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 студентів 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.....45
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.....74
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 С...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 кінцевою 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).

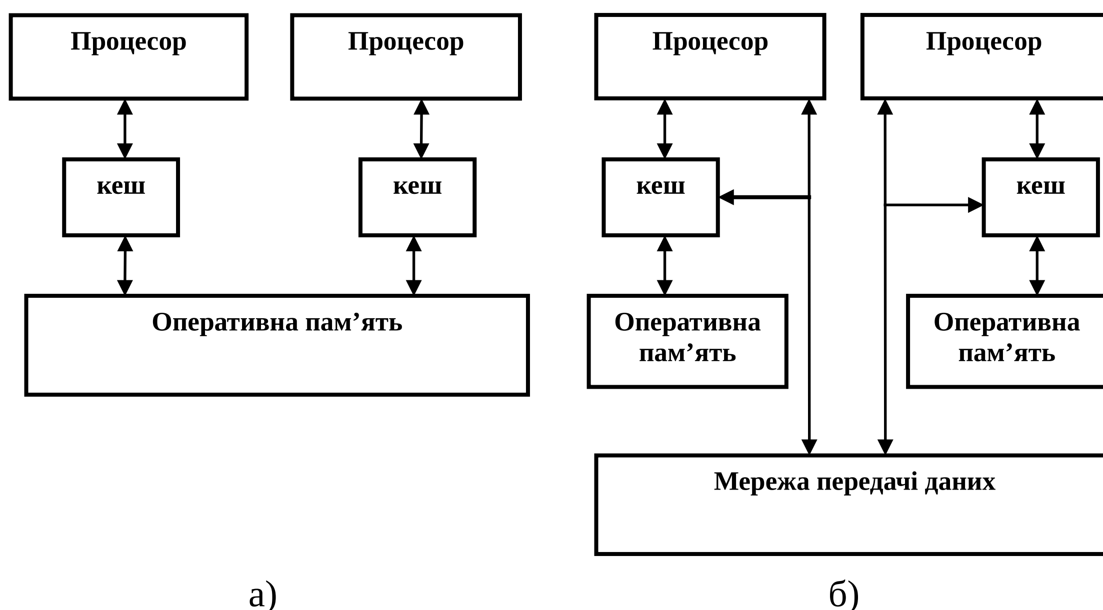

Рисунок 1.1 – Способи підключення до основної пам’яті:
а) спільна пам’ять, б) розподілена пам’ять

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.

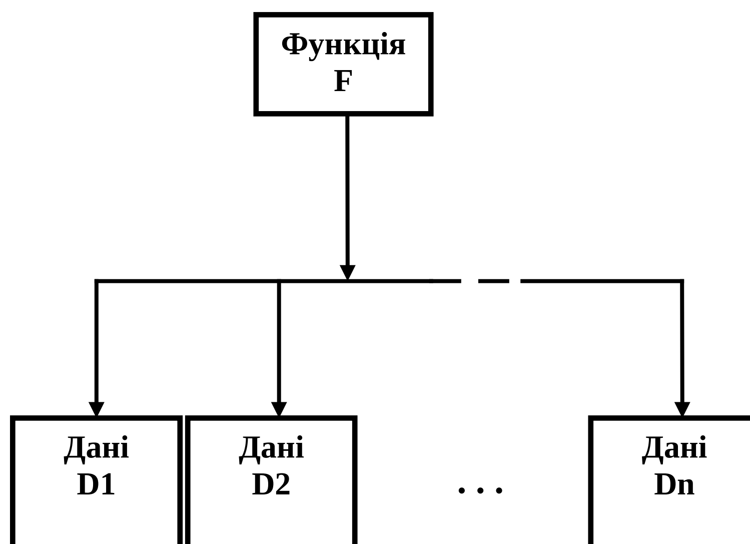

Рисунок 1.2 – Декомпозиція за даними

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).

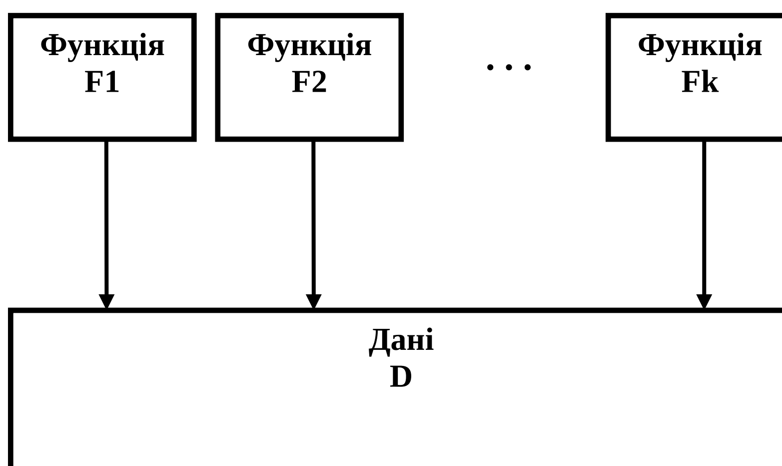

Рисунок 1.3 – Декомпозиція за задачами (функціями, методами)

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 задачах 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 задачами (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 кінцеві автомати при надходженні на їхні входи лише нульових значень (автономні кінцеві автомати) [8]. Можна розрізняти прямі автономні кінцеві автомати, які функціонують при зміні часу з «теперішнього» на «майбутній», та обернені автономні кінцеві 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].

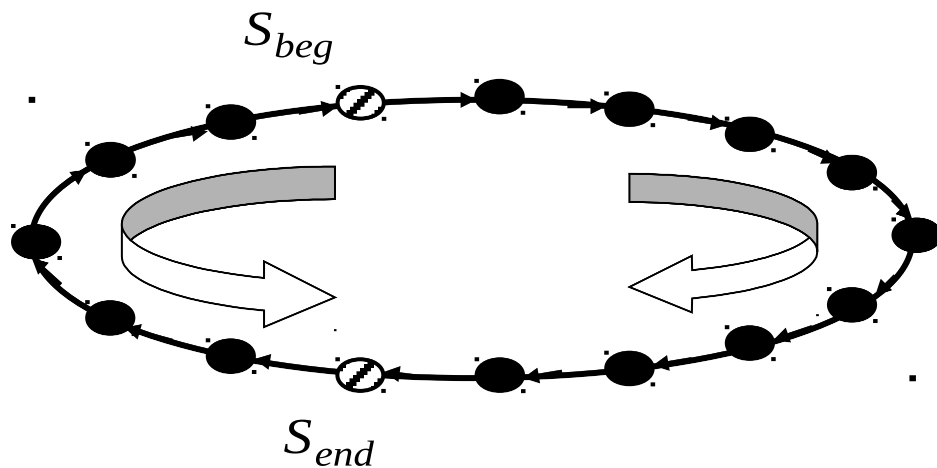

Рисунок 1.4 – Паралелізм за часом виконання для паралельних автономних автоматів

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 були детально розглянуті. 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 компіляції 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 процес у Windows представлений блоком процесу, (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.

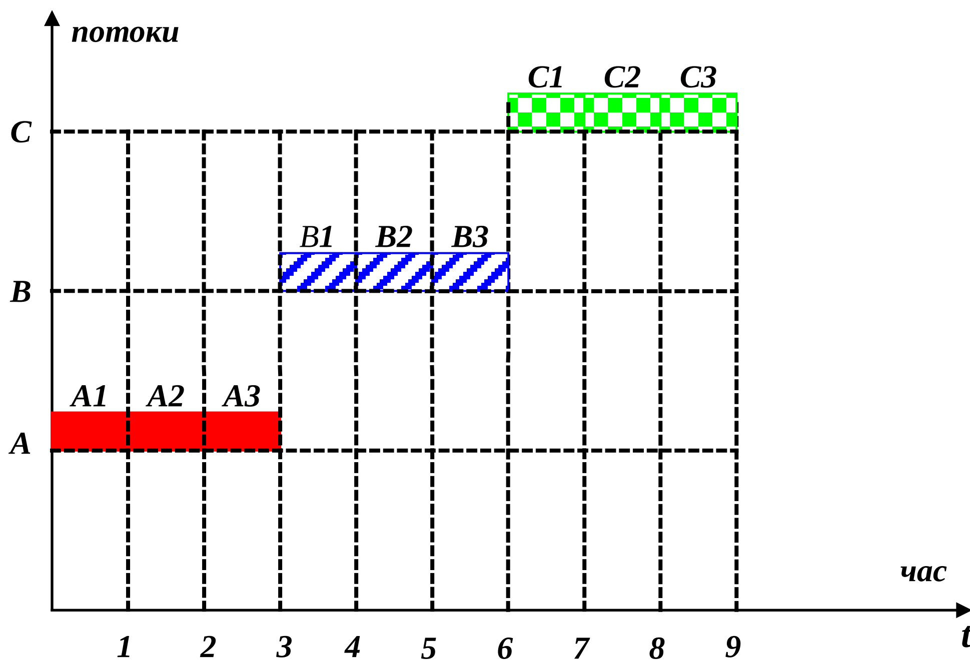

Рисунок 1.5 – Послідовне виконання задач

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.

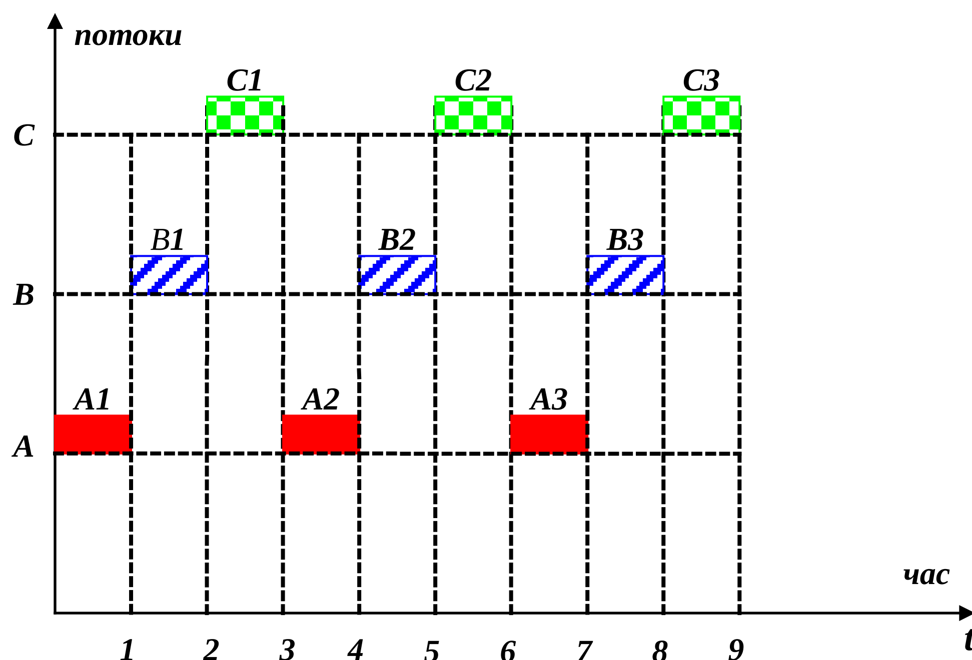

Рисунок 1.6 – Паралельне потактне виконання задач

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 рухомою 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 багато-потоковістю (*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.

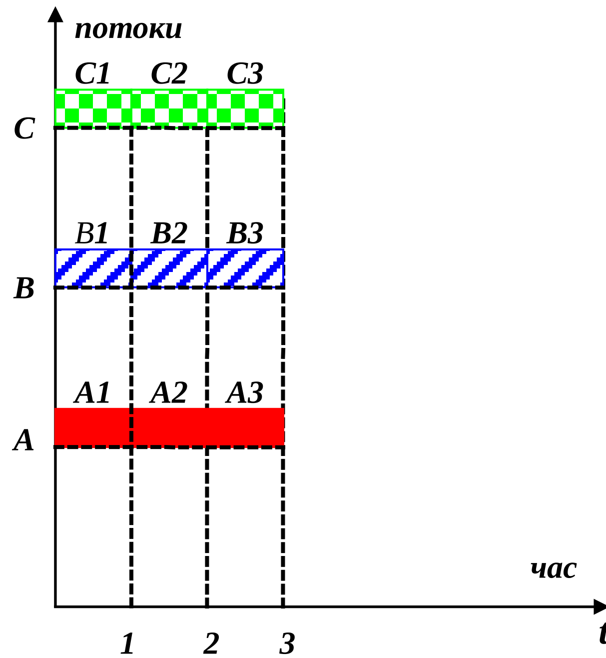

Рисунок 1.7 – Паралельне одночасне виконання задач

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)?

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 __.__.2018 р.
Формат 29,7×42¼. Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. ___.
Наклад 50 (1-й запуск 1–20) пр. Зам. № 2018-___.

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.