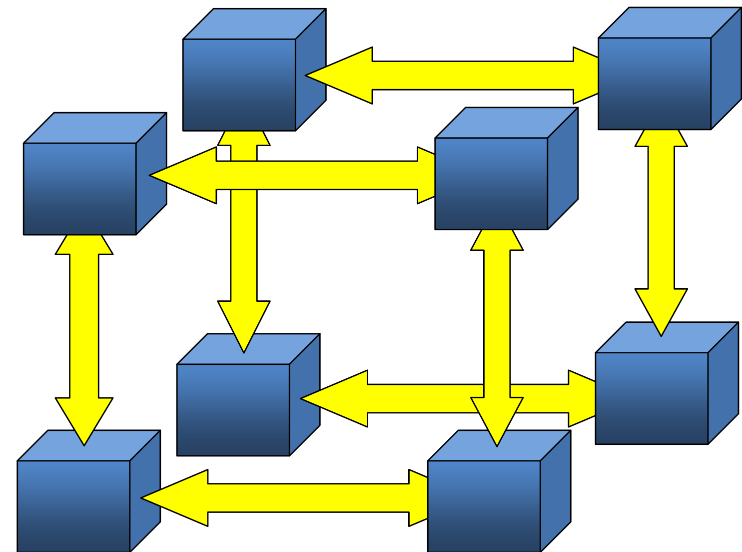
**В. П. Семеренко**

# ТЕХНОЛОГІЇ

# ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ



Міністерство освіти і науки України
Вінницький національний технічний університет

**В. П. Семеренко**

# ТЕХНОЛОГІЇ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

**Навчальний посібник**

Вінниця
ВНТУ
2018

УДК 681.3.06.(075)
С34

Рекомендовано до друку Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 3 від 26.10.2017 р.)

Рецензенти:
**В. А. Лужецький**, доктор технічних наук, професор
**А. М. Пєтух,** доктор технічних наук, професор
**Л. Б. Ліщинська,** доктор технічних наук, професор

**Семеренко, В. П.**
С34 Технології паралельних обчислень : навчальний посібник / Семеренко В. П. – Вінниця : ВНТУ, 2018. – 104 с.

В посібнику розглянуто основні принципи паралельної обробки даних: паралельні комп’ютерні архітектури, види паралелізму, основні етапи розробки паралельних алгоритмів. Дано порівняльну характеристику багатозадачності та багатопотоковості. Розглянуто багатопотокове програмування мовою C# з використанням класів Thread, Task, Parallel. Велику увагу приділено методикам паралельного програмування мовою C++ з використанням технологій OpenMP, MPI, CUDA. Посібник призначений для здобувачів освітнього ступеня бакалавра спеціальності 123 «Комп’ютерна інженерія» для вивчення дисципліни «Паралельні та розподілені обчислення».

**УДК 681.3.06.(075)**

© ВНТУ, 2018

# ЗМІСТ

Вступ ........ 5
1 Основні принципи паралельних обчислень ........ 6
  1.1 Паралельні комп’ютерні архітектури ........ 6
  1.2 Види паралелізму ........ 8
  1.3 Етапи розробки паралельних алгоритмів ........ 12
  1.4 Процеси і потоки ........ 14
  1.5 Багатозадачність, багатопотоковість та паралелізм ........ 16
2 Багатопотокове програмування мовою C# за допомогою класу Thread ........ 20
  2.1 Створення нових потоків виконання ........ 20
  2.2 Пріоритети потоків ........ 21
  2.3 Стани потоків ........ 23
  2.4 Синхронізація потоків ........ 26
  2.5 Асинхронні делегати ........ 31
  2.6 Багатопотокове програмування на основі пулу потоків ........ 32
3 Багатопотокове програмування мовою C# за допомогою бібліотеки TPL ........ 36
  3.1 Нові засоби розпаралелювання в мові C# ........ 36
  3.2 Багатопотокове програмування з використанням класу Task ........ 37
  3.3 Багатопотокове програмування мовою C# з використанням класу Parallel ........ 44
4 Багатопотокове програмування мовою C++ за допомогою стандарту OpenMP ........ 51
  4.1 Принципи паралельної обробки з використанням OpenMP ........ 51
  4.2 Основні синтаксичні конструкції OpenMP ........ 52
  4.3 Директиви паралельної обробки OpenMP ........ 53
  4.4 Бібліотечні функції OpenMP для паралельних обчислень ........ 58
  4.5 Планування паралельної обробки ........ 60
  4.6 Директиви і функції синхронізації обчислень ........ 61
5 Паралельне програмування з використанням стандарту MPI ........ 64
  5.1 Принципи паралельної обробки на основі стандарту MPI ........ 64
  5.2 Організація паралельних обчислень в стандарті MPI-2 ........ 65
  5.3 Бібліотека MPI ........ 67
  5.4 Найпростіша MPI-програма ........ 68
  5.5 Функції для реалізації парних комунікаційних операцій ........ 71
  5.6 Режими передачі даних в MPI ........ 72
  5.7 Функції для реалізації колективних комунікаційних операцій ........ 74
  5.8 Функції для реалізації колективних обчислювальних операцій ........ 78
  5.9 Функції для роботи з групами та комунікаторами ........ 81
6 Паралельне програмування мовою C++ з використанням стандарту CUDA ........ 84

6.1 Еволюція процесорних пристроїв ........ 84
6.2 Архітектури GPGPU та CUDA ........ 85
6.3 Апаратне забезпечення CUDA ........ 88
6.4 Програмне забезпечення CUDA ........ 89
6.5 Основні нововведення в мову C ........ 90
6.6 Основи створення програм в CUDA ........ 96
6.7 Підготовка до виконання програм ........ 98
Післямова ........ 100
Глосарій ........ 101
Список рекомендованої літератури ........ 103

## Вступ

Можливості подальшого нарощування продуктивності комп'ютерів в рамках послідовних принципів обробки даних практично вичерпали себе, що обумовлено в основному скінченою швидкістю розповсюдження сигналів. Пошук вирішення проблеми підвищення продуктивності йде в напрямку розвитку принципів паралельної обробки інформації.

Отримати суттєвий приріст в продуктивності можна лише у використанні принципово нових комп'ютерних архітектур, які основані на паралельній обробці даних.

Основні принципи паралелізму були впроваджені ще в перші експериментальні паралельні машини, які з'явились в 60-70 роках минулого століття. У векторній CRAY-1, матричній ILLIAC-4, ортогональній OMEN та в інших комп'ютерах тих часів були започатковані основні напрямки паралельних обчислень: конвеєризація, процесорні матриці, асоціативна адресація [1].

В наші дні принципи паралелізму використовуються в більшості обчислювальних пристроїв, найбільш поширеними паралельними комп'ютерами є кластерні системи та багатоядерні персональні комп'ютери [2].

Прогрес в обчислювальній техніці викликав інтенсивний розвиток відповідного програмного забезпечення. Утворився окремий розділ в програмуванні – паралельне програмування. До основних питань, якими займається паралельне програмування, відносяться розробка параллельних алгоритмів та їх реалізація мовами паралельного програмування для конкретних паралельних архітектур.

Основна проблема сучасного паралельного програмування полягає у складності побудови схеми паралельних обчислень. Велику допомогу програмісту можуть надати технології паралельних обчислень, які вже стали визнаними стандартами: OpenMP, MPI, CUDA.

За останні роки було запропоновано різноманітні бібліотеки, компілятори, системні та сервісні програми, які допомагають програмісту у написанні та налагоджуванні паралельних програм. Вміння ефективно застосовувати ці програмні інструменти є важливим показником професіоналізму сучасного програміста.

# 1 Основні принципи паралельних обчислень

## 1.1 Паралельні комп'ютерні архітектури

Архітектурою комп'ютера називають сукупність його властивостей та характеристик, які розглядаються з позицій користувача. Архітектура визначає принципи дії і взаємне поєднання основних пристроїв комп'ютера, систему команд і систему адресації, швидкодію процесора та обсяг пам'яті, програмне забезпечення і засоби інтерфейсу користувача.

Класична архітектура комп'ютера склалась в кінці 40-х – на початку 50-х років минулого століття. Суттєвий вплив на її становлення спричинили ідеї Джона фон Неймана [3].

Від самого початку комп'ютерної ери існувала необхідність в підвищенні продуктивності роботи обчислювальних машин. В основному це досягалось в результаті еволюції технології виробництва їх елементної бази, що давало можливість збільшувати швидкодію комп'ютера згідно із законом Мура: продуктивність процесорів має збільшуватись вдвічі кожні півтора–два роки. Більше сорока років обчислювальна потужність комп'ютерів дійсно зростала згідно з цим емпіричним законом. Але поступово зростання тактової частоти процесорів зупинилося і, відповідно, припинилось збільшення продуктивності роботи в рамках традиційної, фон-нейманівської, архітектури.

Подальший прогрес обчислювальної техніки став можливим тільки на основі паралельних обчислень, що змушує здійснити перехід на принципово нові комп'ютерні архітектури.

Звичайно, вже давно ведуться дослідження в сфері паралельної обробки, тому, коли у інженерів виникли проблеми зі зростанням продуктивності обчислень, у теоретиків вже був готовий великий вибір нових комп'ютерних архітектур.

Відомо багато різних класифікацій комп'ютерних архітектур [4], найпопулярнішою з них є класифікація Флінна, яка була запропонована ще у 1966 році. Ця класифікація використовує три основних компоненти: процесори, модулі пам'яті та мережу комутаторів, яка з'єднує між собою процесори та пам'ять. В основу роботи таких систем покладено поняття потоків команд та потоків даних, які обробляються процесорами. Залежно від кількості та співвідношення цих потоків можна виділити 4 архітектурних класи:

*SISD* (*Single Instruction & Single Data* – один потік команд & один потік даних),

*SIMD* (*Single Instruction & Multiple Data* – один потік команд & багато потоків даних),

*MISD* (*Multiple Instruction & Single Data* – багато потоків команд & один потік даних),

*MIMD* (*Multiple Instruction & Multiple Data* – багато потоків команд & багато потоків даних).

До класу *SISD* можна віднести послідовні комп'ютери фон-нейманівської архітектури.

До класу *SIMD* відносять комп'ютери, в яких в кожний момент часу може виконуватись одна й та ж команда для обробки кількох елементів даних. Такий принцип обробки даних використовується у векторних процесорах, в яких завдяки конвеєру за один такт обробляються декілька елементів одного вектора. Як результат, багатоелементний вектор даних обробляється за один такт процесора. Об'єднавши два і більше конвеєрів можна отримати матричний комп'ютер, який обробляє масиви даних подібно тому, як скалярні (послідовні) машини обробляють окремі елементи таких масивів.

Комп'ютер класу *MISD* здатний обробляти один потік даних за допомогою кількох потоків команд. Але в більшості випадків кільком потокам команд необхідно мати кілька елементів даних (щоб від них була користь). Тому таких клас паралельних комп'ютерів використовується лише як теоретична модель.

До класу *MIMD* можна віднести більшість сучасних паралельних комп'ютерів, тому в межах цього класу можна розглянути окрему класифікацію паралельних архітектур.

Головною ознакою подальшої класифікації класу MIMD є спосіб підключення до основної (оперативної) пам'яті: використання єдиної спільної пам'яті (shared memory – SM) або розподіленої пам'яті (distributed memory – DM) (рис. 1.1).

Рисунок 1.1 – Способи підключення до основної пам'яті:
а) спільна пам'ять, б) розподілена пам'ять

На основі *SM* можуть бути побудовані векторні паралельні процесори (*parallel vector processor – PVP* ) і симетричні мультипроцесори (*symmetric multiprocessor – SMP*). Представниками таких комп'ютерних систем є багатоядерні комп'ютери.

Архітектура *DM* передбачає, що кожний процесор може використовувати тільки свою локальну пам'ять, а для доступу до даних інших процесорів необхідно використовувати операції передачі повідомлень *(message passing operations)*. Такій підхід використовується при побудові масивно-паралельних систем (*massively parallel processor – MPP*) та кластерів *(clusters)*.

Використання розподіленої пам'яті спрощує проблеми створення мультипроцесорів (сучасні кластерні системи містять сотні тисяч мікропроцесорів), але приводять до суттєвого ускладнення паралельного програмування.

Для спрощення написання прикладних програм створено різноманітні бібліотеки паралельного програмування, які використовують різні мови програмування та різні архітектури. Наприклад, для мови С++ для багатоядерних комп'ютерів зручною є бібліотека OpenMP, а для систем з розподіленою пам'яттю – бібліотека MPI.

## 1.2 Види паралелізму

Суть паралельної обробки даних полягає в розподілі всієї обчислювальної роботи на окремі частини і їх одночасному виконанні, що в підсумку має дати виграш в часі виконання всієї роботи. Використовуючи математичну термінологію, кажуть про декомпозицію початкової обчислювальної задачі. Відомі такі способи декомпозиції:

– за даними,
– за функціями (підзадачами),
– за часом.

Відповідно можна розрізняти паралелізм за даними, паралелізм за функціями (підзадачами) та паралелізм за часом.

### 1.2.1 Паралелізм за даними та паралелізм за функціями

Основна ідея паралелізму за даними полягає в тому, що одна операція виконується одночасно над всіма елементами масиву даних, наприклад, «помножити всі елементи масиву на задану константу». В програмах, де використовується паралелізм за даними, використовується глобальний простір імен на основі єдиного блоку пам'яті та багатьох процесорних блоків (ядер). Різні фрагменти масиву обробляються на різних процесорах (ядрах) паралельної машини. Таким чином, такий спосіб розпаралелювання виконується на машинах з архітектурою *SIMD*.

Характерною особливістю таких обчислень є їхня слабка синхронізація, тобто процесори працюють незалежно і немає гарантії, що в заданий момент часу на всіх процесорах виконується одна і та ж команда. Розподіл даних між процесорами задається в програмі. Роль програміста зводиться лише до розбиття початкових даних на рівні за величиною блоки $D_1$, $D_2$ , … , $D_n$ (рис. 1.2) та задання відповідних директив (опцій), а власне векторизація чи розпаралелювання виконується на етапі компіляції – під час переведення початкового тексту програми в машинні коди.

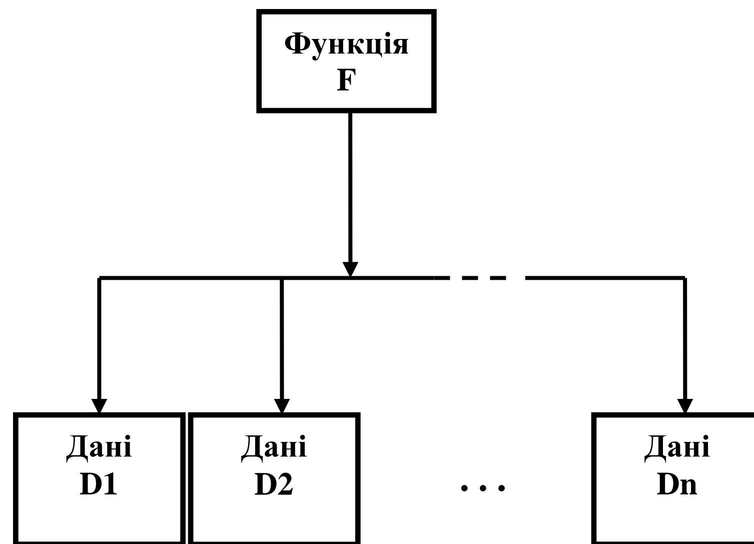



Рисунок 1.2 – Декомпозиція за даними

Стиль програмування, який базується на паралелізмі функцій (підзадач), полягає в розбитті всієї обчислювальної задачі на декілька відносно самостійних підзадач (методів або функцій $F_1$, $F_2$, … , $F_k$), які виконуються в окремому процесорі або ядрі (рис. 1.3).

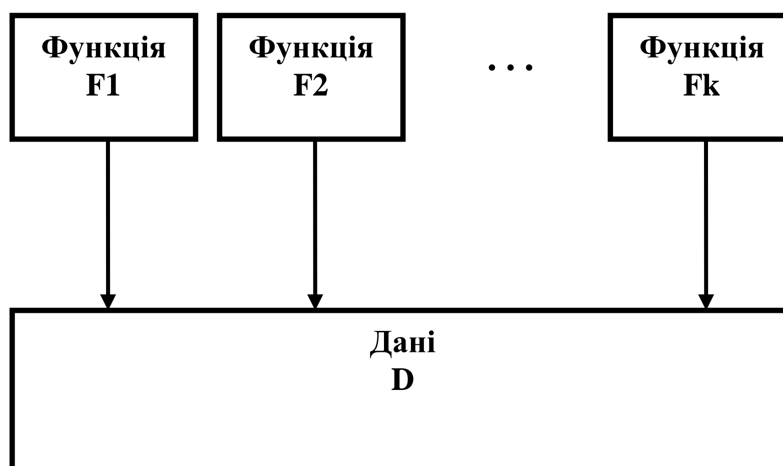



Рисунок 1.3 – Декомпозиція за підзадачами (функціями, методами)

Такому способу розпаралелювання обчислень теоретично відповідає архітектура *MISD*, але на практиці використовуються машини з

архітектурою *MIMD,* оскільки різні підзадачі, як правило, використовують і різні дані.

Для кожної підзадачі пишеться своя окрема функція чи програма, які виконуються на окремих процесорах і ядрах. Пам'ять може бути спільною або розподіленою.

Характерною особливістю таких обчислень є обмін проміжними та кінцевими даними між підзадачами. У випадку розподіленої пам'яті такий обмін даними можливий лише як обмін повідомленнями між паралельними процесами. Розпаралелювання по підзадачах реалізується набагато складніше, ніж розпаралелювання по даних внаслідок таких проблем:

– підвищена трудомісткість розробки програми (підзадачі мають бути приблизно однакового часу виконання та бути взаємно інформаційно незалежними);

– має бути мінімізований обмін даними між підзадачами (оскільки такий обмін потребує багато часу);

– має бути забезпечено оптимальне завантаження всіх процесорних блоків (в найкращому випадку кількість підзадач має відповідати кількості процесорних блоків чи бути кратною їх числу).

Корисно порівняти між собою дві найпоширеніші технології паралельної обробки даних.

По-перше, необхідно обов'язково враховувати особливості та обмеження практичної реалізації обчислень. Розмір блока даних і складність підзадачі мають бути такими, щоб число допоміжних операцій в процесорі (ядрі) не перевищувало число основних операцій. Іншими словами, сумарний час створення і ліквідації потоків має бути меншим часу обчислень. Така вимога буде виконана, якщо, наприклад, процес чи потік буде виконувати роботу, яка за трудомісткістю буде не меншою, ніж 2000 операцій ділення чисел з рухомою комою. Звичайно, кожна задача має свою специфіку і точні оцінки ефективності паралельної обробки можна визначити лише експериментально.

В цілому, паралелізм даних простіше та краще масштабується до дуже паралельного апаратного забезпечення, оскільки це зменшує або усуває спільні дані (тим самим зменшуючи проблему безпеки потоків). Крім того, паралелізм даних використовує той факт, що більше буває значень даних, ніж дискретних задач.

Нарешті, корисно враховувати ступінь структурованості паралелізму. Паралелізм даних має кращу структуровану паралельність, тобто, паралельні процеси та потоки стартують і фінішують в одному місці в програмі. На противагу цьому, паралелізм підзадач має тенденцію бути неструктурованим, а це означає, що паралельні процеси і потоки можуть починатись і закінчуватись в різних місцях програми. Програми з низьким

ступенем структурного паралелізму складніші в налагоджуванні і більше піддаються помилкам.

На практиці паралелізм даних та паралелізм функцій (підзадач) взаємно доповнюють один одного, тому у великих програмах часто застосовуються разом [5].

### 1.2.2 Паралелізм за часом виконання

Якщо розглядати категорію часу тільки з позицій математики, то можна помітити, що фундаментальні закони і класичної, і квантової динаміки передбачають еквівалентність причин та наслідків [6]. Іншими словами, теореми, які справедливі при зміні часу з «теперішнього» на «майбутній», будуть також справедливими при зміні часу з «теперішнього» на «минулий».

В [7] показано, що оберненість в часі справедлива тільки для інтегрованих динамічних систем з одним ступенем свободи. Прикладом таких систем є скінчені автомати при надходженні на їхні входи лише нульових значень (автономні скінчені автомати) [8]. Можна розрізняти прямі автономні скінчені автомати, які функціонують при зміні часу з «теперішнього» на «майбутній», та обернені автономні скінчені автомати, які функціонують при зміні часу з «теперішнього» на «минулий».

Для таких автономних автоматів послідовність змін станів в часі утворює в просторі станів системи замкнуту фазову траєкторію у вигляді кола. Нехай на цій фазовій траєкторії буде початковий стан $S_{beg}$ і завершальний стан $S_{end}$. Прямий автономний автомат буде функціонувати від початкового стану $S_{beg}$ в одну сторону, а обернений автономний автомат – в протилежну сторону.

Мета кожного автомата – найшвидше досягти завершального стану $S_{end}$. В загальному випадку розташування станів $S_{beg}$ і $S_{end}$ на фазовій траєкторії довільне, значить, довжина шляху від $S_{beg}$ до $S_{end}$ по колу в різні боки буде різною (рис. 1.4). Якщо обидва автомати починають працювати паралельно зі стану $S_{beg}$, тоді вони досягнуть стану $S_{end}$ в різні моменти часу. При досягненні стану $S_{beg}$ будь-яким з автоматів інший автомат також завершує роботу, оскільки поставлена загальна мета досягнута. Таким чином, завдяки паралельній роботі прямого і оберненого автоматів ми швидше отримуємо потрібний результат, в середньому – вдвічі швидше.

Паралелізм на основі декомпозиції за часом (темпоральний паралелізм) належить до категорії прихованого паралелізму і для його виявлення потрібно здійснити глибокий аналіз поставленої задачі. В деяких випадках необхідно переформулювати початкову задачу в рамках такого математичного апарату, який дозволяє використання обернених в

часі обчислень. Використання темпорального паралелізму в задачах завадостійкого кодування наведено в [9].

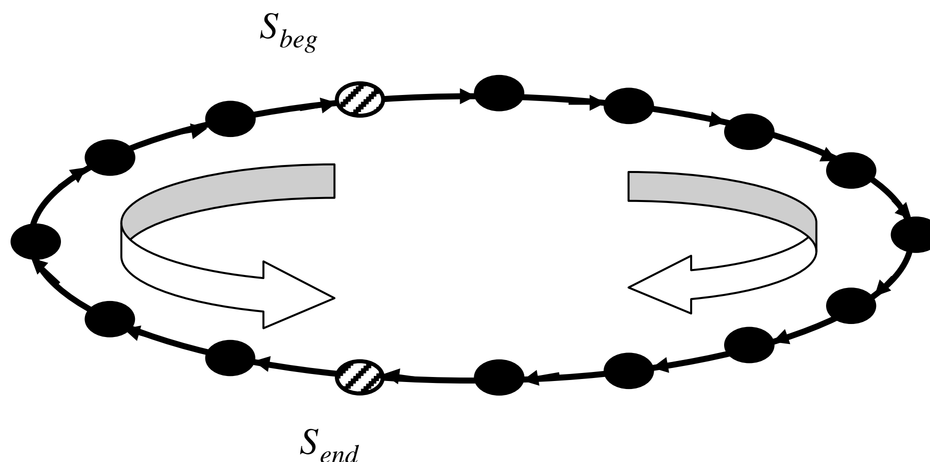



Рисунок 1.4 – Паралелізм за часом виконання для паралельних автономних автоматів

## 1.3 Етапи розробки паралельних алгоритмів

Вважатимемо, що відомо традиційний (послідовний) спосіб розв'язання деякої задачі і далі необхідно організувати її виконання з використанням паралельної обробки даних.

Загальна схема розробки паралельного алгоритму містить такі етапи:

– виконання декомпозиції задачі на складові частини одним із відомих способів;

– виявлення інформаційних залежностей;

– масштабування складових частин задачі;

– розподіл складових частин задачі між процесорами (ядрами).

*Етап декомпозиції* є першим етапом розробки паралельного алгоритму. Суть найвідоміших способів декомпозиції вже було детально розглянуто. Проблема може полягати у виборі того чи іншого способу.

Дуже часто вже наперед відома архітектура паралельної машини, де буде виконуватись задана обчислювальна задача. В цьому випадку архітектура машини визначатиме найоптимальніший варіант декомпозиції. Звичайно, можна використовувати одночасно декілька способів декомпозиції.

Після розбиття початкової задачі на складові частини виконується аналіз зв'язків між ними, тобто здійснюється *етап виявлення інформаційних залежностей*. При цьому необхідно розрізняти такі схеми взаємодії:

– *локальні* (на сусідніх процесорах) і *глобальні* (з участю всіх процесорів);

– *структурні* (відповідають типовим топологіям комунікацій згідно з вибраною на попередньому етапі архітектурою машини) і *довільні*;

– *статичні* (задаються на етапі проектування) та *динамічні* (визначаються під час роботи);

– *синхронні* (наступна операція виконується після закінчення попередньої всіма процесорами) і *асинхронні* (без очікування повного завершення всіх дій із передачі даних).

Дві підзадачі *A* та *B* вважаються інформаційно залежними, якщо результат виконання підзадачі *A* використовується як вхідні дані для підзадачі *B* або навпаки. Підзадачі *A* та *B* будуть також інформаційно залежними, якщо їх результати мають бути отримані одночасно для подальшого використання іншими підзадачами.

*Етап масштабування складових частин задачі* виконується в тому випадку, коли кількість наявних підзадач відрізняється від кількості наявних процесорів (ядер). Тут можливі два основних варіанти.

Якщо кількість $k$ підзадач перевищує кількість $n$ процесорів, то деякі підзадачі необхідно укрупнити, так щоб їх загальна кількість не перевищувала числа $n$. Якщо ж має місце нерівність ($k<n$), тоді варто провести деталізацію підзадач, тобто їх збільшити до числа $n$.

Існує навіть спеціальний термін – «*зернистість*», – який характеризує рівень декомпозиції початкової задачі на окремі підзадачі. Дрібнозернистий паралелізм може оптимально завантажити всі наявні процесори, однак важко аналізувати паралельну програму. При цьому є межа складності підпрограм, коли загальний час виконання програм вже не буде зменшуватись внаслідок зростання числа допоміжних операцій зі створення нових процесів та потоків.

Таким чином, в багатьох випадках необхідний ще один етап розробки паралельних алгоритмів – *етап розподілу підзадач між процесорами.*

Основний критерій успішності виконання цього етапу – ефективність використання процесорів, яка визначається як відносна частка часу, протягом якого процесори використовувались для обчислень, пов’язаних з виконанням поставленої задачі. Способи досягнення задовільних результатів в цьому напрямку базуються на таких самих принципах, як і в попередніх етапах: рівномірний розподіл обчислювального навантаження процесорів та мінімізація обмінів даних між ними.

Варто відзначити, що вимога мінімізації міжпроцесних обмінів може суперечити умові рівномірного завантаження. Можна розмістити всі підзадачі на одному процесорі і тим самим повністю ліквідувати міжпроцесний обмін, але завантаження процесорів в цьому випадку буде неоптимальним.

Вирішення питань балансування обчислювального навантаження значно ускладнюється, якщо схема обчислень може змінюватись під час розв’язання задачі. Для динамічного керування розподілом обчислювального навантаження часто використовується схема «*менеджер*

– *виконавці*». Відповідно до такої схеми виділяється окремий процесор (менеджер), якому доступна вся інформація про стан виконання всіх підзадач на інших процесорах (виконавцях). Процесор-менеджер отримує результати виконання підзадач від процесорів-виконавців, формує нові завдання та необхідні ресурси для їх виконання.

Завершення обчислень відбувається тоді, коли процесори-виконавці завершили виконання всіх переданих їм підзадач, а процесор-менеджер не має більше нових завдань [18].

## 1.4 Процеси і потоки

Після розробки паралельного алгоритму розв'язання поставленої задачі його необхідно записати у формі, яка сприймається комп'ютером. Таку можливість забезпечують алгоритмічні мови програмування двох типів:

– спеціалізовані мови паралельного програмування,

– стандартні мови високого рівня, доповнені операторами для розпаралелювання обчислень.

На практиці найчастіше використовуються мови програмування другого типу, їх і будемо в подальшому використовувати, зокрема, мови C++ та C#.

Після запису алгоритму за допомогою вибраної мови програмування буде отримано комп'ютерну програму, яка зберігається у вигляді файлів на деякому носії даних (зазвичай на магнітному чи оптичному диску). Далі виконується компіляція програми (наприклад, за допомогою пакета Microsoft Visual Studio) для отримання машинного коду програми. Програма в машинних кодах також зберігається у файлах на дисках, але іншого формату (найчастіше типу *.exe для операційної системи Windows).

На цьому завершується етап підготовки задачі до виконання. Далі програміст ініціює початок виконання комп'ютерної програми. Для пояснення всіх подальший дій будемо використовувати терміни «процес» і «потік» [10, 11].

Якщо програма – це статична послідовність команд і операторів, то процес – це програма на стадії її виконання, тобто в динаміці. Кожний раз після запуску на виконання файлу типу *.exe формується новий процес з окремим захищеним адресним простором в 4 Гбайт. Не тільки для різних, але і для однієї і тієї ж програми, викликаної кілька разів, створюється новий процес.

Кожний процес має набір атрибутів, зокрема:

– ідентифікатор процесу,

– базовий пріоритет,

– системні ресурси (квоти на машинний час, обсяг системної пам'яті та інші),

– інструменти міжпроцесної взаємодії,

– файли та системні бібліотеки.

Кожний блок процесу є представником цього процесу у Windows, (EPROCESS), в якому містяться всі атрибути процесу і покажчики на деякі структури даних.

В цілому процес можна розглядати як контейнер для всіх видів ресурсів, окрім одного – процесорного часу. Цей найважливіший ресурс розподіляється операційною системою між потоками.

Потік – це послідовність команд програми, які виконуються в процесорі протягом одного кванту процесорного часу (20 мс у Windows). Кожний процес починається з одного потоку – головного. Потік знаходиться в адресному просторі процесу, має пріоритет в межах базового пріоритету процесу і використовує його ресурси. Якщо в межах процесу динамічно створюються нові потоки, тоді всі ресурси процесу розподіляються між потоками цього процесу. Пріоритети потоків періодично змінюються, і тоді потоки з більшим пріоритетом витісняють потоки з меншим пріоритетом. Подібно процесу потік має свій набір атрибутів.

Процесор за допомогою апаратного таймера визначає момент закінчення чергового кванта, який був виділений даному потоку, і формує переривання. Далі процесор переміщає в структуру даних *CONTEXT* вміст всіх регістрів. Коли цей потік знову отримає квант машинного часу процесор відновить значення регістрів із його структури *CONTEXT*. Вся ця операція називається перемиканням контексту потоку.

Процеси і потоки мають ряд принципових відмінностей.

По-перше, потоки для свого створення, функціонування та ліквідації потребують набагато менше системних витрат, ніж процеси. Перемикання процесора з виконання одного потоку на виконання іншого потоку зводиться до однієї операції перемикання контексту: завантаження в регістри процесора нових значень.

По-друге, процеси «бачать» виділені їм пам’ять та інші ресурси, нічого не «знаючи» про існування інших процесів. Потоки одного процесу не тільки «знають» про існування один одного, але і конкурують між собою за спільні системні ресурси, використовуючи для цього різноманітні засоби синхронізації.

По-третє, дуже складно організувати обмін даними між процесами. Оскільки безпосередній обмін даними між ними заборонений, то для цього необхідно використовувати спеціальні засоби взаємодії: черги, конвеєри та ін. Потоки можуть обмінюватись даними набагато швидше.

В перших операційних системах було реалізовано однозадачний режим роботи, тобто в один момент часу міг бути лише один процес з одним потоком. Тому навіть не розрізняли між собою процеси і потоки.

В сучасних багатозадачних операційних системах в комп’ютері можуть одночасно існувати сотні процесів та потоків. Відповідно системі потрібно паралельно організувати їх спільну роботу.

Розглянемо детальніше особливості паралельної обробки даних в сучасних операційних системах.

### 1.5 Багатозадачність, багатопотоковість та паралелізм

Багатозадачність (*multitasking*) – це властивість операційної системи або середовища програмування забезпечувати можливість паралельної або псевдопаралельної обробки кількох процесів.

Паралелізм при багатозадачності може бути реалізований по-різному. Розглянемо спочатку його реалізацію на одному фізичному процесорі, і саме так функціонували операційні системи протягом кількох десятиріч.

Нехай, наприклад, потрібно виконати три незалежні між собою задачі кожна тривалістю три такти часу. Послідовне виконання цих задач потребуватиме 9 тактів часу (рис. 1.5).

Тепер організуємо роботу таким чином, щоб кожна задача виконувалась потактно (рис. 1.6). Загальний час виконання задач не зміниться (на практиці навіть трохи збільшиться за рахунок частішого перемикання між задачами), однак ми отримаємо паралельне виконання задач.

На основі такого паралелізму можливі два основних типи багатозадачності: кооперативна та пріоритетна.

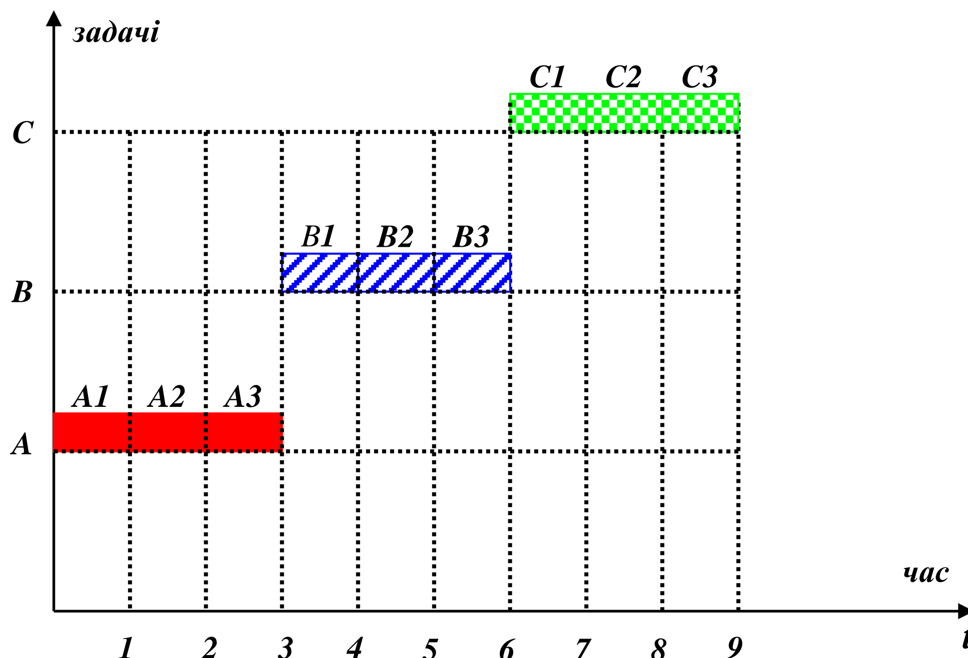



Рисунок 1.5 – Послідовне виконання задач

При кооперативній багатозадачності процес може захопити для себе стільки процесорного часу, скільки необхідно процесу. Наступний процес (задача) починає виконуватись лише тоді, коли попередній процес добровільно погодиться звільнити процесор (наприклад, при відсутності необхідного йому додаткового ресурсу). Недоліки такого підходу: довге

очікування реалізації операцій введення–виведення, неможливість самостійного звільнення процесора за наявності помилок в програмі.

В сучасних операційних системах найчастіше використовується пріоритетна (витісняюча) багатозадачність, коли операційна система слідкує за передачею керування від одного процесу до іншого. Кожний процес отримує свій квант машинного часу, після закінчення якого він зобов'язаний звільнити процесор. Операційна система може завершити будь-який процес і достроково при настанні деяких подій, наприклад, для виконання операцій введення–виведення або появи запиту від більш пріоритетної програми.

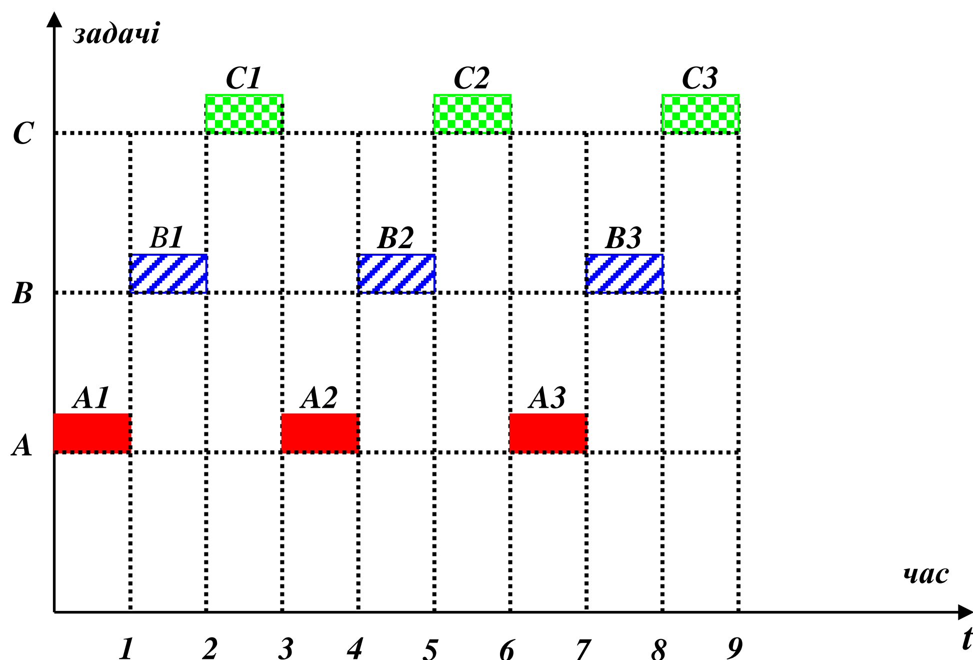


Рисунок 1.6 – Паралельне потактне виконання задач

Звичайно, за наявності лише одного фізичного процесора будь-яка стратегія паралелізму не дасть виграшу в часі. Але навіть і такий псевдопаралелізм має багато переваг, наприклад, це забезпечує швидку реакцію операційної системи на команди користувача. Тому паралельна обробка даних давно вже почала впроваджуватись, навіть незважаючи на відсутність відповідної апаратної підтримки.

І коли така апаратна підтримка з'явилась у вигляді багатоядерних комп'ютерів, тоді можна отримати справжній (одночасний) паралелізм, який точно відповідає англомовному терміну *concurrency.* Щоб зрозуміти, яким чином апаратна підтримка дає вже суттєвий виграш у часі виконання програм, розглянемо детальніше поняття багатопотоковості.

Сучасна багатопотокова обробка має декілька рівнів:

– потоки рівня користувача (створюються і керуються з прикладних програм),

– потоки рівня ядра (створюються і керуються операційною системою),

– апаратні потоки (розглядаються з позицій апаратури).

В найпростішому варіанті процес може мати один програмний потік, який реалізується операційною системою як потік рівня ядра і виконується як один апаратний потік в однопотоковому скалярному процесорі (*Single-Issue, Single-Thread, SIST*).

Трохи складніші суперскалярні мікропроцесори збільшують продуктивність роботи за рахунок одночасного виконання в одному циклі кількох команд одного потоку (тобто апаратно реалізується конвеєризація як один з різновидів паралелізму). За класифікацією Флінна одноядерні суперскалярні процесори відносять до архітектури *SISD*.

Більш суттєвого приросту в продуктивності роботи можна досягти при виконанні багатопотокових програм. Тут важливо розібратись як між собою співвідносяться програмна багатопотоковість (*multithreading*) з апаратними потоками.

Як і у випадку із багатозадачністю, програмна багатопотоковість також може бути реалізована на одноядерному процесорі. Така можливість реалізована в процесорних ядрах, які підтримують технологію гіперпотоковості (*Hyper-Threading Technology, HP Technology*) [5].

Ядро з підтримкою HP Technology має дещо складнішу структуру: арифметико-логічний пристрій (АЛП) для операцій з фіксованою комою, АЛП для операцій з рухомою комою, додаткові регістри та додаткову логіку. Фактично таке апаратне ядро можна розглядати як два логічних ядра: одне логічне ядро для виконання операцій з фіксованою комою, а друге – для операцій з рухомою комою. Якщо два програмних потоки містять команди із різними типами операцій, то вони можуть виконуватись одночасно. Якби кожний потік містив лише операції одного типу, тоді вдалось би вдвічі швидше виконати таку двопотокову програму. Звичайно таких програм майже не буває, тому середній виграш в продуктивності роботи не перевищує 30%, але це також гарний результат.

Такий спосіб обробки даних називають одночасною багатопотоковістю (*Simultaneous Multi-Threading, SMT*).

Коли у процесора є декілька фізичних ядер, тоді це називається багатопроцесорною обробкою на кристалі (*Chip Multiprocessing, CMP*). Багатоядерні процесори забезпечують реальну апаратну багатопотоковість. Кожне ядро виконує апаратні потоки незалежно від інших апаратних потоків, тобто паралелізм забезпечується тим, що кожний з потоків обробляється власним ядром (рис. 1.7). Обмін даними між потоками забезпечує спільна пам’ять.

Сучасні багатоядерні процесори часто підтримують додатково *HP Technology*, тобто кожне фізичне ядро ще поділяється на два логічних ядра, що вдвічі збільшує кількість потоків.

Виконання багатьох потоків на багатоядерному процесорі називають ще багатопотоковістю на кристалі (*Chip Multi-Threading, CMT*).

Якщо на одному процесорі (ядрі) паралелізм лише моделюється, а за наявності кількох фізичних процесорів (ядер) реалізується справжній паралелізм, який вже дає суттєвий виграш у часі виконання системних і прикладних програм.

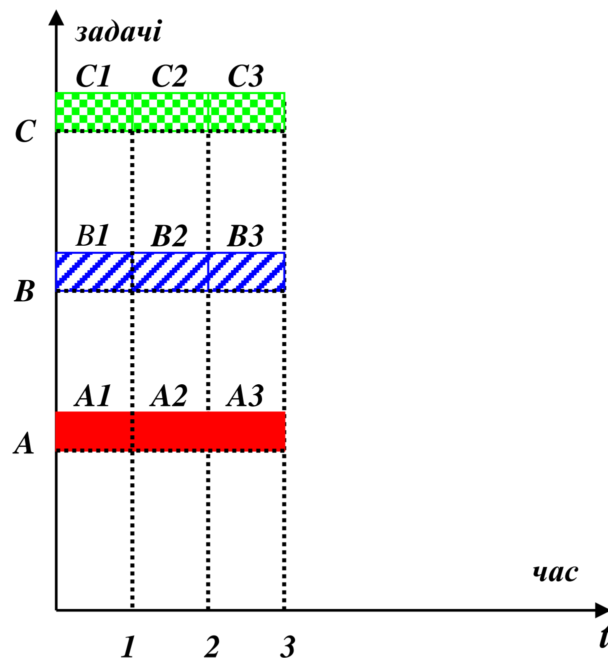



Рисунок 1.7 – Паралельне одночасне виконання задач

**Контрольні запитання**

1. Дайте порівняльну характеристику паралельних комп’ютерних архітектур за класифікацією Флінна.
2. Як типи комп’ютерних архітектур належать до класу MIMD?
3. Які існують типи паралелізму?
4. В чому суть паралелізму за часом виконання?
5. Які основні етапи розробки паралельних алгоритмів?
6. В чому полягає відмінність між процессами і потоками?
7. Які основні атрибути мають процеси та потоки?
8. Чим відрізняється пріоритетна багатозадачність від кооперативної багатозадачності?
9. Які потоки використовуються при багатопотоковій обробці даних?
10. За рахунок чого підвищується продуктивність роботи з використанням технології гіперпотоковості (Hyper-Threading)?

# 2 Багатопотокове програмування мовою C# за допомогою класу Thread

## 2.1 Створення нових потоків виконання

Для роботи з потоками виконання у середовищі *.NET* мовою C# використовується клас *System.Threading.Thread* [12]. Для створення 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. Запускає новий потік метод *Start()* об'єкта класу *Thread*. Розглянемо приклад двопотокової програми в консольному режимі.

**Приклад 2.1.** Двопотокова програма в консольному режимі.

```
using System.Threading;
class Program
  {
public static void ThreadFunc1()
{
   Console.WriteLine("Hello of additional  thread {0}! ",
                       Thread.CurrentThread.GetHashCode());
}
static void Main(string[] args)
{
    Console.WriteLine("Hello of main thread {0}! ",
                       Thread.CurrentThread.GetHashCode());
  Thread thread1 = new Thread(newThreadStart(Program.ThreadFunc1));
   thread1.Start();
   thread1.Join();
   Console.WriteLine("new thread ending");
   Console.ReadLine();
 }
}
```

На рис. 2.1 показано результат виконання програми. В цій програмі спочатку створюється головний потік програми, який видає повідомлення «*Hello of main thread …*». Далі створюється об'єкт *MyThread* класу *Thread*. Метод *Start()* цього об'єкта створює додатковий потік, тобто починає виконуватися метод *ThreadFunc,* на який вказує делегат із конструктора класу *Thread.* Додатковий потік видає повідомлення «*Hello of thread …*». Метод *Join()* блокує виконання головного потоку, поки не завершиться додатковий потік.

```
Hello of main thread 9!
Hello of additional thread 10!
new thread ending
```

Рисунок 2.1 – Результат виконання програми 1 (варіант 1)

Якщо в програмі буде відсутній метод *Join()*, тоді може статися такий варіант виконання програми, коли спочатку виконається головний потік програми, а вже потім виконається додатковий потік, тобто рядок *«Hello from thread ...»* з'явиться останнім (рис. 2.2).

```
Hello of main thread 9!
new thread ending
Hello of additional thread 10!
```

Рисунок 2.2 – Результат виконання програми 1 (варіант 2)

## 2.2 Пріоритети потоків

Всі потоки при створенні отримують однаковий пріоритет (*priority*), тобто мають однакові права на використання центрального процесора. Однак часто виникає потреба в тому, щоб дати окремим потокам деяку перевагу. Наприклад, якщо одночасно з редагуванням одного документа відбувається друк іншого документа, то потоку редагування варто підвищити пріоритет для більш швидкої реакції на запити користувача.

Для задання величини пріоритету використовується властивість *Priority*, яка може мати 5 значень:

- *Lowest* (найнижчий);
- *BelowNormal* (нижче нормального);
- *Normal* (нормальний);
- *AboveNormal* (вище нормального);
- *Highest* (найвищий).

При створенні всі потоки мають однаковий пріоритет, але це значення може бути змінено. Величина пріоритету впливає на черговість виконання і завершення потоків. Розглянемо програму в *Windows Form*, в якій створюються два додаткових потоки.

**Приклад 2.2** Двопотокова програма з використанням пріоритетів.

```
class Program
{
public static void ThreadFunc1()
{
   Console.WriteLine("Hello of first thread {0}! ",
                     Thread.CurrentThread.GetHashCode());
```

```
}
public static void ThreadFunc2()
{
   Console.WriteLine("Hello of second thread {0}! ",
                          Thread.CurrentThread.GetHashCode());
}
static void Main(string[] args)
{
    Console.WriteLine("Hello of main thread {0}! ",
                          Thread.CurrentThread.GetHashCode());
  Thread thread1 = new Thread(newThreadStart(Program.ThreadFunc1));
 Thread thread2 = new Thread(new ThreadStart(Program.ThreadFunc2));
          thread1.Priority = ThreadPriority.Normal;
          thread2.Priority = ThreadPriority.Normal;
   thread1.Start();
   thread2.Start();
   thread1.Join();
   thread2.Join();
 Console.WriteLine("Hello of main thread {0}! ",
                          Thread.CurrentThread.GetHashCode());
   Console.ReadLine();
 }
  }
```

Оскільки обидва потоки мають однакові пріоритети, тому першим виконається той потік, який першим запускається, тобто потік 1 (рис. 2.3).

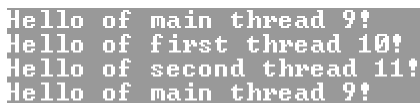



Рисунок 2.3 – Результат виконання програми 2.2 (варіант 1)

Однак, якщо потоку 1 надати найнижчий пріоритет, потоку 2 – найвищий:

*thread1.Priority = ThreadPriority.Lowest*
*thread2.Priority = ThreadPriority.Highest*

то першим завершиться потік 2 (рис. 2.4). Така ситуація, коли пріоритети визначають черговість виконання, характерна тільки для рівноцінних потоків. В загальному випадку пріоритети частіше впливають на кількість процесорного часу, що надається потоку. Тобто потік з з більшою кількістю обчислень і вищим пріоритетом може завершитись пізніше потоку з малою кількістю обчислень та низьким пріоритетом.

```
Hello of main thread 9!
Hello of second thread 11!
Hello of first thread 10!
Hello of main thread 9!
_
```

Рисунок 2.4 – Результат виконання програми 2.2 (варіант 2)

Варто відзначити, що можна відновити початковий порядок виконання потоків, якщо першому потоку надати можливість до свого завершення заблокувати виконання інших потоків:

*MyThread1.Join();*

Метод *Join*() завжди гарантує завершення першим для того потоку, який його викликає.

## 2.3 Стани потоків

Кожний потік користувача починає своє існування зі стану *Unstarted*. Після виклику методу *Start()* він отримує в своє розпорядження центральний процесор, тобто входить в стан *Running*. Після завершення потоку він переходить в стан *Stopped.*

Якщо потік необхідно тимчасово зупинити на задану кількість *n* мілісекунд, то викликається метод *Sleep(n)*, в результаті чого потік переходить в стан *WaitSleepJoin*. Після завершення часу очікування потік знову повернеться в стан *Running*.

В стан *WaitSleepJoin* потік також перейде в результаті блокування (детальніше про це в наступних розділах) або після виклику методів *Join()* чи *Monitor.Wait()*. Потік може вийти з цього стану, якщо інший потік викличе метод *Interrupt()*. Варто відмітити, що при виклику методу *Interrupt()* не вказується конкретний адресат. Це означає, що його може отримати зовсім інший потік, який може бути саме в цей момент часу заблокований, наприклад, чекаючи доступу до якогось спільного ресурсу. В результаті буде розблокований не той потік, на який розраховував програміст.

Блокований потік може бути також звільнений іншим потоком за допомогою методу *Abort()*. Однак в цьому випадку потік перейде в стан *AbortRequested*. Якщо відразу за викликом методу *Abort()* буде викликано метод *ResetAbort()*, тоді потік повернеться в активний стан, тобто в *Running*. В іншому випадку потік аварійно завершиться і перейде в стан *Stopped*. Існує також велика відмінність між методами *Interrupt()* та *Abort(),* якщо їх викликати для незаблокованого потоку. Якщо *Interrupt()* нічого не виконує, поки потік не дійде до наступного блокування, то метод *Abort()* починає діяти негайно і здійснює аварійне завершення незаблокованого потоку.

Розглянемо приклад програми в *Windows Forms*, в якій для обчислення арифметичного виразу *z=a*b+c/d* створюється два додаткових потоки:

потік *thread1* для обчислення *z1=a*b* і потік *thread2* для обчислення *z2=c/d.*

**Приклад 2.3.** Двопотокова програма для арифметичних обчислень.

```
using System.Threading;
class Program
{
static long i;
static int a, b, c, d;
static int y1, y2;
public static void ThreadFunc1()
{
   for (i = 0; i < 500000000; i++) ;
   y1 = a * b;
   Console.WriteLine("Hello of thread {0}! ",
      Thread.CurrentThread.GetHashCode());
}
private static void ThreadFunc2()
{
   for (i = 0; i < 500000000; i++) ;
   y2 = c/d;
   Console.WriteLine("Hello of thread {0}! ",
      Thread.CurrentThread.GetHashCode());
}
public static void CheckTime(Object state)
{
   Console.WriteLine(DateTime.Now);
}

static void Main(string[] args)
{
   int y;
   a = 10; b = 7;
   c = 15; d = 3;
   TimerCallback tc = new TimerCallback(CheckTime);
           Thread thread1 = new Thread(new
  ThreadStart(Program.ThreadFunc1));
   Thread thread2 = new Thread(new
  ThreadStart(Program.ThreadFunc2));
   Timer t = new Timer(tc, null, 0, 1000);

   Console.WriteLine("MainThreadState=" +
  Thread.CurrentThread.ThreadState);
   Console.WriteLine("thread1.ThreadState=" + thread1.ThreadState);
```

```
Console.WriteLine("thread2.ThreadState=" + thread2.ThreadState);

thread1.Start();
thread2.Start();
Console.WriteLine("thread1.ThreadState=" + thread1.ThreadState);
Console.WriteLine("thread2.ThreadState=" + thread2.ThreadState);

thread1.Join();
thread2.Join();
Console.WriteLine("thread1.ThreadState=" + thread1.ThreadState);
Console.WriteLine("thread2.ThreadState=" + thread2.ThreadState);
Console.WriteLine("MainThreadState=" +
                    Thread.CurrentThread.ThreadState);
Console.WriteLine("new threads ending");
Console.WriteLine("y1=" + y1);
Console.WriteLine("y2=" + y2);
y = y1 + y2;
Console.WriteLine("y=" + y);
Console.WriteLine(DateTime.Now);
t.Dispose();
Console.ReadLine();
}
}
```

На рис. 2.5 показано результат виконання програми.

```
MainThreadState=Running
thread1.ThreadState=Unstarted
thread2.ThreadState=Unstarted
23.04.2017 22:56:43
thread1.ThreadState=Running
thread2.ThreadState=Running
23.04.2017 22:56:44
23.04.2017 22:56:45
23.04.2017 22:56:46
Hello of thread 10!
Hello of thread 11!
thread1.ThreadState=Stopped
thread2.ThreadState=Stopped
MainThreadState=Running
new threads ending
y1=70
y2=5
y=75
23.04.2017 22:56:46
```

Рисунок 2.5 – Результат виконання програми 2.3

Проаналізуємо детально послідовність станів всіх потоків. Спочатку існує лише головний потік програми і він знаходиться в стані *Running*. Після створення двох нових потоків *thread1* та *thread2* вони переходять в стан *Unstarted.* Після виклику двох методів *Start()* починає виконуватися метод *ThreadFunc1* першого потоку та метод *ThreadFunc2* другого потоку,

відповідно ці потоки входять в стан *Running*. Звичайно, обчислення двох арифметичних виразів *y1* та *y2* потребує дуже мало часу для сучасного комп’ютера, тому введемо ще штучну затримку на декілька секунд, ввівши цикл *for* до зазначених методів обох потоків.

Для того, щоб додаткові потоки не заважали один одному викликаємо методи *Join()*. Після завершення паралельної роботи обох потоків вони переходять в стан *Stopped*, і головний потік закінчує обчислення виразу: $y=y1+y2$.

### 2.4 Синхронізація потоків

Багатозадачна операційна система дозволяє керувати багатьма процесами і потоками одночасно, причому як системними, так і прикладними. Природно, виникає запитання про те, яким чином в комп’ютері з одним центральним процесором (можна ще уточнити – одноядерним) може одночасно виконуватись багато програм. Адже в будь-який момент часу процесор може працювати лише з одним процесом (потоком).

Суть мультипрограмного режиму роботи полягає в тому, що всі потоки процесів отримують в своє розпорядження центральний процесор тільки в маленькі проміжки часу (кванти часу), а в інший час знаходяться в різних типах станів готовності і очікування (*WaitSleepJoin, Suspended* та ін.). Процесор по черзі обслуговує всі потоки, що і створює ілюзію одночасної роботи.

Будемо називати потоки паралельними, якщо вони знаходяться в активному (*Running*) стані чи в одному із станів готовності або очікування. Паралельні потоки можуть бути незалежними або взаємодіючими.

Незалежні потоки не впливають на результати роботи один одного, оскільки в них не перетинаються файли початкових даних та області оперативної пам’яті, де зберігаються проміжні та кінцеві результати роботи. Вони можуть тільки бути причиною затримки один одного, тому що використовують один процесор.

Взаємодіючі потоки спільно використовують також і деякі загальні об’єкти (файли, області пам’яті); результати виконання одного потоку можуть залежати від ходу виконання іншого потоку. При виконанні взаємодіючих паралельних потоків можуть виникати «гонки» або взаємне блокування. «Гонки» з’являються, коли два чи більше потоків потребують доступу до одного фрагмента пам’яті, і тільки один із них може виконати цю операцію безпечно. Прикладом взаємного блокування може бути ситуація, коли з двох потоків кожний чекає звільнення ресурсу іншим потоком. Обидва чекають один одного, і жодний із них не може продовжити своє виконання, поки не буде виконано те, чого він очікує, в результаті потоки залишаються в стані довічного очікування.

Для вирішення таких проблем і використовують різні механізми синхронізації, які дозволяють взаємодіючим потокам коректно працювати із спільними ресурсами та уникати можливості виникнення тупикових ситуацій: блокування пам'яті, системи переривань, семафорів, моніторів, асинхронних делегатів та ін.

Відразу варто звернути увагу на те, що при розгляді задач синхронізації виключається центральний процесор, оскільки для нього ця задача вирішується диспетчером задач операційної системи за наперед визначеними алгоритмами і користувач не може їх змінювати. В подальшому будуть розглядатись лише ті спільно використовувані ресурси, до яких заборонено одночасний доступ кількох потоків. Такі ресурси, до яких в кожний момент часу може мати доступ тільки один потік, будемо називати критичними.

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.

Блокування пам'яті, тобто тимчасове призупинення, належить до найпростіших способів синхронізації. Суть його полягає в забороні одночасного виконання двох і більше потоків, які звертаються до спільної комірки пам'яті. Як правило, будь-яке поле (змінна), доступне кільком потокам, має читатись і записуватись з блокуванням пам'яті. Блокування само по собі дуже швидке і потребує лише кількох десятків наносекунд, якщо власне блокування не відбувається. Якщо ж блокування необхідне, тоді подальше перемикання задач займає вже мікросекунди або мілісекунди на перепланування потоків.

Для виконання цієї задачі створюється об'єкт *Lock,* що означає «замок». Об'єкти класу *Lock* необов'язково створювати явно, для них в мові C# вже передбачено ключове слово *lock*. Коли два потоки одночасно борються за першочерговий доступ (в нашому випадку за право володіння об'єктом *locker*), один потік переходить в стан *WaitSleepJoin* (блокується), поки блокування не звільняється. В цьому випадку це гарантує те, що тільки один потік зможе виконати критичну область коду (інкремент змінної). Код, який захищений таким чином від невизначеності багатопотокового програмування, називається потокобезпечним.

Безпечний інкремент змінної в потоці є настільки поширеною задачею, що існує спеціальний клас для виконання цієї задачі – *InterLocked*. В таблиці 2.1 перераховано його загальнодоступні статичні методи.

Таблиця 2.1 – Методи класу *InterLocked*

| Метод | Опис |
|---|---|
| *CompareExchange()* | Порівняння двох значень |
| *Increment()* | Зменшення змінної на одиницю (інкремент) |
| *Exchange()* | Обмін значеннями двох змінних |
| *Decrement()* | Збільшення змінної на одиницю (декремент) |

На відміну від класу *Lock* клас *InterLocked* не забороняє іншим потокам змінювати спільну змінну, він лише забезпечує коректну почергову операцію (інкремента чи декремента) від різних потоків.

Варто зазначити, що при неправильному використанні у блокування можуть бути і негативні наслідки – взаємоблокування, «гонки» блокувань, зменшення загального ефекту паралелізму. Остання ситуація може мати місце, коли дуже багато програмного коду міститься в конструкції *lock*, змушуючи інші потоки простоювати, поки один потік виконується.

**Приклад 2.4.** Трипотокова програма з використанням об'єкта класу *Lock*.

```
using System.Threading;
class Program
{
static int m;
static bool done;
static object locker = new object();
public static void PrintFunc1()
{
   lock (locker)
   {
      if (!done)
      {
         m++;
         Console.WriteLine(Thread.CurrentThread.Name + "m=" + m);
         done = true;
      }
   }
}
private static void ThreadFunc1()
{
   PrintFunc1();
   Thread.Sleep(1000);
}
private static void ThreadFunc2()
{
   PrintFunc1();
```

```
        Thread.Sleep(1000);
    }
    private static void ThreadFunc3()
    {
        PrintFunc1();
        Thread.Sleep(1000);
    }
    static void Main(string[] args)
    {
        Thread thread1 = new Thread(new ThreadStart(ThreadFunc1));
        Thread thread2 = new Thread(new ThreadStart(ThreadFunc2));
        Thread thread3 = new Thread(new ThreadStart(ThreadFunc3));
        thread1.Start();
        thread2.Start();
        thread3.Start();
        Thread.Sleep(1000);
        Console.ReadLine();
        }
        }
```

Кожний з потоків звертається до спільної функції *PrintFunc1()*, в якій здійснюється інкремент змінної *m*. Обмеження полягає в тому, що операція інкремента має бути виконана тільки один раз, незалежно від кількості звертань до функції *PrintFunc1()*. Таким чином, той потік, який першим викличе цю функцію, і виконає операцію інкремента, а виклик від іншого потоку має бути заблоковано.

Розглянуте раніше ключове слово *lock* – це насправді скорочений спосіб використання класу *Monitor*. Однак цей клас не обмежується тільки задачею блокування доступу до спільного ресурсу, його можливості значно більші. В табл. 2.2 перераховано його загальнодоступні статичні методи.

При виклику методу *Enter()* робота з об’єктом обмежується лише одним потоком, інші призупиняються і очікують. На відміну від простого блокування за допомогою об’єктів класу *Lock* в цьому випадку жоден призупинений процес не ігнорується.

Потік, що утримує блокування, може викликати метод *Pulse()* і вказати в ньому конкретний потік, який може першим переміщатися в чергу готових до виконання. За допомогою методу *PulseAll()* можна всі призупинені потоки перевести в чергу готових до виконання.

Як тільки блокування знімається (після виклику методу *Enter()* або *Wait()* ), перший потік із черги готових отримує дозвіл працювати з об’єктом. Таким чином всі потоки, які зробили заявку на роботу із певним об’єктом, отримають таку можливість, але не одночасно.

Таблиця 2.2 – Методи класу *Monitor*

| Метод | Опис |
|---|---|
| *Enter()* | Початок блокування об'єкта. Якщо об'єкт вже заблоковано іншим потоком, то виконання поточного потоку призупиняється до звільнення об'єкта |
| *Exit()* | Звільнення блокування об'єкта |
| *Pulse()* | Інформування наступного потоку, який призупинено, про можливість продовження роботи |
| *PulseAll()* | Інформування всіх призупинених потоків про можливість продовження роботи |
| *TryEnter()* | Намагання заблокувати переданий об'єкт |
| *Wait()* | Зняття всіх блокувань і призупинення поточного потоку до тих пір, поки інший потік не викличе метод *Pulse()* |

**Приклад 2.5.** Трипотокова програма з використанням класу *Monitor*.

```
using System.Threading;
class Program
{
static int m;
static bool done;
static object locker = new object();
public static void PrintFunc1()
{

    lock (locker)
    {
        m++;
        Monitor.Pulse(locker);
        Console.WriteLine(Thread.CurrentThread.Name + "m=" + m);
        done = true;
    }
}
private static void ThreadFunc1()
{
    PrintFunc1();
    Thread.Sleep(1000);
}
private static void ThreadFunc2()
{
    PrintFunc1();
    Thread.Sleep(1000);
}
```

```
private static void ThreadFunc3()
{
   PrintFunc1();
   Thread.Sleep(1000);
}
static void Main(string[] args)
{
   Thread thread1 = new Thread(new ThreadStart(ThreadFunc1));
   Thread thread2 = new Thread(new ThreadStart(ThreadFunc2));
   Thread thread3 = new Thread(new ThreadStart(ThreadFunc3));
   thread1.Start();
   thread2.Start();
   thread3.Start();
   Thread.Sleep(1000);
   Console.ReadLine();
  }
 }
```

Ця програма подібна до раніше розглянутої: кожний з трьох потоків звертається до спільної функції *PrintFunc1()*, в якій здійснюється інкремент змінної *m*. Потоки по черзі отримують доступ до операції інкремента, в результаті за значенням змінної *m* можна визначити загальну кількість потоків. Нагадаємо, що за допомогою об'єкта класу *Lock* можна було визначити лише факт звертання одного із потоків.

## 2.5 Асинхронні делегати

Раніше вже розглядався один із варіантів передавання даних в потік. Часто виникає інша потреба – повернення результатів роботи після завершення потоку. Зручним способом виконання цієї задачі є використання асинхронних делегатів (*delegates*), які дозволяють передавати в обох напрямках будь-яку кількість параметрів з одночасним контролем їх типу.

Делегат – це такий клас, об'єкт якого посилається на метод. Делегат може посилатись або на один метод (якщо він є нащадком класу *System.Delegate*), або на багато методів (якщо він є нащадком класу *System.MulticastDelegate*). В подальшому будуть використовуватись делегати-нащадки останнього класу, в якому передбачені методи: *Invoke(), BeginInvoke()* та *EndInvoke().* Метод *Invoke()* виконує синхронний виклик делегата, а два останніх методи – асинхронні виклики.

Розглянемо програму, в якій для обчислення заданого арифметичного виразу використовується функція *Summa().* В цю функцію необхідно передати два початкових параметри і отримати результат обчислень.

**Приклад 2.6.** Програма з використанням делегата.

```
using System.Threading;
class Program
   {
       static float a, b;
       private delegate float MyDelegate(float a, float b);
      //Метод для обчислення заданого виразу
      private static float Summa(float a,float b)
      {
         float y;
         Console.WriteLine("Calculation of expression in thread {0}!",
                   Thread.CurrentThread.GetHashCode());
         y=a+b;
       Thread.Sleep(3000);
       return y;
      }
      static void Main(string[] args)
      {
          a = 10;
          b = 7;
          MyDelegate  del = new MyDelegate(Summa);
         IAsyncResult endOp1 = del.BeginInvoke(a,b, null, null);
         Thread.Sleep(2000);
         //Завершити асинхронний виклик
         Console.WriteLine("Waiting…");
         float result1 = del.EndInvoke(endOp1);
         Console.WriteLine("Sum : {0}",result1);
         Console.ReadLine();
      }
   }
```

Спочатку створюється окремий потік, який здійснює виклик функції *Summa()* для обчислення заданого виразу. Метод *BeginInvoke()* починає асинхронний виклик делегата *del.* Метод *EndInvoke()* завершує асинхронний виклик делегата *del,* повертаючи його значення.

Якщо на момент виклику методу *EndInvoke()* делегат ще не завершив свою роботу, тоді викликаючий потік буде заблоковано до закінчення роботи потоку, який виконує асинхронний виклик.

## 2.6 Багатопотокове програмування на основі пулу потоків

Якщо в програмі багато потоків, то досить важко організувати їх оптимальну взаємодію, багато часу буде витрачатись на очікування потоків в різних чергах. Для таких випадків в C# передбачено спеціальний

механізм – пул (*pool*) потоків, для організації якого передбачено і спеціальний клас *ThreadPool*.

Пул – це така сукупність потоків, для яких передбачено централізоване керування. Немає потреби створювати кожний окремий потік за тією процедурою, що була описана в попередніх розділах, пул створюється весь відразу. Більше того, навіть не потрібно попередньо оголошувати про його створення, пул створюється автоматично при першому звертанні до нього. Для того, щоб поставити один потік в пул, просто викликається метод *ThreadPool.QueueUserWorkItem*, якому передається відповідний делегат. Розглянемо текст програми, в якій створюється пул потоків, які по черзі створюються і завершуються.

**Приклад 2.7.** Програма обчислення суми елементів одновимірного масиву з використанням пулу потоків.

```
using System.Threading;
class Program
{
static int number = 20;    static int sum = 0;
static int[] mas = new int[number];
static Random rnd = new Random();
public class EntryPoint
{
   public static void TimerProc(object state)
   { }
}
public static void Main()
{
     int i;      int workerThreads;     int portThreads;
   // Заповнимо матрицю випадковими числами
          for (i = 0; i < number; i++)
              mas[i] = rnd.Next(1, 10);
   Console.WriteLine("Processor=" + Environment.ProcessorCount);
   ThreadPool.GetMaxThreads(out workerThreads, out portThreads);
   Console.WriteLine("\nMaximum worker threads: \t{0}" +
      "\nMaximum completion port threads: {1}",
      workerThreads, portThreads);
   int MaxThreadsCount = 4;
   // Встановимо максимальну кількість робочих потоків
   ThreadPool.SetMaxThreads(MaxThreadsCount, MaxThreadsCount);
   // Встановимо мінімальну кількість робочих потоків
   ThreadPool.SetMinThreads(2, 2);
   for (i = 0; i < number; i++)
      Console.WriteLine("i=" + i + " mas=" + mas[i]);
   Console.WriteLine("start time=" + DateTime.Now.Millisecond);
```

```
    for (i = 0; i < number; i++)
    // Створимо пул потоків
      ThreadPool.QueueUserWorkItem(Function, mas[i]);
    ThreadPool.GetMaxThreads(out workerThreads, out portThreads);
    Console.WriteLine("\nMaximum worker threads: \t{0}" +
        "\nMaximum completion port threads: {1}",
        workerThreads, portThreads);
    Console.WriteLine("end time=" + DateTime.Now.Millisecond);
    Console.ReadLine();
}
public static void Function(object instance)
{
    int z = (int)instance;
    sum = sum + z;
    Thread.Sleep(500);
    Console.WriteLine(" z=" + z + " sum=" + sum);
  }
  }
```

На рис. 2.7 показано результат виконання програми. Спочатку масив заповнюється випадковими числами. Далі задається максимальна (4) та мінімальна (2) кількості потоків в пулі. При створенні пулу потоків вказується назва методу (*Function*), в якому різні потоки будуть виконувати операцію додавання елементів масиву.

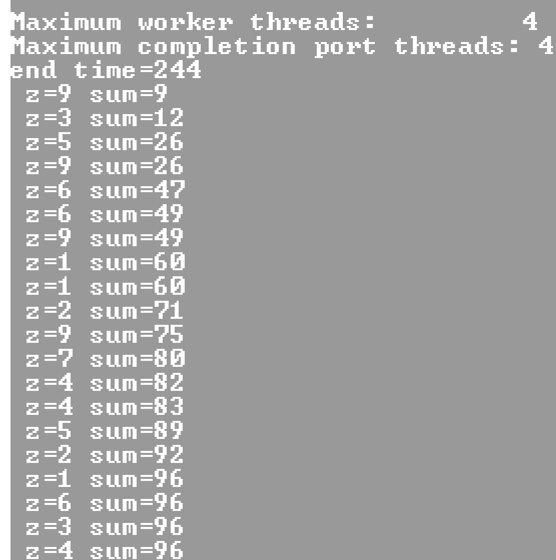



Рисунок 2.7 – Результат виконання програми 2.7

При кожному виклику методу *Function* як аргумент передається один елемент масиву для чергового потоку. Варто звернути увагу на дві особливості роботи потоків в пулі. По-перше, порядок формування проміжної суми (змінна *sum*) не завжди збігається з черговістю

надходження елементів масиву (змінна $z$), що свідчить про відсутність пріоритетів для окремих потоків пулу. По-друге, за наявності $m$ потоків в пулі ми отримуємо кінцевий результат за ($m$-1) тактів, тобто раніше, що підтверджує ефективність розпаралелювання обчислень.

Пул потоків розрахований на максимально ефективну роботу окремих потоків. Пул підтримує паралельну роботу такої кількості потоків, яка максимально можлива в цій системі (за замовчуванням 25). Якщо кількість потоків буде перевищувати цю цифру, то всі додаткові потоки автоматично організовуються в чергу очікування. Всі потоки пулу є фоновими, і вони знищуються автоматично після завершення основного потоку програми.

***Контрольні запитання***

1. В чому переваги і недоліки багатопотокових програм? В яких випадках варто створювати додаткові потоки у програмах?
2. Як створюються потоки в операційних системах Windows?
3. Які пріоритети можуть мати потоки та як їх можна змінювати?
4. В яких станах можуть знаходитись потоки?
5. В чому полягає різниця між основними і фоновими потоками?
6. Для вирішення яких проблем використовуються механізми синхронізації потоків?
7. В чому полягає різниця між використанням об’єкта класу Lock та використанням класу Monitor?
8. Як використовуються асинхронні делегати в багатопотоковому програмуванні?
9. В чому особливість пулу потоків і коли їх доцільно використовувати?

# 3 Багатопотокове програмування мовою C# за допомогою бібліотеки *TPL*

## 3.1 Нові засоби розпаралелювання в мові C#

Для реалізації паралельної обробки даних в багатопроцесорних машинах і в багатоядерних процесорах фірма Microsoft розробила програмні засоби під загальною назвою *Parallel Extensions for the .NET* (інша назва – *Parallel FX Library*, *PFX*) (рис. 3.1). *PFX* ввійшла до складу .NET 4.0 та Visual Studio 2008/2010, підтримується мовою програмування C# в версіях 4.0 і вище [12].

*PFX* забезпечує паралелізм на низькому рівні та паралелізм на високому рівні.

Паралелізм на низькому рівні забезпечується створенням окремих потоків (клас *Thread*), пулу потоків (клас *ThreadPool*), паралельних колекцій (*Concurent Collections*), примітивів *Spinning*, паралельних синтаксичних конструкцій (*Slim Signaling Constructs, Lazy Initialization Types*) і класу *Task* бібліотеки *TPL* (*Task Parallel Library*).

Паралелізм на високому рівні забезпечується за допомогою *PLINQ* (*Parallel Language-Integrated Query*) і класу *Parallel* бібліотеки *TPL.* Високий рівень програмування означає, що необхідно лише оголосити те, що має бути виконано паралельно, без деталей реалізації.

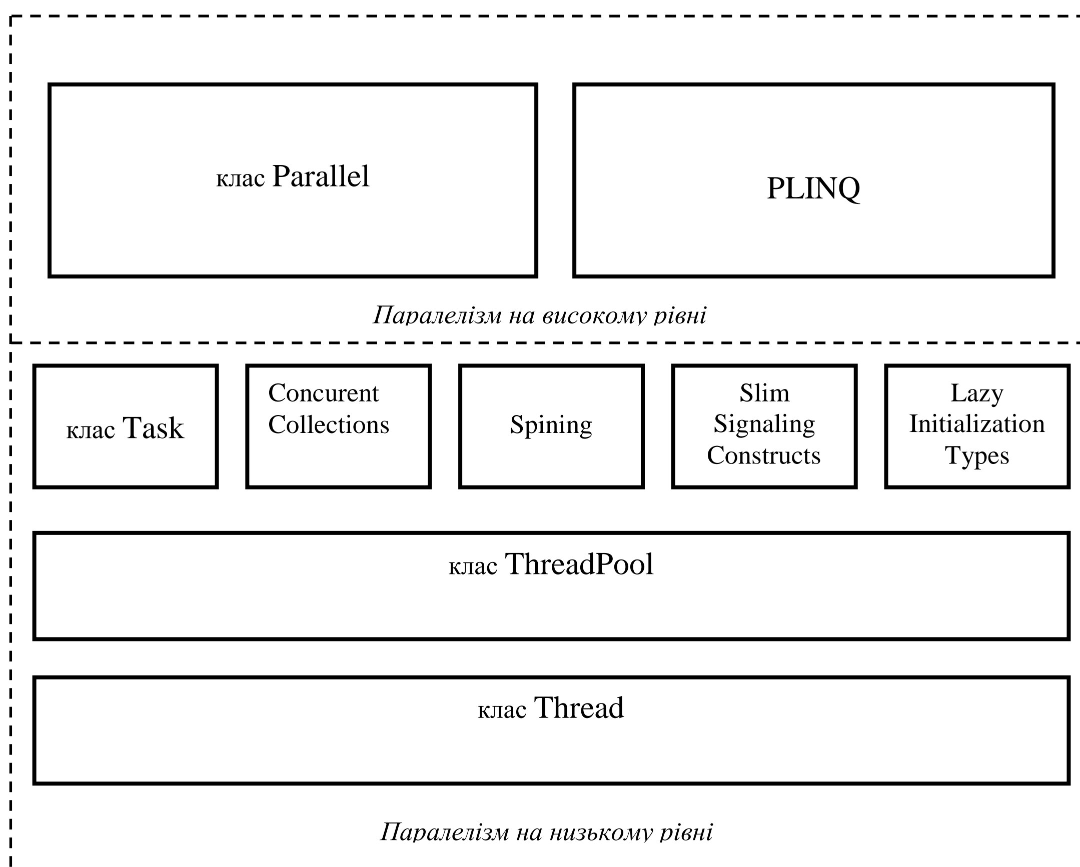



Рисунок 3.1 – Структура *Microsoft Parallel Extensions for the .NET*

Бібліотека *TPL* забезпечує паралелізм за даними і за задачами, а *PLINQ* – розпаралелювання запитів до даних.

В платформі .NET 4.5 (мова C# версії 4.5) над бібліотекою TPL введена бібліотека ще більшого рівня – бібліотека *TPL Dataflow (TDF).* Ця бібліотека використовує задачі, потоково-безпечні колекції та інші можливості, які подано в .NET 4 для додання підтримки паралельної обробки потоків даних. Суть бібліотеки зводиться до того, щоб, об'єднучи різноманітні блоки, організувати різні ланцюги обробки даних. При цьому обробка даних може відбуватись як синхронно, так і асинхронно.

### 3.2 Багатопотокове програмування з використанням класу *Task*

#### 3.2.1 Створення задачі з використанням класу *Task*

В основу TPL покладено поняття задачі – невеликого та відносно незалежного фрагмента коду, який може бути виконано паралельно з іншими фрагментами програми. Створюється задача за допомогою класу *Task*, визначеного в просторі імен *System.Threading.Tasks.* Цей клас відрізняється від раніше розглянутого класу *Thread* тим, що він є абстракцією, яка являє собою асинхронну операцію. А в класі *Thread* створюється потік виконання, і він не рівнозначний об'єкту класу *Task*.

Але відповідність екземпляра об'єкта класу *Task* і потоку виконання не обов'язково має бути взаємно однозначною. Крім того, виконанням задач керує планувальник задач, який працює з пулом потоків. Це, наприклад, означає, що кілька задач можуть використовувати один і той самий потік.

Всіма задачами і потоками керує спеціальний планувальник (*Task Manager*), який підтримує глобальну чергу задач і розподіляє їх для виконання по потоках. *Task Manager* працює з пулом потоків, число яких за замовчуванням встановлюється таким, що відповідає кількості обчислювальних ядер в системі. Пул потоків організований у вигляді двосторонньої черги для кожного процесора (ядра). За наявності в пулі потоків один з них вибираєтся для виконання з кінця черги (*bottom*). Після завершення одного потоку наступний потік вибирається з пулу за принципом LIFO. Якщо в черзі немає вільних потоків, тоді довільно вибирається інший процесор і з протилежної сторони черги (*top*) цього процесора вибирається вільний потік для виконання. Таким чином, кілька задач можуть використовувати один і той самий потік.

Створити нову задачу у вигляді об'єкта класу *Task* і почати його виконання можна  різними способами.

Розглянемо перший спосіб. Спочатку створимо об'єкт типу *Task* за допомогою конструктора і запустимо його, викликавши метод *Start().* Для цієї мети в класі *Task* визначено декілька конструкторів. Виберемо наступний конструктор:

*public Task(Action дія),*

де *дія* – точка входу у фрагмент програми, яка позначає задачу,
*Action* – делегат, який визначено в просторі імен *System*.

Форма делегата *Action*, який ми будемо використовувати, виглядає так:

```
public delegate void Action()
```

Таким чином, точкою входу має бути метод, який не приймає ніяких параметрів і не повертає жодних значень (як буде далі показано, делегату *Action* можна передати аргумент).

Як тільки задача буде створена, її можна запустити на виконання, викликавши метод *Start()*. Нижче наведено одну з його форм:

```
public void Start()
```

Після виклику метода *Start()* планувальник задач планує виконання задачі. В наведеній нижче програмі все вищесказане демонструється на прикладі. В цій програмі окрема задача створюється на основі методу *MyTask()*. Після того як розпочне виконуватись метод *Main(),* задача фактично створюється і запускається на виконання. Обидва методи *MyTask()* і *Main()* виконуються паралельно.

**Приклад 3.1**. Приклад програми з використанням класу *Task.*

```
// Створити і запустити задачу на виконання
using System;
using System.Threading;
using System.Threading.Tasks;
class DemoTask
 {
// Метод, що виконується як задача
static void MyTask()
{
           Console.WriteLine("MyTask() запущено");
          for (int count = 0; count < 10; count++)
          {
 Thread.Sleep(500);
 Console.WriteLine ("В MyTask() результат:  " + count);
          }
          Console.WriteLine("MyTask завершено");
}

static void  Main()
 {
        Console.WriteLine("Запуск основного потоку");
        //Створити об'єкт задачі
        Task tsk = new Task(MyTask);
        //Запустити задачу на виконання
        tsk.Start();
```

```
        //метод Main() активний до завершення методу MyTask().
        for (int i = 0; i < 50; i++)
        {
        Console.Write(".");
        Thread.Sleep(100);
        }
        Console.WriteLine("Завершення основного потоку");
        Console.ReadLine();
        }
    }
}
```

Нижче наведено результат виконання цієї програми. (У вас він може дещо відрізнятись залежно від завантаження задач, операційної системи та інших факторів).

```
        Запуск основного потоку
        MyTask() запущено
.....В MyTask() результат:  0
 ....В MyTask() результат:  1
.....В MyTask() результат:  2
.....В MyTask() результат:  3
.....В MyTask() результат:  4
.....В MyTask() результат:  5
.....В MyTask() результат:  6
.....В MyTask() результат:  7
.....В MyTask() результат:  8
.....В MyTask() результат:  9
        MyTask завершено
        ..Завершення основного потоку
```

За замовчуванням задача виконується у фоновому потоці. Таким чином, після завершення потоку, який був створений, завершується і сама задача. Саме тому в наведеній тут програмі метод *Thread.Sleep()* використано для збереження активним основного потоку до тих пір, поки не завершиться виконання методу *MyTask()*. Як і варто було чекати, організувати очікування завершення задачі можна і більш досконалими способами, що і буде показано далі.

### 3.2.2 Застосування ідентифікатора задачі

Кожна задача під час створення отримує цілочисловий ідентифікатор, який унікально визначає її в домені додатку. Доступ до нього можна отримати за допомогою властивості *Task.Id*.

Цей ідентифікатор корисний для налагоджування багатопотокових програм, зокрема, за допомогою вікон *Parallel Tasks* і *Parallel Stacks*.

На відміну від класу *Thread* в класі *Task* відсутня властивість *Name* для збереження імені задачі. Але замість цього в ньому є властивість *Id* для збереження ідентифікатора задачі, за яким можна розпізнавати задачі. Властивість *Id* доступна лише для читання і відноситься до типу *int.* Вона оголошується таким чином:

```
public int Id
{
    get;
}
```

Значення ідентифікаторів унікальні, але неупорядковані. Тому один ідентифікатор задачі може з'явитись перед іншим, хоча він може і не мати менше значення.

Тому при кожному запуску програми задача може мати різні ідентифікатори. Ідентифікатор задачі, яка виконується в поточний момент, можна виявити за допомогою властивості *CurrentId.* Ця властивість доступна тільки для читання, відноситься до типу *static* і оголошується таким чином:

```
public static Nullable<int> CurrentID
{
    get;
}
```

Вона повертає або задачу, яка в цей момент виконується, або ж пусте значення, якщо викликаний програмний код не є задачею.

В наведеному нижче прикладі програми створюються дві задачі й показується, яка з них виконується.

**Приклад 3.2**. Демонстрація використання властивості *CurrentId.*

```
using System;
using System.Threading;
using System.Threading.Tasks;
class DemoTask
{
// Метод, який виконується як задача.
 static void MyTask()
        {
        Console.WriteLine("MyTask() #" + Task.CurrentId + " запущено");
        for(int count = 0; count < 3; count++)
        {
            Thread.Sleep(400);
        Console.WriteLine("В  MyTask() #" + Task.CurrentId + ",   результат:
        " + count );
        }
```

```
        Console.WriteLine("MyTask #" + Task.CurrentId + " завершено");
        }
  static void Main()
         {
            Console.WriteLine("Запуск основного потоку");
            // Створити об'єкти двох задач.
            Task tsk1 = new Task(MyTask);
            Task tsk2 = new Task(MyTask);
            tsk1.Start();
            tsk2.Start();
            Console.WriteLine(Task.CurrentId);
            for (int i = 0; i < 30; i++)
            {
               Console.Write("."); Thread.Sleep(100);
            }
            Console.WriteLine("Завершення основного потоку");
            Console.ReadLine();
         }
      }
   }
```

Нижче наведено початковий фрагмент виконання цієї програми.

```
            Запуск основного потоку
            MyTask() запущено
      .....В MyTask() #2  результат:  0
         В MyTask() #1  результат:  0
      .....В MyTask() #1  результат:  1
         В MyTask() #2  результат:  1
      .....В MyTask() #2  результат:  2
         В MyTask() #1  результат:  2
```

Обидві задачі використовують свої ідентифікатори. Черговість появи ідентифікаторів довільний.

### 3.2.3 Використання класу TaskFactory для запуска задачі

В класі *TaskFactory* надаються різноманітні методи, які спрощують створення задач і керування ними. За замовчуванням об'єкт класу *TaskFactory* може бути отриманий із властивості *Factory*, доступної тільки для читання в класі *Task*. Використовуючи цю властивість, можна викликати будь-які методи класу *TaskFactory*.

Наприклад, в наведених раніше програмах задачу можна розпочинати виконувати відразу після її створення, викликавши метод *StartNew()*,

Метод *StartNew()* існує в багатьох формах. Найпростіша форма його оголошення:

*public Task StartNew(Action action)*,

де *action* – точка входу в задачу, що виконується.

Спочатку в методі *StartNew()* автоматично створюється екземпляр об'єкта типу *Task* для дії, яка визначається параметром *action*, а потім планується запуск задачі на виконання. Отже, необхідність у виклику метода *Start()* тепер відпадає.

Наприклад, наступний виклик методу *StartNew()* в раніше наведених програмах приведе до створення і запуску задачі *tsk1* однією дією:

*Task tsk = Task.Factory.StartNew(MyTask);*

Після цього оператора відразу починає виконуватись метод *MyTask().* Таким чином, метод *StartNew()* буде більш ефективним в тих випадках, коли задача створюється і без затримок запускається на виконання. Тому саме такий підхід і буде застосовуватись в наступних прикладах програм.

### 3.2.4 Очікування завершення задач

Для коректної роботи з різними задачами важливо знати час завершення задач. В найпростішому варіанті можна використати в головному потоці метод *Thread.Sleep().* Однак такий спосіб є дуже неефективним через можливі великі втрати часу.

Якщо наперед не відомо тривалість виконання задачі, то доцільно скористатись спеціальними методами очікування, які передбачені в класі *Task.*

Найпростішим з них є метод *Wait()*, який призупиняє виконання основного потоку до тих пір, поки не завершиться задача, що була ним викликана.

Розглянемо варіант програми з попереднього прикладу, коли спочатку повністю виконується перша задача, а потім друга задача.

**Приклад 3.3**. Демонстрація черговості виконання задач.

```
using System;
using System.Threading;
using System.Threading.Tasks;
namespace Project_4
{
   class DemoTask
   {
      // Метод, який виконується як задача.
      static void MyTask()
      {
         for (int count = 0; count < 3; count++)
         {
            Thread.Sleep(400);
            Console.WriteLine("В  MyTask() #" + Task.CurrentId + ",
результат: " + count);
```

```
        }
        Console.WriteLine("MyTask №" + Task.CurrentId + " завершено");
    }
    static void Main()
    {
        Console.WriteLine("Запуск основного потоку");
        // Створити об'єкти двох задач.
        Task tsk1 = new Task(MyTask);
        Task tsk2 = new Task(MyTask);
        tsk1.Start();
        tsk1.Wait();
        tsk2.Start();
        tsk2.Wait();
        Console.WriteLine(Task.CurrentId);
        for (int i = 0; i < 30; i++)
        {
            Console.Write("."); Thread.Sleep(100);
        }

        Console.WriteLine("Завершення основного потоку");
        Console.ReadLine();
    }
  }
}
```

Нижче наведено результат виконання цієї програми. Як видно із наведеного прикладу спочатку виконуються перша задача, потім друга задача і останнім – основний потік.

```
Запуск основного потоку
MyTask() запущено
   В MyTask() #1  результат:   0
   В MyTask() #1  результат:   1
   В MyTask() #1  результат:   2
         MyTask  #1  завершено
   В MyTask() #2  результат:   0
   В MyTask() #2  результат:   1
   В MyTask() #2  результат:   2
         MyTask  #2  завершено
   ………………………… Завершення основного потоку
```

Можливо також організувати очікування завершення групи задач. Наприклад, послідовність викликів

```
task1.Wait();
```

*task2.Wait();*
можна замінити на один
*Task.WaitAll(task1, task2);*
*Dispose().*

### 3.2.5 Повернення значень із задачі

На практиці часто виникає потреба передавати дані в задачу та повертати результат її виконання. Для того, щоб повернути результат із задачі, достатньо створити цю задачу з використанням узагальненої форми *Task<TResult>* класу *Task.* Нижче наведено два конструктори цієї форми класу *Task:*

*public Task (Func<TResult> функція)*

*public Task (Func<Object, TResult> функція, Object стан),*

де *функція* означає той делегат, що виконується, його тип має бути *Func.*

Тип *Func* використовується саме в тих випадках, коли задача повертає результат. В першому конструкторі створюється задача без аргументів, а в другому конструкторі – задача приймає аргумент типу *Object*, який передається як *стан*.

Є також інші варіанти методу *StartNew()*, які доступні в узагальненій формі класу *TaskFactory<TResult>* і підтримують повернення результату із задачі. Далі наведено ті варіанти цього методу, які використовуються паралельно з вищенаведеними конструкторами класу *Task.*

*public Task<TResult> StartNew(Func<TResult> функція)*

*public Task<TResult> StartNew(Func<Object, TResult> функція,,*
*Object стан)*

В будь-якому випадку значення, що повертає задача, отримується із властивості *TResult* в класі *Task,* яке визначається таким чином:

*public TResult Result { get; internal set; }*

Аксесор *set* є внутрішнім для даної властивості і тому часто виявляється доступним у зовнішньому коді лише для читання. Тому задача блокує код, що її викликає, до тих пір, поки результат не буде обчислено.

## 3.3 Багатопотокове програмування мовою C# з використанням класу *Parallel*

Розпаралелювання за даними і за задачами (функціями) на високому рівні в бібліотеці TPL забезпечується за допомогою класу *Parallel*.

Цей клас є статичним і використовує три методи:

- метод *For()*, який здійснює розпаралелювання за даними циклу *for*;
- метод *ForEach()*, який здійснює розпаралелювання за даними циклу *foreach*;
- метод *Invoke()*, який здійснює паралельне виконання двох і більше традиційних методів.

### 3.3.1 Розпаралелювання задач методом *Invoke()*

Метод *Parallel.Invoke()* дозволяє виконати паралельно декілька традиційних методів і дочекатись їх виконання. Як аргументами цього методу можна вказати або самі традиційні методи *Func1, Func2. …*:

```
public static void Invoke (Func1, Func2. … );
```

або масив делегатів типу *Action:*

```
public static void Invoke (params Action[] actions);
```

Приклад роботи методу *Invoke()* наведено нижче.

**Приклад 3.4**. Приклад програми з використанням методу *Invoke()*

```
static void Func1()
    {
      Console.WriteLine("Hello");
    }
    static void Func2()
    {
      Console.WriteLine("World!");
    }
    static void Main(string[ ] args)
    {
      Parallel.Invoke(Func1, Func2);
      Console.ReadLine();
  }
```

Такий же результат можна отримати за допомогою делегатів:

```
Action[] actions = new Action[4];
actions[0] = new Action(() => Console.WriteLine("Hello"));
actions[1] = new Action(() => Console.WriteLine("World!"));
Parallel.Invoke(actions);
```

Для порівняння аналогічний результат можна отримати при розпаралелюванні на низькому рівні за допомогою класу *Task:*

```
Task task1 = Task.Factory.StartNew(() => Func1());
Task task2 = Task.Factory.StartNew(() => Func2());
 Task.WaitAll(task1, task2);
```

При використанні методу *Invoke()* відпадає необхідність в організації очікування завершення роботи задач, які розпаралелюються, за допомогою спеціальних методів *Wait()* чи *WaitAll()*.

Варто також відзначити, що немає гарантії параллельного виконання задач (методів), які вказані як аргументи *Invoke()*, хоча це передбачається за наявності кількох процесорів (ядер). Крім того, неможливо вказати порядок виконання цих задач (методів) і він не завжди може збігатись з послідовністю аргументів у списку.

Однак, незважаючи на ці зауваження, метод *Invoke()* рекомендується для застосування в алгоритмах типу «поділяй і владарюй» (швидке сортування, обробка графів).

### 3.3.2 Розпаралелювання даних методом *For()*

Метод *For()* існує в кількох формах. Найпростішою з них є така:

```
public static ParallelLoopResult
        For (int  begin, int end, Action<int> body),
```

де *begin* – початкове значення циклу,
*end* – значення на одиницю більше кінцевого значення циклу.

На кожному кроці циклу змінна керування циклом збільшується на одиницю. Фрагмент коду, який циклічно виконується, передається через параметр *body*. Цей метод має бути сумісним з делегатом *Action*, що оголошується таким чином:

```
public delegate void Action<in T>(T obj)
```

Для методу *For()* узагальнений параметр *T* має бути типу *int*. Значення, яке передається через параметр *obj*, має бути наступним значенням змінної керування циклом. А метод, що передається через параметр *body*, може бути поіменованим або анонімним. *Метод For()* повертає екземпляр об'єкта типу *ParallelLoopResult*, що описує стан завершення циклу. Для простих циклів цим значенням можна знехтувати.

Метод *Parallel.For()* відрізняється від традиційного оператора *for* тим, що всі ітерації можуть виконуватись в окремих потоках. Завдяки цьому можна досягти підвищення продуктивності роботи. Однак, ефект від паралелізму може бути суттєвим лише при обробці достатньо великих масивів даних. Для малих обсягів даних введення паралельної обробки може навіть збільшити час виконання циклу.

Наприклад, якщо деяка функція *Function(i)* повторюється *n* разів згідно з послідовним циклом

```
for (int i = 0; i < n; i++)
Function(i);
```

то паралельна версія цього циклу буде такою:

*Parallel.For (0, n, i => Function(i) );*

або простіше:

*Parallel.For (0, n, Function);*

Розглянемо програму з демонстрацією застосування методу *For()* на практиці.

**Приклад 3.5**. Приклад програми з використанням методу *For()*.

```
using System;
using System.Threading.Tasks;
using System.Threading;

namespace Task_1
{
   class DemoTask
   {
      static int[ ] data; static int[ ] data2;
      // Виконання в паралельному циклі арифметичних операцій
      static void MyTransform(int i)
      {
         if (i < 100) data2[i] = 0;
         if (i >= 100 & i < 300) data2[i] = data[i] * 2;
         if (i >= 300 & i < 500) data2[i] = data[i] * 3;
         if (i >= 500) data2[i] = 1;
      }
      static void Main()
      {
         Console.WriteLine("Запуск основного потоку");
         data = new int[1000]; data2 = new int[1000];
         // Ініціалізація даних в звичайному циклі for.
         for (int i = 0; i < data.Length; i++)
                      data[i] = i;
         // Виклик функції для організації паралельного циклу
         Parallel.For(0, data.Length, MyTransform);
         for (int i = 0; i < data2.Length; i++)
            Console.WriteLine("Data2 " + data2[i] + " i="+i);
         Console.WriteLine("Завершення основного потоку");
         Console.ReadLine();
      }
   }
}
```

В наведеній вище програмі заданий масив *data* спочатку ініціалізується за допомогою традиційного циклу *for*, а далі метод *Parallel.For()* запускає функцію *MyTransform(int i)* для паралельного виконання різних

перетворень над різними частинами масиву *data.* Завдяки великому обсягу початкових даних можна сподіватись на економію часу при виконанні цієї програми на багатоядерному комп'ютері.

### 3.3.3 Розпаралелювання даних методом *ForEach()*

Метод *ForEach()* використовується як паралельний варіант традиційного циклу *foreach*. Існують декілька форм методу *ForEach().*Найпростішою з них є така:

```
public static ParallelLoopResult
        ForEach<TSource>(IEnumerable<TSource> source,
                Action<TSource> body)
```

де *source* – колекція даних, що обробляється в циклі,

*body* – метод, який буде виконуватись на кожному кроці циклу,

*IEnumerable* – інтерфейс.

Аналогічно методу *For()* паралельне виконання циклу методом *ForEach()* можна зупинити, викликавши метод *Break()* для екземпляра об'єкта типу *ParallelLoopState*, який передається через параметр *body*, за умови, що використовується наведена нижче форма методу *ForEach():*

```
public static ParallelLoopResult
        ForEach<TSource>(IEnumerable<TSource> source,
                Actior<TSource, ParallelLoopState> body).
```

Наприклад, якщо деяка функція *Function(i)* виконується згідно з послідовним циклом *foreach*

```
foreach (char  i  in "Hello, world")
Function(i) ;
```

тоді паралельна версія цього циклу буде такою*:*

```
Parallel.ForEach ("Hello, world", Function);
```

Розглянемо програму з демонстрацією застосування методу *ForEach()* на практиці.

**Приклад 3.6**. Приклад програми з використанням методу *ForEach()*

```
using System;
using System.Threading.Tasks;

 class DemoParallelForWithLoopResult
{
 static int[ ] data;

// Метод, який є основою паралельного циклу:
// змінній  v  передається значення елемента масиву
// даних, а не індекс цього елемента.
```

```
static void DisplayData(int v, ParallelLoopState pls)
{
 // Перервати цикл при виявленні від'ємного  значення .
            if(v < 0) pls.Break();

 Console.WriteLine("Значення: " + v);
 }

static void Main()
{
 Console.WriteLine("Основний потік запущено");
 data = new int[10000000];

// Ініціалізація даних в звичайному циклі  for.
 for (int i=0; i < data.Length; i++)
                    data[i] = i;
// Помістити від'ємне значення в масив data,
   data[100000] = -10;

 // Використати паралельний цикл на основі методу ForEach(),
// для відображення даних на екрані.

ParallelLoopResult loopResult = Parallel.ForEach(data,DisplayData);

// Перевірка завершення циклу
  if  (!loopResult.IsCompleted)
 Console.WriteLine("\n Цикл завершився достроково, тому  " +
            " що  виявлено від'ємне значення \n" +
            "на кроці циклу номер " +
            loopResult.LowestBreakIteration + ".\n");

  Console.WriteLine("Основний потік завершено");
}
 }
```

Як і в попередньому прикладі, спочатку заданий масив *data* ініціалізується за допомогою традиційного циклу *for.* Головна відмінність останньої програми полягає в тому, що метод, який виконується на кожному кроці циклу, просто виводить на консоль значення із масиву. Як правило, метод *WriteLine()* в паралельному циклі не використовується, тому що введення–виведення на консоль здійснюється настільки повільно, що цикл виявиться повністю «прив'язаним» до введення–виведення. Але в цьому прикладі метод *WriteLine()* використовується винятково з метою демонстрації можливостей методу *ForEach().* При виявленні від'ємного

значення виконання циклу переривається викликом методу *Break()*. Незважаючи на те, що метод *Break()* викликається в одній задачі, друга задача може продовжувати виконуватись протягом кількох кроків циклу до того моменту, коли вона буде зупинена.

### *Контрольні запитання*

1. Які програмні засоби розробила фірма Microsoft для реалізації паралельної обробки даних?
2. Дайте означення задачі в бібліотеці TPL.
3. Як створюються об'єкти классу *Task*?
4. Яка відмінність між класами *Thread* і *Task?*
5. Як запустити на виконання задачу за допомогою класу *TaskFactory*?
6. Як можна призупинити виконання потоків?
7. Як передати дані в задачу та отримати результат її виконання?
8. Які способи декомпозиції обчислень використовуються при розпаралелюванні задач за допомогою класу Parallel?

# 4 Багатопотокове програмування мовою C++ за допомогою стандарту OpenMP

## 4.1 Принципи паралельної обробки з використанням OpenMP

Початковий стандарт OpenMP був сформульований в 1997 році для підтримки паралельного програмування мовою Фортран. Подальшою розробкою стандарту займається некомерційна організація OpenMP ARB (Architecture Review Board), до якої увійшли представники найбільших компаній-розробників мікропроцесорних архітектур та програмного забезпечення. [5, 13]. Сучасна версія 4.5 2015 року підтримує також мови C і C++.

OpenMP реалізований як високорівнева надбудова над вказаними мовами програмування, тобто як бібліотека, яка містить різні функції та директиви для розпаралелювання обчислень.

Кожна бібліотека паралельної обробки орієнтована на конкретну архітектуру апаратних засобів. OpenMP розроблена для багатопроцесорних обчислювальних систем зі спільною пам'яттю. Такі системи складаються з кількох однорідних процесорів і масиву спільної пам'яті, до якої вони підключені за допомогою загальної шини або комутатора (див. рис. 1.1, а). В моделі зі спільною пам'яттю (*shared memory*) адресний простір всіх процесорів є єдиним. Всі процесори працюють під керуванням єдиної операційної системи. Багатопроцесорні системи подібного типу зазвичай іменуються симетричними мультипроцесорами (*symmetric multiprocessors, SMP*). Найпоширенішим прикладом SMP-систем є багатоядерні процесори.

Як правило, для розпаралелювання наявних послідовних програм більшість відомих бібліотек потребують суттєвої переробки початкового коду. Такий підхід значною мірою ускладнює значне поширення паралельних обчислень.

Зовсім інший підхід прийнято в OpenMP. Початковий текст програми залишається незмінним, а до нього лише додаються директиви паралельної обробки. Це дозволяє, по-перше, поступово вводити паралелізм, і, по-друге, програма залишається роботоздатною також і для послідовної обробки (додані директиви ігноруються послідовним компілятором). Таким чином, немає необхідності підтримувати послідовну та паралельну версії програми.

Зараз засоби OpenMP підтримують більшість компіляторів мов C/C++ (в подальшому будемо говорити тільки про ці мови програмування), їх необхідно лише активізувати. Наприклад, в пакеті Microsoft Visual Studio достатньо виконати такі дії: у властивостях поточного проекту послідовно вибрати пункти *Configuration Properties, C/C++, Language* і потім в полі *OpenMP Support* вказати «*Yes*»).

Робота OpenMP-програми розпочинається з єдиного потоку – головного (*master*) і далі складається з послідовних та паралельних

фрагментів (регіонів), які чергуються (рис. 4.1). При вході в паралельний регіон основний потік створює групи потоків (включно й з основним потоком). В кінці паралельного регіону групи потоків зупиняються, а виконання основного потоку продовжується. В паралельний регіон можуть вкладатись інші паралельні регіони, в яких кожний потік початкового регіону стає основним для своєї групи потоків. Вкладені регіони можуть, і собі, містити регіони більш глибокого рівня вкладеності.

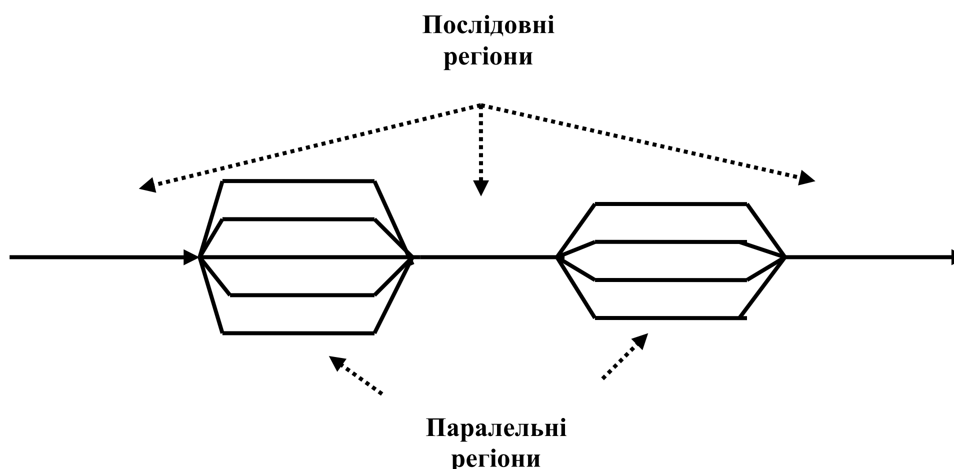



Рисунок 4.1 – структура багатопотокової OpenMP-програми

### 4.2 Основні синтаксичні конструкції OpenMP

В OpenMP можна виділити три основних синтаксичних конструкції:
– директиви компілятору,
– бібліотечні функції,
– змінні середовища.

Директиви слугують для явного вказання компілятору на виконання певних дій, які пов’язані з паралельною обробкою. В програмі мовою C директиви розпочинаються зі слів *#pragma omp* і мають такий формат:

***#pragma omp*** *<directive-name> [опция[[,] опция]...]*

Об’єктом дії більшості директив є один оператор або блок, перед яким розташована директива в початковому тексті програми. В OpenMP такі оператори або блоки називаються асоційованими з директивою. Необхідно, щоб в асоційованому блоці була одна точка входу на початку та одна точка виходу в кінці. Блоки виділяються фігурними дужками.

Порядок опцій в опису директиви несуттєвий, в одній директиві більшість опцій може зустрічатись декілька разів. Після кількох опцій може знаходитись список змінних.

Всі директиви OpenMP можна розділити на 3 категорії: визначення паралельного регіону, розподіл роботи, синхронізація. Кожна директива може мати декілька додаткових атрибутів – опцій (*clause*). Окремо

специфікуються опції для призначення класів змінних, які можуть бути атрибутами різних директив.

Функції OpenMP слугують, в основному, для зміни та отримання параметрів середовища. Крім цього, OpenMP містить API-функції для підтримки деяких типів синхронізації. Щоб задіяти ці функції бібліотеки OpenMP, в програму необхідно ввести заголовочний файл *omp.h*. Якщо в програмі використовуються тільки OpenMP-директиви *pragma*, вводити цей файл немає потреби.

### 4.3 Директиви паралельної обробки в OpenMP

Найбільш важлива та поширена директива – ***parallel***. Вона створює паралельний регіон для наступного за ним структурованого блока:

***#pragma omp parallel*** *[параметр[ [,] параметр]...]*
*{ <блок_програми> }*

Ця директива повідомляє компілятору, що структурований блок коду має бути виконаний паралельно, в кількох потоках. Правила виконання окремих команд блока залежать від параметрів директиви.

Наведемо перелік параметрів директиви *parallel*:

– ***if*** *([ parallel : ] scalar-expression)*
– ***num_threads****(integer-expression)*
– ***default****(private | firstprivate | shared | none)*
– ***private****(list)*
– ***firstprivate****(list)*
– ***shared****(list)*
– ***copyin****(list)*
– ***reduction****(operator: var)*
– ***procbind****(master | close | spread)*

Розглянемо коротко призначення основних параметрів цієї директиви.

Параметр ***if*** задає умову виконання паралельного фрагмента (якщо скалярний вираз параметра ***if*** хибний, то блок директиви *parallel* виконується як звичайний послідовний код). Така умова буде корисною в тому випадку, коли наперед не відома складність (кількість ітерацій) *блока програми* директиви.

Параметр ***num_threads****(integer-expression)* задає кількість потоків, які будуть виконуватись в паралельній частині програми.

Як вже відзначалось раніше, потоки паралельної програми виконуються в спільному адресному просторі і, як результат, всі дані (змінні) є загальнодоступними для всіх потоків, які паралельно виконуються. Однак, в деяких випадках необхідно явно вказати атрибут доступності відповідних змінних. Для цього слугують параметри ***private****(list)*, ***firstprivate****(list)*, ***shared****(list)* і ***copyin****(list)*, які містять списки змінних. Параметр ***private****(list)* означає, що кожна змінна списку обов'язково має мати захищену копію в кожному потоці. Параметр

***firstprivate****(list)* задає список змінних, для яких породжується локальна копія в кожному потоці; початкові значення локальних змінних будуть встановлені в ті значення, які існували в головному потоці до моменту створення локальних копій. Параметр ***shared****(list)* задає список змінних, спільних для всіх потоків. Якщо якимсь змінним явно не призначено атрибут доступності, то всі вони отримують атрибут, вказаний в параметрі ***default****(private | firstprivate | shared | none)*. Параметр ***copyin****(список)* задає список змінних, які оголошені як *threadprivate*, і при вході в паралельний регіон ініціалізуються значеннями відповідних змінних в потоці *master*.

Значення решти параметрів буде пояснено далі.

Розглянемо на прикладі використання директивы *parallel*. Як ілюстрацію використаємо такий фрагмент програми (приклад 4.1).

**Приклад 4.1**. Використання директиви *parallel*.

```
#include <stdio.h>
#include <omp.h>
void main(int argc, char *argv[])
{
        printf("serial region 1\n");
       #pragma omp parallel num_threads(3)
        {
              printf("parallel  region \n");
        }
       printf("serial region 2\n"); }
}
```

Як результат виконання програми на екран комп'ютера буде виведено такий текст:

```
serial region 1
 parallel  region
 parallel  region
 parallel  region
serial region  2
```

В *блоці програми* директиви *parallel* можуть використовуватись інші директиви розпаралелювання обчислень.

Серед різних методів розпаралелювання обчислень найчастіше використовується розпаралелювання за даними, в рамках якого забезпечується одночасне (паралельне) виконання одних і тих самих обчислювальних дій над різними наборами даних. Для реалізації такого підходу в OpenMP використовуються директива розпаралелювання циклів:

***#pragma omp  for*** *[<параметр> ... ]*

***<цикл_for>***

Після цієї директиви ітерації циклу розподіляються між потоками і, як результат, можуть бути виконані паралельно. Для коректного розпаралелювання ітерацій є ряд обмежень для цієї директиви.

По-перше, розпаралелювання можливе тільки за умови, що між ітераціями циклу немає інформаційної залежності. Цикл має мати єдину точку входу та єдину точку виходу. Команди *goto* і *break* можуть виконувати переходи тільки всередині циклу. Команда *exit* завершує всі ітерації. Змінна циклу має бути цілим зі знаком.

Параметрами директиви ***for*** можуть бути також декілька нових, деякі з них розглянемо пізніше. Тут відзначимо лише параметр ***lastprivate****(list),* за допомогою якого забезпечується запам'ятовування вказаного списку локальних змінних після завершення директиви ***for*** (значення локальних змінних копіюються з потоку, що виконав останню ітерацію). Як ілюстрацію розглянемо фрагмент програми (приклад 4.2).

**Приклад 4.2.** Використання директиви *parallel*.

```
int i, j, sum;   float mas[M][N];
sum = 0;
 #pragma omp parallel shared(mas) private(i, j, sum)
 {
 #pragma omp for
       for (i=0; i < M; i++)
          for (j=0; j < N; j++)
             sum=sum+mas[i][j];
    printf ("Сума елементів матриці %f\n", sum);
}
```

Варто відзначити, що коли в блоці директиви *parallel* немає нічого, крім директиви *for*, тоді обидві директиви можна об'єднати в одну

***#pragma omp parallel for*** *[<параметр> ... ]*

Крім розпаралелювання за даними в OpenMP використовується також розпаралелювання за функціями (підзадачами). Для підтримки такого способу організації паралельних обчислень в OpenMP для паралельного фрагмента програми, який створюється за допомогою директиви *parallel*, можна виділити паралельні програмні фрагменти за допомогою такої директиви:

```
 #pragma omp sections [<параметр> ...]
{
 #pragma omp section    <блок_програми>
     . . . . . . . . . . .
  #pragma omp section    <блок_програми> }
```

За допомогою директиви *sections* виділяється програмний код, який далі буде розділений на секції, що паралельно виконуються. Директиви

*section* визначають програмні фрагменти, які можуть бути виконані паралельно. Залежно від кількості потоків та кількості визначених секцій, кожний потік може виконати одну або кілька секцій (однак, за малої кількості секцій деякі потоки можуть виявитись і без секцій, тобто незавантаженими). Згідно зі стандартом, порядок виконання програмних секцій не визначений, вони можуть бути виконані потоками в довільному порядку.

Як приклад розглянемо фрагмент програми (приклад 4.3).

**Приклад 4.3**. Використання паралельних секцій

```
total = 0;
 #pragma omp parallel shared(a,b) private(i,j)
{
#pragma omp sections
          for (i=0; i < M; i++)
          {     sum = 0;     for (j=i; j < N; j++)
                          sum=sum+mas[i][j];
         printf ("Сума дорівнює %f\n",i,sum);
                 total = total + sum;
          }
#pragma  omp section
         for (i=0; i < M; i++)
         {    for (j=i; j < N; j++)
                b[i][j] = mas[i][j];
          }
}
printf ("Сума елементів матриці дорівнює %f\n",total);
```

На практиці часто виникає потреба у здійсненні операцій з локальними змінними з різних потоків. Наприклад, після завершення паралельних потоків необхідно зібрати разом ці локальні змінні в головному потоці і виконати з ними якусь колективну операцію. Виконати таку задачу можна або зі збереженням значень локальних змінних у спільних змінних (за допомогою параметра *lastprivate*), або виконати задану колективну операцію за допомогою параметра *reduction* директиви *for*:

***#pragma omp parallel for reduction****(operator: var)*

Зустрівши параметр *reduction* компілятор створює захищені (*private*) копії змінної *sum* для кожного потоку, а коли потоки завершаються, він виконує колективну операцію (оператор редукції *operator)* і поміщає результат у змінну *sum*. Копія змінної *sum* в кожному потоці ініціалізується відповідним початковим значенням (табл. 4.1). Всі змінні редукції мають бути скалярними (наприклад, *int, long* и *float*).

Таблица 4.1 – Оператори редукції

<table>
<tr><th>Оператор редукції</th><th>Ініціалізувальне значення</th></tr>
<tr><td>+ (додавання)</td><td>0</td></tr>
<tr><td>- (віднімання)</td><td>0</td></tr>
<tr><td>* (множення)</td><td>1</td></tr>
<tr><td>& (порозрядне І)</td><td>~0</td></tr>
<tr><td>| (порозрядне АБО)</td><td>0</td></tr>
<tr><td>^ (порозрядна нерівнозначність)</td><td>0</td></tr>
<tr><td>&& (умовне І)</td><td>1</td></tr>
<tr><td>| | (умовне АБО)</td><td>0</td></tr>
</table>

Як приклад розглянемо фрагмент програми (приклад 4.4).

**Приклад 4.4**. Використання редукції.

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
 {   for (i=0; i < M; i++)
          for (j=0; j < N; j++)
          sum =sum+mas[i][j];
}
```

При виконанні паралельних фрагментів може виявитись необхідним реалізувати частину програмного коду тільки одним потоком (наприклад, відкриття файлу). Таку можливість в OpenMP забезпечують директиви *single* і *master*.

Директива ***single*** визначає блок паралельного фрагмента, який має бути виконаний тільки одним потоком; всі інші потоки очікують завершення виконання даного блока (якщо не вказано параметр *nowait*). Її формат такий:

```
#pragma omp single [<параметр> ...]
{
    <блок_програми>
}
```

Як параметри директиви можуть виконуватись:

– ***private*** *(list),*
– ***firstprivate*** *(list),*
– ***copyprivate*** *(list),*
– ***nowait.***

Параметр ***copyprivate*** забезпечує копіювання змінних, які вказані у списку *list*, після виконання блока директиви ***single*** в локальні змінні решти потоків.

Директива ***master*** визначає блок паралельного фрагмента, який потрібно виконувати лише головним потоком, всі інші потоки пропускають цей фрагмент коду. Формат директиви:

```
#pragma omp master
{
    <блок_програми>
}
```

### 4.4 Бібліотечні функції OpenMP для паралельних обчислень

Як додаток до директив в OpenMP міститься також багато корисних функцій. Нагадаємо, що для їх використання необхідно підключити заголовочний файл *omp.h.*

Спочатку розглянемо функції для отримання довідкової інформації:

– отримання максимально можливої кількості потоків:
*int omp_get_max_threads();*

– отримання фактичної кількості потоків в паралельній області програми: *int omp_get_num_threads();*

– отримання номера потоку:
*int omp_get_thread_num();*

– отримання числа процесорів чи ядер, які доступні програмі:
*int omp_get_num_procs().*

Для установлення заданої кількості *NumThreads* потоків в паралельній області програм слугує функція

*int omp_set_num_threads(int NumThreads).*

Задати кількість потоків можна також і за допомогою змінної середовища:

*OMP_NUM_THREADS NumThreads.*

При використанні кількох способів задання найбільший пріоритет має параметр *num_threads* директиви *parallel*, далі йде функція бібліотеки і потім змінної середовища.

В OpenMP є можливість визначити з великою точністю тривалості виконання фрагментів програми. Для цього передбачено такі функції для роботи із системним таймером.

Функція *omp_get_wtime()* повертає астрономічний час в секундах (дійсне число подвійної точності), який пройшов з деякого моменту в минулому. Функція *omp_get_wtick()* повертає розрізнюваність таймера в частках секунди. Цей час можна розглядати як міру точності таймера.

Якщо деякий фрагмент програми «оточити» викликами цієї функції, тоді різниця значень, що повертаються, покаже час роботи цього фрагмента. Таймери різних потоків можуть бути несинхронізовані і видавати різні значення.

Наступний приклад ілюструє використання функцій *omp_get_wtime()* та *omp_get_wtick()* для роботи з таймерами в OpenMP.

**Приклад 4.5**. Визначення часу виконання програмного коду

```
#include <stdio.h>
#include <conio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
        int i, j, sum;   float mas[200][100];  sum = 0;
        double start_time, end_time, tick;
                start_time = omp_get_wtime();
#pragma omp parallel shared(mas) private(i, j, sum)
 {
        #pragma omp for
      for (i=0; i < 200; i++)
        for (j=0; j < 100; j++)
            sum=sum+mas[i][j];
        end_time = omp_get_wtime();
}
        tick = omp_get_wtick();
         printf("Час обчислень %lf\n", end_time-start_time);
        printf("Точність таймера %lf\n", tick);
        getch();
        return 0;
}
```

Корисною бібліотечною функцією є функція

*void omp_set_nested (int nested),*

яка забезпечує використання вкладених паралельних фрагментів програми.

Виклик цієї функції має здійснюватись з послідовної частини програми. Функція дозволяє (*nested=true*) або забороняє (*nested=false*) режим підтримки вкладених паралельних фрагментів. За замовчуванням для виконання вкладених паралельних фрагментів створюється стільки ж потоків, як і для паралельних фрагментів верхнього рівня.

Для керування режимом вкладених паралельних фрагментів можна скористатись підтримкою змінної середовища OMP_NESTED.

При використанні кількох способів задання найбільший пріоритет має функція бібліотеки, далі – змінна середовища. Для отримання стану режиму підтримки вкладених паралельних фрагментів можна скористатись функцією *int omp_get_nested (void);*

## 4.5 Планування паралельної обробки

Для досягнення високої ефективності паралельних обчислень необхідно оптимально розподілити навантаження між всіма ядрами процесора. При різному обсязі обчислень в різних ітераціях циклу деякі потоки можуть завершуватись значно раніше інших, що призведе до простою процесорних ресурсів і втрати продуктивності. Під час написання складної програми важко знайти такі варіанти ітерацій, які б мали однаковий час виконання. Щоб спростити перерозподіл навантаження між ядрами процесора, OpenMP пропонує чотири способи планування розподілу ітерацій циклу між потоками:

- статичне планування (*static*),
- динамічне планування (*dynamic*),
- кероване планування (*quided*),
- планування часу виконання (*runtime*).

Інформація про спосіб планування передається в директиві *for* за допомогою параметра *schedule*:

*#pragma omp for schedule(<спосіб планування>[, розмір фрагмента])*

При статичному плануванні ітерації циклу поділяються на фрагменти однакового розміру ще до виконання програми. Якщо розмір фрагмента не вказано, ітерації розподіляються якомога рівномірніше, по одному фрагменту на потік. Такий спосіб планування може підійти для програм простої структури, він також приймається за замовчуванням, якщо явно не задано який-небудь інший спосіб.

При динамічному плануванні створюється загальна черга для видачі кожному потокові блока ітерацій циклу, який дорівнює розміру фрагмента, як тільки потік стає доступним. Після завершення своєї роботи потік бере з початку черги наступний блок ітерацій. При такому плануванні всі потоки будуть завантажені роботою, але в цьому випадку необхідні додаткові витрати часу на організацію такої роботи.

Кероване планування нагадує динамічне, але спочатку розмір фрагмента призначається великим, а потім він експоненціально зменшується (але не менше заданого розміру) згідно з формулою:

$$\pi_k = \left\lceil \frac{\beta_k}{n} \right\rceil,$$

де $n$ – кількість потоків,

$\pi_k$ – розмір $k$-го фрагмента,

$\beta_k$ – кількість ітерацій циклу, які залишились незапланованими при обчисленні розміру $k$-го фрагмента.

Основна ідея керованого планування полягає в поступовому зменшенні розміру фрагмента, а коли значення $\pi_k$ стає дуже малим, тоді воно округлюється до величини *s*, яка вказується в операторі

*schedule(quided,s).*

Планування часу виконання не є якимось особливим способом. В цьому випадку користувач має можливість вибору способу планування із раніше розглянутих через задання його за допомогою змінної середовища OMP_SCHEDULE.

В результаті розпаралелювання циклу порядок виконання ітерацій не фіксований: залежно від стану середовища виконання черговість виконання ітерацій може змінюватись. Якщо ж для деяких дій в циклі необхідно зберегти початковий порядок обчислень, який відповідає послідовному виконанню ітерацій в послідовній програмі, тоді бажаного результату можна досягти за допомогою директиви *ordered* (при цьому для директиви *for* має бути вказаний параметр *ordered*).

Пояснимо сказане на прикладі нашої навчальної задачі. Для наведеного раніше варіанта програми друк сум елементів рядків матриці буде відбуватись в деякому довільному порядку; за необхідності друку в порядку розташування рядків необхідно провести таку зміну програмного коду (приклад 4.7).

**Приклад 4.6**. Використання директиви *ordered.*

```
#pragma  omp parallel for shared(a) private(i,j,sum) schedule
(dynamic, 4) ordered
{   for (i=0; i < n; i++)
 {    sum = 0;
       for (j=i; j < n; j++)
       sum=sum+mas[i][j];
#pragma  omp ordered
printf ("Сума елементів рядка %d равна %f\n",i,sum);
}
```

Відзначимо додатково, що директива *ordered* може бути використана в тілі циклу лише один раз і її дія обмежена тільки всередині області між двома конструкціями *ordered.*

## 4.6 Директиви і функції синхронізації обчислень

Для досягнення оптимальної продуктивності багатопотокових програм важливе значення мають різні засоби синхронизації обчислень: бар'єри, критичні секції, замки та інші директиви.

За замовчуванням, кінець кожного паралельного регіону являє собою своєрідний бар'єр, який потоки можуть подолати лише всі разом. За допомогою параметра *nowait* в директивах *parallel, for, section* і *single* бар'єр можна відмінити, і тоді потоки, які завершили роботу раніше, можуть приступити до виконання наступної задачі. Бар'єр можна знову відновити за допомогою директиви

```
#pragma omp barrier
```

Популярним засобом синхронізації, і не тільки в OpenMP, є критичні секції (області). Під критичною секцією розуміють фрагмент програми, який може бути виконаний тільки одним потоком.

Якщо критична секція вже виконується яким-небудь потоком, тоді всі інші потоки, що виконали директиву

*#pragma omp critical [(<ім'я_критичної_секції>)]*

для секції з таким іменем, будуть заблоковані, поки потік, який першим туди зайшов, не закінчить виконання даної критичної секції. Як тільки потік, що виконується, вийде з критичної секції, тоді випадковим чином вибирається один із заблокованих на вході потоків і далі критична секція продовжить виконуватись з новим потоком. А інші заблоковані потоки продовжать знаходитись в стані очікування.

Все непоіменовані критичні секції умовно асоціюються з одним і тим самим іменем. Всі критичні секції з одним іменем розглядаються єдиною секцією, навіть якщо знаходяться в різних паралельних фрагментах. Побічні входи та виходи з критичної секції заборонені.

Приклад 8 ілюструє застосування директиви *critical*. Змінна *n* оголошена за межами паралельної секції, тому за замовчуванням є спільною. Критична секція дозволяє розмежувати доступ до змінної *n*. Кожний потік по черзі присвоїть *n* свій номер і потім надрукує отримане значення.

**Приклад 4.7**. Директива *critical* мовою C.

```
#pragma omp parallel
{
        #pragma omp critical
        {   int     n=omp_get_thread_num();
          printf("Потік %d\n", n);
        }
}
```

Практично у всіх паралельних системах обробки даних існує такий об'єкт синхронізації, як семафор. Класичний семафор регулює доступ кількох потоків, які хочуть отримати доступ до спільного ресурсу. Такий доступ зможе отримати лише наперед визначена кількість потоків, в найпростішому випадку – один потік (такий семафор називають двійковим).

Аналогом семафора в OpenMP є замок (*lock*). Замок може знаходитись в одному из трьох станів: неініціалізованому, розблокованому або заблокованому.

Спочатку замок має бути ініціалізованим, після чого він переходить в розблокований стан, відкриваючи доступ до спільного ресурсу (наприклад, до спільної змінної). Потік, який першим захопив такий замок, переводить його в заблокований стан і забороняє доступ до нього іншим потокам.

Після завершення роботи з ресурсом потік знову переводить замок в розблокований стан, відкриваючи доступ до ресурсу іншим потокам.

Для роботи з замками є декілька директив, які містять спеціальний тип даних *omp_lock_t:*

– ініціалізувати замок: *void omp_init_lock(omp_lock_t *lock);*

– встановити замок: *void omp_set_lock (omp_lock_t &lock);*

– звільнити замок: *void omp_unset_lock (omp_lock_t &lock);*

– перевірити замок: *int omp_test_lock (omp_lock_t &lock);*

– переведення замка в ініціалізований стан:
*void omp_destroy_lock(omp_lock_t &lock*).

Якщо в результаті перевірки замок був вільним, то він закривається і функція *omp_unset_lock* повертає значення *true*; якщо ж замок зайнятий, тоді потік, що його захопив, не блокується, і функція повертає значення *false*.

Є два типи замків: прості і множинні. Множинний замок може багаторазово захоплюватись одним потоком перед його звільненням, а простий замок може бути захоплений тільки один раз. Для множинного замка вводиться поняття коефіцієнта захопленості (*nesting count*). Спочатку він дорівнює нулю, при кожному наступному захопленні збільшується на одиницю, а при кожному звільненні зменшується на одиницю. Множинний замок вважається розблокованим, якщо його коефіцієнт захопленості дорівнює нулю.

**Контрольні запитання**

1. Для яких комп’ютерних архітектур найбільш придатна технологія OpenMP?
2. Які мови програмування підтримуються в OpenMP?
3. Назвіть основні директиви паралельної обробки в OpenMP.
4. Як зберегти результати обчислень в паралельному регіоні після його завершення?
5. Які директиви здійснюють розпаралелювання за даними та за функціями (підзадачами) в OpenMP?
6. Як здійснюється розпаралелювання обчислень при використанні операції редукції (*reduction*)?
7. Яким чином можна визначити час виконання програмного коду?
8. Дайте порівняльну характеристику чотирьох способів планування розподілу ітерацій циклу між потоками.
9. Які директиви і функції використовуються в OpenMP для синхронізації обчислень?
10. Як здійснюється синхронізація обчислень при використанні замків (*lock*)?

## 5 Паралельне програмування з використанням стандарту MPI

### 5.1 Принципи паралельної обробки на основі стандарту MPI

Стандарт паралельної обробки даних MPI був створений в 1994 році групою Message Passing Interface Forum. Його подальшим розвитком став стандарт MPI-2, який з'явився в 1997 році [14, 15].

Зазначимо, що внаслідок своєї складності стандарти MPI та MPI-2 в повному обсязі не були реалізовані у жодній системі. Для теоретичного вивчення та практичного застосування рекомендується використовувати реалізації MPICH та MPICH-2 (відповідно стандартів MPI та MPI-2), які створені в Арагонській національній лабораторії (США). В подальшому будемо використовувати саме їх для операційних систем Windows.

Основна мета створення реалізації MPICH-2 – ефективна підтримка багатоплатформності, враховуючи кластерні системи від супер-комп'ютерних (Blue Gene) до загальнодоступних (настільні системи, багатоядерні процесори), а також комутаційні мережі (Ethernet 10 Гбіт/с, InfiniBand, Myrinet, Quadrics).

Стандарти MPI та MPI-2 як програмні засоби реалізовано у вигляді бібліотек MPI, які містять більше 300 різноманітних програмних функцій. Найголовніші з них будуть далі детально проаналізовані.

Корисно порівняти бібліотеку MPI з раніше розглянутою бібліотекою OpenMP. До їхніх спільних рис можна віднести безкоштовність та використання однакових мов програмування (C, C++ та Фортран). З іншого боку, MPI та OpenMP мають суттєві відмінності.

В рамках різноманітних паралельних архітектур можливі два основних варіанти паралельних обчислень залежно від способу використання основної пам'яті.

Як вже відзначалось в попередній темі, бібліотека OpenMP орієнтована на системи зі спільною пам'яттю (*shared memory*), за допомогою якої здійснюється обмін даними між окремими процесорами (ядрами центрального процесора).

Можлива також паралельна архітектура з розподіленою (індивідуальною) пам'яттю (*distributed memory*). В таких системах, як правило, кластерних системах, процесори працюють незалежно один від одного. Тому безпосередній обмін даними як між ними, так і через основну пам'ять неможливий.

Для організації зв'язку між окремими процесами, які виконуються на різних ядрах кластера, необхідний спеціальний механізм обміну повідомленнями (інформацією) між ними.

З цією метою розроблено стандартизований інтерфейс обміну повідомленнями (*Message Passing Interface, MPI*) між різними процесорами під час їхньої паралельної роботи. Посередниками в такому обміні повідомленнями служать різні *комунікатори*.

Технологія обчислень OpenMP орієнтована на паралельну архітектуру *SIMD* за класифікацією Фліна, в той час, як MPI основана на архітектурі *MIMD*. В OpenMP паралелізм реалізовується за рахунок роботи багатьох потоків одного процесу, а в MPI – за рахунок роботи багатьох незалежних процесів.

Варто відзначити, що багато сучасних реалізацій MPI підтримують роботу також і з потоками.

З позицій складності практичної реалізації стандарт MPI безумовно складніший. Якщо паралельні директиви OpenMP просто додаються до вже налагодженої послідовної програми, то для програмування MPI потрібно написати і налагодити принципово іншу програму.

Окрім складності програмування до недоліків MPI можна також віднести те, що поки не існує реалізацій MPI, які повною мірою забезпечують суміщення обмінів з обчисленнями.

## 5.2 Організація паралельних обчислень в стандарті MPI-2

MPI-програма являє собою набір процесів різних процесорів, які одночасно виконуються своєму власному адресному просторі. Паралельні процеси можуть виконуватись також в режимі розподілу часу, що дає змогу виконати початкову перевірку правильності паралельної програми і на одному процесорі.

Стандарти MPI та MPI-2 підтримують паралельну архітектуру *MIMD*, яка передбачає об’єднання процесів з різними текстами програм. Однак писати та налагоджувати такі програми дуже складно, тому на практиці частіше використовується паралельна архітектура *SIMD*, в рамках якої використовується одна й та ж програма однією з мов паралельного програмування (C, C++, Fortran) для різних паралельних процесів.

Таким чином, кожний процес паралельної програми породжується на основі копії одного програмного коду. Цей код подається у вигляді виконуваної програми (файла типу *.exe), який має бути доступним в момент запуску паралельної програми на всіх процесорах.

Процеси паралельної програми об’єднуються в *групи*. Якщо група містить *n* процесів, то процеси нумеруються всередині групи номерами, які є цілими числами від *0* до *n-1*. Номер процесу іменується *рангом* процесу.

Найважливішим поняттям в MPI є поняття *комунікатора* (*communicator*). Під комунікатором розуміють спеціальний службовий об’єкт, який об’єднує в своєму складі групу процесів і забезпечує передачу повідомлень між ними (рис. 5.1).

Таким чином, кожний процес має два основних атрибути: комунікатор групи та ранг процесу.

Один і той самий процес може належати різним групам і комунікаторам.

Основним способом взаємодії процесів між собою є обмін повідомленнями. *Повідомлення* – це набір даних деякого типу. Кожне повідомлення має декілька атрибутів: ранг процесу-відправника, ранг процесу-отримувача, ідентифікатор повідомлення та інші.

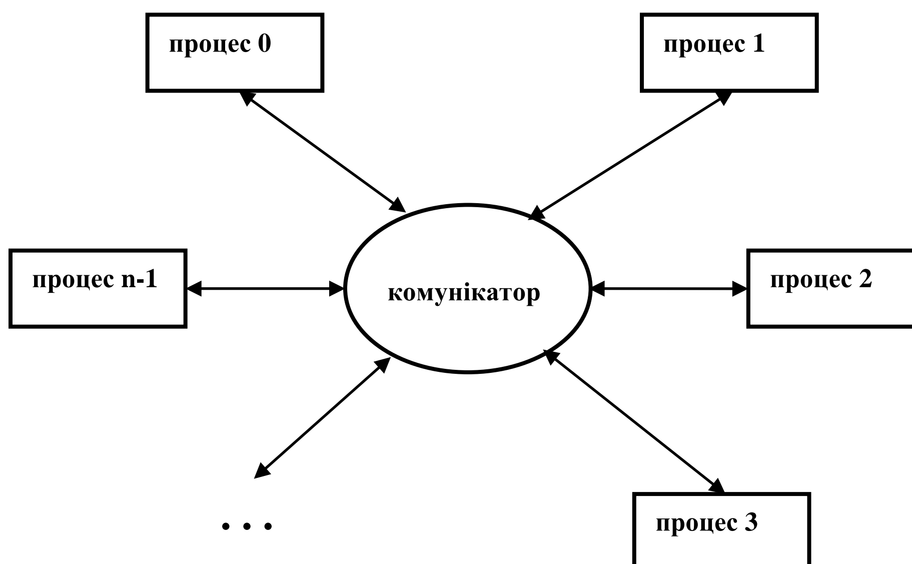



Рисунок 5.1 – Обмін повідомленнями між процесами в MPI

Обмін повідомленнями відбувається, коли частина адресного простору одного процесу скопійована в адресний простір іншого процесу. Ця операція спільна і можлива лише тоді, коли перший процес виконує операцію передачі повідомлення, а інший − операцію його отримання.

Існують парні та колективні операції передачі даних між процесами. Парні операції передачі даних виконуються тільки для двох процесів, що належать одному комунікатору. Колективні операції застосовуються одночасно для всіх процесів одного комунікатора.

За замовчуванням всі наявні в паралельній програмі процеси належать одній групі, для якої створюється локальний комунікатор (*intracommunicator*), з ідентифікатором MPI_COMM_WORLD. Цей комунікатор існує завжди. Також існують завжди комунікатор MPI_COMM_SELF з одним процесом та комунікатор MPI_COMM_NULL без жодного процесу.

За необхідності створюються нові групи та відповідні їм комунікатори.

В стандарті MPI-2 з’явилась можливість створювати глобальні комунікатори (*intercommunicator*), які забезпечують передачу даних між процесами з різних груп.

Розробка та виконання MPI-програми здійснюється за два етапи. Спочатку текст програми налагоджується в одному із відомих програмних середовищ розробки, наприклад, в Microsoft Visual Studio. Результатом першого етапу має бути виконуваний файл (*.exe) програми.

На другому етапі здійснюється запуск програми на виконання. Такий запуск програми (якій відповідає, наприклад, виконуваний файл *prog.exe*) згідно стандарту MPI-2 в ОС Windows відбувається таким чином.

1. Користувач за допомогою програми *mpiexec*, вказує ім’я виконуваного файла MPI-програми та необхідне число (наприклад, чотири) процесів:

*mpiexec -n 4 prog.exe* .

Крім цього, можна вказати ім’я користувача і пароль: процеси будуть запускатися від імені цього користувача.

2. Програма *mpiexec* передає відомості про запуск локальному менеджерові процесів ***smpd.exe***, у якого є список доступних процесорів.

3. Менеджер процесів звертається до процесорів по списку, передаючи запущеним на них менеджерам процесів вказівки щодо запуску MPI-програми.

4. Менеджери процесів запускають на процесорах декілька копій MPI-програми (можливо, по декілька копій на кожному процесорі), передаючи програмам необхідну інформацію для зв’язку між ними.

Отже, кількість процесів визначається в момент запуску паралельної програми засобами середовища виконання MPI-програм і під час обчислень не може змінюватись (в стандарті MPI-2 вже передбачено можливість динамічної зміни кількості процесів).

Можна відзначити особливості організації паралельних обчислень в стандарті MPI-2 для кластерних обчислювальних систем. MPI-програма не копіюється автоматично на всі вузли кластера. Менеджер процесів передає процесорам лише шлях до виконуваного файлу програми в тому ж форматі, як і в команді запуску програми на виконання. Тоді на кожному вузлі буде запущена на виконання своя копія MPI-програми.

Дуже зручно відразу розмістити виконуваний файл програми на спільному мережевому ресурсі, куди всі копії програми зможуть також записувати результати своєї роботи.

### 5.3 Бібліотека MPI

MPI – це не тільки стандарт обміну повідомленнями при паралельних обчисленнях. MPI – це також програмні засоби, що забезпечують можливість передачі повідомлень і при цьому повністю відповідають вимогам згаданого стандарту. Згідно зі стандартом ці програмні засоби мають бути організовані у вигляді бібліотеки програмних функцій (бібліотеки MPI) мовами програмування C, C++ та Фортран. Таким чином, бібліотека MPI буде вказувати на ту чи іншу програмну реалізацію стандарту MPI.

Бібліотека MPI містить близько 130 функцій, куди входять такі основні набори функцій:

– базові функції;
– функції для реалізації парних комунікаційних операцій;
– функції для реалізації колективних комунікаційних операцій;
– функції для роботи з групами процесів та комунікаторами;
– функції для роботи зі структурами даних;
– функції формування топології процесів.

Кожна з MPI функцій характеризується способом виконання і належить до таких типів.

1. *Локальна функція* – виконується всередині процесу, який її викликав. Її завершення не потребує комунікацій.
2. *Нелокальна функція* – для її завершення необхідно виконання MPI функції іншим процесом.
3. *Глобальна функція* – її мають виконати всі процеси групи. Невиконання цієї умови може призвести до зависання задачі.

4. *Блокувальна функція* – передбачає вихід з неї тільки після повного завершення операції, тобто процес блокується, поки операція не буде завершена. Для функції посилання повідомлень це означає, що дані, які мають далі посилатись, спочатку записуються і зберігаються в буфері. Для функції прийому повідомлень блокується виконання інших операцій, поки всі дані з буфера не будуть передані в адресний простір процесу, який приймає повідомлення.

5. *Неблокувальна функція* – передбачає суміщення операцій обміну з іншими операціями. Завершення неблокувальних операцій здійснюється спеціальними функціями.

Варто зазначити, що, незважаючи на велику кількість MPI функцій, для виконання більшості обмінів даних достатньо не більше двох десятків функцій (в мінімальному варіанті не більше шести), які далі будуть детально розглянуті.

При виконанні операцій передачі даних в функціях MPI необхідно вказувати їх тип. MPI містить великий набір базових типів даних, який часто збігається з подібними типами даних в алгоритмічних мовах C та Fortran. В таблиці 5.1 наведено відповідність в MPI типів стандартним типам мови C.

### 5.4 Найпростіша MPI-програма

Розглянемо базові MPI функції на прикладі відомої демонстраційної програми «*Hello world*», яка буде також видавати загальну кількість процесів та їх ранг.

**Приклад 5.1** Найпростіша MPI-програма

```
#include <stdio.h>
#include "mpi.h"
```

Таблиця 5.1 – Відповідність в MPI типів стандартним типам мови C

| Тип даних бібліотеки MPI | Тип мови C |
|---|---|
| *MPI_BYTE* | |
| *MPI_CHAR* | *signed char* |
| *MPI_DOUBLE* | *double* |
| *MPI_FLOAT* | *float* |
| *MPI_INT* | *int* |
| *MPI_LONG* | *long* |
| *MPI_LONG_DOUBLE* | *long double* |
| *MPI_PACKED* | |
| *MPI_SHORT* | *short* |
| *MPI_UNSIGNED_CHAR* | *unsigned char* |
| *MPI_UNSIGNED* | *unsigned int* |
| *MPI_UNSIGNED_LONG* | *unsigned long* |
| *MPI_UNSIGNED_SHORT* | *unsigned short* |

```
int main(int argc, char* argv[])
{
  int procs_rank, procs_count;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &procs_count);
  MPI_Comm_rank(MPI_COMM_WORLD, &procs_rank);
 printf ("\n Hello, World from process %3d of %3d",
procs_rank, procs_count);

  MPI_Finalize();
  return 0;
}
```

Розглянемо детально цей приклад. Спочатку підключається обов’язковий заголовочний файл бібліотеки «*mpi.h*», який містить визначення функцій, типів і констант MPI.

На початку кожної MPI-програми має бути функція *MPI_Init(),* яка здійснює ініціалізацію паралельної частини програми. Тому вона має бути попереду всіх інших функцій MPI і може викликатись тільки один раз.

Її аргументами є кількість аргументів командного рядка процесу і власне командний рядок процесу:

```
int  MPI_Init (int* argc,    char*** argv)
```

де *argc* – покажчик на кількість параметрів командного рядка;
*argv* – параметри командного рядка.

Функція *MPI_Comm_size()* повертає в змінній *procs_count* число запущених для цієї програми процесів. При цьому немає значення, яким способом користувач запускає ці процеси – все залежить від реалізації MPI. Передача повідомлень в цій програмі здійснюється за допомогою стандартного локального комунікатора *MPI_COMM_WORLD*. Оскільки всі процеси об'єднуються в одну групу, то можна сказати, що змінна *procs_count* повертає також розмір цієї групи.

За допомогою функції *MPI_Comm_rank()* кожний процес в групі визначає свій номер (ранг) і повертає його в змінній *procs_rank*. Кожний процес друкує свій ранг та загальну кількість запущених процесів, потім всі процеси виконують функцію *MPI_Finalize()*.

Ця функція має бути виконана кожним процесом MPI і призводить до ліквідації «середовища» MPI. Ніякі інші функції MPI не можуть бути викликані після неї (повторна ініціалізація за допомогою *MPI_Init()* також неможлива). Тому функція *MPI_Finalize()* завжди звершує паралельну частини MPI-програми.

Відзначимо, що для цієї програми неможливо наперед визначити порядок видачі повідомлень від різних процесів. Черговість появи повідомлень залежить від порядку завершення окремих процесів, а він буде різним при кожному запуску програми. Одне з можливих вирішень цієї проблеми – зібрати дані від усіх процесів на одному процесорі (вузлі кластера) і потім організувати їх виведення в бажаній послідовності.

До важливих базових функцій можна віднести функцію відліку часу (таймера):

*double MPI_Wtime(void)*

Функція повертає астрономічний час в секундах, який пройшов з деякого моменту часу в минулому (точки відліку). Гарантується, що ця точка відліку не буде змінена протягом існування процесу. Для хронометражу фрагмента програми виклик функції відбувається на початку та в кінці фрагмента і визначається різниця між показами таймера:

```
{
double starttime, endtime;
 starttime = MPI_Wtime();

            ... фрагмент програми...

 endtime = MPI_Wtime();
printf("Виконання зайняло %f секунд \n", endtime-starttime);
}
```

Такий спосіб вимірювання часу роботи програми не залежить від апаратної платформи, операційної системи та мови програмування.

## 5.5 Функції для реалізації парних комунікаційних операцій

Спочатку розглянемо способи обміну повідомленнями типу «*point-to-point communications*». Це означає, що існують дві точки взаємодії: процес-відправник та процес-отримувач повідомлення. Для таких обмінів використовується один комунікатор, відносно якого задаються ранги процесів.

Для відправки повідомлень від одного процесу до іншого використовується функція

*int MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);*

де *buf* – адреса початку буфера для зберігання даних, які пересилаються;
*count* – число елементів, які пересилаються;
*type* – MPI-тип елементів, які пересилаються;
*dest* – ранг процеса-отримувача повідомлень в групі, яка пов'язана з комунікатором *comm*;
*tag* – ідентифікатор повідомлення;
*comm* – комунікатор зв'язку.

Для прийому повідомлення використовується функція:

*int MPI_Recv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *status);*

де *buf* – адреса початку буфера для зберігання отриманих даних;
*count* – число отриманих елементів;
*type* – MPI-тип отриманих елементів;
*source* – ранг процесу, від якого надходить повідомлення;
*tag* – ідентифікатор отриманого повідомлення;
*comm* – комунікатор зв'язку;
*status* – результат завершення операції обміну.

Далі наведено приклад програми (приклад 5.2), яка описує поведінку двох взаємодіючих процесів, один з яких відсилає дані, а другий процес їх отримує та видає на друк.

**Приклад 5.2** MPI програма з парними комунікаційними операціями.

```
#include <stdio.h>
#include "mpi.h"
int main (int argc, char* argv[])
      {
   int rank;  MPI_Status st; char buf[64];
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0)
    {
      sprintf(buf, "Hello from process "0);
```

```
MPI_Send(buf, 64, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}
   else
{
    MPI_Recv(buf, 64, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &st);

     printf"(Process %d received %s \"n, rank, buf);
 };
 MPI_Finalize();
 return 0;
}
```

Виконання цієї програми приводить до такого результату:

*Process 1 received Hello from process 0*

Розглянемо детальніше текст програми. Спочатку відбувається ініціалізація MPI-бібліотеки і в змінній *rank* запам'ятовується ранг процесу. Далі в операторі *if*, перша вітка виконується процесом з рангом 0, а друга – процесом з рангом 1. В результаті вказаних дій процес з рангом 0 поміщає в буфер *buf* текст привітання, а процес з рангом 1 забирає з буфера це повідомлення та роздруковує його.

Розглянуті вище функції для парних комунікаційних операцій реалізують стандартний режим обміну з блокуванням. Можливі також і інші режими передачі даних.

### 5.6 Режими передачі даних в MPI

В MPI передбачені 4 режими передачі повідомлень:
- стандартний (*standard*),
- синхронний (*synchronous*),
- буферизований (*buffered*),
- режим передачі по готовності (*ready*).

В стандартному режимі виконання операція обміну має три етапи:
1. Сторона , яка відправляє, формує пакет повідомлення і передає його в буфер. На цьому функція відправки повідомлення закінчується;
2. Системні засоби сповіщають адресата про надходження повідомлення;
3. Приймальна сторона забирає повідомлення з буфера тоді, коли в неї виникає потреба в цих даних. Змістовна частина повідомлення поміщається в адресний простір відповідного процесу (параметр *buf*), а службова – в параметр *status*.

Синхронний режим передачі повідомлень полягає в тому, що завершення функції відправки повідомлення відбувається тільки після

отримання від процеса-отримувача підтвердження про початок прийому відправленого повідомлення. Відправлене повідомлення або повністю прийнято адресатом, або знаходиться в стані приймання.

Буферизований режим передачі повідомлень передбачає використання додаткових буферів (системних чи наданих користувачем) для копіювання в них відправлених повідомлень. Функція відправки повідомлення закінчується відразу після копіювання повідомлення в додатковий буфер.

Режим передачі повідомлень за їх готовності може бути використаний тільки за умови, що процес-отримувач виставив ознаку готовності прийому даних. Буфер повідомлення після завершення функції відправки повідомлення може бути повторно використаний.

Для кожного режиму передачі повідомлень в бібліотеці MPI передбачена своя функція для відправлення повідомлень (табл. 5.2).

Принцип формування імен функцій в таблиці 2 полягає в доданні до імен базових функцій операцій обміну різних префіксів:

– синхронний режим: префікс S,
– буферизований режим: префікс B,
– режим по готовності: префікс R,
– для неблокуючих операцій: префікс I.

Таблиця 5.2 – Функції парних операцій обміну для різних режимів

| **Режим передачі** | **Напрям передачі** | **Функція (з блокуванням)** | **Функція (без блокування)** |
|---|---|---|---|
| стандартний | відправка | *MPI_Send()* | *MPI_Isend()* |
| синхронний | відправка | *MPI_Ssend()* | *MPI_Issend()* |
| буферизований | відправка | *MPI_Bsend()* | *MPI_Ibsend()* |
| за готовності | відправка | *MPI_Rsend()* | *MPI_Irsend()* |
| | прийом | *MPI_Recv()* | *MPI_Irecv()* |

При операціях обміну даними в паралельних програмах часто виникає потреба відправити дані одним процесам і в той же час отримати повідомлення від інших процесів. Реалізація таких обмінів за допомогою звичайних парних операцій передачі даних може бути неефективною, крім того, можуть виникнути тупикові ситуації, коли два процеси починають передавати повідомлення один одному з використанням функцій обміну з блокуванням.

Ефективно і гарантовано здійснити обмін даними в таких умовах можна за допомогою спеціальних функцій, які забезпечують взаємний обмін даними між процесами.

Така функція суміщає одночасно відправку і прийом даних:

*int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,*
*int dest, int sendtag, void *recvbuf, int recvcount,*

*MPI_Datatype recvtype, int source, MPI_Datatype, recvtag,MPI_Comm comm, MPI_Status *status),*

де **sendbuf, sendcount, sendtype, dest, sendtag* – параметри повідомлення, що відправляється;

**recvbuf, recvcount, recvtype, source, recvtag* – параметри повідомлення, що приймається;

*comm* – комунікатор зв’язку;

**status* – результат завершення операції обміну.

В тих випадках, коли необхідно здійснити обмін даними одного типу із заміщенням старих даних новими, які приймаються, тоді зручно користуватись такою функцією:

*MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)*

В цій операції обміну дані із буфера *buf* заміщаються новими даними, які приймаються.

Відзначимо, що як адреси *source* і *dest* в операціях пересилання даних можна використати спеціальну адресу *MPI_PROC_NULL*. Комунікаційні операції з такою адресою нічого не виконують. Використання цієї адреси буває зручним замість використання логічних конструкцій для аналізу умов відправки або читання повідомлення.

## 5.7 Функції для реалізації колективних комунікаційних операцій

При виконанні паралельних обчислень часто виникає необхідність в одночасній передачі даних між багатьма процесами. Звичайно, такі операції можна виконати і за допомогою функцій парних передач, які були розглянуті раніше. Однак, такий підхід до реалізації паралелізму буде дуже неефективним і повільним.

Для досягнення високої ефективності при одночасному виконанні операцій передачі даних в бібліотеці MPI передбачено різноманітні функції колективної взаємодії, зокрема:

– розсилання даних від одного процесу всім іншим процесам групи;
– збирання даних від всіх процесів групи в один масив;
– суміщення колективних операцій передачі даних;
– виконання колективних обчислювальних операцій;
– синхронізація роботи всіх паралельних процесів.

Основні відмінності колективних операцій від парних операцій:

– в колективних операціях завжди беруть участь всі процеси групи зі спільним комунікатором;
– колективні операції не взаємодіють з парними операціями;

– колективні операції виконуються в режимі з блокуванням, тобто вихід з функції колективної операції відбувається після завершення всіх процесів групи;

– кількість отриманих даних має дорівнювати кількості відправлених даних;

– типи елементів відправлених та отриманих повідомлень мають збігатись.

Найпростішою функцією колективної взаємодії є широкомовне пересилання даних від головного процесу (процесу з рангом 0) всім іншим процесам групи зі спільним комунікатором. Є функція

*int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm),*

де *buffer* – адреса початку буфера для зберігання даних, які пересилаються,
*count* – число елементів, які пересилаються;
*datatype* – тип елементів, які пересилаються;
*root* – номер процеса-відправника;
*comm* – комунікатор.

Після завершення функції кожний процес зі спільним комунікатором *comm* отримає копію повідомлення від процесу-відправника *root*.

Всі наступні функції колективної взаємодії мають два варіанти:

– простий варіант, коли всі блоки повідомлення, яке відправляється, мають однакову довжину і займають суміжні області в адресному просторі процесів;

– векторний варіант, коли знімаються всі обмеження відносно довжини блоків повідомлення та їх розташування в адресному просторі процесів (векторні варіанти мають додатковий символ «v» в кінці імені функції).

Розглянемо приклад програми (приклад 5.3) підрахунку суми елементів одновимірного масиву (вектора) з використанням функції *MPI_Bcast()*.

**Приклад 5.3**. Програма підрахунку суми елементів масиву з використанням колективних комунікаційних операцій.

```
#include <stdio.h>
#include <conio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{        double x[100], TotalSum, ProcSum = 0.0;
          int  rank, size, N=10, k, i1, i2,i;
 MPI_Status Status;
                         // Ініціалізація
 MPI_Init(&argc,&argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 printf("Process %d numprocs %d \n", rank, size);
                    // Підготовка даних
if (rank == 0)
for (i = 0; i < N; i++)
       x[i]=i;
      // Розсилання даних на всі процеси
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
     // Обчислення на кожному з процесів часткової суми
     // елементів вектора x від  i1 до i2
k = N / size;
i1 = k * rank;
i2 = k * (rank + 1);
if ( rank == size-1 ) i2 = N;
for (i = i1; i < i2; i++)
  ProcSum  = ProcSum + x[i];
                 // Збирання часткових сум  на процесі з рангом 0
if (rank == 0)
{
  TotalSum = ProcSum;
  for (i=1; i < size; i++)
  {
       MPI_Recv(&ProcSum,1,MPI_DOUBLE,MPI_ANY_SOURCE,0,
                    MPI_COMM_WORLD, &Status);
    TotalSum = TotalSum + ProcSum;
  }
}
else            // Всі процеси відсилають свої часткові суми
MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
               // Виведення загальної суми вектора
if (rank == 0)
  printf("\nTotal Sum = %4.2f",TotalSum);
MPI_Finalize();
getch();
return 0;
}
```

Сімейство функцій розподілу блоків даних по всіх процесах групи складається з двох функцій: *MPI_Scatter()* та *MPI_Scatterv()*.

Функція *MPI_Scatter()* розбиває повідомлення буфера процеса-відправника *root* на рівні частини розміром *sendcount* і надсилає *i*-у частину в буфер процеса-приймача з номером *i* (зокрема й самому собі). Процес *root* використовує обидва буфери (відправки та прийому), тому в цій функції всі параметри є суттєвими. Решта процесів з групи з комунікатором *comm* є тільки отримувачами, тому для них параметр, які вказують на буфер відправки, несуттєвий. Формат зазначеної функції:

*int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm),*

де *sendbuf, sendcount, sendtype* – параметри повідомлення, яке передається;
*recvbuf*, *recvcount, recvtype* – параметри повідомлення, яке приймається;
*root* – номер процеса-відправника;
*comm* – комунікатор.

Функція *MPI_Scatterv()* є векторним варіантом функції *MPI_Scatter()*.

Сімейство функцій збору блоків даних від всіх процесів групи складається з чотирьох функцій: *MPI_Gather(), MPI_Allgather(), MPI_Gatherv(), MPI_Allgatherv()*.

Операція, яка виконується функцією *Scatter()* є оберненою до операції, яка виконується функцією *Gather()*.

Функція *MPI_Gather()* виконує збирання блоків даних, які надсилаються всіма процесами групи, в один масив процесу з номером *root*. Довжина блоків передбачається однаковою. Об'єднання відбувається в порядку збільшення номерів процесів-відправників. Тобто дані, які відправлені *i*-м процесом зі свого буфера *sendbuf*, поміщаються в *i*-ю порцію буфера *recvbuf* процесу *root*. Довжина масиву, в якому збираються дані, має бути достатньою для їх розміщення. Таким чином, буфер прийому використовується лише для процесу *root*, для решти процесів буфер *recvbuf* не використовується і є несуттєвим. Формат зазначеної функції:

*int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm),*

де *sendbuf, sendcount, sendtype* – параметри повідомлення, яке передається;
*recvbuf*, *recvcount, recvtype* – параметри повідомлення, яке приймається;
*root* – номер процеса, який здійснює збір даних;
*comm* – комунікатор.

Функція *MPI_Allgather()* виконується так же, як *MPI_Gather(),* але отримувачами є всі процеси групи. Дані, які послані *i*-м процесом зі свого буфера *sendbuf*, поміщаються в *i*-ю порцію буфера *recvbuf* кожного

процесу. Після завершення колективної операції вміст буферів прийому *recvbuf* у всіх процесів буде однаковим. Формат зазначеної функції:

*int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,*
*void* recvbuf, int recvcount, MPI_Datatype recvtype,*
*MPI_Comm comm),*

де *sendbuf, sendcount, sendtype* – параметри повідомлення, яке передається;
*recvbuf*, *recvcount, recvtype* – параметри повідомлення, яке приймається;
*comm* – комунікатор.

Функції *MPI_Gatherv()* та *MPI_Allgatherv()* є векторними варіантами функцій відповідно *MPI_Gather()* та *MPI_Allgather()*.

Функція *MPI_Alltoall()* суміщає в собі операції *Scatter()* та *Gather()* і є розширенням операції *Allgather()*, коли кожний процес надсилає різні дані різним адресатам. Процес *i* посилає *j*-й блок свого буфера *sendbuf* процесу *j*, який поміщає його в *i*-й блок свого буфера *recvbuf*. Кількість надісланих даних має дорівнювати кількості отриманих даних для кожної пари процесів. Формат зазначеної функції:

*int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype,*
*void* recvbuf, int recvcount, MPI_Datatype recvtype,*
*MPI_Comm comm),*

де *sendbuf, sendcount, sendtype* – параметри повідомлення, яке передається;
*recvbuf*, *recvcount, recvtype* – параметри повідомлення, яке приймається;
*comm* – комунікатор.

Функція *MPI_Alltoallv()* реалізовує векторний варіант операції *Alltoall()* і допускає передачу та прийом блоків різної довжини з більш гнучким розміщенням переданих і прийнятих даних.

## 5.8 Функції для реалізації колективних обчислювальних операцій

В паралельному програмуванні математичні операції над блоками даних, які розподілені між процесорами, називають глобальними операціями *редукції*. В загальному випадку операцією редукції називається операція, аргументом якої є вектор, а результатом – скалярна величина, отримана з використанням деякої математичної операції до всіх компонентів вектора. В табл. 5.3 подано математичні операції, які можуть бути виконані при таких розподілених обчисленнях.

Наприклад, якщо *n* компонентів одновимірного масиву (вектора) розподілені між *n* процесами деякої групи процесів, тоді застосування глобальної операції редукції SUM приведе до отримання одного числа, яке буде дорівнювати сумі компонентів цього масиву.

В MPI глобальні операції редукції подано в кількох варіантах:

– зі збереженням результата в адресному просторі одного процесу (*MPI_Reduce()*);

– зі збереженням результата в адресному просторі всіх процесів (*MPI_Allreduce()*);

Таблиця 5.3 – Математичні операції редукції

| **Математична операція** | **Позначення операції в MPI** |
|---|---|
| сума | *MPI_SUM* |
| добуток | *MPI_PROD* |
| мінімум | *MPI_MIN* |
| максимум | *MPI_MAX* |
| логічне І | *MPI_LAND* |
| логічне АБО | *MPI_LOR* |
| логічна нерівнозначність | *MPI_LXOR* |
| порозрядне І | *MPI_BAND* |
| порозрядне АБО | *MPI_BOR* |
| порозрядна нерівнозначність | *MPI_BXOR* |
| мінімальне значення і його індекс | *MPI_MINLOC* |
| максимальне значення і його індекс | *MPI_MAXLOC* |

– префіксна операція редукції, яка в результаті виконання операції повертає вектор. *i*-а компонента цього вектора є результатом редукції перших *i* компонент розподіленого вектора (*MPI_Scan()*);

– суміщена операція редукції і пересилання даних *Reduce/Scatter* (*MPI_Reduce_scatter()*).

Функція *MPI_Reduce()* виконується таким чином. Операція глобальної редукції, яка вказана параметром *op*, виконується над першими елементами вхідного буфера, і результат посилається в перший елемент буфера прийому процесу *root*. Далі те ж саме здійснюється для других елементів буфера і т. д. Формат зазначеної функції:

*int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm),*

де *sendbuf* – адреса початку вхідного буфера;

*recvbuf* – адреса початку буфера результату (використовується тільки в процесі *root,* який отримує кінцевий результат);

*count* – число елементів у вхідному буфері;

*datatype* – тип елементів у вхідному буфері;

*op* – операція, за якою виконується редукція;

*root* – ранг процесу, який отримує результат операції;

*comm* – комунікатор.

Операції *MAXLOC* та *MINLOC* виконуються над спеціальними парними типами, кожний елемент яких зберігає дві величини: значення, за якими йде пошук максимуму чи мінімуму, та індекс елемента. В MPI для мови С є 6 таких типів даних:

– *MPI_FLOAT_INT – float* та *int* ,

– *MPI_DOUBLE_INT – double* та *int* ,
– *MPI_LONG_INT – long* та *int* ,
– *MPI_2INT – int* та *int* ,
– *MPI_SHORT_INT – short* та *int* ,
– *MPI_LONG_DOUBLE_INT – long double* та *int.*

Функція *MPI_Allreduce()* зберігає результат редукції в адресному просторі всіх процесів, тому в списку параметрів функції відсутній ідентифікатор кореневого процесу *root*. Набір решти параметрів такий самий, як і в попередньої функції:

*int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)*

Функція *MPI_Reduce_scatter()* суміщає в собі операції редукції та розподілу результату по процесах. Формат зазначеної функції:

*MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)* ,

де *sendbuf* – адреса початку вхідного буфера;
*recvbuf* – адреса початку буфера прийому;
*revcount* – масив, в якому задаються розміри блоків, які посилаються процесам;
*datatype* – тип елементів у вхідному буфері;
*op* – операція, за якою виконується редукція;
*comm* – комунікатор.

Функція *MPI_Reduce_scatter()* відрізняється від *MPI_Allreduce()* тим, що результат операції розрізається на частини, які не перетинаються, за числом процесів в групі, *i*-а частина надсилається *i*-му процесу в його буфер прийому. Довжини цих частин задає третій параметр, який є масивом.

Функція *MPI_Scan()* виконує префіксну редукцію. Параметри такі ж, як в функції *MPI_Allreduce()*, але отримані кожним процесом результати відрізняються один від одного. Операція пересилає в буфер прийому *i*-го процесу редукцію значень з вхідних буферів процесів з номерами *0, ... i* включно. Формат зазначеної функції:

*int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)* ,

де *sendbuf* – адреса початку вхідного буфера;
*recvbuf* – адреса початку буфера прийому;
*count* – число елементів у вхідному буфері;
*datatype* – тип елементів у вхідному буфері;
*op* – операція, за якою виконується редукція;
*comm* – комунікатор.

В багатьох ситуаціях обчислення, які виконуються незалежно різними процесами, необхідно синхронізувати. Синхронізація процесів, тобто

одночасне досягнення різними процесами заданих етапів обчислень в MPI здійснюється за допомогою таких функцій.

Функція синхронізації процесів *MPI_Barrier()* блокує роботу процесу, який її викликав, до тих пір, поки всі інші процеси групи також не викличуть цю функцію. Завершення роботи цієї функції можливо тільки всіма процесами одночасно (всі процеси «долають бар'єр» одночасно). Використання бар'єра гарантує, що жоден із процесів не приступить до виконання наступного етапу, поки результат роботи попереднього не буде остаточно сформований. Формат зазначеної функції:

*int MPI_Barrier(MPI_Comm comm) ,*

де *comm* – комунікатор.

## 5.9 Функції для роботи з групами та комунікаторами

Групи процесів можуть бути створені тільки із вже існуючих груп. Як початкова може бути вибрана група, що вже пов'язана з комунікатором *MPI_COMM_WORLD*. Також іноді може бути корисним комунікатор *MPI_COMM_SELF*, який визначено для кожного процесу паралельної програми і містить тільки цей процес.

Для отримання групи, яка пов'язана з існуючим комунікатором, використовуєтьтся функція:

*int MPI_Comm_group(MPI_Comm comm, MPI_Group *group),*

де *comm* – комунікатор;

*group* – група, яка пов'язана з комунікатором.

Далі, на основі існуючих груп, можуть бути створені такі нові групи.

1. Створення нової групи *newgroup* із існуючої групи *oldgroup*, яка буде містити в собі *n* процесів – їх ранги перераховуються в масиві *ranks* функції

*int MPI_Group_incl(MPI_Group oldgroup, int n, int *ranks, MPI_Group *newgroup)*

2. Створення нової групи *newgroup* із існуючої групи *oldgroup*, яка буде містити в собі *n* процесів, чиї ранги не збігаються з рангами, які перераховуються в масиві *ranks* функції

*int MPI_Group_excl(MPI_Group oldgroup, int n, int *ranks, MPI_Group *newgroup)*

3. Для отримання нових груп над існуючими групами процесів можуть бути виконані операції об'єднання, перетину та різниці:

а) створення нової групи *newgroup* як об'єднання груп *group1* і *group2* за допомогою функції:

*int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)*

б) створення нової групи *newgroup* як перетину груп *group1* і *group2* за допомогою функції:

*int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)*

в) створення нової групи *newgroup* як різниці груп *group1* і *group2* за допомогою функції:

*int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)*

При конструюванні груп може виявитись корисною спеціальна пуста група *MPI_COMM_EMPTY*.

4. Декілька функцій MPI забезпечує отримання інформації про групу процесів:

а) про кількість процесів в групі:

*int MPI_Group_size(MPI_Group group, int *size),*

де *group* – група;
*size* – число процесів в групі;

б) отримання рангу поточного процесу в групі:

*int MPI_Group_rank(MPI_Group group, int *rank),*

де *group* – група;
*rank* – ранг процесу в групі.

5. Після завершення використання група може бути вилучена за допомогою функції:

*int MPI_Group_free(MPI_Group *group),*

де *group* – група, яка підлягає вилученню.

Виконання цієї операції не впливає на комунікатори, в яких використовується вилучена група).

Тепер коротко розглянемо функції для роботи з комунікаторами, причому тільки з такими, які використовуються для передачі даних всередині однієї групи процесів (тобто з *інтракомунікаторами*).

Для створення нових комунікаторів існують два основних способи.

Перший спосіб полягає в дублюванні вже існуючого комунікатора:

*int MPI_Comm_dup(MPI_Comm oldcom, MPI_comm *newcom),*

де *oldcom* – існуючий комунікатор, копія якого створюється;
*newcom* – новий комунікатор;

Другий спосіб полягає у створенні нового комунікатора із підмножини процесів існуючого комунікатора:

*int MPI_comm_create(MPI_Comm oldcom, MPI_Group group, MPI_Comm *newcom),*

де *oldcom* – існуючий комунікатор;

*group* – підмножина процесів комунікатора *oldcom*;
*newcom* – новий комунікатор.

Дублювання комунікатора може застосовуватись, наприклад, для усунення можливості перетину по тегах повідомлень в різних частинах паралельної програми (зокрема і при використанні функцій різних програмних бібліотек).

Варто відзначити, що операція створення комунікаторів є колективною і, тому має виконуватись всіма процесами початкового комунікатора.

Швидкий спосіб одночасного створення кількох комунікаторов забезпечує функція:

*int MPI_Comm_split(MPI_Comm oldcomm, int split, int key, MPI_Comm *newcomm)*,

де *oldcomm* – початковий комунікатор;
*split* – номер комунікатора, якому має належати процес;
*key* – порядок рангу процеса в новому комунікаторі;
*newcomm* – комунікатор, який створюється.

Оскільки створення комунікаторів відноситься до колективних операцій, тому виклик функції *MPI_Comm_split()* має бути виконаний в кожному процесі комунікатора *oldcomm*. В результаті виконання функції всі процеси розподіляються на групи з однаковими значеннями параметра *split*, які не перетинаються між собою. На основі сформованих груп створюється набір комунікаторів.

Для того щоб вказати, що процес не входить в жодний із нових комунікаторів, необхідно скористатись константою *MPI_UNDEFINED* як значенням параметра *split*. При створенні комунікаторів для рангів процесів в новому комунікаторі вибирається такий порядок нумерації, щоб він відповідав порядку значень параметрів *key* (процес з більшим значенням параметра *key* отримує більший ранг, процеси з однаковим значенням параметра *key* зберігають свою відносну нумерацію).

**Контрольні запитання**

1. Для яких комп'ютерних архітектур найбільш придатна технологія *MPI* ?
2. Яку роль виконують в *MPI* комунікатори?
3. Як здійснюється в *MPI* запуск програми на виконання?
4. Дайте характеристику типів функцій в *MPI*.
5. Які використовуються функції для реалізації парних комунікаційних операцій?
6. Які існують в *MPI* режими передачі повідомлень?
7. Які використовуються функції для реалізації колективних комунікаційних операцій?
8. Як здійснюється синхронізації процесів в *MPI*?

# 6 Паралельне програмування мовою С++ з використанням стандарту CUDA

## 6.1 Еволюція процесорних  пристроїв

Протягом кількох десятиріч з часу своєї появи основним способом підвищення продуктивності персональних комп'ютерів було збільшення тактової частоти процесора. Однак в останні роки зростання частоти більше не спостерігається. Це пов'язано як з обмеженнями технології виробництва мікросхем, так і зі збільшенням тепловиділення, яке пропорційне четвертому степеню частоти.

Існує також інше джерело зростання загальної продуктивності – збільшення числа обчислювальних пристроїв в комп'ютері. Замість того, щоб підвищувати продуктивність одного процесорного ядра, можна взяти декілька ядер. Тому зараз зростання продуктивності відбувається, в основному, за рахунок збільшення числа процесорних ядер, тобто за рахунок розпаралелювання обчислень [2, 4].

Але і просте механічне збільшення числа ядер не є ефективним вирішенням проблеми продуктивності. Необхідно знайти інший, більш інтелектуальний підхід.

Спочатку єдиний процесор комп'ютера виконував всі види обчислень, за що і отримав назву універсального або центрального процесорного пристрою (ЦПП) (*Central Processing Unit, CPU*). Уважний аналіз навантаження процесора показав, що існують відмінності в швидкості обробки різних видів даних.

Так вже склалось історично, що ЦПП був спроектований, насамперед, для обробки числових даних, як цілих, так і з рухомою комою. На початку 1990-х років велику популярність отримали графічні операційні системи і різні прикладні та ігрові програми з великою кількістю графіки. Навантаження на ЦПП значно зросло. Поява тривимірної графіки вимагала ще більше обчислювальних ресурсів.

Для розробників комп'ютерів стала очевидною необхідність введення «помічників», які допомогли б ЦПП справитись з підвищеним навантаженням. І першими такими «помічниками» стали графічні прискорювачі.

Спочатку їх роль була скромною, вони виконували лише нескладні операції з пікселями зображень. Випуск в 2001 році серії мікросхем GeForce 3 компанії Nvidia став найважливішим технологічним проривом у виробництві прискорювачів графіки.

В цій мікросхемі вперше був реалізований новий тоді стандарт Microsoft DirectX 8.0. Можливість програмування обробки графічних даних ознаменувала появу достатньо складних процесорних пристроїв, для яких вже стали використовувати термін ГПП – графічний процесорний пристрій (*Graphics Processing Unit, GPU*).

В чому відмінність архітектур CPU і GPU?

CPU має невелику кількість ядер, які працюють на високій тактовій частоті незалежно один від одного. В CPU є набір універсальних та спеціальних регістрів, кеш-пам'ять, вузол керування, а також АЛП для обробки цілих чисел і чисел з рухомою комою. В універсальних процесорах обчислювальні вузли займають відносно малу частину площі кристала.

GPU працює на низькій тактовій частоті і має сотні дуже простих обчислювальних елементів (маленьких АЛП), які займають основну частину площі кристала. Задачі обробки окремих елементів зображень можна обробляти одночасно на цих маленьких АЛП. Пізніше з'явилась можливість одночасної робот з великими графічними фрагментами. Таким чином, GPU максимально придатна для паралельної обробки.

### 6.2 Архітектури GPGPU та CUDA

Поява GPU означала, фактично, появу потужного паралельного процесора для обробки великих масивів однорідних даних. Але використовувати GPU для розв'язання традиційних задач спочатку було незручно. По-перше, пам'ять для збереження графічних даних має стекову організацію і тому неможливо записувати інформацію в довільні комірки пам'яті. По-друге, багато задач використовують числа з рухомою комою, з якими не працює GPU. Нарешті, для роботи з GPU необхідно було знати функції графічних бібліотек DirectX і OpenGL, а самі алгоритми обчислень мали бути написані спеціальними мовами графічного програмування – шейдерними мовами.

Таким чином, для більшості наукових задач звичайний GPU виявився непридатним. Тому розпочалась інтенсивна робота із пристосування графічних процесорів до загальноприйнятих принципів обчислень та програмування.

В новому GPU було дозволено арифметичні операції з рухомою комою та довільний доступ до пам'яті для читання і записування. Було створено спеціальну мову, яка була пристосована не тільки для програмування графіки, але і для обчислень загального призначення. З'явились також інші нововведення. Як результат виник новий напрям в комп'ютерній архітектурі, відомий нині як GPGPU (*General-Purpose Computing on Graphics Processing Units)* – використання графічних процесорів для розв'язання неграфічних задач.

В 2006 році компанія Nvidia випустила графічний процесор GeForce 8800 GTX з архітектурою GPGPU та відповідне програмне забезпечення. Так з'явилась нова модель обчислень під назвою CUDA (*Compute Unified Device Architecture*). Згодом з'явились подібні системи від інших фірм, але саме CUDA стала найвідомішим стандартом архітектури GPGPU [16].

CUDA основана на тісній взаємодії CPU (який називають *host*) і GPU (який називають *device*). GPU має власну пам'ять різних типів. Програма обчислень містить послідовний код, що виконується на CPU, і паралельний

код, що виконується на GPU. І в CPU, і в GPU можуть паралельно виконуватись потоки. Однак між ними існують суттєві відмінності. Для створення потоків CPU необхідні значні ресурси та час, тому їх може бути небагато (1–2 потоки на ядро). В GPU навпаки: можуть бути задіяні тисячі потоків, оскільки вони швидко створюються. Якщо для перемикання потоків в CPU необхідні сотні машинних тактів, то GPU перемикає декілька потоків за один такт.

Потоки GPU утворюють таку ієрархію (рис. 6.1). На нижньому рівні ієрархії потоки об'єднуються в блоки (*blocks*). Кожний блок – це одновимірний, двовимірний або тривимірний масив потоків (*threads*). Кожний потік в блоці має свою адресу, яка складається з одного, двох чи трьох цілих чисел.

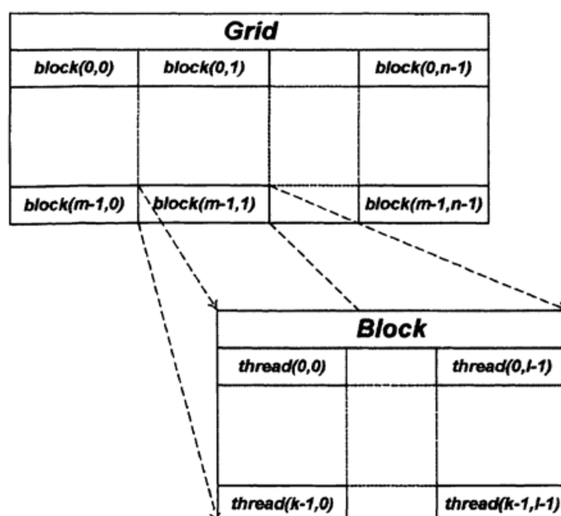



Рисунок 6.1 – Ієрархія потоків в CUDA

Потоки також можуть об'єднуватись в групи, зазвичай по 32 потоки, які називають варпом (*warp*). Потоки одного варпу виконуються фізично одночасно, потоки різних варпів можуть виконуватись на різних стадіях виконання програми. Всі потоки одного варпу належать одному блоку.

Потоки можуть взаємодіяти між собою лише в межах одного блока. Потоки різних блоків не можуть взаємодіяти між собою.

Верхній рівень ієрархії – сітка (*grid*) – утворює одновимірний або двовимірний масив блоків. Кожний блок в сітці має свою адресу, яка складається з одного чи двох цілих чисел.

Однією з суттєвих відмінностей між CPU і GPU є організація пам'яті.

В CPU дані можуть тимчасово зберігатися тільки в кількох регістрах та кеш-пам'яті. В GPU є такі типи пам'яті (рис. 6.2):

– регістрова (*register*),
– локальна (*local*),
– розподільна (*shared*)
– константна (*constant*),
– текстурна (*texture*),
– глобальна (*global*).

Кожний GPU містить 8192 або 16384 32-бітових регістрів. Регістрова пам'ять є найпростішою і найшвидшою. В CUDA немає явних засобів для її використання, всю роботу із розміщення даних в регістрах бере на себе компілятор.

Локальна пам'ять функціонує повільніше, ніж регістрова. Програміст може її використати, створивши локальні змінні, але рекомендується звертатись до цієї пам'яті лише у виняткових випадках.

Розподільна пам'ять використовується спільно різними потоками, що дає змогу обмінюватись даними між потоками в межах одного блока. В цю пам'ять можна явно записати дані за допомогою атрибуту *__shared__* .

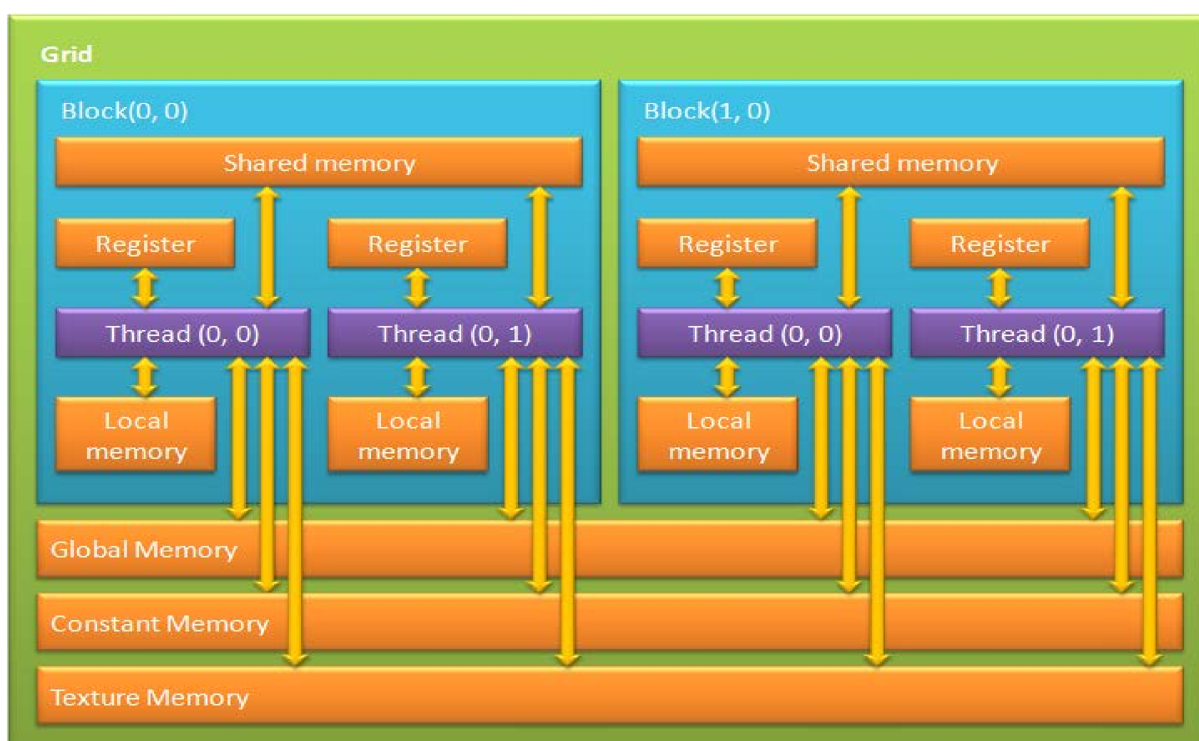



Рисунок 6. 2 – Типи пам'яті в CUDA

Константна пам'ять є доступною як для CPU, так і для GPU. CPU може записати в ній деякі дані, а GPU може їх лише прочитати, що і обумовлює її назву. В цю пам'ять можна явно записати дані за допомогою атрибуту *__constant__* .

Глобальна пам'ять є найповільнішим типом пам'яті, зате вона може зберігати великі обсяги даних, які надходять з CPU на GPU. В цю пам'ять можна явно записати дані за допомогою атрибуту *__ global __* .

Текстурна пам'ять слугує лише для збереження текстур, які використовуються при роботі з графікою.

Наявність різних типів пам'яті при їх правильному використанні дає можливість суттєво прискорити виконання програм. Наприклад, проміжні дані доцільно зберігати в розподільній пам'яті, яка працює в 100 разів швидше глобальної пам'яті. Для підвищення пропускної здатності розподільна пам'ять розбита на 16 або 32 банки. В кожний момент часу

банк пам'яті може виконувати одну операцію читання або одну операцію записування 32-бітного слова. 32-бітні слова, що йдуть поряд, потрапляють в різні банки пам'яті.

Якщо всі 32 потоки варпу звертаються до 32 32-бітних слів, які знаходяться в різних банках, тоді дані будуть отримані паралельно і без затримок. Але іноді можливі конфлікти, якщо потоки одного варпу здійснюють доступ до байтів розподільної пам'яті, які належать різним 32-бітним словам в одному банку пам'яті. Такі звертання до одного банку пам'яті можуть бути виконані тільки послідовно.

## 6.3 Апаратне забезпечення CUDA

Розглянемо детальніше архітектуру GPU, наприклад мікросхему GeForce 8800 GTS (рис. 6.3). Цей графічний процесор складається з восьми текстурних процесорних кластерів (*Texture Processor Cluster, TPC*).

Кожний кластер складається з текстурного блока *TEX* і двох потокових мультипроцесорів (*streaming multiprocessor, SM*). Кожний *SM* містить 8 скалярних процесорів (ядер) (*Scalar Processor, SC*). Кожний потік виконується на одному з цих ядер. До складу *SM* також входять два блоки для обчислення спеціальних функцій (*Special Function Unit, SFU*), блок керування командами (*Instruction Unit, IU*) і власна пам'ять. В більш нових GPU (Tesla C 1060) додатково міститься спеціальний блок для обробки 64-бітних чисел з рухомою комою (*Double Precision Unit, DPU*).

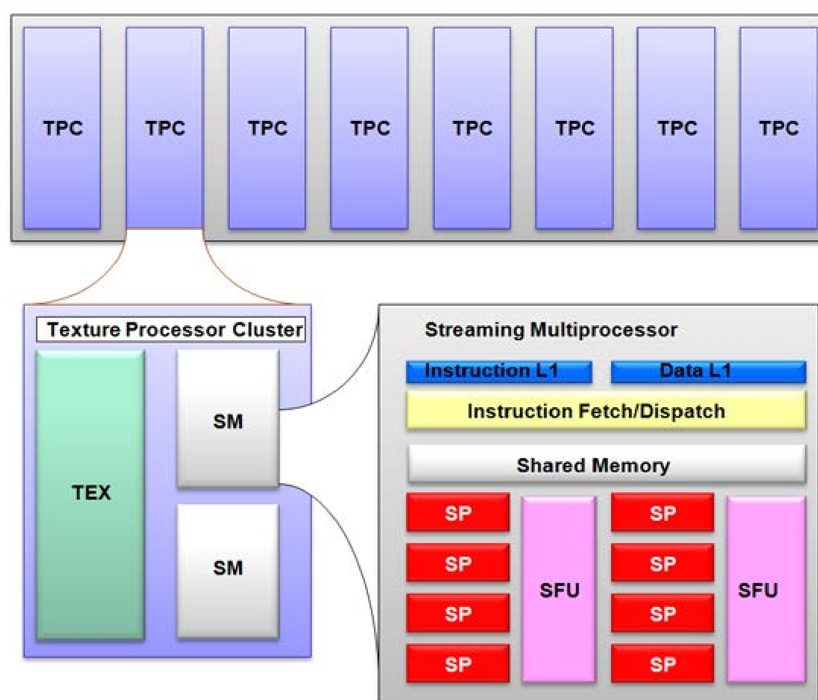



Рисунок 6.3 – Мультипроцесор на базі GeForce 8800 GTS

В одному мультипроцесорі виконуються всі потоки одного блока. Безпосередньо в мультипроцесорі знаходиться пам'ять таких типів:

– регістрова пам’ять (8192–16384 регістрів);

– розподільна пам’ять, яка доступна всім скалярним ядрам;

– кеш константної пам’яті, яка доступна для читання всім скалярним ядрам;

– кеш текстурної пам’яті, яка доступна для читання всім скалярним ядрам.

Розмір розподільної пам’яті становить по 16 кбайт на мультипроцесор. Ця область пам’яті відкриває можливість обміну інформацією між потоками в одному блоці.

Доступ до текстурної пам’яті здійснюється за допомогою *TPC* (див. рис. 6.3).

Кожне нове покоління графічних процесорів вводить нові елементи архітектури. Наприклад, групу з чотирьох мультипроцесорів архітектури Fermi прийнято називати *GPC* (*Graphics Processing Cluster*) замість абревіатури *TPC*.

В мультипроцесорах реалізовано принцип паралельної обробки *SIMD*, коли одна команда (інструкція) застосовується до всіх потоків у *warp*, (в CUDA така група з 32 потоків – мінімальний набір даних, що обробляється мультипроцесорами). Цей спосіб виконання в графічних процесорах ще називають *SIMT* (*single instruction multiple threads* – одна команда (інструкція) і багато потоків). В Geforce GTX 280 на основі GT200 може бути до 1024 потоків на мультипроцесор.

Мультипроцесор не є традиційним багатогоядерним процесором, але він відмінно пристосований до багатопотоковості, підтримуючи до 32 *warp*'ів одночасно. Кожний такт апаратне забезпечення вибирає, який з *warp*'ів має виконуватись, і перемикається від одного до іншого без часових втрат в тактах. Якщо проводити аналогію з центральним процесором – це схоже на одночасне виконання 32 програм.

## 6.4 Програмне забезпечення CUDA

Для значного поширення ідеології розв’язання неграфічних задач на графічних процесорах ще недостатньо наявності лише швидкодійних апаратних засобів. Не кожний програміст захоче вивчити особливості графічних бібліотек OpenGL або DirectX, щоб пристосувати їх для розв’язання своєї конкретної задачі. Тому Nvidia розробила для архітектури CUDA відповідне програмне забезпечення, яке стало доступним та зрозумілим для широкого кола користувачів.

Для програмування в CUDA за основу було взято найвідомішу мову системного і прикладного програмування – мову С. До неї було лише введено необхідні доповнення для врахування особливостей нової архітектури. Також було запропоновано зручні засоби розробки прикладних програм:

– драйвер пристрою NVIDIA;

– комплект засобів розробки CUDA Toolkit;

– компілятор мови CUDA  C.

Компанія Nvidia розробила Сі-компілятор командного рядка *nvcc*, який призначений для трансляції *host*-коду (файлів з розширенням *.c*) і *device*-коду (файлів з розширенням *.cu*) в об'єктні файли, які далі можуть бути скомпоновані у виконуючий файл з використанням або утиліти *make* (*nmake*), або програмного пакета Microsoft Visual Studio.

Основні нововведення, які було введено в стандартну мову С, такі:

а) атрибути функцій, які вказують на місце їх виконання та місце їх виклику;

б) атрибути змінних, які задають тип пам'яті для них;

в) нові типи і нові змінні;

г) нові функції;

д) оператор виклику ядра;

е) бібліотека функцій *CUDA host API*.

## 6.5 Огляд нововведень в мову С

### 6.5.1 Атрибути функцій та змінних в CUDA

Нові атрибути функцій записані в таблиці 6.1.

Таблиця 6.1 – Атрибути функцій

| Атрибут | Місце виконання | Місце виклику |
|---|---|---|
| *__device__* | *device (GPU)* | *device (GPU)* |
| *__global__* | *device (GPU)* | *host (CPU)* |
| *__host__* | *host (CPU)* | *host (CPU)* |

Атрибут *__global__* означає ядро і відповідна функція має повертати значення типу *void*.

Атрибути *__device__* і *__host__* можуть бути використані разом, тобто відповідна функція може виконуватись як на *CPU*, так і на *GPU*. Атрибути *__global__* і *__host__* не можуть бути використані разом.

Для задання розміщення в памяті *GPU*  змінних використовуються такі атрибути: *__device__* , *__constant__* , *__shared__* . На них накладаються такі обмеження:

– ці атрибути не можуть бути використані до полів структури (*struct* або *union*);

– відповідні змінні можуть використовуватись тільки в межах одного файла (не можна оголошувати їх як *extern*);

– запис в змінні типу *__constant__*  може здійснюватись *CPU* лише за допомогою спеціальних функцій;

– змінні типу *__shared__*  не можуть бути ініціалізовані при їх оголошенні.

### 6.5.2 Нові типи в CUDA

В мову додано 1/2/3/4-вимірні вектори з базових типів (*char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, longlong, float, double*). Ім'я векторного типу формується із базового імені та числа елементів, наприклад, *int3*).

Звертання до компонентів вектора йде за іменами *x, y, z, w*). Для створення значень-векторів заданого типу служить синтаксична конструкція *make.* Наприклад, для створення вектора цілого типу необхідно записати:

*int3 mas=make_int3 (4,7,4);*

Для додавання двох векторів необхідно цю операцію виконати окремо для кожної компоненти вектора.

Також доданий тип *dim3*, який використовується для задання розмірності блоків і сіток. Наприклад, *dim blocks (16, 16).*

### 6.5.3 Нові змінні в CUDA

В мову CUDA C додано такі спеціальні змінні:

– *gridDim* – розмір сітки (має тип *dim3*);
– *blockDim* – розмір сітки блока (має тип *dim3*);
– *blockIdx* – індекс поточного блока в сітці (має тип *uint3*);
– *threadIdx* – індекс поточного блока в сітці (має тип *uint3*);
– *warpSize* – розмір варпу (має тип *int*);

### 6.5.4 Нові функції в CUDA

В мові CUDA C підтримуються всі математичні функції із стандартної бібліотеки мови C. Однак, зазначені функції працюють не з *double*-числами, а з *float*-числами, що дає можливість прискорити їх швидкість виконання. Таким чином, для кожної стандартної функції є її *float*-аналог, наприклад, *cosf(x).*

Крім того, надається також спеціальний набір функцій зниженої точності, але ще більшої швидкодії. Наприклад, функції *__logf(x), __sinf(x).*

Для деяких функцій можна задати спосіб округлення:

– *rn* – округлення до найближчого,
– *rz* – округлення до нуля,
– *ru* – округлення вгору,
– *rd* – округлення вниз.

Для виклику ядра використовується така конструкція:

*kernel <<<Dg, Db, Ns, S>>> (args);*

де *kernel* – ім'я відповідної *__global__* функції;
*Dg* (типу *dim3*) – розмірність сітки блоків;
*Db* (типу *dim3*) – розмірність блока потоків;

*Ns* (типу *size_t*) – додатковий обсяг *shared*-пам’яті в байтах, яка має бути динамічно виділена кожному блоку (необов’язковий параметр, за замовчуванням 0);

*S* (типу *cudaStream*) – номер потоку, в якому має відбутись виклик (за замовчуванням використовується потік 0);

*args* – аргументи виклику ядра.

Наприклад, необхідно викликати ядро з іменем *mykernel* паралельно на *n* потоках, використовуючи одновимірний масив із двовимірних блоків потоків 16×16 з передачею ядру параметрів *a* та *n*. при цьому кожному блоку додатково виділяється 512 байт *shared*-пам’яті і запуск ядра відбувається в потоці *my_stream.* Для виконання такої задачі необхідно записати:

*mykernel <<<256, dim3(16,16), 512, my_stream >>> (a, n);*

### 6.5.5 CUDA host API

CUDA надає програмісту набір функцій, які можуть бути використані тільки для CPU. Ці функції, відомі під назвою *CUDA host API*, відповідають за таке:

– керування графічним процесором GPU;
– робота з контекстом;
– робота з пам’яттю;
– робота з модулями;
– керування виконанням коду;
– робота з текстурами;
– взаємодія з графічними бібліотеками OpenGL і Direct3D.

*CUDA host API* виступає в двох формах:

– високорівневий *CUDA runtime API*,
– низькорівневий *CUDA driver API*.

Зазначені API є взаємно протилежними – в одній програмі можна використовувати тільки один з них.

Ці API відрізняються також способом компіляції та отримання виконуючого коду. Для *CUDA runtime API host*-програма і *device*-програма компілюються компілятором, графічний процесор отримує вже готовий PTX код для себе. Для *CUDA driver API host*-програма компілюється компілятором, а *device*-програма компілюються драйвером пристрою.

Високорівневі функції API працюють «зверху» низькорівневого, тобто всі виклики *runtime* транслюються в декілька простих команд, які обробляються низькорівневим *CUDA driver API*. Для кожної функції *runtime API* є аналог функції *driver API*, тому перехід з верхнього рівня на нижній не надто складний. До недоліків *driver API* можна віднести не тільки збільшення розміру програм, але й необхідність явних налаштувань та ініціалізації, а також відсутність режиму емуляції (який дозволяє

компілювати, запускати і налагоджувати програми). Зате низькорівневий *API* дає програмісту більше можливостей в реалізації складних задач.

Важливо відзначити, що над *CUDA runtime API*, існує ще більш високий рівень, який утворюється спеціалізованими математичними бібліотеками, які мають свої API: *CUFFT* (перетворення Фур'є), *CUBLAS* (лінійна алгебра), *CUDPP* (сортування і псевдовипадковий генератор).

В *CUDA runtime API* входять такі групи функцій:

– *Device Management* – містить функції для загального керування GPU,
– *Thread Management* – керування потоками,
– *Event Management* – функція створення і керування подіями,
– *Execution Control* – функції запуску і виконання ядра CUDA,
– *Memory Management* – функції керування пам'яттю GPU,
– *Texture Reference Manager* – робота з об'єктами текстур через CUDA,
– *OpenGL Interoperability* – функції взаємодії з OpenGL API,
– *Direct3D 9 Interoperability* – функції взаємодії з Direct3D 9 API,
– *Direct3D 10 Interoperability* – функції взаємодії з Direct3D 10 API,
– *Error Handling* – функції обробки помилок.

Всі функції *runtime API* реалізовано в динамічній бібліотеці *cudart* і їх імена починаються з префікса *cuda*. Всі функції *driver API* реалізовано в динамічній бібліотеці *nvcuda* і їх імена починаються з префікса *cu*.

Далі буде, в основному, використовуватись *runtime API* як більш простий. За необхідності програмний код може бути переписаний на основі функцій *driver API* .

Важливим моментом роботи з CUDA є те, що багато функцій асинхронні, тобто керування повертається ще до реального завершення необхідної операції. До асинхронних функцій належать:

– функція запуску ядра;
– функції ініціалізації пам'яті;
– функції копіювання пам'яті *device<->device*;
– функції копіювання змінних пам'яті, імена яких закінчуються на *Async*;

Для синхронізації потоку на CPU з GPU використовується функція *cudaThreadSynchronize*, яка чекає завершення виконання всіх операцій CUDA, які були раніше викликані з цього потоку GPU.

CUDA підтримує синхронізацію через потоки: кожен потік задає послідовність операцій в строго визначеному порядку. При цьому порядок виконання операцій між різними потоками не є строго визначеним і може змінюватись.

Кожна функція *CUDA runtime API* (крім запуску ядра) повертає значення типу *cudaError_t*. При успішному виконанні функції

повертається значення *cudaSuccess*, в іншому випадку повертається код помилки.

Розглянемо дуже просту програму (програма 6.1), написану мовою CUDA C з використанням пакета Microsoft Visual Studio.

**Приклад 6.1**. Програма для ілюстрації програмування в CUDA.

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>

__global__void kernel(int a,int b,int *c)
{  *c=a+b;  }
int main(void)
{
        int *y;
HANDLE_ERROR( cudaMalloc( (void**)*c, sizeof(int) );
        kernel <<<1,1>>> (2,3,y);
        printf("Hello");
    _getch();
}
```

В цій програмі є дві функції: *main()*, яка компілюється на CPU, та *kernel(…)*, яка компілюється на GPU. Остання функція, яку називають ядром, викликається з головної функції *tmain(…).* Під час виклику ядра задається кількість потоків, необхідних для виконання поставленої задачі – в цьому випадку достатньо одного. Для змінної *y* має бути виділена пам'ять за допомогою функції *cudaMalloc(…)*, аргументами якої є покажчик адреси виділеної області пам’яті та розмір цієї області. Якщо пам’ять не буде виділена, то зазначена функція поверне значення з кодом помилки.

До важливих функцій *CUDA runtime API* належать ті, які надають різноманітну інформацію про апаратні та програмні засоби CUDA. Наприклад, дізнатись про число CUDA-пристроїв та їх характеристики можна за допомогою функцій *cudaGetDeviceCount()* та *cudaGetDeviceProperties()*. Ці функції повертають свої значення в структурі типу *cudaDeviceProp.* В кожній новій версії CUDA кількість полів в цій структурі збільшується, тому згадаємо лише найголовніші поля.

```
Struct cudaDeviceProp
{
        char name [256];  // Назва пристрою
        size_t totalGlobalMem;  // Повний обсяг глобальної пам’яті в байтах
        size_t sharedMemPerBlock;  // Обсяг розподільної пам’яті в блоці
        int regsPerBlock;  // Кількість 32-бітових регістрів в блоці
        int warpSize;  // Розмір варпа
        size_t memPitch;  // Максимальний pitch в байтах, що допускається
```

```
                    // функціями копіювання пам’яті, яка виділена
                    // через cudaMallocPitch
    int maxThreadsPerBlock;  // Максимальна кількість активних потоків
                             // в блоці
    int maxThreadsDim [3];  // Максимальний розмір блока кожного
                            // виміру
    int maxGridSize;  // Максимальний розмір сітки кожного виміру
    size_t totalConstMem;  // Обсяг константної пам’яті в байтах
    int major;  // Compute Capability, старший номер
    int minor;  // Compute Capability, молодший номер
    int clockRate;  // Частота в кілобайтах
    size_t textuteAlignment;  // Вирівнювання пам’яті для текстур
    int deviceOverlap; // Копіювання паралельно з обчисленнями
    int multiProcessorCount;  // Кількість мультипроцесорів в GPU
    int kernelExecTimeoutEnables;  // 1, якщо є обмеження на термін часу
                                   //  виконання ядра
    int integrated; // 1, якщо GPU вбудовано в системну плату
    int canMapHostMemory;  // 1, якщо відображати пам’ять CPU в
                           // пам’ять CUDA
    int computeMode; // Режим, в якому знаходиться GPU
}
```

Значення основних полів зрозумілі із коротких коментарів. Варто лише звернути увагу на позначення архітектурної версії GPU CUDA. Для цього використовується поняття *Compute Capability*, яке задається парою цілих чисел – *major.minor*. Перше число означає глобальну архітектурну версію, а друге – незначні зміни. Наприклад, GPU GeForce 8800 Ultra/GTX/GTS має значення 1.0.

Фрагмент програми для отримання довідкової інформації про ресурси комп’ютера може бути таким:

**Приклад 6.2**. Отримання довідкової інформації в CUDA.

```
int Count;
cudaDeviceProp devProp;
cudaGetDeviceCount(&Count);
printf("Found %d devices\n",Count);
for(int i=0; i<Count; i++)
{
cudaGetDeviceProperties(&devProp,i);
printf("Total Global Memory %d\n",devProp.totalGlobalMem);
printf("Shared Memory %d\n",devProp.sharedMemPerBlock);
printf("Warp size %d\n",devProp.warpSize);
printf("Max threads per block %d\n",devProp.maxThreadsPerBlock);
printf("MultiProcessor Count %d\n",devProp.multiProcessorCount);
```

```
printf("maxThreadsPerMultiProcessor
%d\n",devProp.maxThreadsPerMultiProcessor);
}
```

Для дослідження ефективності розпаралелювання на графічних процесорах важливим є визначення часу виконання програм. Для цього використовується подія (*events*) CUDA. Подія – це об'єкт типу *cudaEvent_t*, який використовується для позначення «точки» серед викликів CUDA. Кожна подія «прив'язується» до тієї чи іншої «точки» і характеризується тим, пройдена GPU ця «точка» чи ні. За допомогою функції для роботи з подіями можна створювати і знищувати події, «прив'язувати» їх до певних місць в програмі, чекати настання подій, а також отримувати інтервал часу в мілісекундах між настанням двох подій.

### 6.6 Основи створення програм в CUDA

Таким чином, CUDA пропонує спеціальний підхід до розробки програм, не зовсім такий, як прийнято в програмах для CPU.

Модель програмування в CUDA передбачає групування потоків. Потоки (*threads*) об'єднуються в блоки потоків (*thread blocks*) – одновимірні чи двовимірні сітки потоків, що взаємодіють між собою за допомогою *shared* пам'яті та точок синхронізації. Програма (*ядро, kernel*) виконується над сіткою (*grid*) блоків потоків. Одночасно виконується одна сітка.

Модель CUDA передбачає, що програміст спочатку розбиває задачу на незалежні частини (блоки), які можуть виконуватись паралельно. Блоків має бути не менше числа мультипроцесорів. Порядок виконання блоків не визначено, блоки мають бути незалежними один від одного. При запуску ядра блоки сітки нумеруються і розподіляються по мультипроцесорах, які мають достатньо вільний обсяг регістрів, *shared* пам'яті та ресурсів планувальника команд. Мультипроцесор зазвичай складається з 8 простих процесорів, двох процесорів для складних операцій (наприклад, множення), пулу регістрів, розподільної пам'яті.

Далі кожен блок розбивається на сукупність потоків, що паралельно виконуються і можуть залежати один від одного. Кількість потоків в блоці та кількість блоків в сітці вибирається програмістом, виходячи з максимізації завантаження ресурсів мультипроцесора і з врахуванням апаратних обмежень (кількості регістрів, обсягу розподільної пам'яті тощо). Кількість потоків в блоці має бути кратною розміру *warp*, тобто 32. Виходячи з цих обмежень, кожний блок може складатись з 512 потоків на цьому апаратному забезпеченні.

Потоки всередині блока запускаються на одному мультипроцесорі, мають спільну розподільну пам'ять і можуть (повинні) синхронізувати хід виконання задачі. Кожний потік має унікальний ідентифікатор всередині

блока, який записується за допомогою одновимірного, двовимірного або тривимірного індексу (вбудована структурна змінна *threadIdx* типу *dim3*). Розмірність блока доступна через вбудовану змінну *blockDim*. Індекс блока в сітці доступний через вбудовану змінну *blockIdx*.

Для виконання однієї команди порції потоків мультипроцесор має завантажити операнди для всіх потоків порції, виконати команду (одночасно), записати результат.

Підведемо короткі висновки. Типовий, але необов'язковий, порядок розв'язання задач в моделі програмування в CUDA:

– задача розбивається на підзадачі;

– вхідні дані поділяються на блоки, які поміщаються в розподільну пам'ять;

– кожний блок даних обробляється блоком потоків;

– дані підвантажуються в розподільну пам'ять із глобальної;

– над даними в розподільній пам'яті проводяться відповідні обчислення;

– результати копіюються з розподільної пам'яті назад в глобальну.

Ядро, тобто паралельна частина програми, виконується на GPU, а решта програми – на CPU (хості). GPU і CPU може бути декілька. GPU мають бути однаковими. Один процес хоста може використовувати тільки один GPU.

Хост-система має свою ОП, обмін з глобальною пам'яттю GPU і запуск ядра здійснюється через спеціальний інтерфейс. Запуск ядра є асинхроним, тобто керування негайно повертається хостовій програмі. Також асинхроним може бути обмін між хостом і GPU та пересилання всередині GPU. Програма може використовувати як функції низькорівневого інтерфейсу *CUDA driver API*, так і функції високорівневого інтерфейсу *CUDA runtime API*.

Як приклад розглянемо задачу додавання двох векторів з використанням *CUDA runtime API* [ ]. Спочатку розглянемо частину програми для хоста. Для кращого розуміння суті програми дамо коментарі для кожного фрагмента програми.

**Приклад 6.3**. Програма додавання двох векторів з використанням *CUDA runtime API*.

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#define N 20
int main(void)
{
int a[N], b[N], c[N]; int *dev_a, *dev_b, *dev_c;
// виділити пам'ять на  на GPU
HANDLE_ERROR(cudaMalloc( (void**)&dev_a, N * sizeof(int)) );
```

```
HANDLE_ERROR(cudaMalloc( (void**)&dev_b, N * sizeof(int)) );
HANDLE_ERROR(cudaMalloc( (void**)&dev_c, N * sizeof(int)) );
// заповнити масиви ‘a’ та ‘b’ на CPU
for(int i; i<N; i++)
{
        a[i]= –i;    b[i]=i*i;
}
// копіювати масиви ‘a’ та ‘b’ на GPU
HANDLE_ERROR( cudaMemcpy(dev_a, a, N * sizeof(int),
                                    cudaMemcpyHostToDevice) );
HANDLE_ERROR( cudaMemcpy(dev_b, b, N * sizeof(int),
                                    cudaMemcpyHostToDevice) );
// викликати функцію ядра
add<<<N,1>>>(dev_a, dev_b, dev_c);
// копіювати масив ‘c’ з GPU  на  GPU
HANDLE_ERROR( cudaMemcpy(dev_c, c, N * sizeof(int),
                                    cudaMemcpyDeviceToHost) );
// вивести результат
for(int i; i<N; i++)
{
        printf(“% + %d  = %d \n”, a[i], b[i], c[i]);
}
cudeFree(dev_a); cudeFree(dev_b); cudeFree(dev_c);
}
```

Тепер наведемо фрагмент програми для GPU, де відбувається операція додавання двох векторів з використанням паралельних потоків.

```
__global__ void add(int *a, int *b, int *c)
{
int tid=blockIdx.x // обробити дані по цьому індексу
        if (tid<N)
c[tid]=a[tid]+b[tid];
}
```

### 6.7 Підготовка до виконання програм

Для встановлення CUDA необхідно скопіювати з Інтернету за адресою *http://www.nvidia.com/object/cuda_get.html* і інсталювати на комп’ютері користувача такі програмні пакети:

*CUDA driver,*
*CUDA Toolkit,*
*CUDA SDK.*

Нагадаємо, що потрібно вибрати версії зазначених програм відповідно до операційної системи та її розрядності (32 або 64).

Для компіляції прикладних програм на CUDA необхідно також мати відповідний компілятор з мов C/C++, наприклад, *mingw* або *cygwin*. Можна також скористатись утилітою *nvcc* із пакета *CUDA Toolkit*. Для операційних систем Windows найбільш зручним є компілятор, який входить до складу програмного середовища Microsoft Visual Studio. Для новачків можна скористатись спеціальним Майстром CUDA програм (CUDA VS Wizard), який можна знайти за адресою *http://sourceforge.net/projects/cudavswizard*.

**Контрольні запитання**

1. В чому полягає відмінність архітектур CPU і GPU?
2. Дайте характеристику моделі паралельних обчислень CUDA.
3. Поясніть ієрархію потоків в CUDA.
4. Які існують типи пам’яті в CUDA?
5. Розкажіть про апаратне забезпечення CUDA.
6. Розкажіть про програмне забезпечення CUDA.
7. Які з’явились основні нововведення в мові C CUDA?
8. Дайте характеристику функцій CUDA host API.
9. Яка різниця між *host*-програмою і *device*-програмою?
10. Розкажіть про типовий порядок розв’язання задач в моделі програмування в CUDA.

## Післямова

Головною метою введення паралельної обробки в програмування є зменшення часу виконання програм. Однак цього не завжди вдається досягти. Як і всі інші відомі програмні технології, паралельні технології мають свої переваги і недоліки, тому важливо знати ті умови, коли використання паралелізму є доцільним та виправданим.

На практиці для переважної більшості задач розпаралелюванню піддається лише частина задача, тобто програма реалізації такої задачі буде складатись з послідовної та паралельної частин. Згідно з третім законом Амдала величина прискорення обчислень обернено пропорційна відсотку послідовної частини програми [5]. Це означає, що основним інструментом підвищення продуктивності роботи має бути не безкінечне збільшення кількості процесорних ядер, а збільшення відсотку паралельної частини програми за рахунок використання нових паралельних алгоритмів.

Паралелізм в сучасних комп'ютерах реалізовується за рахунок створення додаткових процесів та потоків. Як і стосовно апаратури, процесів та потоків також не має бути занадто багато, інакше більшість з них будуть простоювати в чергах до ядер. Процедура створення та підтримка нових потоків потребує великих системних ресурсів, тому кожен потік має виконувати стільки роботи, щоб виправдати своє існування.

Найскладнішою проблемою в багатоядерних та багатопотокових системах є балансування навантаження ядер. Неможливо забезпечити ефективну роботу апаратури, аналізуючи складну задачу лише теоретично. Велику допомогу у налагодженні паралельних програм надають спеціалізовані програмні продукти фірми Intel (Intel Thread Checker, Intel Debugger та інші).

Таким чином, паралельне програмування – це нова і достатньо складна дисципліна. Вона потребує як глибоких теоретичних знань для розробки паралельних алгоритмів та правильного вибору технологій паралельної обробки, так і практичних навичок для написання та налагодження паралельних програм. Оптимальне поєднання теорії та практики дозволить складати ефективні і швидкодійні паралельні програми для сучасних комп'ютерів.

Посильну допомогу в цьому може надати наш навчальний посібник.

ГЛОСАРІЙ

| **Український варіант** | **Англійський варіант** |
|---|---|
| Апаратно-уніфікована обчислювальна архітектура | *Compute Unified Device Architecture,* CUDA |
| Арифметико-логічний пристрій | *Arithmetic and Logic Unit, ALU* |
| Багатозадачність | *Multitasking* |
| Багатозадачність кооперативна | *Cooperative multitasking* |
| Багатозадачність пріоритетна | *Preemptive multitasking* |
| Багатопотоковість | *Multithreading* |
| Багатопотоковість одночасна | *Simultaneous Multi-Threading, SMT* |
| Багатопотоковість на кристалі | *Chip Multi-Threading, CMT* |
| Багатопроцесорна обробка на кристалі | *Chip Multiprocessing, CMP* |
| Багато потоків команд & один потік даних | *Multiple Instruction & Single Data, MISD* |
| Багато потоків команд & багато потоків даних | *Multiple Instruction & Multiple Data, MIMD* |
| Бібліотека паралельних задач | *Task Parallel Library, TPL* |
| Блок керування командами | *Instruction Unit, IU* |
| Блок для обробки 64-бітових чисел з рухомою комою | *Double Precision Unit, DPU* |
| Блок для обчислення спеціальних функцій | *Special Function Unit, SFU* |
| Гіперпотоковість | *Hyper-Threading, HT* |
| Графічний процесорний пристрій | *Graphics Processing Unit, GPU* |
| Графічні процесори для розв'язання неграфічних задач | *General-Purpose computing on Graphics Processing Units, GPGPU* |
| Декомпозиція за даними | *Decomposition in data* |
| Декомпозиція за задачами (функціями) | *Decomposition in tasks (functions)* |
| Декомпозиція за часом | *Decomposition in time* |
| Делегат | *Delegate* |
| Задача | *Task* |
| Замок для блокування | *Lock* |
| Інтерфейс обміну повідомленнями | *Message Passing Interface, MPI* |
| Кластер | *Cluster* |
| Кластер графічних процесорів | *Graphics Processing Cluster, GPC* |
| Кластер текстурних процесорів | *Texture Processor Cluster, TPC* |
| Комунікатор | *Communicator* |
| Комунікатор глобальний | *intercommunicator* |
| Комунікатор локальний | *intracommunicator* |

| Критична секція | *Critical section* |
|---|---|
| Симетричний мультипроцесор | *symmetric multiprocessor, SMP* |
| Одна команда & багато потоків | *single instruction multiple threads, SIMT* |
| Один потік команд & один потік даних | *Single Instruction & Single Data, SISD* |
| Один потік команд & багато потоків даних | *Single Instruction & Multiple Data, SIMD* |
| Пам'ять глобальна | *Global memory* |
| Пам'ять константна | *Constant memory* |
| Пам'ять локальна | *Local memory* |
| Пам'ять регістрова | *Register memory* |
| Пам'ять розподілена | *Distributed memory* |
| Пам'ять спільна | *Shared memory* |
| Пам'ять текстурна | *Texture memory* |
| Паралелізм одночасний | *Concurrency* |
| Паралелізм загальний | *Parallelism* |
| Паралельні колекції | *Concurent Collections* |
| Повідомлення | *Message* |
| Подія | *Event* |
| Потік | *Thread* |
| Потік головний | *Master thread* |
| Потоковий мультипроцесор | *Streaming multiprocessor, SM* |
| Пріоритет | *Priority* |
| Процес | *Process* |
| Процесор векторний паралельний | *Parallel vector processor, PVP* |
| Процесор графічний | *Graphics Processing Unit, GPU* |
| Процесор масивно-паралельний | *Massively parallel processor, MPP* |
| Процесор скалярний | *Scalar Processor, SC* |
| Процесор суперскалярний | *Superscalar processor* |
| Процесор центральний | *Central Processing Unit, CPU* |
| Пул потоків | *ThreadPool* |
| Ранг процесу | *Process Rank* |
| Редукція | *Reduction* |
| Семафор | *Semaphore* |

# Список рекомендованої літератури


1. Хокни Р. Параллельные ЭВМ. Архитектура, программирование и алгоритмы / Р. Хокни, К. Джессхоуп. – М. : Радио и связь, 1986. – 392 с.

2. Шпаковский Г. И. Реализация параллельных вычислений: кластеры, многоядерные процесоры, грид, квантовые компьютеры. – Минск : БГУ, 2010. – 155 с.

3. Карцев М. А. Архитектура цифровых вычислительных машин / Карцев М. А. – М. : Наука, 1978. – 295 с.

4. Воеводин В. В. Параллельные вычисления / В. В. Воеводин, Вл. В. Воеводин. – СПб. : БХВ-Петербург, 2002. – 608 с.

5. Эхтер Ш. Многоядерное программирование / Ш. Эхтер, Дж. Робертс ; пер. с англ. А. Лашкевич. – СПб. : Питер, 2010.

6. Хокинг С. Краткая история времени: От Большого взрыва до черных дыр/ Хокинг С. – СПб. : Амфора, 2008. – 231 с.

7. Пригожин И. Время, хаос, квант / И. Пригожин, И. Стенгерс. – М. : Издат. группа Прогресс, 1994. – 272 с.

8. Семеренко В. П. Темпоральные модели параллельных вычислений / В. П. Семеренко // Austrian Journal of Technical and Natural Sciences. – 2014. – Vol. 1. – P. 13–25.

9. Семеренко В. П. Теорія циклічних кодів на основі автоматних моделей : монографія / Семеренко В. П. – Вінниця : ВНТУ, 2015. – 444 с.

10. Шеховцов В. А. Операційні системи / Шеховцов В. А. – Київ : Видавнича група BHV, 2005. – 576 с.

11. Таненбаум Э. Современные операционные системы / Э. Таненбаум ; [ 3-е изд.]. – СПб. : Питер, 2010. – 1040 с.

12. Шилдт Г. C# 4.0: полное руководство / Шилдт Г. – М. : ООО «И.Д. Вильямс», 2012. – 1056 с.

13. Антонов А. С. Параллельное программирование с использованием технологии OpenMP : учебное пособие / Антонов А. С. – М. : Изд-во МГУ, 2009. – 77 с.

14. Хьюз К. Параллельное и распределенное программирование на С++ / К. Хьюз, Т. Хьюз. – М. : «И.Д. Вильямс», 2004. – 672 с.

15. Антонов А. С. Параллельное программирование с использованием технологии MPI : учебное пособие / Антонов А. С. – М. : Изд-во МГУ, 2004. – 71 с.

16. Параллельные вычисления на GPU. Архитектура и программная модель CUDA : учебное пособие / [А. В. Боресков, Н. Д. Марковский, Д. Н. Микушин и др.]. – М. : Изд-во МГУ, 2012. – 336 с.

17. Миллер Р. Последовательные и параллельные алгоритмы: Общий подход / Р. Миллер, Л. Боксер. ; пер. с англ. – М. : БИНОМ. Лаборатория знаний, 2006. – 406 с.

18. Интернет-Университет Суперкомпьютерных Технологий: [Електронний ресурс] : www.intuit.ru/studies/courses/1156/190/lecture/4948.

*Навчальне видання*

**Семеренко Василь Петрович**

# ТЕХНОЛОГІЇ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Навчальний посібник

Рукопис оформлено В. Семеренком

Редактор Т. Старічек

Оригінал-макет виготовлено О. Ткачуком

Підписано до друку 04.06.2018 р.
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 5,98.
Наклад 50 (1-й запуск 1–20) пр. Зам. № 2018-091.

Видавець та виготовлювач
інформаційний редакційно-видавничий центр.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
Тел. (0432) 65-18-06.
**press.vntu.edu.ua**;
*E-mail*: kivc.vntu@gmail.com.
Свідоцтво суб’єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.