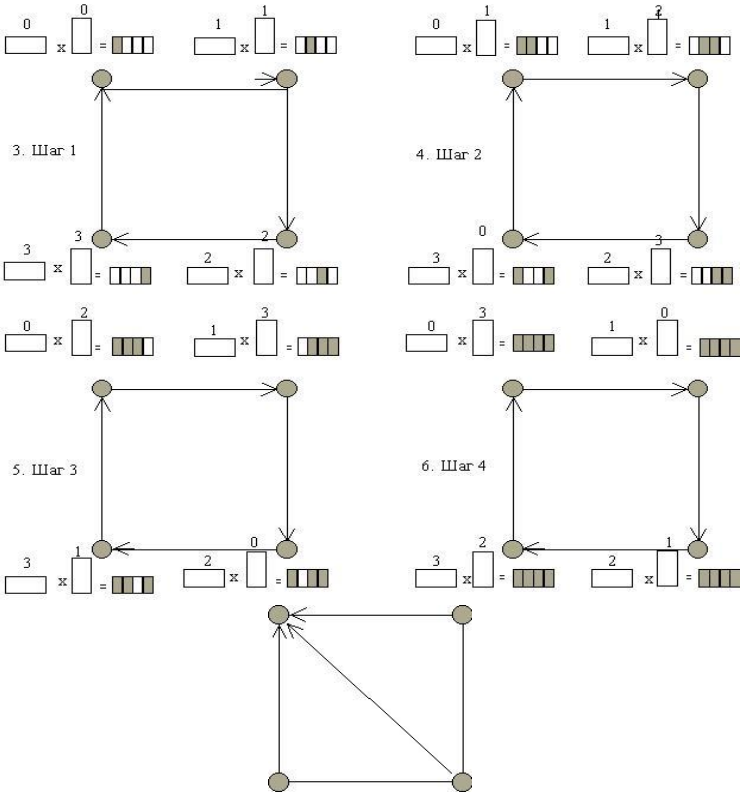
7. Матриця C збирається в нульовому комп'ютері.



1. scatter A



2. scatter B




7. Збір результатів у C

Рисунок 3.2 – Стадії обчислень добутку матриць у кільці комп'ютерів.

Якщо "прокручувати" вертикальні смуги матриці B, то матриця C буде розподілена горизонтальними смугами, а якщо "прокручувати" горизонтальні смуги матриці A, то матриця C буде розподілена вертикальними смугами.

Алгоритм характерний тим, що після кожного кроку обчислень здійснюється обмін даними. Нехай tu, ts, і tp час операцій, відповідно, множення, додавання і пересилання одного числа в сусідній комп'ютер. Тоді сумарний час операцій множень дорівнює:

$$U = (n1*n2)*(n3*n2)*tu,$$

сумарний час операцій додавань дорівнює:

$$S = (n1*n2)*(n3*(n2-1))*ts,$$

сумарний час операцій пересилань даних по всіх комп'ютерах дорівнює:

$$P = (n3*n2)*(p1-1)*tp.$$

Загальний час обчислень визначимо як:

$$T = (U+S+P)/p1$$

Відношення часу "обчислень без обмінів" до загального часу обчислень є величина:

$$K = (U+S)/(U+S+P).$$

Якщо час передачі даних великий в порівнянні з часом обчислень, або канали передачі повільні, то ефективність алгоритму буде не висока. Тут не враховується час початкового завантаження і вивантаження даних у пам'ять системи. У смугах матриць може бути різна кількість рядків, а різниця в кількості рядків між смугами – 1. При великих матрицях цим можна знехтувати.

При достатніх ресурсах пам'яті в системі краще використовувати алгоритм, у якому мінімізовані обміни між комп'ютерами в процесі обчислень. Це досягається за рахунок дублювання деяких даних у пам'яті комп'ютерів. У наступних двох алгоритмах використовується цей підхід.

Алгоритм 2 – Перемноження матриці на матрицю на 2D решітці

Обчислюється добуток C = A x B, де A - матриця n1 x n2, і B - матриця n2 x n3. Матриця результатів C має розмір n1 x n3. Вихідні матриці спочатку доступні на нульовому процесі, і матриця результатів повертається в нульовий процес.

Паралельне виконання алгоритму здійснюється на двовимірній (2D) решітці комп'ютерів розміром p1 x p2. Матриці розрізані, як показано на рис. 3.3: матриця A розрізана на p1 горизонтальних смуг, матриця B розрізана на p2

вертикальних смуг, і матриця результату C розрізана на p1 x p2 підматриці (або субматриці).

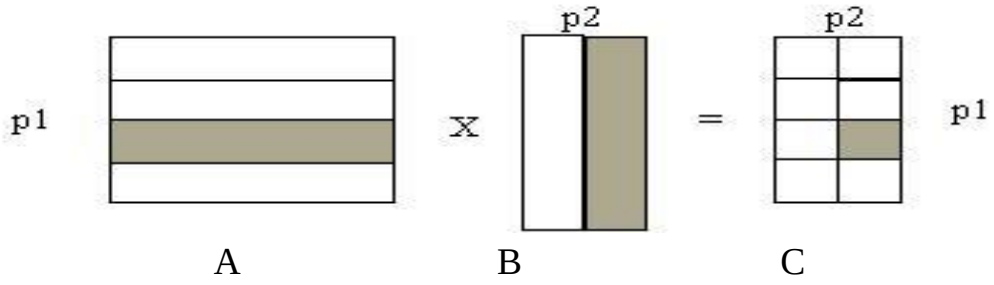



Рисунок 3.3 – Розрізування даних для паралельного алгоритму добутку двох матриць при обчисленні на 2D решітці комп'ютерів. Виділені дані розташовані в одному комп'ютері

Кожен комп'ютер (i,j) обчислює добуток і-й горизонтальної смуги матриці A і j-й вертикальної смуги матриці B, добуток отримується у підматриці (i,j) матриці C.

Послідовність стадій обчислення наведена на рис.3.4:

1. Матриця A розподіляється по горизонтальних смугах уздовж координати (x,0).

2. Матриця B розподіляється по вертикальних смугах уздовж координати (0,y).

3. Смуги A поширюються у вимірі y.

4. Смуги B поширюються у вимірі x.

5. Кожен процес обчислює одну підматрицю добутку.

7. Матриця C збирається з (x,y) площини.

Здійснювати пересилання між комп'ютерами під час обчислень не потрібно, тому що всі смуги матриці A перетинаються з усіма смугами матриці B у пам'яті комп'ютерів системи.

Цей алгоритм ефективніший від попереднього, тому що непродуктивний час пересилань даних здійснюється тільки при завантаженні даних у пам'ять комп'ютерів і при їхньому вивантаженні, і обміни даними в процесі обчислень відсутні. Оскільки час обмінів рівний нулеві, а час завантаження і вивантаження тут не враховується, то загальний час обчислень дорівнює:

$$T = (U+S)/(p1*p2)$$

А відношення часу "обчислень без обмінів" до загального часу обчислень є величина:

$K = (U+S)/(U+S)=1.$

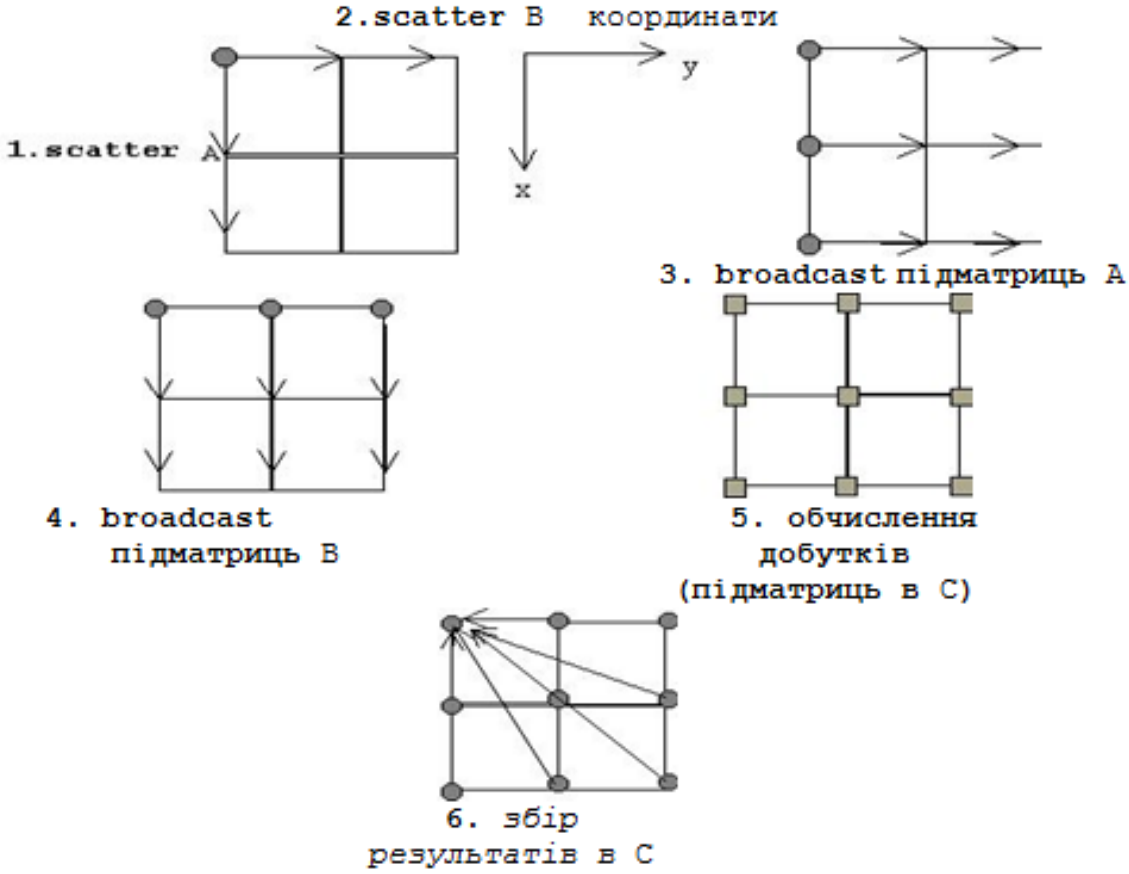



Рисунок 3.4 – Стадії обчислення добутку матриць у 2D паралельному алгоритмі

Алгоритм 3 – Перемноження матриці на просторовій сітці комп’ютерів

Для великих матриць, час обчислення добутків може бути зменшений шляхом застосуванням алгоритму, що здійснює обчислення на 3-мірній (просторовій) сітці комп'ютерів.

У приведеному нижче алгоритмі відображаються основні дані обсягом n1 x n2 + n2 x n3 + n1 x n3 на об'ємну сітку комп'ютерів розміром p1 x p2 x p3. Матриці розрізані, як показано на рис. 3.5: Матриця A розрізана на p1 x p2 субматриці, матриця B розрізана на p2 x p3 субматриці, і матриця C розрізана на p1 x p3 субматриці. Комп'ютер (i,j,k) обчислює добуток субматриці (i,j) матриці A і субматриці (j,k) матриці B. Субматриця (i,k) матриці C виходить підсумовуванням проміжних результатів, обчислених у комп'ютерах (i,j,k), j = 0,...,p2-1.

Послідовність стадій обчислення наведена на рис. 7.6.

1. Субматриці A розподіляються в (x,y,0) площині.
2. Субматриці B розподіляються в (0,y,z) площині.
3. Субматриці A поширюються у вимірі z.
4. Субматриці B поширюються у вимірі x.
5. Кожен процес обчислює одну субматрицю.
6. Проміжні результати редукується у вимірі y.
7. Матриця C збирається з (x,0,z) площини.

Алгоритм подібний до попереднього, але додатково розрізаються ще смуги матриць, і ці розрізані смуги розподіляються в третьому вимірі у. У даному випадку в кожному комп'ютері будуть перемножуватися тільки частини векторів рядків матриці А і частини стовпців матриці В. В результаті буде тільки часткова сума для кожного елемента результуючої матриці С. Операція підсумовування уздовж координати у цих отриманих часткових сум для результуючих елементів і завершує обчислення матриці С.

Загальний час обчислень у цьому алгоритмі дорівнює:

$$T = (U+S)/(p1*p2*p3)$$

А відношення часу "обчислень без обмінів" до загального часу обчислень є величина:

$$K = (U+S)/(U+S)=1.$$

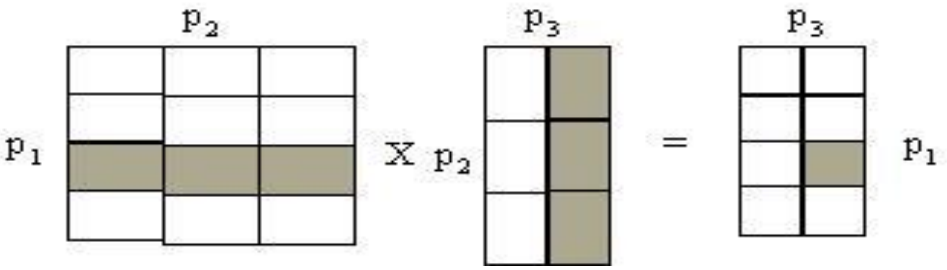



Рисунок 3.5 – Розрізування даних для паралельного алгоритму добутку двох матриць при обчисленні на просторовій сітці комп'ютерів

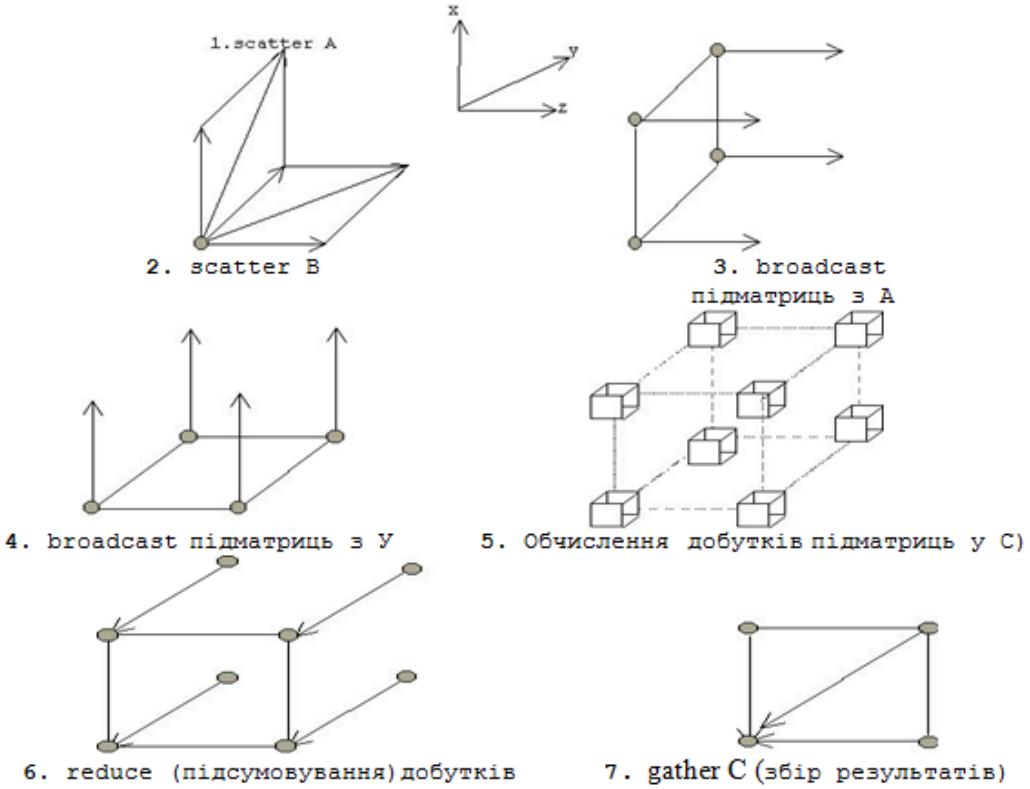



Рисунок 3.6 – Стадії обчислень у 3D паралельному алгоритмі добутку матриць.

### 3.3 Критерії ефективності розв'язання задачі

Відповідно до постановки задачі, обчислення виконуються на паралельній обчислювальній системі з локальною пам'яттю, вузли якої можуть містити спільну для певної невеликої кількості процесорів пам'ять [1, 23-40].

Нехай для задачі виділено K вузлів, при чому i-тий вузол в свою чергу містить n процесорів. Таким чином загальна кількість ядер, які доступні задачі в паралельній обчислювальній системі визначається як $N = \sum_{i=1}^{K} n_i$. Для спрощення вважаємо, що на виділених вузлах виконуються лише задача користувача, а також складові частини операційної системи та система керування паралельними обчисленнями. При цьому обсяги обчислень, що виконуються двома різними процесорами за одиницю часу, можуть відрізнятись.

Позначимо час виконання паралельної реалізації алгоритму, що використовується у задачі, що розглядається, на заданій обчислювальній системі як T. Час виконання вимірюється від моменту активізації задачі (моменту початку обчислень) до моменту отримання кінцевого результату. Але час активного використання i-го процесора задачею складає ti, таке що:

$$t_i \leq T . \tag{3.1}$$

через додаткові витрати часу, які специфічні для паралельних реалізацій: необхідність комунікації з іншими процесами, отримання доступу до спільного ресурсу тощо.

Час, необхідний для виконання тієї самої реалізації алгоритму в послідовному режимі на даній обчислювальній системі дорівнює сумі часу обчислень кожним процесором окремо $\sum_{j=1}^{N} t_j$. Тоді коефіцієнт прискорення визначається відношенням оціненого часу обчислень в послідовному режимі до часу обчислень в паралельному режимі:

$$K_{\text{П}} = \frac{\sum_{j=1}^{N} t_j}{T}. \tag{3.2}$$

Якщо врахувати обмеження (3.1), можна помітити, що коефіцієнт прискорення не може перевищувати кількості наявних процесорів:

$$K_{\text{П}} = \frac{\sum_{j=1}^{N} t_j}{T} \leq \frac{\sum_{j=1}^{N} T}{T} = \frac{NT}{T} = N. \tag{3.3}$$

Фізичний зміст коефіцієнта прискорення: коефіцієнт прискорення показує, в скільки разів швидше можна розв'язати задачу в паралельному режимі, ніж в послідовному [1].

В такому разі коефіцієнт, ефективності визначається як відношення коефіцієнту прискорення до кількості наявних процесорів:

$$K_E = \frac{K_П}{N}, \tag{3.4}$$

при чому, з урахуванням (3.3), коефіцієнт ефективності може приймати тільки значення на проміжку [0; 1], оскільки:

$$0 \le K_E = \frac{K_П}{N} \le \frac{N}{N} = 1.$$

Фізичний зміст коефіцієнта ефективності: коефіцієнт ефективності показує частину машинного часу, під час якої виконувались обчислення. Дійсно, перепишемо (3.4), підставивши (3.2):

$$K_E = \frac{K_П}{N} = \frac{\sum_{j=1}^{N} t_j}{TN}.$$

В чисельнику отримали загальний машинний час, протягом якого проводились обчислення, а в знаменнику – загальний затрачений машинний час.

Але з такого визначення коефіцієнта ефективності (3.4) не можна зробити висновків про те, що саме необхідно змінити, щоб збільшити значення коефіцієнта ефективності, окрім очевидного:

$$КП \rightarrow max.$$

Щоб виявити, що саме підлягає вдосконаленню та сформулювати умови оптимізації, необхідно розглянути, які саме фактори впливають на коефіцієнт прискорення. Зокрема, слід проаналізувати складові сумарного часу обчислень T. Цей час має наступні складові:

$$T = tc + tsys + ti/o + tcomm + tw, \tag{3.5}$$

де:

- tc – час проведення обчислень,
- tsys – час, витрачений на системні виклики ядра,

- ti/o – час введення/виведення даних (окрім мережевої взаємодії з іншими процесами).

- tcomm – час взаємодії з іншими процесами (пересилка даних),

- tw – час очікування.

Слід відмітити, що коефіцієнт ефективності обмежується відношенням часу виконання обчислень до повного часу виконання, а саме, відповідно до (3.4) та (3.5):

$$K_E = \frac{K_\Pi}{N} = \frac{\sum_{j=1}^{N} t_j}{TN} = \frac{t_c}{\left(t_c + t_{sys} + t_{\frac{i}{o}} + t_{comm} + t_w\right)N},$$

що відповідає закону Амдала. Складові часу очікування:

$$tw=tw,in + tw,fin,$$

де:

tw,in – час очікування під задачами вхідних даних, які є вихідними даними інших задач,

tw,fin – час простою ядер наприкінці обчислень, пов'язаний з нерівномірністю навантаження.

Ряд характеристик варто також розглядати не лише для задачі в цілому, а також для кожного процесу окремо. До таких характеристик відносяться час введення та виведення $t_{i/o}^{j}$ в j-тому процесі, час взаємодії $t_{comm}^{j}$ в j-тому процесі, час очікування вхідних даних $t_{w,in}^{j}$ в j-тому процесі, час простою j-го ядра наприкінці обчислень $t_{w,fin}^{j}$.

Введення цих величин дозволяє розглядати ряд статистичних значень, які загалом характеризують процес виконання обчислень, таких як середні за процесами значення часу взаємодії $\hat{t}_{comm}$, часу очікування вхідних даних $\hat{t}_{w,in}$, або максимальні значення цих характеристик $t_{comm}^{*}$, $t_{w,in}^{*}$. Перші впливають на загальний час виконання обчислень в цілому, в той час як останні відіграють роль обмежуючого фактора.

Розглянуті вище характеристики мають опосередкований вплив на результуючий коефіцієнт ефективності, шляхи цього впливу показані на рис. 3.7.

Час пересилки даних $t_{comm}$ залежить від зерна паралелізму.

Чим менше зерно, тим менше кожна окрема пересилка, але кількість пересилок зростає. Чим більше зерно, тим більше кожна пересилка, але

загальна кількість пересилок менша. Аналогічний вплив і на час очікування вхідних даних tw,in. Час простою наприкінці звичайно є величиною такого ж порядку, як і час виконання найбільшої підзадачі. Час виконання найбільшої підзадачі, звичайно, залежить від обраного зерна паралелізму.

За рахунок балансування навантаження не вдасться змінити такі характеристики як час обчислень tc, час роботи ядра tsys та час введення та виведення ti/o. Натомість можна змінювати характеристики, пов'язані з пересилками та незбалансованістю розподілу роботи: tcomm, tw,in та tw,fin. Ці характеристики обведені рамкою на рис. 3.7.

Відповідно до проведеного аналізу, необхідно проводити оптимізацію відповідно до наступних умов [1]:

$$\begin{cases} \sum_{j=1}^{N} t_{comm}^{j} \to min \\ \sum_{j=1}^{N} t_{w,in}^{j} \to min \\ \sum_{j=1}^{N} t_{w,fin}^{j} \to min \\ K_e \to \max \end{cases} . \qquad (3.6)$$

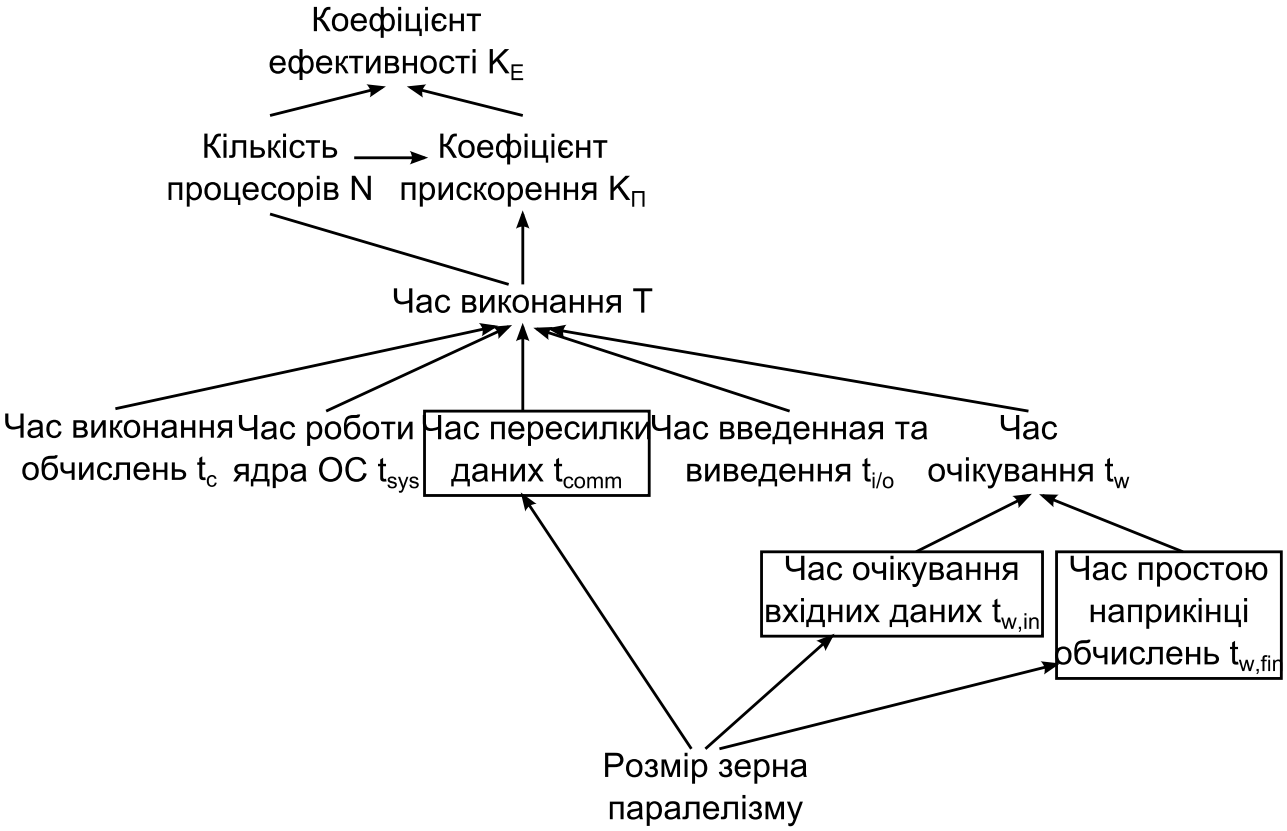



Рисунок 3.7 – Вплив динамічних характеристик виконання програми на коефіцієнт ефективності

**Висновки з розділу 3**

У першу чергу були сформульовані критерії, які дозволяють оцінити ефективність розв’язання задачі та збалансованість навантаження: коефіцієнт ефективності та коефіцієнт прискорення. Було встановлено які саме часові характеристики виконання паралельної програми, та характер цього випливу. На основі проведеного аналізу сформульовано умови оптимізації, які слід ставити в основу під час розробки підходів розв’язання поставленої задачі. Для виконання цієї оптимізації запропоновано застосувати динамічне балансування навантаження між ресурсами, що виділені для задачі.

Програми, що виконуються на високопродуктивних паралельних обчислювальних системах можна класифікувати на два види: програми, що містять паралельні реалізації алгоритмів з регулярним паралелізмом та з програми, що містять паралельні реалізації алгоритмів з нерегулярним паралелізмом [1].

# 4 РОЗРОБКА ПРОГРАМ, ПРОВЕДЕННЯ ЕКСПЕРИМЕНТІВ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

## 4.1 Розробка пакету програм

Усі задачі розроблено для комп'ютерної системи, структуру якої представлено на рисунку 2.3.

### 4.1.1 Задача №1

Реалізація операції множення матриць MA = MB*MC у чотирьох ядерній системі зі спільною пам'яттю з використанням механізму моніторів у мові C#.

Етап 1. Побудова паралельного алгоритму.

Паралельний алгоритм операції MA = MB*MC можна подати у вигляді:

MAн = MB*MCн

Спільним ресурсом є матриця MB.

Етап 2. Розроблення алгоритмів процесів

Задача P1

Таблиця 4.1 – Алгоритм задачі P1

| № | Дія | Точки синхронізації |
|---|---|---|
| 1 | Уведення MB і MC | |
| 2 | Сигнал задачі P2,P3,P4 про введення MB і MC | S234-1 |
| 3 | Копіювання MB | КД |
| 4 | Обчислення MAн = MB*MCн | |
| 5 | Чекати на завершення обчислень в P2,P3,P4 | W234-1 |
| 6 | Виведення результату MA | |

Задача P2,P3,P4

Таблиця 4.2 – Алгоритм задачі P2, P3, P4

| № | Дія | Точки синхронізації |
|---|---|---|
| 1 | Чекати на уведення в P1 | W1-234 |
| 2 | Копіювання MB | КД |
| 3 | Обчислення MAн = MB*MCн | |
| 4 | Сигнал P1 про завершення обчислень | S1-2, S1-3, S1-4 |

Етап 3. Розроблення структурної схеми взаємодії задач.

Клас Data включає поля: MB, signal[0..3], а також набір методів для організації взаємодії потоків:

- Wait() - для очікування завершення уведення даних;
- PulseAll() - для сигналу про завершення уведення даних;
- Pulse()- для сигналу про завершення обчислень;
- WaitAll() - для очікування на завершення обчислень;
- CopyMB() – для роботи зі спільним ресурсом.

Етап 4. Розроблення програми.

Реалізація задачі виконана у вигляді класу Task1 з класом для синхронізації потоків (монітор) Data (Рис. 4.1), функціями потоків ThreadInputCalc для потоку P1 і ThreadCalc для потоків P2,P3,P4.

Лістинг програми представлений у додатку А.3.

### 4.1.2 Задача №2

Реалізація операції множення матриць MA=(MB*MC)*MX у чотирьох ядерній системі зі спільною пам’яттю з використанням механізму моніторів у мові C#.

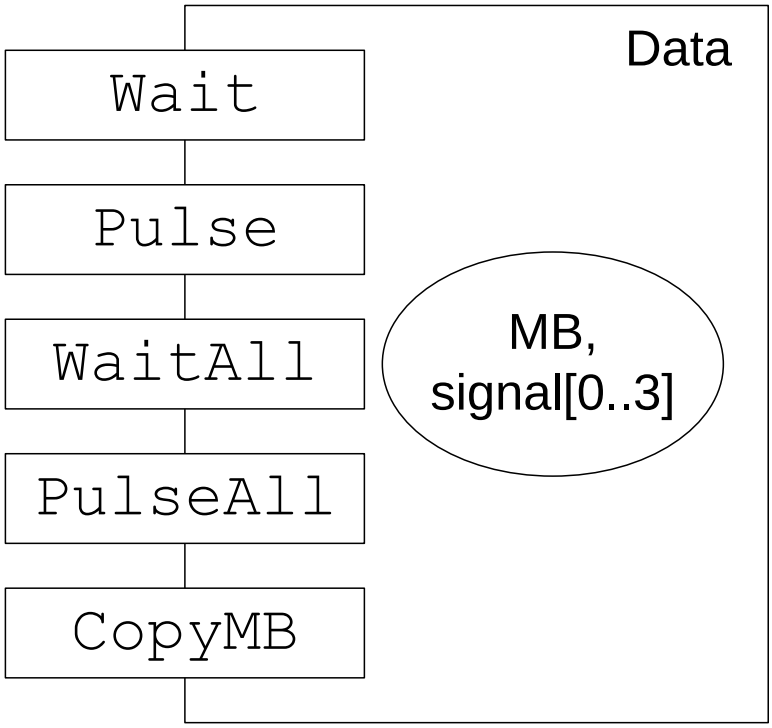



Рисунок 4.1 – Структура класу Data для задачі №1

Етап 1. Побудова паралельного алгоритму.

Паралельний алгоритм операції MA=(MB*MC)*MX можна подати у вигляді:

MAн=(MBн*MC)*MX

Спільним ресурсом є матриці MC та MX.

Етап 2. Розроблення алгоритмів процесів

Задача P1

Таблиця 4.3 – Алгоритм задачі P1

| № | Дія | Точки синхронізації |
|---|---|---|
| 1 | Уведення MB, MC і MX | |
| 2 | Сигнал задачі P2,P3,P4 про введення MB, MC і MX | S234-1 |
| 3 | Копіювання MC та MX | КД |
| 4 | Обчислення MAн=(MBн*MC)*MX | |
| 5 | Чекати на завершення обчислень в P2,P3,P4 | W234-1 |
| 6 | Виведення результату MA | |

Задача P2,P3,P4

Таблиця 4.4 – Алгоритм задачі P2, P3, P4

| № | Дія | Точки синхронізації |
|---|---|---|
| 1 | Чекати на уведення в P1 | W1-234 |
| 2 | Копіювання MC та MX | КД |
| 3 | Обчислення MAн=(MBн*MC)*MX | |
| 4 | Сигнал P1 про завершення обчислень | S1-2, S1-3, S1-4 |

Етап 3. Розроблення структурної схеми взаємодії задач.

Клас Data включає поля: MC, MX, signal[0..3], а також набір методів для організації взаємодії потоків:

- Wait() - для очікування завершення уведення даних;
- PulseAll() - для сигналу про завершення уведення даних;
- Pulse()- для сигналу про завершення обчислень;
- WaitAll() - для очікування на завершення обчислень;
- CopyMCandMX() – для роботи зі спільним ресурсом.

Етап 4. Розроблення програми.

Реалізація задачі виконана у вигляді класу Task2 з класом для синхронізації потоків (монітор) Data (Рис. 4.2), функціями потоків ThreadInputCalc для потоку P1 і ThreadCalc для потоків P2,P3,P4.

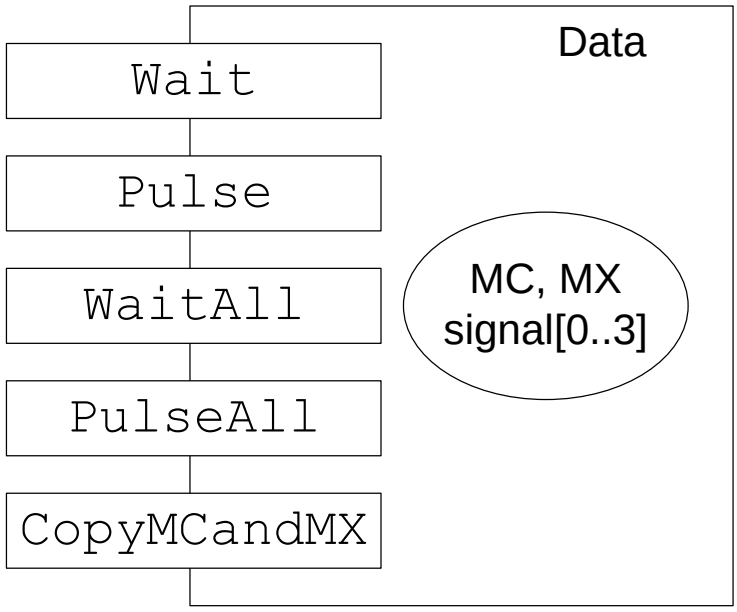



Рисунок 4.2 – Структура класу Data для задачі №2

Лістинг програми представлений у додатку А.4.

### 4.1.3 Задача №3

Реалізація операції множення матриць та векторів A=B*(MC*MX) в чотирьох ядерній системі зі спільною пам’яттю з використанням механізму моніторів у мові C#**.**

Етап 1. Побудова паралельного алгоритму.

Паралельний алгоритм операції A=B*(MC*MX) можна подати у вигляді:

Ан=B*(MC*MXн)

Спільним ресурсом є матриця MC та вектор B.

Етап 2. Розроблення алгоритмів процесів

Задача P1

Таблиця 4.5 – Алгоритм задачі P1

| № | Дія | Точки синхронізації |
|---|---|---|
| 1 | Уведення B, MC і MX | |
| 2 | Сигнал задачі P2,P3,P4 про введення B, MC і MX | S234-1 |
| 3 | Копіювання B та MC | КД |
| 4 | Обчислення Ан=B*(MC*MXн) | |
| 5 | Чекати на завершення обчислень в P2,P3,P4 | W234-1 |
| 6 | Виведення результату MA | |

Задача P2,P3,P4

Таблиця 4.6 – Алгоритм задачі P2, P3, P4

| № | Дія | Точки синхронізації |
|---|---|---|
| 1 | Чекати на уведення в P1 | W1-234 |
| 2 | Копіювання B та MC | КД |
| 3 | Обчислення Ан=B*(MC*MXн) | |
| 4 | Сигнал P1 про завершення обчислень | S1-2, S1-3, S1-4 |

Етап 3. Розроблення структурної схеми взаємодії задач.

Клас Data включає поля: B, MC, signal[0..3], а також набір методів для організації взаємодії потоків:

- Wait() - для очікування завершення уведення даних;
- PulseAll() - для сигналу про завершення уведення даних;
- Pulse()- для сигналу про завершення обчислень;
- WaitAll() - для очікування на завершення обчислень;
- CopyBandMC() – для роботи зі спільним ресурсом.

Етап 4. Розроблення програми.

Реалізація задачі виконана у вигляді класу Task3 з класом для синхронізації потоків (монітор) Data (Рис. 4.3), функціями потоків ThreadInputCalc для потоку P1 і ThreadCalc для потоків P2,P3,P4.

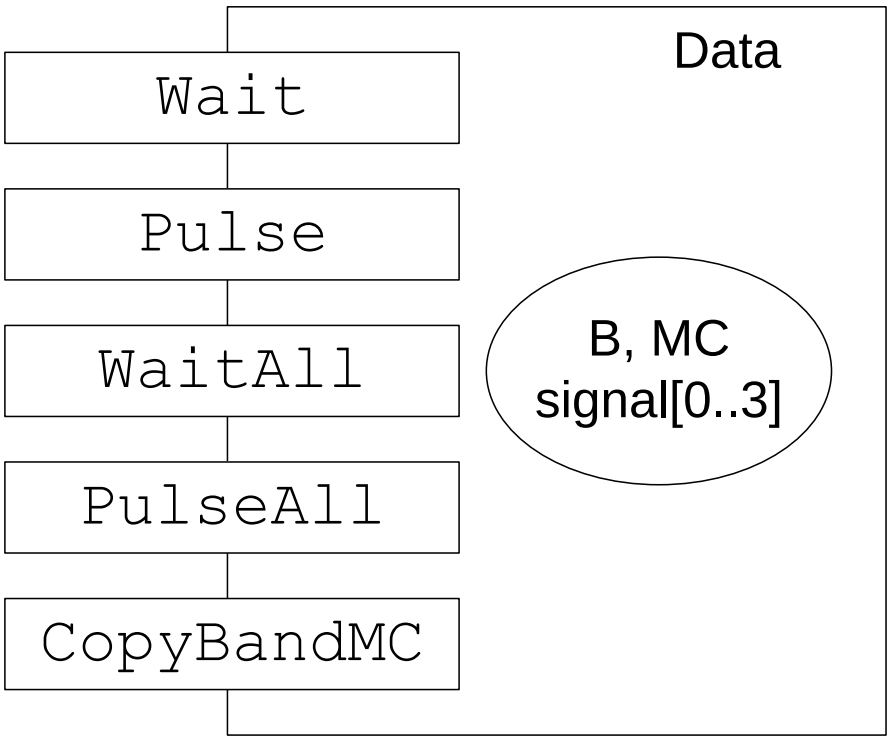



Рисунок 4.3 – Структура класу Data для задачі №3

Лістинг програми представлений у додатку А.5.

### 4.2Виконувач задач – інтерфейс користувача

Так як всі три задачі реалізовані у вигляді класів, то в головному класі (Loader) створюється екземпляр класу відповідної задачі, що призводить до розв’язанні задачі. Інтерфейс користувача виконаний в консольному режимі.

Користувачу пропонується обрати задачу і параметри, з якими вона буде розв’язуватися. Серед параметрів передбачено:

- Тип виводу вхідних/вихідних даних:
  - Без виводу, тільки час розв'язання
  - Вивід на екран
  - Вивід в файли (MA.txt, MB.txt, ...)
  - Вивід на екран і в файли
- Тип даних елементів матриць:
  - цілі
  - дійсні з фіксованою точкою
  - дійсні з плаваючею точкою
  - комплексні
- Тип заповнення матриць випадковими числами:
  - максимальні значення
  - значення зручні для сприйняття
- Розмір матриць N (4..2000, N%4==0)

При введенні некоректних даних з’являється повідомлення «Некоректний ввід даних». Матриці та вектори заповнюються випадковими числами. Після завершення розв’язання задачі виводиться час виконання.

Лістинг програми представлений в додатку А.1.

### 4.3Тестування

Усі тестування виконано на комп’ютері такої конфігурації:
Процесор: тип AMD Athlon II X3 450 3.2 GHz;
Оперативна пам’ять: 2 ГБ;
Накопичувач НЖМД: Maxtor 500 GB.

Приклад виконання задачі №1:

```
Оберіть номер задачі:
 1 - MA=MB*MC
 2 - MA=(MB*MC)*MX
 3 - A=B*(MC*MX)
? 1
Оберіть тип виводу:
 1 - Без виводу, тільки час розв'язання
 2 - Вивід на екран
 3 - Вивід в файли (MA.txt, MB.txt, ...)
 4 - Вивід на екран і в файли
```

```
? 2
Оберіть тип даних елементів матриць:
 1 - цілі
 2 - дійсні з фіксованою точкою
 3 - дійсні з плаваючею точкою
 4 - комплексні
? 1
Оберіть тип заповнення матриць випадковими числами:
 1 - максимальні значення
 2 - значення зручні для сприйняття
? 2
Введіть розмір матриць N (4..2000, N%4==0): 8

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

MC

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

MA

225 104 100 146 128 126 153 148
209 143 186 92 225 156 147 194
295 191 155 187 230 191 194 205
250 140 151 138 180 170 174 191
99 23 54 68 63 20 81 75
```

```
247 121 143 178 140 180 175 166
312 129 195 154 220 174 216 253
131 90 107 68 110 134 83 115

Час розв'язання задачі: 00:00:00.2812500
```

## 4.4Визначення залежності часу виконання задач від розмірності матриць та типу елементів матриць

Таблиця залежності часу виконання задачі від розмірності матриць та типу елементів матриць. Тестування проводилось з виводом вхідних/вихідних даних в файли та з заповненням матриць випадковими числами з максимального діапазону значень.

Таблиця 0.7 – Залежність часу виконання задачі №1 від розмірності матриць та типу елементів

| | | Тип елементів матриць | | | |
|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні |
| Розмірність матриць | 100 | 0.015 | 0.171 | 0.046 | 0.093 |
| | 240 | 0.171 | 1.984 | 0.281 | 0.531 |
| | 500 | 1.359 | 15.937 | 1.718 | 3.937 |
| | 1000 | 10.125 | 1:53.968 | 12.968 | 26.437 |
| | 2000 | 1:12.937 | 15:30.734 | 1:31.796 | 3:28.015 |

Таблиця 0.8 – Залежність часу виконання задачі №2 від розмірності матриць та типу елементів

| | | Тип елементів матриць | | | |
|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні |
| Розмірність матриць | 100 | 0.031 | 0.281 | 0.046 | 0.093 |
| | 240 | 0.218 | 3.734 | 0.375 | 0.828 |
| | 500 | 2.843 | 32.453 | 4.515 | 7.578 |
| | 1000 | 25.531 | 4:26.671 | 32.703 | 1:04.437 |
| | 2000 | 3:49.812 | 36:49.578 | 4:37.656 | 8:56.125 |

Таблиця 0.9 – Залежність часу виконання задачі №3 від розмірності матриць та типу елементів

| | | Тип елементів матриць | | | |
|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні |
| Розмірність матриць | 100 | 0.015 | 0.140 | 0.031 | 0.062 |
| | 240 | 0.109 | 1.562 | 0.203 | 0.406 |
| | 500 | 0.984 | 13.296 | 1.390 | 3 |
| | 1000 | 7.937 | 1:44.046 | 11.218 | 22.203 |
| | 2000 | 1:05.593 | 14:39.843 | 1:30.609 | 3:05.406 |


Рисунок 0.4 – Графік залежності часу виконання задачі №1 від розмірності матриць та типу елементів

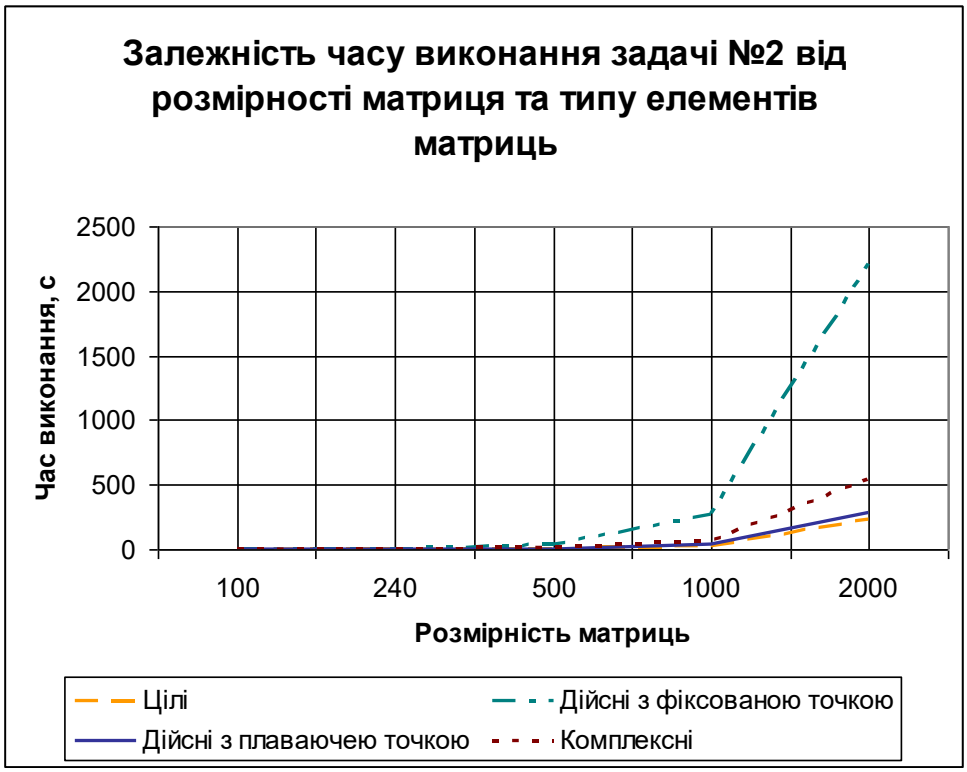



Рисунок 0.5 – Графік залежності часу виконання задачі №2 від розмірності матриць та типу елементів

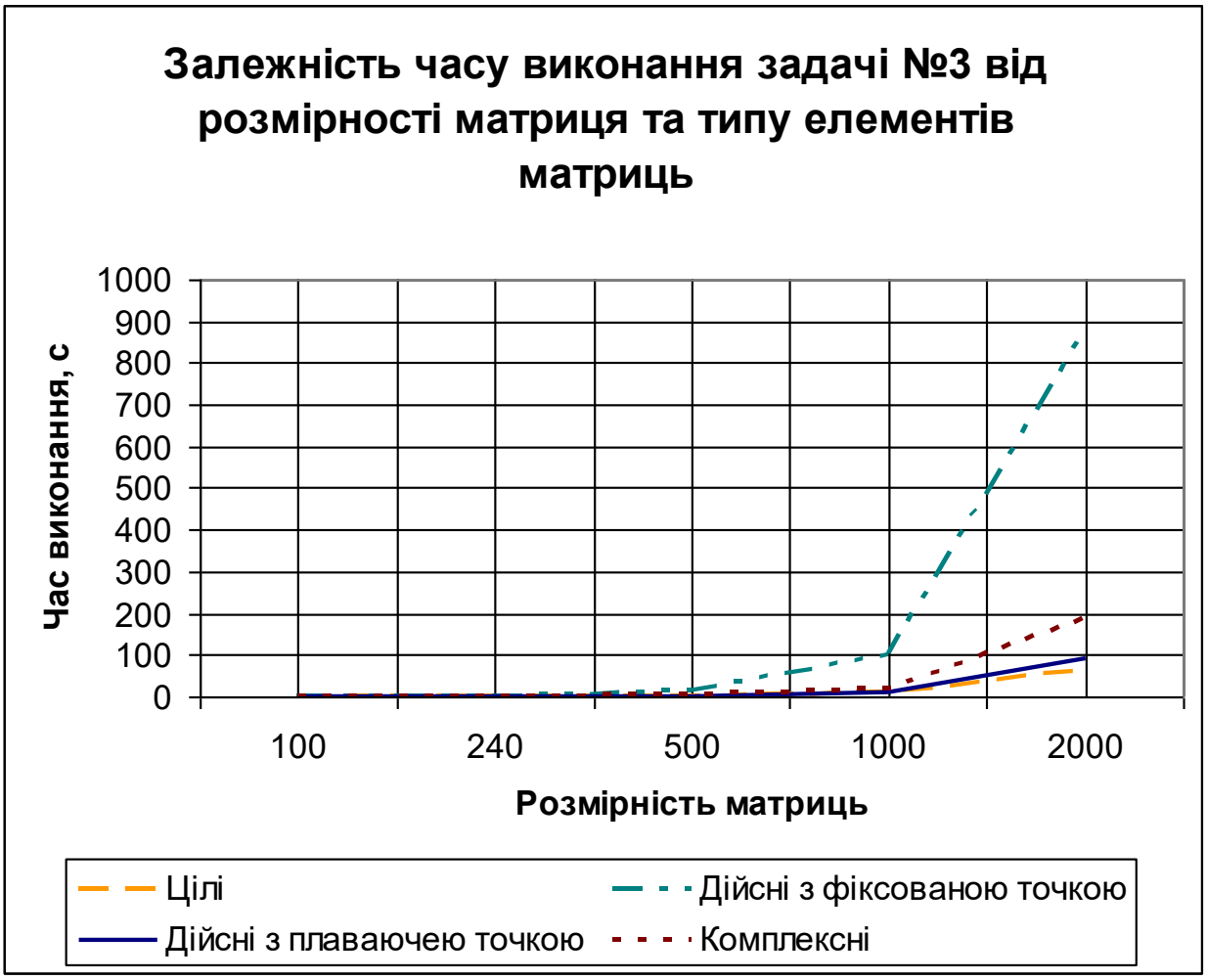



Рисунок 0.6 – Графік залежності часу виконання задачі №3 від розмірності матриць та типу елементів

Отже, з даних першого тестування видно, що час виконання знаходиться в нелінійній залежності від розмірності матриць та стрімко росте при її збільшенні. Менш за все часу для обчислень потрібно при виконанні операцій з цілими числами та дійсними числами з плаваючою точкою. Найбільш «важкими» обчисленнями виявились операції з дійсними числами з фіксованою точкою. Задача №2 включає найбільшу кількість операцій (подвійне перемноження матриць) і тому час виконання задачі є найбільшим серед інших задач при рівних умовах. Задача №1 та №3 не суттєво відрізняються за часом виконання, бо включають приблизно однакові операції.

### 4.5Визначення залежності часу виконання задач від кількості ядер

На слідуючих тестових наборах можна з'ясувати, яке ж насправді підвищення продуктивності дає збільшення кількості ядер.

Таблиця 0.10 – Залежність часу виконання задачі №1 від кількості задіяних ядер

| | | Тип елементів матриць | | | |
|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні |
| Кількість ядер | 1 | 0:29.156 | 7:11.468 | 0:32.203 | 1:32 |
| | 2 | 0:15.640 | 3:37.609 | 0:19.062 | 0:49.812 |
| | 3 | 0:12.093 | 2:45.187 | 0:15.468 | 0:37.296 |
| | 4 | 10.125 | 1:53.968 | 12.968 | 26.437 |

Таблиця 0.11 – Залежність часу виконання задачі №2 від кількості задіяних ядер

| | | Тип елементів матриць | | | |
|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні |
| Кількість ядер | 1 | 1:31.546 | 17:40.984 | 1:47.750 | 4:04.828 |
| | 2 | 0:47.390 | 8:40.890 | 1:01.890 | 2:07.484 |
| | 3 | 0.37.640 | 6:33.484 | 0:46.234 | 1:37.031 |
| | 4 | 25.531 | 4:26.671 | 32.703 | 1:04.437 |

Таблиця 0.12 – Залежність часу виконання задачі №3 від кількості задіяних ядер

| | | Тип елементів матриць | | | |
|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні |
| Кількість ядер | 1 | 0:29.453 | 7:22.390 | 0:41.500 | 1:26.015 |
| | 2 | 0:14.859 | 3:49.937 | 0:19.718 | 0:44.015 |
| | 3 | 0:11.390 | 2:41.265 | 0:16.140 | 0:35.109 |
| | 4 | 7.937 | 1:44.046 | 11.218 | 22.203 |

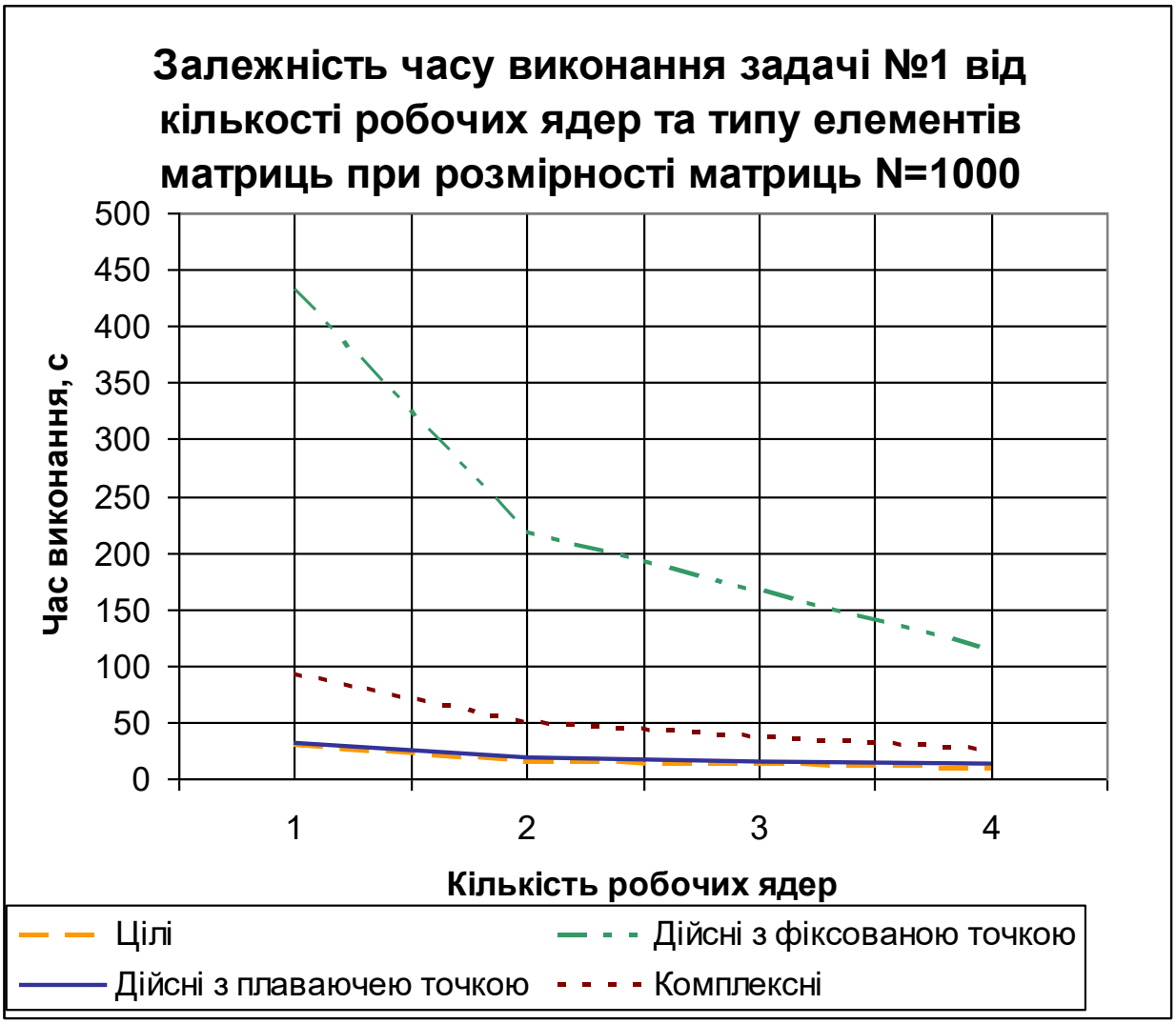



Рисунок 0.7 – Графік залежності часу виконання задачі №1 від кількості задіяних ядер

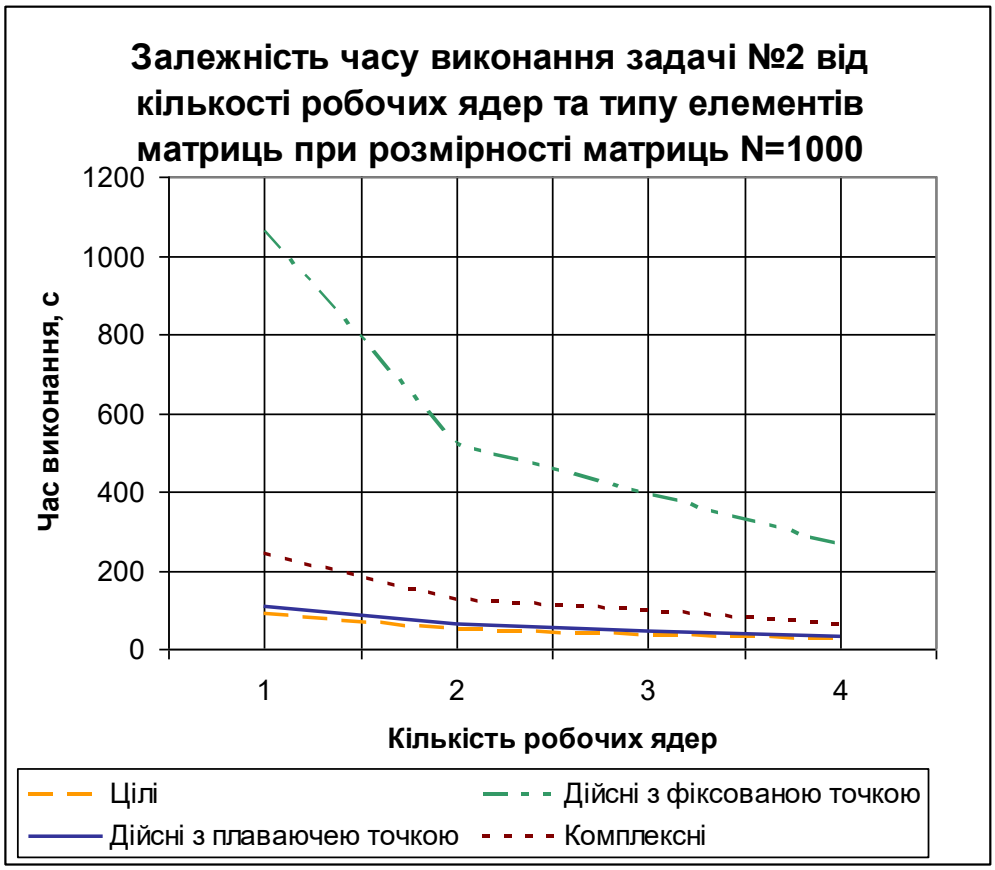



Рисунок 0.8 – Графік залежності часу виконання задачі №2 від кількості задіяних ядер

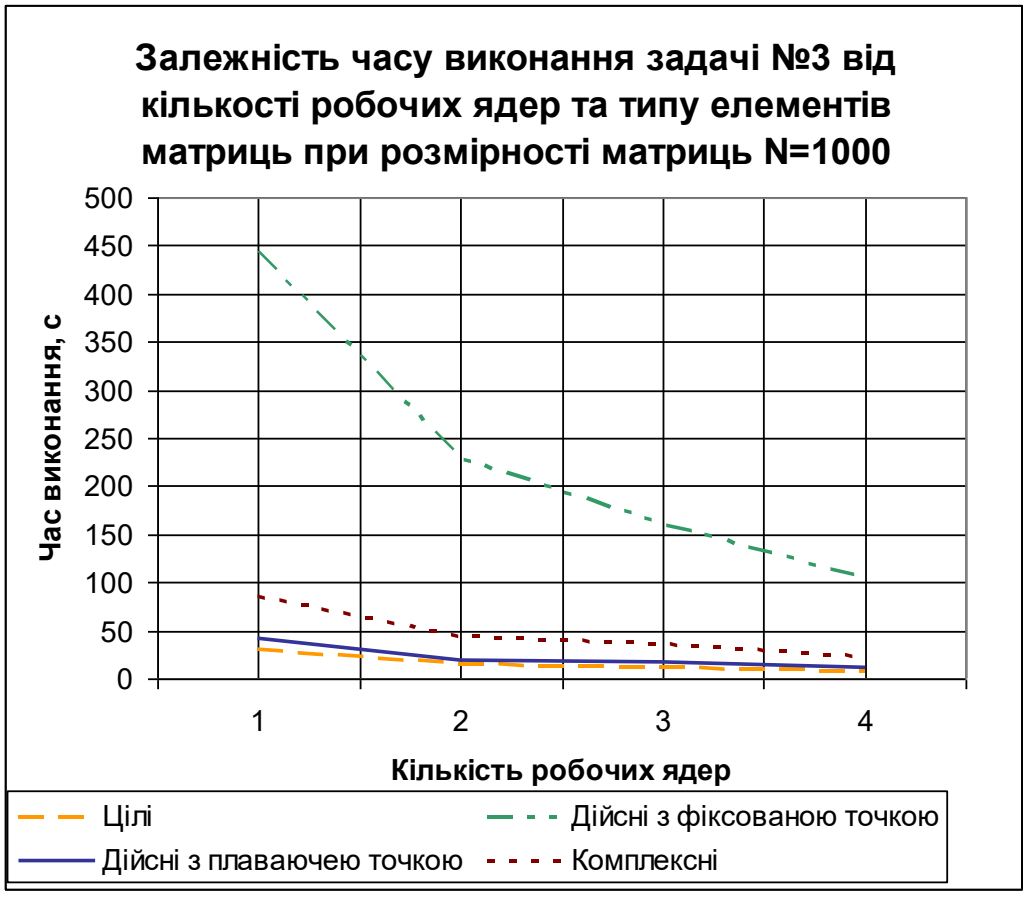



Рисунок 0.9 – Графік залежності часу виконання задачі №3 від кількості задіяних ядер

Друге тестування показало, що час виконання задачі не зовсім пропорційний кількості ядер, які виконують цю задачу. Але варто зауважити, що дані задачі мають в собі ділянки, на яких операції виконуються послідовно і тому перевага в кількості ядер на цих ділянках спростовується, тобто ідеальна пропорційність часу виконання від кількості ядер буде при ідеальній розпаралеленості задачі та відсутності видатків на організацію паралельного виконання.

### 4.6Визначення коефіцієнтів прискорення

Таблиця 0.13 – Коефіцієнти прискорення виконання задачі №1 при використання декількох ядер

| | | Тип елементів матриць | | | | Середнє значення |
|---|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні | |
| Кількість ядер | 1 | 1,00 | 1,00 | 1,00 | 1,00 | **1,00** |
| | 2 | 1,86 | 1,98 | 1,69 | 1,85 | **1,85** |
| | 3 | 2,41 | 2,61 | 2,08 | 2,47 | **2,39** |
| | 4 | 2,88 | 3,79 | 2,48 | 3,48 | **3,16** |

Таблиця 0.14 – Коефіцієнти прискорення виконання задачі №2 при використання декількох ядер

| | | Тип елементів матриць | | | | Середнє значення |
|---|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні | |
| Кількість ядер | 1 | 1,00 | 1,00 | 1,00 | 1,00 | **1,00** |
| | 2 | 1,93 | 2,04 | 1,74 | 1,92 | **1,91** |
| | 3 | 2,43 | 2,70 | 2,33 | 2,52 | **2,50** |
| | 4 | 3,59 | 3,98 | 3,29 | 3,80 | **3,66** |

Коефіцієнти прискорення виконання задач при використанні декількох ядер чітко показують, що сподіване підвищення продуктивності не відбувається. Так замість коефіцієнту прискорення 4 при використанні 4-х ядер ми отримали коефіцієнт 3,16-3,88. Це пояснюється не повною завантаженісттю ядер на деяких етапах виконання, наприклад, при генерування вхідних даних та записі вихідних даних у файли.

Таблиця 0.15 – Коефіцієнти прискорення виконання задачі №3 при використання декількох ядер

| | | Тип елементів матриць | | | | Середнє значення |
|---|---|---|---|---|---|---|
| | | Цілі | Дійсні з фіксованою точкою | Дійсні з плаваючею точкою | Комплексні | |
| Кількість ядер | 1 | 1,00 | 1,00 | 1,00 | 1,00 | **1,00** |
| | 2 | 1,98 | 1,92 | 2,10 | 1,95 | **1,99** |
| | 3 | 2,59 | 2,74 | 2,57 | 2,45 | **2,59** |
| | 4 | 3,71 | 4,25 | 3,70 | 3,87 | **3,88** |

**Висновки з розділу 4**

У даному розділі виконано розробку пакета програм для розв'язання в ПКС ряду матричних операцій з використанням мови C#.

Також проведено експериментальні дослідження пакета в реальній багатоядерній ПКС на базі процесора типу AMD Athlon II X3 450 3.2 GHz, які показали, що час виконання матричних завдань скорочується в 3,16 – 3,8 рази.

# ВИСНОВКИ

1. Розвиток сучасних кластерних систем пов’язаний зі збільшенням обчислювальної потужності вузлів системи, яке на сьогодні можна реалізувати шляхом використання у вузлах багатоядерних процесорів родини AMD. Організація обчислень у комп’ютерній системі з багатоядерною архітектурою (КСБА) пов’язана з вирішенням двох задач – організації ефективної взаємодії вузлів системи і обчислень у кожному вузлі, які у свою чергу вимагають ефективної реалізації обчислень у кожному ядрі і їх взаємодії. КСБА є складною системою, тому організація обчислювальних процесів і розробка програмного забезпечення для неї повинні базуватися на використанні математичних моделей паралельних обчислень, що забезпечують інтерфейс між архітектурою КСБА і моделлю програмування.

Розглянуто апаратні засоби сучасних паралельних комп’ютерних систем (ПКС). Показано, що в них широко використовуються багатоядерні процесори. Виконано аналіз структурної організації багатоядерних процесорів, який показав, що вони відрізняються кількістю ядер (2-16), різними системами зв’язку ядер, використанням кеш-пам’яті різних рівнів і різного обсягу.

2. Розвиток кластерної архітектури пов’язаний з використанням багатоядерних процесорів розробки AMD та Intel. За допомогою CSP (Communicating Sequential Processes) теорії можлива побудова математичної моделі обчислень для КСБА з різною структурною організацією. Розробникові програмного забезпечення для КСБА стане доступним ефективний механізм створення програмного забезпечення, що дозволяє скоротити час проектування програмного продукту і підвищити його надійність.

3. Сформульовано критерії, які дозволяють оцінити ефективність розв’язання задачі та збалансованість навантаження: коефіцієнт ефективності та коефіцієнт прискорення. Встановлено які саме часові характеристики виконання паралельної програми, та характер цього випливу. На основі проведеного аналізу сформульовано умови оптимізації, які слід ставити в основу під час розробки підходів розв’язання поставленої задачі. Для виконання цієї оптимізації запропоновано застосувати динамічне балансування навантаження між ресурсами, що виділені для задачі.

4. Розглянуто особливості програмного забезпечення для ПКС. Наведено, що програмне забезпечення для ПКС розробляється з використанням потоків (процесів), які забезпечують завантаження ядер багатоядерних процесорів. Виконано розробку пакету програм для розв’язання в ПКС ряду матричних операцій. Проведено експериментальні дослідження пакета в реальній багатоядерній ПКС на базі AMD Athlon II X3 450 3.2 GHz, які показали, що час виконання матричних завдань скорочується в 3,16 – 3,8 рази.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Стіренко С.Г. Методи та засоби ефективної обробки паралельних задач в комп'ютерних кластерних системах : дис. д-р техн. наук. : 05.13.05 – комп'ютерні системи та компоненти / С. Г. Стіренко. – К., 2015. – 320 л.
2. Воеводин Вл.В., Жуматий С.А. Вычислительное дело и кластерные системы. – М.: Изд-во МГУ, 2007. – 150 с.
3. Ваврук Є. Я. Організація паралельних обчислень / Є. Я. Ваврук, О. Л. Лашко. – Львів: Львівська політехніка, 2010. – 79 с. – (Навчальне видання).
4. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб: БХВ-Петербург, 2002.
5. М. Борисов, UNIX-кластеры. Открытые системы, # 2, 1995, сс.22-28.
6. Список 500 самых мощных компьютеров мира. 31-я редакция [Електронний ресурс]. – Режим доступу: http:// www.parallel.ru.
7. Каляев И.А., Левин И.И., Семерников Е.А., Шмойлов В.И. Реконфигурируемые мультиконвейерные вычислительные структуры. – М.: ид-во ЮНЦ РАН, 2008. – 320 с.
8. Андрианов С.Н., Дегтярев А.Б. Параллельные и распределенные вычисления. Часть1. – СПб. : Изд–во С– Петерб. унив.- та, 2007. – 61 с.
9. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77 с.
10. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. – СПб.: БХВ. – Петербург, 2002. – 608 с.
11. Гергель В.П. Теория и практика параллельных вычислений. – М.: Бином. Лаборатория знаний, 2007. – 424 с.
12. Жуков І.А., Корочків О.В. Паралельні та розподілені обчислення: Навч. посібник. – К.: Корнейчук, 2007. – 246 с.
13. Тютюнник М.І. Паралельні алгоритми та засоби для розв'язання деяких задач масових обчислень [Електронний ресурс]. – Режим доступу: http://www.iapmm.lviv.ua/chyt2010/materials/pc2010-02-T-32.pdf.
14. Воеводин Вл.В., Жуматий С.А. Вычислительное дело и кластерные системы. – М.: Изд-во МГУ, 2007. – 150 с.

15. Гофф Макс К. Сетевые распределенные вычисления. Достижения и проблемы. – М.: Кудиц-Образ, 2005. – 320 с.
16. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования.: Пер. с англ. – М.: Изд. дом Вильямс. – 2003. – 512 с.
17. Кулаков Ю.О., Луцький Г.М. Комп'ютерні мережі. – К.: Юніор, 2003. – 400 с.
18. Любченко В.С. К проблеме создания модели параллельных вычислений // Материалы 3-го межд. Семинара «Параллельные вычисления и задачи управления». – Самара. - 2006. – С. 181 – 186.
19. Воеводин В.В. Математические модели и методы в параллельных процесах. – М.: Наука, 1984. – 296 с.
20. Топорков В.В. Модели распределенных вычислений. – М.: ФИЗМАТЛИТ, 2004. – 320 с.
21. Любченко В.С. Автоматная модель параллельных вычислений // Труды 3-й межд. конф. «Высокопроизводительные параллельные вычисления на кластерных системах». – М. - 2006. – С. 1359 – 1374.
22. Бендас Ю. В. Математичні моделі організації обчислень в кластерних системах на основі вибору оптимальних засобів взаємодії процесів / Ю. В. Бендас. – Тернопіль, 2012. – 91 с. – (Кафедра комп'ютерної інженерії).
23. Hennessy, John L. Computer architecture: a quantitative approach / John L Hennessy, David A Patterson. – Amsterdam, Noord-Hooland, Netherlands: Elsevier, 2011. – 824 p.
24. Recent trends in the marketplace of high performance computing / Eri¬ch Strohmaier, Jack J. Dongarra, Hans W. Meuer, Horst D. Simon // Parallel Computing. - 2005. – Vol. 31, no. 3-4. – P. 261-273.
25. Flynn, M. J. Very high-speed computing systems / M.J. Flynn. – 1966. – dec. –Vol. 54, no. 12. – P. 1901-1909.
26. An Evaluation of Vectorizing Compilers / S. Maleki, Yaoqing Gao, M.J. Garzaran [et al.] // Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. – Los Alamitos, CA, USA: IEEE Computer Society, 2011. – oct. – P. 372 -382.
27. An Updated Set of Basic Linear Algebra Subprograms (BLAS) / L. S. Blackford, J. Demmel, J. Dongarra [et al.] // ACM Transactions on Mathematical Software. - 2001. - Vol. 28. - P. 135-151.
28. The Nanoelectronic Road Ahead: Despite Challenges, Silicon Offers 20 More Years of Semiconductor Progress [Электронный ресурс]. – Режим доступа: http://gtresearchnews.gatech.edu/newsrelease/FUTURECHIP. html. – Последнее обращение: 01.04.2013. – Название с экрана.

29. Multi-core processors – the next evolution in computing [Електроний ресурс].– Режим доступа: http://www.amd.com/us-en/assets/content_ type/white_papers_and_tech_docs/Multi-Core_Processing_33211A. pdf. – Останнє звернення: 01.04.2013. – Назва з екрану.
30. Sutter, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software / H. Sutter // Dr. Dobb's Journal. –2005. – Vol. 30, no. 3.
31. From a few cores to many: a tera-scale computing research overvi¬ew [Electronic resource]. – Access mode: http://download.intel.com/ research/platform/ terascale/terascale_overview_paper.pdf. – Last access: 01.04.2013. – Title from the screen.
32. Система пространственного распределения заданий в распределённых вычислительных системах / А. В. Симоненко, С. В. Пих, Н. В. Слу¬цкий, В. В. Воробйов // Вісник НТУУ «КПІ». Інформатика, управління та обчислювальна техніка. – 2012. – № 56. – С. 170.
33. Tanenbaum, Andrew S. Distributed Systems: Principles and Paradigms (2nd Edition) / Andrew S. Tanenbaum, Maarten van Steen. – Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.
34. Tanenbaum, Andrew S. Modern Operating Systems / Andrew S. Tanenbaum. – 3rd edition. – Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.
35. Sun. – OpenSPARC T2 Core Microarchitecture Specification, A edition, 2007. – Dec.
36. Cybenko, G. Dynamic load balancing for distributed memory multiprocessors / G. Cybenko // J. Parallel Distrib. Comput. 1989. – Oct. – Vol. 7, no. 2. – P. 279-301.
37. Farina, Fabio. Grid and HPC Dynamic Load Balancing with Lattice Boltzmann Models / Fabio Farina, Gianpiero Cattaneo, Alberto Dennunzio // On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE / Ed. by Robert Meersman, Zahir Tari. – [S. 1.]: Springer Berlin Heidelberg, 2006. – Vol. 4276 of Lecture Notes in Computer Science. – P. 1152¬-1162. – URL: http://dx.doi.org/10.1007/11914952_6.
38. Performance Development | TOP500 Supercomputer Sites [Electronic resource].– Access mode: http://top500.org/statistics/perfdevel/.– Last access: 28.02.2013. – Title from the screen.
39. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers / Lorin Hochstein, Jeff Carver, Forrest Shull [et al] // Proceedings of the 2005 ACM/IEEE conference on Supercomputing. – SC '05. – Washington, DC, USA: IEEE Computer Society, 2005, - P. 35.
40. Симоненко В.П. Организация вычислительных процессов в ЭВМ, ком¬плексах, сетях и системах / Симоненко В.П. – К.: ВЕК, 1997. – 304 с.

# ДОДАТКИ
# ЛІСТИНГИ ПРОГРАМ

## А.1 Файл Main.cs

```
using System;

namespace Tasks
{
 class Loader
 {
 // Головна функція
 static void Main(string[] args)
 {
 int n; // розмір матриць
 int TypeNum;// номер типу елементів матриць
 int TaskNum;// номер задачі
 int OutType;// номер типу виводу
 int RndType;// номер типу заповнення матриць
 Type DataType;// тип елементів матриць
 DateTime tStart, // початок розв'язання задачі
 tEnd;// кінець розв'язання задачі
 bool DoOutScreen;// признак виводу на екран
 bool DoOutFile;// признак виводу в файли
 bool FullRange;// признак заповнення матриць числами максимального значення

 n = 0; DataType = typeof(int); OutType = 0; DoOutScreen = true; DoOutFile = true;
 // Ввід даних
 try
 {

 Console.Write("Оберіть номер задачі:\n" +
 "\t1 - MA=MB*MC\n\t2 - MA=(MB*MC)*MX\n\t3 - A=B*(MC*MX)\n? ");
 try
 {
 TaskNum = int.Parse(Console.ReadLine());
 if (TaskNum < 1 || TaskNum > 3)
 throw null;
 }
 catch (Exception) { throw null; }

 Console.Write("Оберіть тип виводу:\n" +
 "\t1 - Без виводу, тільки час розв'язання\n" +
 "\t2 - Вивід на екран\n\t3 - Вивід в файли (MA.txt, MB.txt, ...)\n" +
 "\t4 - Вивід на екран і в файли\n? ");
 try
 {
 OutType = int.Parse(Console.ReadLine());
 if (OutType < 1 || OutType > 4)
 throw null;
 }
 catch (Exception) { throw null; }
 DoOutScreen = (OutType == 2 || OutType == 4);
 DoOutFile = (OutType == 3 || OutType == 4);

 Console.Write("Оберіть тип даних елементів матриць:\n" +
 "\t1 - цілі\n\t2 - дійсні з фіксованою точкою\n" +
 "\t3 - дійсні з плаваючею точкою\n\t4 - комплексні\n? ");
 try
 { TypeNum = int.Parse(Console.ReadLine());
```

```
 if (TypeNum < 1 || TypeNum > 4)
 throw null;
 }
 catch (Exception) { throw null; }
 switch (TypeNum)
 {
 case 1: DataType = typeof(int); break;
 case 2: DataType = typeof(decimal); break;
 case 3: DataType = typeof(double); break;
 case 4: DataType = typeof(Complex); break;
 }

 Console.Write("Оберіть тип заповнення матриць випадковими числами:\n" +
 "\t1 - максимальні значення\n\t2 - значення зручні для сприйняття\n? ");
 try
 {
 RndType = int.Parse(Console.ReadLine());
 if (RndType < 1 || RndType > 2)
 throw null;
 }
 catch (Exception) { throw null; }
 FullRange = RndType == 1;

 Console.Write("Введіть розмір матриць N (4..2000, N%4==0): ");
 try
 {
 n = int.Parse(Console.ReadLine());
 if (n < 4 || n > 2000 || n % 4 != 0)
 throw null;
 }
 catch (Exception) { throw null; }
 }
 catch (Exception) { Console.WriteLine("Некоректний ввід даних"); return; }

 // Розв'язання задачі
 tStart = DateTime.Now;
 switch (TaskNum)
 {
 case 1: Task1 task1 = new Task1(n, DoOutScreen, DoOutFile, FullRange, DataType); break;
 case 2: Task2 task2 = new Task2(n, DoOutScreen, DoOutFile, FullRange, DataType); break;
 case 3: Task3 task3 = new Task3(n, DoOutScreen, DoOutFile, FullRange, DataType); break;
 }
 tEnd = DateTime.Now;

 Console.WriteLine("\nЧас розв'язання задачі: " + (tEnd - tStart).ToString());
 Console.ReadLine();
 }
 }
}
```

## A.2 Файл Complex.cs

```
namespace Tasks
{
 public struct Complex
 {
 public int re;
 public int im;

 public Complex(int _re, int _im) //constructor
 {
 re = _re; im = _im;
 }

 public static Complex operator +(Complex c1, Complex c2)
 {
 return new Complex(c1.re + c2.re, c1.im + c2.im);
 }
 public static Complex operator -(Complex c1, Complex c2)
 {
 return new Complex(c1.re - c2.re, c1.im - c2.im);
 }
 public static Complex operator *(Complex c1, Complex c2)
 {
 return new Complex(c1.re * c2.re - c1.im * c2.im,
 c1.re * c2.im + c2.re * c1.im);
 }
 public override string ToString()
 {
 return (System.String.Format("{0} + {1}i", re, im));
 }
 }
}
```

## A.3 Файл Task1.cs

```
using System;
using System.Threading;
using System.IO;
namespace Tasks
{
 /// <summary>
 /// Клас задачі №1
 /// </summary>
 class Task1
 {
 Data data;// Монітор
 object[][] MA;// вихідна матриця MA
 object[][] MC;// вхідна матриця MC
 bool DoOutScreen;// признак виводу на екран
 bool DoOutFile;// признак виводу в файли
 bool FullRange;// признак заповнення матриць числами максимального значення
 Type MatrixType;// тип елементів матриць

 /// <summary>
 /// Клас монітору
 /// </summary>
 public class Data
 {
 public object[][] MB;// вхідна матриця MB
```

```
int n;// розмір матриць
object sync = new object();// об'єкт синхронізації
bool[] signal;// сигнали потоків

/// <summary>
/// Конструктор монітору
/// </summary>
/// <param name="_n">розмір матриць</param>
public Data(int _n)
{
int i;// лічильник

n = _n;
MB = new object[n][];
for (i = 0; i < n; i++)
MB[i] = new object[n];
signal = new bool[4];
for (i = 0; i < 4; i++)
signal[i] = false;
}

/// <summary>
/// Очікувати сигналу від потоку №1
/// </summary>
public void Wait()
{
lock (sync)
{
while (!signal[0])
{
Monitor.Wait(sync);
}
}
}

/// <summary>
/// Очікувати всіх сигналів від потоків №2, №3, №4 і сброс
/// </summary>
public void WaitAll()
{
lock (sync)
{
while (!(signal[1] && signal[2] && signal[3]))
{
Monitor.Wait(sync);
}
signal[1] = false; signal[2] = false; signal[3] = false;
}
}

/// <summary>
/// Сигнал від потоку
/// </summary>
/// <param name="thid">номер потоку</param>
public void Pulse(int thid)
{
lock (sync)
{
signal[thid] = true;
Monitor.Pulse(sync);
}
}

/// <summary>
```

```
/// Сигнал від потоку №1 (потокам №2, №3, №4)
/// </summary>
public void PulseAll()
{
lock (sync)
{
signal[0] = true;
Monitor.PulseAll(sync);
}
}

/// <summary>
/// Копіювання матриці MB
/// </summary>
/// <param name="dstMB">посилання на приймача матриці MB</param>
public void CopyCR(ref object[][] dstMB)
{
dstMB = (object[][])MB.Clone();
}
}

//------------------------------------------------------------------
/// <summary>
/// Конструктор задачі
/// </summary>
/// <param name="n">розмір матриць</param>
/// <param name="_DoOutScreen">признак виводу на екран</param>
/// <param name="_DoOutFile">признак виводу в файли</param>
/// <param name="_FullRange">признак заповнення матриць числами максимального
значення</param>
/// <param name="_MatrixType">тип елементів матриць</param>
public Task1(int n, bool _DoOutScreen, bool _DoOutFile, bool _FullRange, Type
_MatrixType)
{
Thread[] ths = new Thread[4];// список потоків
int i;// лічильник

DoOutScreen = _DoOutScreen;
DoOutFile = _DoOutFile;
FullRange = _FullRange;
MatrixType = _MatrixType;

MA = new object[n][];
MC = new object[n][];
for (i = 0; i < n; i++)
{
MA[i] = new object[n];
MC[i] = new object[n];
}
data = new Data(n);

// Створення і старт потоків №2, №3, №4
for (i = 0; i < 3; i++)
{
ths[i] = new Thread(ThreadCalc);
ths[i].Start(new int[] { n, i + 1 });
}
// Створення і старт потоку №1
ths[3] = new Thread(ThreadInputCalc);
ths[3].Start(n);

for (i = 0; i < 3; i++)
ths[i].Join();
ths[3].Join();
```

```
}

/// <summary>
/// Функція потоку №1
/// </summary>
/// <param name="_n">розмір матриць</param>
void ThreadInputCalc(object _n)
{
int i, j;// лічильник
Random rnd = new Random();// об'єкт для генерування випадкових чисел
int n = (int)_n;// розмір матриць
object[][] MB;// копія MB
FileStream fs;// файловий потік
StreamWriter sw;// записувач в файловий потік

int MaxRndInt;// максимальне ціле
double MaxRndDouble;// максимальне дійсне з плаваючою точкою
double MaxRndDecimal;// максимальне з фіксованою точкою
int MaxRndComplex;// максимальне комплексне

if (FullRange)
{
MaxRndInt = (int)Math.Sqrt(int.MaxValue) / 2000;
MaxRndDouble = Math.Sqrt(double.MaxValue) / 2000;
MaxRndDecimal = Math.Sqrt((double)decimal.MaxValue) / 2000;
MaxRndComplex = (int)Math.Sqrt(int.MaxValue) / 2000 / 2;
}
else
{
MaxRndInt = 10;
MaxRndDouble = 10;
MaxRndDecimal = 10;
MaxRndComplex = 10;
}
MB = new object[n][];
for (i = 0; i < n; i++)
MB[i] = new object[n];

// заповнення вхідних матриць випадковими числами залежно від їх типу
if (MatrixType == typeof(int))
{
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
{
data.MB[i][j] = rnd.Next(MaxRndInt);
MC[i][j] = rnd.Next(MaxRndInt);
}
}
else if (MatrixType == typeof(double))
{
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
{
data.MB[i][j] = rnd.NextDouble() * MaxRndDouble;
MC[i][j] = rnd.NextDouble() * MaxRndDouble;
}
}
else if (MatrixType == typeof(decimal))
{
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
{
data.MB[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
MC[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
```

```
}
}
else if (MatrixType == typeof(Complex))
{
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
{
data.MB[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
MC[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
}
}
// Вивід вхідних матриць на екран
if (DoOutScreen)
{
Console.WriteLine("MB");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(data.MB[i][j].ToString() + " ");
Console.WriteLine();
}
Console.WriteLine("MC");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(MC[i][j].ToString() + " ");
Console.WriteLine();
}
}
// Вивід вхідних матриць в файли
if (DoOutFile)
{
fs = new FileStream("t1_MB.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(data.MB[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
fs = new FileStream("t1_MC.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(MC[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
}
// Сигнал потокам №2, №3, №4 про завершення вводу(заповнення матриць)
data.PulseAll();
// Копіювання загального ресурсу - матриці MB -- критична секція
data.CopyCR(ref MB);
// Обчислення MAh=MB*MCh
```

```
calc(0, n, ref MB, MatrixType);
// Очікування сигналу від потоків №2, №3, №4 про завершення обчислень
data.WaitAll();

// Вивід вихідної матриці на екран
if (DoOutScreen)
{
Console.WriteLine("MA");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(MA[i][j].ToString() + " ");
Console.WriteLine();
}
}
// Вивід вихідної матриці в файл
if (DoOutFile)
{
fs = new FileStream("t1_MA.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(MA[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
}
}

/// <summary>
/// Функція потоків №2, №3, №4
/// </summary>
/// <param name="arg">параметри потоку</param>
void ThreadCalc(object arg)
{
int n = ((int[])arg)[0];// розмір матриць
int thid = ((int[])arg)[1];// номер потоку (2,3,4)
object[][] MB;// копія MB
int i;// лічильник

MB = new object[n][];
for (i = 0; i < n; i++)
MB[i] = new object[n];

// Очікувати сигналу від потоку №1 про завершення вводу
data.Wait();
// Копіювання загального ресурсу - матриці MB -- критична секція
data.CopyCR(ref MB);
// Обчислення MAh=MB*MCh
calc(thid, n, ref MB, MatrixType);
// Сигнал потоку №1 про завершення обчислень
data.Pulse(thid);
}

/// <summary>
/// Обчислення MAh=MB*MCh залежно від типу
/// </summary>
/// <param name="thid">номер потоку</param>
/// <param name="n">розмір матриць</param>
```

```
/// <param name="_MB">посилання на матрицю MB</param>
/// <param name="t">тип елементів матриць</param>
void calc(int thid, int n, ref object[][] _MB, Type t)
{
int i, j, k;// лічильник

if (t == typeof(int))
{
int s;
for (k = 0; k < n / 4; k++)
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (int)_MB[j][i] * (int)MC[i][thid * (n / 4) + k];
MA[j][thid * (n / 4) + k] = s;
}
}
else if (t == typeof(double))
{
double s;
for (k = 0; k < n / 4; k++)
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (double)_MB[j][i] * (double)MC[i][thid * (n / 4) + k];
MA[j][thid * (n / 4) + k] = s;
}
}
else if (t == typeof(decimal))
{
decimal s;
for (k = 0; k < n / 4; k++)
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (decimal)_MB[j][i] * (decimal)MC[i][thid * (n / 4) + k];
MA[j][thid * (n / 4) + k] = s;
}
}
else if (t == typeof(Complex))
{
Complex s;
for (k = 0; k < n / 4; k++)
for (j = 0; j < n; j++)
{
s = new Complex(0, 0);
for (i = 0; i < n; i++)
s += (Complex)_MB[j][i] * (Complex)MC[i][thid * (n / 4) + k];
MA[j][thid * (n / 4) + k] = s;
}
}
}
}
```

## A.4 Файл Task2.cs

```
using System;
```

```
using System.Threading;
using System.IO;

namespace Tasks
{
 /// <summary>
 /// Клас задачі №2
 /// </summary>
 class Task2
 {
 Data data;// Монітор
 object[][] MA;// вихідна матриця MA
 object[][] MB;// вхідна матриця MC
 bool DoOutScreen;// признак виводу на екран
 bool DoOutFile;// признак виводу в файли
 bool FullRange;// признак заповнення матриць числами максимального значення
 Type MatrixType;// тип елементів матриць

 /// <summary>
 /// Клас монітору
 /// </summary>
 public class Data
 {
 public object[][] MC;// вхідна матриця MC
 public object[][] MX;// вхідна матриця MX
 int n;// розмір матриць
 object sync = new object();// об'єкт синхронізації
 bool[] signal;// сигнали потоків

 /// <summary>
 /// Конструктор монітору
 /// </summary>
 /// <param name="_n">розмір матриць</param>
 public Data(int _n)
 {
 int i;// лічильник

 n = _n;
 MC = new object[n][];
 for (i = 0; i < n; i++)
 MC[i] = new object[n];
 MX = new object[n][];
 for (i = 0; i < n; i++)
 MX[i] = new object[n];
 signal = new bool[4];
 for (i = 0; i < 4; i++)
 signal[i] = false;
 }

 /// <summary>
 /// Очікувати сигналу від потоку №1
 /// </summary>
 public void Wait()
 {
 lock (sync)
 {
 while (!signal[0])
 {
 Monitor.Wait(sync);
 }
 }
 }

 /// <summary>
```

```
/// Очікувати всіх сигналів від потоків №2, №3, №4 і сброс
/// </summary>
public void WaitAll()
{
lock (sync)
{
while (!(signal[1] && signal[2] && signal[3]))
{
Monitor.Wait(sync);
}
signal[1] = false; signal[2] = false; signal[3] = false;
}
}

/// <summary>
/// Сигнал від потоку
/// </summary>
/// <param name="thid">номер потоку</param>
public void Pulse(int thid)
{
lock (sync)
{
signal[thid] = true;
Monitor.Pulse(sync);
}
}

/// <summary>
/// Сигнал від потоку №1 (потокам №2, №3, №4)
/// </summary>
public void PulseAll()
{
lock (sync)
{
signal[0] = true;
Monitor.PulseAll(sync);
}
}

/// <summary>
/// Копіювання матриці MC та MX
/// </summary>
/// <param name="dstMC">посилання на приймача матриці MC</param>
/// <param name="dstMX">посилання на приймача матриці MX</param>
public void CopyMCandMX(ref object[][] dstMC, ref object[][] dstMX)
{
dstMC = (object[][])MC.Clone();
dstMX = (object[][])MX.Clone();
}
}

//---------------------------------------------------------------------
/// <summary>
/// Конструктор задачі
/// </summary>
/// <param name="n">розмір матриць</param>
/// <param name="_DoOutScreen">признак виводу на екран</param>
/// <param name="_DoOutFile">признак виводу в файли</param>
/// <param name="_FullRange">признак заповнення матриць числами максимального
значення</param>
/// <param name="_MatrixType">тип елементів матриць</param>
public Task2(int n, bool _DoOutScreen, bool _DoOutFile, bool _FullRange, Type
_MatrixType)
{
```

```
Thread[] ths = new Thread[4];// список потоків
int i;// лічильник

DoOutScreen = _DoOutScreen;
DoOutFile = _DoOutFile;
FullRange = _FullRange;
MatrixType = _MatrixType;

MA = new object[n][];
MB = new object[n][];
for (i = 0; i < n; i++)
{
MA[i] = new object[n];
MB[i] = new object[n];
}
data = new Data(n);

// Створення і старт потоків №2, №3, №4
for (i = 0; i < 3; i++)
{
ths[i] = new Thread(ThreadCalc);
ths[i].Start(new int[] { n, i + 1 });
}
// Створення і старт потоку №1
ths[3] = new Thread(ThreadInputCalc);
ths[3].Start(n);

for (i = 0; i < 3; i++)
ths[i].Join();
ths[3].Join();
}

/// <summary>
/// Функція потоку №1
/// </summary>
/// <param name="_n">розмір матриць</param>
void ThreadInputCalc(object _n)
{
int i, j;// лічильник
Random rnd = new Random();// об'єкт для генерування випадкових чисел
int n = (int)_n;// розмір матриць
object[][] MC;// копія MC
object[][] MX;// копія MX
FileStream fs;// файловий потік
StreamWriter sw;// записувач в файловий потік

int MaxRndInt;// максимальне ціле
double MaxRndDouble;// максимальне дійсне з плаваючою точкою
double MaxRndDecimal;// максимальне з фіксованою точкою
int MaxRndComplex;// максимальне комплексне

if (FullRange)
{
MaxRndInt = (int)Math.Sqrt(Math.Sqrt(int.MaxValue) / 2000) / 2000;
MaxRndDouble = Math.Sqrt(Math.Sqrt(double.MaxValue) / 2000) / 2000;
MaxRndDecimal = Math.Sqrt(Math.Sqrt((double)decimal.MaxValue) / 2000) / 2000;
MaxRndComplex = (int)Math.Sqrt(Math.Sqrt(int.MaxValue) / 2000 / 2) / 2000 / 2;
}
else
{
MaxRndInt = 10;
MaxRndDouble = 10;
MaxRndDecimal = 10;
MaxRndComplex = 10;
```

```
}

MC = new object[n][];
MX = new object[n][];
for (i = 0; i < n; i++)
{
MC[i] = new object[n];
MX[i] = new object[n];
}

// заповнення вхідних матриць випадковими числами залежно від їх типу
if (MatrixType == typeof(int))
{
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
{
MB[i][j] = rnd.Next(MaxRndInt);
data.MC[i][j] = rnd.Next(MaxRndInt);
data.MX[i][j] = rnd.Next(MaxRndInt);
}
}
}
else if (MatrixType == typeof(double))
{
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
{
MB[i][j] = rnd.NextDouble() * MaxRndDouble;
data.MC[i][j] = rnd.NextDouble() * MaxRndDouble;
data.MX[i][j] = rnd.NextDouble() * MaxRndDouble;
}
}
}
else if (MatrixType == typeof(decimal))
{
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
{
MB[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
data.MC[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
data.MX[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
}
}
}
else if (MatrixType == typeof(Complex))
{
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
{
MB[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
data.MC[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
data.MX[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
}
}
}
// Вивід вхідних матриць на екран
if (DoOutScreen)
{
Console.WriteLine("MB");
for (i = 0; i < n; i++)
```

```
{
for (j = 0; j < n; j++)
Console.Write(MB[i][j].ToString() + " ");
Console.WriteLine();
}
Console.WriteLine("MC");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(data.MC[i][j].ToString() + " ");
Console.WriteLine();
}
Console.WriteLine("MX");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(data.MX[i][j].ToString() + " ");
Console.WriteLine();
}
}
// Вивід вхідних матриць в файли
if (DoOutFile)
{
fs = new FileStream("t2_MB.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(MB[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
fs = new FileStream("t2_MC.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(data.MC[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
fs = new FileStream("t2_MX.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(data.MX[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
}
// Сигнал потокам №2, №3, №4 про завершення вводу(заповнення матриць)
```

```
data.PulseAll();
// Копіювання загального ресурсу - матриці MC та MX -- критична секція
data.CopyMCandMX(ref MC, ref MX);
// Обчислення MAh=(MBh*MC)*MX
calc(0, n, ref MC, ref MX, MatrixType);
// Очікування сигналу від потоків №2, №3, №4 про завершення обчислень
data.WaitAll();

// Вивід вихідної матриці на екран
if (DoOutScreen)
{
Console.WriteLine("MA");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(MA[i][j].ToString() + " ");
Console.WriteLine();
}
}
// Вивід вихідної матриці в файл
if (DoOutFile)
{
fs = new FileStream("t2_MA.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(MA[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
}
}

/// <summary>
/// Функція потоків №2, №3, №4
/// </summary>
/// <param name="arg">параметри потоку</param>
void ThreadCalc(object arg)
{
int n = ((int[])arg)[0];// розмір матриць
int thid = ((int[])arg)[1];// номер потоку (2,3,4)
object[][] MC;// копія MC
object[][] MX;// копія MX
int i;// лічильник

MC = new object[n][];
MX = new object[n][];
for (i = 0; i < n; i++)
{
MC[i] = new object[n];
MX[i] = new object[n];
}

// Очікувати сигналу від потоку №1 про завершення вводу
data.Wait();
// Копіювання загального ресурсу - матриці MC та MX -- критична секція
data.CopyMCandMX(ref MC, ref MX);
// Обчислення MAh=(MBh*MC)*MX
calc(thid, n, ref MC, ref MX, MatrixType);
```

```
// Сигнал потоку №1 про завершення обчислень
data.Pulse(thid);
}

/// <summary>
/// Обчислення MAh=MB*MCh залежно від типу
/// </summary>
/// <param name="thid">номер потоку</param>
/// <param name="n">розмір матриць</param>
/// <param name="_MC">посилання на матрицю MC</param>
/// <param name="_MX">посилання на матрицю MX</param>
/// <param name="t">тип елементів матриць</param>
void calc(int thid, int n, ref object[][] _MC, ref object[][] _MX, Type t)
{
int i, j, k;// лічильник

if (t == typeof(int))
{
int s;
int[] temp_row = new int[n];

for (k = 0; k < n / 4; k++)
{
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (int)MB[thid * (n / 4) + k][i] * (int)_MC[i][j];
temp_row[j] = s;
}
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (int)temp_row[i] * (int)_MX[i][j];
MA[thid * (n / 4) + k][j] = s;
}
}
}
else if (t == typeof(double))
{
double s;
double[] temp_row = new double[n];

for (k = 0; k < n / 4; k++)
{
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (double)MB[thid * (n / 4) + k][i] * (double)_MC[i][j];
temp_row[j] = s;
}
for (j = 0; j < n; j++)
{
s = 0;
for (i = 0; i < n; i++)
s += (double)temp_row[i] * (double)_MX[i][j];
MA[thid * (n / 4) + k][j] = s;
}
}
}
else if (t == typeof(decimal))
{
```

```
 decimal s;
 decimal[] temp_row = new decimal[n];

 for (k = 0; k < n / 4; k++)
 {
 for (j = 0; j < n; j++)
 {
 s = 0;
 for (i = 0; i < n; i++)
 s += (decimal)MB[thid * (n / 4) + k][i] * (decimal)_MC[i][j];
 temp_row[j] = s;
 }
 for (j = 0; j < n; j++)
 {
 s = 0;
 for (i = 0; i < n; i++)
 s += (decimal)temp_row[i] * (decimal)_MX[i][j];
 MA[thid * (n / 4) + k][j] = s;
 }
 }
 }
 else if (t == typeof(Complex))
 {
 Complex s;
 Complex[] temp_row = new Complex[n];

 for (k = 0; k < n / 4; k++)
 {
 for (j = 0; j < n; j++)
 {
 s = new Complex(0, 0);
 for (i = 0; i < n; i++)
 s += (Complex)MB[thid * (n / 4) + k][i] * (Complex)_MC[i][j];
 temp_row[j] = s;
 }
 for (j = 0; j < n; j++)
 {
 s = new Complex(0, 0);
 for (i = 0; i < n; i++)
 s += (Complex)temp_row[i] * (Complex)_MX[i][j];
 MA[thid * (n / 4) + k][j] = s;
 }
 }
 }
 }
 }
}
```

## A.5 Файл Task3.cs

```
using System;
using System.Threading;
using System.IO;

namespace Tasks
{
 /// <summary>
 /// Клас задачі №3
 /// </summary>
 class Task3
 {
```

```
Data data;// Монітор
object[][] MX;// вхідна матриця MX
object[] A;// вихідна матриця MA
bool DoOutScreen;// признак виводу на екран
bool DoOutFile;// признак виводу в файли
bool FullRange;// признак заповнення матриць числами максимального значення
Type MatrixType;// тип елементів матриць

/// <summary>
/// Клас монітору
/// </summary>
public class Data
{
public object[][] MC;// вхідна матриця MC
public object[] B;// вхідний вектор B
int n;// розмір матриць
object sync = new object();// об'єкт синхронізації
bool[] signal;// сигнали потоків

/// <summary>
/// Конструктор монітору
/// </summary>
/// <param name="_n">розмір матриць</param>
public Data(int _n)
{
int i;// лічильник

n = _n;
MC = new object[n][];
for (i = 0; i < n; i++)
MC[i] = new object[n];
B = new object[n];
signal = new bool[4];
for (i = 0; i < 4; i++)
signal[i] = false;
}

/// <summary>
/// Очікувати сигналу від потоку №1
/// </summary>
public void Wait()
{
lock (sync)
{
while (!signal[0])
{
Monitor.Wait(sync);
}
}
}

/// <summary>
/// Очікувати всіх сигналів від потоків №2, №3, №4 і сброс
/// </summary>
public void WaitAll()
{
lock (sync)
{
while (!(signal[1] && signal[2] && signal[3]))
{
Monitor.Wait(sync);
}
signal[1] = false; signal[2] = false; signal[3] = false;
}
```

```
}

/// <summary>
/// Сигнал від потоку
/// </summary>
/// <param name="thid">номер потоку</param>
public void Pulse(int thid)
{
lock (sync)
{
signal[thid] = true;
Monitor.Pulse(sync);
}
}

/// <summary>
/// Сигнал від потоку №1 (потокам №2, №3, №4)
/// </summary>
public void PulseAll()
{
lock (sync)
{
signal[0] = true;
Monitor.PulseAll(sync);
}
}

/// <summary>
/// Копіювання вектора B матриці MC
/// </summary>
/// <param name="dstB">посилання на приймача вектора B</param>
/// <param name="dstMC">посилання на приймача матриці MC</param>
public void CopyBandMC(ref object[] dstB, ref object[][] dstMC)
{
dstMC = (object[][])MC.Clone();
dstB = (object[])B.Clone();
}
}

//---------------------------------------------------------------------
/// <summary>
/// Конструктор задачі
/// </summary>
/// <param name="n">розмір матриць</param>
/// <param name="_DoOutScreen">признак виводу на екран</param>
/// <param name="_DoOutFile">признак виводу в файли</param>
/// <param name="_FullRange">признак заповнення матриць числами максимального
значення</param>
/// <param name="_MatrixType">тип елементів матриць</param>
public Task3(int n, bool _DoOutScreen, bool _DoOutFile, bool _FullRange, Type
_MatrixType)
{
Thread[] ths = new Thread[4];// список потоків
int i;// лічильник

DoOutScreen = _DoOutScreen;
DoOutFile = _DoOutFile;
FullRange = _FullRange;
MatrixType = _MatrixType;

A = new object[n];
MX = new object[n][];
for (i = 0; i < n; i++)
MX[i] = new object[n];
```

```
data = new Data(n);

// Створення і старт потоків №2, №3, №4
for (i = 0; i < 3; i++)
{
ths[i] = new Thread(ThreadCalc);
ths[i].Start(new int[] { n, i + 1 });
}
// Створення і старт потоку №1
ths[3] = new Thread(ThreadInputCalc);
ths[3].Start(n);

for (i = 0; i < 3; i++)
ths[i].Join();
ths[3].Join();
}

/// <summary>
/// Функція потоку №1
/// </summary>
/// <param name="_n">розмір матриць</param>
void ThreadInputCalc(object _n)
{
int i, j;// лічильник
Random rnd = new Random();// об'єкт для генерування випадкових чисел
int n = (int)_n;// розмір матриць
object[][] MC;// копія MC
object[] B;// копія B
FileStream fs;// файловий потік
StreamWriter sw;// записувач в файловий потік

int MaxRndInt;// максимальне ціле
double MaxRndDouble;// максимальне дійсне з плаваючою точкою
double MaxRndDecimal;// максимальне з фіксованою точкою
int MaxRndComplex;// максимальне комплексне

if (FullRange)
{
MaxRndInt = (int)Math.Sqrt(Math.Sqrt(int.MaxValue) / 2000) / 2000;
MaxRndDouble = Math.Sqrt(Math.Sqrt(double.MaxValue) / 2000) / 2000;
MaxRndDecimal = Math.Sqrt(Math.Sqrt((double)decimal.MaxValue) / 2000) / 2000;
MaxRndComplex = (int)Math.Sqrt(Math.Sqrt(int.MaxValue) / 2000 / 2) / 2000 / 2;
}
else
{
MaxRndInt = 10;
MaxRndDouble = 10;
MaxRndDecimal = 10;
MaxRndComplex = 10;
}

MC = new object[n][];
B = new object[n];
for (i = 0; i < n; i++)
MC[i] = new object[n];

// заповнення вхідних матриць випадковими числами залежно від їх типу
if (MatrixType == typeof(int))
{
for (i = 0; i < n; i++)
{
data.B[i] = rnd.Next(MaxRndInt);
for (j = 0; j < n; j++)
{
```

```
data.MC[i][j] = rnd.Next(MaxRndInt);
MX[i][j] = rnd.Next(MaxRndInt);
}
}
}
else if (MatrixType == typeof(double))
{
for (i = 0; i < n; i++)
{
data.B[i] = rnd.NextDouble() * MaxRndDouble;
for (j = 0; j < n; j++)
{
data.MC[i][j] = rnd.NextDouble() * MaxRndDouble;
MX[i][j] = rnd.NextDouble() * MaxRndDouble;
}
}
}
else if (MatrixType == typeof(decimal))
{
for (i = 0; i < n; i++)
{
data.B[i] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
for (j = 0; j < n; j++)
{
data.MC[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
MX[i][j] = (decimal)(rnd.NextDouble() * MaxRndDecimal);
}
}
}
else if (MatrixType == typeof(Complex))
{
for (i = 0; i < n; i++)
{
data.B[i] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
for (j = 0; j < n; j++)
{
data.MC[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
MX[i][j] = new Complex(rnd.Next(MaxRndComplex), rnd.Next(MaxRndComplex));
}
}
}
// Вивід вхідних матриць на екран
if (DoOutScreen)
{
Console.WriteLine("B");
for (i = 0; i < n; i++)
{
Console.Write(data.B[i].ToString() + " ");
}
Console.WriteLine();
Console.WriteLine("MC");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(data.MC[i][j].ToString() + " ");
Console.WriteLine();
}
Console.WriteLine("MX");
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
Console.Write(MX[i][j].ToString() + " ");
Console.WriteLine();
}
```

```
}
// Вивід вхідних матриць в файли
if (DoOutFile)
{
fs = new FileStream("t3_B.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
sw.Write(data.B[i].ToString() + " ");
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
fs = new FileStream("t3_MC.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(data.MC[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
fs = new FileStream("t3_MX.txt", FileMode.Create);
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
for (j = 0; j < n; j++)
sw.Write(MX[i][j].ToString() + " ");
sw.WriteLine();
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
}
// Сигнал потокам №2, №3, №4 про завершення вводу(заповнення матриць)
data.PulseAll();
// Копіювання загального ресурсу - вектора B та матриці MC -- критична секція
data.CopyBandMC(ref B, ref MC);
// Обчислення Ah=B*(MC*MXh)
calc(0, n, ref B, ref MC, MatrixType);
// Очікування сигналу від потоків №2, №3, №4 про завершення обчислень
data.WaitAll();

// Вивід вихідної матриці на екран
if (DoOutScreen)
{
Console.WriteLine("A");
for (i = 0; i < n; i++)
{
Console.Write(A[i].ToString() + " ");
}
}
// Вивід вихідної матриці в файл
if (DoOutFile)
{
fs = new FileStream("t3_A.txt", FileMode.Create);
```

```
sw = new StreamWriter(fs);
for (i = 0; i < n; i++)
{
sw.Write(A[i].ToString() + " ");
}
sw.Flush();
sw.Close();
fs.Close();
fs.Dispose();
sw.Dispose();
}
}

/// <summary>
/// Функція потоків №2, №3, №4
/// </summary>
/// <param name="arg">параметри потоку</param>
void ThreadCalc(object arg)
{
int n = ((int[])arg)[0];// розмір матриць
int thid = ((int[])arg)[1];// номер потоку (2,3,4)
object[][] MC;// копія MC
object[] B;// копія B
int i;// лічильник

MC = new object[n][];
B = new object[n];
for (i = 0; i < n; i++)
MC[i] = new object[n];

// Очікувати сигналу від потоку №1 про завершення вводу
data.Wait();
// Копіювання загального ресурсу - вектора B та матриці MC -- критична секція
data.CopyBandMC(ref B, ref MC);
// Обчислення Ah=B*(MC*MXh)
calc(thid, n, ref B, ref MC, MatrixType);
// Сигнал потоку №1 про завершення обчислень
data.Pulse(thid);
}

/// <summary>
/// Обчислення MAh=MB*MCh залежно від типу
/// </summary>
/// <param name="thid">номер потоку</param>
/// <param name="n">розмір матриць</param>
/// <param name="_B">посилання на матрицю B</param>
/// <param name="_MC">посилання на матрицю MC</param>
/// <param name="t">тип елементів матриць</param>
void calc(int thid, int n, ref object[] _B, ref object[][] _MC, Type t)
{
int i, j, k;// лічильник

if (t == typeof(int))
{
int scx;
int sa;
for (k = 0; k < n / 4; k++)
{
sa = 0;
for (j = 0; j < n; j++)
{
scx = 0;
for (i = 0; i < n; i++)
scx += (int)_MC[j][i] * (int)MX[i][thid * (n / 4) + k];
```

```
sa += (int)_B[j] * scx;
}
A[thid * (n / 4) + k] = sa;
}
}
else if (t == typeof(double))
{
double scx;
double sa;
for (k = 0; k < n / 4; k++)
{
sa = 0;
for (j = 0; j < n; j++)
{
scx = 0;
for (i = 0; i < n; i++)
scx += (double)_MC[j][i] * (double)MX[i][thid * (n / 4) + k];
sa += (double)_B[j] * scx;
}
A[thid * (n / 4) + k] = sa;
}
}
else if (t == typeof(decimal))
{
decimal scx;
decimal sa;
for (k = 0; k < n / 4; k++)
{
sa = 0;
for (j = 0; j < n; j++)
{
scx = 0;
for (i = 0; i < n; i++)
scx += (decimal)_MC[j][i] * (decimal)MX[i][thid * (n / 4) + k];
sa += (decimal)_B[j] * scx;
}
A[thid * (n / 4) + k] = sa;
}
}
else if (t == typeof(Complex))
{
Complex scx;
Complex sa;
for (k = 0; k < n / 4; k++)
{
sa = new Complex(0, 0);
for (j = 0; j < n; j++)
{
scx = new Complex(0, 0);
for (i = 0; i < n; i++)
scx += (Complex)_MC[j][i] * (Complex)MX[i][thid * (n / 4) + k];
sa += (Complex)_B[j] * scx;
}
A[thid * (n / 4) + k] = sa;
}
}
}
}
}
```