Міністерство освіти і науки України
Кам'янець-Подільський національний університет імені Івана Огієнка

М. О. МЯСТКОВСЬКА,
В. С. ЩИРБА

ПІДГОТОВКА ДО ЛАБОРАТОРНИХ РОБІТ З ДИСЦИПЛІНИ «ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ»

НАВЧАЛЬНО-МЕТОДИЧНИЙ ПОСІБНИК

ЕЛЕКТРОННЕ ВИДАННЯ

Кам’янець-Подільський
2026

УДК 37.091
М99

Рекомендувала вчена рада фізико-математичного факультету
Кам’янець-Подільського національного університету
імені Івана Огієнка (протокол № 1 від 16 січня 2026 року)

РЕЦЕНЗЕНТИ:

М. І. Громяк, кандидат фізико-математичних наук, доцент, доцент кафедри математики та методики її навчання Тернопільського національного педагогічного університету імені Володимира Гнатюка;

І. Б. Ковальська, кандидат фізико-математичних наук, доцент, доцент кафедри математики Кам’янець-Подільського національного університету імені Івана Огієнка.

Мястковська М. О., Щирба В. С.

М99 Підготовка до лабораторних робіт з дисципліни «Технології розподілених систем та паралельних обчислень»: навчально-методичний посібник. [Електронний ресурс]. Кам’янець-Подільський: Кам’янець-Подільський національний університет імені Івана Огієнка, 2026. 29 с.

Електронна версія посібника доступна за покликанням:

URL: http://elar.kpnu.edu.ua/xmlui/handle/123456789/9985

Навчально-методичний посібник створений відповідно до робочої програми з курсу «Технології розподілених систем та паралельних обчислень» з метою наповнення навчально-методичного комплексу для цієї дисципліни.

Представлені методичні матеріали охоплюють фундаментальні аспекти розробки багатопотокового програмного забезпечення мовою C#. Структура матеріалу, що охоплює базові питання розробки ПЗ, логічно вибудована: від базового створення потоків через теоретичні засади декомпозиції задач до специфіки паралельних чисельних методів та механізмів синхронізації.

Найбільш сильна теоретична частина – це декомпозиція. Це дає студентам розуміння архітектурного проєктування, а не просто написання коду. Особливо цінним є матеріал про алгоритм впровадження синхронізації, який привчає до системного підходу. Велика увага приділена типовим помилкам та порадам щодо балансування навантаження.

Посібник може бути корисним для студентів технічних спеціальностей.

УДК 37.091

ЗМІСТ

АНОТАЦІЯ

Навчально-методичний посібник присвячено вивченню фундаментальних аспектів багатопотокового та паралельного програмування на платформі .NET. Матеріал структуровано у вигляді чотирьох логічно пов'язаних тем, що охоплюють шлях від створення базових потоків до складних механізмів синхронізації та чисельних методів:

- Створення багатопотокових програм в середовищі C#: робота з класом Thread, керування життєвим циклом потоків та використання пулу потоків (ThreadPool) для оптимізації ресурсів.
- Декомпозиція задачі: системний підхід до розділення задач за даними, функціями, об’єктами та часом (конвеєризація).
- Чисельні методи в паралельній інтерпретації: специфіка модифікації класичних алгоритмів для роботи у багатоядерних системах.
- Технології та механізми синхронізації роботи потоків: методи запобігання стану гонитви (Race Condition) та взаємоблокувань (Deadlock) за допомогою м’ютексів, семафорів, бар’єрів та атомарних операцій.

Видання містить ґрунтовні теоретичні відомості, приклади програмного коду на мові C# та методичні рекомендації щодо аналізу станів потоків. Посібник розрахований на студентів вищих навчальних закладів технічних спеціальностей, що опановують навички створення високоефективного розподіленого програмного забезпечення.

Тема №1

СТВОРЕННЯ БАГАТОПОТОКОВИХ ПРОГРАМ В СЕРЕДОВИЩІ C#

Мета роботи: Вивчення можливостей створення багатопотокових програм мовою C#. Вироблення навиків використання пауз в роботі програми.

ТЕОРЕТИЧНІ ВІДОМОСТІ

1.1. БАГАТОПОТОКОВА ПРОГРАМА ВИВЕДЕННЯ ТЕКСТУ. ПАУЗИ

Для роботи з потоками виконання у середовищі .NET мовою C# використовується клас System.Threading.Thread.

При створенні нового потоку у прикладній програмі необхідно спочатку створити об’єкт класу Thread. Конструктор цього класу приймає один параметр – об’єкт (екземпляр) делегата ThreadStart. Делегат вказує на той метод, що буде виконуватись як новий потік.

Будь-яка багатопотокова програма на C# включає в себе рядок коду:

using System.Threading;

Також потрібно помістити оператор using System, який завантажує системні бібліотеки. Без нього не можливо, наприклад, вивести текст на монітор. Ці два оператори можна переставляти місцями.

Запускає новий потік метод Start() об’єкта класу Thread.

Розглянемо приклад двопотокової програми в консольному режимі, яка забезпечує вивід тексту на екран. Для зручності пояснення особливостей роботи програми рядки лістингу програми пронумеруємо.

Приклад двопотокової програми в консольному режимі.

```
1. using System;
2. using System.Threading;
3. //Ці два попередні оператори можна переставляти місцями
4. class Program
```

```
5.  {
6.  // Лістинг головної програми
7.  static void Main()
8.  {
9.  Console.OutputEncoding = System.Text.Encoding.UTF8;
10. Console.WriteLine("Стартує головна програма");
11. Thread Thread1 = new Thread(Program.ThreadThread1);
12. Thread1.Start();
13. Thread1.Join();
14. Console.WriteLine("Головна програма завершує роботу");
15. Console.ReadLine();
16. }
17. // Лістинг допоміжної (потокової) програми
18. public static void ThreadThread1()
19. {
20. Console.WriteLine("Стартує потокова програма");
21. Thread.Sleep(1000);
22. }
23. }
```

Лістинг головної програми розпочинається з оператора під номером 9:

```
Console.OutputEncoding = System.Text.Encoding.UTF8;
```

Він забезпечує коректність виведення тексту, що містить кирилицю, наприклад, літеру «і».

Оператор в 10 рядку повідомляє, що головна програма розпочинає роботу. Оператор в 11 рядку оголошує допоміжну (потокову) програму, а оператор в 12 рядку запускає її командою Start. Крім оголошення потокової програми (рядок 11) потрібно помістити фрагмент коду команд програми, виконання яких забезпечуватиме потокова програма. Зразком цих процедур можуть служити рядки від 18 по 22. Оголошення потокової програми можна розміщувати як перед, так і після лістингу потокової програми.

Далі головна програма (головний потік) і допоміжна програма (допоміжний потік) працюють паралельно і текст «Стартує потокова про-

грама» та «Головна програма завершує роботу» може виводитися в довільній послідовності. Щоб забезпечити бажану послідовність виведення тексту використовуються паузи (перерви).

Перерви можуть бути з наперед відомою тривалістю (наприклад, як обідня перерва), невідомої наперед тривалості (наприклад, як повітряна тривога) та поки не відбудеться певна подія (наприклад, поки не завантажиться певна програма).

Перерву **з наперед відомою тривалістю** забезпечує команда Sleep(n), де n – тривалість у мілісекундах, а невідомої тривалості – команда Join().

Оператор в 13 рядку блокує виконання головного потоку, поки не завершиться додатковий потік Thread1 (**перерва невідомої наперед тривалості**). Після завершення роботи потокової програми управління передається головній програмі.

Оператор Console.ReadLine() в 15 рядку забезпечує третій вид паузи (**перерва поки не відбудеться певна подія) –** переривання, призупиняючи роботу програми поки не буде натиснена клавіша введення. Вона використовується, щоб надати можливість прочитати текст з екрану.

Результат виконання програми:

Стартує головна програма
Стартує потокова програма
Головна програма завершує роботу

1.2. МЕТОДИ ФОРМУВАННЯ ДЕКІЛЬКОХ ПОТОКІВ

При використанні потоків (одного чи декількох) їх потрібно спочатку оголосити як змінну, надавши деяке ім’я (див. оператор в 11 рядку попередньої програми). Потік, зазвичай, оголошують одним із трьох методів. Наприклад, потік Thread1 можна оголосити оператором типу:

```
Thread Thread1 = new Thread(new ThreadStart(Program.ThreadThread1));
```

або

```
Thread Thread1 = new Thread(Program.ThreadThread1);
```

або

```
Thread Thread1 = new Thread(Program.ThreadFunc1);
```

Ім’я може бути формальне (Thread1, Thread2, Func1, F1 …) або з розшифровкою призначення потоку (Summa, Drib3 …).

Створюються такі потоки, аналогічно, як і в попередній програмі, командами (**формально**) типу:

```
public static void ThreadFunc1()
private static void ThreadFunc2()
```

або (**з розшифровкою призначення потоку**):

```
public static void ThreadSumma()
private static void ThreadDrib()
```

Після цього іде блок команд, виконання яких забезпечує відповідний потік (див. рядки 19 – 22) попередньої програми.

Запускаються на виконання зазначені вище потоки відповідно командами:

```
thread1.Start();
thread2.Start();
Summa.Start();
Drib.Start();
```

Зручно поєднати оголошення і лістинг потоку в один блок. Наприклад,

```
Thread thread1 = new Thread(() =>
{
y = a + b;
Console.WriteLine("Сума рівна "+ y);
});
```

Якщо програма має декілька потоків, час виконання кожного з яких практично співпадає з часом виконання іншого потоку, то, провівши декілька запусків програми, можна помітити, що в різних випадках порядок завершення роботи потоків буде різний. У низці прикладних програм актуально, щоб цей порядок був передбачуваний і стабільний. Наприклад, якщо перший потік обчислює частину алгебраїчного виразу (чисельник) і ділить його на значення іншого виразу (знаменник), яке

обчислює другий потік, то важливо, щоб другий потік завершив роботу раніше ніж перший. Регулювати процес завершення роботи декількох потоків в певній послідовності можна за допомогою пауз (перерв).

1.3. ПУЛ ПОТОКІВ

Пул потоків – це механізм керування потоками, який дозволяє ефективно виконувати велику кількість короткочасних завдань, уникаючи витрат на постійне створення та знищення системних потоків. Замість того, щоб створювати нові потоки для кожного завдання, програма використовує вже існуючу «чергу» потоків, які знаходяться в режимі очікування. При цьому економиться час і ресурси на ініціалізацію потоків. Пул сам автоматично обмежує кількість одночасних потоків, запобігаючи перевантаженню пам'яті та надмірному перемиканню контексту. Система сама вирішує, коли додати новий потік або видалити зайвий. Алгоритм роботи наступний:

1. Завдання ставиться в чергу.
2. Вільний потік із пулу забирає завдання і виконує його.
3. Після завершення потік не «вмирає», а повертається в пул для наступних задач.

У .NET найпростіший спосіб використати пул потоків – клас ThreadPool. Приклад на C#:

```
using System;
using System.Threading;

class Program
{
   static void Main()
   {
      Console.WriteLine("Основний потік розпочав роботу.");

      // Передаємо завдання в пул потоків
      ThreadPool.QueueUserWorkItem(DoWork, "Завдання 1");
      ThreadPool.QueueUserWorkItem(DoWork, "Завдання 2");
```

```
            // Даємо час потокам виконатися (оскільки потік з пулу – фоновий)
            Thread.Sleep(1000);
            Console.WriteLine("Основний потік завершив роботу.");
        }

        static void DoWork(object state)
        {
            string name = (string)state;
            Console.WriteLine($"{name} виконується в потоці: {Thread.CurrentThread.ManagedThreadId}");
        }
    }
```

Порада: використовуйте пул потоків для коротких операцій.

ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

При розв'язуванні першого завдання поекспериментуйте узгодженність роботи головної програми та потокової. У другому завданні використайте різні методи оголошення потоків. В третьому завданні поекспериментуйте з оголошення пауз. В четвертому завданні використайте пул потоків.

1. Створіть програму, головний потік якої виводить на консоль текст «Лабораторна робота № 1»; потокова з паузою в 5 секунд – «Рік навчання (вкажіть рік)» та «Студент(ка) (Ваше ім'я і прізвище)» і після цього головний потік доповнив речення – «перше завдання виконав(ла)».
2. Створіть програму, в якій генерується масив з 20 випадкових цілих чисел від мінус 20 до плюс 20, одна потокова програма визначає суму всіх елементів, друга – добуток додатних, а третя – кількість від'ємних значень елементів масиву. Головна програма виводить значення елементів масиву та результати роботи трьох потоків. Використайте усі три зазначені вище методи оголошення потоків.
3. Створіть програму, в якій головна і дві потокових програми із затримкою 2 секунди виводить на консоль слова речення «Я люблю Україну!».

Запустивши програму на виконання декілька раз, переконайтеся, що порядок виведення слів може бути різним. Змінюючи параметри методу Sleep() підберіть шість наборів значень параметрів, при яких стабільно відображалися б всеможливі перестановки з цих трьох слів.

4. Створити програму обчислення суми від'ємних елементів масиву 30 випадкових цілих чисел від мінус 10 до плюс 10, в якій використовується пул потоків.

Тема №2

ДЕКОМПОЗИЦІЯ ЗАДАЧІ

Мета роботи: Вироблення навиків створювати потокові обчислення на основі методу декомпозиції задачі, навиків застосовувати чисельні методи та алгоритми для паралельних структур.

ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1. ДЕКОМПОЗИЦІЯ ЗАДАЧІ

Ключовим завданням паралельного програмування при розв'язанні конкретної задачі є побудова алгоритму, який дозволяє переробити відомий послідовний алгоритм розв'язання цієї задачі на алгоритм, що розв'язує декілька незалежних підзадач, які допускають їх одночасне виконання (декілька потоків, які можна виконувати паралельно). Такий процес називають декомпозицією задачі.

Декомпозиція – це фундаментальний етап проектування паралельних алгоритмів, який полягає у розділенні загальної задачі на дрібніші частини (підзадачі), що можуть виконуватися одночасно різними процесорами або потоками. Якісна декомпозиція визначає, наскільки ефективно буде використано апаратні ресурси та чи отримаєте ви реальний приріст швидкості.

На практиці, зазвичай, використовують чотири методи декомпозиції задачі:

1) за даними;
2) за функціями (підзадачами);
3) за об'єктами дослідження;
4) за часом.

Вибір стратегії (методу) залежить від природи даних та алгоритму, який ви намагаєтесь паралелізувати.

Декомпозиція за даними – це найбільш розповсюджений підхід. Дані, які потрібно обробити, розбиваються на блоки, і кожен обчислювальний вузол виконує ту саму операцію над своїм блоком. Наприклад, при множенні матриць чи обробці зображень кожен потік обробляє свій рядок або фрагмент пікселів.

При виборі стратегії функціональної декомпозиції на частини розділяється не масив даних, а сам алгоритм. Різні процесори виконують абсолютно різні функції над одними й тими самими або різними даними. Наприклад, у відеогрі один потік відповідає за фізику, інший – за штучний інтелект, третій – за рендеринг графіки.

При об'єктно-орієнтованій декомпозиції задачу розбивають на основі взаємодії автономних об'єктів. Кожен об'єкт інкапсулює дані та методи їх обробки, а паралелізм виникає завдяки одночасній активності цих об'єктів (наприклад, модель акторів).

У темпоральній (часовій) декомпозиції задачу розбивають на етапи, які виконуються послідовно один за одним (конвеєрно).

При проведенні декомпозиції потрібно балансувати між декількома факторами. Зокрема, якщо задача розбита на дуже велику кількість маленьких частин, то це дає гнучкість, але збільшує витрати на комунікацію між потоками. З іншої сторони, якщо велика задача розбита лише на кілька великих блоків, то важче досягти рівномірного завантаження процесорів. Процес розподілу роботи потрібно робити так, щоб усі процесори завершували обчислення приблизно в один час. Якщо один потік працює, а десять чекають на нього – паралелізм неефективний. Також важливо, щоб підзадачі були якомога незалежнішими. Кожна точка синхронізації (де один потік чекає на дані від іншого) уповільнює систему.

Процес декомпозиції можна розбити на чотири етапи:

1. Поділ обчислень та даних на дрібні підзадачі.
2. Визначення того, якими даними мають обмінюватися ці підзадачі.
3. Групування дрібних задач у більші блоки для підвищення ефективності на конкретній архітектурі.
4. Призначення кожної групи задач на конкретний фізичний процесор/ядро.

Декомпозиція – це не просто поділ роботи, а пошук балансу між максимальним паралелізмом та мінімальними накладними витратами на управління цим паралелізмом.

2.2. СПЕЦИФІКА ДЕКОМПОЗИЦІЇ ЗАДАЧІ ЗА ДАНИМИ

Паралелізм за даними базується на застосуванні однакової операції до різних елементів великого набору даних. Якщо задача передбачає однотипну обробку масиву (наприклад, помножити всі елементи масиву на задану константу), то дані розділяються на блоки, які обробляються потоками паралельно.

Хоча в ідеальних задачах потоки працюють автономно, на практиці слід враховувати, що найвищий приріст швидкості досягається, коли блоки даних не перетинаються, а потоки не потребують обміну інформацією під час роботи. Навіть при відсутності проміжної комунікації, точка синхронізації майже завжди існує на етапі збору результатів (наприклад, запис оброблених блоків назад у пам'ять або підсумовування локальних сум у глобальну). Якщо потоки використовують спільний ресурс лише для читання – це не створює проблем. Проте спільний запис вимагає механізмів атомарності або бар'єрної синхронізації.

Розподіл зазвичай виконується головним потоком або автоматично середовищем виконання.

Масив ділиться на неперервні сегменти великого розміру. Елементи розподіляються по одному (1-й потік бере 1-й сегмент, 2-й – 2-й і т.д.).

Типовими прикладами є задачі лінійної алгебри, зокрема, матрично-векторні операції. Матриці можна розбивати за рядами (горизонтальні смуги), стовпцями (вертикальні) або блоками (клітинками). Проміжок інтегрування розбивається на декілька майже рівних проміжків.

Намагайтеся розбивати дані так, щоб кожен потік працював з неперервною ділянкою пам'яті.

2.3. ДЕКОМПОЗИЦІЯ ЗА ФУНКЦІЯМИ

Декомпозиція задачі за функціями – це метод розбиття складної задачі на менші, більш прості підзадачі, які відповідають за конкретні, окремі функції чи дії, що дозволяє спростити розробку, підвищити читабельність коду та полегшити його підтримку, створюючи ієрархію функцій (модулів) для виконання складних операцій. На відміну від декомпозиції за даними, де всі роблять одне й те саме, тут кожен потік виконує свою унікальну роль.

Велика задача ділиться на невеликі частини, кожна з яких виконує одну певну функцію. Кожна підзадача (або модуль) відповідає за свою функцію та володіє необхідними даними для її виконання. Ключова ідея полягає в переході від послідовного виконання «Крок А -> Крок Б -> Крок В» до одночасного виконання цих кроків, якщо вони не мають жорсткої залежності за даними.

Наприклад, нам потрібно написати програму, яка рахує кількість голосних і приголосних у введеному слові. Алгоритм працює наступним чином: головна програма отримує слово від користувача. Одна функція (потік) підраховує голосні, а інша – приголосні літери. Після цього головна програма виводить результат.

Алгоритм проведення декомпозиції полягає в наступному:

1. Аналіз потоку даних: визначте, які дані передаються між етапами. Чи може Крок Б початися до завершення Кроку А?
2. Виокремлення незалежних гілок: знайдіть функції, які не використовують результати роботи одна одної. Це ідеальні кандидати для паралелізму.
3. Оцінка накладних витрат: створення потоку для занадто простої функції (наприклад, просто виведення тексту) може зайняти більше часу, ніж саме виконання цієї функції.
4. Визначення точок синхронізації: чітко окресліть момент, де всі паралельні функції мають «зустрітися», щоб головна програма могла підбити підсумок.

Декомпозиція за функціями зазвичай складніший процес ніж декомпозиція за даними. Тут досить часто допускають типові помилки. Якщо

Функція 1 постійно чекає на проміжні дані від Функції 2, паралелізм зникає, і програма працює фактично послідовно, але з додатковими витратами на перемикання контексту. Розбиття на занадто дрібні функції призводить до "шторму повідомлень" або великої кількості блокувань, що паралізує роботу процесора. Коли дві функції одночасно намагаються записати дані в одну змінну (наприклад, у спільний лог-файл), виникає стан гонитви (Race Condition).

Важливе правило: при декомпозиції за функціями прагніть до того, щоб кожна функція була чистою – отримувала вхідні дані, повертала результат і не змінювала глобальний стан програми без нагальної потреби.

2.4. ОБ'ЄКТНО-ОРІЄНТОВАНА ДЕКОМПОЗИЦІЯ

Цей підхід базується на розбитті системи на незалежні сутності (об'єкти), кожна з яких має власний стан і методи обробки. В паралельному програмуванні кожному такому об'єкту може відповідати окремий потік або процес. Вся система сприймається як набір активних об'єктів, що взаємодіють між собою через обмін повідомленнями. Наприклад, симуляція екосистеми. Кожна тварина – це об'єкт. Кожна тварина «обчислює» свій наступний крок (пошук їжі, рух) незалежно від інших.

Для об'єктно-орієнтованої декомпозиції характерна висока модульність, легкість масштабування (можна просто додавати нові об'єкти).

Для ефективної реалізації цього підходу рекомендується використовувати модель акторів. У цій моделі кожен об'єкт («актор») є повністю ізольованим: він не має прямого доступу до пам'яті інших об'єктів. Взаємодія відбувається виключно через асинхронні повідомлення. Це знімає проблему «стану гонитви», оскільки об'єкт сам керує своїми внутрішніми даними.

Кожен паралельний об'єкт повинен мати монопольний доступ до своїх даних. Якщо два об'єкти (наприклад, дві «тварини» в симуляції) потребують доступу до одного ресурсу (наприклад, одного куща їжі), цей ресурс сам має стати окремим об'єктом, який приймає запити від інших, або ж доступ має регулюватися через посередника.

Важливо знайти баланс між кількістю об'єктів та накладними витратами. Якщо кожен елементарний крок (наприклад, рух однієї кліти-

ни) створює окремий потік, система витратить більше часу на перемикання контексту між потоками, ніж на самі обчислення. Якщо ж один об'єкт керує великою групою сутностей, то виникає ризик послідовного виконання там, де міг би бути паралелізм.

При обміні повідомленнями між об'єктами слід уникати блокувань. Об'єкт-відправник не повинен чекати, поки об'єкт-отримувач опрацює повідомлення, якщо це не критично для логіки. Це дозволяє системі працювати максимально плавно.

Однією з переваг цього підходу є можливість легко адаптувати систему під навантаження. Наприклад, у хмарних сервісах або складних іграх можна динамічно створювати нові активні об'єкти на різних ядрах процесора або навіть на різних серверах, що забезпечує відмінну горизонтальну масштабованість.

Типовими сферами застосування є моделювання натовпу, транспортних потоків, фізичних частинок, мікросервісна архітектура, де кожен сервіс – це автономний об'єкт, ігрова індустрія.

Через асинхронну природу взаємодії важко відстежити послідовність подій, що призвела до помилки («ефект метелика»). Якщо об'єкти занадто часто обмінюються повідомленнями, пропускна здатність шини даних може стати вузьким місцем.

Порада: вибирайте об'єктно-орієнтовану декомпозицію тоді, коли поведінка сутностей у вашій задачі є складнішою за самі дані, якими вони оперують.

2.5. ТЕМПОРАЛЬНА ДЕКОМПОЗИЦІЯ

Декомпозиція за часом (часова декомпозиція) – це стратегія розділення великої задачі на етапи або компоненти відповідно до часової послідовності їх виконання. Тут задача розбивається на етапи, які виконуються один за одним. Паралелізм досягається завдяки тому, що різні етапи обробки для різних порцій даних виконуються одночасно.

На відміну від функціональної декомпозиції (що робити), часова фокусується на тому, коли і в якому порядку мають відбуватися дії.

В контексті паралельного програмування темпоральна декомпозиція (або декомпозиція за часом) – це стратегія розпаралелювання, за якої різні процесори виконують різні етапи одного обчислювального процесу над потоком даних.

Найкращою аналогією для темпоральної декомпозиції є конвеєр. Тут темпоральна декомпозиція не зменшує час обробки одного елемента, але значно збільшує кількість елементів, що обробляються за одиницю часу. Тут важливо звернути увагу на фокус одночасного завершення етапів роботи. Якщо одна фаза конвеєра виконується 10 мс, а інша 100 мс, то весь конвеєр буде працювати зі швидкістю найповільнішої ланки. У контексті паралельного програмування, говорячи про темпоральну декомпозицію з фокусом на одночасність завершення, ми переходимо до концепції бар'єрної синхронізації та балансування навантаження. Потрібно балансувати навантаження потоків так, щоб вони працювали синхронно.

Головне при використанні темпоральної декомпозиції спроектувати **з**балансований конвеєр. Якщо, наприклад, Етап А триває 10 мс, а Етап Б – 50 мс, то Етап А, передавши готову порцію даних другому етапу, попрацює ще 10 мс і буде простоювати 40 мс часу поки Етап Б не буде готовим прийняти нову порцію даних. Розбийте найповільніший етап на два менших або запустіть кілька паралельних копій цього етапу (наприклад, два потоки для Етапу Б), щоб вирівняти загальний темп.

Темпоральна декомпозиція вигідна для потоку даних, але вона збільшує час обробки однієї конкретної одиниці даних (через витрати на передачу між етапами). Якщо ваша задача критична до часу відгуку, не робіть конвеєр занадто довгим. Кожен новий етап додає затримку на перемикання контексту та копіювання даних.

Який з цих запропонованих методів варто застосувати в значній мірі залежить від специфіки задачі. В складних моделях, як правило, використовують декілька підходів одночасно дотримуючись алгоритму розпаралелення.

ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

Головна мета цих завдань – навчитися проводити декомпозицію задачі одним із зазначених методів.

1. Створіть програму, в якій генерується масив з 20 випадкових цілих чисел від мінус 10 до плюс 10, а дві потокові програм, йдучи назустріч одна одній (одна з початку масиву, а інша з кінця) визначають максимальний елемент. Головна програма виводить значення елементів масиву та максимального елементу. Узгодити момент завершення роботи потоків.
2. Створіть програму, яка заповнює випадковими цілими числами дві матриці розмірності 4×4, а чотири потоки знаходять їх добуток. Завантаженість потоків має бути однаковою. Вивести значення усіх трьох матриць на екран.
3. На прямокутній ділянці поверхні екологи рівномірно встановили датчики, які щосекунди фіксують приріст забруднення, що відбувається хаотично. Час, за яких інформація з датчика потрапляє на обробку, становить десять мілісекунд. Визначити значення, місце та час, коли стався найбільший приріст; час, коли на поверхні відбувся найбільший приріст; місце, де за період спостереження відбулося найбільше забруднення. В програмі передбачити декілька потоків. Визначте час роботи програми в безпотоковому та потоковому варіанті обчислень.
4. Визначити час, коли кожен з потоків переходить в стан закінчення роботи. Підберіть параметри дослідження так, щоб період від стану предстартовий головної програми до стану завершення роботи становив не менше 10 секунд. Визначте стан головної програми і потоків через 10 секунд після старту. Оптимізуйте програму з метою рівномірного завантаження потоків.

Тема №3
ЧИСЕЛЬНІ МЕТОДИ В ПАРАЛЕЛЬНІЙ ІНТЕРПРЕТАЦІЇ

Мета роботи: Вироблення навиків створювати потокові обчислення на основі різних методів декомпозиції задачі. Розвиток компетентностей і навиків застосовувати чисельні методи та алгоритми для паралельних структур, застосовувати пул потоків. Закріплення навиків розробки програмного забезпечення з попередніх занять.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Чисельні розрахунки є найбільш поширеним типом задач, що трапляються в інженерних і наукових дослідженнях. Класичними серед них є обчислення елементарних функцій (схема Горнера, обчислення значень елементарних функцій за допомогою степеневих рядів, обчислення значень елементарних функцій за допомогою ланцюгових дробів), розв'язання нелінійних рівнянь (метод поділу відрізка пополам, метод січних, метод дотичних, комбінований метод, метод ітерацій), розв'язування систем лінійних алгебраїчних рівнянь, розв'язання систем нелінійних рівнянь, наближення функцій, чисельне диференціювання та інтегрування функцій, чисельне розв'язання диференціальних рівнянь.

Модифікація чисельних методів для паралельної інтерпретації – це процес перетворення класичних послідовних алгоритмів у таку форму, яка дозволяє одночасно задіяти декілька обчислювальних вузлів. Це не просто запуск коду на різних ядрах, а часто повний перегляд математичної структури методу.

Практично усі типові задачі розв'язуються ітераційними алгоритмами, що змушує пошук декомпозиції задачі зосереджувати всередині ітераційного циклу. Все залежить від змісту конкретної задачі. Тут може бути, наприклад, розбиття процесу обчислення значення деякого складного виразу на кілька незалежних частин.

Одним із ключових аспектів модифікації класичних послідовних алгоритмів чисельних методів зміна природи ітераційних процесів. У пос-

лідовних методах кожна наступна ітерація зазвичай використовує результати поточної ітерації. Це створює жорстку часову залежність.

Перехід від методів «послідовного уточнення» (як метод Зейделя) до методів «одночасного уточнення» (як метод Якобі). Хоча метод Якобі може збігатися повільніше з точки зору кількості ітерацій, він ідеально паралелиться, оскільки всі нові значення обчислюються незалежно на основі даних попереднього кроку.

В інших випадках застосовують метод декомпозиції області. Для чисельного розв'язання диференціальних рівнянь (наприклад, метод скінченних різниць або скінченних елементів) фізична область розбивається на піддомени. Головна модифікація полягає в обробці «віртуальних меж». Процесори мають обмінюватися даними про значення на межах своїх областей після кожного кроку обчислень.

Багато чисельних методів вимагають знаходження сум, добутків або екстремумів величезних масивів даних (наприклад, скалярний добуток векторів). Замість лінійного перебору використовується деревоподібна редукція. Дані розбиваються на пари, кожна пара обробляється окремим потоком, результати знову групуються, доки не залишиться одне фінальне значення.

У паралельній інтерпретації ефективність чисельного методу вимірюється не загальною кількістю операцій, а рівномірністю навантаження.

Якщо чисельний метод адаптивний (наприклад, сітка стає густішою там, де функція різко змінюється), один процесор може отримати значно більше роботи, ніж інші. Застосовують динамічний перерозподіл задач під час розрахунку, щоб уникнути простою вільних ядер.

При зміні порядку операцій (наприклад, сумування елементів у різному порядку при паралельній редукції) результати можуть відрізнятися через особливості представлення чисел з рухомою комою. Модифікований паралельний метод має проходити повторну перевірку на чисельну стійкість, оскільки зміна послідовності дій може призвести до швидшого накопичення похибок округлення.

Модифікація чисельних методів – це завжди компроміс. Ми часто свідомо вибираємо математично «гірший» метод (той, що потребує більше ітерацій), якщо він дозволяє задіяти сотні ядер одночасно, оскільки в підсумку це дає значно більшу швидкість отримання результату.

ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

1. Побудувати трипотокову програму розв’язання системи п’яти рівнянь з п’ятьма невідомими методом Жордана-Гауса.
2. Побудувати 10-потокову програму обчислення означеного інтегралу деякої функції на деякому проміжку.
3. Побудуйте програму обчислення методом лівих прямокутників означеного інтегралу від функції $y = x^2$ на відрізку [0, 2] з кроком 0,0000001. В програмі використайте декілька потоків. Обчисливши цей же інтеграл, зменшивши крок інтегрування вдвічі, знайдіть вірні цифри означеного інтегралу.
4. Організувати методом декомпозиції за часом двопотокову програму розв’язування системи лінійних алгебраїчних рівнянь методом ітерацій, коефіцієнти матриці якої задаються випадковим чином так, що алгоритм збіжний. Значення невідомих оновлюються в кінці ітерації. Проаналізувати час роботи програми в безпотоковому та двопотоковому режимі.

Тема №4

ТЕХНОЛОГІЇ ТА МЕХАНІЗМИ СИНХРОНІЗАЦІЇ РОБОТИ ПОТОКІВ

Мета роботи: Наочно побачити випадки гонитви, блокування, взаємоблокування роботи потоків та навчитися їх усувати. Вироблення навиків вирішення проблем, пов’язаних із синхронізацією роботи потоків.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Синхронізація потоків – це набір методів і технологій, які забезпечують правильну послідовність виконання потоків та безпечний доступ до спільних ресурсів у багатопотоковому середовищі. Без синхронізації виникає стан «гонитви», коли результат обчислень залежить від випадкового порядку виконання потоків, що призводить до помилок.

Механізми синхронізації поділяються на дві основні категорії взаємовиключення і сигналізація. Взаємовиключення запобігає одночасному доступу кількох потоків до критичної секції (спільних даних). Сигналізація передбачає сповіщення одного потоку іншим про те, що відбулася певна подія або дані готові до обробки.

Основні технології синхронізації:

1. М’ютекс. Найпростіший об'єкт синхронізації. Працює як замок: тільки один потік може «володіти» м’ютексом у певний момент часу. Інші потоки чекають, поки власник його звільнить.
2. Семафор. Лічильник, який дозволяє доступ певній кількості потоків одночасно. Якщо лічильник > 0, потік проходить; якщо 0 – чекає.
3. Критична секція. Об'єкт, аналогічний м’ютексу, але оптимізований для використання в межах одного процесу (швидший за м’ютекс).
4. Умовні змінні. Використовуються разом із м’ютексом для того, щоб потік міг заснути до моменту виконання певної умови (наприклад, появи даних у черзі).
5. Бар'єри. Точка в програмі, де всі потоки повинні зупинитися і чекати, поки останній потік не досягне цієї точки, перш ніж усі зможуть продовжити рух.

6. Атомарні операції. Низькорівневі операції, які виконуються як одна неподільна дія на рівні процесора (наприклад, інкремент змінної). Не потребують блокувань.

Процес впровадження синхронізації має бути системним, щоб уникнути взаємоблокувань та зниження продуктивності.

Етап 1. Ідентифікація спільних ресурсів

Визначте всі змінні, структури даних, файли або пристрої, до яких будуть звертатися одночасно кілька потоків. Якщо потік тільки читає дані, які не змінюються, синхронізація зазвичай не потрібна.

Етап 2. Визначення критичних секцій

Окресліть мінімальний обсяг коду, де відбувається робота зі спільним ресурсом. Критична секція має бути якомога коротшою. Не виконуйте всередині неї тривалі операції (ввід-вивід, мережеві запити), щоб не блокувати інші потоки.

Етап 3. Вибір механізму

- Якщо потрібен доступ до змінної – використовуйте атомарні операції.
- Якщо потрібно захистити складний об'єкт – використовуйте м’ютекс.
- Якщо є обмежений ресурс (наприклад, пул з 5 з'єднань до бази даних) – використовуйте семафор.
- Якщо потоки мають працювати етапами (наприклад, у темпоральній декомпозиції) – використовуйте бар'єр.

Етап 4. Реалізація доступу

Типова схема (псевдокод):

```
lock(mutex); // Вхід у критичну секцію (очікування, якщо зайнято)
/* робота з даними */
unlock(mutex); // Вихід та сповіщення інших потоків
```

Для ефективного використання синхронізації уникайте його надмірного застосування. Кожне блокування – це накладні витрати часу. Якщо можливо, використовуйте локальні копії даних для кожного потоку, а в кінці об’єднуйте результати.

Для профілактики взаємоблокувань завжди захоплюйте м’ютекси в одному і тому самому порядку в усіх потоках. Використовуйте тайм-аути при очікуванні блокування. Якщо дані часто читаються і рідко зміню-

ються, використовуйте спеціальні замки, які дозволяють багатьом потокам працювати одночасно, але блокують усіх лише на час запису.

Синхронізація – це «світлофор» для потоків. Її методика базується на правилі: «Заблокуй якомога пізніше, розблокуй якомога раніше». Правильно обраний механізм дозволяє зберегти баланс між цілісністю даних і швидкістю паралельного виконання.

Якщо кілька потоків хочуть користуватись критичним ресурсом в режимі розподілу часу, їм необхідно синхронізувати свої дії таким чином, щоб ресурс завжди знаходився в розпорядженні не більше ніж в одного потоку. Якщо один потік користується в цей момент часу критичним ресурсом, то всі інші потоки мають в цей час очікувати його звільнення. Можливі також і такі критичні ресурси, які можуть знаходитися в одночасному розпорядженні кількох потоків, але кількість цих потоків також наперед обмежена.

Питання про стан роботи потоків є критичним для діагностики та налагодження паралельних програм. Стан потоку визначає, що саме він робить у конкретний момент часу з точки зору операційної системи та планувальника задач. Розуміння цих станів дозволяє розробнику зрозуміти, чому програма «зависла», чому вона працює повільно або де виникає конфлікт ресурсів. Більшість сучасних систем використовують стандартну модель станів:

- **Створення.** Потік ініціалізований, для нього виділено стек, але він ще не почав виконувати код.
- **Готовність.** Потік готовий до роботи, але процесор зараз зайнятий іншим потоком. Він стоїть у черзі планувальника.
- **Виконання.** Потік безпосередньо виконує інструкції на ядрі процесора.
- **Очікування / Блокування.** Потік не може продовжувати роботу. Це найцікавіший стан для аналізу. Він може чекати на:
  - Звільнення м'ютекса іншим потоком.
  - Завершення операції вводу-виводу (читання з диска/мережі).
  - Сигнал від умовної змінної.
- **Завершення.** Потік закінчив виконання функції або був примусово зупинений.

Коли ми розглядаємо стан потоків, ми зазвичай шукаємо аномалії в їхній поведінці. Розгляд стану потоків – це перехід від написання коду до

спостереження за його реальною поведінкою в пам'яті та на процесорі. Головна мета розробника: максимізувати час перебування корисних потоків у стані виконання та мінімізувати час їхнього очікування на блокуваннях.

ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ

1. Деяке підприємство кожні 0,1 секунди виготовляє від 1 до 9 одиниць деякої продукції і передає у складське приміщення. Якщо на складі виявиться більше 50 одиниць цієї продукції, то менеджер повідомляє про можливість її одержання представникам двох підприємств споживачів, перший з яких може відразу забрати 45, а інший 48 одиниць цієї продукції. Видавати продукцію одночасно двом підприємствам немає можливості. Видача не допускається, якщо на складі залишилось менше 50 одиниць продукції, або якщо підприємство-споживач, яке першим отримало продукцію, не повернуло тару. На розвантаження продукції і повернення тари першому підприємству потрібно 1,1, а другому – 1,2 секунди. Змоделювати і проаналізувати роботу підприємства з виготовлення та видачі продукції впродовж однієї хвилини.
2. Для реалізації 50 одиниць замовленої продукції підприємство може задіяти дві бригади пакувальників. Одна бригада, залежно від випадкових обставин, може підготувати від 10 до 60 одиниць товару, а інша від 20 до 55 одиниць. Змоделювати роботу бригад і, провівши 10 раз експерименти, з’ясувати у скількох випадках підприємство не зможе вчасно підготувати замовлену продукцію.
3. Змоделювати роботу продавця магазину з обслуговування чотирьох покупців одного і того ж виду товару, якщо кількість одиниць такого товару обмежена, об’єм покупки для кожного з покупців є випадковим значенням і можливі випадки, коли останнім у черзі покупцям не вистачить товару.
4. Створити об'єкт «Рахунок» із початковим балансом 1000 грн. Запустити 10 потоків, кожен з яких 100 разів знімає по 10 грн, і 10 потоків, які 100 разів кладуть по 10 грн. Спочатку запустити програму без синхронізації (отримати неправильний фінальний баланс; пояснити чому результат не правильний. Підказка. Операція Balance += 10 насправді складається з трьох кроків на рівні процесора: Зчитати → Додати → Записати). Виправити цю помилку за допомогою оператора lock або Monitor.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Архелюк О. Д. Конспект лекцій з навчальної дисципліни «Розподілені сервісні системи» для студентів всіх форм навчання спеціальності «Телекомунікації та радіотехніка» відділу Інфокомунікацій та інженерії. Чернівці: ЧНУ імені Юрія Федьковича. 2021. 94 с.
2. Семеренко В. П. Технології паралельних обчислень: навчальний посібник Вінниця: ВНТУ, 2018. 104 с.
3. Литвинов О. А., Хандецький В. С. Розподілена обробка інформації: [моногр.]. Дніпропетровськ: ТОВ «Баланс-Клуб», 2013. 314 с.
4. Мамай Л. М., Самусь Є. І. Методичні вказівки і завдання до лабораторних робіт з курсу «Паралельні та розподілені обчислення», для студентів 4-го курсу інженерно-технічного факультету спеціальності 123 Комп'ютерна інженерія. Ужгород, 2021. 53 с.
5. Минайленко Р. М. Паралельні та розподілені обчислення: навч. посіб. Кропивницький: Видавець Лисенко В. Ф., 2021. 153 с.
6. Яровий А. А. Методи та засоби організації високопродуктивних паралельно-ієрархічних обчислювальних систем із рекурсивною архітектурою: монографія. Вінниця: ВНТУ, 2016. 363 с.
7. Пул потоків у .NET (The Managed Thread Pool). Офіційна документація Microsoft. URL: https://learn.microsoft.com/dotnet/standard/threading/the-managed-thread-pool
8. Синхронізація примітивів (Overview of synchronization primitives). Microsoft Documentation. URL: https://learn.microsoft.com/dotnet/standard/threading/-overview-of-synchronization-primitives

Міністерство освіти і науки України
Кам’янець-Подільський національний університет імені Івана Огієнка

НАВЧАЛЬНЕ ЕЛЕКТРОННЕ ВИДАННЯ

МЯСТКОВСЬКА Марина Олександрівна,
кандидат педагогічних наук, старший викладач
кафедри комп’ютерних наук Кам'янець-Подільського
національного університету імені Івана Огієнка
ЩИРБА Віктор Самуїлович,
кандидат фізико-математичних наук, доцент, професор кафедри
комп’ютерних наук Кам'янець-Подільського національного
університету імені Івана Огієнка

ПІДГОТОВКА ДО ЛАБОРАТОРНИХ РОБІТ З ДИСЦИПЛІНИ «ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ»

НАВЧАЛЬНО-МЕТОДИЧНИЙ ПОСІБНИК

ЕЛЕКТРОННЕ ВИДАННЯ

Підписано 17.03.2026. Формат 60x84/16. Гарнітура «Cambria».
Об’єм даних 0,75 Мб. Обл.-вид. арк. 1,2. Зам. № 1231.

Кам’янець-Подільський національний університет
імені Івана Огієнка,
вул. Огієнка, 61, м. Кам’янець-Подільський, 32300.
Свідоцтво серії ДК № 3382 від 05.02.2009 р.

Виготовлено в Кам’янець-Подільському національному
університеті імені Івана Огієнка,
вул. Огієнка, 61, м. Кам’янець-Подільський, 32300.