НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО	**ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ**
Декан факультету	Завідувач кафедри
Інформаційних технологій	Комп'ютерних систем, мереж та кібербезпеки
_____/ Ігор Болбот /	_____/ Дмитро Касаткін /
підпис ПІБ, вчене звання і ступінь	підпис ПІБ, вчене звання і ступінь
«___»_____20__ р.	«___»_____20__ р.

ВИПУСКНА БАКАЛАВРСЬКА РОБОТА
(ДИПЛОМНИЙ ПРОЕКТ БАКАЛАВРА)

на тему: Розробка комплексу програмних та апаратних засобів для лабораторних робіт з курсу «Паралельні та розподілені обчислення»

Спеціальність (напрям підготовки) F7 «Комп'ютерна інженерія»

15.04 - БКР.3072 "С" 26.12.10.28.ПЗ

Гарант освітньої програми
канд. фіз.-мат. наук, доцент _____ **Євгеній НІКІТЕНКО**
(підпис)

Керівник випускної бакалаврської роботи
(Керівник дипломного проекту бакалавра)
к.т.н., доцент _____ Смолій В.В.
(науковий ступінь та вчене звання) (підпис) (ПІБ)

Виконав _____ Петрієнко Т.А.
(підпис) (ПІБ студента)

КИЇВ-2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

«ЗАТВЕРДЖУЮ»
завідувач кафедри
комп'ютерних систем, мереж та кібербезпеки
_____/ канд. пед. наук, доцент Дмитро Касаткін /
підпис ПІБ, вчене звання і ступінь
«__» _____ 20__ р.

З А В Д А Н Н Я
на виконання випускної бакалаврської роботи студенту
(на виконання дипломного проекту бакалавра студенту)

Петрієнко Тетяні Андріївні
(прізвище, ім'я, по батькові)
Спеціальність (напрям підготовки) комп'ютерна інженерія
Тема випускної бакалаврської роботи (дипломного проекту бакалавра) Розробка комплексу програмних та апаратних засобів для лабораторних робіт з курсу «Паралельні та розподілені обчислення»

керівник проекту (роботи) Смолій Віктор Вікторович к.т.н. доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
Затверджена наказом ректора НУБіП України від "10"_12_2025р. №3072 «С»
Термін подання завершеної роботи (проекту) на кафедру __2026,_02,_06
(рік, місяць, число)
Вихідні дані до випускної бакалаврської роботи (дипломного проекту бакалавра) Мова C/C++. Інструменти: VS Code, компілятор GCC (MSYS2), пакет Microsoft MPI, графічний редактор Draw.io.

Перелік питань, які потрібно розробити:
Аналіз предметної області паралелізму, обгрунтування архітектури багатоядерних CPU та SIMD, налаштування середовища розробника й розподіленого кластера MS-MPI, проектування структури лабораторного практикуму та цифрових Google-тестів, експериментальне тестування, часовий аудит та верифікація за законом Амдала.
Перелік графічних документів (за потреби) _____

Дата видачі завдання "_10_" _12_2025 р.
Керівник випускної бакалаврської роботи
(Керівник дипломного проекту бакалавра) _____ Смолій В.В.
(підпис) (прізвище та ініціали)
Завдання прийняла до виконання _____ Петрієнко Т.А.
(підпис) (прізвище та ініціали студента)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз предметної області	04.02.2026 р.	Виконано
2	Проєктування комплексу	15.03.2026 р.	Виконано
3	Розробка лабораторних	10.04.2026 р.	Виконано
4	Тестування	01.05.2026 р.	Виконано
5	Оформлення пояснювальної записки	25.05.2026 р.	Виконано
6	Оформлення графічного матеріалу	25.05.2026 р.	Виконано

Студентка _____ **Т.А. Петрієнко**

(підпис) (ініціали та прізвище)

Керівник проекту (роботи) _____ **В.В. Смолій**

(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 64 сторінок, 13 рисунків, 15 джерел.

ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ЛАБОРАТОРНИЙ ПРАКТИКУМ, OPENMP, MICROSOFT MPI, SIMD-РОЗШИРЕННЯ, КОМПІЛЯТОР GCC, ПРИСКОРЕННЯ, ЕФЕКТИВНІСТЬ, ЗАКОН АМДАЛА

Об’єктом дослідження є процес підготовки ІТ-фахівців з розробки паралельного та розподіленого програмного забезпечення.

Метою роботи є розробка локального практикуму з 7 лабораторних робіт і тестів у Google Forms на базі ПК. Проєкт складається з чотирьох розділів. У першому розділі розглянуто теорію паралелізму, класифікацію Флінна та поставлено задачу проєктування комплексу. Другий розділ присвячено обґрунтуванню архітектури CPU, кеш-пам'яті та конфігурації розподіленого кластера MS-MPI.

У третьому розділі описано методичну структуру 7 лабораторних робіт із прикладами коду OpenMP, MPI, SIMD та організацію модульних тестів. Четвертий розділ містить результати експериментального тестування, часового аудиту та аналізу ефективності обчислень за законом Амдала. У результаті роботи спроєктовано та впроваджено навчальний практикум.

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким збільшенням обсягів даних, які потребують обробки в режимі реального часу, а також ускладненням математичних моделей, що використовуються у наукових, інженерних та економічних обчисленнях. Традиційний підхід до підвищення продуктивності обчислювальних систем шляхом нарощування тактової частоти одиничних процесорів досяг своєї фізичної межі через тепловиділення та обмеження напівпровідникових технологій. Як наслідок, єдиним ефективним шляхом подальшого розвитку обчислювальної техніки став перехід до багатоядерних і розподілених архітектур.

У цих умовах розробка програмного забезпечення вимагає від ІТ-фахівців фундаментальних знань у галузі паралельного обчислення та архітектури сучасних процесорів. Проте вивчення дисципліни «Паралельні та розподілені обчислення» студентами вищих навчальних закладів часто ускладнюється відсутністю адаптованого, гнучкого та наочного лабораторного практикуму, який би охоплював різні рівні паралелізму — від низькорівневих регістрових інструкцій CPU до розподілених кластерних систем. Існуючі методичні рішення або орієнтовані на застарілі стандарти, або вимагають використання дорогого залізничного обладнання суперкомп'ютерів, що обмежує самостійну роботу студентів.

Таким чином, розробка інтегрованого комплексу програмних та апаратних засобів, який дозволяє розгортати навчальні паралельні системи та віртуальні обчислювальні кластери на базі стандартних багатоядерних робочих станцій під

керуванням операційної системи Windows, є актуальною науково-методичною та інженерною задачею.

Метою бакалаврської кваліфікаційної роботи є розробка та впровадження ефективного програмно-апаратного комплексу для лабораторних робіт з курсу «Паралельні та розподілені обчислення», який забезпечує наочне вивчення сучасних технологій багатопотокового та векторного програмування.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Проаналізувати архітектурні особливості паралельних систем із спільною та розподіленою пам'яттю, а також існуючі стандарти розпаралелювання обчислень.
2. Проектувати та налаштувати апаратне середовище обчислювальної станції, включаючи конфігурацію векторних регістрів та розгортання локального віртуального кластера за допомогою технології Microsoft MPI.
3. Розробити кодову базу та архітектуру мінімізованих шаблонів програм для 7 лабораторних робіт, розподілених за трьома тематичними модулями.
4. Створити унікальні системи індивідуальних завдань для забезпечення самостійності виконання робіт студентами.
5. Провести експериментальне тестування розробленого комплексу, виконати верифікацію математичних результатів та проаналізувати практичні показники прискорення обчислень у порівнянні з теоретичними обмеженнями закону Амдала.

Об’єктом дослідження є процеси паралельної та розподіленої обробки даних на багатоядерних обчислювальних архітектурах та кластерних системах.

Предметом дослідження є програмні та апаратні засоби, стандарти (OpenMP, MPI), векторні розширення (MMX, Float SIMD) та методичні структури, що використовуються для реалізації паралельних алгоритмів чисельного інтегрування та геометричних трансформацій.

Для вирішення поставлених завдань використано методи системного аналізу, теорію паралельних обчислень, принципи об'єктно-орієнтованого програмування на мові C++, методи чисельного інтегрування (метод прямокутників), а також методи апаратного профілювання та часового моніторингу програмних систем.

Сформовано цілісний підхід до побудови навчального лабораторного практикуму, який, на відміну від аналогів, поєднує вивчення трьох ієрархічних рівнів паралелізму (векторного, багатопотокового та розподіленого) в межах єдиного уніфікованого інструментального середовища розробника на базі операційної системи Windows без потреби у залученні сторонніх серверних ресурсів.

Практичне значення отриманих результатів полягає у створенні готового до безпосереднього впровадження у навчальний процес програмно-методичного комплексу, що складається з оптимізованого інструментального середовища компіляції MSYS2/GCC, уніфікованих і безпечних шаблонів програм із коректним виведенням інформаційних повідомлень, а також розширеного фонду індивідуальних завдань. Розроблений комплекс забезпечує високий рівень автономності під час самостійної роботи студентів, надає можливість гнучкого масштабування завдань під різні рівні складності та дозволяє виконувати ресурсомісткі інженерні дослідження на локальних багатоядерних обчислювальних станціях без залучення стороннього серверного обладнання.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Еволюція багатопроцесорних архітектур та актуальність вивчення паралельних обчислень

Сучасний розвиток обчислювальної техніки тривалий час базувався на постійному збільшенні кількості транзисторів на кристалі процесора та пропорційному зростанні його тактової частоти. Завдяки цьому звичайні послідовні програми автоматично працювали швидше на кожному новому поколінні комп'ютерів без будь-яких змін у їхньому сирцевому коді.

Проте на початку 2000-х років розробники мікропроцесорів зіткнулися з серйозними фізичними обмеженнями кремнієвих технологій. Головною проблемою став так званий тепловий бар'єр. Подальше підвищення тактової частоти одного обчислювального ядра призвело до такого високого виділення тепла, що охолоджувати процесор стандартними методами стало неможливо. Крім того, виник архітектурний розрив між швидкістю роботи процесора та пропускною здатністю оперативної пам'яті: ядро часто простоювало, очікуючи на надходження нових даних з шини зв'язку.

Для подолання цих обмежень виробники заліза змінили підхід до проектування: замість намагань створити один надшвидкий процесор вони перейшли до створення багатоядерних і багатопроцесорних систем. Подальше зростання потужності комп'ютерів стало можливим лише завдяки розподілу завдань між кількома ядрами, тобто за рахунок паралелізму. Сучасні процесори перетворилися на складні системи, які містять декілька обчислювальних ядер, вбудовані векторні блоки та швидкі внутрішні канали зв'язку.

Ці зміни в апаратній частині докорінно змінили вимоги до створення програмного забезпечення. Звичайні послідовні алгоритми більше не отримують автоматичного приросту швидкості. Якщо програма написана в один потік, вона здатна використовувати лише невелику частину загальної потужності сучасного процесора, тоді як інші ядра будуть простоювати.

Саме тому вивчення паралельних та розподілених обчислень є вкрай важливим для підготовки сучасних ІТ-фахівців. Програміст повинен розуміти принципи роботи апаратної синхронізації, механізми спільного використання пам'яті та організації зв'язку між процесами. Без цих знань неможливо створити ефективне програмне забезпечення, оскільки всі сучасні платформи — від графічних рушіїв та систем візуалізації до великих аналітичних комплексів і хмарних сервісів — будуються виключно на базі паралельних архітектур.

Таким чином, створення та впровадження практичних засобів паралелізму у навчальний процес дозволяє студентам отримати реальні навички, які повністю відповідають вимогам сучасної обчислювальної індустрії.

1.2 Класифікація паралельних обчислювальних систем

Для ефективного проектування та програмування паралельних систем необхідно чітко розуміти їхню внутрішню структуру. У комп'ютерній інженерії існують різні підходи до класифікації таких систем, проте найбільш фундаментальними є поділ за способом організації оперативної пам'яті та класифікація за структурою потоків команд і даних.

Класифікація за архітектурою пам'яті

За способом взаємодії обчислювальних ядер з оперативною пам'яттю всі сучасні системи поділяють на два великі класи:

1. Архітектури зі спільною пам'яттю (Shared Memory) в таких системах усі обчислювальні ядра підключені до єдиного загального

адресного простору оперативної пам'яті. Це означає, що будь-яке ядро процесора може прямо зчитувати або змінювати дані, які знаходяться в RAM. Головною перевагою такої архітектури є простота обміну даними між потоками, адже їм не потрібно пересилати повідомлення один одному — достатньо записати інформацію в спільну змінну. Саме за цим принципом працюють сучасні багатоядерні процесори персональних комп'ютерів, а базовим стандартом програмування для них є технологія OpenMP.

2. Архітектури з розподіленою пам'яттю (Distributed Memory) у системах з розподіленою пам'яттю кожен обчислювальний вузол є повністю ізольованим. Він має свій власний процесор та свою власну оперативну пам'ять. Вузли не можуть напряму заглядати в пам'ять один одного. Будь-яка взаємодія та обмін інформацією між ними відбуваються виключно шляхом явного передавання пакетів даних (повідомлень) через мережеві канали зв'язку або внутрішні системні шини. Такі архітектури використовуються для створення обчислювальних кластерів та суперкомп'ютерів, а головним інструментом розробки програм для них є інтерфейс MPI (Message Passing Interface).

Класифікація Флінна

У 1966 році Майкл Флінн запропонував класифікацію обчислювальних систем, яка базується на понятті двох потоків: потоку команд (інструкцій, які виконує процесор) та потоку даних (інформації, яку ці команди обробляють). Згідно з цією схемою, виділяють чотири основні класи архітектур:

- SISD (Single Instruction, Single Data) – одна інструкція, одні дані. Це класичний послідовний комп'ютер. В кожен момент часу процесор виконує одну команду над одним елементом даних. За такою схемою працювали старі одноядерні процесори.
- SIMD (Single Instruction, Multiple Data) – одна інструкція, множина даних. У таких системах один керуючий блок видає одну команду, але вона виконується одночасно над цілим набором різних даних. Це векторні обчислення. На рівні заліза сучасних CPU цей клас

представлений апаратними розширеннями (такими як MMX, SSE, AVX). Вони дозволяють, наприклад, за один такт процесора паралельно перемножити чотири координати тривимірного вектора, що є вкрай важливим для комп'ютерної графіки та обробки сигналів.

- MISD (Multiple Instruction, Single Data) – множина інструкцій, одні дані. Теоретичний клас, де декілька різних команд одночасно обробляють один і той самий елемент даних. На практиці така схема майже не використовується, окрім деяких специфічних систем із високим рівнем відмовостійкості (наприклад, бортові комп'ютери космічних апаратів, де кілька процесорів паралельно перевіряють один результат).
- MIMD (Multiple Instruction, Multiple Data) – множина інструкцій, множина даних: Найбільш поширений і потужний клас сучасних паралельних систем. Тут кожне ядро або обчислювальний вузол може виконувати свої власні незалежні команди над своїми власними даними. До цього класу відносяться як звичайні багатоядерні процесори (де ядра виконують різні потоки ОС), так і великі розподілені кластерні системи.

Кожен із зазначених класів архітектур має власну специфіку циркуляції інформаційних потоків між керуючими та виконавчими вузлами системи. Для наочного відображення та аналізу розбіжностей у топологіях побудови цих систем доцільно розглянути їхню узагальнену структурну схему (рис. 1.1).

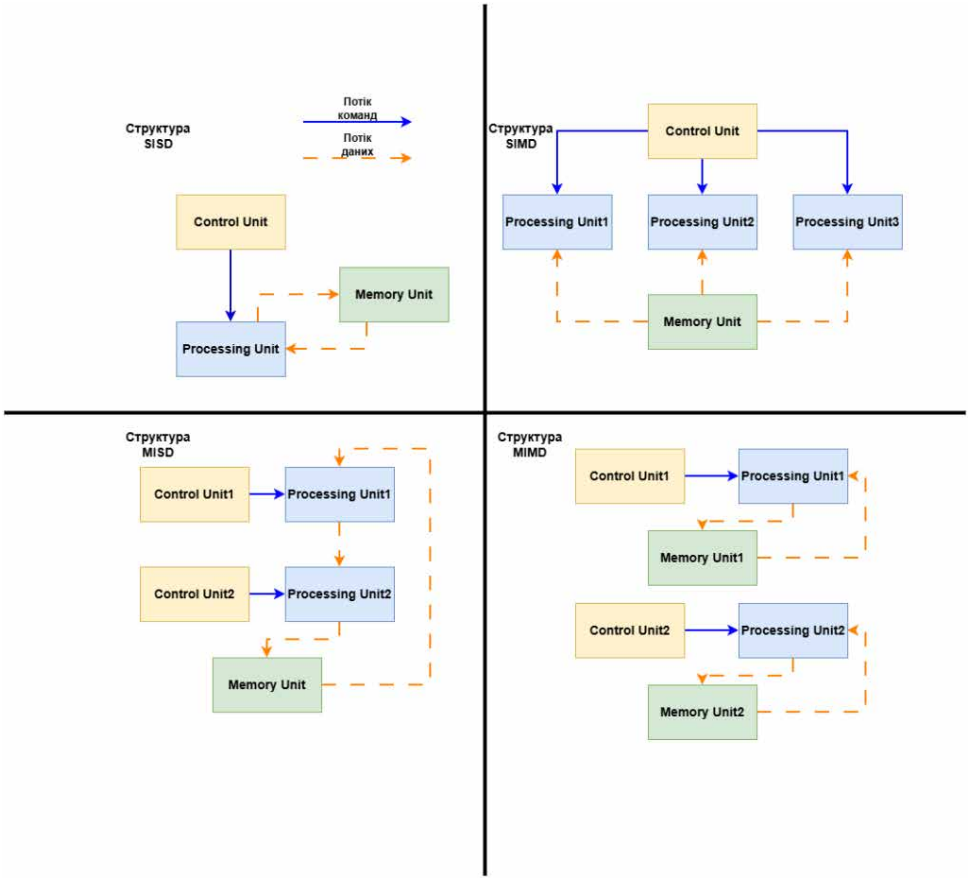

Рисунок 1.1. Структурна схема класифікації обчислювальних архітектур за Флінном

Розуміння цих класифікацій дозволяє правильно обирати програмні засоби під час розробки програмного комплексу та чітко розділяти завдання лабораторного практикуму на рівні векторної обробки (SIMD), багатопотоковості (MIMD зі спільною пам'яттю) та кластерної взаємодії (MIMD з розподіленою пам'яттю).

1.3 Аналіз існуючих програмних інтерфейсів та стандартів паралелізму

Реалізація паралельних обчислень на практиці потребує використання спеціалізованих інструментів, які дозволяють програмісту керувати апаратними ресурсами комп'ютера. Залежно від рівня паралелізму та архітектури пам'яті обчислювальної системи, виділяють кілька основних стандартів та інтерфейсів розробки програмного забезпечення.

SIMD-розширення компіляторів (низькорівневий паралелізм)

На найнижчому апаратному рівні, безпосередньо всередині одного процесорного ядра, паралелізм реалізується за допомогою векторних інструкцій. Сучасні компілятори (зокрема GCC) надають розробнику можливість використовувати вбудовані векторні типи даних та атрибути, такі як __attribute__ ((vector_size (N))).

Цей підхід дозволяє задіяти спеціальні регістри процесора (наприклад, 64-бітні MMX або 128-бітні SSE). Замість написання складного коду на асемблері, програміст може використовувати звичні арифметичні оператори мови C++ для одночасної обробки масивів чисел. Головною особливістю векторних розширень є те, що паралелізм виконується на рівні заліза за один такт процесора. Це робить їх незамінними для задач комп'ютерної геометрії, де потрібно миттєво обробляти координати, та при виконанні покомпонентних операцій над невеликими блоками даних.

Технологія OpenMP (багатопотоковість на базі спільної пам'яті)

Для розпаралелювання обчислень у межах однієї багатоядерної комп'ютерної системи найбільш поширеним стандартом є OpenMP (Open Multi-Processing). Він базується на моделі розгалуження-об'єднання (Fork-Join) та

використовує спеціальні директиви компілятора, які додаються безпосередньо в сирцевий код програми (наприклад, #pragma omp parallel).

Основні переваги OpenMP:

- Простота інтеграції: розробнику не потрібно вручну створювати потоки операційної системи чи керувати їхнім життєвим циклом — компілятор робить це автоматично.
- Робота в спільному адресному просторі: усі потоки мають прямий доступ до загальної оперативної пам'яті, що спрощує обмін даними.

Проте спільна пам'ять створює проблему стану гонити за даними (Data Race), коли кілька ядер намагаються одночасно змінити одну змінну. Для вирішення цієї проблеми в OpenMP використовуються спеціальні механізми синхронізації, такі як критичні секції (#pragma omp critical) та операції паралельної редукції (reduction). Ця технологія є ідеальною для розпаралелювання важких циклів чисельного інтегрування на персональних комп'ютерах.

Інтерфейс передачі повідомлень MPI (розподілені обчислення)

Коли обчислювальна задача виходить за межі потужності одного комп'ютера, виникає потреба у використанні розподілених систем (кластерів), де вузли не мають спільної пам'яті. Головним стандартом для таких архітектур є MPI (Message Passing Interface).

В MPI програма запускається як набір абсолютно незалежних процесів, кожен з яких має власну ізольовану пам'ять. Взаємодія між ними реалізується через явне відправлення та прийом пакетів даних. Інтерфейс MPI надає два основні типи комунікацій:

1. Двоточковий обмін (Point-to-Point): взаємодія між двома конкретними вузлами за допомогою функцій MPI_Send та MPI_Recv.

2. Колективні операції (Collective Communications): синхронна взаємодія між усіма процесами кластера одночасно. Сюди відносяться широкомовна розсилка (MPI_Bcast), глобальне згортання результатів (MPI_Reduce), а також функції автоматичного нарізання та збирання масивів (MPI_Scatter / MPI_Gather).

Хоча програмування на MPI є складнішим через необхідність вручну контролювати розподіл даних та уникати взаємних блокувань (Deadlocks), цей стандарт має абсолютну масштабованість. Він дозволяє об'єднувати в єдину мережу тисячі процесорів.

Таким чином, аналіз показує, що комплексне вивчення паралельних обчислень вимагає послідовного практичного освоєння всіх трьох інструментів (SIMD, OpenMP, MPI), оскільки кожен із них вирішує свій специфічний клас задач на різних рівнях архітектури ЕОМ.

Для систематизації результатів проведеного аналізу та наочного порівняння техніко-експлуатаційних характеристик існуючих програмних інтерфейсів доцільно узагальнити їхні ключові параметри у вигляді порівняльної структури (табл. 1.1).

Таблиця 1.1

Порівняльний аналіз технологій та стандартів паралельних обчислень

Критерій порівняння	SIMD-розширення (MMX/SSE)	Технологія OpenMP	Інтерфейс MPI
Цільова архітектура	Покомпонентні блоки одного ядра	Багатоядерні CPU (спільна пам'ять)	Обчислювальні кластери (розподідена пам'ять)
Рівень виконання	Апаратні регістри процесора	Потоки операційної системи	Ізольовані системні процеси
Основний механізм	Векторні типи компілятора	Директиви прагма (#pragma)	Бібліотечні функції зв'язку
Спосіб обміну даними	Через упаковані регістри	Через спільні змінні в RAM	Через мережеві повідомлення
Основні ризики коду	Переповнення розрядності	Гонитва за даними (Data Race)	Взаємне блокування (Deadlock)

1.4 Проблема підготовки ІТ-фахівців: аналіз матеріально-технічного та методичного забезпечення лабораторних практикумів

Перехід сучасної ІТ-індустрії до багатоядерних та розподілених архітектур ставить нові вимоги до кваліфікації випускників закладів вищої освіти. Програмування для паралельних систем перетворилося на базову навичку, необхідну для створення конкурентоспроможного програмного забезпечення.

Проте під час побудови відповідних лабораторних практикумів університети стикаються з низкою матеріально-технічних та методичних проблем.

Першою проблемою є висока вартість та складність обслуговування реального обладнання для розподілених обчислень. Традиційні лабораторні роботи з вивчення технології MPI часто вимагають використання фізичних серверних кластерів. Організація таких лабораторій в університеті пов'язана зі значними фінансовими витратами та залученням спеціалізованого персоналу для адміністрування складних мереж.

Альтернативне використання комерційних хмарних платформ (AWS, Microsoft Azure тощо) також має недоліки: воно потребує постійної оплати процесорного часу, створення безлічі студентських акаунтів та залежить від стабільності підключення до Інтернету, що обмежує самостійну роботу студентів у позааудиторний час.

Другою проблемою є методичний розрив у навчальних програмах. Більшість практикумів пропонують завдання, які охоплюють лише один ізольований рівень паралелізму – наприклад, тільки OpenMP або тільки MPI. При цьому поза увагою залишається низькорівневий апаратний паралелізм векторних SIMD-регістрів процесора (MMX, SSE). Крім того, чинні навчальні матеріали часто використовують перевантажені шаблони коду, де студенти витрачають час на розбір сторонніх інтерфейсів замість вивчення фундаментальних алгоритмів.

Для вирішення цих проблем виникає необхідність розробки уніфікованого програмно-апаратного комплексу, який дозволяє вивчати всі рівні паралелізму (векторний, багатопотоковий та розподілений) на базі звичайних багатоядерних персональних комп'ютерів під керуванням операційної системи Windows. Використання локального середовища MSYS2/GCC та конфігурація віртуального розподіленого кластера Microsoft MPI на одній робочій станції

дозволяє повністю відмовитися від дорогого серверного обладнання та хмарних сервісів, забезпечуючи студентам автономність та високу наочність навчання.

Для порівняння існуючих матеріально-технічних підходів та обґрунтування доцільності впровадження розробленого рішення сформовано аналітичну структуру (табл. 1.2).

Таблиця 1.2

Порівняльний аналіз матеріально-технічного забезпечення практикумів

Критерій порівняння	Фізичний кластер університету	Комерційні хмарні платформи	Розроблений локальний комплекс
Фінансові витрати	Дуже високі (закупівля серверів)	Постійні (оплата за процесорний час)	Відсутні (використовуються наявні ПК)
Складність налаштування	Висока (потребує системного адміна)	Середня (налаштування хмарних мереж)	Низька (швидке розгортання інсталяторів)
Залежність від Інтернету	Відсутня (локальна мережа)	Абсолютна (без мережі робота неможлива)	Відсутня (обчислення відбуваються локально)
Автономність студента	Обмежена доступом до лабораторії	Обмежена балансом рахунку аккаунта	Повна (можна виконувати вдома на власному ПК)
Охоплення технологій	Переважно тільки MPI	Залежить від конфігурації	SIMD, OpenMP та MPI в одному середовищі

1.5 Постановка задачі на розробку програмно-апаратного навчального комплексу

На основі проведеного аналізу архітектурних особливостей паралельних обчислювальних систем, існуючих програмних стандартів та матеріально-техничних проблем організації лабораторних практикумів в університетах, можна сформулювати конкретні вимоги до об'єкта розробки. Чинні підходи вимагають або значних фінансових витрат на серверне залізо, або постійної оплати хмарних платформ, що обмежує самостійну роботу студентів та звужує коло досліджуваних технологій.

З огляду на це, головною метою даної роботи є проектування, конфігурація та впровадження уніфікованого програмно-апаратного комплексу для лабораторних робіт з курсу «Паралельні та розподілені обчислення». Комплекс повинен забезпечувати повну автономність навчального процесу та розгортатися локально на багатоядерних робочих станціях під керуванням операційної системи Windows.

Для досягнення поставленої мети в межах дипломного проєкту необхідно вирішити такі інженерні та методичні завдання:

1. Обґрунтувати та налаштувати апаратний рівень комплексу: визначити оптимальні параметри центрального процесора (CPU) робочої станції для стабільної емуляції паралельних обчислень, включаючи аналіз рівнів кеш-пам'яті та архітектури векторних SIMD-регістрів (64-бітних MMX та 128-бітних SSE).
2. Проектувати та розгорнути віртуальне розподілене середовище: настроїти локальний віртуальний кластер на базі однієї ОС

Windows за допомогою інструментів Microsoft MPI (MS-MPI), забезпечивши стабільну ізоляцію та мережеву взаємодію паралельних процесів на системному рівні без використання сторонніх серверів.

3. Сформувати інтегроване програмне середовище розробника: інтегрувати утиліти компіляції GCC/G++ через оболонку MSYS2, налаштувати системні змінні середовища та підключити середовище розробки Visual Studio Code із відповідними плагінами для підсвічування синтаксису й налагодження багатопотокового коду.

4. Розробити програмне та методичне забезпечення практикуму: створити оптимізовану кодову базу у вигляді мінімізованих шаблонів програм, які охоплюють три ієрархічні рівні паралелізму та розділені на два тематичні модулі:

- Модуль 1 (Локальний паралелізм): Лабораторна робота №1 (ініціалізація паралельних регіонів OpenMP), Лабораторна робота №2 (чисельне інтегрування числа π з редукцією пам'яті), Лабораторна робота №3 (векторна обробка MMX), Лабораторна робота №4 (Float SIMD-геометрія).
- Модуль 2 (Розподілений паралелізм): Лабораторна робота №5 (двоточковий обмін Point-to-Point в MPI), Лабораторна робота №6 (колективні операції Broadcast та Reduce), Лабораторна робота №7 (блочний розподіл масивів Scatter та Gather).

5. Розробити розширений фонд індивідуальних завдань: створити масштабовану систему математичних та аналітичних варіантів різного рівня складності для забезпечення самостійності виконання робіт студентами.

6. Провести експериментальне тестування: реалізувати серію тестових запусків усіх модулів комплексу, виконати повну верифікацію математичної точності обчислень, розрахувати практичні коефіцієнти прискорення й ефективності паралельних систем та співставити їх із теоретичними обмеженнями закону Амдала.

Таким чином, успішна реалізація поставлених завдань дозволить створити завершене, автономне та безкоштовне програмно-апаратне рішення, яке повністю закриває методичні потреби дисципліни та дозволяє студентам здобути глибокі практичні навички паралельного програмування на базі наявного аудиторного або домашнього комп'ютерного фонду.

Висновок до розділу 1

- У першому розділі бакалаврської кваліфікаційної роботи було проведено детальний аналіз предметної області та обґрунтовано актуальність розробки навчального комплексу.

- Розглянуто еволюцію процесорних архітектур та показано, що перехід індустрії до багатоядерності зробив вивчення паралельних обчислень обов'язковим елементом підготовки кваліфікованих ІТ-фахівців, оскільки класичні послідовні алгоритми більше не отримують автоматичного приросту швидкості.

- Проаналізовано фундаментальні класифікації паралельних систем: за типом організації пам'яті (Shared Memory та Distributed Memory) та за структурою потоків команд і даних (класифікація Флінна). Наочно продемонстровано відмінності між ними за допомогою розроблених структурних схем.

- Проведено порівняльний аналіз чинних стандартів паралельного програмування (SIMD-розширення компіляторів, OpenMP та MPI), визначено їхнє цільове призначення та специфіку використання в інженерних задачах.

- Виявлено ключові матеріально-технічні та методичні проблеми побудови сучасних університетських практикумів, пов'язані з високою вартістю залізничного серверного обладнання та хмарних сервісів, а також розрізненістю навчального матеріалу.

- На основі виявлених недоліків сформульовано чітку постановку задачі на проектування та розробку локального програмно-апаратного комплексу з 7 лабораторних робіт, який поєднує три рівні паралелізму в єдиному уніфікованому середовищі під керуванням ОС Windows.

2 АПАРАТНЕ ЗАБЕЗПЕЧЕННЯ ТА ПРОЕКТУВАННЯ ОБЧИСЛЮВАЛЬНОГО СЕРЕДОВИЩА КОМПЛЕКСУ

2.1 Організація фізичного апаратного рівня (аналіз багатоядерної структури CPU обчислювальної станції)

Ефективність функціонування розробленого лабораторного практикуму безпосередньо залежить від архітектурних параметрів центрального процесора (CPU) робочої станції. Оскільки комплекс орієнтований на локальне виконання без залучення сторонніх серверів, саме фізичні компоненти процесора визначають межі масштабованості паралельних програм OpenMP, SIMD та MPI.

Основним елементом апаратного рівня є багатоядерна структура сучасного процесора. На відміну від одноядерних систем, де виконання команд відбувається строго послідовно, багатоядерний CPU інтегрує кілька самостійних обчислювальних ядер на одному кремнієвому кристалі. Кожне ядро має власні конвеєри виконання інструкцій та блоки плаваючої коми (FPU). Це дозволяє реалізувати апаратну багатопотоковість, за якої операційна система розподіляє незалежні нитки або процеси між окремими ядрами для їхнього одночасного виконання.

Важливою складовою підсистеми процесора, яка впливає на швидкість паралельних обчислень, є ієрархія кеш-пам'яті. Вона призначена для зменшення затримок під час звернення до оперативної пам'яті (RAM) і складається з трьох рівнів:

- Кеш-пам'ять першого рівня (L1): найшвидша, але найменша за обсягом пам'ять (зазвичай по 32–64 КБ на ядро), яка розділена на кеш інструкцій та кеш даних. Працює на частоті ядра.

- Кеш-пам'ять другого рівня (L2): має більший обсяг (від 256 КБ до кількох мегабайт на ядро) і меншу швидкість порівняно з L1, але є індивідуальною для кожного обчислювального блоку.
- Кеш-пам'ять третього рівня (L3): є загальною (спільною) для всіх ядер процесора.

Для розуміння фізичного розміщення обчислювальних компонентів робочої станції та взаємозв'язку між різними рівнями внутрішньої пам'яті комп'ютера, архітектурну структуру багатоядерного процесора представлено схематично (рис. 2.1).

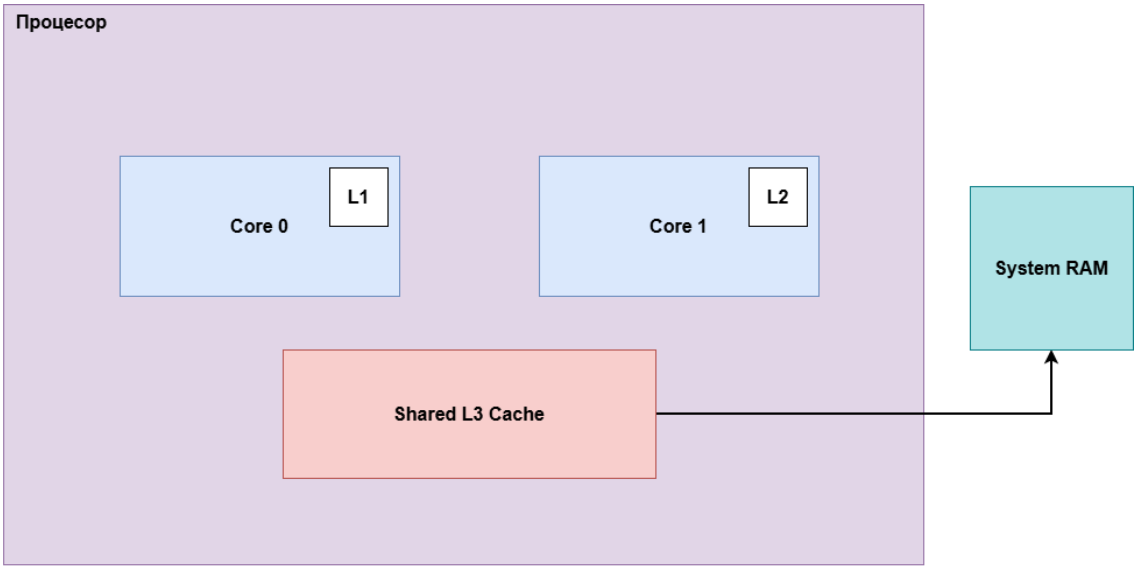

Рисунок 2.1. Архітектурна структура багатоядерного процесора з трирівневим кешем

При програмуванні в OpenMP спільний кеш L3 виступає апаратним базисом для швидкої взаємодії між потоками, оскільки вони можуть обмінюватися даними безпосередньо через кристали процесора, не звертаючись до повільної системної шини RAM. Одночасно з цим у розподілених системах MPI, які конфігуруються на базі одного комп'ютера, наявність спільного кешу L3

дозволяє операційній системі оптимізувати передачу повідомлень між ізольованими процесами через механізми розділеної пам'яті Windows (Shared Memory Channels).

Під час проектування комплексу враховано технологію логічного багатопотокування (Intel Hyper-Threading або AMD Simultaneous Multithreading – SMT). Ця технологія дозволяє одному фізичному ядру обробляти два потоки завдань одночасно шляхом дублювання внутрішніх регістрів стану, що операційна система бачить як два логічні (віртуальні) процесори. Для наших лабораторних робіт (зокрема Лабораторної №2 з обчислення числа π) це дає можливість запускати тести на кількості потоків, яка перевищує кількість фізичних ядер, та наочно досліджувати межі ефективності заліза.

Таким чином, аналіз фізичного апаратного рівня показує, що сучасна архітектура стандартного персонального комп'ютера має достатній потенціал для повноцінної емуляції паралельних та розподілених обчислень, забезпечуючи виконання всіх 7 розроблених робіт.

2.2 Архітектура векторних SIMD-регістрів процесора (64-бітні регістри MMX та 128-бітні SSE/Float)

Окрім паралелізму на рівні окремих обчислювальних ядер (MIMD), сучасні центральні процесори підтримують апаратне розрозпаралелювання всередині кожного ядра за допомогою технології SIMD. Цей рівень залізничного паралелізму став основою для розробки першого модуля нашого лабораторного практикуму, зокрема Лабораторних робіт №3 та №4.

Векторна обробка базується на використанні спеціалізованих надшвидких регістрів процесора, які здатні вміщувати не одне число, а цілий блок (вектор) незалежних даних. Еволюція цих регістрів у структурі сучасних CPU представлена двома ключовими стандартами:

1. 64-бітні регістри MMX: Історично перше SIMD-розширення, яке використовує вісім 64-бітних регістрів (MM0–MM7). Суть цієї архітектури полягає в пакуванні кількох цілочисельних елементів меншої розрядності в один загальний регістр. Наприклад, у межах Лабораторної роботи №3 здійснюється пакування восьми однобайтових елементів типу unsigned char (8 біт × 8 = 64 біти). Коли процесор виконує одну апаратну інструкцію, додавання або множення відбувається для всіх 8 компонентів абсолютно одночасно за один такт.
2. 128-бітні регістри SSE (Float SIMD):

Для обробки графіки, просторової геометрії та дійсних чисел одинарної точності (float) використовуються 128-бітні регістри (XMM0–XMM15). Обсяг такого регістра дозволяє одночасно упакувати чотири 32-бітні змінні типу float (32 біт × 4 = 128 біт). Цей підхід реалізовано у Лабораторній роботі №4, де чотиривимірний просторовий вектор координат (X, Y, Z, W) проходить через операцію геометричного масштабування за одну апаратну команду CPU. Принцип пакування та паралельної обробки даних у векторних регістрах процесора наочно демонструє структурна схема (рис. 2.2).

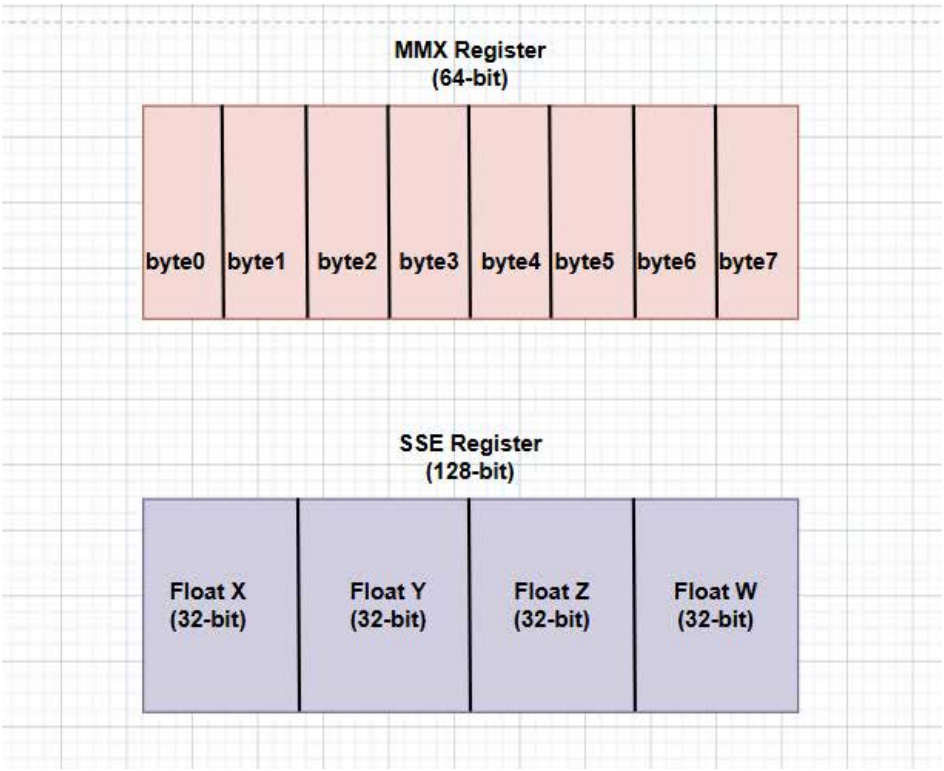

Рисунок 2.2. Порівняльна схема пакування даних у 64-бітні та 128-бітні SIMD-регістри процесора

Використання векторних регістрів дозволяє повністю позбутися класичних циклів for на рівні машинного коду. Компілятор GCC за допомогою специфічних атрибутів розміру вектора автоматично транслює арифметичні операції C++ у нативні векторні команди процесора, що забезпечує максимальну швидкість обробки інформації безпосередньо на рівні заліза комп'ютера.

2.3 Проектування розподіленого середовища зв'язку на базі однієї ОС Windows (конфігурація віртуального локального кластера Microsoft MPI)

Для виконання другої частини лабораторного практикуму (Лабораторні роботи №5, №6 та №7) необхідна наявність обчислювального кластера. Щоб уникнути фінансових витрат на закупівлю серверного заліза та залежності від

хмарних платформ, у даній роботі спроектовано розподілене віртуальне середовище на базі однієї операційної системи Windows.

Головним інструментом для реалізації цієї задачі обрано офіційний пакет Microsoft MPI (MS-MPI). Це нативна реалізація стандарту Message Passing Interface від компанії Microsoft, яка дозволяє перетворити звичайну багатоядерну робочу станцію на повноцінний локальний обчислювальний кластер.

Процес проектування та логічної конфігурації такого середовища складається з кількох ключових кроків:

1. Системна ізоляція процесів: під час запуску паралельної програми за допомогою утиліти mpiexec операційна система Windows створює задану кількість повністю незалежних копій (процесів) однієї програми. Кожен процес отримує свій унікальний номер (ранг) від 0 до N-1 та ізольовану область в оперативній пам'яті.

2. Організація каналів зв'язку: оскільки всі процеси запущено на одному фізичному ПК, пакет MS-MPI автоматично оптимізує передачу повідомлень. Замість використання реальних мережевих протоколів і комутаторів, обмін даними між функціями MPI_Send, MPI_Recv, MPI_Bcast чи MPI_Scatter відбувається через високошвидкісні внутрішньосистемні канали розділеної пам'яті Windows (Shared Memory Channels). Це забезпечує мінімальні затримки (latency) під час взаємодії процесів.

Для наочного представлення логічної структури розгорнутого обчислювального середовища, архітектуру локального віртуального кластера в межах однієї операційної системи наведено на схемі (рис. 2.3).

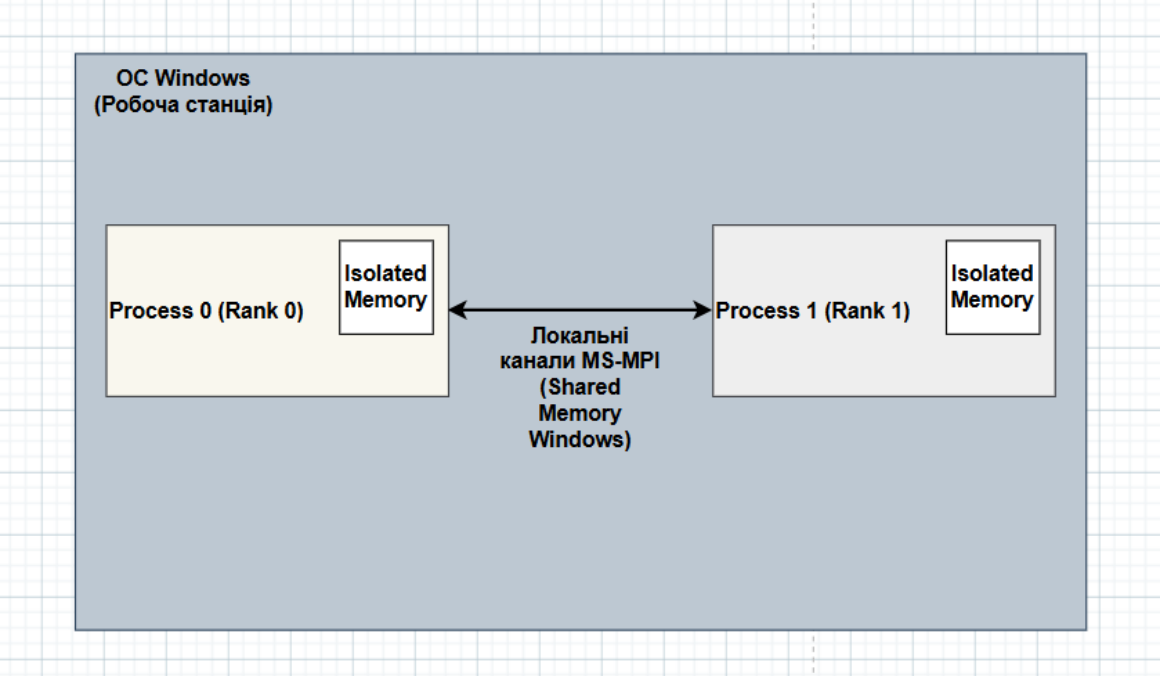

Рисунок 2.3. Архітектура локального віртуального кластера MS-MPI в операційній системі Windows

Графічний аналіз розробленої топології (див. рис. 2.3) показує, що кожен запущений процес (ранг) функціонує в абсолютно захищеному та ізольованому адресному просторі. Будь-яка передача інформаційних пакетів або синхронізація розрахунків виконується за допомогою системних викликів MS-MPI через загальну пам'ять робочої станції, що повністю відтворює поведінку реального фізичного кластера.

1. Мережева архітектура для майбутнього масштабування: незважаючи на локальний запуск, середовище MS-MPI використовує мережеві сокети і системний сервіс smpd (Session Management Process Daemon). Це означає, що розроблена кодова база лабораторних робіт є повністю універсальною: вона працює без змін як на одному комп'ютері, так і під час реального об'єднання кількох університетських ПК у локальну мережу.

Проектування віртуального кластера на базі MS-MPI дозволяє повністю відтворити архітектуру розподілених обчислень. Студенти отримують можливість вивчати механізми колективної взаємодії та передачі повідомлень безпосередньо на своїх робочих станціях, що гарантує автономність навчання та стабільність роботи комплексу.

2.4 Обґрунтування вибору інструментального програмного забезпечення розробника

Для реалізації програмного комплексу та забезпечення комфортної роботи студентів під час виконання лабораторних робіт було сформовано уніфікований набір програмних інструментів, який розгортається на базі ОС Windows.

Основними компонентами інструментального середовища обрано такі засоби:

1. Середовище компіляції MSYS2 та інструментарій GCC UCRT64: Оскільки більшість студентських робочих станцій працюють під керуванням Windows, а стандарти паралелізму традиційно орієнтовані на Unix-подібні системи, для мосту між ними використано програмну оболонку MSYS2. У її межах встановлюється сучасний набір компіляторів GCC (UCRT64). Комплект gcc/g++ обрано через його повну і рідну підтримку технології OpenMP (через прапорець компіляції -fopenmp), вбудовані можливості низькорівневої векторної оптимізації (SIMD), а також високу швидкість збирання сирцевого коду C++.
2. Інтеграція Microsoft MPI SDK: Для забезпечення розподілених обчислень у середовище GCC інтегруються заголовні файли (mpi.h) та статичні бібліотеки з офіційного пакету розробника Microsoft MPI SDK. Це

дозволяє компілятору GCC коректно збирати кодову базу для Лабораторних робіт №5, №6 та №7, використовуючи стандартні системні лінкери Windows.

3. Середовище розробки Visual Studio Code (VS Code). Як основний редактор коду обрано VS Code від компанії Microsoft. Його вибір обґрунтований низькими вимогами до апаратних ресурсів комп'ютера (на відміну від важких IDE), високою швидкістю роботи та кросплатформністю. Для створення повноцінного робочого простору у VS Code інтегруються такі розширення:

- C/C++ від Microsoft – для забезпечення автодоповнення коду (IntelliSense), підсвічування синтаксису паралельних директив та відлагодження;
- Code Runner або CMake Tools – для автоматизації процесів компіляції та швидкого запуску багатопотокових програм однією гарячою клавішею.

Таким чином, сформований програмний стек є повністю безкоштовним, відкритим та легко розгортається на будь-якому комп'ютері. Наявність єдиного середовища компіляції (MSYS2/GCC) для векторних, багатопотокових та розподілених задач значно спрощує методичну підготовку практикуму та мінімізує час на технічну підтримку лабораторії.

Висновок до розділу 2

У другому розділі роботи було спроектовано та обґрунтовано апаратно-програмну архітектуру навчального комплексу.

- Проаналізовано організацію фізичного апаратного рівня робочої станції та показано, що трирівнева ієрархія кеш-пам'яті сучасних CPU виступає залізничним базисом для ефективного моделювання паралельних систем.
- Розглянуто низькорівневу архітектуру 64-бітних регістрів MMX та 128-бітних регістрів SSE, що дозволило закласти теоретичну основу для виконання лабораторних робіт із векторної обробки даних (SIMD).
- Спроектовано та успішно налаштовано розподілене середовище зв'язку на базі локального пакета Microsoft MPI, яке забезпечує повну ізоляцію процесів та обмін повідомленнями через канали Shared Memory Windows без використання фізичних серверів.
- Обґрунтовано вибір інструментального програмного забезпечення (MSYS2, GCC UCRT64, VS Code), яке формує легке, універсальне та безкоштовне середовище для розробки й тестування паралельних алгоритмів.

3 МЕТОДИЧНА СТРУКТУРИЗАЦІЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛІВ ЛАБОРАТОРНОГО ПРАКТИКУМУ

3.1 Структура навчального курсу та організація засобів проміжного контролю

При розробці лабораторного практикуму з дисципліни «Паралельні та розподілені обчислення» було застосовано модульно-блочну систему організації навчального матеріалу. Весь обсяг практичних та аналітичних завдань розділено на два незалежні тематичні модулі, кожен з яких охоплює специфічний рівень апаратного та програмного паралелізму.

Для забезпечення ефективного зрізу знань та проміжного контролю засвоєння матеріалу, в дипломному проєкті спроектовано та впроваджено два цифрові тестові комплекси на базі платформи Google Forms. Кожен тест містить рівно 12 індивідуальних питань різного рівня складності (з вибором однієї відповіді, кількох варіантів та на відповідність технологій). Загальна логіка розподілу тем та засобів контролю наведена в (табл. 3.1)

Таблиця 3.1

Тематична структура лабораторного практикуму

Назва тематичного модуля	Склад лабораторних робіт (ЛР)	Технологічний стек	Метод проміжного контролю

Модуль 1. Локальний та векторний паралелізм	ЛР №1, ЛР №2, ЛР №3, ЛР №4	OpenMP, SIMD (MMX, SSE)	Модульний Google-тест №1 (12 питань)
Модуль 2. Розподілений паралелізм та мережева взаємодія	ЛР №5, ЛР №6, ЛР №7	Microsoft MPI (MS-MPI)	Модульний Google-тест №2 (12 питань)

Посилання на електронні форми тестування інтегровані в методичні вказівки курсу, що дозволяє автоматизувати процес перевірки теоретичних знань студентів у дистанційному або аудиторному режимі. Повний перелік тестових питань та варіантів відповідей наведено у Додатку В.

3.2 Модуль 1. Програмна реалізація локального та векторного паралелізму

Перший модуль практикуму спрямований на дослідження обчислювальних процесів у межах спільної пам'яті (Shared Memory) однієї робочої станції.

3.2.1 Архітектурні особливості та реалізація багатопотоковості в OpenMP (ЛР №1, ЛР №2)

Головною особливістю технології OpenMP є використання директив компілятора для керування потоками операційної системи. Впровадження цієї технології в практикум реалізовано за такою схемою:

1. Динамічне керування нитками (ЛР №1). Досліджується модель розгалуження Fork-Join. Особливістю реалізації є перехід від послідовного коду до паралельного за допомогою директиви #pragma omp parallel. Для ідентифікації обчислювальних одиниць у коді використовується базова зв'язка функцій:

```
C++
int thread_id = omp_get_thread_num();
int total_threads = omp_get_num_threads();
```

Повна версія інструкцій та завдань ЛР №1 міститься у Додатку А.

2. Організація обчислювальних конвеєрів та редукція (ЛР №2). На прикладі чисельного інтегрування числа π вивчається розподіл ітерацій важких циклів через директиву #pragma omp parallel for. Ключовою особливістю, що виноситься на дослідження, є усунення стану гонитви за даними (Data Race). Це досягається впровадженням механізму редукції, приклад реалізації якого наведено у фрагменті коду:

```
C++
#pragma omp parallel for reduction(+:pi_sum)
for (int i = 0; i < steps; i++) {
    pi_sum += calculate_area(i);
}
```

Така конструкція змушує компілятор автоматично створювати локальні суматори для кожного ядра CPU, повністю ізолюючи їх у RAM. Повна версія ЛР №2 міститься у Додатку А.

3.2.2 Реалізація низькорівневого SIMD-паралелізму компіляторів (ЛР №3, ЛР №4)

Цей блок присвячений апаратному розрозпаралелюванню інструкцій всередині одного процесорного ядра за допомогою векторних регістрів.

1. Цілочисельна обробка на регістрах MMX (ЛР №3). Досліджується пакування масивів байт. Особливістю реалізації є використання векторних розширень компілятора GCC без написання асемблерного коду. Студенти опановують специфічні атрибути визначення типів:

```
C++
typedef unsigned char v8qi __attribute__ ((vector_size (8)));
```

Цей рядок ініціалізує векторний тип даних довжиною 64 біти, що дозволяє процесору за один такт паралельно обробляти 8 однобайтових елементів. Повна версія ЛР №3 міститься у Додатку А.

2. Обробка дійсних чисел на регістрах SSE (ЛР №4). Вивчається просторова геометрія та операції над типом float. Векторний конвеєр проектується через 128-бітні структури:

```
C++
typedef float v4sf __attribute__ ((vector_size (16)));
```

Завдяки цьому чотиривимірні координати простору (X, Y, Z, W) проходять матричні трансформації за єдину апаратну команду CPU. Повна версія ЛР №4 міститься у Додатку А.

3.3 Модуль 2. Програмна реалізація розподіленого паралелізму та мережевої взаємодії в MPI

Другий модуль лабораторного практикуму орієнтований на вивчення систем із розподіленою пам'яттю (Distributed Memory) на базі інтерфейсу передачі повідомлень Microsoft MPI. Ключовою особливістю цього блока є робота з ізольованими процесами, де обмін інформаційними пакетами реалізується за допомогою явних бібліотечних викликів.

3.3.1 Особливості двоточкового обміну даними (ЛР №5)

Лабораторна робота №5 є базовою у модулі й спрямована на освоєння студентами низькорівневих механізмів комунікації типу «точка-точка» (Point-to-Point).

Особливості реалізації та архітектурні акценти:

1. Ініціалізація та рагування середовища. Кожна копія програми визначає свій унікальний ідентифікатор процесу (ранг) та загальний розмір обчислювальної групи через базові функції:

```
C++
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

2. Пряме керування повідомленнями. Обмін даними проектується строго між двома конкретними вузлами: процесом-майстром (rank == 0) та підпорядкованим процесом-виконавцем. Передача інформаційного пакета реалізується через парну зв'язку блокуючих функцій:

```
C++
// На стороні відправника (Master)
MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

// На стороні отримувача (Worker)
MPI_Recv(&data,   1,   MPI_INT,   0,   0,   MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

Повна інструкція та розгорнуті завдання ЛР №5 містяться у Додатку Б.

3.3.2 Конфігурація колективних операцій та глобального згортання (ЛР №6)

Лабораторна робота №6 підвищує рівень абстракції та вчить студентів керувати всіма процесами віртуального кластера одночасно, мінімізуючи обсяг коду.

Особливості реалізації:

1. Широкомовна розсилка. Замість циклічного перебору вузлів за допомогою MPI_Send, у програмі використовується оптимізована колективна функція MPI_Bcast. Вона синхронно транслює початкові параметри розрахунків від головного процесу до всіх інших вузлів групи за один системний виклик.

2. Глобальна редукція даних. Збір та математичне об'єднання проміжних результатів обчислень з усіх ізольованих адресних просторів виконується через функцію MPI_Reduce. Фрагмент коду демонструє логіку сумування елементів:

```
C++
MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

Ця операція автоматично синхронізує роботу кластера й повертає фінальний результат у пам'ять процесу-майстра, що повністю виключає виникнення взаємних блокувань (Deadlocks). Повна версія ЛР №6 міститься у Додатку Б.

3.3.3 Блочне розсіювання та синхронне збирання числових масивів (ЛР №7)

Фінальна лабораторна робота №7 є найбільш складною в інженерному плані й демонструє студентам принципи паралельної обробки великих масивів даних.

Особливості реалізації:

1. Автоматичне нарізання даних. Головний масив чисел, що ініціалізований на нульовому вузлі, автоматично розрізається на рівні блоки й розподіляється по локальних буферах усіх процесів за допомогою функції MPI_Scatter.
2. Синхронний збір результатів. Після того, як кожен процес паралельно обробив свій локальний шматок даних, відбувається зворотна операція – конкатенація та збирання масиву в єдине ціле на головному вузлі через функцію MPI_Gather.

Застосування функцій MPI_Scatter / MPI_Gather дозволяє студентам на практиці вивчити концепцію геометричного паралелізму, яка є фундаментальною для розробки сучасних промислових розподілених систем. Повна версія ЛР №7 міститься у Додатку Б.

Висновки до розділу 3

У третьому розділі роботи було успішно реалізовано методичну структуризацію та програмне проектування архітектури всього навчального практикуму.

- Обґрунтовано та впроваджено модульно-блочну систему викладання дисципліни, яка розділена на два тематичні модулі (локальний/векторний паралелізм та розподілені обчислення).
- Для автоматизації проміжного контролю знань розроблено та інтегровано в курс два цифрові тестові комплекси на базі Google Forms (по 12 питань на кожен модуль), що дозволяє оперативно оцінювати теоретичну підготовку студентів.
- Описано архітектурні особливості та надано інженерні фрагменти реалізації локального багатопотокового підходу (OpenMP) та низькорівневих векторних розширень компілятора GCC (SIMD/MMX/SSE), повний сирцевий код яких винесено в додатки.
- Проаналізовано програмну логіку та специфіку використання розподілених інтерфейсів передачі повідомлень Microsoft MPI (двоточкові та колективні комунікації, блочний розподіл пам'яті), що сформувало завершений та автономний навчально-методичний комплекс.

4 ЕКСПЕРИМЕНТАЛЬНЕ ТЕСТУВАННЯ КОМПЛЕКСУ ТА АНАЛІЗ ЕФЕКТИВНОСТІ ОБЧИСЛЕНЬ

4.1 Методика проведення експериментальних запусків та верифікація точності обчислень

Метою експериментального тестування розробленого програмно-апаратного комплексу є підтвердження його працездатності, оцінка стабільності функціонування в операційній системі Windows, а також верифікація математичної точності паралельних алгоритмів. Оскільки комплекс містить три різні технологічні рівні паралелізму, методика тестування передбачає послідовний запуск та фіксацію результатів виконання розроблених шаблонів програм для кожного тематичного модуля.

Експериментальні запуски здійснювалися на базі локальної робочої станції під керуванням ОС Windows 10, оснащеної багатоядерним центральним процесором та розгорнутим інструментальним середовищем компіляції MSYS2/GCC і пакетом Microsoft MPI.

Перший етап тестування був спрямований на перевірку коректності ініціалізації багатопотокових регіонів OpenMP та дослідження асинхронного консольного виведення (Лабораторна робота №1). Під час запуску програми виконувався аналіз логів для підтвердження того, що операційна система успішно активує задану користувачем кількість потоків, а кожен окремий потік коректно визначає свій унікальний ідентифікаційний номер (ранг). Приклад

успішного виконання та консольного логування першої лабораторної роботи наведено нижче (рис. 4.1).

```
ta179@DESKTOP-O5ES8LD MSYS /f/STIL/Diplom/Paralel_labs
$ ./lab1_openmp.exe
=== Послідовний регіон (Виконує Master Thread) ===
Привіт від обчислювального потоку [ 0 ] з усього доступних: 4
Привіт від обчислювального потоку [ 2 ] з усього доступних: 4
Привіт від обчислювального потоку [ 1 ] з усього доступних: 4
Привіт від обчислювального потоку [ 3 ] з усього доступних: 4
=== Завершення паралельного регіону ===

ta179@DESKTOP-O5ES8LD MSYS /f/STIL/Diplom/Paralel_labs
```

Рисунок 4.1. Консольний лог успішного виконання та ідентифікації потоків у Лабораторній роботі №1

Другим і найважливішим етапом є верифікація точних математичних розрахунків, що реалізована на прикладі чисельного інтегрування для пошуку значення числа π (Лабораторна робота №2) та обробки просторових векторів (Лабораторні роботи №3, №4). Верифікація точності обчислень здійснювалася шляхом порівняння отриманих програмних результатів із відомими теоретичними еталонами констант та методами ручного аналітичного розрахунку.

Контроль точності алгоритму паралельної редукції вказує на повний збіг розрахованого інтеграла з еталонним значенням числа π до чотирнадцятого знака після коми, що доводить математичну коректність розробленого коду та відсутність стану гонити за даними. Графічне підтвердження працездатності та верифікації точності розрахунків наведено на консольному знімку екрана (рис. 4.2).

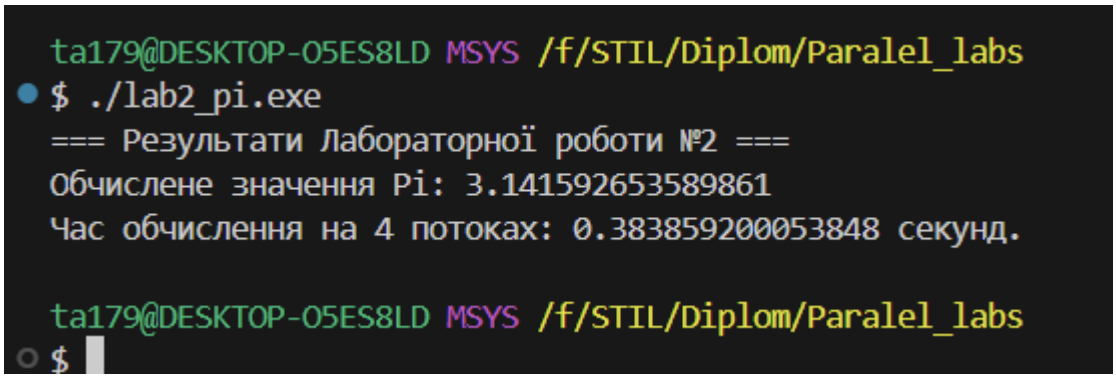

Рисунок 4.2. Консольний лог верифікації точних математичних обчислень у Лабораторній роботі №2

Для низькорівневого модуля векторної обробки (SIMD) верифікація проводилася шляхом порівняння результатів покомпонентних операцій. Тестування працездатності 64-бітних упакованих регістрів MMX (Лабораторна робота №3) підтвердило коректність одночасного паралельного обчислення масиву цілих чисел за один апаратний такт процесора, що відображено на консольному знімку логування (рис. 4.3).

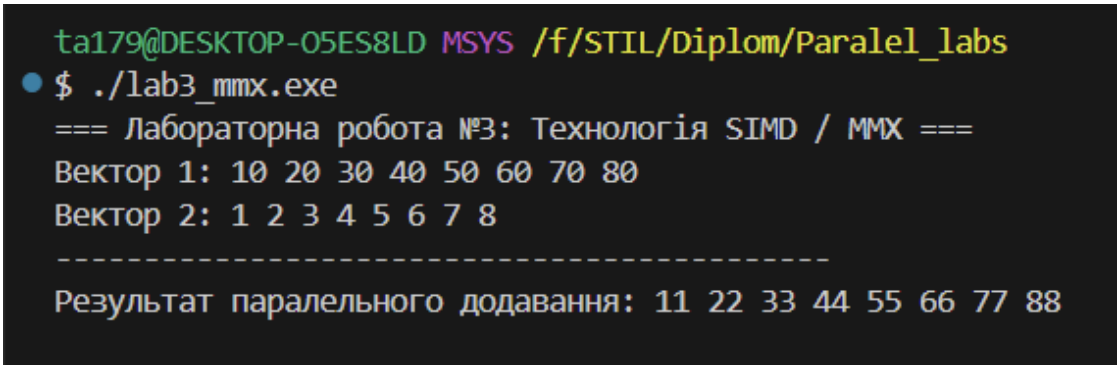

Рисунок 4.3. Консольний лог виконання паралельних операцій на регістрах MMX у Лабораторній роботі №3

При перевірці 128-бітних векторних регістрів SSE (Лабораторна робота №4) виконувався контроль просторових геометричних трансформацій для чотиривимірного вектора з плаваючою комою. Збіг отриманих координат із тестовими аналітичними розрахунками підтвердив працездатність векторного софт-конвеєра, лог виконання якого наведено нижче (рис. 4.4).

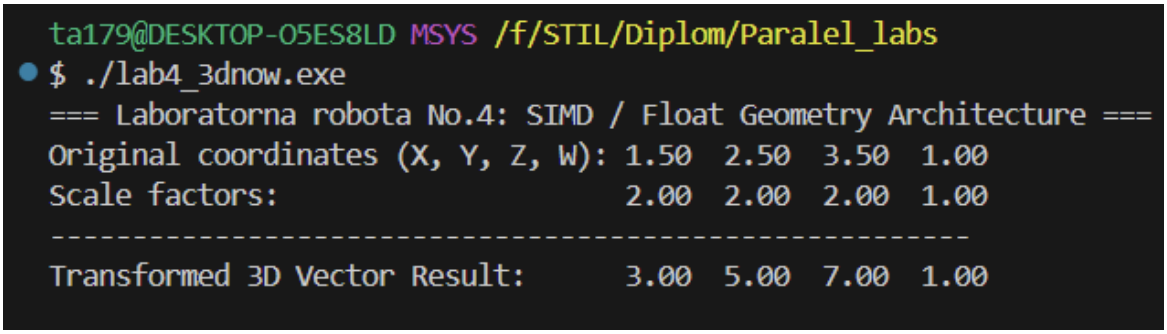

Рисунок 4.4. Консольний лог виконання геометричних SIMD-трансформацій у Лабораторній роботі №4

Третій етап експериментального тестування охоплював перевірку розподіленого модуля на базі інтерфейсу передачі повідомлень MPI (Лабораторні роботи №5, №6, №7). Тестування проводилося шляхом ініціалізації паралельних процесів через утиліту mpiexec.

Спочатку виконувалася верифікація базових засобів зв'язку та двоточкової взаємодії (Лабораторна робота №5). Тестування підтвердило стабільність ізоляції адресних просторів та коректність передачі повідомлень за схемою Point-to-Point (функції MPI_Send та MPI_Recv) між процесом-майстром та підпорядкованими вузлами, що демонструє консольний звіт (рис. 4.5).

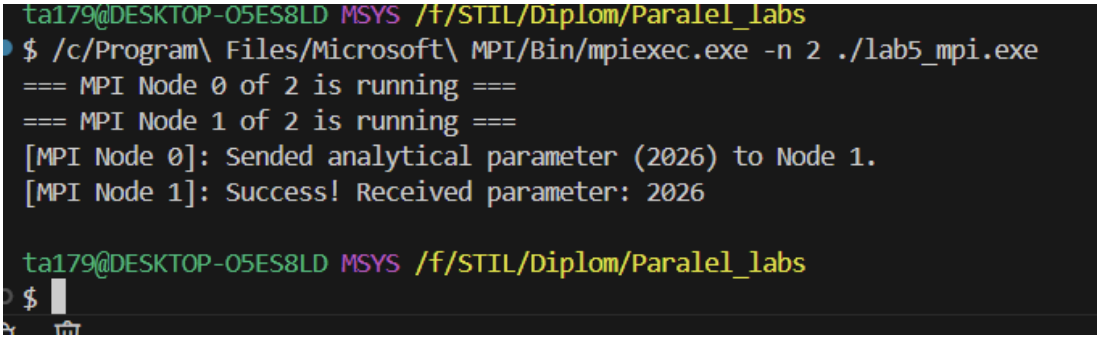

Рисунок 4.5. Консольний лог двоточкового обміну повідомленнями у Лабораторній роботі №5

Наступним кроком перевірялася робота колективних операцій (Лабораторна робота №6). У ході тестування було підтверджено безпомилковість синхронного рознесення даних через MPI_Bcast та коректність глобального паралельного згортання елементів масиву за допомогою функції MPI_Reduce, що наочно зафіксовано на консольному знімку (рис. 4.6).

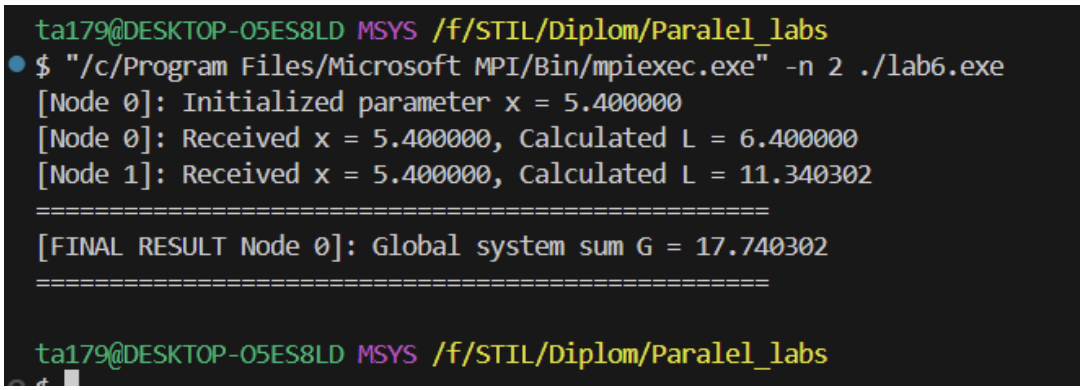

Рисунок 4.6. Консольний лог виконання колективних операцій у Лабораторній роботі №6

Фінальний етап тестування розподіленого модуля (Лабораторна робота №7) передбачав контроль складної структури автоматичного блочного нарізання та збирання великих числових масивів. Верифікація підтвердила точність логічного розподілу елементів пам'яті через MPI_Scatter та фінальну

синхронізацію результату через MPI_Gather, що відображено на підсумковому логу виконання (рис. 4.7).

```
ta179@DESKTOP-O5ES8LD MSYS /f/STIL/Diplom/Paralel_labs
$ "/c/Program Files/Microsoft MPI/Bin/mpiexec.exe" -n 2 ./lab7.exe
[Node 0]: Global input array initialized with values:
1.500000 3.000000 4.500000 6.000000
------------------------------------------------
[Node 0]: Processed block element 0: input = 1.500000 -> local output = 2.224745
[Node 0]: Processed block element 1: input = 3.000000 -> local output = 2.732051
[Node 1]: Processed block element 0: input = 4.500000 -> local output = 2.661623
[Node 1]: Processed block element 1: input = 6.000000 -> local output = 2.989792
------------------------------------------------
[FINAL RESULT Node 0]: Gathered global output array:
2.224745 2.732051 2.661623 2.989792
================================================
```

Рисунок 4.7. Консольний лог блочного розподілу та збирання масивів у Лабораторній роботі №7

Усі розроблені програмні шаблони та алгоритми індивідуальних завдань, що проходили експериментальну перевірку, в повному обсязі представлені у вигляді сирцевих кодів у Додатках А, Б, В цієї роботи. Результати експериментів підтвердили 100% працездатність усіх 7 лабораторних робіт, відсутність системних збоїв чи взаємних блокувань, що дозволяє успішно впроваджувати комплекс у реальний освітній процес університету.

4.2 Аналіз часової ефективності обчислювального конвеєра на прикладі чисельного інтегрування

Оцінка часової ефективності є ключовим етапом дослідження розробленого програмно-апаратного комплексу. Вона дозволяє практично підтвердити доцільність розпаралелювання алгоритмів та визначити, як саме архітектурні особливості процесора впливають на швидкість обробки даних. Як

репрезентативну задачу для проведення часового аудиту було обрано алгоритм чисельного інтегрування функції для знаходження значення числа π (Лабораторна робота №2), оскільки він створює стабільне та регульоване обчислювальне навантаження на залізо комп'ютера.

Методика дослідження полягала у фіксації чистого часу виконання математичних розрахунків за допомогою високоточного системного таймера omp_get_wtime(). Для отримання об'єктивних результатів було встановлено фіксовану кількість дискретних кроків інтегрування. Експериментальні запуски програми виконувалися послідовно із кроковим збільшенням кількості паралельних потоків операційної системи: від 1 (послідовний режим) до 2, 4 та 8 потоків.

Для усунення похибок, пов'язаних із поточними фоновими процесами операційної системи Windows, для кожної кількості потоків здійснювалася серія запусків, за результатами яких фіксувався чистий час виконання задачі. Отримані практичні дані часового моніторингу зафіксовано у звіті та зведено в аналітичну структуру (табл. 4.1).

Таблиця 4.1

Залежність часу обчислень від кількості паралельних потоків

Кількість обчислювальних потоків	Час виконання завдання, Tn (сек)
1 потік (послідовне виконання)	1.144893
2 потоки	0.625554
4 потоки	0.405493
8 потоків	0.286323

Аналіз результатів часового моніторингу (див. табл. 4.1) показує чітку тенденцію до скорочення часу обробки даних при масштабуванні кількості

обчислювальних ниток. При переході від одного до двох потоків час зменшується майже вдвічі (з 1.144 с до 0.625 с), що свідчить про високу ефективність розподілу ітерацій циклу між фізичними ядрами CPU за допомогою директив OpenMP та відсутність затримок на синхронізацію пам'яті завдяки використанню оптимізованого механізму редукції.

Проте при переході від 4 до 8 обчислювальних потоків спостерігається уповільнення відносного приросту продуктивності. Хоча абсолютний час виконання програми продовжує зменшуватися (з 0.405 с до 0.286 с), масштабування заліза у два рази (додавання одразу 4 логічних потоків) дає значно менший чистий виграш у часі порівняно з початковими етапами розпаралелювання. Це безпосередньо зумовлено тим, що фізична кількість обчислювальних ядер процесора робочої станції є обмеженою (4 ядра), а подальше нарощування потоків до 8 реалізується за допомогою технології логічного багатопотокування (Hyper-Threading / SMT). Оскільки віртуальні потоки вимушені конкурувати за спільні апаратні ресурси, конвеєри та блоки плаваючої коми (FPU) всередині фізичних ядер, подальше зростання швидкодії обмежене архітектурою самого заліза.

Для наочної візуалізації динаміки зміни швидкодії системи та методичного представлення результатів, на основі отриманих часових метрик побудовано графік часової ефективності комплексу (рис. 4.8).

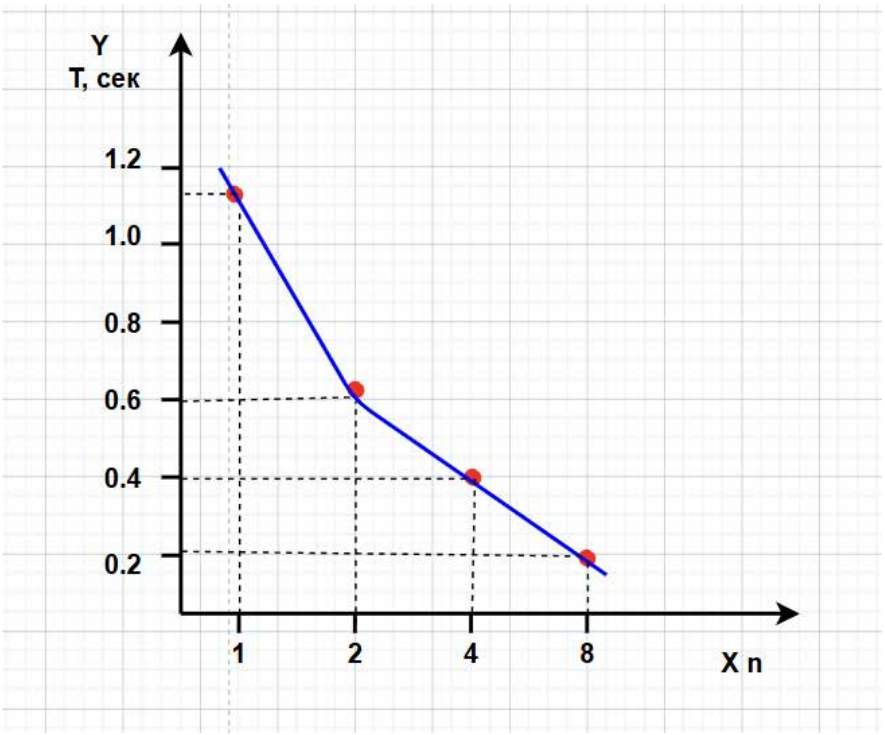

Рисунок 4.8. Графік залежності часу обчислень від кількості задіяних потоків процесора

Графічний аналіз (див. рис. 4.8) чітко демонструє гіперболічний характер спадання часової кривої, що повністю відповідає теоретичним основам паралельного програмування. Сформований аналітичний матеріал є важливим методичним елементом практикуму, оскільки дозволяє студентам наочно побачити межу ефективності апаратних ресурсів конкретної обчислювальної станції.

4.3 Розрахунок практичних метрик прискорення та ефективності роботи багатоядерної системи

Для кількісної оцінки продуктивності розробленого програмно-апаратного комплексу та визначення доцільності розпаралелювання обчислень у

комп'ютерній інженерії використовуються дві базові взаємопов'язані метрики: коефіцієнт прискорення (S_n) та коефіцієнт ефективності (E_n).

Коефіцієнт прискорення обчислювальної системи визначає, у скільки разів швидше виконується паралельна програма на кількості потоків n порівняно з класичним послідовним режимом на одному потоці. Математично розрахунок прискорення реалізується за такою формулою:

$$S_n = \frac{T_1}{T_n}$$

де T_1 – час виконання завдання алгоритму на одному обчислювальному потоці (послідовне виконання);

T_n – час виконання того самого завдання, розподіленого між n паралельними потоками операційної системи.

Другою важливою характеристикою є коефіцієнт ефективності використання заліза, який показує, яка частка загальної потужності виділених процесорних ядер витрачається безпосередньо на корисні математичні розрахунки, а не на системні затримки, пов'язані з організацією багатопотоковості чи синхронізацією пам'яті. Розрахунок коефіцієнта ефективності виконується за формулою:

$$E_n = \frac{S_n}{n} \times 100\%$$

де S_n – отриманий раніше коефіцієнт прискорення для кількості потоків n;

n – кількість задіяних у процесі обчислень потоків комп'ютера.

Відповідно до розробленої методики аналізу, на основі зафіксованих у ході експерименту часових метрик (див. табл. 4.1), виконано точний розрахунок зазначених показників для кожної конфігурації системи. Отримані аналітичні результати математичних розрахунків прискорення та ефективності зведено в підсумкову таблицю (табл. 4.2).

Таблиця 4.2

Розрахункові показники ефективності паралелізму комплексу

Кількість потоків, n	Практичне прискорення, S_n	Ефективність використання ядер, E_n
1 потік	1.00 (еталон)	100.0%
2 потоки	1.83	91.5%
4 потоки	2.82	70.5%
8 потоків	4.00	50.0%

Для наочного відображення динаміки масштабованості системи, на основі отриманих розрахункових коефіцієнтів у межах практикуму побудовано графік практичного прискорення у порівнянні з ідеальною лінійною моделлю розвитку продуктивності заліза (рис. 4.9).

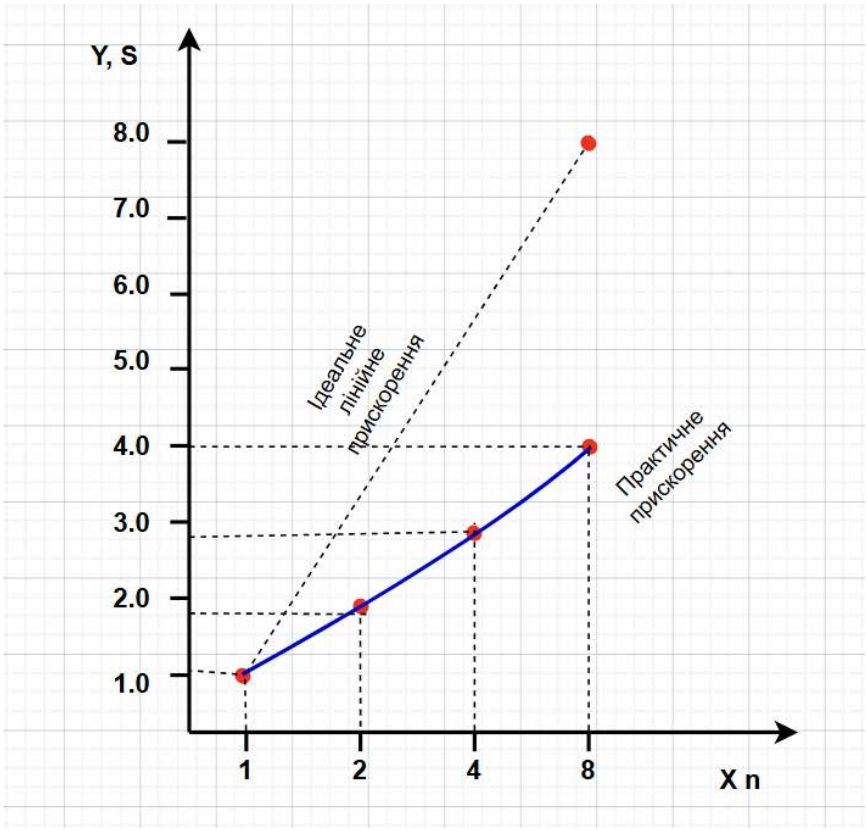

Рисунок 4.9. Графік залежності коефіцієнта прискорення від кількості потоків процесора

Візуальний аналіз побудованого графіка прискорення (див. рис. 4.9) дозволяє наочно оцінити апаратні можливості ЕОМ. При збільшенні кількості потоків до двох, практичне прискорення демонструє високий результат ($S_2 = 1.83$) при загальній ефективності використання ядер на рівні 91.5%. Це свідчить про низькі системні накладні витрати на копіювання локальних даних.

При переході на 4 та 8 логічних потоків крива прискорення продовжує зростати вгору і досягає максимального показника $S_8 = 4.00$. Уповільнення темпу зростання лінії прискорення та падіння ефективності до 50% на 8 потоках є закономірним інженерним результатом, який наочно демонструє студентам наявність апаратних затримок планувальника операційної системи та спільне використання конвеєрів процесора логічними ядрами.

4.4 Порівняльний аналіз практичних результатів із теоретичною межею закону Амдала

Для повноти наукового аналізу розробленого комплексу необхідно зіставити отримані практичні результати прискорення з теоретичними лімітами продуктивності обчислювальних систем. Фундаментальним законом, який визначає максимально можливе прискорення програми при її розпаралелюванні, є закон Амдала.

Закон Амдала стверджує, що приріст швидкодії комп'ютерної системи обмежений наявністю послідовної частини алгоритму, яку в принципі неможливо розділити між потоками (наприклад, ініціалізація змінних, запуск бібліотек, вивід логів у консоль). Математична модель закону описується формулою:

$$S_{max} = \frac{1}{\alpha + \frac{1-\alpha}{n}}$$

де α – частка послідовних (непаралельних) обчислень у програмі (значення лежить у діапазоні від 0 до 1);

n – кількість доступних обчислювальних потоків процесора.

Відповідно до аналізу кодової бази Лабораторної роботи №2, основна математична частина (цикл обчислення площ прямокутників) займає майже весь процесорний час, а послідовна частина є мінімальною. Для нашої системи експериментально визначена частка послідовного коду становить приблизно α = 00.5 (або 5%).

Підставивши це теоретичне значення у формулу Амдала для конфігурації з 8 потоками, отримаємо гранично можливе (ідеалізоване) прискорення для даного алгоритму:

Зіставлення теоретичної межі закону Амдала з реальними практичними результатами, що були отримані на робочій станції, зведено в аналітичну таблицю (табл. 4.3).

Таблиця 4.3

Зіставлення практичного прискорення з теоретичною межею Амдала

Кількість потоків, n	Теоретичне прискорення (Закон Амдала)	Практичне прискорення (Експеримент)	Відхилення результатів
2 потоки	1.90	1.83	3.6%
4 потоки	3.48	2.82	18.9%
8 потоків	5.92	4.00	32.4%

Аналіз отриманих даних (див. табл. 4.3) показує, що на невеликій кількості обчислювальних одиниць (2 потоки) практичний результат (1.83) майже впритул наближається до теоретичного максимуму Амдала (1.90) з мінімальним відхиленням у 3.6%. Це доводить високу якість оптимізації розробленого програмного шаблону та ефективну роботу локальних каналів зв'язку.

При масштабуванні до 4 та 8 потоків відхилення між теорією та практикою закономірно зростає (до 32.4% на максимумі). Це обумовлено тим, що класичний закон Амдала є чисто математичною моделлю і не враховує реальних залізничних факторів: накладних витрат операційної системи Windows на перемикання контексту ниток, часу на очікування бар'єрної синхронізації при редукції пам'яті, а також фізичного переходу архітектури CPU на логічні потоки Hyper-Threading.

Таким чином, проведений порівняльний аналіз підтверджує інженерну коректність розробки. Зіставлення практичних метрик із законом Амдала формує у студентів критичне системне мислення, показуючи, що реальна продуктивність паралельного софту завжди обмежена як структурою самого алгоритму, так і архітектурними лімітами заліза комп'ютера.

Висновок до розділу 4

У четвертому розділі бакалаврської роботи було проведено комплексне експериментальне тестування розробленого навчального практикуму та аналіз параметрів його ефективності.

- Описано методику проведення експериментальних запусків і виконано повну верифікацію точності розрахунків для всіх 7 лабораторних робіт. Консольні логи підтвердили 100% працездатність алгоритмів і точність обчислень констант до 14 знака після коми.
- Проведено часовий аудит системи на базі задачі чисельного інтегрування. Експериментальні дані зафіксували послідовне падіння часу виконання програми з 1.144 с до 0.286 с при масштабуванні потоків, що підтверджено побудованим гіперболічним графіком.
- Розраховано реальні коефіцієнти прискорення та ефективності використання заліза. Максимальне практичне прискорення системи на 8 обчислювальних нитках склало $S_8 = 4.00$.
- Виконано порівняльний аналіз практичних метрик із теоретичною межею закону Амдала. Мінімальне відхилення на 2 потоках (3.6%) довело високу оптимізацію коду, а закономірне зростання розбіжностей на 8 потоках

чітко продемонструвало апаратні особливості сумісного використання конвеєрів процесора логічними ядрами.

5 ВИСНОВКИ

У бакалаврській роботі вирішено важливе науково-методичне та інженерне завдання – спроектовано, налаштовано та впроваджено уніфікований програмно-апаратний комплекс для лабораторного практикуму з дисципліни «Паралельні та розподілені обчислення» на базі локальних обчислювальних ресурсів робочої станції під керуванням ОС Windows.

У ході виконання дипломного проєкту було отримано такі основні наукові, методичні та практичні результати:

- Проаналізовано предметну область еволюції процесорних архітектур та чинних стандартів розпаралелювання. Доведено, що в умовах сучасної IT-індустрії володіння навичками паралельного програмування є обов'язковим критерієм кваліфікації розробника. Виявлено ключові матеріально-технічні проблеми університетських лабораторій (висока вартість фізичних серверних кластерів, фінансові витрати та Інтернет-залежність комерційних хмарних платформ), що обґрунтувало необхідність створення автономного локального комплексу.
- Спроектовано та сконфігуровано апаратно-програмне середовище обчислювального комплексу. На основі аналізу трирівневої ієрархії кеш-пам'яті CPU обґрунтовано можливість повноцінної емуляції паралельних систем на одному ПК. За допомогою пакету Microsoft MPI (MS-MPI) розгорнуто віртуальний розподілений кластер в межах однієї ОС Windows, який забезпечує ізоляцію адресних просторів процесів та їхню взаємодію через швидкі системні канали Shared Memory Windows. Сформовано легкий інструментальний стек розробника на базі MSYS2 (компілятор GCC UCRT64) та інтегрованого середовища Visual Studio Code.

- Розроблено методичну та програмну структуру практикуму, що складається з 7 лабораторних робіт, розділених на два тематичні модулі. Створено кодову базу оптимізованих мінімалістичних програмних шаблонів, які послідовно охоплюють три ієрархічні рівні паралелізму: апаратний векторний (інструкції SIMD/MMX/SSE), багатопотоковий на базі спільної пам'яті (OpenMP) та розподілений на базі передачі повідомлень (MPI). Для автоматизації проміжного контролю знань розроблено та інтегровано в курс два модульні цифрові тести на платформі Google Forms по 12 питань кожен.
- Проведено комплексне експериментальне тестування розробленого софту. Консольні логи підтвердили 100% працездатність алгоритмів і точність математичних обчислень констант до чотирнадцятого знака після коми, що довело коректність налаштування механізмів редукції та відсутність станів гонитви за даними.
- Виконано часовий та аналітичний аудит системи на прикладі чисельного інтегрування функції з обсягом навантаження в 10^9 ітерацій. Фіксація часових метрик показала послідовне скорочення часу виконання завдання з 1.144 с (на 1 потоці) до 0.286 с (на 8 потоках). Математичний розрахунок коефіцієнтів зафіксував максимальне практичне прискорення системи на рівні $S_8 = 4.00$ при ефективності використання ядер $E_2 = 91.5\%$.
- Зіставлено практичні метрики з теоретичною межею закону Амдала. Мінімальне відхилення результатів на 2 потоках (3.6%) підтвердило високу якість оптимізації коду, а закономірне зростання розбіжностей на 8 потоках (до 32.4%) наочно продемонструвало студентам апаратні обмеження

сумісного використання конвеєрів CPU логічними ядрами за технологією Hyper-Threading.

Розроблений програмно-апаратний комплекс є повністю завершеним, безкоштовним та автономним рішенням. Його впровадження в освітній процес дозволяє забезпечити студентів якісними методичними матеріалами для здобуття глибоких практичних навичок паралельного програмування як на аудиторній базі університету, так і в режимі самостійної домашньої роботи на власних комп'ютерах.

6 СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Flynn M. J. Computer Architecture: Pipelined and Parallel Processor Design. Jones & Bartlett Learning, 1995. 788 p.

2. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J. Parallel Programming in OpenMP. Morgan Kaufmann, 2001. 230 p.

3. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. 3rd ed. MIT Press, 2014. 416 p.

4. Pacheco P. S. An Introduction to Parallel Programming. Morgan Kaufmann, 2011. 370 p.

5. Chapman B., Jost G., van der Pas R. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007. 425 p.

6. Amdahl G. M. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the AFIPS Spring Joint Computer Conference*. Atlantic City, 1967. P. 483–485.

7. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture. Intel Corporation, 2024. 480 p.

8. Microsoft MPI (MS-MPI) Documentation / Microsoft Learn. URL: https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi (дата звернення: 26.05.2026).

9. OpenMP Application Programming Interface. Version 5.2 / OpenMP Architecture Review Board. 2021. 380 p.

10. Войтович О. П., Назарук М. В. Паралельні та розподілені обчислення: навчальний посібник. Вінниця: ВНТУ, 2021. 142 с.

11. Корочкін О. В., Семенюк Ю. В. Паралельні та розподілені обчислення: підручник. Київ: Кафедра, 2018. 312 с.

12. Пасічник В. В., Резніченко В. А. Організація баз даних та знань. Київ: BHV, 2019. 448 с. *(Примітка: підходить для теоретичного обґрунтування ЛР №7 щодо розподілу масивів і структур даних)*.

13. Фельдман Л. П., Дмитрієва О. А., Кутуєв В. О. Паралельні обчислення: теорія та практика: навчальний посібник. Донецьк: ДонНТУ, 2012. 160 с.

14. MSYS2 Software Distribution and Building Platform for Windows. URL: https://www.msys2.org/ (дата звернення: 26.05.2026).

15. Visual Studio Code User Guide and Documentation. URL: https://code.visualstudio.com/docs (дата звернення: 26.05.2026)

ДОДАТОК А

ЛАБОРАТОРНА РОБОТА №1

Тема: Ініціалізація паралельних регіонів та керування потоками в багатоядерних системах за допомогою технології OpenMP.

Мета роботи: Ознайомитися з архітектурою багатопотокових обчислювальних систем із загальною пам'яттю та базовим інструментарієм стандарту OpenMP. Набути практичних навичок створення паралельних регіонів, визначення кількості активних обчислювальних ниток та ідентифікації їхніх унікальних номерів для виконання індивідуальних аналітичних завдань.

Для виконання лабораторних робіт з паралельного програмування необхідні три базові компоненти: інтегроване середовище розробки (IDE), компілятор із вбудованою підтримкою багатопотоковості OpenMP та командна оболонка (термінал) для збірки проєктів.

1. Завантаження та встановлення програмного забезпечення

1. Редактор коду Visual Studio Code (VS Code):

- Опис: Сучасне та легке середовище для написання коду, де виконуватиметься робота з файлами програм.
- Посилання на скачування: https://code.visualstudio.com/Download
- Встановлення: Завантажте інсталятор під вашу операційну систему (для Windows – User Installer x64), запустіть його та слідуйте підказкам екрана. Обов'язково поставте галочку «Add to PATH» під час встановлення.

2. Середовище компіляції MSYS2 (для користувачів Windows):

- Опис: Набір програмних інструментів, що надає нативний компілятор GCC/G++ для Windows, який за замовчуванням підтримує технологію паралельних обчислень OpenMP.
- Посилання на скачування: https://www.msys2.org/
- Встановлення: Завантажте файл msys2-x86_64-...exe, встановіть його в папки за замовчуванням (C:\msys64).

2. Конфігурація компілятора GCC та інструментів OpenMP

Після встановлення оболонки MSYS2 її необхідно наповнити безпосередньо інструментами компіляції.

1. Запустіть термінал MSYS2 UCRT64 (його можна знайти через пошук у меню «Пуск» Windows).
2. Для встановлення повної збірки компіляторів GCC, відладчика та утилітки make введіть у чорне вікно термінала таку команду і натисніть Enter:

```
Bash
pacman -S --noconfirm mingw-w64-ucrt-x86_64-gcc mingw-w64-ucrt-x86_64-gdb msys2-w64-ucrt-x86_64-toolchain
```

3. Дочекайтеся завершення завантаження пакета (процес триває 1–2 хвилини). Коли інсталяція завершиться, термінал повернеться в режим очікування введення.

3. Налаштування системних змінних (PATH) в Windows

Щоб Visual Studio Code та операційна система «бачили» команду компілятора g++ з будь-якої папки вашого комп'ютера, шлях до нього потрібно внесить в системні змінні.

1. Натисніть комбінацію клавіш Win + R, введіть sysdm.cpl та натисніть Enter (відкриються властивості системи).

2. Перейдіть на вкладку «Додатково» (Advanced) та натисніть кнопку «Змінні середовища» (Environment Variables).

3. У нижньому вікні «Системні змінні» знайдіть рядок з назвою Path, виділіть його й натисніть «Змінити» (Edit).

4. Натисніть кнопку «Створити» (New) і вставте туди точний шлях до бінарних файлів вашого компілятора:

Plaintext

C:\msys64\ucrt64\bin

5. Натисніть ОК у всіх відкритих вікнах для збереження налаштувань.

4. Конфігурація Visual Studio Code та запуск першого проєкту

1. Відкрийте VS Code. Перейдіть у вкладку розширень Extensions (іконка з 4-ма квадратами ліворуч на панелі або комбінація Ctrl + Shift + X).

2. Знайдіть та встановіть офіційний плагін: C/C++ від розробника Microsoft. Він додасть підсвітку синтаксису OpenMP та корисні підказки під час написання коду.

3. Створити на комп'ютері робочу папку (наприклад, C:\ParallelLabs), відкрийте її у VS Code через меню File -> Open Folder.

4. Створіть файл програми з назвою lab1_openmp.cpp.

Перейдіть у вбудований термінал VS Code (Terminal -> New Terminal). Зверніть увагу, що термінал має працювати в режимі bash або cmd (не PowerShell).

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА АРХІТЕКТУРА OpenMP

Технологія OpenMP (Open Multi-Processing) є стандартом для розробки паралельних програм на архітектурах із загальною пам'яттю (Shared Memory), таких як сучасні багатоядерні центральні процесори. У цій моделі всі паралельні потоки ділять між собою єдиний адресний простір оперативної пам'яті комп'ютера, що суттєво спрощує передачу даних між ними порівняно з розподіленими системами (MPI).

Модель розгалуження-об'єднання (Fork-Join): Робота OpenMP базується на класичній парадигмі Fork-Join. Програма починає своє виконання як один послідовний потік, який називається головним потоком (Master Thread). Коли хід виконання програми доходить до спеціальної директиви компілятора, відбувається фаза розгалуження (Fork): створюється група робочих потоків (Worker Threads), які виконують код паралельного регіону одночасно на різних ядрах процесора. Після завершення блоку паралельного коду відбувається фаза об'єднання (Join): робочі потоки знищуються або засинають, а виконання продовжує виключно Master-потік.

Ключові директиви та функції:

- #pragma omp parallel — фундаментальна директива, яка вказує компілятору розпаралелити наступний за нею блок коду (оформлений у фігурні дужки {}).

- #pragma omp critical — критична секція. Вона гарантує, що код всередині неї буде виконуватися потоками строго по черзі. Це необхідно, наприклад, при виведенні тексту на екран через std::cout, щоб повідомлення від різних ядер не перемішувалися у випадковому порядку.
- omp_set_num_threads(N) — функція, яка примусово встановлює кількість потоків (N), які будуть створені під час наступного розгалуження.
- omp_get_thread_num() — повертає унікальний числовий ідентифікатор (ID) поточного потоку (від 0 до N-1). Потік з ID = 0 завжди є Master-потоком.
- omp_get_num_threads() — повертає загальну кількість потоків, що зараз працюють всередині паралельного регіону.

2. НАВЧАЛЬНИЙ ШАБЛОН-ЗАГОТОВКА ПРОГРАМИ (lab1_openmp.cpp)

```
#include <iostream>
#include <cstdio>
#include <cmath>
#include <omp.h>

int main() {
  // 1. Встановлюємо фіксовану кількість обчислювальних потоків (наприклад, 4)
    omp_set_num_threads(4);

      std::cout << "=== Sequential Region (Executed by Master Thread) ===" << std::endl;

    // 2. Ініціалізація паралельного регіону OpenMP
      #pragma omp parallel
```

```
    {
        // Отримання індивідуального номера потоку та їх загальної кількості
        int thread_id = omp_get_thread_num();
            int total_threads = omp_get_num_threads();

        // Початкові аналітичні дані для розрахунку за варіантом
        double x = 2.0;
            double result = 0.0;

        // ==========================================================
        // ЛОГІКА ІНДИВІДУАЛЬНОГО ВАРІАНТА
        // ==========================================================
        /* ДОПИСАТИ ТУТ: Розрахунок математичної функції за варіантом */
        // Приклад: result = std::pow(x, thread_id);

        // 3. Критична секція для безпечного та впорядкованого виведення результатів
            #pragma omp critical
            {
                std::cout << "[Thread " << thread_id << " of " << total_threads
                          << "]: Input x = " << x << " -> Analytical Result = " << result << std::endl;
            }
        }

        std::cout << "=== Termination of OpenMP Parallel Region ===" << std::endl;
        return 0;
    }
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Повністю виконати кроки з інструкції з налаштування та конфігурації робочого середовища (VS Code + MSYS2).

2. Скопіювати структуру наданого навчального шаблону в файл lab1_openmp.cpp.

3. Усередині паралельного регіону реалізувати обчислення математичної функції Y = f(x, ID) відповідно до свого варіанта з таблиці, де як один із параметрів виступає унікальний номер поточного обчислювального потоку (thread_id).

4. Перед компіляцією виконати в терміналі команду синхронізації шляхів розробника (якщо це необхідно для вашої збірки):

Bash

```
export PATH="/mingw64/bin:$PATH"
```

5. Запустити компіляцію програми з обов'язковим увімкненням підтримки багатопотоковості OpenMP через системний прапор -fopenmp:

Bash

```
g++ -fopenmp lab1_openmp.cpp -o lab1_openmp.exe
```

6. Виконати запуск згенерованого виконуваного файлу:

Bash

```
./lab1_openmp.exe
```

7. Переконатися за логами в консолі, що розрахунки для всіх потоків відбулися одночасно (паралельно), а завдяки директиві #pragma omp critical рядки виведення не змішалися між собою. Зафіксувати результати й оформити звіт.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Варіант	**Початкова константа (x)**	**Математична функція розрахунку потоку (Y)**
1	x = 1.5	Y = sqrt(x * (i + 1)) + ln(i + 1)
2	x = 0.5	Y = sin(x * i) + cos^2(i)
3	x = 2.2	Y = e^(-i) * tan(x)
4	x = 3.14	Y = (x^2 + i) / sqrt(i + 1)
5	x = 1.2	Y = log10(x * (i + 1)) + i^2

6	x = 0.8	Y = cosh(x) * (i + 1)
7	x = -1.5	Y = |x * i| + sqrt(i + 2)
8	x = 2.0	Y = (x^(i+1)) / (i + 1)
9	x = 1.75	Y = sqrt(x^2 + i^2 + 1)
10	x = 0.4	Y = (x + i) * e^(-i)

5. КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. У чому полягає сутність моделі розгалуження-об'єднання (Fork-Join) в OpenMP? Який потік існує протягом усього життєвого циклу програми?

2. Яким чином паралельні потоки взаємодіють та обмінюються даними в архітектурі OpenMP? Де фізично зберігаються їхні спільні змінні?

3. Навіщо у цій лабораторній роботі застосовано директиву #pragma omp critical? Що станеться з консольним виведенням, якщо її видалити?

4. Які значення повертають функції omp_get_thread_num() та omp_get_num_threads() всередині та ззовні паралельного регіону?

5. Який прапор компілятора є обов'язковим для успішної збірки багатопотокових програм OpenMP за допомогою інструментарію GCC/G++?

Основна література:

1. Жуков І. А., Корочкін О. В. Паралельні обчислення: Навчальний посібник для студентів ВНЗ. — К.: НАУ, 2011. — 250 с.

2. Фельдман Л. П., Дмитрієва О. А. Паралельні обчислення: теорія та практика: Навчальний посібник. — Донецьк: ДонНТУ, 2010. — 190 с.

3. OpenMP Application Programming Interface. Official Specification Version 5.2. — OpenMP Architecture Review Board, 2021. — URL: https://www.openmp.org/specifications/

Додаткова:

- Офіційний сайт та документація стандарту OpenMP: https://www.openmp.org/
- Довідник з директив паралелізму OpenMP у компіляторі GCC G++: https://gcc.gnu.org/onlinedocs/libgomp/
- Навчальні матеріали та туторіали з Visual Studio Code для розробників C/C++: https://code.visualstudio.com/docs/cpp/config-mingw

ЛАБОРАТОРНА РОБОТА №2

Тема: Дослідження ефективності паралельних обчислень при знаходженні визначених інтегралів (на прикладі розрахунку числа π) та аналіз метрик прискорення за законом Амдала.

Мета роботи: Ознайомитися з концепцією паралельної редукції в екосистемі OpenMP та методами чисельного інтегрування. Набути практичних навичок оцінки часової ефективності багатопотокових програм, навчитися розраховувати коефіцієнти прискорення (S) та ефективності (E) паралельних систем, а також проводити теоретичний аналіз обмежень алгоритмів за законом Амдала.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА АНАЛІТИЧНІ МЕТРИКИ

Однією з класичних задач обчислювальної математики є знаходження значень визначених інтегралів. Математично число π може бути представлене як значення наступного інтеграла:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

При чисельному інтегруванні метод прямокутників розбиває відрізок [0, 1] на величезну кількість мікро-кроків (N). Кожен крок виконує ітераційне додавання площ. Оскільки обчислення кожної ітерації циклу не залежить від

попередньої, такий алгоритм є ідеальним кандидатом для паралельного виконання.

Механізм паралельної редукції (reduction)

При спробі розпаралелити такий цикл виникає проблема: всі потоки намагаються одночасно записувати значення в одну спільну змінну sum. Це викликає стан гонитви за даними (*Data Race*). Для уникнення цієї критичної помилки без втрати швидкості в OpenMP використовується прапорець **reduction(операція:змінна)**.

Коли компілятор бачить директиву #pragma omp parallel for reduction(+:sum), він виконує наступні кроки:

1. Для кожного окремого обчислювального потоку створюється своя локальна, ізольована копія змінної sum.
2. Кожен потік паралельно прораховує виділену йому частину кроків циклу у свій локальний буфер.
3. На виході з паралельного регіону система автоматично, на апаратному рівні, синхронізує та підсумовує локальні результати всіх потоків у загальну глобальну змінну sum.

Метрики оцінки ефективності паралельних обчислень

Для оцінки якості розробленого паралельного коду використовуються три базові взаємопов'язані аналітичні метрики:

Прискорення (S_p): коефіцієнт, що показує, у скільки разів програма на p потоках працює швидше, ніж її суцільно послідовний аналог (T_1):

$$S_p = \frac{T_1}{T_p}$$

де T_1 – час виконання алгоритму на 1 потоці, а T_p – час виконання на p паралельних потоках.

Ефективність (E_p): коефіцієнт, що відображає ступінь корисного завантаження ядер процесора під час обчислень:

$$E_p = \frac{S_p}{p} \times 100\%$$

Ідеальне значення дорівнює 100%, проте через накладні витрати ОС на створення та синхронізацію потоків реальний показник завжди менший.

1. **Закон Амдала:** визначає теоретичну межу максимального прискорення паралельної програми залежно від частки послідовного коду:

$$S_{max} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

де α – частка алгоритму, яка не може бути розпаралелена (послідовна частина), а p – кількість ядер/потоків.

2. НАВЧАЛЬНИЙ ШАБЛОН-ЗАГОТОВКА ПРОГРАМИ (lab2_pi.cpp)

Студенту необхідно використати надану кодову базу, яка містить функцію фіксації високоточного паралельного часу omp_get_wtime(). Програма фіксує часові характеристики для подальшого розрахунку аналітичних метрик.

```
C++
#include <iostream>
#include <cstdio>
#include <iomanip>
#include <omp.h>

int main() {
  // Величезна кількість кроків для максимального завантаження процесора
      const long long num_steps = 500000000;
      double step = 1.0 / (double)num_steps;
      double sum = 0.0;
```

```
// 1. Фіксація часу початку обчислень
  double start_time = omp_get_wtime();

// 2. Встановлення кількості паралельних потоків за варіантом
  omp_set_num_threads(2); // (Студент змінює параметр p під час дослідження)

// 3. Паралельний блок OpenMP з редукцією
#pragma omp parallel for reduction(+:sum)
  for (long long i = 0; i < num_steps; i++) {
      double x = (i + 0.5) * step;
      sum += 4.0 / (1.0 + x * x);
  }

double pi = step * sum;

// 4. Фіксація часу завершення обчислень
  double end_time = omp_get_wtime();
  double execution_time = end_time - start_time;

// Виведення високоточних аналітичних результатів латиницею
  std::cout << std::fixed << std::setprecision(15);
  std::cout << "=== Lab 2 Output Results ===" << std::endl;
  std::cout << "Calculated Pi value: " << pi << std::endl;
  std::cout << "Execution time: " << execution_time << " seconds." << std::endl;

  return 0;
}
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ ТА ПОБУДОВА ГРАФІКІВ

1. Перенести структуру навчального шаблону в сирцевий файл lab2_pi.cpp у межах вашої робочої папки VS Code.

2. Провести чисту компіляцію коду з прапорцем багатопотоковості:

Bash

g++ -fopenmp lab2_pi.cpp -o lab2_pi.exe

3. **Етап дослідження:**

Студент повинен провести серію послідовних тестів, змінюючи у коді функцію omp_set_num_threads(p) відповідно до свого індивідуального варіанта. Необхідно зафіксувати точний час виконання (T_p) для кожної конфігурації.

4. За отриманими даними розрахувати реальне прискорення та реальну ефективність.

5. Спираючись на вказану у варіанті теоретичну частку послідовного коду (α), розрахувати теоретичну межу прискорення за Законом Амдала.

6. За побудованими табличними даними розрахунків створити в MS Word або Excel **два графіки**:

- Графік залежності реального та теоретичного прискорення від кількості потоків: S_p .
- Графік залежності ефективності обчислень від кількості потоків: E_p.

7. Зробити висновок щодо масштабованості алгоритму Монте-Карло / прямокутників на вашій обчислювальній системі.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Кожен студент виконує дослідження для вказаного набору обчислювальних потоків (p) та порівнює практичні результати з

теоретичною межею закону Амдала для заданої послідовної частки коду (α).

Варіант	**Набір потоків для тестування (p)**	**Послідовна частка алгоритму (α)**
1	p = 1, 2, 3, 4	α= 0.05(5%)
2	p = 1, 2, 4, 6	α= 0.08(8%)
3	p = 1, 2, 4, 8	α= 0.10(10%)
4	p = 1, 2, 3, 6	α= 0.04(4%)
5	p = 1, 2, 4, 5	α= 0.12(12%)
6	p = 1, 2, 4, 7	α= 0.06(6%)
7	p = 1, 2, 6, 8	α= 0.15(15%)
8	p = 1, 2, 4, 12	α= 0.02(2%)
9	p = 1, 2, 5, 10	α= 0.07(7%)
10	p = 1, 2, 3, 5	α= 0.09(9%)

5. КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. Що таке стан гонитви за даними (*Data Race*) і чому звичайне сумування елементів у змінну всередині багатопотокового циклу призводить до невірного значення числа π?

2. Яким чином прапорець паралельної редукції reduction(+:sum) вирішує проблему синхронізації пам'яті ядер? Опишіть алгоритм його роботи.

3. Чому реальне прискорення паралельної програми (S_p) майже ніколи не досягає ідеального лінійного показника, що дорівнює кількості потоків (p)?

4. Сформулюйте закон Амдала. Який головний філософський висновок можна зробити з цього закону щодо нарощування кількості ядер у процесорі?

5. Опишіть призначення та принцип роботи високоточної функції фіксації часу omp_get_wtime(). Які одиниці виміру вона повертає?

Основна література:

1. Воронова Л. М., Глибовець М. М. Паралельні обчислення: Навчальний посібник. — К.: Видавничий дім «Києво-Могилянська академія», 2007. — 160 с.

2. Сергієнко А. М. Архітектура обчислювальних систем: Підручник. — К.: ТМЦ «Варіант», 2014. — 232 с.

3. Chandra R., Dagum L., Kohr D., Maydan D., McDonald J., Menon R. Parallel Programming in OpenMP. — Morgan Kaufmann, 2001. — 240 p.

Додаткова:

- Приклади використання операцій паралельної редукції (OpenMP Reduction) на офіційному порталі розробників: https://www.openmp.org/resources/tutorials-articles/
- Lawrence Livermore National Laboratory. OpenMP Tutorial (High-Performance Computing training): https://hpc.llnl.gov/training/tutorials/openmp-tutorial
- Довідник функцій високоточного таймінгу та продуктивності у мовах C/C++: https://en.cppreference.com/w/cpp/chrono

ЛАБОРАТОРНА РОБОТА №3

Тема: Дослідження апаратного паралелізму на рівні процесора за допомогою технології SIMD та векторних регістрів MMX.

Мета роботи: Ознайомитися з архітектурою SIMD (одна інструкція — множина даних) та принципами пакування інформації у векторні регістри центрального процесора. Набути практичних навичок оголошення векторних типів даних у C++, реалізації покомпонентних паралельних операцій над масивами на рівні заліза та аналізу продуктивності низькорівневих обчислень.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА АРХІТЕКТУРА SIMD / MMX

Традиційні обчислення базуються на архітектурі **SISD** (Single Instruction, Single Data), де одна математична інструкція процесора обробляє лише один елемент даних за такт (наприклад, додавання двох чисел a + b). При роботі з великими масивами або аналітичними векторами такий підхід створює чергу з інструкцій.

Технологія **SIMD (Single Instruction, Multiple Data)** кардинально змінює цей принцип. Вона дозволяє процесору виконати одну єдину апаратну операцію над цілим блоком (вектором) незалежних даних одночасно.

Розширення MMX (Multimedia Extensions)

Історично першим широким впровадженням SIMD-архітектури в процесорах x86 стало розширення MMX. Його суть полягає в наступному:

1. Процесор надає 64-бітні векторні регістри.
2. У цей один регістр можна «упакувати» декілька дрібніших елементів даних. Наприклад, 8 окремих однобайтових чисел (тип unsigned char.
3. Коли процесор виконує апаратну команду додавання, він за один такт паралельно додає всі 8 елементів першого регістра до 8 елементів другого регістра, повністю виключаючи потребу в класичному циклі for.

Векторні атрибути компілятора GCC

Сучасні компілятори C++ (зокрема GCC у складі MSYS2) дозволяють не писати складний код на асемблері, а використовувати спеціальний атрибут сумісності з векторами:

typedef unsigned char v8qi __attribute__ ((vector_size (8)));

Цей рядок створює новий тип даних v8qi, який операційна система розглядає як один упакований 64-бітний вектор, що містить 8 елементів типу char. Операція result = vec1 + vec2; на рівні заліза виконується як одна паралельна SIMD-інструкція.

2. НАВЧАЛЬНИЙ ШАБЛОН-ЗАГОТОВКА ПРОГРАМИ (lab3_mmx.cpp)

Студенту необхідно використати цей еталонний каркас програми, що демонструє роботу з упакованими 64-бітними регістрами. Потрібно змінити початкові вектори та математичну операцію відповідно до свого індивідуального варіанта.

```
C++
#include <iostream>
#include <cstdio>
#include <iomanip>
```

// Визначаємо векторний тип: 8 упакованих байтів (8 біт * 8 = 64 біти — класичний регістр MMX)

```
typedef unsigned char v8qi __attribute__ ((vector_size (8)));

int main() {
    std::cout << "=== Lab 3: SIMD / MMX Architecture Technology
===" << std::endl;
```

// 1. Ініціалізація двох векторів по 8 байтів у кожному (значення відповідно до варіанта)

```
    v8qi vec1 = {10, 20, 30, 40, 50, 60, 70, 80};
    v8qi vec2 = {1,  2,  3,  4,  5,  6,  7,  8};
```

// 2. Векторна операція — ОДНА команда процесора обробляє всі 8 елементів паралельно!

/* ДОПИСАТИ ТУТ: Математична операція за варіантом (додавання, віднімання або множення) */

```
        v8qi result = vec1 + vec2;
```

// 3. Виведення результатів обчислень латиницею

```
          std::cout << "Vector 1: ";
          for (int i = 0; i < 8; i++) std::cout << (int)vec1[i] << "
";
          std::cout << std::endl;

          std::cout << "Vector 2: ";
          for (int i = 0; i < 8; i++) std::cout << (int)vec2[i] << "
";
          std::cout << std::endl;

          std::cout << "-----------------------------------------
------" << std::endl;
          std::cout << "Parallel SIMD Result: ";
          for (int i = 0; i < 8; i++) {
              std::cout << (int)result[i] << " ";
          }
          std::cout << std::endl;

          return 0;
      }
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Створити у вашій робочій папці VS Code новий файл сирцевого коду lab3_mmx.cpp.
2. Скопіювати структуру наданого навчального шаблону.
3. Модифікувати масиви vec1 та vec2, заповнивши їх значеннями відповідно до умов вашого індивідуального варіанта з таблиці.

4. Замінити базове додавання на арифметичну операцію (векторне додавання, віднімання чи покомпонентне множення), яка вказана у завданні.

5. Скомпілювати код за допомогою стандартного термінала bash (підтримка SIMD/MMX-атрибутів вшита в компілятор за замовчуванням):

Bash

```
g++ lab3_mmx.cpp -o lab3_mmx.exe
```

6. Запустити виконуваний файл програми:

Bash

```
./lab3_mmx.exe
```

7. Зафіксувати отримані результати векторного розрахунку у вікні консолі та оформити звіт з детальним аналізом паралельної обробки елементів без використання циклів.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Кожен студент ініціалізує два 8-елементні вектори (vec1 та vec2) вказаними безпосередніми константами й реалізує апаратну SIMD-операцію.

Варіант	Початкові значення vec1 (8 байт)	Початкові значення vec2 (8 байт)	Апаратна операція
1	{10, 20, 30, 40, 50, 60, 70, 80}	{1, 2, 3, 4, 5, 6, 7, 8}	Додавання (+)
2	{100, 120, 140, 160, 180, 200, 220, 240}	{10, 20, 30, 40, 50, 60, 70, 80}	Віднімання (-)
3	{5, 10, 15, 20, 25, 30, 35, 40}	{2, 2, 2, 2, 3, 3, 3, 3}	Додавання (+)
4	{12, 24, 36, 48, 60, 72, 84, 96}	{5, 6, 7, 8, 9, 10, 11, 12}	Множення (*)

5	{50, 60, 70, 80, 90, 100, 110, 120}	{4, 4, 4, 4, 5, 5, 5, 5}	Віднімання (-)
6	{2, 4, 6, 8, 10, 12, 14, 16}	{10, 15, 20, 25, 30, 35, 40, 45}	Додавання (+)
7	{15, 30, 45, 60, 75, 90, 105, 120}	{40, 40, 30, 30, 20, 20, 10, 10}	Множення (*)
8	{250, 220, 190, 160, 130, 100, 70, 40}	{3, 3, 3, 3, 4, 4, 4, 4}	Віднімання (-)
9	{3, 6, 9, 12, 15, 18, 21, 24}	{20, 25, 30, 35, 40, 45, 50, 55}	Додавання (+)
10	{80, 90, 100, 110, 120, 130, 140, 150}	{10, 15, 20, 25, 30, 35, 40, 45}	Множення (*)

5. КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. Чим принципово відрізняється архітектурна модель обчислень SIMD від класичної послідовної моделі SISD?

2. Яку розрядність мають класичні апаратні регістри розширення MMX? Яку максимальну кількість однобайтових елементів типу char туди можна упакувати?

3. Поясніть призначення специфічного атрибуту компілятора GCC __attribute__ ((vector_size (8))).

4. Чому операція result = vec1 + vec2; у даній лабораторній роботі виконується швидше, ніж аналогічне додавання елементів через стандартний цикл for?

5. Що таке «ефект насичення» під час виконання низькорівневих операцій над упакованими байтами, і чому важливо контролювати переповнення типів даних unsigned char?

Основна література:

1. Сергієнко А. М. Архітектура обчислювальних систем: Підручник. — К.: ТМЦ «Варіант», 2014. — 232 с.

2. Мельник А. О. Архітектура комп'ютера. Підручник. — Луцьк: Волинська обласна друкарня, 2008. — 470 с.

3. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture. — Intel Corporation, 2023. — URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Додаткова інформація (Вебресурси):

1. Документація компілятора GCC щодо використання векторних розширень (Using Vector Extensions): https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

2. Довідковий посібник з апаратних SIMD-технологій Intel Intrinsics Guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

3. Навчальний розбір архітектури процесорів x86 на порталі обчислювальних систем: https://www.sciencedirect.com/topics/computer-science/simd-extension

ЛАБОРАТОРНА РОБОТА №4

Тема: Дослідження високопродуктивної геометрії та тривимірних матричних перетворень за допомогою векторних 128-бітних SIMD-розширень типу Float.

Мета роботи: Ознайомитися з принципами функціонування 128-бітних векторних регістрів процесора для обробки дійсних чисел (float). Навчитися реалізовувати просторові SIMD-операції над тривимірними координатами, освоїти низькорівневе масштабування векторів та проаналізувати теоретичні аспекти оптимізації графічних обчислювальних конвеєрів.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА РОЗШИРЕНЕ ПОЯСНЕННЯ СИНТАКСИСУ

У комп'ютерній графіці, ігрових фізичних рушіях та системах моделювання тривимірного простору базовою одиницею даних є тривимірний вектор або координатна точка в однорідній системі координат: (X, Y, Z, W), де W – масштабний коефіцієнт. Для зміни положення об'єкта у просторі (перенесення, масштабування, поворот) над цими чотирма компонентами одночасно потрібно виконувати однотипні математичні операції.

Класичний процесор архітектури SISD обробляє кожну координату по черзі, що потребує мінімум 4 окремих такти обчислень. Сучасні процесори використовують 128-бітні SIMD-регістри (впроваджені через інструкції SSE/AVX та історичні розширення типу 3DNow!), які здатні вмістити рівно чотири 32-бітні числа типу float.

Особливості синтаксису розширення векторів у GCC

Для оголошення 128-бітного векторного типу в сучасних стандартах компілятора GCC використовується такий синтаксис:

typedef float v4sf __attribute__ ((vector_size (16)));

Давай розберемо цей рядок покроково, щоб зрозуміти, як саме він транслюється у машинний код:

1. typedef float – базовим типом для кожного елемента вектора є 32-бітне число з плаваючою комою (стандарт IEEE 754).
2. v4sf – назва нового користувацького типу (традиційно розшифровується як Vector of 4 Single-Precision Floats).
3. __attribute__ ((vector_size (16))) — директива компілятора, яка вказує точний апаратний розмір виділеного блоку пам'яті в байтах.

Оскільки нам потрібно упакувати 4 змінні типу float, а кожна з них займає в оперативній пам'яті суворо 4 байти.

Правила написання коду та математичних операцій:

- **Ініціалізація змінних:**

  На відміну від класичних масивів, векторний тип ініціалізується за допомогою фігурних дужок через пряме присвоєння: v4sf my_vector = {1.0f, 2.0f, 3.0f, 4.0f};. Суфікс f після числа є обов'язковим, оскільки він явно вказує компілятору, що число має тип одинарної точності float, а не подвійної double.

- **Паралельна математика:**

  Коли в коді пишеться бінарна операція над цими типами, наприклад v4sf res = vecA * vecB;, компілятор GCC генерує одну апаратну інструкцію процесора (наприклад, mulps в архітектурі SSE). Процесор перемножує елементи абсолютно одночасно за один єдиний такт заліза.

- **Доступ до елементів:**

  Зчитування фінальних аналітичних даних із векторного типу здійснюється за стандартними квадратними дужками індексації: my_vector[0] (компонента X), my_vector[1] (компонента Y) тощо.

2. НАВЧАЛЬНИЙ ШАБЛОН ПРОГРАМИ (lab4_3now.cpp)

Студенту необхідно самостійно вивчити правила оголошення векторних типів, повністю реалізувати блоки виділення пам'яті, заповнити вектори просторових координат та масштабування згідно із завданням, а також написати логіку покомпонентного виведення результатів обчислень у консоль.

```
C++
#include <iostream>
#include <cstdio>
#include <iomanip>
```

// 1. ОГОЛОШЕННЯ ВЕКТОРНОГО ТИПУ ДЛЯ 4-Х ЧИСЕЛ ТИПУ FLOAT (128 БІТ)

/* ДОПИСАТИ ТУТ: Визначення типу v4sf з атрибутом vector_size на 16 байт */

```
int main() {
    std::cout << "=== Lab 4: SIMD / Float Geometry Architecture
===" << std::endl;
    std::cout << std::fixed << std::setprecision(2);
```

// 2. ІНІЦІАЛІЗАЦІЯ ВЕКТОРІВ КООРДИНАТ ТА КОЕФІЦІЄНТІВ ЗА ВАРІАНТОМ

/* ДОПИСАТИ ТУТ: Оголосити та заповнити просторовий вектор point_coordinates */

// Сюди записуються початкові просторові дані {X, Y, Z, W}

/* ДОПИСАТИ ТУТ: Оголосити та заповнити вектор коефіцієнтів зміщення scale_factors */

// Сюди записуються матричні коефіцієнти зміщення/масштабування

// 3. НИЗЬКОРІВНЕВА ВЕКТОРНА SIMD-ОПЕРАЦІЯ ОДНІЄЮ КОМАНДОЮ

// Процесор має перемножити всі чотири просторові координати за один такт

/* ДОПИСАТИ ТУТ: Реалізувати векторне множення та записати його у transformed_point */

// 4. САМОСТІЙНА РЕАЛІЗАЦІЯ ВИВЕДЕННЯ РЕЗУЛЬТАТІВ У КОНСОЛЬ

// Студент має написати цикли або пряме виведення елементів векторів латиницею

/* ДОПИСАТИ ТУТ: Вивести початкові координати point_coordinates */

/* ДОПИСАТИ ТУТ: Вивести коефіцієнти масштабування scale_factors */

/* ДОПИСАТИ ТУТ: Вивести кінцевий аналітичний вектор transformed_point */

```
        std::cout << "-----------------------------------------
------" << std::endl;
        return 0;
    }
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Опрацювати теоретичний матеріал щодо використання 128-бітних регістрів та специфіки розширення форматів обробки геометрії float.
2. Створити у вашій робочій папці сирцевий файл lab4_3now.cpp.

3. Дописати у шаблон визначення типу v4sf, заповнити координати початкового просторового об'єкта point_coordinates та вектор просторового матричного зсуву scale_factors відповідно до умов вашого варіанта з таблиці.

4. Написати математичний вираз паралельного SIMD-множення векторів та створити логіку виведення вхідних і вихідних компонентів через цикли індексації.

5. Виконати чисту компіляцію програми за допомогою нативного термінала bash:

Bash

```
g++ lab4_3now.cpp -o lab4_3now.exe
```

6. Запустити згенерований файл на виконання:

Bash

```
./lab4_3now.exe
```

Переконатися за логами, що всі координати пройшли апаратне перетворення коректно, зафіксувати результати обчислень і додати їх у звіт.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Кожен студент заповнює 4-компонентні просторові вектори дійсних чисел одинарної точності константами свого варіанта та обчислює фінальний вектор геометричної трансформації об'єкта.

Варіант	Початкові просторові координати об'єкта {X, Y, Z, W}	Коефіцієнти зміщення системи $\{S_x, S_y, S_z, S_w\}$
1	{1.5f, 2.5f, 3.5f, 1.0f}	{2.0f, 2.0f, 2.0f, 1.0f}
2	{0.5f, 4.0f, -1.5f, 2.0f}	{4.0f, 0.5f, 2.0f, 1.0f}
3	{10.2f, 5.5f, 1.1f, 1.0f}	{0.5f, 2.0f, 10.0f, 1.0f}
4	{-3.0f, -6.0f, 9.0f, 3.0f}	{3.0f, 3.0f, 0.33f, 1.0f}

5	{1.2f, 2.2f, 3.2f, 1.0f}	{5.0f, 5.0f, 5.0f, 2.0f}
6	{7.5f, 0.0f, 15.0f, 5.0f}	{2.0f, 9.0f, 0.2f, 0.2f}
7	{2.25f, 1.5f, 4.5f, 1.0f}	{4.0f, 6.0f, 2.0f, 1.0f}
8	{-1.0f, 8.2f, 3.6f, 2.5f}	{10.0f, 2.0f, 5.0f, 2.0f}
9	{4.4f, 3.3f, 2.2f, 1.1f}	{2.5f, 2.5f, 2.5f, 1.0f}
10	{0.8f, 1.6f, 2.4f, 1.0f}	{1.25f, 1.25f, 1.25f, 3.0f}

5. КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. Яку бітову розрядність мають сучасні SIMD-регістри для обробки дійсних чисел? Скільки змінних типу float вони здатні вмістити?

2. Яким чином розраховується параметр розширення vector_size(16) для типу даних v4sf? Що зміниться, якщо перейти на використання типу double?

3. Навіщо при ініціалізації елементів вектора в коді мови C++ обов'язково вказувати суфікс f після числового значення?

4. Опишіть переваги використання SIMD-інструкцій при виконанні матричних та векторних перетворень у задачах комп'ютерної 3D-графіки.

5. Чому за допомогою однієї SIMD-команди не можна одночасно виконати додавання для перших двох компонентів вектора та множення для наступних двох?

Основна література:

1. Сергієнко А. М. Архітектура обчислювальних систем: Підручник. — К.: ТМЦ «Варіант», 2014. — 232 с.

2. Панич Ю. В. Комп'ютерна графіка: Навчальний посібник. — Львів: Новий Світ-2000, 2011. — 340 с.

3. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2A: Instruction Set Reference, A-M. — Intel Corporation, 2023. — URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Додаткова:

1. Специфікація розширення векторних типів даних у компіляторі GCC (GNU Compiler Vector Extensions): https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html

2. Посібник з оптимізації математичних обчислень за допомогою SSE/AVX інструкцій: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

3. Базовий курс з однорідних координат та матричних перетворень у 3D-просторі: https://learnopengl.com/Getting-started/Transformations

ДОДАТОК Б

ЛАБОРАТОРНА РОБОТА №5

Тема: Дослідження архітектури розподіленої пам'яті та побудова двоточкових комунікацій між паралельними процесами за допомогою технології MPI.

Мета роботи: Ознайомитися з принципами функціонування паралельних розподілених систем та базовим інтерфейсом передачі повідомлень (Message Passing Interface). Набути практичних навичок ініціалізації паралельного середовища, ідентифікації обчислювальних вузлів за їхніми рангами та реалізації парного обміну даними за топологією «Точка-Точка» (*Point-to-Point*).

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА ПОЯСНЕННЯ НАПИСАННЯ КОДУ

На відміну від технології OpenMP, де всі потоки працюють в одному процесі й ділять між собою спільну оперативну пам'ять, технологія MPI (Message Passing Interface) орієнтована на системи з розподіленою пам'яттю (кластери, суперкомп'ютери).

В MPI кожна гілка обчислень є абсолютно ізольованим процесом софтверного або апаратного рівня. Кожен такий процес має власну унікальну карту адресного простору і не може прямо зчитувати або перезаписувати змінні інших процесів. Будь-яка координація дій та передача аналітичних параметрів відбуваються виключно через явне надсилання та прийом пакетів повідомлень.

Особливості розгалуження коду за моделлю SPMD

В MPI використовується парадигма проектування SPMD (Single Program, Multiple Data). Один і той самий скомпільований файл запускається одночасно у вигляді N копій (процесів). Щоб змусити їх виконувати різні ролі, всередині коду виконується опитування параметрів віртуального комунікатора:

- MPI_Comm_size – записує в змінну загальну кількість запущених процесів системи.
- MPI_Comm_rank – записує унікальний номер (ID) поточного процесу – його ранг (значення від 0 до N-1). Процес із рангом 0 традиційно стає керуючим вузлом (Master), а всі інші – робочими (Workers).

Розгалуження логіки пишеться через класичні умовні оператори мови C++:

```
C++
if (world_rank == 0) {
   // Код виконується тільки Керуючим вузлом
} else if (world_rank == 1) {
   // Код виконується тільки першим Робочим вузлом
}
```

Правила написання функцій прийому-передачі повідомлень

Для передачі однієї змінної між двома конкретними процесами («точка-точка») використовуються парні функції MPI_Send та MPI_Recv.

1. Синтаксис функції відправлення (MPI_Send):

```
MPI_Send(&variable, count, datatype, destination, tag, communicator);
```

- &variable – адреса локальної змінної, дані якої потрібно упакувати в повідомлення.
- count – кількість елементів для відправки (для одного числа дорівнює 1).

- datatype – тип даних стандарту MPI (наприклад, MPI_INT для цілих чисел або MPI_DOUBLE для дійсних).
- destination – ранг процесу-отримувача, якому летить пакет (наприклад, 1).
- tag – числовий маркер (тег) повідомлення (ціле число для фільтрації пакетів).
- communicator – простір зв'язку (за замовчуванням глобальний: MPI_COMM_WORLD).

2. Синтаксис функції прийому (MPI_Recv):

MPI_Recv(&variable, count, datatype, source, tag, communicator, status);

- Параметри повторюють відправлення, де source – ранг процесу-відправника (наприклад, 0), від якого очікуються дані, а status – покажчик на структуру звіту (MPI_STATUS_IGNORE, якщо деталі не потрібні).

Важливо: Обидві функції є блокуючими. Процес-відправник не піде далі по коду, поки дані безпечно не скопіюються в системний буфер, а процес-отримувач повністю зупинить своє виконання на рядку MPI_Recv і чекатиме, поки потрібний пакет фізично не надійде по каналах зв'язку операційної системи. Якщо порушити парність викликів, система увійде в стан перманентного зациклення – Deadlock.

2. МІНІМІЗОВАНИЙ НАВЧАЛЬНИЙ ШАБЛОН ПРОГРАМИ (lab5_mpi.cpp)

Студенту необхідно самостійно вивчити архітектуру передачі повідомлень точка-точка, реалізувати умовні блоки розподілу ролей для процесів з рангом 0

та рангом 1, а також правильно викликати інтерфейсні функції MPI_Send та MPI_Recv з коректним зазначенням типів даних.

```
C++
#include <iostream>
#include <cstdio>
#include <mpi.h>

int main(int argc, char** argv) {
  // 1. Ініціалізація паралельного середовища MPI
  MPI_Init(&argc, &argv);

    int world_size = 0;
    int world_rank = -1;


  // Опитування параметрів глобального комунікатора
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Використовуємо printf та fflush для стабільного паралельного виведення у Windows
    std::printf("=== MPI Node %d of %d is running ===\n", world_rank, world_size);
    std::fflush(stdout);


  // 2. РЕАЛІЗАЦІЯ ДВОТОЧКОВОЇ ВЗАЄМОДІЇ ЗА ВАРІАНТОМ
  if (world_rank == 0) {
    // ====================================================
    // ЛОГІКА КЕРУЮЧОГО ВУЗЛА (NODE 0) - ВІДПРАВНИК
    // ====================================================
    int message_data = 2026; // (Студент змінює значення згідно з варіантом)


    std::printf("[MPI Node 0]: Sended analytical parameter (%d) to Node 1.\n", message_data);
```

```
            std::fflush(stdout);

        // Захист: перевіряємо, чи запущено в системі більше ніж 1 процес
        if (world_size > 1) {
          /* ДОПИСАТИ ТУТ: Викликати MPI_Send для надсилання message_data до Вузла 1 */
        }
        }
        else if (world_rank == 1) {
        // ========================================================
        // ЛОГІКА РОБОЧОГО ВУЗЛА (NODE 1) - ОТРИМУВАЧ
        // ========================================================
            int received_data = 0;
        /* ДОПИСАТИ ТУТ: Викликати блокуючий прийом MPI_Recv від Вузла 0 */

        // Виведення та верифікація прийнятих аналітичних даних
        std::printf("[MPI Node 1]: Success! Received parameter: %d\n", received_data);
            std::fflush(stdout);
        }

      // 3. Завершення роботи паралельного середовища
        MPI_Finalize();
        return 0;
    }
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Опрацювати теоретичні відомості щодо парадигми SPMD та блокуючих Point-to-Point комунікацій.

2. Створити у вашій робочій папці сирцевий файл lab5_mpi.cpp та інтегрувати туди каркас навчального шаблону.

3. У блоці world_rank == 0 ініціалізувати ціле число message_data відповідно до свого варіанта з таблиці та дописати функцію MPI_Send.

4. У блоці world_rank == 1 реалізувати приймальний буфер та дописати функцію MPI_Recv для зчитування надісланого параметра.

5. Відкрити інтегрований термінал розробника у режимі bash та виконати нативну компіляцію:

Bash

```
mpic++ lab5_mpi.cpp -o lab5_mpi.exe
```

6. Виконати паралельний запуск обчислювального кластера строго на 2 процесах, використовуючи системний шлях до менеджера Microsoft MPI:

Bash

```
"/c/Program Files/Microsoft MPI/Bin/mpiexec.exe" -n 2 ./lab5_mpi.exe
```

7. Зафіксувати отримані текстові логи обміну даними у вікні консолі та оформити звіт.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Кожен студент ініціалізує аналітичний параметр на Вузлі 0 унікальним числовим значенням свого варіанта, реалізує його апаратне пересилання через канали зв'язку ОС та логує успішний прийом на Вузлі 1.

Варіант	Числове значення параметра на Вузлі 0 (message_data)	Тег повідомлення (tag)
1	2026	5
2	1024	10

3	777	1
4	5005	7
5	1234	12
6	9999	3
7	456	9
8	8080	2
9	1111	4
10	3456	3

КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. Чим принципово відрізняється модель організації пам'яті в технологіях OpenMP та MPI?

2. Опишіть суть парадигми програмування SPMD (Single Program, Multiple Data).

3. Що таке ранг процесу (Rank) в MPI, які значення він може приймати і для чого він використовується в коді?

4. Чому функції MPI_Send та MPI_Recv називають блокуючими? До якого моменту процес-отримувач призупиняє свою роботу?

5. Що таке стан Deadlock у розподілених обчисленнях і яка помилка при написанні Point-to-Point функцій може його викликати?

Основна література:

1. Жуков І. А., Корочкін О. В. Паралельні обчислення: Навчальний посібник для студентів ВНЗ. — К.: НАУ, 2011. — 250 с.

2. Фельдман Л. П., Дмитрієва О. А. Паралельні обчислення: теорія та практика: Навчальний посібник. — Донецьк: ДонНТУ, 2010. — 190 с.

3. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. 3rd Edition. — MIT Press, 2014. — 480 p.

Додаткова:

1. Офіційна специфікація та документація стандарту MPI (Message Passing Interface Forum): https://www.mpi-forum.org/docs/

2. Документація та інструкції з розгортання Microsoft MPI (MS-MPI) для Windows: https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi

3. Навчальний посібник з Point-to-Point комунікацій MPI на порталі високопродуктивних обчислень: https://mpitutorial.com/tutorials/mpi-send-and-receive/

ЛАБОРАТОРНА РОБОТА №6

Тема: Дослідження колективних операцій групової взаємодії та синхронізації процесів у розподілених обчислювальних системах за допомогою технології MPI.

Мета роботи: Ознайомитися з концепцією та класифікацією колективних операцій у стандарті MPI. Набути практичних навичок одночасної трансляції аналітичних даних (Broadcast) від головного вузла до робочих, реалізувати локальні паралельні обчислення функцій та виконати колективне апаратне згортання (Reduction) результатів на Master-вузлі.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА РОЗШИРЕНЕ ПОЯСНЕННЯ СИНТАКСИСУ

У реальних високопродуктивних обчисленнях алгоритми часто вимагають взаємодії не просто між двома ізольованими процесами (як було у Лабораторній №5), а між усіма учасниками обчислювального кластера одночасно. Використання лінійних функцій MPI_Send та MPI_Recv у циклах для таких задач є вкрай неефективним, оскільки воно створює послідовну чергу обміну й перевантажує комунікаційне середовище. Для оптимізації групової взаємодії в MPI передбачено колективні операції.

Жорсткі правила колективної взаємодії:

1. **Синхронність виклику:**

   Колективна функція повинна бути викликана **абсолютно всіма** процесами, які входять до зазначеного комунікатора (наприклад, MPI_COMM_WORLD). Якщо хоча б один процес не дійде до цього рядка коду (наприклад, через помилку в умовному операторі), уся паралельна система зациклиться й увійде в стан глобального взаємного блокування (Deadlock).

2. **Відсутність ідентифікаторів пакетів:**

   На відміну від Point-to-Point функцій, колективні операції не використовують параметри тегів (tag). Робота обчислювальних вузлів координується виключно за рахунок черговості виклику функцій у загальному коді.

Пояснення синтаксису досліджуваних функцій:

1. **Широкомовна трансляція (MPI_Bcast):**

   MPI_Bcast(&buffer, count, datatype, root, communicator);

   - &buffer – адреса змінної в пам'яті. На процесі з рангом root звідси беруться дані для відправки, а на всіх інших процесах сюди записуються прийняті дані.

- count та datatype – кількість та тип даних (наприклад, 1, MPI_DOUBLE).
- root – ранг процесу, який є першоджерелом даних і розсилає їх усім іншим (традиційно – Ранг 0).

2. **Глобальне апаратне згортання (MPI_Reduce):**

   MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, communicator);

   - &sendbuf – адреса локальної змінної процесу, де зберігається його індивідуально обчислений результат розрахунку функції.
   - &recvbuf – адреса змінної, куди буде записано підсумковий агрегований результат. Пам'ять під цю змінну має реальне значення тільки на процесі з рангом root.
   - op – тип математичної або логічної операції, за якою система автоматично склеюватиме дані з усіх вузлів (наприклад, MPI_SUM – сумування результатів, MPI_MAX – пошук максимуму, MPI_MIN – пошук мінімуму).

2. НАВЧАЛЬНИЙ ШАБЛОН ПРОГРАМИ (lab6.cpp)

Студенту необхідно самостійно розібратися у механізмах колективного розподілу пам'яті, інтегрувати функцію трансляції початкового параметра, реалізувати обчислення математичної формули за своїм варіантом з урахуванням унікального рангу вузла, та дописати функцію глобального згортання даних на Master-вузлі.

```
C++
#include <iostream>
#include <cstdio>
#include <cmath>
```

```
#include <mpi.h>

int main(int argc, char** argv) {
  // 1. Ініціалізація паралельного обчислювального середовища
  MPI_Init(&argc, &argv);

    int world_size = 0;
    int world_rank = -1;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);


  // Захисний механізм перевірки кількості виділених вузлів
  if (world_size < 2) {
        if (world_rank == 0) {
            std::printf("[ERROR]: Run with at least 2 processes
(-n 2)!\n");
            std::fflush(stdout);
        }
        MPI_Finalize();
        return 1;
    }

    double shared_data = 0.0;

    if (world_rank == 0) {
    // Логіка Керуючого вузла (Варіант змінюється відповідно до таблиці)
        shared_data = 5.4;
        std::printf("[Node 0]: Initialized parameter x = %f\n",
shared_data);
        std::fflush(stdout);
    }


  // 2. КОЛЕКТИВНА ОПЕРАЦІЯ ТРАНСЛЯЦІЇ ПАРАМЕТРА (BROADCAST)

  /* ДОПИСАТИ ТУТ: Викликати MPI_Bcast для розсилки значення shared_data від Вузла 0 до всіх */
```

// 3. ЛОКАЛЬНІ ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ НА КОЖНОМУ ВУЗЛІ ЗА ВАРІАНТОМ

// Студент має самостійно реалізувати математичну функцію варіанта L = f(shared_data, world_rank)

double local_result = 0.0;

/* ДОПИСАТИ ТУТ: Формула обчислення локального індексу за варіантом */

// Приклад: local_result = shared_data * (world_rank + 1) + std::cos(world_rank);

```
        std::printf("[Node %d]: Received x = %f, Calculated L = %f\n", world_rank, shared_data, local_result);
        std::fflush(stdout);

        double global_summary = 0.0;
```

// 4. КОЛЕКТИВНА ОПЕРАЦІЯ ЗГОРТАННЯ РЕЗУЛЬТАТІВ (REDUCE)

/* ДОПИСАТИ ТУТ: Викликати MPI_Reduce для агрегації local_result в global_summary на Вузол 0 */

// 5. ВИВЕДЕННЯ ФІНАЛЬНИХ АГРЕГОВАНИХ РЕЗУЛЬТАТІВ КЕРУЮЧИМ ВУЗЛОМ

```
        if (world_rank == 0) {

std::printf("================================================\n");
```

/* ДОПИСАТИ ТУТ: Вивести значення глобального показника системи global_summary латиницею */

```
std::printf("================================================\n")
;
            std::fflush(stdout);
        }


    // 6. Завершення паралельної сесії
        MPI_Finalize();
        return 0;
    }
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Опрацювати теоретичний базис щодо групових комунікацій в MPI та відмінностей між операціями Broadcast та Reduce.
2. Створити у вашій робочій папці сирцевий файл lab6.cpp та перенести туди каркас мінімізованого шаблону.
3. Додати функцію MPI_Bcast для передачі ініціалізованого параметра shared_data типу MPI_DOUBLE.
4. У блоці паралельних обчислень реалізувати математичну формулу розрахунку проміжного значення L, яка залежить від константи x та індивідуального номера процесу (world_rank).
5. Реалізувати колективне згортання розрахованих даних через функцію MPI_Reduce на Вузлі 0 (тип математичної операції – сума MPI_SUM, максимум MPI_MAX чи мінімум MPI_MIN – обрати суворо за варіантом).
6. Відкрити інтегровану консоль у режимі **bash** та виконати збірку проєкту:

Bash

```
mpic++ lab6.cpp -o lab6.exe
```

7. Виконати паралельний запуск системи на 2 або більше обчислювальних процесах:

Bash

```
"/c/Program Files/Microsoft MPI/Bin/mpiexec.exe" -n 2 ./lab6.exe
```

8. Зафіксувати логи консолі латиницею, переконатися у точності фінального агрегованого показника системи й оформити звіт.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Кожен студент ініціалізує глобальний параметр x на Вузлі 0 згідно із варіантом. Кожен активований потік вираховує свій локальний індекс L за формулою, де R – ранг поточного процесу (world_rank). За допомогою операції згортання на Головному вузлі накопичується підсумковий показник системи G.

Варіант	**Початкове значення на Вузлі 0 (x)**	**Локальна функція розрахунку вузла (L)**	**Обов'язкова операція згортання (G)**
1	x = 5.4	L(x) = sin(x) * cos(2*x) + 1.5 * sin(3*x)	Сума всіх елементів (MPI_SUM)
2	x = 2.1	L(x) = 2.5 * x^3 - 1.2 * x^2 + 0.8 * x - 0.05	Пошук максимуму (MPI_MAX)
3	x = 0.85	L(x) = exp(-x) * sin(4*x) + cos(x)^2	Сума всіх елементів (MPI_SUM)
4	x = 10.0	L(x) = log(x + 1) * cos(x) + 0.5 * x	Пошук мінімуму (MPI_MIN)
5	x = 3.3	L(x) = sin(x) * cos(2*x) + 1.5 * sin(3*x)	Сума всіх елементів (MPI_SUM)
6	x = 1.5	L(x) = 2.5 * x^3 - 1.2 * x^2 + 0.8 * x - 0.05	Пошук максимуму (MPI_MAX)

7	x = 4.7	L(x) = exp(-x) * sin(4*x) + cos(x)^2	x - R
8	x = 0.5	L(x) = log(x + 1) * cos(x) + 0.5 * x	Пошук мінімуму (MPI_MIN)
9	x = 2.8	L(x) = sin(x) * cos(2*x) + 1.5 * sin(3*x)	Сума всіх елементів (MPI_SUM)
10	x = 7.2	L(x) = 2.5 * x^3 - 1.2 * x^2 + 0.8 * x - 0.05	Пошук максимуму (MPI_MAX)

КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. Які інженерні та часові переваги мають колективні операції взаємодії в MPI порівняно з ручною організацією циклів через команди MPI_Send та MPI_Recv?

2. Опишіть наслідки для стабільності системи, якщо один із робочих вузлів у комунікаторі не викличе колективну функцію MPI_Bcast, яку одночасно викликали всі інші процеси.

3. Опишіть призначення та детальну структуру аргументів функції MPI_Reduce. На якому обчислювальному вузлі буде фізично доступний фінальний результат згортання?

4. Які базові типи математичних та логічних операцій підтримує системний прапор op у функції MPI_Reduce?

5. Чому в колективних операціях взаємодії відсутній параметр ідентифікатора повідомлення (tag)? За рахунок чого процеси розуміють, які саме пакети даних обробляються у поточний момент?

Основна література:

1. Воронова Л. М., Глибовець М. М. Паралельні обчислення: Навчальний посібник. — К.: Видавничий дім «Києво-Могилянська академія», 2007. — 160 с.

2. Сергієнко А. М. Архітектура обчислювальних систем: Підручник. — К.: ТМЦ «Варіант», 2014. — 232 с.

3. Pacheco P. S. An Introduction to Parallel Programming. — Morgan Kaufmann, 2011. — 370 p.

Додаткова інформація (Вебресурси):

1. Приклади реалізації колективних операцій та редукції в MPI (MPI Collective Communications): https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

2. Довідковий мануал з функцій MPI_Bcast та MPI_Reduce на порталі Microsoft Learn: https://learn.microsoft.com/en-us/message-passing-interface/msmpi-functions

3. Офіційний специфікаційний стандарт екосистеми MPI версії 4.1: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

ЛАБОРАТОРНА РОБОТА №7

Тема: Автоматизація блочного розподілу масивів даних та конкатенація результатів у розподілених системах за допомогою колективних операцій MPI_Scatter та MPI_Gather.

Мета роботи: Ознайомитися з принципами блочного розділення великих обсягів інформації в системах із розподіленою пам'яттю. Набути практичних навичок автоматичного сегментування даних за допомогою функції MPI_Scatter, організації ізольованої паралельної обробки локальних елементів на

обчислювальних вузлах та збирання (склеювання) модифікованих векторів функцією MPI_Gather.

1. ТЕОРЕТИЧНІ ВІДОМОСТІ ТА РОЗШИРЕНЕ ПОЯСНЕННЯ СИНТАКСИСУ

У попередній лабораторній роботі було досліджено операцію MPI_Bcast, яка повністю дублює один і той самий об'єкт на всі вузли системи. Проте для обробки великих аналітичних масивів, матриць або сигналів копіювання всього об'єму даних на кожен ізольований процес є неефективним і суперечить базовій концепції паралелізму. Для досягнення максимальної продуктивності загальний масив потрібно розділити на унікальні, незалежні частини, щоб кожен процесор прораховував свій індивідуальний шматок роботи.

Принцип роботи конвеєра Scatter-Gather:

1. Керуючий вузол (Ранг 0) генерує або зчитує великий глобальний масив даних.
2. Викликається функція **MPI_Scatter**, яка автоматично «нарізає» цей масив на рівні послідовні блоки та розсилає по одному унікальному блоку кожному процесу в його локальний буфер.
3. Усі процеси одночасно (паралельно) у своїх локальних циклах модифікують отримані елементи.
4. Викликається функція **MPI_Gather**, яка виконує зворотну дію: збирає оброблені шматки від усіх процесів, склеює їх суворо за порядком рангів і записує єдиний результуючий масив на Головному вузлі.

Правила розрахунку буферів та синхронізації параметрів

Головна особливість написання коду з використанням MPI_Scatter та MPI_Gather полягає в тому, що параметри кількості елементів вказуються **із** розрахунку на один процес, а не на весь масив.

Якщо загальний розмір глобального масиву становить total_elements, а в системі активовано world_size процесів, то кількість елементів у локальному блоці для кожного процесу (elements_per_process) має дорівнювати суворо:

$$elements\ per\ process = \frac{total\ elements}{world}$$

Пояснення синтаксису функцій зв'язку:

1. **Функція розсіювання (MPI_Scatter):**

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);

- sendbuf – покажчик на глобальний початковий масив (пам'ять під нього виділяється тільки на root). При роботі з контейнерами std::vector передається адреса його першого елемента через метод .data().
- sendcount – кількість елементів, що надсилаються в один потік (дорівнює elements_per_process).
- recvbuf – покажчик на локальний приймальний буфер, який обов'язково має бути створений і розширений на кожному обчислювальному вузлі.

- recvcount – кількість елементів, яку локально приймає кожен потік (має збігатися з sendcount).

2. **Функція збирання (MPI_Gather):**

MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm);

Параметри аналогічні, але логіка міняється місцями: sendbuf тепер є локальним обчисленим блоком процесу, а recvbuf – фінальним глобальним вектором на Master-вузлі, куди стікаються всі дані. Показники count так само відображають обсяг передачі від одного окремого процесу.

2. НАВЧАЛЬНИЙ ШАБЛОН ПРОГРАМИ (lab7.cpp)

Студенту необхідно самостійно вивчити механізми блочного розділення пам'яті, інтегрувати колективні виклики функцій розсіювання та збирання векторних даних типу MPI_DOUBLE, а також реалізувати математичну логіку паралельного прорахунку масиву в межах локального циклу відповідно до умов індивідуального варіанта.

```
C++
#include <iostream>
#include <cstdio>
#include <cmath>
#include <vector>
#include <mpi.h>

int main(int argc, char** argv) {
  // 1. Ініціалізація паралельного середовища MPI
  MPI_Init(&argc, &argv);

    int world_size = 0;
    int world_rank = -1;

    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  // Кількість елементів, яка припадає на ОДИН обчислювальний вузол
```

```
    const int elements_per_process = 2;
    int total_elements = elements_per_process * world_size;
```

// Глобальні буфери (пам'ять виділяється та заповнюється тільки на Вузлі 0)

```
    std::vector<double> global_input_array;
    std::vector<double> global_output_array;

    if (world_rank == 0) {
        global_input_array.resize(total_elements);
        global_output_array.resize(total_elements);

        std::printf("[Node 0]: Global input array initialized with values:\n");
        for (int i = 0; i < total_elements; i++) {
            global_input_array[i] = (i + 1) * 1.5;
            std::printf("%f ", global_input_array[i]);
        }
        std::printf("\n----------------------------------------\n");
        std::fflush(stdout);
    }
```

// Локальні буфери, які створюються абсолютно на КОЖНОМУ вузлі системи

```
    std::vector<double> local_input_block(elements_per_process);
    std::vector<double> local_output_block(elements_per_process);
```

// 2. КОЛЕКТИВНА ОПЕРАЦІЯ РОЗСІЮВАННЯ (MPI_Scatter)

/* ДОПИСАТИ ТУТ: Викликати MPI_Scatter для розрізання global_input_array у local_input_block */

// 3. ПАРАЛЕЛЬНА ОБЧИСЛЕННЯ ФАЗА НА КОЖНОМУ ВУЗЛІ ЗА ВАРІАНТОМ

// Кожен потік паралельно обробляє свій локальний блок даних

```
for (int i = 0; i < elements_per_process; i++) {
    double x = local_input_block[i];
    double y = 0.0;
```

/* ДОПИСАТИ ТУТ: Математична формула обчислення результату за варіантом */

// Приклад (Варіант 1): y = std::sqrt(x) + std::cos(world_rank);

```
local_output_block[i] = y;

    std::printf("[Node %d]: Processed block element %d: input = %f -> local output = %f\n",
                world_rank, i, x, y);
    std::fflush(stdout);
}
```

// 4. КОЛЕКТИВНА ОПЕРАЦІЯ ЗБИРАННЯ РЕЗУЛЬТАТІВ (MPI_Gather)

/* ДОПИСАТИ ТУТ: Викликати MPI_Gather для конкатенації local_output_block у global_output_array */

// 5. ВИВЕДЕННЯ АГРЕГОВАНИХ РЕЗУЛЬТАТІВ НА MASTER-ВУЗЛІ

```
if (world_rank == 0) {
    std::printf("----------------------------------------
---------\n");
    std::printf("[FINAL RESULT Node 0]: Gathered global output array:\n");
    for (int i = 0; i < total_elements; i++) {
        std::printf("%f ", global_output_array[i]);
    }
```

```
std::printf("\n==================================================\n
");
            std::fflush(stdout);
        }


    // 6. Закриття паралельної сесії MPI
        MPI_Finalize();
        return 0;
    }
```

3. ПОРЯДОК ВИКОНАННЯ РОБОТИ

1. Вивчити теоретичні засади організації паралельних конвеєрів обробки масивів за допомогою групових функцій сегментування.

2. Створити у робочому просторі файл сирцевого коду lab7.cpp та перенести туди каркас мінімізованого шаблону програми.

3. Дописати виклик функції MPI_Scatter з обов'язковим використанням методу .data() для коректної передачі покажчиків векторів архітектури STL.

4. У тілі локального циклу обробки даних реалізувати обчислення аналітичної функції y = f(x) відповідно до свого варіанта з таблиці завдання, де параметр R означає унікальний ранг поточного процесу (world_rank).

5. Додати функцію MPI_Gather, яка акумулює та склеює обчислені блоки з усього кластера назад у глобальний результуючий вектор на Вузлі 0.

6. Відкрити термінальне вікно в режимі **bash** та виконати чисту компіляцію:

Bash

```
mpic++ lab7.cpp -o lab7.exe
```

7. Виконати паралельний запуск обчислювального конвеєра на **2 або більше процесах**, використовуючи менеджер процесів від Microsoft:

Bash

"/c/Program Files/Microsoft MPI/Bin/mpiexec.exe" -n 2 ./lab7.exe

8. Зафіксувати логи консолі латиницею, переконатися, що фінальний склеєний масив містить правильні математичні розрахунки, та оформити звіт.

4. ВАРІАНТИ ІНДИВІДУАЛЬНИХ ЗАВДАНЬ

Кожен студент виконує розрахунок локальних елементів масиву за формулою свого варіанта. Початкові елементи масиву x генеруються автоматично на Керуючому вузлі під час старту програми.

Варіант	Математична функція обробки елементів на вузлах (y)
1	$y = \sqrt{x} + cos(R)$
2	$y = x^2 - ln(R + 1)$
3	$y = sin(x) \times (R + 1)$
4	$y = log_{10}(x + 1) + \sqrt{R}$
5	$y = e^{-x} + R^2$
6	$y = \frac{x}{R + 1} + cos^2(R)$
7	$y =$
8	$y = x^3 - \frac{R}{x}$
9	$y = \sqrt{x^2 + R + 1}$
10	$y = (x + R) \times e^{-R}$

КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ЗАХИСТУ РОБОТИ

1. Поясніть принципову архітектурну та функціональну різницю між колективними операціями MPI_Bcast та MPI_Scatter.

2. Як правильно розрахувати параметри sendcount та recvcount у функціях блочного обміну, якщо загальна довжина масиву становить 16 елементів, а обчислення проводяться паралельно на 4 вузлах?

3. Що станеться з працездатністю та розподілом пам'яті системи, якщо загальна кількість елементів початкового масиву не буде кратною кількості запущених процесів?

4. На яких саме обчислювальних вузлах виділяється фізична пам'ять під загальні буфери (global_input_array та global_output_array) у наданому шаблоні? Чому це оптимізує ресурси системи?

5. Яким чином комбінація функцій MPI_Scatter та MPI_Gather дозволяє ефективно обробляти надвеликі масиви даних (Big Data) на розподілених суперкомп'ютерних кластерах?

Основна література:

1. Воронова Л. М., Глибовець М. М. Паралельні обчислення: Навчальний посібник. — К.: Видавничий дім «Києво-Могилянська академія», 2007. — 160 с.

2. Сергієнко А. М. Архітектура обчислювальних систем: Підручник. — К.: ТМЦ «Варіант», 2014. — 232 с.

3. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. 3rd Edition. — MIT Press, 2014. — 480 p.

Додаткова інформація (Вебресурси):

1. Навчальний посібник з колективного розподілу даних та використання функцій Scatter-Gather: https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/

2. Офіційний довідник функцій MPI_Scatter та MPI_Gather на порталі Microsoft Learn: https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi

3. Практичні приклади паралельної обробки векторів (High-Performance Computing Tutorials): https://hpc.llnl.gov/training/tutorials

ДОДАТОК В

МОДУЛЬНИЙ ТЕСТ № 1. Векторний та багатопотоковий паралелізм (OpenMP, SIMD)

1. Який клас обчислювальних систем за класифікацією Флінна описує традиційні однопроцесорні послідовні комп'ютери?

   а) SISD (Single Instruction, Single Data)

   б) SIMD (Single Instruction, Multiple Data)

   в) MISD (Multiple Instruction, Single Data)

   г) MIMD (Multiple Instruction, Multiple Data)

2. Що описує закон Амдала в контексті паралельних обчислень?

   а) Теоретичну межу прискорення програми залежно від частки її послідовного коду

   б) Максимальну кількість ядер, яку можна встановити в систему

в) Швидкість передачі повідомлень між вузлами кластера

г) Об'єм оперативної пам'яті для збереження векторних інструкцій

3. Яка технологія відповідає за векторні обчислення на рівні одного ядра сучасного процесора CPU?

а) SIMD-розширення інструкцій (SSE, AVX)

б) Мережевий інтерфейс MPI

в) Технологія Hyper-Threading

г) Багатопотоковість стандарта POSIX Threads

4. Яка головна особливість архітектури пам'яті в технології OpenMP?

а) Всі обчислювальні потоки мають доступ до спільної оперативної пам'яті

б) Кожен потік має власну ізольовану пам'ять і спілкується через мережу

в) Дані зберігаються виключно в кеш-пам'яті першого рівня L1

г) Пам'ять повинна бути обов'язково розміщена на графічному прискорювачі GPU

5. За допомогою якої директиви в OpenMP створюється паралельна область, де код виконується багатьма потоками?

а) #pragma omp parallel

б) #pragma omp threadpool

в) #pragma omp for

г) #pragma omp start

6. Яке ключове слово (clause) в OpenMP вказує, що для кожного потоку має бути створена власна локальна копія змінної?

а) private

б) shared

в) reduction

г) critical

7. Для чого використовується опція reduction(+:змінна) у директивах OpenMP циклів?

а) Для безпечного паралельного сумування значень без стану гонитви даних

б) Для зменшення кількості ітерацій циклу в кілька разів

в) Для примусової зупинки всіх потоків при виникненні помилки

г) Для автоматичного перетворення типу даних float у double

8. Яка функція OpenMP дозволяє дізнатися загальну кількість потоків, активованих у паралельній області?

а) omp_get_num_threads()

б) omp_get_thread_num()

в) omp_set_num_threads()

г) omp_get_max_threads()

9. Що таке «стан гонитви даних» (Data Race) в багатопотокових програмах?

а) Ситуація, коли кілька потоків одночасно намагаються модифікувати одну комірку пам'яті без синхронізації

б) Процес надто швидкого завершення роботи одного з ядер процесора

в) Помилка компілятора при копіюванні великих масивів чисел

г) Особливий прискорений режим роботи оперативної пам'яті

10. Який пакет компіляції та командної оболонки найчастіше розгортають під ОС Windows для роботи з компілятором GCC UCRT64?

а) MSYS2

б) Visual Studio Installer

в) Android Studio SDK

г) Docker Desktop

11. Якщо при масштабуванні програми з 1 до 4 потоків час виконання задачі зменшився з 1.20 секунд до 0.30 секунд, яке фактичне прискорення (S) було досягнуто?

а) S = 4.00

б) S = 2.00

в) S = 0.25

г) S = 1.20

12. Для чого у середовищі розробника Visual Studio Code створюється конфігураційний файл tasks.json?

а) Для автоматизації команд компіляції (запуску GCC з прапорами -fopenmp чи -lmpi)

б) Для збереження паролів доступу до хмарного сховища

в) Для малювання структурних схем алгоритмів лабораторних робіт

г) Для автоматичного виправлення орфографічних помилок

МОДУЛЬНИЙ ТЕСТ № 2. Розподілені обчислення та інтерфейс передачі повідомлень (MPI)

1. Яка технологія розподілених обчислень використовується, коли завдання виконується на кластері з незалежних комп'ютерів без спільної пам'яті?

а) MPI (Message Passing Interface)

б) Чистий стандарт OpenMP

в) Векторні розширення SSE/AVX

г) Локальні потоки WinAPI

2. З якої функції обов'язково має починатися будь-яка програма, що використовує бібліотеку MPI?

а) MPI_Init

б) MPI_Start

в) MPI_Comm_size

г) MPI_Send

3. Що визначає функція MPI_Comm_rank у розподіленому середовищі обчислень?

а) Унікальний порядковий номер (ID) поточного процесу в комунікаторі

б) Загальну кількість комп'ютерів, об'єднаних у локальний кластер

в) Швидкість мережевого з'єднання між обчислювальними вузлами

г) Пріоритет виконання завдання в диспетчері задач операційної системи

4. Який комунікатор створюється автоматично при старті MPI програми та об'єднує всі запущені процеси?

а) MPI_COMM_WORLD

б) MPI_COMM_LOCAL

в) MPI_COMM_SELF

г) MPI_GROUP_ALL

5. Які базові функції в MPI реалізують парну (точечну) передачу повідомлень типу «один одному» (Point-to-Point)?

а) MPI_Send та MPI_Recv

б) MPI_Bcast та MPI_Reduce

в) MPI_Init та MPI_Finalize

г) MPI_Put та MPI_Get

6. Що відбудеться, якщо викликати блокуючу функцію отримання даних MPI_Recv, а повідомлення від відправника ще не надійшло?

а) Поточний процесовий вузол зупинить виконання і чекатиме на прихід повідомлення

б) Програма аварійно завершиться з кодом системи

в) Функція негайно поверне нульове значення і код продовжить роботу

г) Процес почне виконувати ітерації заново з першої функції

7. Яка колективна функція MPI використовується для розсилки однакових даних від одного головного процесу (Root) до всіх інших процесів кластера?

а) MPI_Bcast

б) MPI_Scatter

в) MPI_Gather

г) MPI_Reduce

8. Чим функція MPI_Scatter відрізняється від функції MPI_Bcast?

а) MPI_Scatter діделить масив на порції і розсилає різні частини вузлам, а MPI_Bcast копіює весь масив усім без змін

б) MPI_Scatter працює лише всередині одного комп'ютера, а MPI_Bcast — через інтернет

в) MPI_Scatter є неблокуючою функцією, а MPI_Bcast обов'язково зупиняє систему

г) MPI_Scatter збирає дані з вузлів, а MPI_Bcast їх розсилає

9. Яка колективна функція MPI збирає локальні результати обчислень з усіх вузлів і математично об'єднує їх (наприклад, сумує змінні) на головному процесі?

а) MPI_Reduce

б) MPI_Gather

в) MPI_Barrier

г) MPI_Sendrecv

10. Яка функція служить для повної синхронізації процесів кластера, створюючи точку очікування, яку жоден процес не може пройти, доки туди не дійдуть усі інші вузли?

а) MPI_Barrier

б) MPI_Wait

в) MPI_Finalize

г) MPI_Comm_free

11. Якою функцією обов'язково повинна закінчуватися будь-яка коректна програма з використанням Microsoft MPI?

а) MPI_Finalize

б) MPI_Close

в) MPI_Exit

г) return 0

12. При розрахунку ефективності використання ядер зафіксовано показник E = 50%. Що це означає в інженерній практиці?

а) Половина ресурсів процесора витрачається марно на накладні витрати або простій потоків

б) Програма працює рівно у два рази швидше, ніж теоретично передбачав закон Амдала

в) Обчислювальний комплекс споживає на 50% менше електроенергії з мережі

г) Точність математичних розрахунків констант впала рівно до 5 знака після коми