ДВНЗ «Ужгородський національний університет»
Факультет інформаційних технологій

В. М. Коцовський

Теорія паралельних обчислень

Навчальний посібник

Ужгород – 2021

УДК 004.04 + 519.6
К75

Рецензенти:

Завілопуло А. М. заступник директора ІЕФ НАН України, доктор фізико-математичних наук, професор;

Тилищак О. А., завідувач кафедри алгебри ДВНЗ «Ужгородський національний університет», доктор фізико-математичних наук, доцент.

Коцовський В. М.

К75 Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. 188 с.

У навчальному посібнику наведено основні поняття та моделі теорії паралельних обчислень, а також розглянуто реалізацію парадигми багатопотокового програмування у сучасних програмних платформах. Видання містить стислі теоретичні відомості та фрагменти паралельного програмного коду, написаного на мовах C++, Java та C#. Посібник може бути використаний студентами спеціальності 121 «Інженерія програмного забезпечення» у процесі виконання лабораторних робіт, а також для підготовки до модульного та семестрового контролів.

Рекомендовано до друку
Вченою Радою
ДВНЗ «Ужгородський національний університет»
(протокол № 7 від 23 червня 2021 р.)

Рекомендовано до друку
редакційно-видавничою радою
ДВНЗ «Ужгородський національний університет»
(протокол № 4 від 22 червня 2021 р.)

ISBN

ЗМІСТ

ВСТУП

Навчальна дисципліна «Теорія паралельних обчислень» вивчається студентами спеціальності 121 «Інженерія програмного забезпечення» на першому курсі магістратури. Актуальність вивчення основних понять теорії паралельних обчислень спеціалістами в галузі програмної інженерії зумовлена наявністю численних практично важливих задач великої розмірності, розв'язання яких потребує паралельного використання значної кількості розподілених обчислювальних ресурсів [1, 2]. Тому важливою складовою процесу підготовки фахівців сфери IT є вироблення компетенцій та навичок, які стосуються розуміння та використання сучасних методів та технологій паралельного програмування.

У посібнику навчальний матеріал розбитий на 10 розділів. У перших чотирьох розділах викладено базові поняття теорії паралельних обчислень, і включений у них матеріал в основному повторює [1, 3]. У першому розділі наведено основні означення теорії паралельних обчислень. У другому розділі розглянуто паралельну обробку даних і основні характеристики паралельних систем та описано найбільш популярні підходи до класифікації таких систем. Завершується розділ описом принципів функціонування та оглядом можливостей глобальної технології GRID. У третьому розділі розглянуто основні моделі, які використовуються для опису паралельних систем і використовують поняття графа алгоритму [4], проаналізовано концепцію необмеженого паралелізму та наведено приклади виявлення і використання внутрішнього паралелізму у «звичайних» послідовних алгоритмах.

Навчальний матеріал, який стосується основних понять та парадигм паралельного програмування, стисло наведено у розділах 4–6. Обсяг та вибір матеріалу цих розділів співставний з [2] та [5]. Розділи 7 та 8 присвячені таким механізмам синхронізації, як семафори та монітори. При цьому використовується опис алгоритмів синхронізації з використанням моделей та позначень, запропонованих у [6]. У 9-му розділі розглянуто програмування з використанням потоків у середовищі Java та задач (*tasks*) на платформі .NET Framework. Десятий розділ присвячений особливостям використання технології OpenMP для реалізації парадигм конкурентного програмування для здійснення паралельних обчислень.

Детальний опис та приклади використання паралельних систем, а також опис технологій, які не увійшли у посібник, можна знайти в [4, 6–13] або на мережевих ресурсах [18–26]. Для розуміння матеріалу посібника необхідним є знання дискретних моделей та методів у обсязі [15], а також володіння навичками структурного та об'єктно-орієнтованого програмування на мовах C++, Java та C# [15–17]. Умови лабораторних робіт до курсу наведені у [14] та за посиланням [27].

1. ПОНЯТТЯ ПРО ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

1.1. Послідовні обчислення

У процесі розробки та використання програмного забезпечення (ПЗ) для *послідовних обчислень* (*serial computation*):

- Задача розбивається на дискретну послідовність інструкцій (операторів).
- Інструкції виконуються послідовно одна за одною.
- Код програми виконується на єдиному процесорі (single processor).
- В кожний момент часу може виконуватися тільки одна інструкція.

Схема послідовних обчислень наведена на рис. 1.1.

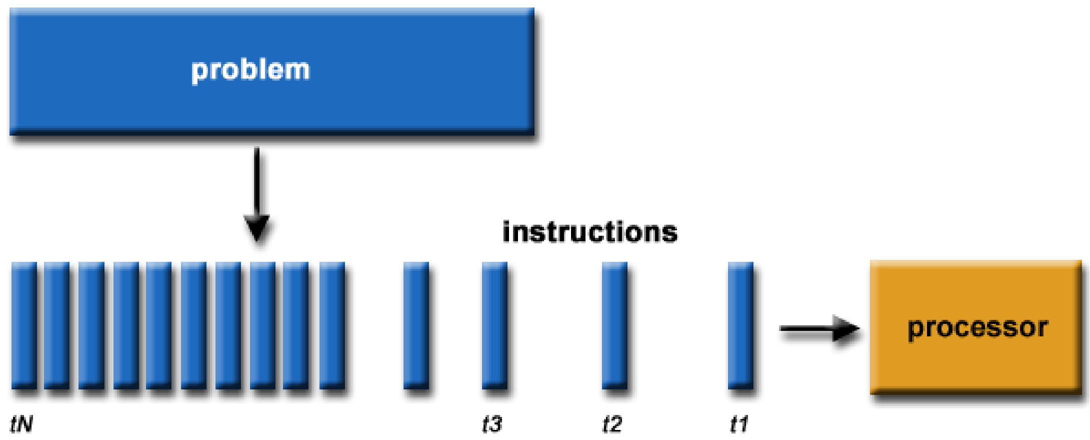

Рис. 1.1. Схема послідовних обчислень

Приклад 1.1. Розглянемо процедуру нарахування заробітної плати працівників підприємства з використанням послідовного підходу. Відповідна схема наведена на рис. 1.2.

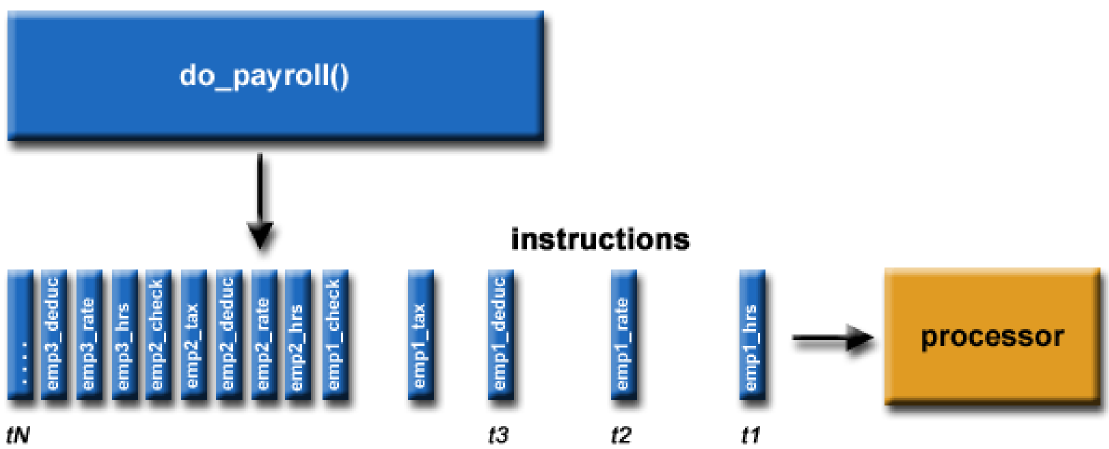

Рис. 1.2. Схема обчислення зарплати

1.2. Паралельні обчислення

Паралельні обчислення (ПО) — одночасне використання кількох ресурсів ЕОМ для розв’язування обчислювальних задач:

- Задача розбивається на підзадачі, які можуть виконуватися у один і той самий момент часу.
- Кожна підзадача в свою чергу розбивається на послідовність інструкцій.
- Інструкції кожної підзадачі виконуються одночасно на різних процесорах.
- У процесі обчислень використовується загальний механізм контролю-координації.

Схема паралельних обчислень наведена на рис. 1.3.

Обчислювальна задача має:

- Допускати розбиття на незалежні підзадачі, які можна виконувати одночасно (допускати паралелізм).
- З використанням кількох обчислювальних ресурсів, працюючих паралельно, розв’язуватися за коротший проміжок часу, ніж з використанням одного процесора.

Паралелізм — це сукупність математичних, алгоритмічних, програмних і апаратних засобів, що забезпечують можливість паралельного виконання задачі.

Типові обчислювальні ресурси, які використовують у ПО:

- Один комп’ютер з кількома процесорами.
- Довільна кількість комп’ютерів з’єднаних у мережу.

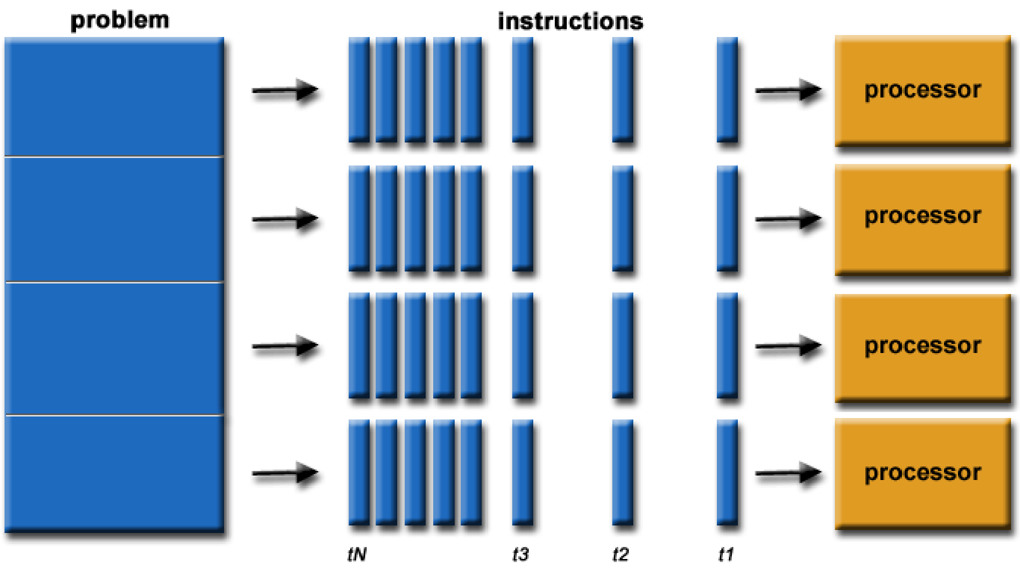

Рис. 1.3. Схема паралельних обчислень

Приклад 1.2. Розглянемо процедуру нарахування заробітної плати працівників підприємства з використанням паралельного підходу. Відповідна схема наведена на рис. 1.4.

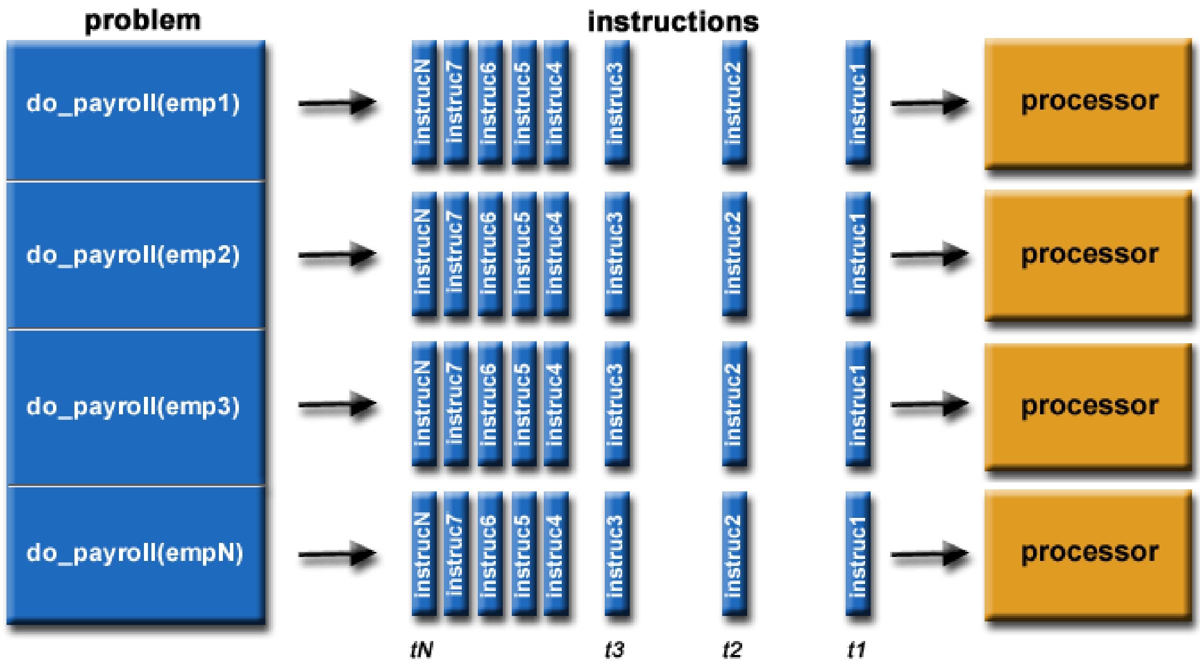

Рис. 1.4. Схема паралельного обчислення зарплати

1.3. Засоби для здійснення паралельних обчислень

Засоби, які дають змогу втілити парадигму паралелізму, можна класифікувати таким чином:

- Апаратні:
  - засоби для проведення обчислень (обчислювальна техніка):
    - обчислювальна техніка, зібрана з стандартних комплектуючих;
    - обчислювальна техніка, зібрана з спеціальних комплектуючих;
  - засоби візуалізації;
  - засоби для зберігання і обробки даних.
- Програмні:
  - програмні засоби загального призначення (операційні системи, стандартні бібліотеки, мови програмування, компілятори, профайлери, дебагери і т. п.);
  - спеціальні програмні засоби: бібліотеки (PVM, MPI); засоби об'єднання ресурсів (Dynamite, Globus та інші).

1.4. Паралельні комп'ютери

Паралельні комп'ютери поділяються на:

- Сучасні автономні комп'ютери. Вони є паралельними з точки зору їх внутрішньої будови:
  - Наявність численних функціональних модулів (L1-кеш, L2-кеш, пристрої попереджувального вибору команд (prefetch), decode, floating-point,

графічні процесори (GPU), модулі для цілої та дійсної арифметики тощо);

- Багатоядерність чи багатопроцесорність;
- Підтримка використання потоків (Multiple hardware threads).

* Кластери, які складаються із кількох робочих станцій, з'єднаних мережею (див рис. 1.5).

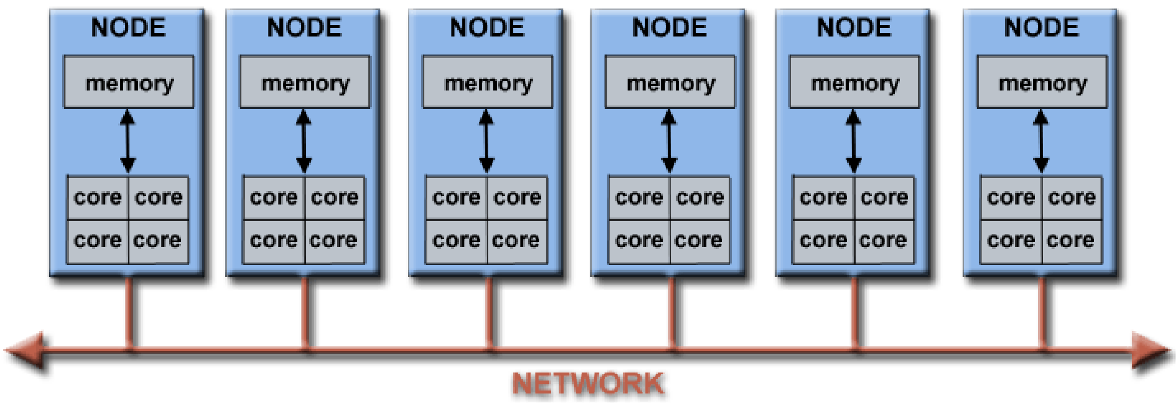

Рис. 1.5. Схема кластера

Наприклад, на рис. 1.6 наведено схему типового кластера для паралельних обчислень, які використовуються в Lawrence Livermore National Laboratory.

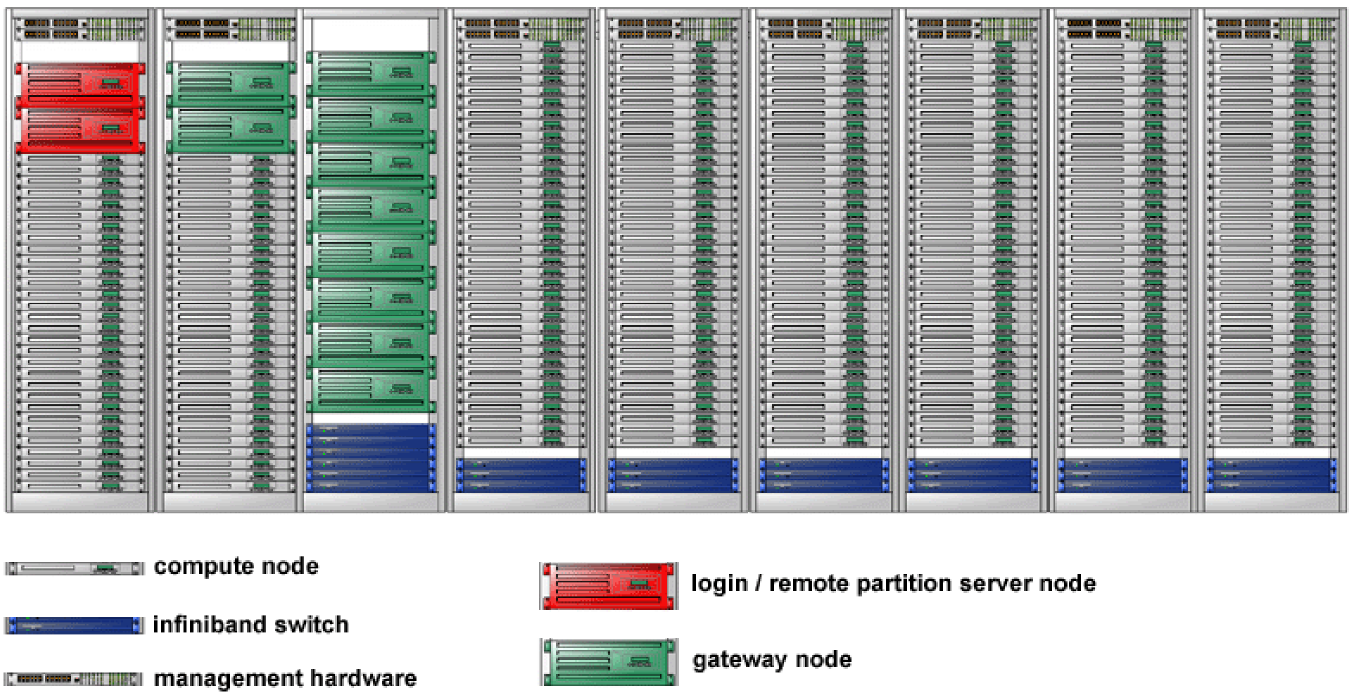

Рис. 1.6. Схема будови кластера LLNL

Властивості:

* Кожний обчислювальний вузол сам є багатопроцесорним комп'ютером.

- Вузли кластера з’єднані у Infiniband мережу.
  - Кластер містить багатопроцесорні вузли спеціального призначення, які не використовуються для обчислень

Дані про найбільші у світі паралельні комп’ютери (суперкомп’ютери), переважна більшість яких є кластерами, наведено на сайтс Top500.org.

1.5. Актуальність використання паралельних обчислень

Важливість дослідження та використання ПО зумовлена наступними причинами:

а) *Явища у реальному світі відбуваються паралельно*.

Тому ПО є значно більш придатними для моделювання складних взаємопов’язаних об’єктів, явищ, систем та процесів порівняно із послідовними обчисленнями. Приклади деяких таких об’єктів наведені на рис. 1.7.

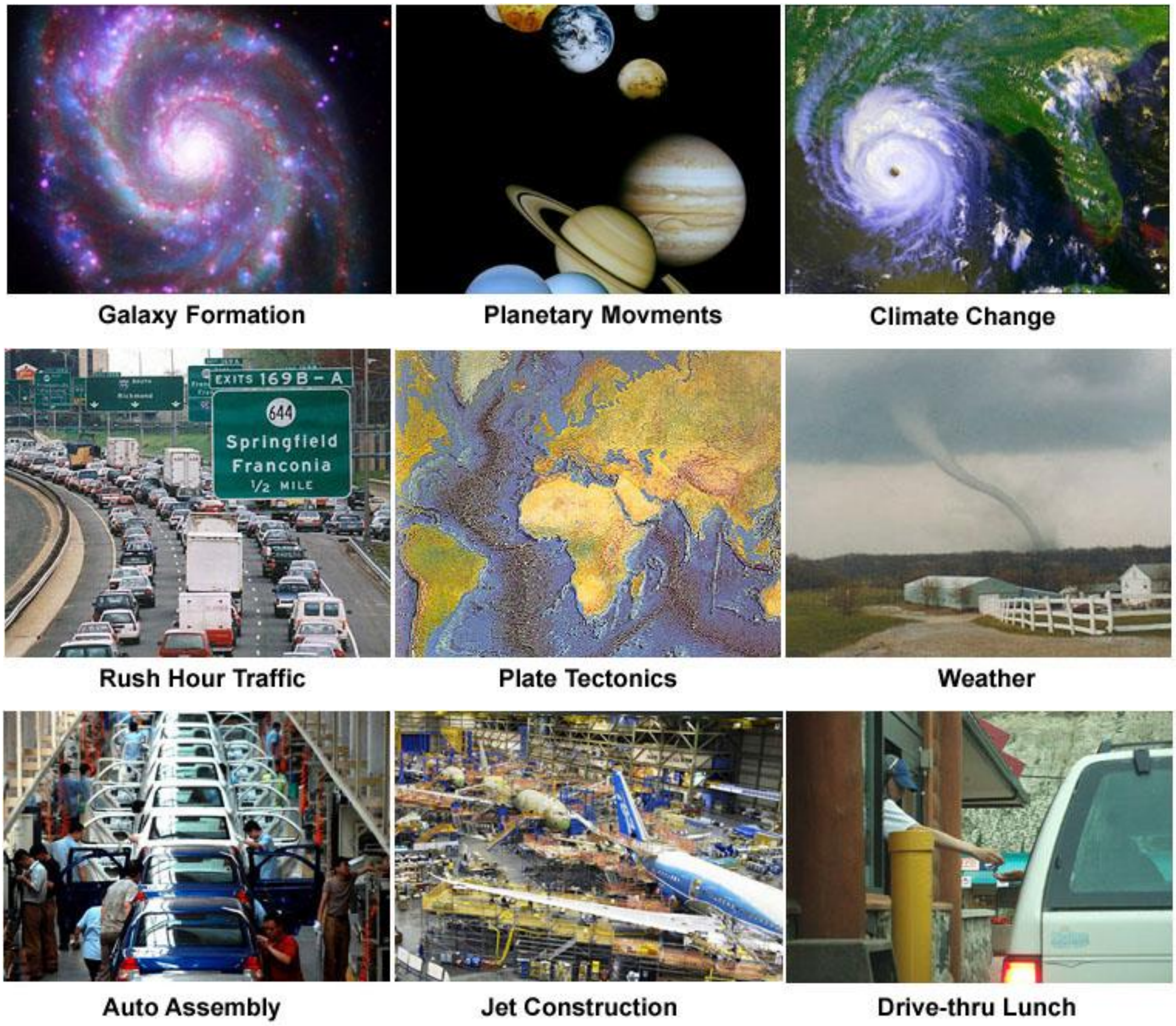

Рис. 1.7. Складні процеси, моделювання яких вимагає застосування паралельних обчислень

б) Економія часу та грошей.

Використання значних обчислювальних ресурсів може пришвидшити процес знаходження розв'язків складних задач, причому виграш від оперативності часто перевищує вартість використання додаткових ресурсів

в) Паралельні комп'ютери можуть бути побудовані із дешевих зручних компонентів.

г) Можливість знаходження розв'язків складних практичних та теоретичних задач.

Добре відомо, що багато практично важливих задач мають настільки велику розмірність, що практично нереально їх розв'язати на одному персональному комп'ютері, особливо із обмеженою пам'яттю.

Прикладом таких задач є задачі із переліку "Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge), які вимагають використання PetaFLOPS і PetaBytes комп'ютерних ресурсів.

Ще один приклад: Web-пошуковики, які виконують мільйони транзакцій щосекунди, та задачі прогнозу погоди. Область розв'язку (атмосфера) розбивається на окремі просторові зони, причому для розв'язку часових змін обчислень в кожній зоні повторюється багато разів. Якщо об'єм зони рівний 1 км^3, то для моделювання 10 км шару атмосфери необхідно 5 x 10^8 таких зон. Припустимо, що обчислення в кожній зоні вимагає 200 операцій з плаваючою крапкою, тоді за один часовий крок необхідно виконати 10^{11} операцій з плаваючою крапкою. Для того, щоб провести розрахунок прогнозу погоди з передбаченням на 10 днів з 10-ти хвилинним кроком в часі, ЕОМ продуктивністю 100 Mflops (10^8 операцій з плаваючою крапкою за секунду) необхідно 10^7 секунд чи понад 100 днів. Для того, щоб провести розрахунок за 10 хв, необхідна ЕОМ продуктивністю 1.7 Tflops.

д) Забезпечення одночасності багатьох дій (concurrency).

Наприклад, корпоративні мережі дають можливість одночасно виконувати роботу багатьом працівникам.

е) Використання глобальних ресурсів:

Використання ресурсів LAN або Internet у випадку, коли локальні обчислювальні ресурси недостатні.

є) Краще використання паралельного апаратного забезпечення:

Сучасні ПЕОМ та гаджети мають паралельну багатоядерну архітектуру. Паралельне ПЗ значно краще використовує можливості цих пристроїв у зв'язку із врахуванням багатопотоковості та багатоядерності, ніж програми, розроблені у межах «послідовного підходу».

1.6. Перспективи використання паралельних обчислень

Згідно до прогнозів, наведених у [4], тренд останніх 20 років, який полягає у використанні швидких мереж, багатопроцесорних архітектур та розподілених архітектур, ясно вказує на те, що *майбутнє в технологіях комп'ютерних обчислень за паралелізмом.*

На рис. 1.8 можна переконатися, що за цей період спостерігається зростання продуктивності суперкомп'ютерів понад у 500000 разів. Є сподівання, що процес зростання продуктивності не буде значно сповільнятися у майбутньому. Зараз мова йде про Exascale-обчислення (Exascale Computing), де Exaflop = 10^{18} обчислень у секунду.

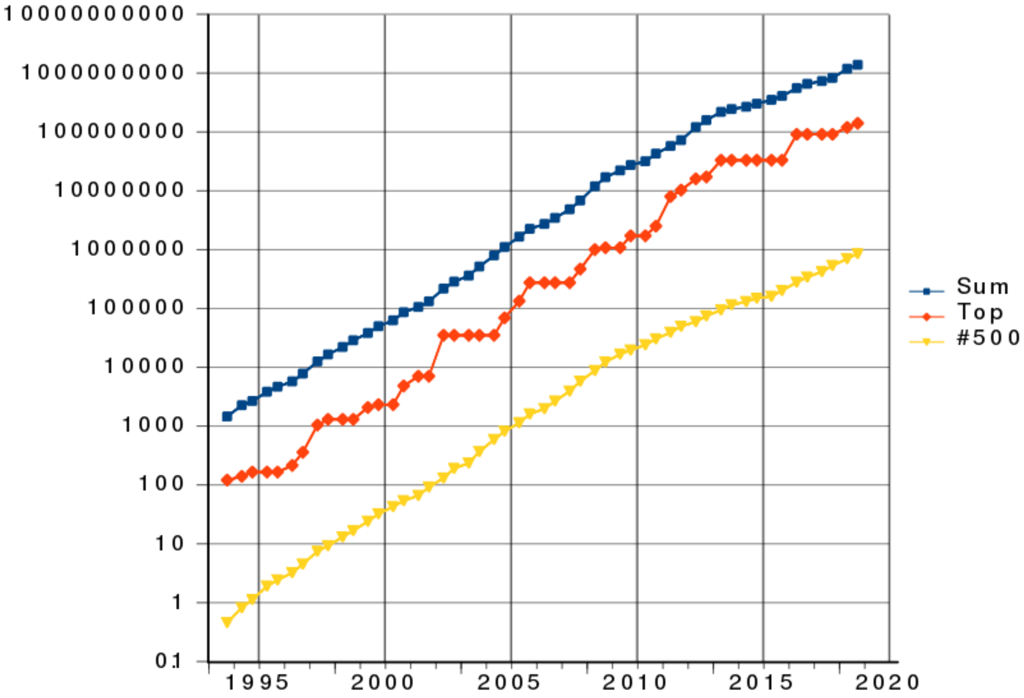

Рис. 1.8. Зростання продуктивності суперкомп'ютерів (у Teraflops за даними Top500.org)

Станом на листопад 2020 року рейтинг Top500 мав такий вигляд (за тестом LINPACK benchmarks):

1. Fugaku (Японія) 442 петафлопс, 7 630 848 ядер, 5 087 232 Гб
2. Summit (США) — 144,8 петафлопс
3. Sierra (США) — 94,6 петафлопс
4. Sunway TaihuLight (КНР) — 93,0 петафлопс
5. Selene (США) — 63,4 петафлопс
6. Tianhe-2A (КНР) — 61,4 петафлопс.

Summit — суперкомп'ютер розроблений IBM, Nvidia та Mellanox для національної лабораторії Оук-Ридж, який у 2018 р. став найпотужнішим комп'ютером у світі. 4608 серверів IBM Power Systems AC922 суперкомп'ютера займають площу, еквівалентну площі двох тенісних кортів. До складу цих серверів входять

9 тисяч 22-ядерних процесорів IBM POWER9 і більше 27 тисяч графічних процесорів NVIDIA Tesla V100. Загальний об'єм зовнішньої пам'яті — 250 PB. Summit споживає 15 МВт енергії, якої вистачило б на постачання 8100 середньостатистичних житлових будинків. Summit — перший комп'ютер, який подолав межу exaop (досягши 1,88 під час розв'язування задачі аналізу генома). Дослідники використовують Summit для розв'язування задач космології, медицини та кліматології. Сумарна вартість Summit та Sierra — 325 мільйонів доларів.

1.7. Сфери застосування паралельних обчислень

а) Наука та інженерія.

Історично, паралельні обчислення розглядалися як засіб для моделювання складних явищ та розв'язування різноманітних громіздких у таких галузях:

- науки про природу — моделювання атмосферних явищ, дослідження забруднення оточуючого середовища, задачі геології та сейсмології;
- фізика, астрономія та інженерія — розв'язування задач прикладної та ядерної фізики, електротехніки, фізики елементарних частинок, матеріалознавства, нанотехнологій, мікроелектроніки;
- біологія, хімія та медицина — дослідження у галузях біотехнологій, генетики, екології, популяційної біології, вірусології, фармакології та молекулярної хімії;
- математика та комп'ютерні науки.

б) Індустрія та комерція.

Індустріальні та комерційні застосування є основним драйвером розвитку інформаційних технологій. Паралельні обчислення використовуються при розв'язуванні задач:

- розвідування корисних копалин;
- фінансового та економічного моделювання;
- обробки великих БД, Big Data, Data Mining;
- Web-пошуку, web-торгівлі;
- обробки медичних зображень та діагностування;
- обробки графіки, мультимедіа та віртуальна реальність;
- військової справи, розробки зброї.

2. ПАРАЛЕЛЬНІ ОБЧИСЛЮВАЛЬНІ СИСТЕМИ

2.1. Рівні розпаралелювання

Існують різні підходи до визначення типів паралелізму [4, 7].

2.1.1. Перший підхід до класифікації паралелізму

Класифікація паралельності за рівнями [4], що відрізняються показниками абстрактності розпаралелювання задач, наведена в табл. 2.1. Чим «глибше» рівень, який має ознаки паралельності, тим детальнішим буде розпаралелювання, що стосується елементів програми (інструкція, елементи інструкції тощо). Чим вище розміщено рівень абстракції, тим більші блоки має паралельність.

Таблиця 2.1. Рівні паралельності

Рівні	**Об'єкт обробки**	**Приклад системи**
Програмний	Робота/Задача	Мультизадачна ОС
Процедурний	Процес	MIMD-система
Рівень формул	Інструкція	SIMD-система
Біт-рівень	В межах інструкції	Машина фон Неймана

Кожний рівень має різні аспекти паралельної обробки. Методи і конструктиви рівня обмежуються тільки самим рівнем і не можуть бути поширені на інші рівні. Найбільший інтерес становить рівень процедур (великоблокова, асинхронна паралельність) та рівень арифметичних виразів (малоелементна, детальна або масивна синхронна паралельність).

Програмний рівень

На цьому найвищому рівню одночасно (або принаймні розподілено за часом) виконуються комплектні програми (рис. 2.1). Машина, що виконує ці програми, *не повинна бути паралельною ЕОМ*, досить того, що в ній наявна багатозадачна операційна система (наприклад, реалізована як система *розподіленого часу).* В цій системі кожному користувачеві відповідно до його пріоритету *планувальник* (scheduler) виділяє відрізок процесорного часу різної тривалості. Користувач одержує ресурси центрального процесорного блоку тільки впродовж короткого часу, а потім стає в чергу на обслуговування.

У тому випадку, коли в ЕОМ недостатня кількість процесорів для всіх користувачів (або процесів), що, як правило, найбільш імовірно, в системі моделюється паралельне обслуговування користувачів за допомогою «квазіпаралельних» процесів.

Рис. 2.1. Паралельність на програмному рівні

Рівень процедур

На цьому рівні різні частини однієї і тієї самої програми мають виконуватися паралельно. Ці частини називаються «*процесами*» і приблизно відповідають послідовним процедурам. При проектуванні ПЗ задачі розбиваються на майже незалежні одна від інших частини так, щоб по можливості рідше виконувати операції обміну даними між процесами, які потребують відносно великих витрат часу. Цей рівень паралельності ні в якому разі не обмежується розпаралелюванням послідовних програм. Існує велика кількість задач, які потребують паралельних структур цього типу навіть тоді, коли так само, як і на програмному рівні, у користувача є тільки один процесор.

Основне застосування процедурного рівня — загальна паралельна обробка інформації, де застосовується поділ задач на паралельні підзадачі, які розв'язуються на багатопроцесорній системі з метою підвищення обчислювальної продуктивності. Відповідний приклад наведено на рис. 1.4.

Рівень формул

Арифметичні вирази виконуються паралельно покомпонентно, причому в значно простіших синхронних методах. Якщо, наприклад, йдеться про додавання 2×2-матриць, то воно синхронно розпаралелюється дуже просто тому, що кожному процесору відповідає один елемент матриці-результату.

При застосуванні n^2 процесорних елементів можна одержати суму двох матриць порядку n за час, необхідний для виконання однієї операції додавання (за винятком часу, потрібного на зчитування та запис даних). Цьому рівню притаманні засоби векторизації та так званої *паралельності даних*.

Рівень двійкових розрядів

На рівні розрядів (instruction-level parallelism) відбувається паралельне виконання двійкових операцій в межах одного машинного слова. Паралельність на рівні бітів присутня в будь-якому мікропроцесорі. Наприклад, у 64-розрядному арифметико-логічному пристрої побітова обробка виконується паралельними апаратними засобами.

2.1.2. Другий підхід до класифікації паралелізму

В [7] визначені наступні типи паралелізму: *паралелізм на рівні бітів, команд, даних* та *задач*.

Паралелізм на рівні команд заснований на використанні *конвеєрів* та *суперскалярних процесорів* для збільшення продуктивності виконання складних операцій. Конвеєри, у яких команди діляться на мікрокоманди, які можуть виконуватися одночасно, описані у п. 2.2.2.

При виконанні однієї інструкції скалярний процесор обробляє одне або два числа (скаляра). Суперскалярні процесори — процесори, які для виконання операцій використовують не один, а кілька однотипних конвеєрних функціональних пристроїв і можуть обробляти масив чисел (векторна обробка). Сучасні процесори є суперскалярними та містять спеціалізовані ALU- та FPU-блоки для цілочислової арифметики, операцій з плаваючою крапкою, розгалужень, завантаження даних тощо.

У сучасних процесорів є SIMD-команди (Single Instruction Many Data), які можуть використовуватися для масивів даних. Одними з перших машин, у яких було реалізовано паралелізм даних, були векторно-конвеєрні комп'ютери родини CRAY. Приклад коду на мові С для обчислення покомпонентної суми двох цілочислових масивів довжини 4 (попередньо потрібно підключити бібліотеку `smmintrin.h`).

```
    int x[4] = {1, 2, 3, 4};
    int y[4] = {5, 6, 7, 8};
    int z[4];
*(__m128i*)z = _m_add_epi32(*(__m128i*)x, *(__m128i*)y);
```

2.2. Способи обробки даних в обчислювальних системах

2.2.1. Послідовна обробка даних

Припустимо, що потрібно знайти суму **c** двох векторів **a** та **b**, кожний з яких має 100 дійсних координат, з використанням обчислювального пристрою (або комп'ютера), який виконує додавання пари чисел за 5 тактів роботи і у процесі обчислень комп'ютер не може виконувати ніяких інших корисних дій. У таких умовах сума векторів може бути знайдена за 500 тактів.

Тепер припустимо, що є два така самі пристрої, які можуть працювати одночасно і незалежно один від іншого, і при цьому відсутні додаткові витрати ресурсів по отриманню пристроями вхідних даних та збереженням результатів. В такому випадку можна отримати шукану суму векторів вже за 250 тактів — тобто маємо подвійне прискорення.

У випадку використання 10 однакових пристроїв результат отримується за 50 тактів, а у загальному випадку система із *N* пристроїв витратить на обчислення суми приблизно 500 / N тактів. Схеми, які ілюструють процес обчислень, наведені в [1] та [3].

2.2.2. Конвеєрна обробка даних

Розглянемо шляхи покращення ефективності роботи системи із попереднього параграфу. Для цього можна використати форму запису дійсних чисел в пам’яті комп’ютера. Додавання чисел пов’язане виконанням таких мікрооперацій як порівняння та вирівнювання порядків, додавання мантис, нормалізація та т. п. Суттєвим є те, що у процесі обробки кожна мікрооперація задіяна тільки один раз і завжди у тій самій послідовності одна за іншою. Це означає, що якщо перша мікрооперація виконала свою роботу і передала результат другій, то для обробки поточної пари дійсних чисел вона більше не знадобиться, і отже, цілком може бути використання для обробки наступної пари аргументів.

Виходячи із попередніх міркувань, можна сконструювати пристрій наступним чином. Кожну мікрооперацію виділимо у окрему частину пристрою і розташуємо їх у порядку виконання. Після виконання першої мікрооперації перша частина передає свій результат другій частині, а сама отримує для обробки нову пару. Коли вхідні аргументи пройдуть через усі етапи обробки, на виході пристрою з’явиться результат виконання операції.

Такий спосіб організації обчислень має назву *конвеєрної обробки*. Кожна частина пристрою називається *стадією конвеєра*, а загальна кількість стадій — *довжиною конвеєра*.

Сучасні процесори мають багатоетапні конвеєри команд. Кожен етап (стадія) конвеєра відповідає іншій дії, що виконує процесор. Класичним прикладом процесора з конвеєром є процесор архітектури RISC (Reduced Instruction Set Computing), що має п'ять етапів: завантаження інструкції, декодування інструкції, виконання, доступ до пам'яті, та запис результату. Так, процесор Intel 80386 мав 5-стадійний конвеєр (див. рис. 2.2), Pentium 4 — конвеєр з 35 стадій.

Рис. 2.2. Стандартний п’ятикроковий конвеєр в машині RISC.
IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory access), WB (Register write back)

Припустимо, що для виконання операції додавання дійсних чисел спроектовано конвеєрний пристрій, який складається із п’яти стадій, які спрацьовують за один такт. Час виконання операції на конвеєрному пристрої рівний сумі часів спрацьовування усіх стадій конвеєра. Це означає, що одна операція додавання двох чисел триває п'ять тактів, тобто так само довго, як і на послідовному пристрої у попередньому прикладі.

Тепер розглянемо процес додавання двох векторів (рис. 2.3). Як і раніше, через п'ять тактів отримано суму елементів першої пари. Проте слід зазначити, що поряд із першою парою пройшли часткову обробку (на різній кількості стадій) і інші пари аргументів. Кожний наступний такт на виході конвеєрного пристрою буде з'являтися сума чергової пари координат вектора **c**. На виконання усієї операції знадобиться 104 такти, замість 500 тактів при використанні послідовних пристроїв — виграш у часі приблизно у п'ять разів.

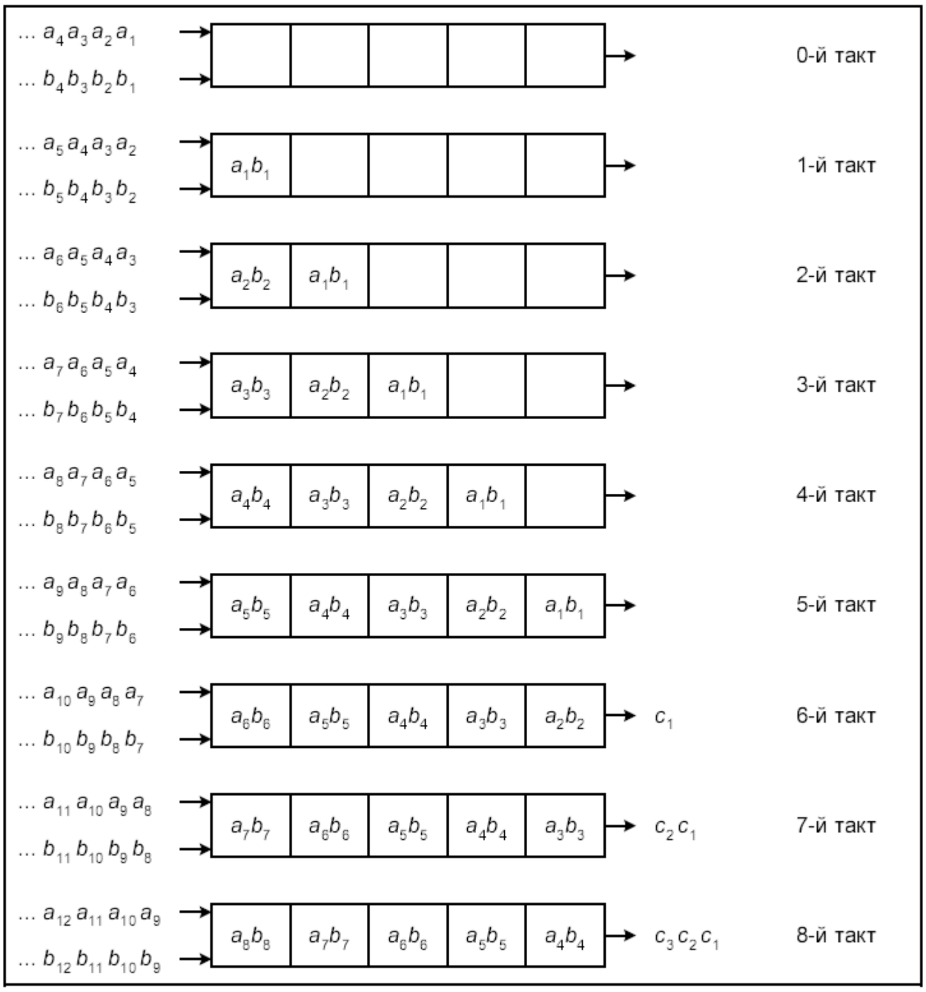

Рис. 2.3. Знаходження суми **c** = **a** + **b** за допомогою 5-стадійного конвеєрного пристрою

Якщо конвеєрний пристрій є l-стадійним і обробка даних на кожній стадії триває один такт, то для виконання n послідовних операцій на цьому пристрої потрібно витратити $l+n-1$ тактів. Якщо ж цей пристрій використовувати у послідовному режимі, то кількість тактів буде рівна $l \cdot n$. Отже, для великих n

отримуємо «прискорення» майже у l разів за рахунок використання конвеєрної обробки даних.

При використанні векторних команд у формулі для тривалості обробки даних на конвеєрному пристрої додається ще один доданок σ — це час (у тактах), необхідний для ініціалізації векторної команди. Тому загальний час рівний $\sigma + l + n - 1$.

Оскільки ні σ, ні l не залежать від значення n, то із збільшенням довжини вхідних векторів *ефективність конвеєрної обробки даних зростає*. Якщо під ефективністю обробки розуміти реальну продуктивність конвеєрного пристрою, рівну відношенню числа виконаних операцій n до часу їх виконання $t(n)$, то залежність продуктивності від довжини вхідних векторів визначається наступним співвідношенням:

$$\pi(n) = \frac{n}{t(n)} = \frac{n}{(\sigma + l + n - 1)\tau} = \frac{1}{\big(1 + (\sigma + l - 1)/n\big)\tau},$$

де τ — це тривалість такту роботи комп'ютера.

На рис. 2.4 наведено приблизний вигляд графіка цієї залежності.

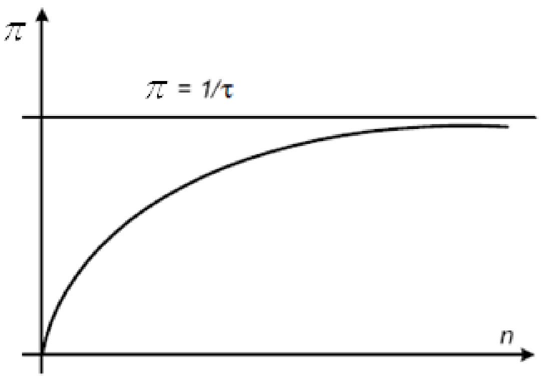

Рис. 2.4. Залежність продуктивності конвеєрного пристрою від довжини вхідного набору

Із зростанням довжини вхідних даних реальна продуктивність конвеєрного пристрою все більше наближається до його пікової продуктивності $1/\tau$. Однак *пікова продуктивність ніколи недосяжна на практиці*.

Розглянемо тепер модель конвеєрного пристрою, яка передбачає різну тривалість стадій. Нехай $t_1, t_2, \ldots, t_l$ — тривалості відповідних стадій конвеєра. Тоді перша операція обробки буде тривати $t_1 + \ldots + t_l$ тактів, а кожна наступна операція буде завершуватися через $t_{\max}$ тактів після попередньої, де $t_{\max} = \max\limits_{1 \le i \le l} t_i$. Тому загальна кількість тактів, необхідних для виконання n операцій, рівна $t_1 + \ldots + t_l + (n-1)t_{\max} + \sigma$. Тоді прискорення $S(n)$ при виконанні n операцій, яке досягається за рахунок використання конвеєрного способу обробки даних, визначається величиною

$$S(n)=\frac{n(t_1+\ldots+t_l)}{t_1+\ldots+t_l+(n-1)t_{\max}+\sigma}.$$

З останньої формули випливає, що граничне прискорення $S=\lim\limits_{n\to\infty}S(n)$ рівне

$$S=\frac{t_1+\ldots+t_l}{t_{\max}}.$$

Залежність продуктивності від довжини вхідних векторів визначається наступним співвідношенням:

$$\pi(n)=\frac{n}{(t_1+\ldots t_l+(n-1)t_{\max}+\sigma)\tau}.$$

Тому пікова продуктивність рівна $\dfrac{1}{t_{\max}\cdot\tau}$.

Приклад 2.1. Обробка даних на конвеєрному пристрої складається із 5 стадій, тривалості яких рівні 3, 5, 2, 6 та 4 такти відповідно. Виконати наступні завдання, вважаючи, що ініціалізація конвеєра потребує 2 тактів та тривалість одного такту складає 5 нс:

1) Обчислити кількість тактів, необхідну для виконання 1000 операцій обробки даних за умови, що пристрій працює:
   а) у послідовному режимі;
   б) у конвеєрному режимі.
2) Підрахувати пікову продуктивність системи.
3) Визначити найменшу кількість операцій, при виконанні яких у конвеєрному режимі досягається прискорення не менше за 90% від граничного прискорення.

Розв'язок. Запишемо характеристики конвеєрного пристрою:

$l=5,\ \sigma=2,\ \tau=5\cdot10^{-9}$ с, $t_1=3, t_2=5, t_3=2, t_4=6, t_5=4.$ Тоді

$$t_{\max}=\max\{t_1,\ldots,t_5\}=6.$$

1) Знайдемо шукану кількість тактів у випадку $n=1000$:
   а) у послідовному режимі:
   $t_s(n)=(t_1+\ldots+t_l)\cdot n;$
   $t_s(1000)=(3+5+2+6+4)\cdot1000=20000;$
   б) у конвеєрному режимі:

$$t_c(n) = (t_1 + \ldots + t_l) + (n-1)t_{\max} + \sigma;$$
$$t_c(1000) = 20 + 999 \cdot 6 + 2 = 6016.$$

2) Підрахуємо пікову продуктивність системи для обох режимів:

$$\pi_s = \frac{1}{(t_1 + \ldots + t_l)\tau} = \frac{1}{20 \cdot 5 \cdot 10^{-9}} = \frac{10^9}{100} = 10^7.$$

$$\pi_c = \frac{1}{t_{\max}\tau} = \frac{1}{6 \cdot 5 \cdot 10^{-9}} = \frac{10^9}{30} \approx 3.33 \cdot 10^7.$$

3) Визначимо кількість операцій n, яка задовольняє умову $S(n) \geq 0,9S$:

$$\frac{(t_1 + \ldots + t_l)n}{(t_1 + \ldots + t_l) + (n-1)t_{\max} + \sigma} \geq 0,9 \cdot \frac{t_1 + \ldots + t_l}{t_{\max}},$$

$$\frac{20n}{20 + 6(n-1) + 2} \geq 0,9\frac{20}{6},$$

$$\frac{n}{6n+16} \geq \frac{0.9}{6},$$

$$6n \geq (6n+16) \cdot 0,9$$

$$6n \geq 5,4n + 14,4$$

$$0.6n \geq 14,4$$

$$n \geq \frac{144}{6} = 24.$$

Відповідь: $n = 24$.

2.3. Характеристики систем функціональних пристроїв

Будь-яка обчислювальна система являє собою сукупність деяких функціональних пристроїв (ФП). Для оцінки якості її роботи вводяться різні характеристики.

Нехай задана система відліку часу і задана деяка одиниця часу, наприклад, секунда. Будемо вважати, що всі спрацьовування одно і того самого ФП системи мають однакову тривалість.

Назвемо ФП *простим*, якщо ніяка наступна операція не може виконуватися на ньому до тих пір, поки не виконається попередня. Основна властивість простого ФП — він монопольне використовує своє обладнання для виконання кожної окремої операції.

На відміну від простого, *конвеєрний* ФП розподіляє своє обладнання для одночасного виконання кількох операцій. Дуже часто (але необов'язково) конвеєрні ФП конструюються як лінійні ланцюжки простих ФП (стадій). Простий ФП можна завжди вважати конвеєрним ФП з довжиною конвеєра, рівною 1.

Назвемо *вартістю операції* час її реалізації, а *вартістю роботи* — сумарну вартість усіх виконаних операцій. *Завантаженістю* пристрою p на даному проміжку часу будемо називати відношення вартості реально виконаної роботи до максимально можливої вартості. Наступні два твердженню містять опис основних властивостей ФП та систем ФП.

Твердження 2.1. *Максимальна вартість, яку може виконати ФП за час T, рівна T для простого ФП та nT для конвеєрного ФП довжини n.*

Будемо називати *реальною продуктивністю* системи пристроїв кількість операцій, які реально виконуються у середньому за одиницю часу. *Піковою продуктивністю* називають максимальну кількість операцій, які могла би виконати система за одиницю часу у випадку відсутності звя'зків між її ФП. З визначення випливає, що *реальна (пікова) продуктивність системи рівна сумі реальних (пікових) продуктивностей ФП*, які входять до її складу.

Твердження 2.2. *Якщо система складається із l пристроїв, які мають пікові продуктивності* $\pi_1,\ldots,\pi_l$ *і працюють із завантаженістю* $p_1,\ldots,p_l$, *то реальна продуктивність системи r обчислюється за формулою*

$$r=\sum_{i=1}^{l} p_i\pi_i\,. \tag{2.1}$$

Розглянемо тепер, яким чином визначається завантаженість системи пристроїв. Якщо пристрої мають пікові продуктивності $\pi_1,\ldots,\pi_l$ і працюють із завантаженістю $p_1,\ldots,p_l$, то будемо вважати за означенням, що *завантаженість системи* є величина

$$p=\sum_{i=1}^{l}\alpha_i p_i\,, \tag{2.2}$$

де

$$\alpha_i=\frac{\pi_i}{\sum_{j=1}^{l}\pi_j},\ \left(i=1,\ldots,l\right).$$

Завантаженість системи є зваженою сумою завантаженості окремих пристроїв, так як із визначення коефіцієнтів α_i випливає, що

$$\sum_{i=1}^{l}\alpha_i=1,\ \alpha_i\ge 0,\ \left(i=1,\ldots,l\right) \tag{2.3}$$

Тому для завантаженості системи p виконуються нерівності $0\le p\le 1$. Крім того, з (2.2) та (2.3) випливає, що завантаженість системи рівна 1 тоді і тільки тоді, коли завантаженості окремих пристроїв системи рівні 1.

Слід зазначити, що визначення завантаженості системи згідно до (2.2) узгоджується із формулою реальної продуктивності (2.1). Дійсно, оскільки *пікова продуктивність π системи пристроїв* рівна $\pi_1 + \ldots + \pi_l$, то згідно до (2.1) та (2.2) справджується рівність

$$r = p \cdot \pi . \tag{2.4}$$

Велика кількість ФП, так само як і конвеєрні ФП, використовуються тоді, коли виникає потреба розв'язати задачу швидше. Для того, щоб зрозуміти, наскільки швидше це вдається зробити, потрібно увести у розгляд поняття «прискорення». Як і у випадку завантаженості це можна зробити по-різному. Розглянемо один із способів визначення прискорення. Будемо порівнювати швидкість роботи системи із швидкістю найпродуктивнішого пристрою системи. Відношення $S = r / \max \pi_i$ будемо називати *прискоренням реалізації алгоритму* на даній обчислювальній системі або просто *прискоренням*. Тобто,

$$S = \frac{\sum_{i=1}^{l} p_i \pi_i}{\max\limits_{1 \le i \le l} \pi_i} . \tag{2.5}$$

Аналіз формули (2.5) показує, що прискорення обчислювальної системи, яка складається із l пристроїв, не може перевищувати l і може досягати l тоді і тільки тоді, коли усі пристрої системи мають однакові пікові продуктивності і є цілком завантаженими.

Твердження 2.3. *Якщо система складається із l пристроїв (простих чи конвеєрних), які мають однакові пікові продуктивності, то*

- *завантаженість системи рівна середньому арифметичному завантаженості усіх пристроїв;*
- *пікова продуктивність системи у l разів більша за продуктивність одного пристрою;*
- *прискорення системи рівне сумі завантаженості усіх пристроїв.*

Одним із основних питань теорії обчислювальних систем є питання досягнення високого рівня ефективності. Із (2.4) випливає, що для цього потрібно досягти високого рівня завантаженості системи. Цього у свою чергу можна досягти шляхом підвищення завантаженості окремих пристроїв. Проте залишається відкритим питання, як можна це зробити. Якщо пристрій не є завантаженим на 100%, то завантаженість можна завжди підвищити тільки у тому випадку, якщо він не пов'язаний із іншими пристроями. В іншому випадку ситуація не є очевидною.

Без обмеження загальності будемо вважати, що усі пристрої є простими, тому що довільний конвеєрний пристрій можна зобразити у вигляді ланцюжка простих пристроїв. Припустимо, що між пристроями встановлено направлений зв'язки, які не змінюються у процесі функціонування. Побудуємо орієнтований граф, вершини якого взаємно однозначно відповідають пристроям, а дуги — зв'язкам між ними. З вершини A проведемо дугу у вершину B тоді і тільки тоді, коли результат роботи пристрою, якому відповідає вершина A, передається у якості вхідного аргументу пристрою, якому ставиться у відповідність вершина B. Назвемо отриманий граф *графом системи*.

Твердження 2.4 [4]. *Якщо система складається із l простих пристроїв, які мають пікові продуктивності* $\pi_1,\ldots,\pi_l$, *і граф системи є слабко зв'язним (відповідний йому неорієнтований граф є зв'язним), то максимальна продуктивність системи* $r_{\max}$ *виражається формулою*

$$r_{\max} = l \cdot \min_{1 \le i \le l} \pi_i .$$

Наслідок 2.1. В умовах твердження 2.4:

- *асимптотично усі пристрої виконують однакову кількість операцій;*
- *завантаженість кожного пристрою не перевищує завантаженість найменш продуктивного пристрою;*
- *якщо який-небудь пристрій завантажено повністю, то цей пристрій має найменшу продуктивність у системі;*
- *завантаженість системи не більша за число* $\dfrac{l \cdot \min_{1 \le i \le l} \pi_i}{\sum_{i=1}^{l} \pi_i}$;
- *прискорення системи не перевищує* $\dfrac{l \cdot \min_{1 \le i \le l} \pi_i}{\max_{1 \le i \le l} \pi_i}$

Наслідок 2.2 (перший закон Амдала). *Продуктивність обчислювальної системи, яка складається із пов'язаних між собою пристроїв, у загальному випадку визначається найменш продуктивним пристроєм.*

Кажучи, що система працює з максимальною можливою реальною продуктивністю, маємо на увазі те, що у системі забезпечується такий розклад команд, який мінімізує простій ФП системи.

Наслідок 2.3. *Нехай система утворена простими пристроями і має зв'язний граф. Тоді асимптотична продуктивність системи буде максимальною, якщо усі пристрої мають однакові пікові продуктивності.*

Максимальна продуктивність системи може досягатися при різних режимах роботи. Зокрема, вона досягається при синхронному режимі із тактом, обернено пропорційним продуктивності найповільнішого ФП системи. Із наслідку 3 можна зробити висновок, що продуктивність системи покращується, якщо усі пристрої системи мають однакову продуктивність.

Приклад 2.2. Граф системи ФП наведений на рис. 2.5. Відомі пікові продуктивності пристроїв системи: $\pi_1 = 10,\ \pi_2 = 5,\ \pi_3 = 8,\ \pi_4 = 6,\ \pi_5 = 7,\ \pi_6 = 9,\ \pi_7 = 12,$ $\pi_8 = 8, \pi_9 = 10, \pi_{10} = 4,\ \pi_{11} = 6,\ \pi_{12} = 4,\ \pi_{13} = 6$. Знайти:

1) завантаженості усіх пристроїв системи;
2) реальну продуктивність системи;
3) завантаженість системи;
4) прискорення системи.

Розв'язок. Як видно з рис. 2.5, система складається з трьох незалежних підсистем. Згідно до 1-го закону Амдала реальна продуктивність кожної із підсистем визначається продуктивністю найменш продуктивного пристрою цієї підсистеми.

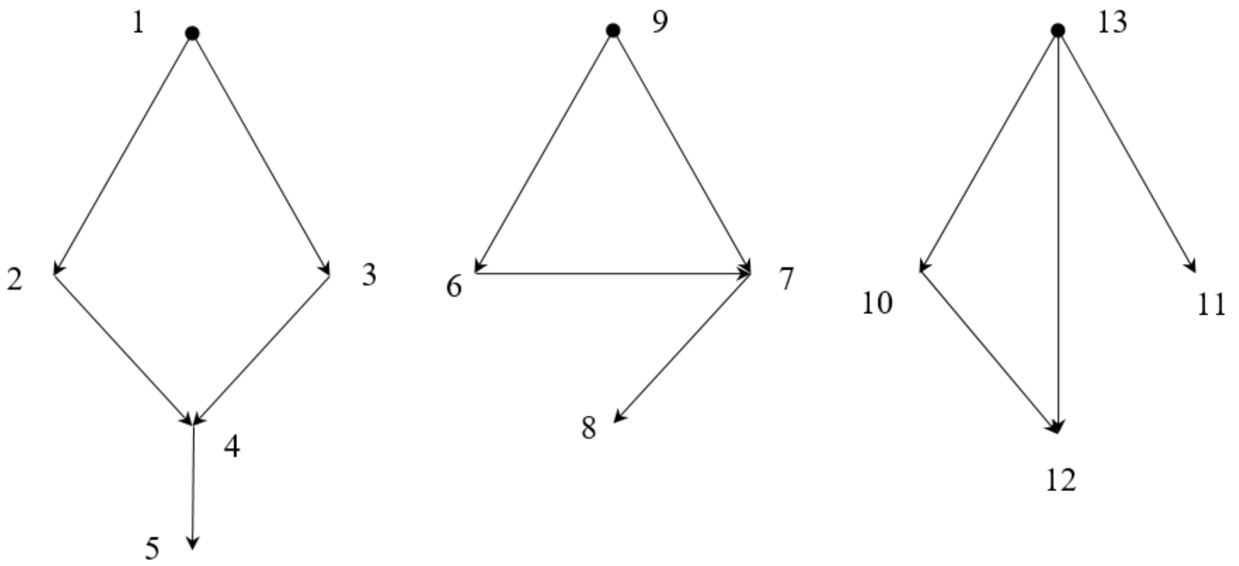

Рис. 2.5. Граф системи ФП

1) Нехай $\bar{r}^{(i)}$ — реальні продуктивності, з якими працюють усі пристрої i-системи, $i = 1,2,3$. Тобто,

$$r_1 = \ldots = r_5 = \bar{r}^{(1)},\ r_6 = \ldots = r_9 = \bar{r}^{(2)},\ r_{10} = \ldots = r_{13} = \bar{r}^{(3)}.$$

Тоді

$$\bar{r}^{(1)} = \min\{\pi_1, \ldots, \pi_5\} = 5,\ \bar{r}^{(2)} = \min\{\pi_6, \ldots, \pi_9\} = 8,\ \bar{r}^{(3)} = \min\{\pi_{10}, \ldots, \pi_{13}\} = 4.$$

Завантаженості p_i пристроїв першої підсистеми рівні $\bar{r}^{(1)}/\pi_i,\ (i = 1, \ldots 5)$:

$$p_1 = \frac{5}{10},\ p_2 = 1,\ p_3 = \frac{5}{8},\ p_4 = \frac{5}{6},\ p_5 = \frac{5}{7}.$$

Завантаженості p_i пристроїв другої підсистеми рівні $\overline{r}^{(2)}/\pi_i$, $(i=6,\ldots 9)$:

$$p_6=\frac{8}{9},\ p_7=\frac{2}{3},\ p_8=1,\ p_9=\frac{4}{5}.$$

Завантаженості пристроїв третьої підсистеми рівні $\overline{r}^{(3)}/\pi_i$, $(i=10,\ldots 13)$:

$$p_{10}=1,\ p_{11}=\frac{2}{3},\ p_{12}=1,\ p_{13}=\frac{2}{3}.$$

2) $r=r^{(1)}+r^{(2)}+r^{(3)}$, де $r^{(i)}$ — реальна продуктивність i-ї підсистеми, $i=1,2,3$. За першим законом Амдала (див. твердження 2.4) $r^{(i)}=l_i\cdot\overline{r}^{(i)}$, де l_i — кількість пристроїв і-ї підсистеми. Тому

$$r^{(1)}=5\cdot 5=25,\ r^{(2)}=4\cdot 8=32,\ r^{(3)}=4\cdot 4=16.$$

Отже, $r=25+32+16=73$.

3) Для знаходження завантаженості системи використаємо формулу $r=p\cdot\pi$, де p — завантаженість, π — пікова продуктивність системи. Тоді

$$\pi=\pi_1+\ldots+\pi_{13}=10+5+8+6+7+9+12+8+10+4+6+4+6=95.$$

Тому $p=r/\pi=73/95\approx 0,77$.

4) Скористаємося формулою $S=r/\max\limits_{1\le i\le l}\pi_i$. Тоді $S=73/12\approx 6,08$.

Припустимо, що усі пристрої системи є простими, універсальними (тобто на них можна виконувати різноманітні операції) та мають однакову продуктивність. Нехай у системі реалізується деякий алгоритм, а сама реалізація відповідає деякій його паралельній формі. Припустимо, що висота паралельної форми (кількість ярусів) рівна m, *ширина* (максимальна кількість вершин на одному ярусі) — q, а всього у алгоритмі виконується N операцій.

Твердження 2.5. *Для системи, яка задовольняє наведені вище умови, максимальне прискорення не більше за* N/m.

Наслідок 2.4. *Мінімальна кількість пристроїв системи, при якій може бути досягнуто максимально можливе прискорення, рівна ширині алгоритму.*

Припустимо, що у алгоритмі n операцій із N виконуються послідовно. Причини цього можуть бути різними. Наприклад, операції можуть бути пов'язані послідовними інформаційними зв'язками. Також цілком можливим є те, що при реалізації алгоритму просто не розпізнали паралелізм, наявний у відповідній його частині. Відношення $\beta=n/N$ назвемо *часткою послідовних обчислень*.

Наслідок 2.5 (другий закон Амдала). *Нехай система складається із l однакових простих універсальних пристроїв. Тоді максимальне можливе прискорення системи рівне*

$$S_l = \frac{l}{\beta l + (1-\beta)}.$$

Наслідок 2.6 (третій закон Амдала). *Нехай система складається із l однакових простих універсальних пристроїв. При будь-якому режимі роботи її прискорення менше за обернену величину до частки послідовних обчислень* $(S_l < 1/\beta)$.

Другий та третій закони Амдала проілюстровані на рис. 2.6. Величина $S_{\max} = 1/\beta$ називається *граничним прискоренням* системи для заданого алгоритму [1].

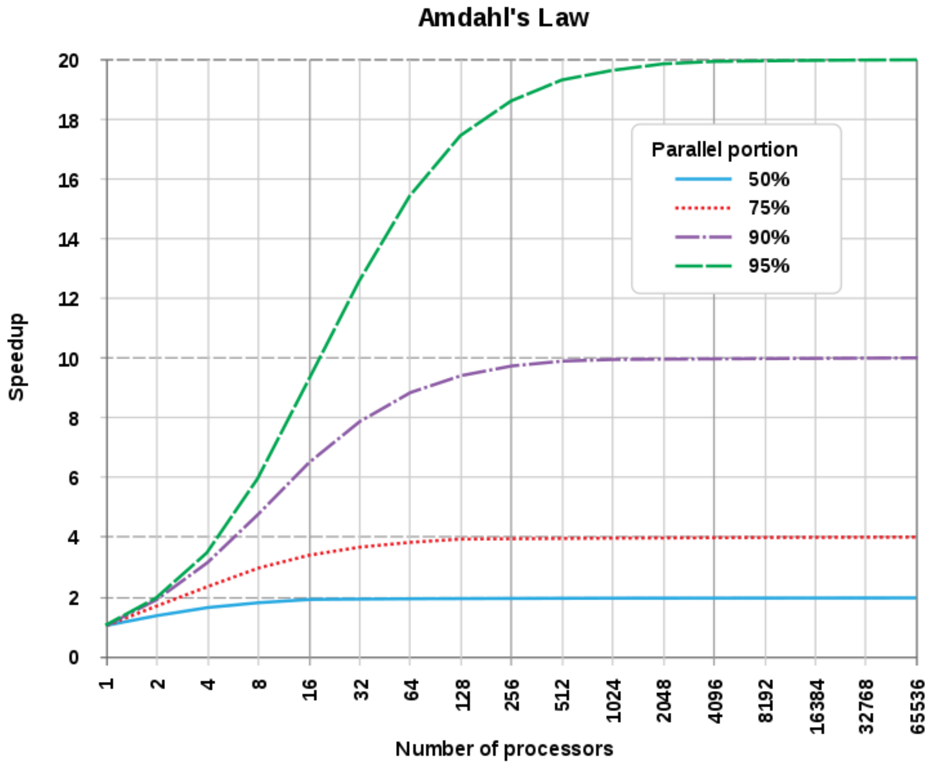

Рис. 2.6. Графік залежності прискорення від кількості пристроїв системи

Для системи, яка складається із l однакових простих пристроїв величина $E_l = S_l / l$ називається *ефективністю системи* (при реалізації заданого алгоритму) [7]. Легко переконатися, що для розглядуваних систем *ефективність співпадає із завантаженістю p*. Максимальні значення прискорення та ефективності рівні l та 1, відповідно.

Приклад 2.3. Визначити максимальне можливе прискорення і ефективність системи, яка складається з однакових пристроїв і призначена для реалізації алгоритму, граф якого наведений на рис. 2.7 (вершини графу відповідають операціям).

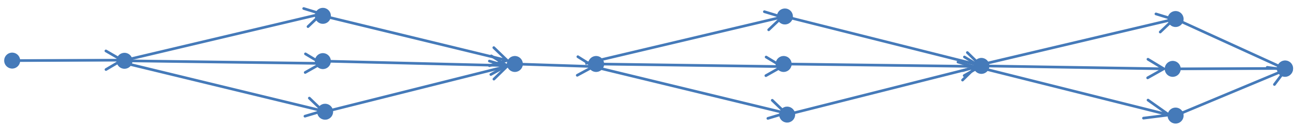

Рис. 2.7. Граф системи

Розв'язок. Використаємо 2-й закон Амдала. З рис. 2.7 можна зробити висновок, що, $N=15$, $n=6$. Звідси $\beta=6/15=2/5$. Ширина алгоритму — 3. Тому

$$S_3=\frac{3}{3\cdot 2/5+\left(1-2/5\right)}=\frac{3\cdot 5}{9}=\frac{5}{3},\quad E_3=\frac{S_3}{3}=\frac{5}{9}\approx 55,56\%.$$

Приклад 2.4. Нехай у алгоритмі паралельні обчислення складають 5/6. Визначити:

1) Максимальне можливе прискорення у випадку використання 5 однакових універсальних процесорів;
2) Мінімальну кількість процесорів, використання яких може забезпечити 85% граничного прискорення.

Розв'язок. Визначимо частку послідовних обчислень: $\beta=1-5/6=1/6$.

1) Скористаємося 2-м законом Амдала:

$$S_5=\frac{5}{5\cdot 1/6+1-1/6}=\frac{5\cdot 6}{10}=3.$$

2) Використаємо 3-й закон Амдала:

$S_l\geq 0,8\cdot S_{\max}$,

$\dfrac{l}{l/6+5/6}\geq\dfrac{1}{1/6}$,

$\dfrac{6l}{l+5}\geq 0,85\cdot 6$,

$l\geq 0,85\cdot\left(l+5\right)$,

$0,15l\geq 4,25$,

$l\geq 85/3\approx 28,33$

Відповідь: 29.

Задача 2.1. Нехай відомо, що для деякого алгоритму частка послідовних обчислень рівна β_1, частка обчислень, які допускають розпаралелення на k процесорів рівна — β_k, $k=2,\ldots,l$, $\beta_1+\ldots\beta_l=1$. Довести, що у випадку реалізації алгоритму у паралельній системі, яка складається з l однакових пристроїв, для максимального можливого прискорення S_l має місце формула:

$$S_l = \left(\beta_1 + \beta_2 / 2 + \ldots + \beta_l / l\right)^{-1}.$$

2.4. Класифікація паралельних обчислювальних систем

З попереднього матеріалу зрозуміло, що існує багато різних способів організації паралельних обчислювальних систем. Серед найбільш розповсюдженої архітектури можна вказати векторно-конвеєрні, масивно-паралельні та матричні системи, спецпроцесори, кластери, комп'ютери із багатопотоковою архітектурою тощо.

У зв'язку з різноплановістю розроблених систем виникла потреба класифікувати паралельні системи.

2.4.1. Класифікація Флінна

Ця класифікація архітектур була запропонована в 1966 р. М. Флінном і вважається першою і найбільш розповсюдженою класифікацією [4, 9]. Класифікація Флінна заснована на понятті потоку, під яким мається на увазі послідовність команд або даних, які опрацьовує процесор. На основі кількості потоків команд та даних Флінн вирізняє чотири класи архітектури.

SISD (Single Instruction stream / Single Data stream) — одиничний потік команд та одиничний потік даних, наведений на рис. 2.8 (ПР — процесор, ПД — пам'ять даних, УУ — пристрій керування).

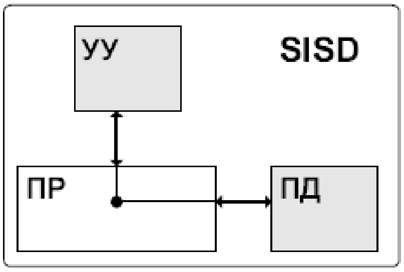

Рис. 2.8. Клас SISD класифікації Флінна

До класу SISD належать, перед усім, класичні послідовні машини із архітектурою фон Неймана, наприклад, PDP-11 або VAX 11/780. В таких машинах є тільки один потік команд, усі команди обробляються послідовно одна за одною і кожна з них породжує одну скалярну операцію. При цьому неважливо, що для збільшення швидкості обробки команд і швидкості арифметичних операцій може бути застосована конвеєрна обробка даних.

SIMD (Single Instruction stream / Multiple Data stream) — одиничний потік команд та множинний потік даних (рис. 2.9).

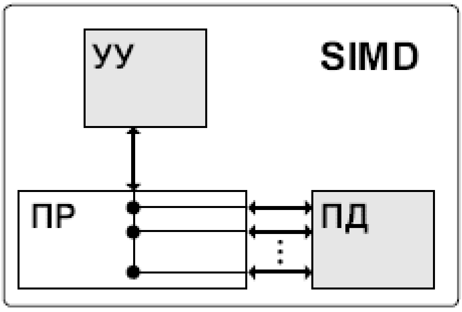

Рис. 2.9. Клас SIMD класифікації Флінна

У подібній архітектурі зберігається один потік команд, який включає, на відміну від попереднього класу, векторні команди. Це дає змогу виконувати арифметичні операції відразу з багатьма даними, наприклад елементами вектора. Спосіб виконання строго не фіксується. Він може бути реалізований або з використанням процесорної матриці, як у ILLIAC IV, або за допомогою конвеєра, як у машині Cray-1.

MISD (Multiple Instruction stream / Single Data stream) — множинний потік команд і одиночний потік даних (рис. 2.10).

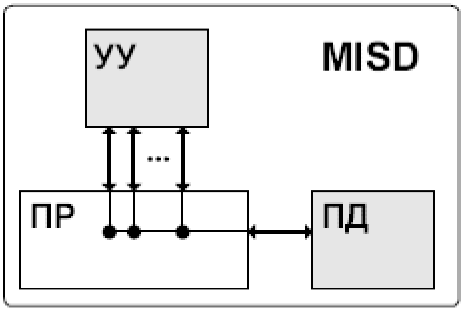

Рис. 2.10. Клас MISD класифікації Флінна

У визначенні мають на увазі, що наявність у архітектурі багатьох процесорів, які опрацьовують один і той самий потік даних. У [4] наведено аргументи на користь того, що даний клас потрібно вважати порожнім.

MIMD (Multiple Instruction stream / Multiple Data stream) — множинний потік команд та множинний потік даних (див. рис. 2.11). Цей клас містить обчислювальні системи, які мають кілька пристроїв обробки даних. Він є надзвичайно широким і, зокрема, містить різноманітні мультипроцесорні системи: Cm*, C.mmp, Cray Y-MP, Intel Paragon тощо. Якщо конвеєрну обробку розглядати як виконання послідовності різних команд (стадій конвеєра) не над одиночним векторним потоком даних, а над множинним скалярним потоком, то усі векторно-конвеєрні комп'ютери можна віднести до класу MIMD.

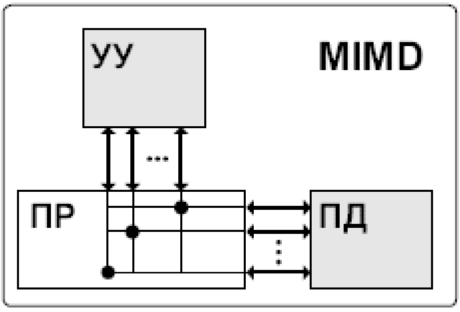

Рис. 2.11. Клас MIMD класифікації Флінна

Недоліком класифікації Флінна є те, що деякі важливі системи, наприклад dataflow та векторно-конвеєрні машини, чітко не вписуються у дану класифікацію. Інший недолік — надмірна наповненість останнього класу MIMD. Цей недолік подолано у класифікації Р. Хокні, який провів більш ретельну класифікацію машин класу MIMD [1].

2.4.2. Класифікація Фенга

Принципові інший підхід до класифікації був запропонований Т. Фенгом у 1972 році. Згідно до цього підходу класифікація проводиться по двом простим характеристикам. Перша — число n бітів у машинному слові, які опрацьовуються паралельно при виконанні машинних інструкцій. Майже для усіх сучасних машин це число співпадає із довжиною машинного слова. Друга характеристика рівна числу слів m, які одночасно обробляє дана обчислювальна система [4].

Кожну обчислювальну систему можна описати парою чисел (n,m). Добуток $P = n \times m$ визначає інтегральну характеристику потенціалу обчислювальної системи, яку Фенг назвав *максимальною ступеню паралелізму обчислювальної системи*. По сутті, це не що інше, як пікова продуктивність, виражена у інших одиницях.

Покажемо обчислення характеристик Фенга на прикладі комп'ютера Advanced Scientific Computer фірми Texas Instruments (TI ASC). У основному режимі він обробляє 64-розрядне слово, причому усі розряди опрацьовуються паралельно. Арифметично-логічний пристрій має чотири одночасно працюючих 8-стадійних конвеєрів. При такій організації $4 \times 8 = 32$ слова можуть оброблятися одночасно, і отже комп'ютер TI ASC може бути поданий у вигляді (64, 32).

На основі запропонованої Фенгом класифікації можна виокремити чотири класи комп'ютерів [3]:

1) Розрядно-послідовні, пословно-послідовні $(n=1,\ m=1)$. У кожний момент часу на таких машинах обробляється тільки один двійковий розряд.
2) Розрядно-паралельні, пословно-послідовні $(n>1,\ m=1)$. Більшість класичних послідовних комп’ютерів, так само як багато обчислювальних систем з описами $(16,1)$ або $(32,1)$, які існували до ери багатоядерних машин.
3) Розрядно-послідовні, пословно-паралельні $(n=1,\ m>1)$. Обчислювальні системи цього класу складаються із великої кількості однорозрядних процесорних елементів, кожний з яких працює незалежно від інших. Типовим прикладом є ICL DAP (1, 4096).
4) Розрядно-паралельні, пословно-паралельні $(n>1,\ m>1)$. Переважна більшість паралельних обчислювальних систем, які опрацьовують одночасно $n\times m$ двійкових розрядів, відноситься до цього класу: ILLIAC IV (64, 64), TI ASC (64, 32).

Недолік класифікації пов’язані зі способом обчислення числа *m*. При цьому Фенг ігнорує відмінність між процесорними матрицями, векторно-конвеєрними та багатопроцесорними системами.

Слід зазначити, що запропоновано значне число інших способів класифікації: Хендлера, Шнайдера, Скіллкорна [3, 4, 7] і т. д.

2.5. GRID та метакомп’ютинг

Усі перші паралельні системи належали потужним установам та корпораціям. Але у наш час ситуація різко змінилася. Обчислювальний кластер можна зібрати у більшості лабораторій, відштовхуючись лише від потреб у обчислювальній потужності та наявного бюджету. Для цілого класу задач, які не передбачають тісної взаємодії між паралельними обчислювальними процесами, рішення на базі звичайних робочих станцій та мережі Fast Ethernet є цілком ефективними.

Продовженням цього є ідея вважати будь-які пристрої паралельною обчислювальною системою, якщо вони працюють одночасно і їх можна використовувати для розв’язання однієї задачі. Способи організації паралельних обчислень та можливості системи можуть бути різні, але принципова можливість паралельних обчислень має бути присутня.

У цьому сенсі унікальні можливості надає мережа Інтернет, яку можна розглядати як найбільший у світі комп’ютер. Жодна обчислювальна система не може зрівнятися ні по піковій продуктивності, ні по об’єму оперативної чи дискової пам’яті із тими сумарними ресурсами, які мають комп’ютери, які підключені до мережі Інтернет. Звідси і походить спеціальна назва для процесу організації обчислень на такій системі — *метакомп’ютинг.* У принципі, необов’язково розглядати Інтернет як єдине можливе комунікаційне середовище метакоп’ютера, цю роль може виконувати будь-яка мережева технологія. У даному випадку головним є принцип функціонування, а технічних можливостей на даний час існує достатньо.

Перші прототипи реальних систем метакомп'ютингу з'явилися наприкінці 90-х років XX століття. У деяких системах використовуються високопродуктивні мережі та спеціальні протоколи, а десь за основу береться звичайні канали зв'язку та робота з протоколом HTTP. Приклади відповідних систем наведені у [1, 4].

Об'єднання у межах однієї мережі різноманітні пристроїв дає змогу сформувати спеціальне обчислювальне середовище. Певні комп'ютери можуть вмикатися чи вимикатися, але, з позиції користувача, це середовище є єдиним метакомп'ютером. Працюючи у такому середовищі, користувач лише формує запит та завдання на розв'язування задачі. Усе інше метакомп'ютер робить сам: шукає доступні обчислювальні ресурси, відслідковує їх працездатність, виконує передачу даних, виконує перетворення даних у потрібний формат тощо.

Описаний процес багато у чому аналогічний до електричної мережі. Вмикаючи чайник, користувач не замислюється над тим, яка мережа виробляє електроенергію. Користувачу потрібен лише ресурс, він його використовує. Саме за аналогією з електричною мережею розподілена обчислювальна система отримала в англомовній літературі назву "Grid" або обчислювальна мережа. Терміни Grid та метакомп'ютинг використовуються як синоніми.

Метакомп'ютер має цілий набір притаманних лише йому ознак [4]:

- ресурси метакомп'ютера *значно перевищують* ресурси звичайного комп'ютера по усім параметрам: кількість процесорів, об'єм пам'яті, кількість активних програм, користувачів тощо.
- метакомп'ютер *є розподіленим* за своєю природою. Його компоненти можуть буди віддаленими на сотні та тисячі кілометрів, що може впливати на оперативність та швидкість взаємодії.
- метакомп'ютер *може динамічно змінювати конфігурацію*. Але для користувача робота з метакомп'ютером повинна залишатися прозорою. Система керування метакомп'ютера має вміти шукати відповідні ресурси, перевірці їх працездатності та розподілі задач, що надходять у систему.
- метакомп'ютер *є неоднорідним*. При розподілі завдань потрібно враховувати особливості операційних систем, які входять до його складу та мають різні системи команд і формати даних.
- метакомп'ютер *об'єднує ресурси різних установ*, політика доступу в яких може сильно відрізнятися.

3. ОСНОВНІ ПОНЯТТЯ ТЕОРІЇ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

3.1. Граф алгоритму

Будь-яка програма для «звичайного комп'ютера» описує деяку родину алгоритмів. Вибір конкретного алгоритму при її реалізації визначається тим, як «спрацьовують» умовні оператори, які залежать від вхідних даних. Тому «звичайний» комп'ютер завжди виконує деяку послідовність дій, яка *однозначно* визначається програмою та вхідними даними, причому в кожний момент часу виконується рівно одна дія.

Інакша ситуація в системах з паралельною архітектурою. Для них в кожний момент часу може виконуватися цілий набір операцій, які не залежать одна від іншої. На довільній конкретній паралельній системі ці набори та послідовність їх виконання однозначно визначаються програмою та вхідними даними. На різних системах ці набори та послідовності можуть бути *різними*. Тим не менш, для гарантування отримання однакового результату порядок виконання послідовності операцій має задовольняти певні умови.

Деякі операції алгоритму використовують результати виконання інших операцій і тому *обов'язково мають виконуватися* після цих операцій. Таким чином на множині операцій розробник програми явно чи неявно задає деякий «частковий порядок», який для довільних двох операцій вказує, яка з них має виконуватися раніше, або констатує, що операції можуть виконуватися незалежно.

Кожній операції алгоритму ставиться у відповідність *вершина графа* [15]. Вершина графа *з'єднується дугою* з іншою вершиною, якщо перша вершина безпосередньо передує іншій (тобто результат першої операції є аргументом другої операції). Отриманий граф називається *графом алгоритму*.

У множині вершин графа також виділяють дві групи вершин: *вхідні вершини*, у які не входить жодна дуга, та *вихідні вершини*, з яких не виходять дуги.

Граф алгоритму майже завжди залежить від вхідних даних. Якщо у програмі відсутні умовні оператори, то він залежить від розмірів вхідних (вихідних) масивів, бо вони визначають загальну кількість операцій, а отже, і вершин графа. Таким чином на практиці майже завжди мають справу з алгоритмами, граф яких є параметризованим. Від значення параметрів залежить не тільки кількість вершин, а й уся сукупність дуг.

Якщо програма не містить умовних операторів, то її саму і також алгоритм, який вона реалізовує, називається *детермінованим.* Існує принципова відмінність між детермінованими та недетермінованими алгоритмами. Для детермінованого алгоритму завжди існує взаємно однозначна відповідність між всіма операціями та усіма вершинами графу алгоритму незалежно від значення вхідних параметрів. Для недетермінованого алгоритму такої відповідності нема.

Надалі, будуть розглядатися лише детерміновані алгоритми. Їх аналіз простіший, ніж у загальному випадку. Крім того, багато недетермінованих алгоритмів є «майже» детермінованими, тобто зводяться до детермінованих.

Уведений у розгляд граф алгоритму є орієнтованим ациклічним мультиграфом [15]. Його ациклічність випливає із того, що у довільних програмах реалізовують лише явні обчислення і ніяка величина не визначається сама через себе.

Твердження 3.1. Нехай $G=(V,E)$ — зв'язний ациклічний граф, який має *n* вершин. Тоді існує таке число $H<n$, що усі вершини графа можна так помітити одним із індексів $0,\ldots,H$, що усі значення індексу задіяні і якщо дуга виходить із вершини з індексом *i* та входить у вершини з індексом *j*, то $i<j$.

Граф, отриманим у відповідності із твердженням 3.1, називається *строгою паралельною формою* графа алгоритму [3]. Група вершин, які мають однакові індекси називається *ярусом* паралельної форми, а кількість вершин у групі — *шириною* ярусу. Кількість ярусів у паралельній формі (без врахування 0-го ярусу) називається *висотою* паралельної форми (вона рівна числу *H*), максимальна ширина ярусу — її *шириною*. Відповідні «ботанічні» терміни застосовуються також і безпосередньо до алгоритмів.

Якщо для простоти вважати, що усі операції алгоритму виконуються за одиницю часу, то висота паралельної форми алгоритму рівна часу реалізації алгоритму. Якщо алгоритм реалізовується на «звичайному» комп'ютері, то усі яруси паралельної форми (крім, можливо, 0-го) містять одну вершину. Така паралельна форма називається лінійною.

У [4] показано, що незалежно від того, яка паралельна форма алгоритму реалізується на комп'ютері, результат реалізації буде одним і тим самим.

Твердження 3.2 [4]. Нехай при виконанні операцій помилки заокруглень визначаються лише значеннями аргументів. Тоді при одних і тих самих вхідних даних усі реалізації алгоритму, які відповідають одному і тому самому частковому порядку на операціях, дають однаковий результат включно із помилками заокруглень.

3.2. Концепція необмеженого паралелізму

Поява перших паралельних обчислювальних систем в 60-х роках XX століття зумовила необхідність математичної концепції побудови *паралельних алгоритмів*, тобто алгоритмів, пристосованих до реалізації на таких системах. Швидкий розвиток елементної бази підказував дослідникам, що кількість обчислювальних пристроїв у системі невдовзі може стати дуже великим. Відповідна концепція отримала назву *концепції необмеженого паралелізму* [1, 4].

В основі концепції лежить припущення, що алгоритм реалізується на паралельній обчислювальній системі, яка не накладає на нього практично ніяких обмежень. Згідно до концепції вважається, що

- процесорів може бути як завгодно багато;
- усі процесори системи є універсальними;
- процесори працюють у синхронному режимі;
- усі запам'ятовуючі пристрої системи спільні;
- передавання інформації у системі виконується миттєво і без конфліктів.

Концепція необмеженого паралелізму є ідеалізованою математичною моделлю паралельної обчислювальної системи. Вона має як свої переваги, так і недоліки.

Для знаходження розв'язку однієї і тої самої задачі можуть використовуватися алгоритми різної паралельної складності. Серед них можуть бути і алгоритми найменшої висоти. Розглянемо класичний приклад, який демонструє принципи побудови алгоритмів «малої висоти».

3.2.1. Обчислення добутку елементів масиву

Нехай потрібно обчислити добуток n чисел $a_1, a_2, \ldots, a_n$.

Розглянемо випадок $n = 8$. Тоді звичайна схема, у якій реалізовується послідовний процес обчислень, має наступний вигляд:

Дані: $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$

Ярус 1: $a_1 a_2$

Ярус 2: $(a_1 a_2) a_3$

Ярус 3: $(a_1 a_2 a_3) a_4$

Ярус 4: $(a_1 a_2 a_3 a_4) a_5$

Ярус 5: $(a_1 a_2 a_3 a_4 a_5) a_6$

Ярус 6: $(a_1 a_2 a_3 a_4 a_5 a_6) a_7$

Ярус 7: $(a_1 a_2 a_3 a_4 a_5 a_6 a_7) a_8$

Висота паралельної форми рівна 7, ширина рівна 1. Якщо обчислювальна система має більше одного процесора, то за даної схеми усі вони крім одного будуть простоювати.

Наступна паралельна форма іншого алгоритму обчислення добутку використовує процесори більш ефективно:

Дані: $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$

Ярус 1: $a_1 a_2 \quad a_3 a_4 \quad a_5 a_6 \quad a_7 a_8$

Ярус 2: $(a_1 a_2)(a_3 a_4) \quad (a_5 a_6)(a_7 a_8)$

Ярус 3: $(a_1a_2a_3a_4)(a_5a_6a_7a_8)$

Висота паралельної форми рівна 3, ширина рівна 4. Суттєве зменшення висоти зумовлене більшим завантаженням процесорів виконанням корисної роботи. Відповідні графи описаних алгоритмів наведені на рис. 3.1 (початкові вершини символізують ввід даних).

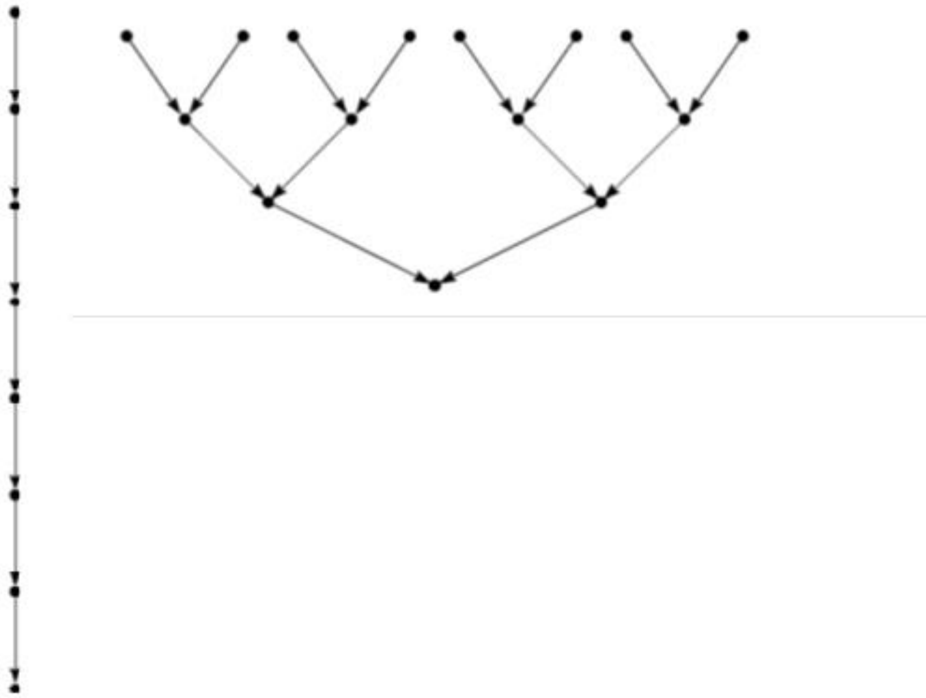

Рис. 3.1. Граф послідовного алгоритму та алгоритму «здвоєння»

Остання схема очевидним чином розповсюджується на випадок довільного *n*. Для її реалізації потрібно на кожному ярусі виконувати максимально можливу кількість множень пар чисел, які отримані на попередньому ярусі (і не мають спільних множників). В загальному випадку висота паралельної форми рівна $\lceil \log_2 n \rceil$, $\lceil \alpha \rceil$ означає ціле число, яке «найближче зверху» до числа α. Ця паралельна форма може бути реалізована на $\lfloor n/2 \rfloor$ процесорах, причому ступінь завантаженості процесорів зменшується від ярусу до ярусу. Процес побудови елементів кожного ярусу називається процесом *здвоєння* [3] (*каскадною схемою* [7]).

З використанням процесу здвоєння можна будувати алгоритми логарифмічної висоти для обчислення значень довільної асоціативної операції, наприклад додавання векторів, множення матриць і т. п.

Твердження 3.3 [4]. Нехай функція суттєво залежить від *n* аргументів і зображується у вигляді суперпозиції скінченної кількості операцій, кожна з яких має не більше *p* аргументів. Припустимо, що значення функції обчислюється за допомогою деякого алгоритму з використанням тих самих операцій. Тоді якщо H — висота алгоритму (без урахуванням вводу даних), то $H \geq \log_p n$.

Якщо деяка задача має n вхідних даних і вдається розробити алгоритм висоти $\log^\alpha n$, де $\alpha \geq 1$, то такий алгоритм можна вважати достатньо ефективним з точки зору його реалізації у паралельній системі [3].

Якщо система містить l однакових простих універсальних процесорів, то прискорення реалізації алгоритму на цій системі $S_l(n)$ можна обчислити за формулою [1]

$$S_l(n) = \frac{T_1(n)}{T_l(n)},$$

де $T_1(n)$ — час, за який можна реалізувати алгоритм на одному процесорі, $T_l(n)$ — час реалізації алгоритму в системі (висота алгоритму), n — розмірність задачі. Очевидно, що $1 \leq S_l(n) \leq l$.

Ефективністю $E_l(n)$ реалізації алгоритму у паралельній системі (завантаженістю) називається відношення прискорення до кількості процесорів системи [7], тобто

$$E_l(n) = \frac{S_l(n)}{l} = \frac{T_1(n)}{l \cdot T_l(n)}.$$

Для задачі обчислення добутку $n = 8$ чисел з використанням алгоритму здвоєння у системі з 4 процесорів $S_4(8) = 7/3$, $E_4(8) = \frac{7}{3 \cdot 4} = \frac{7}{12}$. У загальному випадку

$$S_{n/2}(n) \sim \frac{n}{\log_2 n}, \; E_{n/2}(n) \sim \frac{2}{\log_2 n}.$$

З останньої формули видно, що при $n \to \infty$ $E \to 0$.

Комбінований метод

Нехай кількість наявних процесорів рівна l, де $l < n$. Для спрощення запису будемо вважати, що l є дільником числа n (інакше можна додати потрібну кількість множників-одиниць). Поділимо усі множники на l порцій по n/l елементів кожна. Для кожної порції будемо рахувати добуток традиційним послідовним алгоритмом на окремому процесорі. Після цього обчислимо добуток l отриманих проміжкових добутків за схемою здвоєння. Граф комбінованого алгоритму наведений на рис. 3.2.

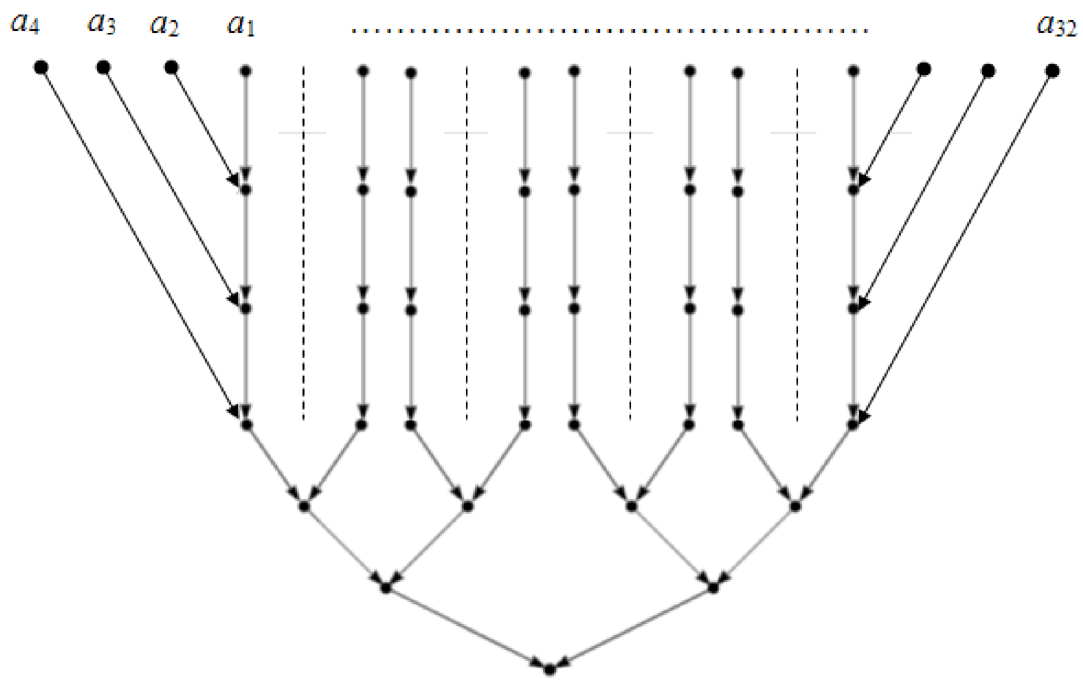

Рис. 3.2. Комбінований алгоритм у випадку n = 32, l = 8.

Обчислимо характеристики алгоритму:

$$T_l(n) = \left(\frac{n}{l} - 1\right) + \lceil \log_2 l \rceil, \qquad S_l(n) = \frac{T_1(n)}{T_l(n)} = \frac{n-1}{n/l - 1 + \lceil \log_2 l \rceil},$$

$$E_l(n) = \frac{T_1(n)}{l \cdot T_l(n)} = \frac{n-1}{n - l + l\lceil \log_2 l \rceil}.$$

В формулі для $T_l(n)$ перший доданок відповідає обчисленню часткових добутків, другий — схемі здвоєння. Таким чином при великих n: $S_l(n) \sim l$, $E_l(n) \sim 1$.

Приклад 3.1. Зобразити граф алгоритму паралельного обчислення значення виразу

$$a_1a_2 + a_2a_3 + a_3a_4 + a_4a_5 + a_5a_6 + a_6a_7 + a_7a_1$$

на обчислювальному пристрої

1) із одним універсальним процесором;
2) із трьома універсальними процесорами;
3) в умовах концепції необмеженого паралелізму.

Для кожної паралельної форми обчислити її висоту, ширину, прискорення та ефективність реалізації алгоритму.

Розв’язок. Будемо вважати, що у алгоритмі спочатку обчислюються зліва направо усі 7 добутків, а потім — 6 сум (у такому самому порядку).

1) Розглянемо реалізацію алгоритму у послідовній системі. Відповідний граф наведено на рис 3.3.

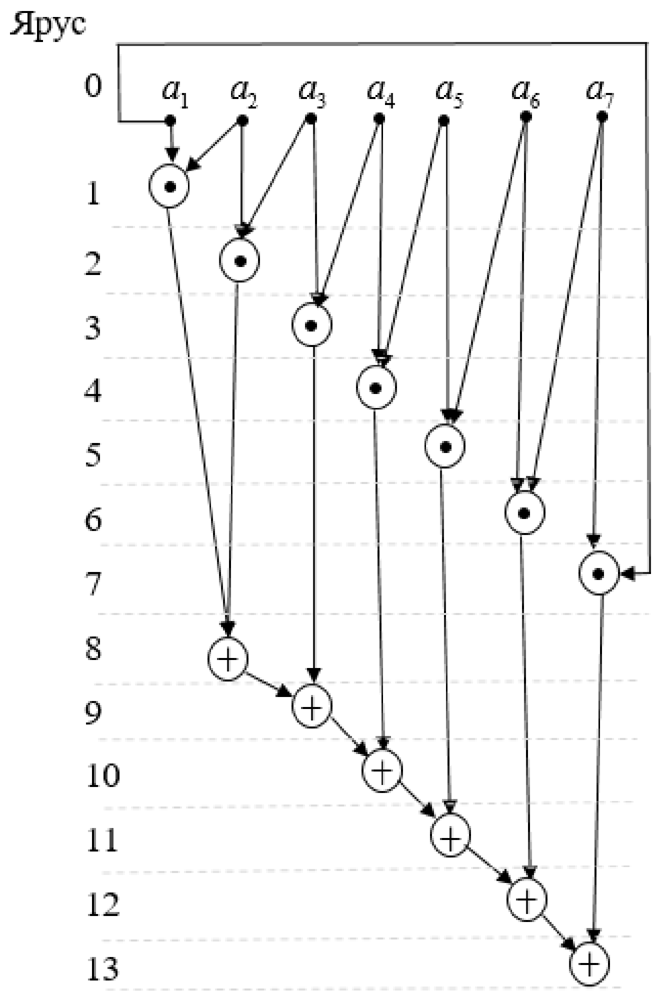

Рис. 3.3. Граф алгоритму у випадку $l = 1$.

Запишемо відповідні характеристики алгоритму:

$$n = 7,\ l = 1,\ T_1(7) = 13,\ S_1(7) = 1,\ E_1(7) = 1.$$

2) Розглянемо випадок системи з трьома процесорами. Відповідний граф алгоритму ширини 3 наведено на рис. 3.4. Як видно з рис. 3.4

$$n = 7,\ l = 3,\ T_3(7) = 6,\ S_3(7) = \frac{13}{6} \approx 2{,}17,\ E_3(7) = \frac{13}{6} : 3 = 13/18 \approx 0{,}72.$$

3) Розглянемо реалізацію алгоритму у випадку системи, кількість процесорів якої необмежена. Оскільки у алгоритмі операції додавання не можуть виконуватися раніше, ніж відповідні доданки-множники будуть обчисленні, то з урахуванням твердження 3.3 для висоти алгоритму $T_l(7)$ справджується оцінка $T_l(7) \le 1 + \lceil \log_2 7 \rceil = 4$.

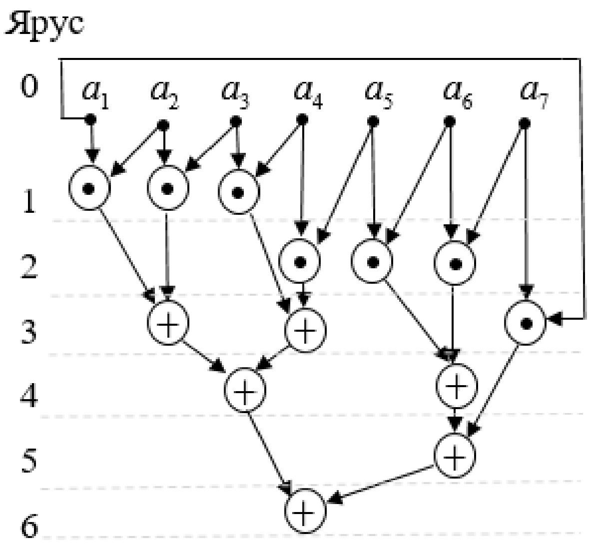

Рис. 3.4. Паралельна форма графа алгоритму у випадку $l = 3$.

У випадку $l = 6$ можна вказати алгоритм висоти 4, граф якого наведено на рис. 3.5.

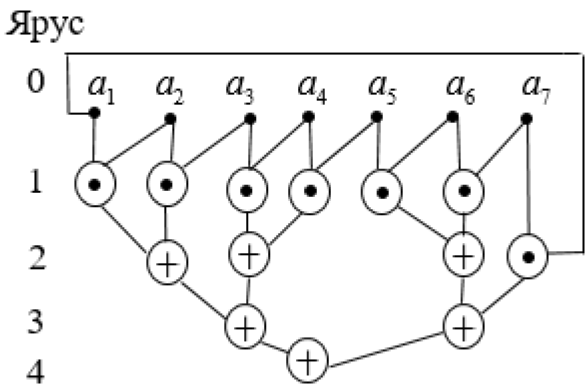

Рис. 3.5. Граф алгоритму у випадку $l = 6$.

Для цієї паралельної форми $l = 6,\ T_6(7) = 4,\ S_6(7) = \dfrac{13}{4} = 3{,}25,\ E_6(7) = \dfrac{13}{4} : 6 =$ $13/24 \approx 0{,}54$. Можна показати, що якщо $l < 6$, то $T_l(7) > 4$. Дійсно, із специфіки операцій алгоритму випливає, що на останньому ярусі виконується тільки одна операція, на передостанньому — не більше двох операцій, на третьому знизу — не більше чотирьох. Тому загалом на трьох нижніх ярусах може виконуватися не більше 7 операцій. Тому для попередніх ярусів залишається не менше, ніж $13 - 7 = 6$ операцій, які у випадку $l < 6$ не можуть бути виконані на одному ярусі. Тому кількість ярусів має бути не меншою за 5.

Задача 3.1. Знайти мінімальну висоту алгоритму, за допомогою якого можна обчислити вираз $a_1 + a_1^2 a_2 + a_1^4 a_2^2 a_3 + ... + a_1^{2^{n-1}} a_2^{2^{n-2}} \ldots a_{n-1}^2 a_n$ в паралельній обчислювальній системі в умовах концепції необмеженого паралелізму. Знайти мінімальну кількість процесорів, які забезпечують досягнення максимально можливого прискорення.

3.2.2. Обчислення добутку матриці на вектор

Нехай вектор $\mathbf{y} = (y_1, \ldots, y_n)$ обчислюється як добуток квадратної матриці $A = (a_{ij})\ (i, j = 1, \ldots, n)$ та вектора $\mathbf{x} = (x_1, \ldots, x_n)$, тобто

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \,.$$

Припустимо, що обчислювальна система має n^2 процесорів. Тоді на першому кроці можна паралельно обчислити усі n^2 добутків $a_{ij} x_j$, а потім, з використанням схеми здвоєння для додавання, за $\lceil \log_2 n \rceil$ кроків обчислити паралельно усі n сум, які визначають координати вектора $\mathbf{y}$. Тобто, ми отримали алгоритм обчислення добутку квадратної матриці та n-вимірного вектора, який має ширину n^2 та висоту $\lceil \log_2 n \rceil + 1$. Процесори використовуються нерівномірно. Усі процесори задіяні тільки на першому кроці. Далі кількість працюючих процесорів зменшується удвічі на кожному кроці. Для наведеного алгоритму

$$S = \left(n^2 + n(n-1)\right) / \left(\lceil \log_2 n \rceil + 1\right),\ E = \left(2 - \frac{1}{n}\right) / \left(\lceil \log_2 n \rceil + 1\right).$$

За допомогою аналогічних міркувань будується паралельний алгоритм розв’язування задачі множення двох квадратних матриць. Цю задачу можна звести до n задач множення першої матриці на послідовні стовпці другої матриці. Якщо використати попередній алгоритм, то отримуємо алгоритм, який має висоту порядку $\log_2 n$ та ширину n^3.

Слід звернути увагу на те, що у розглянутих паралельних версіях алгоритмів множення матриць та векторів виникає необхідність одночасного надсилання одним і тих самих даних на різні процесори. Такі операції не можна виконати дуже швидко. Тому на практиці процес обчислень може значно уповільнюватися.

3.2.3. Недоліки концепції необмеженого паралелізму

З використанням концепції необмеженого паралелізму розроблено велику кількість алгоритмів невеликої висоти. З деякими з них можна познайомитися в [1, 4, 7].

Однак слід зазначити, що переважна більшість з цих алгоритмів виявилися практично непридатними на практиці. *Основні причини цього — велика кількість необхідних процесорів, складні інформаційні зв'язки між операціями, катастрофічна обчислювальна нестійкість, велика кількість конфліктів пам'яті*.

Докази практичної непридатності алгоритмів з використанням концепції необмеженого паралелізму можна також отримати проаналізувавши прикладні програмні пакети, які постачаються разом із популярними паралельними обчислювальними системами. По суті, усі вони складаються із програм, які реалізують ті самі методи, які добре себе зарекомендували на послідовних комп'ютерах. Реально в деякій мірі використовується лише принцип здвоєння для обчислення сум та добутків чисел.

3.3. Внутрішній паралелізм

У процесі тривалого використання послідовних комп'ютерів був накопичений значний багаж обчислювальних алгоритмів та програм. Поява паралельних комп'ютерів повинна була зумовити розробку нових ефективних паралельних методів. Але на практиці цього не відбулося. Тому постає питання, як тоді розв'язувати задачі на паралельних комп'ютерах?

Відповідь на поставлене питання у термінах підрозділу 3.1. зводиться до наступного. Візьмемо довільний придатний алгоритм, записаний у вигляді математичних співвідношень, послідовних програм чи якимось іншим способом. Припустимо, що для цієї форми запису побудовано граф алгоритму. Припустимо також, що для цього графа знайдено паралельну форму із достатньою шириною ярусів. Тоді розглянутий алгоритм, принаймні принципово, можна реалізувати на паралельній обчислювальній системі. Важливо зауважити, що згідно з твердженням 3.2, паралельна реалізація буде мати такі самі обчислювальні властивості, що й звичайна. Подібний паралелізм у алгоритмах отримав назву *внутрішнього паралелізму.*

Виявилося, що багато існуючих ефективних послідовних алгоритмів мають значний запас «внутрішнього паралелізму». Складність полягає лише у тому, як виявити цей паралелізм.

3.3.1. Паралелізм у алгоритмі множення матриць

Розглянемо наступний приклад. Нехай потрібно обчислити добуток $A = BC$ двох квадратних матриць B та C порядку n. Згідно визначення операції множення матриць

$$a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}\,, \quad \left(i, j = 1, \ldots, n\right). \tag{3.1}$$

Самі ці формули не визначають алгоритм однозначно, оскільки не вказано порядок обчислення суми доданків $b_{ik}c_{kj}$. Однак відразу помітним є паралелізм обчислень. Він полягає у відсутності вказівок про якого-небудь порядку перебору індексів i та j.

Якщо операції множення та додавання виконуються точно, то усі порядки підсумування у (3.1) є еквівалентними і приводять до одного і того самого результату. Нехай вибрано наступний алгоритм реалізації формул (3.1):

$$a_{ij}^{(0)} = 0\,,$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} + b_{ik} c_{kj}, \quad \left(i, j, k = 1, \ldots, n\right), \tag{3.2}$$

$$a_{ij} = a_{ij}^{(n)}.$$

Знову явно вказано паралелізм перебору індексів i, j. Однак по індексу k паралелізму вже нема, так як цей індекс має послідовно перебиратися від 1 до n.

Побудуємо тепер граф алгоритму (3.2). Вершини графа не можна розташовувати довільним чином. Спосіб розташування підказує сам вигляд формул (3.2). Елементи графу будемо розташовувати у вузлах прямокутної ґратки у тривимірному просторі. Аналізуючи формулу (3.2) неважко помітити, що у вершину з координатами $\left(i, j, k\right)$ буде передаватися результат операції, якій відповідає вершина $\left(i, j, k-1\right)$.

Граф алгоритму влаштований достатньо просто. Він розпадається на n^2 незв'язних компонент. Кожний підграф містить n вершин і являє собою ланцюг, розташований паралельно осі k. У випадку $n = 2$ граф наведений на рис. 3.6, *а*.

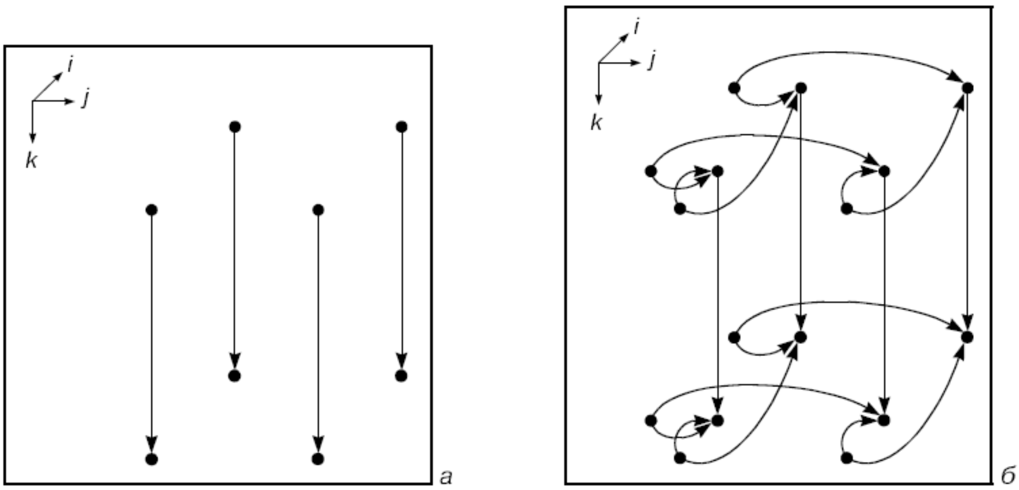

Рис. 3.6. Граф алгоритму множення матриць

У повному графі присутня множинна розсилка даних. Елемент b_{ik} розсилається по усім вершинам, які мають ті самі значення координат i та k. Аналогічною є розсилка елемента c_{kj} Для випадку $n=2$ відповідні розсилки елементів матриць B та C наведені на рис. 3.6, *б*. Наведений приклад також демонструє, як важливо правильно розташовувати вершини графа.

Слід зауважити, що якщо додавання у (3.1) виконується за схемою здвоєння, то кожний вертикальний ланцюг у графі має бути замінений на дерево, наведене на рис. 3.1.

3.3.2. Паралелізм у алгоритмі розв'язування системи лінійних алгебраїчних рівнянь

Нехай потрібно знайти розв'язок системи лінійних алгебраїчних рівнянь $A\mathbf{x}=\mathbf{b}$ з невиродженою трикутною матрицею порядку n методом зворотної підстановки. Припустимо, що матриця A є лівою трикутною матрицею з одиничною діагоналлю. Тоді маємо

$$x_1=b_1,\;\; x_i=b_i-\sum_{j=1}^{i-1}a_{ij}x_j,\;\left(2\le i\le n\right). \tag{3.3}$$

Цей запис також не визначає алгоритм однозначно, бо не вказано порядок обчислення сум. Розглянемо наступне уточнення процесу (3.3):

$$\begin{aligned} &x_i^{(0)}=b_i\,,\\ &x_i^{(j)}=x_i^{(j-1)}-a_{ij}x_j^{(j-1)},\;\; i=1,2,\dots,n,\;\; j=1,2,\dots,i-1,\\ &x_i=x_i^{(i-1)}. \end{aligned} \tag{3.4}$$

Основна операція алгоритму має вигляд $a-bc$. Вона виконується для усіх допустимих значень індексів i та j. Для побудови графа алгоритму в декартовій системі координат з осями i та j побудуємо прямокутну сітку і розмістимо у вузлах при $2\le i\le n,\; 1\le j\le i-1$ вершини графа, які відповідають операціям $a-bc$. Також зобразимо на графі вершини, які відповідають вводу вхідних даних a_{ij} та b_j. Цей граф у випадку $n=5$ зображено на рис. 3.7. Верхня кутова вершина знаходиться у точці (1, 0).

На цьому рисунку зображена одна із максимальних паралельних форм. Її яруси помічені пунктиром. Загальна кількість ярусів (без урахування вводу) рівна $n-1$.

Вибір зростаючого по j напрямку додавання у (3.3), який призвів до алгоритму (3.4), був зроблений, взагалі кажучи, випадково.

Аналогічно можна побудувати алгоритм зворотної підстановки з використанням додавання за спаданням індексу j:

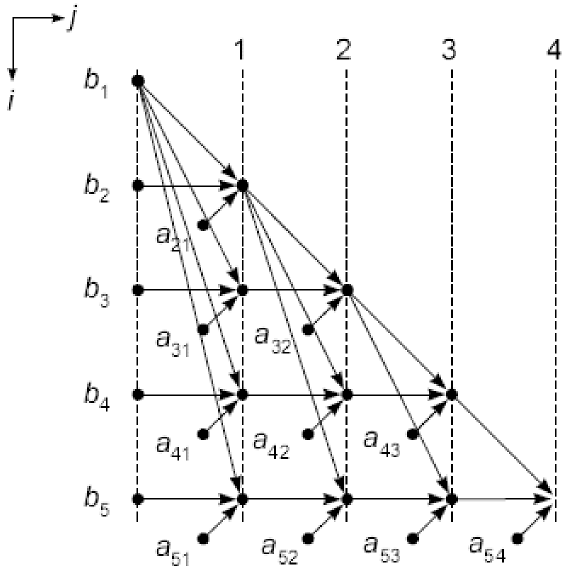

Рис. 3.7. Граф для алгоритму зворотної підстановки (9) для трикутної системи

$$x_i^{(i)} = b_i,$$

$$x_i^{(j)} = x_i^{(j+1)} - a_{ij}x_j^{(1)},\ \ i = 1, 2, \ldots, n,\ \ j = i-1, i-2, \ldots, 1. \tag{3.5}$$

$$x_i = x_i^{(1)}.$$

Відповідний граф для випадку $n = 5$ наведено на рис. 3.8. Верхня кутова вершина розташована у точці (1, 1).

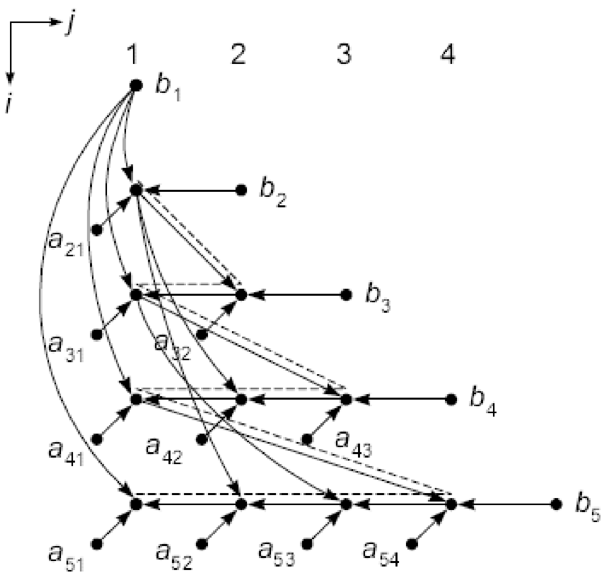

Рис. 3.8. Граф для алгоритму зворотної підстановки (10) для трикутної системи

Пробуючи розташувати вершини, які відповідають операціям $a - bc$, за ярусами хоча б однієї паралельної форми, приходимо до висновку, що тепер у кожному ярусі завжди може знаходитися тільки одна вершина. Цей факт пояснюється тим, що усі вершини графа на рис. 3.8 лежать на одному шляху, який позначений на рисунку пунктиром. Тому загальна кількість ярусів алгоритму (3.5), які містять операції вигляду $a - bc$, завжди рівна $1 + (1 + 2 + \ldots + n - 1) = (n^2 - n + 2) / 2$, що набагато більше за число $n - 1$ — кількість ярусів для відповідних операцій у алгоритмі (3.4).

Отриманий результат є досить несподіваним. Обидва алгоритми (3.4) та (3.5) призначені для розв’язування тієї самої задачі і розроблені на основі формул (3.3). Обидва алгоритми абсолютно однакові з точки зору їх реалізації на багатопроцесорній системі, оскільки потребують виконання однакової кількості операцій множення та віднімання і однакового об’єму пам’яті і є еквівалентними з точки зору помилок заокруглення.

Тим не менш, паралельні графи алгоритмів принципово різні. Якщо ці алгоритми реалізувати на паралельній системі з n універсальними процесорами, то алгоритм (3.4) можна реалізувати за час, пропорційний n, а алгоритм (3.5) — лише за час пропорційний n^2. У першому випадку завантаженість процесорів близька до $0,5$, а у другому — до 0.

Таким чином, алгоритми, цілком однакові при послідовній реалізації, можуть виявитися принципові відмінними при реалізацій на паралельній обчислювальній системі. В цьому, взагалі кажучи, і полягає основна складність програмування програмного забезпечення для обчислень на паралельних комп’ютерах.

4. ОСНОВИ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

4.1. Основні поняття паралельного програмування

Паралельна програма містить кілька процесів, які працюють паралельно над виконанням деякої задачі [6]. Кожний процес — це послідовна програма, тобто послідовність операторів, які виконуються один за одним. Послідовна програма має *один потік керування*, паралельна — *декілька*.

Сумісна робота процесів паралельної програми здійснюється за допомогою їх *взаємодії*. Взаємодія реалізується за рахунок використання *спільних змінних* (*shared variables*) або *передачі повідомлень*. У першому випадку один процес може змінювати значення змінних, які використовуються у інших процесах. У другому випадку процес надсилає повідомлення, яке отримують інші процеси.

При будь-якій взаємодії між процесами необхідною є взаємна *синхронізація*. Існує два основних види синхронізації — взаємне виключення (*mutual exclusion*) та умовна синхронізація (*condition synchronization*). *Взаємне виключення* забезпечує, щоб критичні секції операторів не виконувалися одночасно. *Умовна синхронізація* затримує процес до тих пір, поки не виконається певна умова. Наприклад, взаємодія процесів виробника (*producer*) та споживача (*consumer*) часто забезпечується з використанням буфера у спільній пам'яті. Виробник записує у буфер, споживач читає з нього. Для уникнення одночасного використання буфера виробником і споживачем (що може призвести до зчитування не повністю записаного повідомлення) використовується взаємне виключення. Умовна синхронізація використовується для перевірки того, чи було зчитано споживачем останнє записане у буфер повідомлення.

4.2. Нотація паралельних операторів та процесів

За замовчуванням оператори виконуються послідовно, тобто один за другим. Оператор `co` (concurrent — паралельний або такий, що відбувається одночасно) вказує на те, що кілька операторів можуть виконуватися паралельно. Розглядають дві форми оператора `co`. В першій формі оператор `co` має кілька гілок (arms):

```
co оператор₁,
// ...
// операторn,
oc
```

Кожна гілка містить оператор (або список операторів). Гілки відокремлюються *символом паралелізму* «`//`». Оператор, наведений вище, означає:

почати паралельне виконання усіх операторів та очікувати їх завершення.

Оператор `co`, таким чином, завершується після виконання усіх операторів, які містяться у його гілках.

У другій формі оператор co використовує один або кілька квантифікаторів, які вказують на те, що набір операторів має виконуватися паралельно для кожної комбінації значень параметрів циклу. Наприклад, наступний тривіальний оператор заповнює масиви a та b нулями:

```
co[i = 0 to n-1] {
    a[i] = 0;
    b[i] = 0;
}
```

Цей оператор створює *n* процесів, по одному для кожного значення змінної i. Область видимості лічильника циклу — опис процесу, і у кожного процесу своє, відмінне від інших, значення змінної i. Дві форми оператора co можна поєднувати. Наприклад, одна гілка може мати квантифікатор в квадратних дужках, а друга — ні.

Декларація *процесу* є, по суті, скороченою формою оператора co з одною гілкою. Вона починається з ключового слова process та назви процесу. Тіло процесу містить визначення локальних змінних та список операторів.

В наступному простому прикладі визначається процес foo, який додає числа від 1 до 10, записуючи результат в глобальну зміну x.

```
process foo{
    int sum = 0;
    for [i = 1 to 10]
        sum += i;
    x = sum;
}
```

Опис process записується на тому самому рівні програми, що й опис функцій та процедур, тобто процес не є оператором, на відміну від co. Крім того, оголошені процеси виконується у фоновому режимі, тоді як виконання оператора, який йде за оператором co, починається після завершення усіх операторів, які містяться у гілках розпаралелювання co. Декларації процесів, на відміну від оператору co, не можуть бути вкладені у інші декларації або оператори.

Ще один приклад: запис значень від 1 до n у стандартний потік виведення:

```
process bar1 {
    for [i = 1 to n]
        write(i);
}
```

Масив процесів оголошується шляхом додавання квантифікатора (в квадратних дужках) до назви процесу:

```
process bar2[i = 1 to n] {
    write(i);
}
```

`bar1` та `bar2` записують в стандартний вивід значення від 1 до `n`. Проте порядок, у якому їх записує масив процесів `bar2`, є *недетермінованим*, оскільки масив `bar2` складається з `n` окремих процесів, які виконується у довільному порядку. Існує `n!` різних можливих порядків виводу чисел для цього масиву.

Для запису однорядкових коментарів використовується символ «#» (використання традиційного коментаря «`//`» є неможливим, оскільки «`//`» використовується для відокремлення гілок оператору `co`).

4.3. Парадигми паралельного програмування

Не зважаючи на існування великої кількості паралельних програмних комплексів, в них використовується лише невелика кількість патернів або парадигм паралельного програмування. У [6] виокремлено п’ять основних парадигм:

1) ітеративний паралелізм;
2) рекурсивний паралелізм;
3) «виробники та споживачі» (конвеєри);
4) «клієнти та сервери»;
5) «взаємодіючі рівні» (interacting peers).

Ітеративний паралелізм використовується, коли у програмі є кілька процесів (часто ідентичних), кожний із яких містить один або кілька циклів. Таким чином, кожний процес є ітеративною програмою. Процеси програми *працюють сумісно над однією задачею*; вони взаємодіють та синхронізуються з допомогою спільних змінних або передачі повідомлень. Ітеративний паралелізм частіше за все зустрічається в наукових обчисленнях, які виконуються на кількох процесорах.

Рекурсивний паралелізм може використовуватися у випадку наявності у програмі однієї або кількох рекурсивних процедур, виклики яких незалежні, тобто кожний із них опрацьовує свою частини загальних даних. Рекурсія часто застосовується у імперативних мовах програмування, особливо при реалізації алгоритмів типу «поділяй та пануй» або «перебір з поверненням». Рекурсія є однією з фундаментальних парадигм також і в символьних, логічних, та функціональних мовах програмування. Рекурсивний паралелізм використовується для розв’язування таких комбінаторних проблем, як сортування, планування (задача комівояжера) та ігри (шахи та інші).

Виробники та споживачі — це взаємодіючі процеси. Вони часто організовані у конвеєр, через який проходить інформація. Кожний процес конвеєра є *фільтром*, який споживає вихідні дані свого попередника та продукує вхідні дані для свого користувача. Фільтри зустрічаються на рівні прикладного ПЗ в операційних системах типу ОС Unix, всередині самих операційних систем, якщо один процес виробляє вихідні дані, які споживає (читає) інший процес.

Клієнти та сервери — найбільш поширена модель взаємодії в розподілених системах, від локальних мереж до World Wide Web. Клієнтський процес робить запит до сервісу та очікує відповіді. Сервер очікує запити від клієнтів, а потім діє відповідно до цих запитів. Сервер може бути реалізований у вигляді одиночного процесу, який не може обробляти одночасно кілька запитів клієнтів, або (при необхідності паралельного обслуговування запитів) як багатопотокова програма. Клієнти та сервери — паралельне програмне узагальнення процедур та їх викликів: сервер виконує роль процедури, а клієнт її викликає. Але якщо код клієнта та код сервера розташовані на різних машинах, звичайний механізм виклику процедур недоступний. Замість цього необхідно використовувати віддалений виклик процедури або *рандеву*.

Взаємодіючі рівні — остання парадигма взаємодії. Вона зустрічається в розподілених програмах, в яких кілька процесів для розв’язування задачі виконують один і той самий код та обмінюються повідомленнями. Взаємодіючі рівні використовуються для реалізації розподілених паралельних програм, особливо при ітеративному паралелізмі та децентралізованому прийняті рішень в розподілених системах.

4.3.1. Ітеративний паралелізм: множення матриць

Ітеративна послідовна програма використовує для обробки даних та обчислення результатів цикли типу `for` та `while`. Ітеративна паралельна програма містить кілька ітеративних процесів. Кожний процес обчислює результати для підмножини даних, а потім ці результати збираються разом.

В якості прикладу розглянемо задачу із галузі наукових обчислень. Припустимо, що задані дві $n \times n$-матриці `a` та `b`. Мета — обчислити $c = a \cdot b$.

Матриці `a, b, c` є спільними глобальними змінними, індекси рядків та стовпців змінюються від `0` до `n-1`, елементи матриці є дійсними числами з плаваючою крапкою.

Добуток матриць можна знайти з використанням послідовної програми, яка має наступний псевдокод:

```
for [i = 0 to n-1] {
    for [j = 0 to n-1] {
        c[i,j] = 0;
        for [к = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Множення матриць — приклад *задачі з масовим паралелізмом*, оскільки програма містить велику кількість операцій, які можуть виконуватися паралельно. Дві операції можуть виконуватися паралельно, якщо вони незалежні. Припустимо, що *множина зчитування* операції містить змінні, які вона читає, але не змінює, а *множина запису* — змінні, які вона змінює (і, можливо, зчитує). Дві операції є *незалежними*, якщо їх множини запису не перетинаються. Неформально кажучи, процеси завжди можуть безпечно читати змінні, які не змінюються. Однак двом процесам взагалі кажучи небезпечно змінювати значення однієї і тієї самої змінної або одному процесу зчитувати змінну, яка записується іншим процесом.

При обчислені добутку матриць обчислення скалярних добутків рядків та стовпців є незалежними операціями. У рядках 3–5 попередньої програми виконується ініціалізація та обчислення елемента матриці `c`. Внутрішній цикл програми зчитує рядок матриці `a` та стовпець матриці `b`, а потім зчитує та записує один елемент матриці `c`. Множина зчитування для скалярного добутку — це рядок матриці `a` та стовпець матриці `b`, множина запису — елемент матриці `c`.

Оскільки множини запису скалярних добутків не перетинаються, їх можна виконувати паралельно. Можливими є варіанти, коли паралельно обчислюються результуючі рядки, результуючі стовпці чи групи рядків та стовпців.

Для початку розглянемо паралельне обчислення рядків матриці `c`:

```
co [i = 0 to n-1] { # паралельне обчислення рядків
    for [j = 0 to n-1] {
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Ця програма відрізняється від послідовного варіанта лише тим, що у зовнішньому циклі оператор `for` замінений оператором `co`. Але семантична різниця велика: оператор `co` вказує, що його тіло для кожного значення індексу `i` буде виконуватися паралельно (принаймні теоретично, в залежності від кількості наявних процесорів).

Інший спосіб паралельного множення — паралельне обчислення стовпців матриці `c`:

```
co [j = 0 to n-1] { # паралельне обчислення стовпців
    for [i = 0 to n-1] {
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

У цій версії два зовнішні цикли (за i та за j) помінялися місцями (якщо тіла двох циклів незалежні, то їх можна безпечно міняти місцями).

Дві отримані версії програми по суті мають ідентичні обчислювальні властивості. Проте у випадку множення $m \times n$ та $n \times p$ прямокутних матриць числа m та p можуть сильно відрізнятися, а тому з огляду на параметри обчислювальної системи обчислення з паралелізмом за рядками у випадку $m < p$ можуть виявитися значно ефективнішими, ніж обчислення з паралелізмом за стовпцями.

Можна паралельно обчислювати усі скалярні добутки.

```
co [i = 0 to n-1, j = 0 to n-1]
    c[i,j] = 0;
    for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```

Тіло оператора `co` виконується паралельно для кожної комбінації значень індексів `i` та `j`, тому програма визначає n^2 процесів. (Чи будуть вони всі виконуватися паралельно, залежить від конкретної реалізації). Другий спосіб паралельного обчислення скалярних добутків полягає у використанні вкладених операторів `co`.

```
co [i = 0 to n-1] { # рядки паралельно, потім

    co [j = 0 to n-1] { # стовпці паралельно
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

Для кожного рядка (зовнішній оператор `co`) і потім для кожного стовпця (внутрішній оператор `co`) задається по одному процесу. Результат усіх обох програм однакові: виконання внутрішнього циклу для усі n^2 комбінацій значень `i` та `j`. Різниця між ними — у визначенні процесів, а отже, і у часі їх створення.

Слід зазначити, що усі попередні паралельні програми були отримані заміною оператора `for` на `co`, причому це було зроблено тільки для індексів `i` та `j`. Виникає питання, як бути з внутрішнім циклом за індексом `k`? Чи можна цей оператор замінить оператором `co`? Відповідь — «ні», оскільки тіло внутрішнього циклу як зчитує, так і змінює значення змінної `c[i,j]`. Можна обчислити суму у циклі `for` за змінною `k`, використовуючи каскадну схему здвоєння, але як зазначено в [10], це не може забезпечити суттєвого прискорення для більшості машин.

Інший спосіб розпаралелити обчислення — використати ключове слово `process` замість оператора `co` (тобто використати декларацію процесу). По суті, `process` — це оператор `co`, який виконується у фоновому режимі. Наприклад, перша паралельна програма з наведених вище (з паралелізмом по рядкам результату) може бути записана наступним чином:

```
process row[i = 0 to n-1] { # рядки паралельно
        for [j = 0 to n-1] {
        c[i,j] = 0;
        for [k = 0 to n-1]
            c[i,j] = c[i,j] + a[i,k]*b[k,j];
    }
}
```

В програмі визначений масив процесів — `row[1]`, `row[2]` і т.д. — по одному для кожного значення індексу `i`. Ці *n* процесів створюються і починають виконуватися, коли зустрічається рядок декларації процесу. Якщо за декларацією процесу йдуть оператори, то вони виконуються паралельно з процесом, тоді як оператори, записані після оператора `co`, не виконуються до його завершення.

В програмах, наведених вище, для кожного елемента, рядка або стовпця результуючої матриці використано по одному процесу. Припустимо, що число процесорів в системі менше за `n` (так зазвичай і буває, коли `n` велике). У цьому випадку є очевидний спосіб повного використання усіх процесорів: поділити матрицю на смуги (рядків чи стовпців) і для кожної смуги створити робочий процес, який обчислює результати для елементів своєї смуги. Припустимо, що є `P` процесорів і `n` кратне `P`. Тоді у випадку смуг рядків робочі процеси можна запрограмувати так:

```
process worker[w = 1 to P] {
    int first = (w-1) * n/P;     # перший рядок смуги
    int last = first + n/P - 1; # останній рядок смуги
    for [i = first to last] {
        for [j = 0 to n-1] {
            c[i,j] = 0;
            for [k = 0 to n-1]
                c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
    }
}
```

В програму додані оператори, необхідні для визначення першого та останнього рядка кожної смуги. Потім рядки смуги вказуються у циклі (за індексом `i`) для обчислення елементів матриці `c`.

Таким чином, суттєвою умовою розпаралелювання програм є наявність незалежних обчислень, тобто обчислень з множинами запису, що не перетинаються. Для добутку матриць незалежними обчисленнями є скалярні добутки рядків на стовпці, оскільки кожний із них записує (та читає) свій елемент `c[i,j]`. Тому можна паралельно обчислювати усі скалярні добутки, рядки, стовпці або смуги з використанням оператора `co` або декларації процесу.

4.3.2. Рекурсивний паралелізм: адаптивна квадратура

Програма є рекурсивною, якщо вона містить процедури, які викликають самі себе — прямо або опосередковано. Рекурсія дуальна до ітерації, тобто рекурсивні програми можна перетворити у ітеративні і навпаки. У тілі багатьох рекурсивних процедур звертання до самої себе зустрічається більше одного разу. Наприклад, алгоритм `quicksort` розбиває масив на дві частини, а потім двічі викликає себе для окремого сортування лівої та правої частин. Значна кількість алгоритмів обробки дерев та графів мають подібну структуру.

Рекурсивну програму можна реалізувати за допомогою паралелізму, якщо вона містить кілька незалежних рекурсивних викликів. Два виклики процедури (або функції) є незалежними, якщо їх множини запису не перетинаються. Ця умова виконується, якщо:

1) процедура не звертається до глобальних змінних або тільки читає їх;

2) аргументи и результуючі змінні процедур — різні змінні.

Наприклад, якщо процедура не звертається до глобальних змінних і має тільки параметри-значення (за механізмом їх передачі), то будь-який її виклик буде незалежним (процедура читає та змінює тільки локальні змінні, і кожний екземпляр процедури має свою локальну копію змінних).

Розглянемо *задачу квадратури*, яка полягає у наближеному обчисленні інтеграла неперервної функції. Припустимо, що це функція $f(x)$. Як показано на рис. 4.1, інтеграл функції $f(x)$ від a до b — це площа фігури, обмеженої графіком $f(x)$, віссю абсцис та прямими $x = a$ і $x = b$.

Існує два основних способи апроксимації значення інтеграла. Перший — розбити інтервал від a до b на фіксоване число відрізків, а потім апроксимувати площу на кожному з них за правилом трапецій або за правилом Сімпсона.

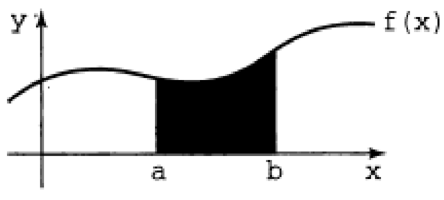

Рис. 4.1. Задача квадратури

```
double fleft = f(a), fright, area = 0.
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width]{
    fright = f(x);
    area = area + (fleft + fright) * width / 2;
    fleft = fright;
}
```

Кожна ітерація обчислює площа малої фігури за правилом трапецій і додає її до загального значення площі. Змінна `width` — ширина кожної трапеції, відрізки перебираються зліва направо, тому праве значення кожної ітерації стає лівим значенням наступної ітерації.

Другий спосіб апроксимації інтеграла — використовувати парадигму «поділяй і пануй» і змінне число інтервалів. Зокрема, спочатку обчислюють значення m — середину відрізка$[a, b]$. Потім апроксимують площу трьох областей під кривою $f(x)$: від а до m, від m до b та від a до b. Якщо сума менших площ дорівнює більшій площі з деякою заданою точністю ε, то апроксимацію можна вважати достатньою [6]. Якщо ні, то задача ділиться на дві підзадачі: від a до m та від m до b, і процес повторюється. Цей спосіб називається *адаптивною квадратурою*, оскільки алгоритм «адаптується» до форми кривої. Його можна запрограмувати так.

```
double quad(double left,right,fleft,fright,lrarea){
    double mid = (left + right) / 2;
    double fmid = f(mid);
    double larea = (fleft+fmid) * (mid-left) / 2;
    double rarea = (fmid+fright) * (right-mid) / 2;
    if(abs((larea+rarea) - lrarea) > EPSILON){
        larea = quad(left, mid, fleft, fmid, larea);
        rarea = quad(mid, right, fmid, fright, rarea);
    }
    return (larea + rarea);
}
```

Інтеграл функції $f(x)$ від a до b апроксимується таким викликом функції:

```
area = quad (a, b, f(a), f(b), (f(a)+f(b)) * (b-a) / 2);
```

У функції знову використовується правило трапеції. Значення функції f у крайніх точках відрізка і наближена площа цього інтервалу передаються в кожен виклик функції `quad`, щоб не обчислювати їх більше одного разу.

Ітеративну програму не можна розпаралелити (у наведеній формі), оскільки тіло циклу і зчитує, і записує значення змінної `area`. Проте в рекурсивної програмі виклики функції `quad` незалежні за умови, що обчислення функції $f(x)$ не дає побічних ефектів. Зокрема, аргументи функції `quad` передаються за значенням, і в її тілі немає присвоювання глобальних змінних. Таким чином, для задачі паралельного виконання рекурсивних викликів функції можна використовувати оператор `co`.

```
co larea = quad (left, mid, fleft, fmid, larea);
// rarea = quad (mid, right, fmid, fright, rarea);
oc
```

Це єдина зміна, необхідна для того, щоб зробити цю програму паралельною. Оскільки оператор `co` не закінчується до тих пір, поки не будуть завершені обидва виклику функцій, значення змінних `larea` і `rarea` обчислюються до того, як функція `quad` поверне їх суму.

Отже, програму з декількома рекурсивними викликами функцій можна легко перетворити в паралельну рекурсивну програму, якщо виклики незалежні. Проте існує чисто практична проблема: паралельно виконуваних операцій може стати занадто багато. Кожен оператор `co` в наведеній вище програмі створює два процеси, по одному для кожного виклику функції. Якщо глибина рекурсії велика, то виникне занадто велика для паралельного виконання кількість процесів. Вирішення цієї проблеми полягає *в скороченні* (або відтинанні) дерева рекурсії при занадто великій кількості процесів, тобто перехід з паралельних рекурсивних викликів на послідовні.

4.3.3. Виробники і споживачі: канали ОС Unix

Процес-виробник виконує обчислення і виводить потік результатів. Процес-споживач вводить і аналізує потік значень. Багато програм в тій чи іншій формі є виробниками та / або споживачами. Поєднання стає особливо цікавим, якщо виробники і споживачі об'єднані в конвеєр — послідовність процесів, в якій кожен з них споживає дані виходу попередника і виробляє дані для подальшого процесу. Класичним прикладом є конвеєри в операційній системі Unix. Однією з найбільш потужних функцій, запропонованих в ОС Unix, була можливість прив’язки стандартних «пристроїв» введення-виведення до різних типів файлів. Зокрема, файли `stdin` або `stdout` можуть бути пов’язані з файлом даних або з «файлом» особливого типу, який називається каналом.

Канал — це буфер (черга типу FIFO) між процесом-виробником і процесом-споживачем, який містить зв’язану послідовність символів, до якої виробник може дописувати нові символи. Символи видаляються, коли процес-споживач зчитує їх з каналу. Процес-виробник очікує (при необхідності), поки в буфері з’явиться вільне місце, потім додає в кінець буфера новий рядок. Процес-споживач очікує, поки в буфері не з’явиться рядок даних, потім зчитує його з буфера.

4.3.4. Клієнти і сервери: файлові системи

Між виробником і споживачем існує односпрямований потік інформації. Цей вид взаємодії між процесами часто зустрічається в паралельних програмах і не має аналогів в послідовних, оскільки в послідовній програмі тільки один потік управління, тоді як виробники і споживачі — незалежні процеси з власними потоками управління і власними швидкостями виконання.

Ще однією типовою схемою в паралельних програмах є взаємозв'язок типу клієнт-сервер. Процес-клієнт запитує сервіс, потім очікує обробки запиту. Процес-сервер багаторазово очікує запит, обробляє його, потім посилає відповідь. Як показано на рис. 4.2, існує двонапрямлений потік інформації: від клієнта до сервера і назад.

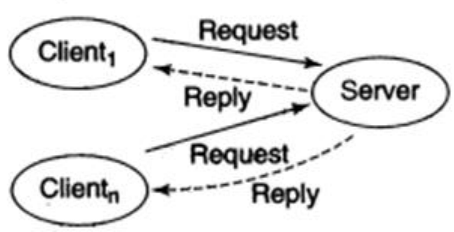

Рис. 4.2. Клієнти та сервери

Зв'язки між клієнтом і сервером в паралельному програмуванні аналогічні відносинам між програмою, що викликає підпрограму, і самої підпрограмою в послідовному програмуванні. Більш того, як підпрограма може бути викликана з кількох місць програми, так і у сервера зазвичай є багато клієнтів. Запити кожного клієнта повинні оброблятися незалежно, проте паралельно може оброблятися декілька запитів, подібно до того, як одночасно можуть бути активні кілька викликів однієї і тієї ж процедури.

Взаємодія типу клієнт-сервер зустрічається в операційних системах, об'єктно-орієнтованих системах, мережах, базах даних і багатьох інших програмах. Типовий приклад — читання і запис файлу. Для визначеності припустимо, що є модуль файлового сервера, що забезпечує дві операції з файлом: `read` (читати) і `write` (писати). Коли процес-клієнт хоче отримати доступ до файлу, він викликає операцію читання або запису у відповідному модулі файлового сервера.

На однопроцесорній машині або в іншій системі з пам'яттю файловий сервер зазвичай реалізується набором підпрограм (для операцій `read`, `write` і т. д.) і структурами даних, що зображають файли (наприклад, дескрипторами файлів). Отже, взаємодія між процесом-клієнтом і файлом зазвичай реалізується викликом відповідної процедури. Проте, якщо файл розділяється, важливо, щоб запис в нього вівся одночасно тільки одним процесом, а зчитуватися він може одночасно кількома. Цей різновид задачі — приклад задачі про «читачів і письменників», класичної задачі паралельного програмування.

4.3.5. Взаємодіючі рівні: розподілене множення матриць

Розглянемо два способи вирішення цієї задачі з використанням процесів, взаємодіючих за допомогою пересилання повідомлень. Перша програма використовує керуючий процес і масив незалежних робочих процесів. У другій програмі робочі процеси рівні і їх взаємодія забезпечується круговим конвеєром. Рис. 4.3 ілюструє схему взаємодії цих процесів.

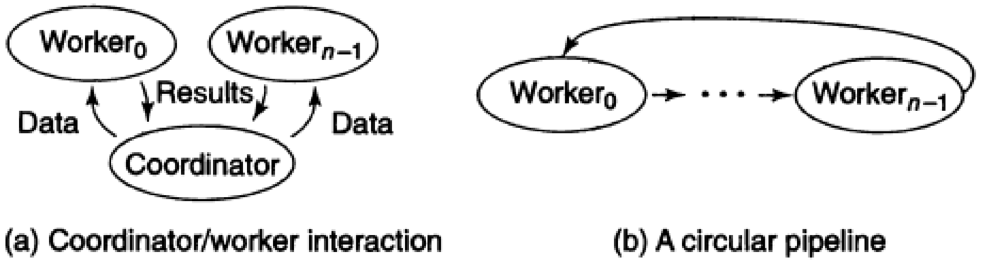

Рис. 4.3. Множення матриць з використанням передачі повідомлень

На машинах з розподіленою пам'яттю кожен процесор має доступ тільки до власної локальної пам'яті. Це означає, що програма не може використовувати глобальні змінні, тому будь-яка змінна повинна бути локальною для деякого процесу і може бути доступною тільки цьому процесу або процедури. Отже, для взаємодії процеси повинні використовувати передачу повідомлень.

Нехай нам необхідно знайти добуток квадратних $n \times n$ матриць a і b, а результат записати в матрицю c. Припустимо, що у системі є n процесорів. Можна використовувати масив з n робочих процесів, змусивши кожен робочий процес обчислювати один рядок результуючої матриці c:

```
process worker[i = 0 to n-1] {
    receive початкові значення вектора a та матриці b;
    # вектори a, c — i-ті рядки відповідних матриць
    for [j = 0 to n-1] {
        c[j] = 0;
        for [k = 0 to n-1]
            c[j] = c[j] + a[k] * b[k,j];
    }
    send вектор-результат c керуючому процесу;
}
```

Робочий процес `i` обчислює i-й рядок результуючої матриці c. Щоб це зробити, він повинен отримати рядок i вихідної матриці a і всю вихідну матрицю b.

Робочий процес спочатку отримує ці значення від керуючого процесу. Потім він обчислює свій рядок результатів і відсилає її назад керуючому.

Керуючий процес ініціює обчислення, збирає і виводить їх результати. Зокрема, спочатку керуючий процес посилає кожному робочому відповідну рядок матриці *a* і всю матрицю *b*. Потім керуючий процес очікує отримання рядків матриці від кожного робочого процесу. Схема керуючого процесу така.

```
process coordinator {
    double a[n, n]; # Вихідна матриця a
    double b[n, n]; # Вихідна матриця b
    double c[n, n]; # Результуюча матриця c
    # ініціалізувати a та b;
    for [i = 0 to n-1] {
        send рядок i матриці a процесу worker[i];
        send всю матрицю b процесу worker[i];
    }
    for [i = 0 to n-1]
        receive рядок i матриці c від процесу worker[i];
    вивести результат, який тепер в матриці c;
}
```

Оператори `send` та `receive` — це *примітиви* передачі повідомлень. Операція `send` надсилає повідомлення іншому процесу; операція `receive` чекає повідомлення від іншого процесу, а потім зберігає його в локальних змінних.

Тепер припустимо, що у кожного процесу є тільки один стовпець, а не вся матриця *b*. Отже, в початковому стані робочий процес `i` має *i*-й стовпець матриці *b*. Маючи лише ці вихідні дані, робочий процес може обчислити тільки значення `c[i,i]`. Для того щоб робочий процес `i` міг вирахувати весь рядок матриці `c`, він повинен отримати всі стовпці матриці *b*. Для цього можна використовувати кругової конвеєр (див. рис. 4.3). Кожен робочий процес виконує послідовність раундів; в кожному раунді він надсилає свій стовпець матриці *b* наступному процесу і отримує інший її стовпець від попереднього. Програма має наступний вигляд:

```
process worker[i = 0 to n-1] {
    double a[n];      # Рядок i матриці a
    double b[n];      # Один стовпець матриці b
    double c[n];      # Рядок i матриці c
    double sum = 0;   # Для проміжних добутків
    int nextCol = i; # Наступний стовпець результатів
    receive рядок i матриці a та стовпець i матриці b;
    # Обчислити c[i, i] = a[i,*] b[*,i]
    for [k = 0 to n-1]
        sum = sum + a[k] * b[k];
    c[nextCol] = sum;
```

```
    # Пустити по колу стовпці та обчислити інші c[i,*]
    for [j = 1 to n-1] {
        send мій стовпець матриці b наступному процесу;
        receive новий стовпець матриці b від попереднього;
        sum = 0;
        for [k = 0 to n-1]
            sum = sum + a[k] * b[k];
        if (nextCol == 0)
            nextCol = n-1;
        else
            nextCol = nextCol-1;
        c[nextCol] = sum;
    }
    send вектор-результат c керуючому процесу;
}
```

Робочі процеси впорядковані відповідно до їх індексів. (Для процесу $n - 1$ наступним є процес 0, а попереднім для 0 — процес $n - 1$.) Стовпці матриці b передаються по колу між робочими процесами, тому кожен процес врешті-решт отримає кожен стовпець. Змінна `nextCol` відстежує, куди у векторі `c` помістити черговий проміжний добуток. Як і в першому обчисленні, передбачається, що керуючий процес надсилає рядки матриці a і стовпці матриці b робочим процесам, а потім отримує від них рядки матриці c.

У попередній програмі використано відношення між процесорами, яке називається *взаємодіючі рівні*. Кожен робочий процес виконує той самий алгоритм і взаємодіє з іншими робочими процесами, щоб обчислити свою частину необхідного результату.

У першій з наведених програм матриця b *дублюється* в кожному процесі. У другій програмі в будь-який момент часу у кожного процесу є один рядок матриці a і тільки один стовпець матриці b. Це знижує витрати пам'яті для кожного процесу, але друга програма виконується довше першої, оскільки на кожній її ітерації кожен робочий процес повинен відіслати повідомлення одному сусідові і отримати повідомлення від іншого. Дані програми ілюструють класичне протиріччя між часом і використанням ресурсів в обчисленнях.

5. ПРОГРАМУВАННЯ ІЗ СПІЛЬНИМИ ЗМІННИМИ

У послідовних програмах часто використовуються спільні змінні, наприклад, для зберігання глобальних структур даних, але вважається, що краще обходитися без них. Разом з тим, паралельні програми цілком залежать від спільних компонентів, оскільки процеси можуть працювати над однією задачею, тільки взаємодіючи. А єдиний спосіб взаємодії — можливість для одного процесу записувати в *щось*, звідки інший процес читає. Цим чимось може бути колективна змінна або канал зв'язку. Тому взаємодія програмується як запис і читання спільних змінних або як передача і прийом повідомлень.

Взаємодія підвищує необхідність *синхронізації*. Існує два основних її типи: взаємне виключення і умовна синхронізація. Взаємне виключення зустрічається, коли два процеси повинні по черзі звертатися до таких спільних об'єктів, як, наприклад, записи в системі замовлення авіаквитків. Умовна синхронізація відбувається, коли одному процесу доводиться чекати інший процес, наприклад, коли процес-споживач очікує дані від процесу-виробника.

5.1. Стан, дії, історія та властивості

Стан паралельної програми складається зі значень змінних програми в деякий момент часу. Змінні можуть бути описані явно певними програмістом або неявними (на кшталт програмного лічильника кожного процесу), що зберігають приховану інформацію про стан. Паралельна програма починає виконання в деякому вихідному стані. Кожен процес програми виконується незалежно, і в міру виконання він перевіряє і змінює стан програми.

Процес виконує послідовність операторів. Оператор, в свою чергу, реалізується послідовністю *неподільних дій*. Ці дії перевіряють чи змінюють стан програми *неподільним чином*. Прикладами неподільних дій є неперервні машинні інструкції, які завантажують і зберігають слова пам'яті.

Виконання паралельної програми призводить до чергування послідовностей неподільних дій, вироблених кожним процесом. Конкретне виконання кожної програми може бути розглянуто як *історія* $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$, де s_0 — початковий стан. Переходи між станами здійснюються неподільними діями, що змінюють стан. Навіть паралельне виконання можна подати у вигляді лінійної історії, оскільки паралельна реалізація набору неподільних дій еквівалентна їх виконання в деякому послідовному порядку. Мета синхронізації — виключити небажані історії паралельної програми.

Властивістю програми називається атрибут, який є істинним за будь-якої можливої історії програми і, отже, при всіх її виконаннях. Є два типи властивостей: безпека та живучість. *Властивість безпеки* полягає в тому, що програма ніколи не потрапляє в «поганий» стан (при якому деякі змінні можуть мати небажані значення). *Властивість живучості* означає, що програма в кінці кінців завжди потрапляє в «гарний» стан, тобто стан, в якому всі змінні мають бажані значення.

Взаємне виключення — це приклад властивості безпеки в паралельній програмі. При поганому стані два процеси такої програми одночасно виконують дії в різних критичних секціях. Можливість врешті-решт *увійти в критичну секцію* — приклад властивості живучості в паралельній програмі. В хорошому стані кожен процес виконується в своїй критичній секції.

Постає питання: як перевірити, що дана програма має бажану властивість? Звичайний підхід полягає в тестуванні. Його можна охарактеризувати фразою «запусти програму і подивися, що вийде». Це відповідає перебору деяких можливих історій програми і перевірці їх прийнятності. Недолік такої перевірки полягає в тому, що кожен тест стосується тільки однієї історії виконання, а в загальному випадку кожний запуск програми приводить до нової історії.

Другий підхід — використання *операторних міркувань*, які можна назвати «вичерпним аналізом випадків» (перебираються всі можливі історії виконання програми). Для цього аналізуються способи чергування неподільних дій процесів. На жаль, в паралельній програмі число можливих історій зазвичай дуже велике. Припустимо, що програма містить n процесів і кожен з них виконує послідовність з m неподільних дій. Тоді число різних історій програми складе $\frac{(n \cdot m)!}{(m!)^n}$. Для програми з трьох процесів, кожен з яких виконує лише дві неподільні операції, можливі 90 різних історій.

Третій підхід — використання *стверджувальних міркувань* (assertional reasoning). Цей підхід можна назвати «абстрактним аналізом» [2, 6].

5.2. Розпаралелювання: пошук зразка в файлі

Розглянемо одну просту задачу і вивчимо способи її розпаралелювання. Розглянемо задачу пошуку всіх входжень шаблону `pattern` у файлі. Відповідна послідовна програма має вигляд:

```
string line;
прочитати рядок в line;
while(! EOF){
    шукати pattern в line;
    if(pattern є в line)
        вивести line;
    прочитати наступний рядок;
}
```

Тепер бажано з'ясувати, чи можна розпаралелити цю програму? Основна вимога для можливості розпаралелювання будь-якої програми полягає в тому, що вона повинна містити незалежні частини. Дві частини взаємно залежні, якщо кожна з них породжує результати, необхідні для іншої; це можливо, тільки якщо вони зчитують і записують спільні змінні. Отже, дві частини програми незалежні, якщо вони не виконують читання і запис одних і тих же змінних. Більш точне визначення таке:

> **Незалежність паралельних процесів.** Дві частини програми є незалежними, якщо перетин множини запису однієї та множини зчитування іншої — порожній.

При цьому вважаємо, що зчитування або запис будь-якої змінної неподільні. Іноді дві частини програми можуть безпечно виконуватися паралельно, навіть змінюючи одні і ті самі змінні. Це можливо, якщо не важливий порядок, в якому відбувається зміна. Наприклад, якщо кілька процесів періодично оновлюють графічний екран, і будь-який порядок виконання оновлень не псує вигляду екрана.

Повернемося до задачі пошуку шаблону в файлі. Які частини програми є незалежними і, отже, можуть бути виконані паралельно? Програма починається читанням першого рядку введення; це *повинно* бути зроблено перед усіма іншими діями. Після цього програма входить в цикл пошуку шаблону, виводить рядок, якщо шаблон був знайдений, а потім зчитує новий рядок. Вивести рядок до того, як в ній був виконаний пошук шаблону, не можна, тому перші два рядки циклу виконати паралельно неможливо. Розглянемо іншу, паралельну, версію попередньої програми,

```
string line;
прочитати вхідний рядок в line;
while(!EOF){
    co
        шукати pattern у line;
        if(pattern є у line)
            вивести line;
    // Прочитати наступний рядок та записати його в line;
    oc;
}
```

Відзначимо, що перша гілка оператора `co` є послідовністю операторів. Але два процеси не є незалежними, оскільки перший читає `line`, а інший записує в неї. Тому, якщо другий процес виконується швидше першого, він буде перезаписувати рядок до того, як її встигне перевірити перший процес.

Частини програми можуть виконуватися паралельно тільки в тому випадку, якщо вони читають і записують різні змінні. Припустимо, що другий процес записує не в ту змінну, яку перевіряє перший процес, і розглянемо наступну програму.

```
string line1, line2;
прочитати рядок введення в line1;
while(!EOF){
    co
        шукати pattern в line1;
        if(pattern є в line1)
            вивести line1;
    // прочитати наступний рядок та записати його в line2;
    oc;
    line1 = line2;
}
```

Процеси всередині оператора `co` незалежні, але їх дії пов'язані останнім оператором у циклі, який копіює `line2` в `line1`. Паралельна програма, наведена вище, правильна, але абсолютно неефективна. По-перше, в останньому рядку циклу вміст змінної `line2` копіюється в змінну `line1`. Це вимагає копіювання великої кількості символів, а це — накладні витрати. По-друге, в тілі циклу міститься оператор `co`, а це означає, що при кожному повторенні циклу `while` будуть створюватися, виконуватися і знищуватися по два процеси.

Вирішення проблеми у тому, що замість використання оператора `co` всередині циклу `while`, можна помістити цикли `while` в кожну гілку оператора `co`. Отримана програма є прикладом схеми типу «виробник-споживач». Другий процес є виробником, а перший — споживачем. Вони взаємодіють з допомогою змінної `buffer`. Відзначимо, що оголошення змінних `line1` і `line2` тепер стали локальними для процесів. Перевага стилю «`while` всередині `co`» полягає в тому, що процеси створюються тільки одного разу, а не при кожному повторенні циклу. Недоліком є необхідність використовувати два буфера і програмувати синхронізацію. Оператори, що передують зверненню до буфера `buffer` і наступні за ним, вказують тип необхідної синхронізації.

```
string buffer;          # Містить один рядок введення
bool done = false;      # Сигнал про завершення
co # процес 1: знайти шаблони
    string line1;
    while (true) {
        очікувати заповнення буфера або значення true змінної done;
        if(done) break;
        line1 = buffer;
        сигналізувати, що буфер порожній;
        шукати pattern в line1;
```

```
        if (pattern є в line1)
            надрукувати line1;
    }
    // # процес 2: прочитати нові рядки
    string line2;
    while (true) {
        прочитати наступний рядок введення в line2;
        if (EOF) {done = true; break;}
        очікувати спустошення буфера
        buffer = line2;
        сигналізувати про заповнення буфера;
    }
oc
```

5.3. Синхронізація: пошук максимального елемента масиву

Розглянемо іншу задачу, яка вимагає синхронізації процесів. Вона полягає у пошуку максимального елемента масиву `a[n]`. Припустимо, що `n` додатне і всі елементи масиву — додатні цілі числа. Для знаходження розв'язку можна використовувати таку послідовну програму:

```
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

Розглянемо способи розпаралелювання наведеної програми. Припустимо, що цикл повністю розпаралелений за допомогою паралельної перевірки всіх елементів:

```
int m = 0;

co [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

Ця програма є некоректною, оскільки процеси не є незалежними: кожен з них і читає, і записує змінну `m`. Зокрема, припустимо, що всі процеси виконуються з однаковою швидкістю, і, отже, кожен з них порівнює свій елемент масиву `a[i]` зі змінною `m` в один і той же час. Всі процеси визначать, що нерівність виконується (оскільки всі елементи масиву більші за початкове значення змінної `m`). Тому всі процеси спробують оновити значення `m`. Апаратне забезпечення пам'яті буде виконувати оновлення в порядку деякої черги. Кінцевим значенням `m` буде значення `a[i]`, присвоєне їй останнім процесом.

Розглянемо версію програми з використанням неподільних операцій:

```
int m = 0;
co [i = 0 to n-1]
    ⟨if (a[i] > m)
        m = a[i];⟩
```

Кутові дужки вказують, що кожен оператор `if` виконується як неподільна операція, тобто перевіряє поточне значення `m` і оновлює його в одній, неподільний дії.

На жаль, остання програма — це майже те ж саме, що і послідовна. У послідовній програмі елементи масиву `a` перевіряються в установленому порядку — від `a[0]` до `a[n-1]`. У останній програмі елементи масиву `a` перевіряються в порядку виконання процесів. Але через синхронізації перевірки все ще виконуються по одній.

Спробуємо забезпечити, щоб оновлення змінної `m` були неподільними операціями, а значення `m` було дійсно максимальним. Припустимо, що порівняння виконуються паралельно, але поновлення виробляються по одному, як в наступній програмі.

```
int m = 0;
co [i = 0 to n-1]
    if (a[i] > m)
        ⟨m = a[i];⟩
```

Ця версія програми некоректна, оскільки ця програма насправді є тим же, що і перша паралельна програма: кожен процес може порівняти своє значення елемента масиву `a` із змінною `m` і потім оновити значення змінної `m`.

Вихід полягає в поєднанні останніх двох програм. Можна безпечно виконувати паралельні порівняння, оскільки це дії, які тільки читають змінні. Але необхідно забезпечити, щоб при завершенні програми значення `m` дійсно було максимальним. Це досягається таким чином:

```
int m = 0;
co [i = 0 to n-1]
    if (a[i] > m)      # перевірка значення m
        ⟨if (a[i] > m) # повторна перевірка значення m
            m = a[i];⟩
```

Ідея полягає в тому, щоб спочатку перевірити нерівність, а потім, якщо воно виконується, провести ще одну перевірку перед оновленням значення змінної. Вона може здатися зайвою, але це не так. Наприклад, якщо певний процес оновив значення `m`, то частина інших процесів визначить, що їх значення `a[i]` менше нового значення `m`, і не буде виконувати тіло оператора `if`. Після подальших оновлень ще менше процесів визначать, що умова в першій перевірці істинна.

В розглянутих прикладах є три ключових моменти:

- синхронізація необхідна для отримання правильних результатів, якщо процеси і зчитують, і записують спільні змінні;
- неподільність дій позначається у псевдокоді кутовими дужками;
- метод подвійної перевірки перед оновленням спільної змінної часто є корисним.

5.4. Неподільні дії

Неподільна дія виконує неподільне перетворення стану. Тобто будь-який проміжний стан, який може виникнути у процесі виконання дії, є невидимим для інших процесів.

У паралельних програмах оператор присвоювання не є неподільним. Розглянемо наступний фрагмент коду:

```
int y = 0, z = 0;
co
    x = y+z;
    // y = 1; z = 2;
oc;
```

Якщо вираз `x = y + z` реалізовується за допомогою завантаження значень змінних у регістри та виконанням операції додаванням, то змінна `x` може набувати значення 0, 1, 2, 3. Це відбувається тому, що додавання може проводитися як із початковими значеннями змінних, так і з кінцевими значеннями або якоюсь їх комбінацією в залежності від того, до якої міри виконано 2-й процес. Ще однією особливістю програми є те, що *не можна* зупинити програму і побачити її стан, у якому сума `y + z` приймає значення 2.

Передбачається, що машини мають наступні характеристики:

- Значення базових типів (наприклад `int`) зберігаються в елементах пам’яті (наприклад словах), які зчитуються і записуються неподільними операціями.
- Значення обробляються так: їх поміщають в регістри, там до них застосовують операції і потім записують результати назад в пам’ять.
- Кожен процес має власний набір регістрів. Це реалізується або шляхом надання кожному процесу окремого набору регістрів, або шляхом збереження і відновлення значень регістрів при виконанні різних процесів.
- Будь-які проміжні результати, що з’являються при обчисленні складних виразів, зберігаються в регістрах або областях пам’яті, що належать процесу, що виконується, наприклад, в його стеку.

У цій моделі машини, якщо вираз в одному процесі не звертається до змінної, зміненої іншим процесом, обчислення виразу завжди буде неподільною операцією, навіть якщо для цього необхідно виконати кілька дрібномодульних дій. За тих самих умов присвоювання буде неподільною операцією.

На жаль, багато операторів в паралельних програмах, що посилаються на колективні змінні, не задовольняють цим умовам. Однак часто виконуються більш м'які умови.

Умова «не більше одного». *Критичним посиланням* у виразі називається посилання на змінну, яка змінюється іншим процесом. Оператор присвоювання `x = e` задовольняє умову «не більше одного», якщо або вираз `e` містить не більше одного критичного посилання, а змінна `x` не зчитується іншим процесом, або вираз `e` не містить критичних посилань, а інші процеси можуть зчитувати змінну `x`.

Ця умова називається «не більше одного», оскільки в такому випадку можлива лише одна спільна змінна, і на неї посилаються не більше одного разу. Аналогічне визначення застосовується до виразів, які не є операторами присвоювання. Такий вираз задовольняє умові «не більше одного», якщо він містить не більше одного критичного посилання.

Якщо оператор присвоювання задовольняє вимогам умови «не більше одного», то виконання оператора присвоювання *буде здаватися неподільної операцією,* оскільки єдина спільна змінна у виразі буде записуватися або зчитуватися тільки один раз. Наприклад, якщо вираз `e` не містить критичних посилань, а змінна `x` — проста змінна, що читається іншими процесами, то вони не можуть розпізнати, чи обчислюється вираз неподільним чином. Аналогічно, якщо `e` містить одне критичне посилання, то процес, що виконує присвоювання, не зможе розрізнити, яким чином змінюється значення змінної; він побачить тільки деяке кінцеве значення.

Наведемо кілька прикладів. У наступній програмі обидва присвоювання задовольняють умові «не більше одного»:

```
int x = 0, y = 0;
co x = x + 1; // y = y + 1; oc;
```

Тут немає критичних посилань, тому кінцевим значенням і `x`, і `y` буде 1.
Обидва присвоювання в наступній програмі також задовольняють умові:

```
int x = 0, y = 0;
co x = y + 1; // y = y + 1; oc;
```

Перший процес посилається на `y` (одне критичне посилання), але змінна `x` не читається другим процесом, і в другому процесі немає критичних посилань. Кінцевим значенням змінної `x` буде або 1, або 2, а кінцевим значенням `y` — 1. Перший процес побачить змінну `y` або перед її збільшенням, або після, але в паралельній програмі він ніколи не знає, яке значення є він бачить, оскільки порядок виконання програми недетермінований.

У наступному прикладі жодне присвоювання не відповідає вимозі «не більше одного»:

```
int x = 0, y = 0;
co x = y + 1; // y = x + 1; oc;
```

Вираз в кожному процесі містить критичне посилання, і кожен процес присвоює значення змінної, що зчитується іншим процесом. Кінцевими значеннями змінних x і y можуть бути 1 і 2, 2 і 1, або навіть 1 і 1 (якщо процеси зчитують значення змінних x і y до присвоєння їм значень). Однак, оскільки кожне присвоювання посилається тільки один раз і тільки на одну змінну, яку змінює інший процес, кінцевими будуть ті значення, які дійсно існували в деякому стані. Це відрізняється від прикладу, наведеного вище, в якому вираз y + z посилався на дві змінні, що змінюються іншим процесом.

Задача 5.1. Розглянемо програму:

```
int x = 1, y = 1;
co
    ⟨x = x + y;⟩
    // y = 0;
    // x = x - y;
oc
```

а) поясніть, чи задовольняє ця програма властивості «не більше одного»;

б) вкажіть можливі кінцеві значення змінних x та y.

5.5. Задача синхронізації: оператор очікування

Можливо, вираз або оператор присвоювання не задовольняє умові «не більше одного», проте необхідно виконати його неподільним чином. Тоді потрібен механізм синхронізації, що дозволяє задати крупномодульну неподільну дію, — послідовність дрібномодульних неподільних операцій, яка виглядає як неподільна.

Як конкретний приклад уявимо, що база даних містить два значення x і y, які завжди повинні бути однакові в тому сенсі, що жоден процес, використовує базу даних, не повинен бачити стану, в якому x і y розрізняються. Отже, якщо процес змінює x, він повинен змінити і y в тій самій неподільній дії.

Ще один приклад: нехай один процес вставляє елементи в чергу, реалізовану у вигляді зв'язаного списку. Інший процес видаляє елементи зі списку за умови, що вони там є. Вставка і видалення повинні бути неподільними діями. До того ж, якщо список порожній, необхідно відкласти виконання операції видалення до того, як в список буде вставлений елемент.

Запис ⟨e⟩ вказує, що вираз e має бути обчислений неподільним чином.

Синхронізація визначається за допомогою оператора `await`:

```
⟨await (B) S;⟩
```

Булевий вираз `B` задає *умову затримки* (delay condition), а `S` — це список послідовних операторів. Оператор `await` розташований у кутових дужках для вказівки, що він виконується як неподільна дія. Зокрема, вираз `B` гарантовано є істинним, коли починається виконання `S`, і жодний проміжний стан в `S` не видно іншим процесам. Наприклад, виконання коду

```
⟨await (s > 0) s = s-1;⟩
```

відкладається до моменту, коли значення `s` стане позитивним, а потім воно зменшується на 1. Оператор `await` є дуже потужним, оскільки може бути використаний для визначення будь-яких крупномодульних неподільних дій.

Загальна форма оператора `await` визначає як взаємне виключення, так і синхронізацію за умовою. Для визначення тільки взаємного виключення можна використовувати скорочену форму оператора `await`:

```
⟨S;⟩
```

Наприклад, в наступному операторі значення x і y збільшуються в неподільній дії:

```
⟨x = x + 1; y = y + 1;⟩
```

Проміжний стан, в якому змінна `x` була збільшена на одиницю, а `y` — ще ні, за визначенням не буде видимим для інших процесів, що посилаються на ці змінні.

Якщо потрібно виконати тільки умовну синхронізацію використовують скорочену форму оператора `await`:

```
⟨await (B);⟩
```

Наприклад, наступний оператор відкладає виконання процесу до моменту, коли значення змінної `count` стане позитивним:

```
⟨await (count > 0);⟩
```

Якщо вираз `B` задовольняє умову «не більше одного», то ⟨`await (B);`⟩ можна реалізувати як

```
while (!B);
```

Це приклад так званого *циклу очікування* (spin loop). Тіло оператора `while` порожнє, тому він просто зациклюється до тих пір, поки значенням `B` не стане `true`.

Безумовна неподільна дія — це дія, яке не містить в тілі умови затримки `B`. Така дія може бути виконаною негайно. *Умовна неподільна дія* — це оператор `await` з умовою `B`. Така дія не може бути виконаною, поки умова `B` не стане істинною. Якщо `B` хибне, воно може стати істинним тільки в результаті дій інших процесів. Процес, який чекає виконання умовного неподільної дії, може виявитися затриманим непередбачувано довго.

Задача 5.2. Розглянемо наступну програму:

```
int x = 2, y = 3;
co
    ⟨x = x + y;⟩
    // ⟨y = x * y;⟩
oc
```

а) вкажіть можливі кінцеві значення змінних `x` та `y`;

б) припустимо, що кутові дужки прибрані, і кожний оператор присвоювання реалізується трьома неподільними діями: зчитування змінної, додавання або множення та запис змінної. Якими тепер можуть бути кінцеві значення змінних `x` та `y`?

5.6. Синхронізація типу «виробник-споживач»

Дано два процеси: `Producer` і `Consumer`. Процес `Producer` має локальний масив цілих чисел `a[n]`; `Consumer` — `b[n]`. Мета — скопіювати вміст масиву `a` в масив `b`. Нехай змінна `buf` — це спільна цілочислова змінна, яка використовується у якості буфера взаємодії. Процеси `Producer` і `Consumer` повинні отримувати доступ до змінної `buf` по черзі. Нехай спільні змінні `p` та `c` — лічильники числа поміщених і вибраних елементів, відповідно. Їх початкові значення — 0. Тоді умови синхронізації процесів `Producer` і `Consumer` можуть бути записані в наступному вигляді:

```
PC: c <= p <= c + 1
```

Процеси `Producer` і `Consumer` використовують змінні `p` та `c` для синхронізації доступу до буфера `buf`. Оператори `await` застосовуються для припинення процесів до тих пір, поки буфер не стане повним або порожнім. Якщо істинна умова `p == c`, то буфер порожній (останній поміщений в нього елемент був вибраний), а якщо `p > c` — заповнений. Якщо синхронізація реалізується описаним способом, кажуть, що процес знаходиться в *стані активного очікування*, або *зациклений*, оскільки він зайнятий перевіркою умови в операторі `await`, але все, що він робить, — це повторення циклу до тих пір, поки умова не виконається. Це звичайний тип синхронізації, необхідний на найнижчих рівнях програмних систем, наприклад, в операційних системах:

```
int buf, p = 0, c = 0;
process Producer {
    int a[n];
    while (p < n) {
        ⟨await (p == c);⟩
        buf = a[p];
        p = p + 1;
    }
}
process Consumer {
    int b[n];
    while (c < n) {
        ⟨await (p > c);⟩
        b[c++] = buf;
    }
}
```

5.7. Стратегії планування і справедливість

Більшість властивостей живучості залежить від *справедливості* (*fairness*), пов'язаної з гарантіями того, що кожен процес отримує шанс на продовження виконання незалежно від дій інших процесів. *Стратегія планування* (*scheduling policy*) визначає, яку неподільну дію буде виконано наступною. Далі буде розглянуто три ступеня справедливості, які можуть бути забезпечені стратегією планування.

5.7.1. Безумовна справедливість

Розглянемо наступну просту програму:

```
bool continue = true;
co while (continue);
// continue = false;
oc
```

Припустимо, що стратегія планування призначає процесор для процесу до тих пір, поки той не завершиться, чи не буде припинений. У випадку одного процесора дана програма не завершиться, якщо спочатку буде виконуватися перший процес. Однак, якщо 2-й процес врешті-решт отримає право на виконання, програма буде завершена. Дана ситуація відображена в наступному визначенні:

Стратегія планування *безумовно справедлива*, якщо *будь-яка безумовна неподільна дія врешті-решт виконується*.

Для програми, наведеної вище, безумовно справедливою стратегією планування на одному процесорі було б *циклічне* (round-robin) виконання, а на мультипроцесорі — *синхронне*.

5.7.2. Слабка справедливість

Якщо програма містить умовну неподільну дію (оператор `await` з логічною умовою `B`), необхідно робити більш сильні припущення, щоб гарантувати просування процесу. Причина в тому, що умовна неподільна дія не почне виконуватися, поки умова `B` не стане істинною.

Стратегія планування *справедлива в слабкому сенсі*, якщо:

1) вона безумовно справедлива;
2) кожна умовна неподільна дія врешті-решт виконується, якщо її умова стає і потім залишається істинною, поки її бачить процес, який виконує умовну неподільну дію.

Таким чином, якщо для дії ⟨`await (B) S;`⟩ умова `B` стає істинною, то `B` залишається істинною принаймні до закінчення виконання умовної неподільної дії. Циклічна стратегія і тактика квантування часу є справедливими в слабкому сенсі, якщо кожному процесу дається можливість виконання. Причина в тому, що будь-який призупинений процес врешті-решт побачить, що умова закінчення його затримки істинна.

5.7.3. Сильна справедливість

Справедливості в слабкому сенсі, однак, недостатньо для гарантії, що будь-який оператор `await` в кінці кінців виконається. Це пов'язано з тим, що умова може змінити своє значення (від хибного до істинного і навпаки), поки процес припинений. У цьому випадку необхідна сильніша стратегія планування.

Стратегія планування *справедлива в сильному сенсі*, якщо:

1) вона безумовно справедлива;
2) будь-яка умовна неподільна дія в кінці кінців виконується в припущенні, що її умова буває істинною нескінченно часто.

Умова буває істинною нескінченно часто, якщо вона істинна нескінченне число разів у кожній історії програми (що не закінчується). Щоб стратегія планування була справедливої в сильному сенсі, дія повинна вибиратися для виконання не тільки тоді, коли її умова хибна, але і тоді, коли вона істинна.

Щоб побачити відмінності між справедливістю в слабкому і сильному сенсі, розглянемо наступну програму.

```
bool continue = true, try = false;
co
    while (continue){
        try = true;
        try = false;
    }
```

```
// ⟨await (try) continue = false;⟩
oc
```

При стратегії, справедливій в сильному сенсі, ця програма в кінці кінців завершиться, оскільки значення змінної `try` нескінченно часто істинно. Однак при стратегії планування, справедливої в слабкому сенсі, програма може не завершитися, оскільки значення змінної `try` також нескінченно часто є хибним.

На жаль, *неможливо розробити стратегію планування процесора, яка була б і практичною, і справедливою в сильному сенсі* [6]. Ще раз розглянемо програму, наведену вище. На одному процесорі диспетчер, який чергує дії двох процесів, буде справедливим в сильному сенсі, оскільки другий процес буде бачити стан, в якому значення змінної `try` істинно. Циклічне планування і планування з квантуванням часу практичні, але в загальному випадку не є справедливими в сильному сенсі, оскільки процеси виконуються в непередбачуваному порядку. Диспетчер мультипроцесора, виконуючий процеси паралельно, також практичний, але не є справедливим в сильному сенсі. Причина в тому, що другий процес може перевіряти значення змінної `try` тільки тоді, коли воно хибне. Це, звичайно, малоймовірно, але теоретично можливо.

Повернемося до програми копіювання масиву, наведеної у підрозділі 5.6. (синхронізація типу «виробник-споживач»). Ця програма вільна від блокувань. Таким чином, програма буде завершена, оскільки кожен процес регулярно отримує можливість просунутися в своєму виконанні. Процес буде просуватися, оскільки стратегія справедлива в слабкому сенсі. Справа в тому, що, коли один процес робить істинним умову закінчення затримки іншого процесу, ця умова залишається істинною, поки інший процес не буде продовжений і не змінить спільні змінні.

Задача 5.3. Розглянемо наступну програму:

```
int x;
co
    ⟨await (x >= 3) x = x - 2;⟩
    // ⟨await (x >= 2) x = x - 3;⟩
    // ⟨await (x == 1) x = x + 5;⟩
oc
```

Для яких початкових значень змінної `x` програма завершиться, якщо застосовується справедлива у слабкому сенсі стратегія планування? Якими будуть кінцеві значення `x`?

6. БЛОКУВАННЯ ТА БАР’ЄРИ

6.1. Задача критичної секції

Задача критичної секції (КС) — це одна з класичних задач паралельного програмування. Вона була першою всебічно вивченою задачею, але інтерес до неї не згасає, оскільки критичні секції коду є в більшості паралельних програм. Крім того, розв’язок цієї задачі можна використовувати для реалізації операторів `await`.

У задачі критичної секції `n` процесів багаторазово виконують спочатку критичну, а потім некритичну секцію коду. Критичній секції передує протокол входу, а за нею йде протокол виходу. Таким чином, передбачається, що процес має наступний вигляд:

```
process CS [i = 1 to n] {
    while (true) {
        протокол входу;
        критична секція;
        протокол виходу;
        некритична секція;
    }
}
```

Кожна критична секція є послідовністю операторів, що мають доступ до деякого спільного об’єкта. Кожна некритична секція — це ще одна послідовність операторів. Передбачається, що процес, який увійшов в критичну секцію, обов’язково коли-небудь з неї вийде; таким чином, процес може завершитися тільки поза критичною секцією. Завдання — розробити протоколи входу і виходу, які задовольняють наступним властивостям:

(1) **Взаємне виключення**. У будь-який момент часу тільки один процес може виконувати свою критичну секцію.

(2) **Відсутність взаємного блокування (живого блокування)**. Якщо кілька процесів намагаються увійти в свої критичні секції, хоча б один це здійснить.

(3) **Відсутність зайвих затримок**. Якщо один процес намагається увійти в свою критичну секцію, а інші виконують свої некритичні секції або завершені, першому процесу дозволяється вхід в критичну секцію.

(4) **Можливість входу**. Процес, який намагається увійти в критичну секцію, коли-небудь це зробить.

Перші три властивості є властивостями безпеки, четверта — властивістю живучості.

Тривіальний спосіб вирішення задачі критичної секції полягає в обмеженні кожної критичної секції кутовими дужками, тобто у використанні безумовних операторів await. З семантики кутових дужок відразу випливає умова (1) — взаємне виключення. Інші три властивості задовольнятимуться при безумовно справедливій стратегії планування, оскільки вона гарантує, що процес, який намагається виконати неподільну дію, відповідну його критичної секції, в кінці кінців це зробить, незалежно від дій інших процесів. Однак при такому «розв'язку» виникає питання про те, як реалізувати кутові дужки.

Всі чотири зазначених властивості важливі, однак найбільш істотним є взаємне виключення. Спочатку ми зосередимося на ньому, а потім дізнаємося, як забезпечити виконання інших властивостей. Для опису властивості взаємного виключення необхідно визначити, чи знаходиться процес у своїй критичній секції. Щоб спростити запис, розглянемо розв'язок задачі у випадку двох процесів, CS1 і CS2; він легко узагальнюється для n процесів.

Нехай in1 та in2 — логічні змінні з початковим значенням false. Якщо процес CS1 (CS2) знаходиться в своїй критичній секції, змінній in1 (in2) присвоюється значення true. Поганий стан, якого ми будемо намагатися уникнути, — якщо обидві змінні in1 та in2 є істинними. Таким чином, нам потрібно, щоб для будь-якого стану виконувалося заперечення умови поганого стану:

```
MUTEX: !(in1 && in2)
```

Предикат MUTEX має бути глобальним інваріантом. Для цього він повинен виконуватися в початковому стані і після кожного присвоювання змінним in1 та in2. Зокрема, перед тим, як процес CS1 увійде в критичну секцію, зробивши тим самим in1 істинною, він повинен переконатися, що in2 хибна. Це можна реалізувати так:

```
⟨await (!in2) in1 = true;⟩
```

Вхідний протокол процесу CS2 аналогічний. При виході з критичної секції затримуватися ні до чого, тому захищати оператори, які надають змінним in1 та in2 значення false, немає необхідності.

Розв'язок наведено у наступній програмі:

```
bool in1 = false, in2 = false;
process CS1 {
    while (true) {
        ⟨await (!in2) in1 = true;⟩    # вхід
        критична секція;
        in1 = false;                   # вихід
        некритична секція;
    }
}
```

```
process CS2 {
    while (true) {
        ⟨await (!in1) in2 = true;⟩    # вхід
        критична секція;
        in2 = false;                   # вихід
        некритична секція;
    }
}
```

За побудовою програма задовольняє умові взаємного виключення. Взаємне блокування тут не виникне: якби кожен процес був заблокований в своєму протоколі входу, то обидві змінні, і `in1`, і `in2`, були б істинними, а це суперечить тому, що в даній точці коду обидві вони хибні. Зайвих затримок також немає, оскільки один процес блокується, тільки якщо інший перебуває в критичній секції, тому небажані паузи при виконанні програми не виникають.

Нарешті, розглянемо властивість живучості: процес, який намагається увійти в критичну секцію, в кінці кінців зможе це зробити. Якщо процес `CS1` намагається увійти, але не може, то змінна `in2` істинна, і процес `CS2` знаходиться в критичній секції. За припущенням процес врешті-решт виходить з критичної секції, тому змінна `in2` коли-небудь стане хибною, а змінна захисту входу процесу `CS1` — істинної.

Якщо процесу `CS1` вхід все ще не дозволено, це означає, що або диспетчер несправедливий, або процес `CS2` знову досяг входу в критичну секцію. В останньому випадку описаний вище сценарій повторюється, так що коли-небудь змінна `in2` стане хибною. Таким чином, або змінна `in2` стає хибною нескінченно часто, або процес `CS2` завершується, і змінна `in2` приймає значення `false` і залишається в такому стані. Для того щоб процес `CS1` в будь-якому випадку входив в критичну секцію, потрібно забезпечити справедливу в сильному сенсі стратегію планування.

6.2. Критичні секції: активні блокування

У крупномодульному розв'язку, наведеному в попередній програмі, використовуються дві змінні. При узагальненні для випадку `n` процесів знадобиться `n` змінних. Однак існує тільки два цікаві для нас стани: або певний процес знаходиться в своїй критичній секції, або жодного там немає. Незалежно від числа процесів, для того, щоб розрізнити ці два стани, досить однієї змінної.

Нехай `lock` — логічна змінна, яка показує, чи знаходиться процес в критичній секції, тобто

```
lock == (in1 || in2)
```

Використовуючи `lock`, можна реалізувати протоколи входу і виходу так:

```
# критичні секції на основі блокування
bool lock = false;
process CS1 {
    while (true) {
        ⟨await (!lock) lock = true;⟩  # вхід
        критична секція;
        lock = false;                 # вихід
        некритична секція;
    }
}
process CS2 {
while (true) {
    while (true) {
        ⟨await (!lock) lock = true;⟩  # вхід
        критична секція;
        lock = false;                 # вихід
        некритична секція;
    }
}
```

Перевага останньої програми полягає в тому, що її можна використовувати для вирішення задачі критичної секції при будь-якому числі процесів. Всі вони будуть спільно використовувати змінну `lock` і виконувати одні і ті ж протоколи.

Використання змінної `lock` замість `in1` та `in2` дуже важливе, оскільки майже у всіх машин, особливо у мультипроцесорів, є спеціальна інструкція для реалізації умовних неподільних дій. Тут застосовується інструкція, яка називається «перевірити-встановити» (test and set — `TS`), в якості аргументу отримує змінну `lock` і повертає логічне значення. У неподільний дії інструкція `TS` зчитує і зберігає значення змінної `lock`, присвоює їй значення `true`, а потім повертає збережене попереднє значення змінної `lock`. Результат дії інструкції `TS` описується наступною функцією:

```
bool TS(bool lock) {
    ⟨bool initial = lock; # зберегти початкове значення
     lock = true;         # встановити lock
     return initial; ⟩    # повернути початкове значення
}
```

Використовуючи інструкцію `TS`, можна реалізувати крупномодульний варіант програми для задачі критичної секції. Умовні неподільні дії замінюються циклами. Цикли не закінчуються, поки змінна `lock` не стане хибною, тобто інструкція `TS` повертає значення «фальш». Наведений розв’язок працює при будь-якому числі процесів. Використання блокуючої змінної, зазвичай називається *циклічним блокуванням* (*spin lock*), оскільки процес постійно повторює цикл, очікуючи зняття блокування.

```
# критичні секції на основі інструкції TS

bool lock = false; # колективна змінна
process CS [i = 1 to n] {
    while (true) {
        while (TS(lock)) skip;    # протокол входу
        критична секція;
        lock = false;             # протокол виходу
        некритична секція;
    }
}
```

Взаємне виключення (1) забезпечено: якщо кілька процесів намагаються увійти в критичну секцію, тільки один з них першим змінить значення змінної `lock` з хибного на істинне, отже, тільки один з процесів успішно завершить свій вхідний протокол і увійде в критичну секцію. Відсутність взаємного блокування (2) випливає з того, що, якщо обидва процеси знаходяться в своїх вхідних протоколах, то значення `lock` хибне, і, отже, один з процесів увійде в свою критичну секцію. Небажані затримки (3) не виникають, оскільки, якщо обидва процеси виходять за межу своїх критичних секцій, `lock` хибна, і, отже, один з процесів може успішно увійти в критичну секцію, якщо інший виконує некритичну секцію або був завершений.

З іншого боку, виконання властивості можливості входу (4) не гарантовано. Якщо використовується справедлива у сильному сенсі стратегія планування, то спроби процесу увійти в критичну секцію завершаться успіхом, оскільки змінна `lock` нескінченно часто буде приймати значення «фальш». При справедливою в слабкому сенсі стратегії планування, яка зустрічається найчастіше, процес може назавжди зациклитися в протоколі входу. Однак це може статися, тільки якщо інші процеси весь час успішно входять в свої критичні секції, чого на практиці бути не повинно. Отже, наведений розв'язок з великою вірогідністю задовольняє умові справедливої стратегії планування.

Побудований розв'язок з циклічної блокуванням має суттєву властивість:

у розв'язку задачі критичної секції з циклічної блокуванням протокол виходу повинен присвоювати спільним змінним їх початкові значення.

У початковому стані обидві змінні `in1` та `in2` хибні, як і змінна `lock` (див. 6.1, 6.2).

Хоча остання наведена програма логічно вірна, експерименти на мультипроцесорних машинах показують її низьку продуктивність, якщо кілька процесів змагаються за доступ до критичної секції. Причина в тому, що кожен призупинений процес безперервно звертається до змінної `lock`. Ця «гаряча точка» *викликає конфлікт при зверненні до пам'яті*, який знижує продуктивність модулів пам'яті і шин, що зв'язують процесор і пам'ять.

До того ж інструкція `TS` при кожному виклику *записує значення в* `lock`, *навіть якщо воно не змінюється*. Оскільки в мультипроцесорних машинах для зменшення числа звернень до основної пам'яті використовуються кеші, `TS` виконується набагато довше, ніж просте читання значення спільної змінної. (Коли змінна записується одним з процесорів, її копії потрібно оновити або зробити недійсними в кешах інших процесорів.)

Витрати на оновлення вмісту кеш-пам'яті і конфлікти при зверненні до пам'яті можна скоротити. Для цього можна використовувати наступний протокол входу:

```
# КС на основі протоколу «перевірити-перевірити-встановити»
bool lock = false
process CS [i = 1 to n] {
    while (lock) skip; # поки lock встановлена, повторювати
    while (TS(lock)) {     # спробувати захопити lock
        while (lock) skip; # повторювати, якщо не вдалося
    }
    критична секція
    lock = false;          # протокол виходу
    некритична секція;
}
```

Цей протокол називається «*перевірити-перевірити-встановити*», оскільки процес просто перевіряє `lock` до тих пір, поки не з'явиться можливість виконання `TS`. У двох додаткових циклах `lock` просто перевіряється, так що її значення можна прочитати з кеш-пам'яті, не впливаючи на інші процесори. Таким чином, конфлікти при зверненні до пам'яті скорочуються (але не зникають). Якщо прапорець блокування `lock` не встановлений, то як мінімум один, а можливо, і всі призупинені процеси можуть виконати інструкцію `TS`, хоча продовжувати роботу буде тільки один з них.

6.3. Реалізація операторів `await`

Будь-який розв'язок задачі критичної секції можна використовувати для реалізації безумовної неподільної дії ⟨`S;`⟩. Нехай `CSenter` — вхідний протокол критичної секції, а `CSexit` — вихідний. Тоді дію ⟨`S;`⟩ можна реалізувати так:

```
CSenter;
S;
CSexit;
```

Тут передбачається, що всі секції коду процесів, які змінюють або посилаються на змінні, що змінюються в `S` (або змінюють змінні, на які посилається `S`), захищені аналогічними вхідними та вихідними протоколами. По суті, кутові дужки замінені процедурами `CSenter` та `CSexit`.

Цей підхід можна використовувати для реалізації оператора ⟨`await (B) S;`⟩. Щоб забезпечити неподільність усієї дії, можна використовувати протокол критичної секції, приховуючи проміжні стани у `S`. Для циклічної перевірки умови `B`, поки вона не стане істинною, можна використовувати наступний цикл:

```
CSenter;
while (!B) {???}
S;
CSexit;
```

Тут передбачається, що критичні секції всіх процесів, що змінюють змінні, які використовуються у `B` або `S`, або використовують змінні, що змінюються в `S`, захищені такими ж протоколами входу і виходу.

Залишається з'ясувати, як реалізувати тіло циклу, зазначеного вище. Якщо тіло виконується, значить, умова `B` була хибною. Отже, єдиний спосіб зробити умову `B` істинною — змінити в іншому процесі значення змінних, що входять в цю умову. Передбачається, що всі оператори, що змінюють ці змінні, знаходяться в критичних секціях, тому, чекаючи, поки умова `B` виконається, потрібно вийти з критичної секції. Але для забезпечення неподільності обчислення `B` і виконання `S` перед повторним обчисленням умови `B` необхідно знову увійти в критичну секцію. Можливим уточненням зазначеного вище протоколу може бути:

```
CSenter;
while (!B){
    CSexit;
    CSenter;
}
S;
CSexit;
```

Дана реалізація зберігає семантику умовних неподільних дій за умови, що протоколи критичних секцій гарантують взаємне виключення.

Попередня програма вірна, але не ефективна, оскільки процес, що виконує її, повторює «жорсткий» цикл, постійно виходячи з критичної секції і входячи в неї, але не може просунутися далі, поки який-небудь інший процес не змінить змінних в умові B. Це призводить до конфлікту звернення до пам'яті, оскільки кожен призупинений процес постійно звертається до змінних, що використовуються в протоколах критичної секції і умови B.

Щоб скоротити кількість конфліктів звернення до пам'яті, процес перед повторною спробою увійти в критичну секцію повинен робити паузу. Нехай Delay — деякий код, уповільнюючий виконання процесу. Тоді програму можна замінити наступним протоколом, який реалізує умовну неподільну дію.

```
CSenter;
while (!B){
    CSexit;
    Delay;
    CSenter;
}
S;
CSexit;
```

Кодом Delay може бути, наприклад, порожній цикл, який виконується випадкове число разів. (Щоб уникнути конфліктів пам'яті в коді Delay слід використовувати лише локальні змінні.) Цей тип протоколу «відходу» ("back-off") корисний і в самих протоколах CSenter, наприклад, його можна використовувати замість skip в циклі затримки простого протоколу «перевірити-встановити».

Синхронізація з активним очікуванням часто застосовується в апаратному забезпеченні. Фактично протокол, аналогічний до наведеного у останній програмі, використовується для синхронізації доступу в локальних мережах Ethernet. Щоб передати повідомлення, контролер Ethernet надсилає його в мережу і стежить, не виник при передачі конфлікт з повідомленнями, надісланим приблизно в цей же час іншими контролерами. Якщо конфлікт не виявлено, то вважається, що передача завершилася успішно. В іншому випадку контролер робить невелику паузу, а потім повторює передачу повідомлення. Щоб уникнути стану гонитви, в якому два контролера постійно конфліктують через те, що роблять однакові паузи, їх тривалість вибирається випадковим чином з інтервалу, який подвоюється при кожному виникненні конфлікту. Такий протокол називається *двійковим експоненціальним протоколом відходу.*

6.4. Критичні секції: розв'язок із справедливою стратегією

Розв'язок задачі критичної секції з циклічним блокуванням забезпечують взаємне виключення, відсутність взаємних блокувань, активних тупиків і небажаних пауз. Проте для забезпечення властивості можливості входу (4) їм необхідна справедлива в сильному сенсі стратегія планування. Стратегії планування, які застосовуються на практиці, є справедливими лише в слабкому сенсі.

Малоймовірно, що процес, який намагається увійти в критичну секцію, ніколи цього не зробить, однак може статися, що кілька процесів будуть без кінця змагатися за вхід. Зокрема, розв'язки з циклічним блокуванням не керують порядком, в якому кілька призупинених процесів намагаються увійти в критичні секції.

Нижче наведено три розв'язки задачі критичної секції із справедливою стратегією планування: алгоритми розриву вузла, полікліники та квитка. Вони залежать тільки від справедливої в слабкому сенсі стратегії планування при якому кожен процес періодично отримує можливість виконання, а умови затримки, ставши справжніми, залишаються такими. Алгоритм розриву вузла досить простий для двох процесів і не залежить від спеціальних машинних інструкцій, але складний для *n* процесів. Алгоритм квитка простий для будь-якого числа процесів, але вимагає спеціальної інструкції «отримати і додати». Алгоритм полікліники — це варіант алгоритму квитка, для якого не потрібні спеціальні машинні інструкції.

6.4.1. Алгоритм розриву вузла

Розглянемо розв'язок задачі критичної секції для двох процесів. Його недолік в тому, що у ньому не вирішено, який із процесів, які намагаються увійти в критичну секцію, туди дійсно потрапить. Наприклад, один процес може увійти в критичну секцію, виконати її, потім повернутися до протоколу входу і знову успішно увійти в критичну секцію. Щоб розв'язок був справедливим, треба дотримуватися черговості входу в критичну секцію, якщо кілька процесорів намагаються туди увійти.

Алгоритм розриву вузла (також відомий як алгоритм Пітерсона) — це варіант протоколу критичної секції, який «розриває вузол», коли два процеси намагаються увійти в критичну секцію. У алгоритмі використовується додаткова змінна `last` — цілочислова змінна, яка показує, який із процесів `CS1` і `CS2` почав виконувати протокол входу останнім. Якщо обидва процеси намагаються увійти в критичні секції, тобто `in1` і `in2` істинні, виконання останнього з них призупиняється.

```
bool in1 = false, in2 = false;
int last = 1;
process CS1 {
    while (true) {
        last = 1; in1 = true;         # вхід
        ⟨await (!in2 || last == 2);⟩
        критична секція;
        in1 = false;                  # вихід
        некритична секція;
    }
}
```

```
process CS2 {
    while (true) {
        last = 2; in2 = true;          # вхід
        ⟨await (!in1 || last == 1);⟩
        критична секція;
        in1 = false;                   # вихід
        некритична секція;
    }
}
```

Алгоритм програми дуже близький до дрібномодульних розв'язків, для якого не потрібні оператори `await`. Зокрема, якщо всі оператори `await` задовольняють умові «не більше одного» то їх можна реалізувати у вигляді циклів активного очікування. На жаль, оператори `await` звертаються до двох змінним, кожну з яких змінює інший процес. Однак в даному випадку немає необхідності в неподільному обчисленні умов затримки [6], а тому кожний оператор `await` можна замінити циклом `while`, який повторюється, поки умова закінчення затримки хибна. Таким чином, отримуємо дрібномодульний алгоритм розриву вузла:

```
# дрібномодульний алгоритм розриву вузла
bool inl = false, in2 = false;
int last = 1;
process CS1 {
    while (true) {
        last = 1; in1 = true;          # вхід
        while (in2 && last == 1) skip
        критична секція;
        in1 = false;                   # вихід
        некритична секція;
    }
}
process CS2 {
    while (true) {
        last = 2; in2 = true;          # вхід
        while (in1 && last == 1) skip
        критична секція;
        in1 = false;                   # вихід
        некритична секція;
    }
}
```

У цій програмі вирішується проблема критичних секцій для двох процесів. Таку ж ідею можна використовувати при будь-якому числі процесів. Зокрема, для кожного з n процесів протокол входу повинен складатися з циклу, який проходить n - 1 етапів. На кожному етапі використовуються екземпляри алгоритму розриву вузла для двох процесів, щоб визначити, які процеси проходять на наступний етап. Якщо гарантується, що всі n-1 етапів може пройти не більше, ніж один процес, то в критичній секції одночасно буде знаходитися не більше одного процесу.

Нехай in[1: n] та last[1: n] — цілочислові масиви. Значення елемента in[i] показує, який етап виконує процес CS[i]. Значення last[j] показує, який процес останнім почав виконувати етап j. Внутрішній цикл по j процесу CS[i] перевіряє всі інші процеси. Процес CS[i] чекає, якщо деякий інший процес знаходиться на етапі з рівним або більшим номером етапу, а процес CS[i] був останнім процесом, що досягли етапу j. Як тільки етапу j досягне ще один процес, або всі процеси «перед» процесом CS[i] вийдуть зі своїх критичних секцій, процес CS[i] отримає можливість виконуватися на наступному етапі.

```
int in[1: n]; int last[1: n]; # масиви з нульовими елементами
process CS[i = 1 to n] {
    while (true) {
        for [j = 1 to n] # протокол входу
            # процес i знаходиться на етапі j та є там останнім
            last[j] = i; in[i] = j;
            for [k = 1 to n && i != k] {
            /* Чекати, якщо процес k знаходиться на етапі з
            більшим номером та процес i був останнім з
            досягших етап j* /
                while (in[k] > = in[i] && last[j] == i) skip;
            }
        }
        критична секція;
        in[i] = 0; # Протокол виходу
        некритична секція;
    }
}
```

Таким чином, не більше n-1 процесів можуть пройти перший етап, n-2 — другий і так далі. Це гарантує, що пройти всі n етапів і виконувати свою критичну секцію процеси можуть тільки по одному.

Розв’язок для n процесів уникає стан активного тупика, непотрібні затримки і гарантує можливість входу. Ці властивості випливають з того, що даний процес затримується, тільки якщо деякий інший процес знаходиться в протоколі входу попереду даного, і з припущення, що кожен процес врешті-решт виходить зі своєї критичної секції.

6.4.2. Алгоритм квитка

Алгоритм розриву вузла для *n* процесів достатньо складний та малозрозумілий. Алгоритм квітка пропонує більш прозорий розв'язок задачі. Назва пов'язана із тим, що він заснований на витягуванні квитків (номерів) та наступному очікуванні черги.

Нехай `number` та `next` — цілі змінні з початковими значеннями 1, а `turn[1: n]` — масив цілих чисел, початкові значення яких рівні нулю. Щоб увійти в критичну секцію, процес `CS[i]` спочатку присвоює елементу `turn[i]` поточне значення `number` та збільшує `number` на 1. Для того, щоб процеси отримували унікальні номери, ці дії мають бути неподільні. Після цього `CS[i]` очікує, поки значення `next` не стане рівним отриманому ним номеру. При завершенні критичної секції `CS[i]` неподільною дією збільшує на 1 значення `next`.

```
# Алгоритм квитка — крупномодульний розв'язок
int number = 1, next = 1, turn[1:n];
process CS[i = 1 to n] {
    while (true) {
        ⟨turn[i] = number; number = number + 1;⟩
        ⟨await (turn[i] == next);⟩
        критична секція;
        ⟨next = next + 1;⟩
        некритична секція;
    }
}
```

У попередній програмі значення елементів масиву `turn` унікальні. Оператор затримки

```
⟨await (turn[i] == next);⟩
```

гарантує, що тільки один `turn[i]` рівний `next`, тобто тільки один процес може знаходитися у своїй критичній секції.

Алгоритм квитка має потенційний недолік: значення `number` та `next` не обмежені, що може призвести до арифметичного переповнення.

У програмі використовуються три крупномодульні дії. Деякі процесори мають інструкції, які повертають попереднє значення змінної та збільшують (зменшують) її в одній неподільній дії. Прикладом є інструкція «отримати та збільшити» `FA(var, incr)` (Fetch-and-Add), з використанням якої можна отримати наступну версію програми:

```
# Алгоритм квитка — дрібномодульний розв'язок
int number = 1, next = 1, turn[1:n];
process CS[i = 1 to n] {
    while (true) {
        turn[i] = FA(number,1);          # протокол входу
        while (turn[i] != next) skip;
        критична секція;
        next = next + 1;                 # протокол виходу
        некритична секція;
    }
}
```

Оператор `next = next + 1` без кутових дужок, що є допустимим, оскільки значення `next` змінюється тільки тим єдиним процесом, який увійшов у критичну секцію.

6.4.3. Алгоритм поліклініки

Алгоритм квитка можна безпосередньо реалізовувати лише на машинах із інструкцією «отримати та збільшити». *Алгоритм поліклініки*, схожий на алгоритм квитка. Він забезпечує справедливість планування і не вимагає спеціальних машинних інструкцій. За алгоритмом квитка кожен відвідувач отримує унікальний номер і чекає, поки значення `next` не стане рівним цьому номеру. Алгоритм поліклініки використовує інший підхід. Входячи, відвідувач дивиться на всіх інших і вибирає номер, більший за будь-який інший. Всі відвідувачі повинні чекати, поки назвуть їх номер. Як і в алгоритмі квитка, наступним обслуговується відвідувач з найменшим номером. Відмінність полягає в тому, що для визначення черговості обслуговування відвідувачі порівнюють номери один одного та не використовують загальний лічильник.

Як і в алгоритмі квитка, нехай `turn[1: n]` — цілочисловий масив з початковими значеннями 0. Щоб увійти в критичну секцію, процес `CS[i]` спочатку присвоює змінної `turn[i]` значення, яке на 1 більше, ніж максимальне серед поточних значень елементів масиву `turn`. Потім `CS[i]` очікує, поки значення `turn[i]` не стане найменшим серед ненульових елементів масиву `turn`. Виходячи з критичної секції, процес `CS[i]` присвоює `turn[i]` значення `0`.

```
# Алгоритм поліклініки — крупномодульний розв'язок
int turn[1:n];
process CS[i = 1 to n] {
    while (true) {
        ⟨turn[i] = max(turn[1:n]) + 1;⟩
        for [j = 1 to n && j != i]
```

```
            ⟨await (turn[j] == 0 || turn[i] < turn[j]);⟩
        критична секція;
        turn[i] = 0;
        некритична секція;
    }
}
```

Перша неподільна дія забезпечує унікальність всіх ненульових значень у масиві `turn`.

Ненульові значення елементів масиву `turn` унікальні і, як зазвичай, передбачається, що кожен процес врешті-решт виходить зі своєї критичної секції, тому взаємних блокувань немає. Відсутні також зайві затримки процесів, оскільки відразу після виходу процесу `CS[i]` з критичної секції `turn[i]` набуває значення `0`. Нарешті, алгоритм поліклініки гарантує можливість входу в критичну секцію, якщо планування справедливе в слабкому сенсі. Значення елементів масиву `turn` продовжують зростати, тільки якщо завжди є хоча б один процес, який намагається увійти в критичну секцію. Алгоритм поліклініки можна безпосередньо реалізувати на сучасних машинах. Щоб присвоїти значення `turn[i]`, необхідно знайти максимальне з `n` значень, а оператор `await` двічі звертається до спільної змінної `turn[j]`. Ці операції можна було б реалізувати неподільним чином, використовуючи ще один протокол критичної секції, наприклад, алгоритм розриву вузла, але це занадто неефективно. Але є більш простий вихід. Якщо необхідно синхронізувати `n` процесів, корисно спочатку розробити розв'язок для двох процесів, а потім узагальнити його (так робилося у випадку алгоритму розриву вузла).

Розглянемо наступний протокол входу для процесу CS1:

```
turn1 = turn2 + 1;
while (turn2 ! = 0 and turn1 > turn2) skip;
```

Аналогічний і наступний протокол входу для процесу CS2.

```
turn2 = turn1 + 1;
while (turn 1! = 0 and turn2 > turn1) skip;
```

Процеси можуть почати виконання своїх протоколів входу приблизно одночасно, і обидва присвоять змінним `turn1` і `turn2` значення `1`. Якщо це трапиться, обидва процеси виявляться в своїх критичних секціях в один і той же час. Частково вирішити цю проблему можна за аналогією з дрібномодульним алгоритмом розриву вузла для двох процесів: якщо обидві змінні `turn1` та `turn2` мають значення `1`, то один з процесів повинен виконуватися, а інший — припинятися.

На жаль, обидва процеси все ще можуть одночасно виявитися в критичній секції. Припустимо, що процес `CS1` зчитує значення `turn2` і отримує `0`. Процес `CS2` починає виконувати свій протокол входу, визначає, що змінна `turn1` все ще має значення 0, присвоює `turn2` значення `1` і входить в критичну секцію. У цей момент `CS1` може продовжити виконання свого протоколу входу, присвоїти `turn1` значення 1 і потім увійти в критичну секцію, оскільки змінні `turn1` і `turn2` мають значення 1, і процес `CS1` в цьому випадку отримує перевагу. Така ситуація називається *станом гонитви* (*race condition*), оскільки процес `CS1` «обганяє» `CS2` і не враховує, що процес `CS2` змінив змінну `turn2`. Щоб уникнути стану гонитви, необхідно, щоб кожен процес присвоював своїй змінній `turn` значення 1 (або будь-яке відмінне від нуля) на самому початку протоколу входу. Після цього процес повинен перевірити значення змінної `turn` інших процесів і переприсвоїти значення своєї змінної, тобто протокол входу процесу `CS1` виглядає наступним чином.

```
turn1 = 1; turn1 = turn2 + 1;
while (turn2 ! = 0 && turn1 > turn2) skip;
```

Протокол входу процесу `CS2` аналогічний.

```
turn2 = 1; turn2 = turn1 + 1;
while (turn1 ! = 0 && turn2 >= turn1) skip;
```

Тепер один процес не може вийти з циклу `while`, поки інший не закінчить розпочате раніше присвоювання `turn`. У цьому розв'язку процесу `CS1` віддається перевага перед `CS2`, коли у обох процесів ненульові значення змінної `turn`.

Протоколи входу процесів несиметричні, оскільки умова затримки другого циклу трохи відрізняється. Однак їх можна записати і в симетричному вигляді. Нехай `(a, b)` та `(c, d)` — пари цілих чисел. Визначимо відношення порівняння для них таким чином:

```
(a,b) > (c,d) == true, якщо a > c або (a == c та b > d)
```

Тепер можна переписати умову `turn1 > turn2` процесу `CS1` у вигляді `(turn1,1) > (turn2,2)`, а умову `turn2 > = turn1` у процесі `CS2` — `(turn2,2) > (turn1,1)`. Перевага симетричного запису в тому, що тепер алгоритм поліклініки для двох процесів легко узагальнити на випадок `n` процесів.

Кожен з процесів спочатку показує, що він збирається увійти в критичну секцію, присвоюючи своїй змінній `turn` значення 1. Потім він знаходить максимальне значення з усіх `turn[i]` і додає до нього 1. Нарешті, процес запускає цикл `for` і, як в крупномодульному розв'язку, очікує своєї черги. Відзначимо, що максимальне значення масиву визначається зчитуванням всіх його елементів і вибором найбільшого. Ці дії не є неподільними, тому точний результат не гарантується. Однак, якщо кілька процесів отримують одне й те саме значення, вони упорядковуються відповідно до правила, описаного вище.

```
# Алгоритм полікліники — дрібномодульний розв'язок
int turn[l:n];
process CS[i = 1 to n] {
    while (true) {
        turn[i] = max(turn[l:n]) + 1;
        for [j = 1 to n && j != i]
            while(turn[j] == 0 && (turn[i],i) > (turn[j],j)
            skip
        критична секція;
        turn[i] = 0;
        некритична секція;
    }
}
```

6.5. Бар'єрна синхронізація

Основною властивістю більшості паралельних ітераційних алгоритмів є залежність результатів кожної ітерації від результатів попередньої. Один із способів побудувати такий алгоритм — реалізувати тіло кожної ітерації, використовуючи оператори `co`. Якщо вважати, що на кожній ітерації виконується `n` задач, отримаємо такий загальний вигляд алгоритму:

```
while (true) {
    co [i = 1 to n]
        код розв'язку задачі i;
    oc
}
```

На жаль, цей підхід досить неефективний, оскільки оператор `co` породжує `n` процесів на кожній ітерації. Створювати і знищувати процеси набагато дорожче, ніж реалізувати їх синхронізацію. Тому альтернативна структура алгоритму робить його набагато ефективнішим — процеси створюються один раз на початку обчислень, а потім синхронізуються в кінці кожної ітерації.

```
process Worker[i = 1 to n] {
    while (true) {
        код розв'язку задачі i;
        очікування завершення усіх задач;
    }
}
```

Точка затримки в кінці кожної ітерації є *бар'єром*, якого для продовження роботи повинні досягти всі процеси, тому цей механізм називається *бар'єрною синхронізацією*. Бар'єри можуть знадобитися в кінці циклів або на проміжних стадіях. Нижче розглянуто кілька реалізацій бар'єрної синхронізації, що використовують різні способи взаємодії процесів.

6.5.1. Спільний лічильник

Найпростіший спосіб описати вимоги до бар'єра — використовувати лічильник count з нульовим початковим значенням. Припустимо, що є n робочих процесів, які повинні зібратися біля бар'єру. Коли процес доходить до бар'єру, він збільшує значення count. Коли значення count стане рівним n, всі процеси зможуть продовжити роботу:

```
process Worker[i = 1 to n] {
    while (true) {
        код розв'язку задачі i;
        ⟨count = count + 1;⟩
        ⟨await (count == n);⟩
    }
}
```

Проте цей код не повною мірою відповідає поставленому завданню. Складність полягає в тому, що значенням count повинен бути 0 на початку кожної ітерації, тобто count потрібно обнуляти кожного разу, коли всі процеси пройдуть бар'єр. Більш того, вона повинна мати значення 0 перед тим, як будь-який з процесів знову спробує її збільшити. Цю проблему можна вирішити за допомогою двох лічильників, один з яких збільшується до n, а інший зменшується до 0. Їх ролі міняються місцями після кожної стадії. Однак використання спільних лічильників призводить до чисто практичних труднощів. По-перше, збільшувати і зменшувати їх значення потрібно неподільним чином. По-друге, коли процес призупиняється, він безперервно перевіряє значення змінної count. У гіршому випадку n-1 процесів чекатимуть, поки останній процес досягне бар'єра. В результаті виникне серйозний конфлікт звернення до пам'яті, якщо тільки програма не виконується на мультипроцесорній машині з узгодженою кеш-пам'яттю. Крім того, число n повинно бути відносно малим.

6.5.2. Керуючі процеси

Один із способів уникнути конфліктів звернення до пам'яті — реалізувати лічильник count за допомогою n змінних, значення яких додаються до одного і того ж значення. Нехай є масив цілих чисел arrive[1: n] з нульовими початковими значеннями. Замінимо операцію збільшення лічильника count в програмі так: arrive[i] = 1. Якщо елементи масиву arrive зберігаються в різних рядках кеш-пам'яті, то конфліктів звернення до пам'яті не буде. Залишилося реалізувати оператор await і обнулити елементи масиву arrive в кінці кожної ітерації. Оператор await можна записати в такому вигляді.

```
⟨await ((arrive[1] + ... + arrive[n]) == n);⟩
```

Але в такому випадку знову виникають конфлікти звернення до пам'яті, причому це рішення також неефективне, оскільки суму елементів `arrive[i]` тепер постійно обчислює кожний процес `Worker`, який очікує продовження.

Обидві проблеми можна вирішити, використовуючи додатковий набір спільних значень і ще один процес, `Coordinator`. Нехай кожен процес `Worker` замість того, щоб додати усі елементи масиву `arrive`, чекає, поки не стане істинним логічне значення. Нехай `continue[1: n]` — додатковий масив цілих з нульовими початковими значеннями. Після того як `Worker[i]` присвоїть 1 елементу `arrive[i]`, він повинен чекати, поки значенням змінної `continue[i]` не стане 1.

```
⟨arrive[i] = 1; (await (continue[i] == 1);⟩
```

Процес `Coordinator` очікує, поки всі елементи масиву `arrive` не стануть рівні 1, потім присвоює значення 1 всім елементам масиву `continue`.

```
for [i = 1 to n] ⟨await (arrive[i] == 1);⟩
for [i = 1 to n] continue[i] = 1;
```

Оскільки для продовження процесів `Worker` повинні бути встановлені всі елементи `arrive`, процес `Coordinator` може перевіряти їх в будь-якому порядку. Конфліктів звернення до пам'яті тепер не буде, оскільки процеси очікують зміни різних змінних, кожна з яких може зберігатися в своєму рядку кеш-пам'яті.

Змінні `arrive` та `continue` є прикладами так званого «*прапорця*». Його встановлює один процес, щоб повідомити інші про виконання умови синхронізації. Використовуються два основних правила *синхронізації за допомогою прапорців*:

а) прапорець синхронізації скидається тільки процесом, що очікує його установки;
б) прапорець не можна встановлювати до тих пір, поки невідомо точно, що він скинутий.

Перше правило гарантує, що прапорець не буде скинутий, поки процес не визначить, що він встановлений. Відповідно до цього правила прапорець `continue[i]` має скидатися процесом `Worker[i]`, а обнуляти всі елементи масиву `arrive` має `Coordinator`. Згідно з другим правилом один процес не встановлює прапорець, поки він не скинутий іншим. В іншому випадку, якщо інший синхронізований процес надалі очікує повторного встановлення прапорця, можливе взаємне блокування. Це означає, що `Coordinator` повинен скинути `arrive[i]` перед установкою `continue[i]`. `Coordinator` може також скинути `arrive[i]` відразу після того, як дочекався його установки. Додавши код скидання прапорів, отримаємо бар'єр з керуючим процесом:

```
int arrive[l:n], continue[l:n];
process Worker[i = 1 to n] {
    while (true) {
        код розв'язку задачі i;
        arrive[i] = 1;
        ⟨await (continue[i] == 1);⟩
        continue[i] = 0;
    }
}
process Coordinator {
    while (true) {
        for [i = 1 to n] {
            ⟨await (arrive[i] == 1);⟩
            arrive[i] = 0;
        }
        for [i = 1 to n]
            continue[i] = 1;
    }
}
```

Хоча в програмі бар'єрна синхронізація реалізована так, що конфлікти звернення до пам'яті виключаються, у даного розв'язку є дві небажані властивості. По перше, потрібен додатковий процес. Синхронізація з активним очікуванням ефективна, якщо кожний процес виконується на окремому процесорі, так що процесу `Coordinator` потрібен свій власний процесор. Але, можливо, було б краще використовувати цей процесор для іншого робочого процесу. Другий недолік використання керуючого процесу полягає в тому, що час виконання кожної ітерації процесу `Coordinator`, і, отже, кожного екземпляра бар'єрної синхронізації пропорційно числу процесів `Worker`. В ітераційних алгоритмах часто всі робочі процеси мають ідентичний код. Це означає, що якщо кожний робочий процес виконується на окремому процесорі, то всі вони підійдуть до бар'єра приблизно в один час. Таким чином, всі прапори `arrive` будуть встановлені практично одночасно. Проте процес `Coordinator` перевіряє прапорці в циклі, по черзі чекаючи, коли кожен з них буде встановлено.

Обидві проблеми можна подолати, об'єднавши дії керуючого і робочих процесів так, щоб кожний робочий процес був одночасно і керуючим. Організуємо робочі процеси в дерево (рис. 6.1).

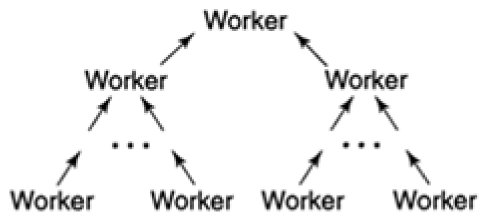

Рис. 6.1. Деревоподібний бар'єр

Сигнал про те, що процес підійшов до бар'єра (прапор `arrive[i]`), надсилається вгору по дереву, а сигнал про дозвіл продовження виконання (прапор `continue[i]`) — вниз. Вузол робочого процесу чекає, коли до бар'єра підійдуть його «діти», після чого повідомляє батьківський вузол про те, що він теж підійшов до бар'єра. Коли всі «діти» кореневого вузла підійшли до бар'єра, це означає, що все інші робочі вузли теж підійшли до бар'єра. Тоді кореневої вузол може повідомити нащадкам, що вони можуть продовжити виконання. Ті, в свою чергу, дозволяють продовжити виконання своїх синів, і так далі. Таким чином, отримуємо наступний розв'язок:

```
# Бар'єрна синхронізація за допомогою об'єднуючого дерева
лист L:
    arrive [L] = 1;
    ⟨await (continue[L] == 1);⟩
    continue[L] = 0;
проміжний вузол I:
    ⟨await (arrive[left] == 1);⟩
    arrive[left] = 0;
    ⟨await (arrive[right] == 1)⟩
    arrive[right] = 0;
    arrive[I] = 1;
    ⟨await (continue[I] == 1);⟩
    continue[I] = 0;
    continue[left] = 1; continue[right] = 1;
кореневий вузол R:
    ⟨await (arrive[left] == 1)⟩
    arrive[left] = 0;
    ⟨await (arrive[right] == 1);⟩
    arrive[right] = 0;
    continue[left] = 1; continue[right] = 1;
```

Отримана реалізація називається *бар'єром з об'єднуючим деревом*, оскільки кожен процес об'єднує результати роботи своїх дочірніх процесів і відправляє батьківському. Цей бар'єр використовує тільки ж змінних, скільки і «централізована» версія з керуючим процесом, але він набагато ефективніше при великих `n`, оскільки висота дерева пропорційна $\log_2 n$. Оператори `await` в цьому випадку можна реалізувати у вигляді циклів активного очікування.

6.6. Алгоритми, паралельні за даними

6.6.1. Паралельні префіксні обчислення

Часто буває потрібно застосувати деяку операцію до всіх елементів масиву. Наприклад, щоб обчислити середнє значення числового масиву `a[n]`, потрібно спочатку додати всі елементи масиву, а потім поділити суму на `n`. Іноді потрібно отримати середні значення для всіх префіксів `a[0:i]` масиву. Паралельні префіксні обчислення використовуються в багатьох задачах, включно з обробкою зображень, матричними обчисленнями і аналізом регулярних мов.

Розглянемо, як паралельно обчислюються суми всіх префіксів масиву. Ця операція називається паралельним префіксним обчисленням. Базовий алгоритм може бути використаний з будь-яким асоціативним бінарним оператором (додавання, множення, логічні оператори, обчислення максимуму та інші). Нехай дано масив `a[n]` і потрібно обчислити `sum[n]`, де `sum[i]` означає суму перших `i` елементів масиву `a`. Очевидний послідовний розв'язок:

```
sum[0] = a[0];
for [i = 1 to n-1]
    sum[i] = sum[i-1] + a[i];
```

Тепер подивимося, як цей алгоритм можна розпаралелити. Спочатку присвоїмо всім елементам `sum[i]` значення `a[i]`. Потім паралельно додамо `sum[i-1]` та `sum[i]` для всіх, тобто додамо всі елементи, які знаходяться на відстані 1. Тепер подвоїмо відстань і додамо елементи `sum[i-2]` з `sum[i]`. Якщо продовжувати подвоювати відстань, то після $\lceil \log_2 n \rceil$ кроків будуть обчислені всі часткові суми.

Наступна схема ілюструє кроки алгоритму для масиву з шести елементів.

Початкові значення елементів `a`	1	2	3	4	5	6
Значення `sum` на відстані 1	1	3	5	7	9	11
Значення `sum` на відстані 2	1	3	6	10	14	18
Значення `sum` на відстані 4	1	3	6	10	15	21

Нижче наведена реалізація цього алгоритму. Кожен процес спочатку ініціалізує один елемент масиву `sum`, а потім циклічно обчислює часткові суми. Процедура `barrier(i)`, що викликається в програмі, реалізує точку бар'єрної синхронізації, аргумент `i` — ідентифікатор процедури процесу. Вихід з процедури відбувається, коли усі `n` процесів виконають команду `barrier`. У тілі процедури може бути використаний один з алгоритмів, описаних в попередньому підрозділі.

```
# Обчислення часткових сум елементів масиву

process Sum[i = 0 to n-1] {
    int d = 1;
    sum[i] = a[i]; # ініціалізація елементів sum
    barrier(i);
    while (d < n) {
        old[i] = sum[i]; # зберегти попереднє значення
        barrier(i);
        if (i-d >= 0)
            sum[i] = old[i-d] + sum[i];
        barrier(i);
        d = d+d; # подвоїти відстань
    }
}
```

Точки бар'єрів потрібні для усунення взаємного впливу процесів. Наприклад, всі елементи масиву `sum` повинні бути проініціалізовані до того, як який-небудь процес звернеться до них. Цей алгоритм можна використовувати з будь-яким асоціативним бінарним оператором. Програму можна адаптувати для числа процесів меншого за `n`; тоді кожен процес буде відповідати за об'єднання часткових сум смуги масиву.

6.6.2. Операції зі зв'язаними списками

При роботі з структурами даних типу дерев для пошуку і вставки елементів за логарифмічний час часто використовуються збалансовані бінарні дерева. Проте при використанні алгоритмів, паралельних за даними, багато операцій навіть з лінійними списками можна реалізувати за логарифмічний час. Покажемо, як знайти кінець зв'язаного списку. Цей же алгоритм можна використовувати і для інших операцій над зв'язаними списками, наприклад, вставки елемента в список пріоритетів або поелементного порівняння двох списків.

Припустимо, що є зв'язаний список (див. рис. 6.2), що містить не більше `n` елементів. Зв'язки зберігаються в масиві `link[n]`, а дані — в масиві `data[n]`. На початок списку вказує змінна `head`. Якщо елемент `i` є частиною списку, то `head == i`, або `link[j] == i` для деякого `j` від `0` до `n - 1`. Поле `link` останнього елемента списку рівне `null`. Припустимо, що список вже ініціалізований.

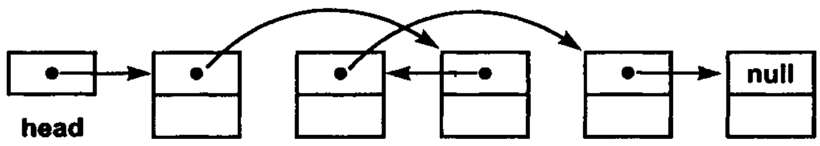

Рис. 6.2. Будова списку

Задача полягає в тому, щоб знайти кінець списку. Звичайний послідовний алгоритм починає роботу з початку списку head і рухається за посиланнями, поки не знайде порожній покажчик. Час роботи цього алгоритму пропорційний довжині списку. Проте пошук кінця списку можна виконати за логарифмічний час, якщо використовувати алгоритм, паралельний за даними, і метод здвоєння з попереднього пункту.

Кожному елементу списку призначається процес Find. Нехай end[n] спільний масив цілих чисел. Якщо елемент i є частиною списку, то завдання процесу Find[i] — присвоїти змінній end[i] значення, рівне індексу останнього елемента списку, інакше процес Find[i] повинен присвоїти end[i] значення null. Припустимо, що список містить хоча б два елементи. На початку роботи кожен процес записує у end[i] значення link[i], тобто індекс наступного елемента списку (якщо він є). Потім процеси виконують ряд етапів. На кожному етапі процес розглядає елемент з індексом end[end[i]]. Якщо елементи end[i] та end[end[i]] — не порожні вказівники, то процес присвоює елементу end[i] значення end[end[i]]. Далі процес повторюються. Таким чином, після *d*-го циклу змінна end[i] буде вказувати на елемент списку, що знаходиться на відстані в 2^{d-1} від *i*-го (якщо такий є). Після $\lceil \log_2 n \rceil$ циклів кожен процес знайде кінець списку.

```
# Пошук кінця послідовного звязаннго списку
process Find[i = 0 to n-1] {
    int new, d = 1;
    end[i] = link[i]; # ініціалізація елементів
    barrier(i);
    while (d < n) {
        new = null; # переврка, чи треба обновити end[i]
        if (end[i] != null && end[end[i]] != null)
            new = end[end[i]];
        barrier(i);
        if (new != null)
            end[i] = new;
        barrier(i);
        d = d << 1;
    }
}
```

Для ілюстрації роботи алгоритму у випадку списку із 6 елементів розглянемо рис. 6.3. Після третьої ітерації кожний елемент вказує на кінець списку.

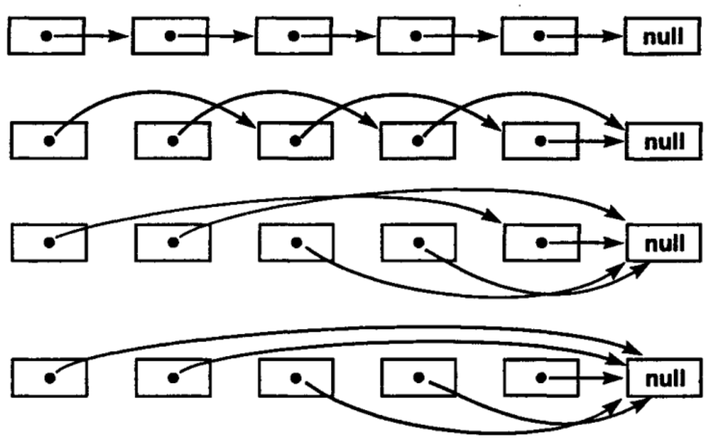

Рис. 6.3. Паралельний пошук кінця списку

6.7. Паралельні обчислення з портфелем задач

Розглянемо ще один спосіб реалізації паралельних обчислень, в якому використовується так званий *портфель задач*. Задача є незалежною одиницею роботи. Задачі поміщаються в портфель, спільний для кількох робочих процесів. Кожен робочий процес виконує наступний основний код.

```
while (true) {
    отримати задачу з портфеля;
    if (задач більше нема)
        break;
    виконати задачу (можливо, породжуючи нові задачі);
}
```

Цей підхід можна використовувати для реалізації рекурсивного паралелізму; тоді задачі будуть являти собою рекурсивні виклики. Його також можна використовувати для ітеративних задач з фіксованим числом незалежних завдань.

Парадигма портфеля задач має кілька корисних властивостей. По-перше, вона дуже проста у використанні. Досить визначити зображення задачі, реалізувати портфель, запрограмувати виконання задачі і з'ясувати, як розпізнається завершення роботи алгоритму. По друге, програми, що використовують портфель задач, є масштабованими. Їх можна використовувати з будь-яким числом процесорів; для цього достатньо просто змінити кількість робочих процесів. І, нарешті, ця парадигма спрощує реалізацію балансування навантаження. Якщо тривалості виконання задач різні, то, ймовірно, деякі із задачі будуть виконуватися довше інших. Але поки задач більше, ніж робочих процесів (в дватри рази), загальні обсяги обчислень, здійснювані робочими процесами, будуть приблизно однаковими.

Покажемо, як за допомогою портфеля задач реалізується множення матриць. При множенні матриць використовується фіксоване число задач. Для захисту доступу до портфелю застосовуються критичні секції, а для виявлення закінчення — бар'єроподібна синхронізація.

Припустимо, що програма буде виконуватися на машині з числом процесорів `PR`. Тоді бажано використовувати `PR` робочих процесів, по одному на кожний процесор. Щоб збалансувати обчислювальну завантаження, процеси повинні обчислювати приблизно порівну проміжних добутків. Кожний робочий процес буде захоплювати задачі при необхідності. Якщо число PR набагато менше, ніж `n`, то відповідний для задачі обсяг роботи — один або кілька рядків результуючої матриці `c`. Для простоти використовуємо окремі рядки. У початковому стані портфель містить `n` задач, по одному на рядок. Робочий процес отримує задачу з портфеля, виконуючи неподільну дію

```
⟨row = nextRow; nextRow++;⟩
```

Тут `row` — локальна змінна. Портфель порожній, коли значення `row` не менше за `n`. У програмі передбачається, що матриці ініціалізовані. Програма завершується, коли всі робочі процеси вийдуть з циклу `while`.

```
# Множення матриць за допомогою портфеля задач

int nextRow = 0; # портфель задач
process Worker[w = 1 to RP] {
    int row;
    double sum; # для проміжних добутків
    while (true) {
        # отримати задачу
        ⟨ row = nextRow; nextRow++; ⟩
        if (row >= n)
            break;
        обчислити скалярні добутки для c[row, * ];
    }
}
```

Для визначення моменту завершення можна скористатися лічильником `done` з нульовим початковим значенням. Перед тим як робочий процес виконає оператор `break`, він повинен збільшити значення лічильника в неподільній дії. Якщо потрібно, щоб останній `Worker` вивів результат, то можна додати наступний код до кожного процесу

```
if (done == n)
    print matrix c;
```

Змінна `done` використовується в ролі бар'єра-лічильника.

7. СЕМАФОРИ

7.1. Синтаксис та семантика

Семафор — це особливий тип спільної змінної, яка обробляється тільки двома *неподільними* операціями `P` і `V`. Семафор можна вважати екземпляром класу семафор, його операції — методами цього класу з додатковим атрибутом неподільності.

Значення семафора є *невід'ємним цілим числом*. Операція `V` використовується для сигналізації про те, що подія відбулася, тому вона збільшує значення семафора. Операція `P` призупиняє процес до моменту, коли відбудеться деяка подія, тому вона, дочекавшись, коли значення семафора стане додатним, зменшує його. Сила семафорів обумовлена тим, що виконання операції `P` може бути призупинено.

Семафор оголошується так:

```
sem s;
```

За замовчуванням початковим значенням є 0, але семафор можна ініціалізувати будь-яким додатним значенням, наприклад:

```
sem lock = 1;
```

Масиви семафорів можна оголошувати і при необхідності ініціалізувати таким чином:

```
sem forks[5] = ([5] 1);
```

Якби в цій декларації не було ініціалізації, то початковим значенням кожного семафора в масиві `forks` був би 0.

Після оголошення та ініціалізації семафор можна обробляти тільки за допомогою операцій `P` і `V`. Кожна з них є неподільним дією з одним аргументом. Нехай `s` — семафор. Тоді операції `P(s)` та `V(s)` визначаються наступним чином.

```
P(s): ⟨await (s > 0) s = s - 1;⟩
V(s): ⟨s = s + 1;⟩
```

Припустимо, що `s` — семафор з поточним значенням 1. Якщо два процеси намагаються одночасно виконати операцію `P(s)`, то це вдасться зробити тільки одному з них. Але якщо один процес намагається виконати операцію `P(s)`, а інший — `V(s)`, то обидві операції будуть успішно виконані в непередбачуваному порядку, а кінцевим значенням семафора `s` стане 1.

Звичайний семафор може приймати будь-які невід'ємні значення, *двійковий семафор* — тільки значення 1 або 0. Це означає, що операція `V` для двійкового семафора може бути виконана, тільки коли його значення 0. Властивості справедливості для операцій з семафором випливають з їх визначення з допомогою оператора `await`. Якщо умова `s > 0` стає і надалі залишається істинною, виконання операції `P(s)` завершиться при справедливій в слабкому сенсі стратегії планування. Якщо умова `s > 0` стає істинною нескінченно часто, то виконання операції `P(s)` завершиться при справедливій в сильному сенсі стратегії планування. Операція `V` для звичайного семафора є безумовною неподільною дією, тому вона завершиться, якщо стратегія планування безумовно справедлива.

7.2. Основні задачі та методи

Семафори безпосередньо підтримують реалізацію взаємного виключення. Крім того, вони забезпечують підтримку простих форм умовної синхронізації, де вони використовуються для сигналізації про події. Для розв'язування складніших задач ці два способи застосування семафорів можна комбінувати.

7.2.1. Критичні секції: взаємне виключення

Семафори були придумані в томі числі і для розв'язування задачі критичної секції. Саме ці операції і підтримуються семафором. Нехай `mutex` — семафор з початковим значенням 1. Виконання операції `P(mutex)` — це те саме, що і очікування, поки значення змінної `lock` (яка використовувалася у розв'язку задачі критичної секції у попередньому розділі) не стане рівним 1, і подальше присвоєння їй значення 0. Аналогічно виконання операції `V(mutex)` — це те саме, що присвоєння `lock` значення 1 (за умови, що це можна зробити, тільки коли вона має значення 0).

```
# Семафорний розв'язок задачі критичної секції
sem mutex = 1;
process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        критична секція;
        V(mutex);
        некритична секція;
    }
}
```

7.2.2. Бар'єри: сигналізація подій

Семафори полегшують реалізацію бар'єрної синхронізації. Основна ідея — використовувати семафор в якості прапора синхронізації. Виконуючи операцію `V`, процес встановлює прапор, а при операції `P` — скидає його.

Спочатку розглянемо задачу реалізації бар'єру для двох процесів. Нагадаємо, що необхідно виконати дві вимоги. По-перше, жоден процес не повинен перейти за бар'єр, поки до нього не підійшли обидва процеси. По-друге, бар'єр має допускати багаторазове використання, оскільки зазвичай одні й ті ж процеси синхронізуються після кожного етапу обчислень. Для задачі критичної секції досить лише одного семафора для блокування, оскільки потрібно просто визначити, чи знаходиться процес у критичній секції. Але при бар'єрній синхронізації необхідні два семафора в якості сигналів, щоб знати, приходить процес до бар'єра або йде від нього.

Сигнальний семафор `s` — це семафор з нульовим (як правило) початковим значенням. Процес сигналізує про подію, виконуючи операцію `V(s)`; інші процеси очікують події, виконуючи `P(s)`. Для бар'єра у випадку двох процесів дві істотних події полягають у тому, що процеси прибувають до бар'єра. Отже, задачу можна вирішити за допомогою двох семафорів `arrive1` і `arrive2`. Кожен процес повідомляє про своє прибуття до бар'єра, виконуючи операцію `V` для свого семафора, і потім очікує прибуття іншого процесу, виконуючи для його семафора операцію `P`.

```
# Бар'єрна синхронізація за допомогою семафорів
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1); # Сигнал про прибуття
    P(arrive2); # Очікування іншого процесу
    ...
}
process Worker2 {
    ...
    V(arrive2); # Сигнал про прибуття
    P(arrive1); # Очікування іншого процесу
    ...
}
```

За допомогою аналогічного прийому можна реалізувати бар'єр для `n` процесів. Для цього знадобиться масив семафорів `arrive`. На кожному етапі `i`-й процес спочатку повідомляє про своє прибуття, виконуючи операцію `V(arrive[i])`, а потім очікує прибуття інших процесів, виконуючи `P` для їх елементів масиву `arrive`. На відміну від ситуації з змінними-прапорцями тут потрібний тільки один масив семафорів `arrive`, оскільки дія операції `V` «запам'ятовується», тоді як значення прапорця змінної може бути перезаписане. Семафори можна використовувати і в якості сигнальних прапорів в реалізації бар'єрної синхронізації для n процесів з керуючим процесом або деревом (див. попередній розділ). Операції `V` запам'ятовуються, тому використовується менше семафорів, ніж прапорців змінних. У керуючому процесі `Coordinator`, наприклад, потрібен всього один семафор.

7.2.3. Виробники і споживачі: розділені семафори

В даному пункті знову розглядається задача про виробників і споживачів. Раніше передбачалося, що є тільки один виробник і один споживач. Тут розглядається загальний випадок, коли є кілька виробників і кілька споживачів.

У задачі про виробників і споживачів виробники посилають повідомлення споживачам. Процеси спілкуються за допомогою буфера та операцій `deposit` та `fetch`. Виконуючи операцію `deposit`, виробники поміщають повідомлення в буфер; споживачі отримують повідомлення за допомогою операції `fetch`. Щоб повідомлення не перезаписувалися і кожне з них отрималося тільки один раз, виконання операцій `deposit` і `fetch` має чергуватися, причому першою повинна бути `deposit`.

Запрограмувати необхідне чергування операцій можна за допомогою семафорів. Нехай `empty` (порожній) і `full` (повний) — два семафора, що відображають стан буфера. У початковому стані буфер порожній, семафор `empty` має значення 1 (тобто відбулася подія «спорожнити буфер»), а `full` — 0. Перед виконанням операції `deposit` виробник спочатку очікує спорожнення буфера. Коли виробник поміщає в буфер повідомлення, буфер стає заповненим. І, навпаки, перед виконанням операції `fetch` споживач спочатку очікує заповнення буфера, а потім спорожнює його. Процес очікує події, виконуючи операцію `P` для відповідного семафора, і повідомляє про подію, виконуючи `V`. Змінні `empty` і `full` є двійковими семафора. Разом вони утворюють так званий розділений двійковий семафор, оскільки в будь-який момент часу тільки один з них може мати значення 1. Термін «розділений двійковий семафор» пояснюється тим, що змінні `empty` і `full` можуть розглядатися як єдиний двійковий семафор, поділений на дві частини. У загальному випадку розділений двійковий семафор може бути утворений будь-яким числом двійкових семафорів.

```
# Виробники і споживачі, які використовують семафори
typeT buf; # Буфер деякого типу T
sem empty = 1, full = 0;
process Producer [i = 1 to M] {
    while (true) {
        # Помістити дані в буфер
        P(empty);
        buf = data;
        V(full);
    }
}
```

```
process Consumer[j = 1 to N] {
    while (true) {
        # Одержати дані
        P(full);
        result = buf;
        V(empty);
    }
}
```

Кожен процес `Producer` позмінно виконує операції `P(empty)` та `V(full)`, а кожний процес-споживач `Consumer` — `P(full)` та `V(empty)`.

7.2.4. Кільцеві буфери: облік ресурсів

З останнього прикладу видно, як синхронізувати доступ до одного буферу обміну. Якщо дані виробництво і споживання яких приблизно з однаковою частотою, то процесу не доводиться довго чекати доступу до буферу. Однак зазвичай споживач і виробник працюють нерівномірно. Наприклад, виробник може швидко створити відразу кілька елементів, а потім довго обчислювати наступну серію елементів. У таких випадках збільшення ємності буфера може істотно підвищити продуктивність програми, зменшуючи число блокувань процесів.

Припустимо поки, що є тільки один виробник і тільки один споживач. Виробник поміщає повідомлення в спільний буфер, споживач отримує їх звідти. Буфер містить чергу уже розміщених, але ще не прочитаних повідомлень. Ця чергу може бути зображена зв'язаним списком або масивом. Реалізуємо буфер масивом `buf[n]`, де `n > 1`. Нехай змінна `front` є індексом першого повідомлення черзі, а `rear` — індексом першої порожньої комірки після повідомлення в кінці черги. Спочатку змінні `front` і `rear` мають однакові значення, скажімо, 0. Виробник поміщає в буфер повідомлення з значенням `data`, виконавши такі дії:

```
buf[rear] = data;
rear = (rear + 1)% n;
```

Аналогічно споживач отримує повідомлення в свою локальну змінну `result`, виконуючи дії:

```
result = buf[front];
front = (front + 1)% n;
```

Черга буферизованих повідомлень зберігається в комірках від `buf[front]` до `buf[rear]` (не включно). Змінна `buf` інтерпретується як кільцевий масив, в якому за `buf[n-1]` йде `buf[0]`.

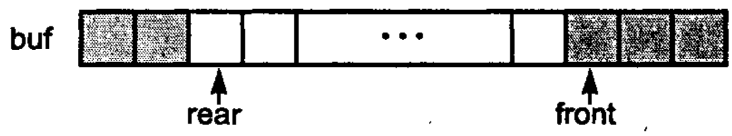

Рис. 7.1. Схема кільцевого буфера

Якщо використовується тільки один буфер (як в схемі «виробник-споживач»), то виконання операцій `depos` та `fetch` має чергуватися. При наявності кількох буферів операцію `deposit` можна виконати, якщо є порожня комірка, а `fetch` — якщо збережено хоча б одне повідомлення. Вимоги синхронізації для одноелементного і кільцевого буфера *однакові*. Єдина відмінність полягає в тому, що семафор `empty` ініціалізується значенням `n`, а не 1, оскільки в початковому стані є `n` порожніх комірок.

```
# Кільцевій буфер з використанням семафорів
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0; # n-2 <= empty + full <= n

process Producer {
    while (true) {
        створити повідомлення data;
        # Помістити data в буфер
        P(empty);
        buf[rear] = data;
        rear = (rear + 1)% n;
        V(full);
    }
}

process Consumer {
    while (true) {
        # Отримати повідомлення result
        P(full);
        result = buf[front];
        front = (front + 1)% n;
        V(empty);
    }
}
```

Семафори грають роль *лічильників ресурсів*: кожен враховує кількість елементів ресурсу: `empty` — число порожніх комірок буфера, а `full` — заповнених. Коли жодний процес не виконує `deposit` або `fetch`, сума значень обох семафорів дорівнює загальному числу комірок `n`.

У попередній програмі передбачалося, що є тільки один виробник і один споживач. Це гарантувало неподільне виконання операцій `deposit` та `fetch`. Припустимо, що є кілька процесів-виробників. При наявності хоча б двох вільних комірок два з них могли б виконати операцію `deposit` одночасно. Але тоді обидва спробували б помістити свої повідомлення в одну і ту ж комірку! Аналогічно, якщо є кілька споживачів, два з них одночасно можуть виконати `fetch` і отримати одне і те ж повідомлення. Таким чином, `deposit` і `fetch` стають критичними секціями. Однакові операції повинні виконуватися з взаємним виключенням, але різні можуть виконуватися одночасно, оскільки при роботі семафорів `empty` та `full` виробники і споживачі звертаються до різних комірок буфера.

```
# Кілька виробників і споживачів, що використовують семафори
typeT buf[n];
int front = 0, rear = 0;
sem empty = n, full = 0;   # n-2 <= empty + full <= n
sem mutexD = 1, mutexF = 1; # для взаємного виключення

process Producer [i = 1 to M] {
    while (true) {
        створити повідомлення data;
        # Помістити data в буфер
        P(empty);
        P(mutexD);
        buf[rear] = data;
        rear = (rear + 1)% n;
        V(mutexD);
        V(full);
    }
}

process Consumer [j = 1 to N] {
    while (true) {
        # Прочитати повідомлення result
        P(full);
        P(mutexF);
        result = buf[front];
        front = (front + 1)% n;
        V(mutexF);
        V(empty);
    }
}
```

7.3. Задача про обід філософів

У попередньому підрозділі було показано, як використовувати семафори для вирішення задачі критичної секції. На основі цього розв'язку будується реалізація вибіркового взаємного виключення для двох класичних задач: про філософів, що обідають і про читачів та письменників. Розв'язок задачі про філософів ілюструє реалізацію взаємного виключення між процесами, що конкурують за доступ до множин спільних змінних, задача про читачів та письменників — реалізацію комбінації паралельного та виняткового доступу до спільних змінним.

Хоча задача про філософів скоріше цікава, ніж практична, вона аналогічна до реальних проблем, в яких процесу необхідний одночасний доступ до кількох ресурсів. Тому її часто використовують для ілюстрації та порівняння різних механізмів синхронізації. Наведемо формулювання задачі про філософів, яке належить Е. Дейкстрі [6]. П'ять філософів сидять біля круглого столу. Вони проводять життя, чергуючи прийоми їжі та роздуми. У центрі столу знаходиться велике блюдо спагеті. У процесі їжі філософи повинні користуватися двома виделками. На жаль, їм дали всього п'ять виделок. Між кожною парою філософів лежить одна виделка і вони домовилися, що кожен буде користуватися тільки тими виделками, які лежать поруч з ним (зліва і справа). Потрібно написати програму, що моделює поведінку філософів. Програма повинна уникати ситуації, в якій всі філософи голодні, але жоден з них не може взяти обидві виделки — наприклад, коли кожен з них тримає по одній вилці і не хоче віддавати її.

Задача проілюстрована на рис. 7.2. Ясно, що два філософа, які сидять поруч, не можуть їсти одночасно. Крім того, оскільки виделок всього п'ять, одночасно можуть їсти не більше, ніж двоє філософів. Припустимо, що періоди роздумів і прийомів їжі різні — для їх імітації в програмі можна використовувати генератор випадкових чисел. Змоделюємо поведінку філософів наступним чином.

```
process Philosopher [i = 0 to 4] {
    while (true) {
        поміркувати;
        взяти виделки;
        поїсти;
        віддати виделки;
    }
}
```

Для розв'язання задачі потрібно запрограмувати операції «взяти виделки» та «звільнити виделки». Виделки є ресурсом. Зосередимося на їх отриманні та звільненні.

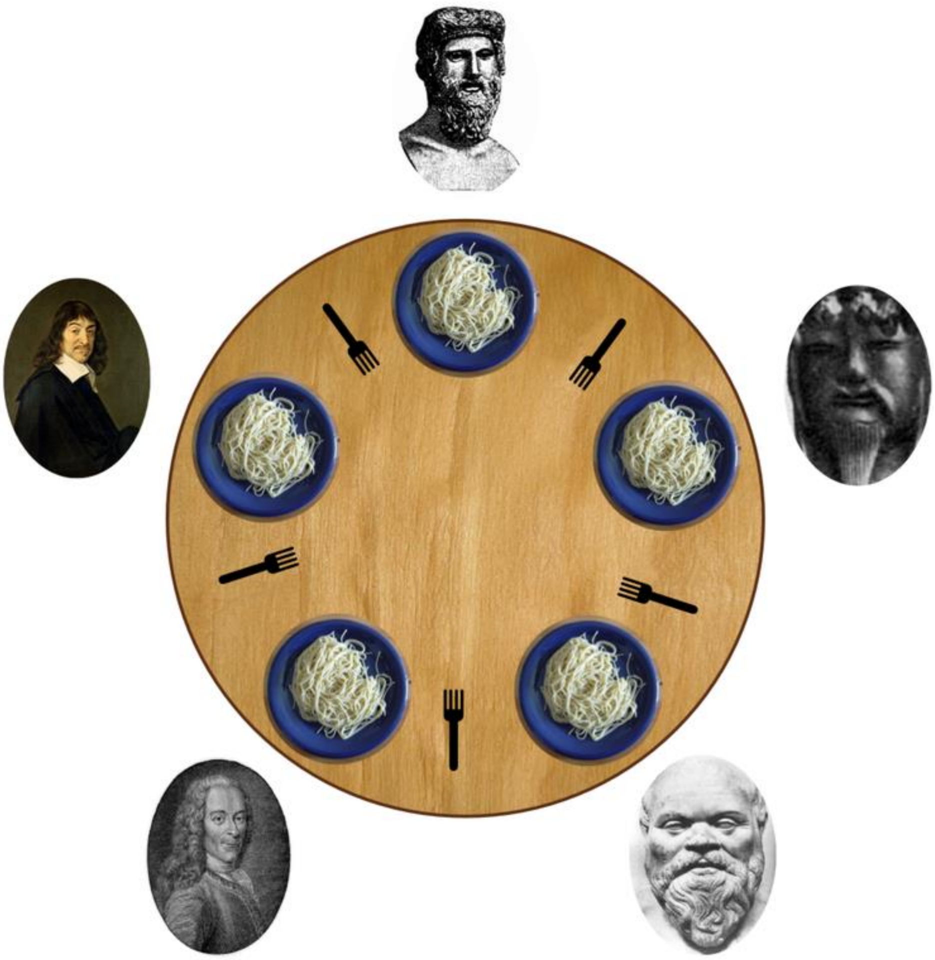

Рис. 7.2. Ілюстрація до задачі про обід філософів

Кожна виделка схожа на блокування критичної секції: в будь-який момент часу володіти нею може тільки один філософ. Отже, виделки можна уявити масивом семафорів, ініціалізованих значенням 1. Взяття виделки імітується операцією `P` для відповідного семафора, а звільнення — операцією `V`. Процеси, по суті, ідентичні, тому природно припускати, що вони виконують однакові дії. Наприклад, кожен процес може спочатку взяти ліву виделку, потім праву. Однак це може привести до взаємного блокування процесів. Наприклад, якщо все філософи візьмуть свої ліві виделки, то вони назавжди залишаться в очікуванні можливості взяти праву виделку.

Необхідна умова взаємного блокування — можливість кругового очікування, коли один процес чекає ресурс, зайнятий другим процесом, який чекає ресурс, зайнятий третім, і так далі до деякого процесу, що очікує ресурс, зайнятий першим процесом. Таким чином, щоб уникнути взаємного блокування, досить забезпечити неможливість виникнення кругового очікування. Для цього можна змусити один з процесів, скажімо, `Philosopher[4]`, спочатку взяти праву виделку. Можливий також варіант розв'язку, в якому філософи з парним номером беруть виделки в одному порядку, а з непарним — у іншому.

```
# Семафорний розв'язок задачі про обід філософів
sem fork [5] = {1,1,1,1,1};
process Philosopher[i = 0 to 3] {
    while (true) {
        P(fork[i]);    # Взяти ліву виделку
        P(fork[i+1]);  # потім праву
        поїсти;
        V(fork[i]); V(fork[i+1]);
        поміркувати;
    }
}
process Philosopher[4] {
    while (true) {
        P(fork[0]);    # Взяти праву виделку
        P(fork[4]);    # потім ліву
        поїсти;
        V(fork[0]); V(fork[4]);
        поміркувати;
    }
}
```

7.4. Задача про читачів та письменників

Задача про читачів та письменників — ще одна класична задача синхронізації. Її часто використовують для порівняння механізмів синхронізації. Вона також дуже важлива для практичного застосування. Формулювання: базу даних поділяють два типи процесів — читачі та письменники. Читачі виконують транзакції, які переглядають записи бази даних, а транзакції письменників і переглядають, і змінюють записи. Передбачається, що спочатку база даних знаходиться в несуперечливому стані. Кожна окрема транзакція переводить базу даних з одного несуперечливого стану в інший. Для запобігання взаємного впливу транзакцій процес-письменник повинен мати винятковий доступ до бази даних. Якщо до бази даних не звертається ні один з процесів-письменників, то виконувати транзакції можуть одночасно скільки завгодно читачів.

Задача про читачів і письменників — це приклад задачі вибіркового взаємного виключення. Класи процесів конкурують за доступ до бази даних. Процеси-читачі конкурують з письменниками, а окремі процеси-письменники — між собою. Це також приклад задачі загальної умовної синхронізації: процеси-читачі повинні чекати, поки до бази даних має доступ хоча б один процес-письменник; процеси-письменники повинні чекати, поки до бази даних мають доступ процеси-читачі або інший процес-письменник.

У цьому підрозділі дається два різних розв'язки задачі про читачів і письменників. У першому вона розглядається як задача взаємного виключення. Цей розв'язок є коротким, і його легко реалізувати. Однак в ньому читачі отримують перевагу перед письменниками і його важко модифікувати. У другому розв'язку задача розглядається як задача умовної синхронізації. Він здається більш складним, але насправді його теж легко реалізувати. Більш того, він легко змінюється для того, щоб реалізувати для читачів і письменників різні стратегії планування. В другому підході використовується потужний метод програмування, який називається передачею естафети та може застосовуватися для розв'язування довільної задачі умовної синхронізації.

7.4.1. Задача про читачів і письменників як задача виключення

Процесам-письменникам потрібен взаємовиключний доступ до бази даних. Доступ процесів-читачів як групи також повинен бути взаємовиключним по відношенню до будь-якого процесу-письменника. Корисний для будь-якої задачі вибіркового взаємного виключення підхід — спочатку ввести додаткові обмеження, реалізувавши більше винятків, ніж потрібно, а потім послабити обмеження. Очевидне додаткове обмеження — забезпечити винятковий доступ до бази даних кожному читачеві і письменнику. Нехай змінна `rw` — це семафор взаємного виключення з початковим значенням `1`. В результаті отримаємо наступний розв'язок з додатковим обмеженням:

```
# Задача про читачів і письменників з додатковим обмеженням
sem rw = 1;
process Reader [i = 1 to M] {
    while (true) {
        P(rw); # Захопити блокування виключного доступу
        читати базу даних;
        V(rw); # Звільнити блокування
    }
}
process Writer [i = 1 to N] {
    while (true) {
        P(rw); # Захопити блокування виключного доступу
        записати в базу даних;
        V(rw); # Звільнити блокування
    }
}
```

Розглянемо, як послабити обмеження в програмі, щоб процеси-читачів могли працювати паралельно. Читачі як група повинні блокувати роботу письменників, але тільки перший читач повинен захопити блокування взаємного виключення, виконавши операцію `P(rw)`. Решта читачі можуть відразу звертатися до бази даних. Читач, закінчуючи роботу, повинен знімати блокування, тільки якщо є останнім активним процесом-читачем. Отримуємо наступний розв'язок:

```
# Схема розв'язку задачі про читачів і письменників
int nr = 0;    # Кількість активних читачів
sem rw = 1;    # Блокування доступу до бази даних
process Reader [i = 1 to M] {
    while (true) {
        ⟨nr = nr + 1;
        if (nr == 1) P(rw);⟩ # Отримати блокування, якщо перший
        читати базу даних;
        ⟨nr = nr-1;
        if (nr == 0) V(rw);⟩ # Зняти блокування, якщо останній
    }
}
```

Змінна `nr` слугує для підрахунку числа активних читачів. Щоб уникнути взаємного впливу процесів-читачів, додавання та перевірка повинні виконуватися як критична секція, тому для забезпечення неподільного виконання протоколу входу використані кутові дужки. Процеси `Writer` описуються без змін.

Тепер реалізуємо неподільні дії за допомогою семафорів. Нехай `mutexR` — семафор, що забезпечує взаємне виключення процесів-читачів.

```
# Семафорний розв'язок задачі про читачів і письменників
int nr = 0;        # Число активних читачів
sem rw = 1;        # Блокування доступу до бази даних
sem mutexR = 1;    # Блокування доступу читачів до nr
process Reader [i = 1 to m] {
    while (true) {
        P(mutexR);
        nr = nr + 1;
        if (nr == 1) P(rw); # Отримати блокування, якщо перший
        V(mutexR);
        читати базу даних;
        P(mutexR);
        nr = nr-1;
        if (nr == 0) V(rw); # Зняти блокування, якщо останній
        V(mutexR);
    }
}

process Writer [i = 1 to n] {
    while (true) {
```

```
            P(rw);
            записати в базу даних;
            V(rw);
        }
    }
```

Алгоритм реалізує розв’язок задачі з перевагою читачів. Якщо деякий процес-читач звертається до бази даних, а інший читач і письменник досягають протоколів входу, то новий читач отримує перевагу перед письменником. Отже, це рішення не є справедливим, оскільки нескінченний потік процесів-читачів може постійно блокувати доступ письменників до бази даних.

7.4.2. Розв’язок задачі про читачів і письменників з використанням умовної синхронізації

Побудуємо інший розв’язок, почавши з більш простого визначення необхідної синхронізації. В цьому розв’язку буде використовуватися загальний метод програмування, який називається *передачею естафети* і використовує розділені двійкові семафори як для виключення, так і для сигналізації призупиненим процесам. Метод передачі естафети можна застосувати для реалізації будь-яких операторів типу `await` і, таким чином, для реалізації довільної умовної синхронізації.

Для збереження несуперечності (цілісності) бази даних письменникам необхідний винятковий доступ, але процеси-читачі можуть працювати паралельно в будь-якій кількості. Простий спосіб опису такої синхронізації полягає в підрахунку процесів кожного типу, які звертаються до бази даних, і обмеження значень лічильників. Наприклад, нехай `nr` і `nw` — змінні з невід’ємними цілими значеннями, що зберігають відповідно число процесів-читачів і процесів-письменників, які отримали доступ до бази даних. Схема основної частини процесу-читача виглядає так:

```
⟨nr = nr + 1;⟩
читати базу даних;
⟨nr = nr - 1;⟩
```

Відповідна схема процесу-письменника така:

```
⟨nw = nw + 1;⟩
записати в базу даних;
⟨nw = nw - 1;⟩
```

Щоб уточнити ці схеми, потрібно захистити операції присвоювання. У процесах-читачів для цього необхідно захистити збільшення `nr` умовою `(nw == 0)`, оскільки при збільшенні змінної `nr` значенням `nw` повинно бути рівне 0. У процесах-письменниках необхідно дотримуватися умови `(nr == 0`

`&& nw == 0)`. Однак в захисті операцій віднімання немає необхідності, оскільки *ніколи не потрібно затримувати процес, який звільняє ресурс*. Після врахування необхідних для захисту умов отримаємо крупномодульний розв'язок:

```
# Крупномодульний розв'язок задачі про читачів та письменників
int nr = 0, nw = 0;
process Reader [i = 1 to M] {
    while (true) {
        ⟨await (nw == 0) nr = nr+1;⟩
        читати базу даних;
        ⟨nr = nr-1;⟩
    }
}
process Writer [i = 1 to N] {
    while (true) {
        ⟨await (nr == 0 && nw == 0) nw = nw+1;⟩
        записати в базу даних;
        ⟨nw = nw-1;⟩
    }
}
```

7.4.3. Метод передачі естафети

Іноді оператори `await` можна реалізувати шляхом прямого використання семафорів або інших елементарних операцій, але в загальному випадку це неможливо. Розглянемо дві умови захисту операторів `await` у програмі з попереднього пункту. Ці умови перекриваються: умова захисту в протоколі входу письменника вимагає, щоб і `nw`, і `nr` дорівнювали 0, а в протоколі входу читача — щоб `nw` дорівнювала 0. Жоден семафор не може розрізнити ці умови, тому для реалізації таких операторів `await`, як зазначений тут, потрібен загальний метод. Викладений далі метод називається *передачею естафети*. Цей метод досить потужний, щоб реалізувати будь-який оператор `await`.

Для реалізації операторів `await` можна використовувати розділені двійкові семафори. Нехай `e` — двійковий семафор з початковим значенням 1. Він буде застосовуватися для управління входом в будь-яку неподільну дію.

З кожною умовою захисту пов'яжемо один семафор і один лічильник з нульовими початковими значеннями. Семафор будемо використовувати для припинення процесу до моменту, коли умова захисту стане істинною. У лічильнику буде зберігатися число припинених процесів. Оскільки є два різні умови захисту, по одному в протоколах входу письменників і читачів, то потрібні два семафора і два лічильника. Нехай `r` — семафор, пов'язаний з умовою захисту в процесі-читачі, а `dr` — відповідний йому лічильник призупинених процесів-читачів.

Аналогічно нехай з умовою захисту в процесі письменника пов'язані семафор `w` і лічильник призупинених процесів-письменників `dw`.
Для виходу з неподільних дій може використовуватися наступний код:

```
SIGNAL{
    if (nw == 0 && dr > 0) {
        dr = dr-1; V(r); # відновити процес-читач
    }
    else if (nr == 0 && nw == 0 && dw > 0) {
        dw = dw-1; V(w); # відновити процес-письменник
    }
    else
        V(e); # звільнити блокування входу
}
```

Роль коду `SIGNAL` — сигналізувати тільки одному з трьох семафорів. Це означає, що якщо немає активних письменників, але є призупинений читач, то він може бути продовжений за допомогою операції `V(r)`. Якщо немає активних читачів або письменників, але є призупинений письменник, то він може бути продовжений завдяки операції `V(w)`. Інакше, якщо немає відкладених процесів, які можна безпечно продовжити, то вхідний семафор отримає сигнал за допомогою операції `V(e)`.

```
# Схема читачів і письменників з передачею естафети
int nr = 0, nw = 0;
sem e = 1,          # керує входом в критичні секції
r = 0,              # використовується для призупинки читачів
w = 0;              # для призупинки письменників
                    # завжди 0 <= (e + r + w) <= 1
int dr = 0;         # число призупинених читачів
int dw = 0;         # число призупинених письменників
process Reader [i = 1 to M] {
    while (true) {
        # реалізація ⟨await (nw == 0) nr = nr + 1;⟩
        P(e);
        if (nw > 0){
            dr = dr + 1;
            V(e);
            P(r);
        }
        nr = nr + 1;
        SIGNAL;
        читати базу даних;
        # реалізація ⟨nr = nr - 1;⟩
```

```
            P(e);
            nr = nr - 1;
            SIGNAL;
        }
    }
    process Writer [i = 1 to N] {
        while (true) {
            # реалізація ⟨await (nr == 0 and nw == 0) nw = nw + 1;⟩
            P(e);
            if (nr > 0 || nw > 0) {
                dw = dw + 1;
                V(e);
                P(w);
            }
            nw = nw + 1;
            SIGNAL;
            записати в базу даних;
            # реалізація ⟨nw = nw-1;⟩
            P(e);
            nw = nw-1;
            SIGNAL;
        }
    }
```

Три семафора утворюють розділений двійковий семафор, оскільки в будь-який момент часу тільки один з них може мати значення 1, а всі гілки коду починаються операціями `P` і закінчуються операціями `V`. Отже, оператори між кожною парою `P` та `V` виконуються із взаємним виключенням. Перетворення коду не приводить до взаємного блокування, оскільки семафор затримки отримує сигнал, тільки якщо певний процес знаходиться в стані очікування або повинен в нього перейти. (Процес може збільшити лічильник очікуючих процесів і виконати операцію `V(e)`, але не може виконати операцію `P` для семафора затримки.)

Описаний метод програмування називається *передачею естафети* у зв’язку із способом вироблення сигналів семафора. Коли процес виконується всередині критичної секції, вважається, що він отримав естафету, яка підтверджує його право на виконання. Передача естафети відбувається, коли процес доходить до фрагмента програми `SIGNAL`. Якщо деякий процес очікує умову, яка тепер стала істинною, естафета передається одному з тих процесів, який в свою чергу виконує критичну секцію і передає естафету наступному процесу. Якщо жоден з процесів не очікує умови, естафета передається наступному процесу, який вперше намагається увійти в критичну секцію, тобто наступному процесу, який виконує `P(e)`.

7.5. Розподіл ресурсів та планування

Розподіл ресурсів — це задача визначення того, коли процес може отримати доступ до ресурсу. У паралельних програмах ресурсом є все те, при спробі отримання чого робота процесу може бути припинена. Сюди включається вхід в критичну секцію, доступ до бази даних, доступ до пам'яті, використання принтера і тому подібне. У більшості попередніх програм використовувалася найпростіша стратегія розподілу ресурсів: якщо певний процес чекає і ресурс вільний, то ресурс розподіляється. Наприклад, в задачі критичної секції дозвіл на вхід дається *якомусь* з очікуючих процесів; спроба визначити, *який саме* процес отримає дозвіл на вхід, не робиться. Лише в задачі про читачів та письменників розглядалася більш складна стратегія планування. Але там було надано перевагу класу процесів, а не окремим процесам.

7.5.1. Постановка задачі та загальна схема розв'язку

У будь-якій задачі розподілу ресурсів процеси конкурують за використання елементів ресурсу. Процес запитує один або кілька елементів, виконуючи операцію `request`, часто реалізовану у вигляді процедури. Параметри операції `request` вказують необхідну кількість елементів ресурсу, визначають особливі характеристики (наприклад, розмір блоку пам'яті) і ідентифікують процес, який зробив запит на ресурс. Запит може бути задоволений, тільки коли всі необхідні елементи вільні. Таким чином, процедура `request` очікує звільнення достатньої кількості елементів ресурсу, а потім повертає потрібні елементи. Після використання елементів ресурсу процес звільняє їх операцією `release`.

Загальна спрощена схема операцій `request` та `release` така:

```
request(параметри):
    ⟨await (запит може бути задоволений) отримати
    елементи;⟩
release(параметри):
    ⟨повернути елементи;⟩
```

Операції повинні бути неподільними, оскільки кожної з них необхідний доступ до подання елементів ресурсу.

Цю загальну схему рішення можна реалізувати за допомогою методу передачі естафети. Операція `request` має вигляд звичайного оператора `await`, тому реалізується наступним фрагментом програми.

```
request(параметри):
    P(e);
```

```
    if (запит не може бути задоволений) DELAY;
    отримати елементи;
    SIGNAL;
```

Операція `release` теж має вигляд простої неподільної дії і реалізується таким чином:

```
release(параметри):
    P(e);
    повернути елементи;
    SIGNAL;
```

Як і раніше, семафор `e` керує входом в критичну секцію, а `SIGNAL` запускає на виконання призупинені процеси (якщо очікуючий запит може бути задоволений) або знімає блокування семафора входу за допомогою операції `V(e)`. Код `DELAY` операції `request` аналогічний фрагментами коду на початку «Схема читачів і письменників з передачею естафети» (див. останній лістинг попереднього підрозділу). Він запам'ятовує, що з'явився новий запит, який має бути призупинений, знімає блокування з семафора входу за допомогою операції `V(e)` і блокує процес на семафорі затримки.

7.5.2. Розподіл ресурсів за схемою «найкоротше завдання»

«Найкоротше завдання» (НЗ, Shortest-Job-Next) — це стратегія розподілу ресурсів, яка зустрічається у багатьох різновидах і використовується для різних типів ресурсів. Припустимо, що ресурс складається з одного елемента. Тоді стратегія *«найкоротше завдання»* визначається наступним чином:

> Кілька процесів змагаються у використанні одного ресурсу. Процес запитує ресурс, виконуючи операцію `request(time, id)`, де цілочисловий параметр `time` визначає тривалість використання ресурсу цим процесом, а ціле число `id` ідентифікує процес. Якщо в момент виконання операції `request` ресурс вільний, він виділяється для процесу негайно, інакше процес призупиняється. Після використання ресурсу процес звільняє його, виконуючи операцію `release`. Звільнений ресурс розподіляється призупиненому процесу (якщо такий є) з найменшим значенням параметра `time`. Якщо у декількох процесів значення `time` рівні, то ресурс надається тому з них, хто найдовше чекав.

Стратегію НЗ можна використовувати, наприклад, для розподілу процесорів (параметр `time` означає час виконання), для виведення файлів на принтер (`time` — час друку) або для обслуговування віддаленої передачі файлів по протоколу `ftp` (`time` — передбачуваний час передачі файлу).

Стратегія НЗ приваблива, оскільки мінімізує загальні витрати часу на виконання. Разом з тим, їй притаманне несправедливе планування: процес може бути припинений назавжди, якщо існує безперервний потік запитів з меншим часом використання ресурсу. Якщо несправедливість небажана, то можна трохи змінити стратегію НЗ, щоб перевага віддавалася процесам, які чекають занадто довго. Цей метод називається *витримкою*, або *старінням* (*aging*).

Ресурс один, тому для зберігання відомостей про його доступності досить однієї змінної. Нехай `free` – логічна змінна, яка істинна, коли ресурс доступний, і хибна, коли він зайнятий. Для реалізації стратегії НЗ потрібно запам'ятовувати і впорядковувати очікують запити. Нехай `pairs` — набір записів вигляду `(time, id)`, упорядкованих за значеннями поля `time`. Якщо два записи мають однакові значення поля `time`, то вони залишаються у множині `pairs` в порядку їх появи.

Нижче наведено крупномодульний розв'язок:

```
request(time, id):
    P(e);
    if (!free) DELAY;
    free = false;
    SIGNAL;

release():
    P(e);
    free = true;
    SIGNAL;
```

В операції `request` передбачається, що операції `P` над семафором входу `e` обслуговуються в порядку їх появи, тобто за правилом «першим прийшов — першим обслужений». Якщо цього немає, то порядок обробки запитів може не відповідати стратегії НЗ.

Залишилося реалізувати стратегію розподілу ресурсів НЗ. Для цього використовуємо впорядковану множину `pairs` і семафори, що реалізують фрагменти коду `SIGNAL` і `DELAY`. Якщо запит не може бути задоволений, його слід зберегти, щоб до нього можна було повернутися після звільнення ресурсу. Таким чином, у фрагменті коду `DELAY` процес повинен:

- вставити параметри запиту в упорядковану множину `pairs`;
- звільнити управління критичної секцією, виконавши операцію `V(e)`;
- зупинитися на семафорі до задоволення запиту.

Якщо після звільнення ресурсу `pairs` не є порожньою, то у відповідності зі стратегією НЗ тільки один процес повинен отримати ресурс. Таким чином, якщо є призупинений процес, який тепер може продовжити роботу, він повинен отримати сигнал за допомогою операції `V` для семафора затримки.

У кожного процесу є своя умова затримки, яка залежить від його позиції в наборі `pairs`: перший в `pairs` процес повинен бути запущений перед другим і так далі. Тому кожний процес повинен очікувати на своєму семафорі затримки.

Припустимо, що ресурс використовують `n` процесів. Нехай `b[n]` — масив семафорів, кожний елемент якого має початкове значення `0`. Будемо вважати, що значення ідентифікаторів процесів `id` унікальні і знаходяться в межах від `0` до `n-1`. Тоді процес `id` призупиняється на семафорі `b[id]`. Доповнивши операції `request` і `release` відповідної обробкою `pairs` і масиву `b`, отримаємо:

```
# Семафорний розв'язок задачі розподілу ресурсів за схемою НЗ
bool free = true;
sem e = 1, b[n] = ([n] 0);    # для входу і затримки
typedef Pairs = set of (int, int);
Pairs pairs = ∅;

request (time, id):
    P(e);
    if (!free) {
        вставити (time, id) в pairs;
        V(e);                  # зняти блокування входу
        P(b[id]);              # очікувати відновлення
    }
    free = false;
    V(e);

release ():
    P(e);
    free = true;
    if (pairs != ∅) {
        видалити першу пару (time, id) з pairs;
        # Передати естафету процесу id:
        V(b[id]);
    }
    else V(e);
```

Елементи масиву семафорів `b` є прикладом так званих *приватних семафорів*. Семафор s називається приватним, якщо операцію `P` над ним виконує тільки один процес. Коли процес повинен бути призупинений, він виконує операцію `P(b[id])` для блокування на власному елементі масиву `b`.

Задача 7.1. Узагальнити попередню програму для використання ресурсів, що складаються з кількох елементів. (У цій ситуації кожен елемент може бути вільний або зайнятий, а операції `request` і `release` повинні використовувати параметр `amount`, що визначає необхідну кількість елементів ресурсу.)

8. МОНІТОРИ

Семафори є фундаментальним механізмом синхронізації. Їх використання полегшує програмування взаємного виключення і сигналізації. Однак семафори — низькорівневий механізм; користуючись ними, легко наробити помилок. Програміст повинен стежити за тим, щоб випадково не пропустити виклики операцій `P` і `V` або задати їх більше, ніж потрібно. Семафори глобальні по відношенню до всіх процесів, тому, щоб розібратися, як використовується семафор або інша колективна змінна, необхідно переглянути всю програму. Нарешті, при використанні семафорів взаємне виключення і умовна синхронізація програмуються однієї і тієї ж парою примітивів. Через це важко зрозуміти, для чого призначені конкретні `P` і `V`, не подивившись на інші операції з даними семафорами. Взаємне виключення і умовна синхронізація — це різні поняття, тому і програмувати їх краще різними способами.

Концепція монітора була розроблена Ч. Хоаром [6]. *Монітори* — це програмні модулі, які забезпечують більшу структурованість, ніж семафори, хоча реалізуються так само ефективно. В першу чергу, монітори є механізмом абстракції даних. Монітор інкапсулює поняття абстрактного об'єкта і забезпечує набір операцій, тільки за допомогою яких він обробляється. Монітор містить змінні, що зберігають стан об'єкта, і процедури, що реалізують операції над ним. Процес отримує доступ до змінних в моніторі тільки шляхом виклику процедур цього монітора. Взаємне виключення забезпечується неявно тим, що процедури в одному моніторі не можуть виконуватися паралельно. Умовна синхронізація в моніторах забезпечується явно за допомогою *умовних змінних* (*condition variables*). Вони аналогічні семафорам, але мають істотні відмінності у визначенні та, отже, в використанні для сигналізації.

Паралельна програма, яка використовує монітори для взаємодії і синхронізації, містить два типи модулів: активні процеси і пасивні монітори. За умови, що всі колективні змінні знаходяться всередині моніторів, два процеси взаємодіють, викликаючи процедури одного і того ж монітора. Отримана модульність має два важливих переваги. Перше — процес, що викликає процедуру монітора, може не знати про конкретну реалізації процедури; роль грають лише видимі результати виклику процедури. Друге — програміст монітора може не піклуватися про те, де і як використовуються процедури монітора, і вільно змінювати його реалізацію. Ці переваги дозволяють розробляти процеси і монітори відносно незалежно, що полегшує створення та розуміння паралельної програми.

8.1. Синтаксис та семантика

Монітор використовується, щоб згрупувати зображення і реалізацію спільного ресурсу (класу). Він складається з інтерфейсу та тіла. Інтерфейс визначає надані ресурсом операції (методи). Тіло містить змінні, що описують стан ресурсу, і процедури, що реалізують операції інтерфейсу.

Надалі будемо вважати, що монітор є статичним об'єктом, а його тіло і інтерфейс описані так:

```
monitor mname {
    оголошення постійних змінних
    оператори ініціалізації
    процедури
}
```

Процедури реалізують видимі операції. *Постійні змінні* (*permanent variables*) поділяються всіма процедурами тіла монітора. Вони називаються *постійними*, оскільки існують і зберігають своє значення, поки існує монітор. У процедурах можна використовувати локальні змінні, копії яких створюються для кожного виклику функції.

Монітор як представник абстрактних типів даних має три властивості. По перше, поза монітором видно тільки імена процедур. Щоб змінити стан ресурсу, процес повинен викликати одну з процедур монітора. Виклик процедури монітора має такий вигляд.

```
mname.opname(arguments)
```

Тут `mname` — ім'я монітора, `opname` — ім'я однієї з його операцій (процедур), що викликається з аргументами `arguments`.

По-друге, оператори всередині монітора (в оголошеннях і процедурах) *не можуть звертатися до змінних, оголошених поза монітором*.

По-третє, постійні змінні ініціалізуються до виклику його процедур.

8.1.1. Взаємне виключення

Синхронізацію найпростіше зрозуміти і запрограмувати, якщо взаємне виключення і умовна синхронізація виконуються різними способами. Найкраще, якщо взаємне виключення відбувається неявно, чим автоматично усувається взаємний вплив. Крім того, програми легше читати, оскільки в них немає явних протоколів входу в критичні секції і виходу з них.

На відміну від взаємного виключення, умовну синхронізацію потрібно програмувати явно, оскільки різні програми вимагають різних умов синхронізації. Хоча часто простіше синхронізувати за допомогою логічних умов, як в операторах `await`, низькорівневі механізми можна реалізувати набагато ефективніше.

Відповідно до цих зауваженнями взаємне виключення в моніторах *забезпечується неявно*, а умовна синхронізація програмується за допомогою так званих *умовних змінних*.

Зовнішній процес викликає процедуру монітора. Поки певний процес виконує оператори процедури, вона *активна*. У будь-який момент часу може бути активним тільки один екземпляр тільки однієї процедури монітора, тобто одночасно не можуть бути активними ні два виклики різних процедур, ні два виклики однієї і тієї ж процедури.

Процедури моніторів за визначенням виконуються із взаємним виключенням. Воно забезпечується реалізацією мови, бібліотекою або операційною системою, але не програмістом, який використовує монітори. На практиці взаємне виключення в мовах і бібліотеках реалізується за допомогою блокування та семафорів.

8.1.2. Умовні змінні

Умовна змінна використовується для припинення роботи процесу, безпечне виконання якого неможливо до переходу монітора в стан, що задовольняє деяку логічну умову. Умовні змінні також застосовуються для запуску призупинених процесів, коли певна логічна умова стає істинною. Умовна змінна оголошується наступним чином:

```
cond cv;
```

Таким чином, `cond` — це новий тип даних. *Умовні змінні можна оголошувати і використовувати тільки в межах моніторів*. Значенням умовної змінної `cv` є черга призупинених процесів (черга затримки). Спочатку вона порожня. Програміст не може безпосередньо звертатися до значення змінної `cv`. Замість цього він отримує непрямий доступ до черги за допомогою декількох спеціальних операцій, описаних нижче.

Процес може запросити стан умовної змінної за допомогою виклику

```
empty(cv)
```

Якщо черга змінної `cv` порожня, ця функція повертає значення `true`, інакше — `false`.

Процес блокується на умовній змінній `cv` за допомогою виклику

```
wait(cv)
```

Операція `wait` змушує працюючий процес затриматися в кінці черги змінної `cv`. Щоб інший процес міг врешті-решт увійти в монітор для запуску призупиненого процесу, виконання операції `wait` *відбирає у процесу, що викликав її, винятковий доступ до монітора*.

Процеси, заблоковані на умовних змінних, запускаються операторами `signal`. При виконанні виклику

```
signal(cv)
```

перевіряється черга затримки змінної cv. Якщо вона порожня, ніякі дії не відбуваються. Однак, якщо призупинені процеси є, оператор `signal` запускає процес *у початку черги*. Таким чином, операції `wait` та `signal` забезпечують порядок сигналізації FIFO: процеси припиняються в порядку викликів операції `wait`, а запускаються в порядку викликів операції `signal`.

8.1.3. Дисципліни сигналізації

Виконуючи оператор `signal`, процес працює в моніторі і, отже, може керувати блокуванням, неявно пов'язаним із монітором. В результаті виникає дилема. Якщо операція `signal` запускає інший процес, то виходить, що могли б виконуватися два процеси: той, що викликав операцію `signal` та запущений нею. Але наступним може виконуватися тільки один з них, оскільки лише один процес може мати винятковий доступ до монітора. Таким чином, можливі два варіанти:

- *сигналізувати і продовжити*: сигналізатор продовжує роботу, а процес, що отримав сигнал, виконується пізніше;
- *сигналізувати і очікувати*: сигналізатор чекає деякий час, а процес, який отримав сигнал, виконується відразу.

Дисципліна (порядок) «сигналізувати і продовжити» *не перериває обслуговування*. Процес, що виконує операцію `signal`, зберігає винятковий доступ до монітора, а процес, що запускається, почне роботу трохи пізніше, коли отримає винятковий доступ до монітора. По суті, операція `signal` просто вказує процесу, який запускається, на можливість виконання, після чого він повертається в чергу процесів, що очікують.

Порядок «сигналізувати і очікувати» має властивість *переривання обслуговування*. Процес, що виконує операцію `signal`, передає управління блокуванням монітора процесу, який запускається, тобто переривається робота процесу-сигналізатора. В цьому випадку сигналізатор переходить в чергу процесів, які очікують на звільнення монітора.

Діаграма станів на рис. 8.1 ілюструє роботу синхронізації в моніторах. Викликаючи процедуру монітора, процес поміщається у вхідну чергу, якщо в моніторі виконується ще один процес; в іншому випадку процес, який викликав операцію, негайно починає виконання в моніторі. Коли монітор звільняється (після повернення з процедури або виконання операції `wait`), один процес з вхідної черги може перейти до роботи в моніторі. Виконуючи операцію `wait(cv)`, процес переходить від роботи в моніторі в чергу, пов'язану з умовною змінною. Якщо процес виконує операцію `signal(cv)`, то при порядку «Сигналізувати і продовжити» (Signal and Continue — SC) процес з початку черги умовної змінної переходить до вхідної. При порядку «сигналізувати і очікувати»

(Signal and Wait — SW) процес, що виконується в моніторі, переходить до вхідної черги, а процес з початку черги умовної змінної переходить до виконання в моніторі.

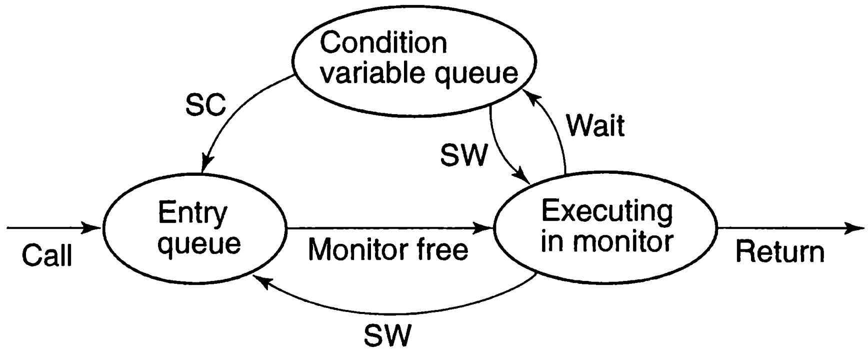

Рис. 8.1. Діаграма станів

Розглянемо приклад, який демонструє можливість реалізації семафор за допомогою монітора.

```
# Реалізація семафора за допомогою монітора
monitor Semaphore {
    int s = 0;
    cond pos; # отримує сигнал, коли s > 0
    procedure Psem() {
        while (s == 0) wait(pos);
        s = s - 1;
    }
    procedure Vsem() {
        s = s + 1;
        signal(pos);
    }
}
```

Викликаючи операцію `Psem`, процес призупиняється, поки значення змінної `s` не стане додатним, потім зменшує його на 1. Затримка програмується за допомогою циклу `while`, який призводить процес до очікування на умовній змінній `pos`, якщо `s` дорівнює 0. Операція `Vsem` збільшує `s` на 1 і виробляє сигнал для змінної `pos`. Якщо є призупинені процеси, запускається «найстаріший» з них.

Програма коректно працює як при порядку SW, так і при SC. Порядки роботи відрізняються тільки послідовністю виконання процесів. При порядку SW процес, що запускається, виконується відразу та зменшує значення змінної `s`. При порядку SC процес, що запускається, виконується через деякий час *після* процесу, який виробив сигнал. Тому він має ще раз перевірити значення семафора щоб переконатися, що воно ще додатне. Це треба зробити для того, що можливо інший процес із вхідної черги викликав `Psem` і вже зменшив `s`. (Тому, на відміну від порядку SW, для SC ми не обов'язково отримуємо FIFO-семафор). Вищенаведений монітор можна змінити так, щоб він коректно працював при обох порядках запуску процесів (SC і SW), не використав цикл `while` та реалізовував семафор з порядком обслуговування FIFO. Як вже зазначалося проблема полягає у тому, що збільшене значення `s` може прочитати не той процес, який був запущений за допомогою `signal`. Вихід — виклик `VSem` у випадку наявності призупинених процесів не має збільшувати значення `s`. Такий метод називається *передачею умови*:

```
# Семафор FIFO з передачею умови

monitor FIFOsemaphore {
    int s = 0;
    cond pos; # Отримує сигнал, коли s > 0

    procedure Psem() {

        if (s == 0)
            wait(pos);
        else
            s = s - 1;
    }

    procedure Vsem() {

        if (empty(pos))
            s = s + 1;
        else
            signal(pos);
    }
}
```

Метод передачі умови можна застосовувати всюди, де в процедурах з викликами `wait` та `signal` є дії, що доповнюють одна одну. У програмі такими є збільшення змінної `s` в процедурі `Psem` та її зменшення в `Vsem`.

З попередньої програми видно, що умовні змінні аналогічні операціям `P` і `V` семафора. Операція `wait` призупиняє процес, а операція `signal` запускає його. Однак є дві істотні відмінності. Перша — операція `wait` завжди призупиняє процес до подальшого виконання операції `signal`, тоді як операція `P` викликає зупинку процесу, тільки якщо поточне значення семафора дорівнює нулю. Друге — операція `signal` не виробляє ніяких дій, якщо немає процесів, призупинених на умовній змінній, тоді як `V` або запускає призупинений процес, або збільшує значення семафора, тобто факт здійснення операції `signal` не запам'ятовується. Через ці відмінності умовна синхронізація з моніторами програмується не так, як з семафорами.

Надалі для моніторів *будемо використовують лише порядок SC*, який був прийнятий в ОС Unix, мові Java, бібліотеці Pthreads та .NET.

8.2. Методи синхронізації

8.2.1. Кільцеві буфери: базова умовна синхронізація

Повернемося до задачі про кільцевий буфер (див. підрозділ 7.2). Процес-виробник і процес-споживач взаємодіють за допомогою спільного буфера, який містить `n` комірок.

Нижче наведений монітор, який реалізує кільцевої буфер. Для черги повідомлень знову використані масив `buf` і дві цілочислові змінні `front` та `rear`. У змінній `count` зберігається кількість повідомлень в буфері. Операції з буфером `deposit` та `fetch` стають процедурами монітора. Умовна синхронізація реалізована за допомогою двох умовних змінних. Обидва оператора `wait` знаходяться в циклах.

```
# Реалізація кільцевого буфера за допомогою монітора
monitor Bounded_Buffer {
    typeT buf[n];  # масив деякого типу T
    int front = 0,     # індекс першої заповненої комірки
    rear = 0,          # індекс першої порожньої комірки
    count = 0;         # Число заповнених комірок
                       # rear == (front + count)%n
    cond not_full;     # отримує сигнал, коли count < n
    cond not_empty;    # отримує сигнал, коли count > 0
    procedure deposit(typeT data) {
        while (count == n) wait(not_full);
        buf[rear] = data;
        rear = (rear + 1) % n;
        count++;
        signal(not_empty);
    }

    procedure fetch(typeT& result) {
        while (count == 0) wait(not_empty);
        result = buf[front];
        front = (front + 1) % n;
        count--;
        signal(not_full);
    }
}
```

Виконуючи операцію `signal`, процес просто повідомляє, що тепер деяка умова істинна. Оскільки процес-сигналізатор і, можливо, інші процеси можуть виконуватися в моніторі до відновлення процесу, запущеного операцією `signal`, в момент початку його роботи умова запуску може вже не виконуватися. Наприклад, процес-виробник був припинений в очікуванні вільної комірки у буфері, потім процес-споживач зчитав повідомлення і запустив призупинений процес. Однак до того, як цьому виробникові прийшла черга виконуватися, інший процес-виробник міг уже увійти в процедуру `deposit` і зайняти вільну позицію. Аналогічна ситуація може виникнути із споживачами. Таким чином, умову припинення необхідно перевіряти ще раз. Оператори `signal` в процедурах `deposit` і `fetch` виконуються безумовно, оскільки в момент їх виконання умова, про яке вони сигналізують, є істиною. Оператори `wait` знаходяться в циклах, тому оператори `signal` можуть виконуватися в будь-який момент часу, оскільки вони просто підказують припиненим процесам. Однак програма виконується більш ефективно, коли `signal` виконується, тільки якщо відомо напевно (або хоча б з великою ймовірністю), що деякий призупинений процес може бути продовжений.

8.2.2. Читачі та письменники: сигнал сповіщення

Задача про читачів і письменників була наведена у підрозділі 7.4. Хоча база даних — загальний ресурс, її не можна уявити монітором, оскільки тоді читачі не зможуть працювати з нею паралельно (весь код усередині монітора виконується зі взаємним виключенням). Монітор використовується для упорядкування доступу до бази даних.

У задачі про читачів і письменників монітор дає дозвіл на доступ до бази даних. Для цього необхідно, щоб процеси інформували монітор про своє бажання отримати доступ і про завершення роботи з базою даних. Є два типи процесів і по два види дій на процес, тому отримуємо чотири процедури монітора:

`request_read`, `release_read`, `request_write`, `release_write`.

Для синхронізації доступу до бази даних необхідно вести облік числа процесів-читачів та письменників. Як і раніше, нехай значення змінної `nr` — це число читачів, а `nw` — письменників. У початковому стані `nr` і `nw` рівні 0. Їх значення збільшуються при виклику процедур запиту і зменшуються при виклику процедур звільнення. Нижче наведено монітор, що відповідає цій специфікації.

```
# Розв'язок задачі про читачів та письменників
monitor RW_Controller {
    int nr = 0, nw = 0;
    # (nr == 0 || nw == 0) && nw <= 1
```

```
    cond oktoread;  # Отримує сигнал, коли nw == 0
    cond oktowrite; # Отримує сигнал, коли nr == 0 і nw == 0

    procedure request_read() {
        while (nw > 0) wait(oktoread);
        nr = nr + 1;
    }

    procedure release_read() {
        nr = nr - 1;
        if (nr == 0) signal(oktowrite);
        # запустити процес-письменник
    }

    procedure request_write() {
        while (nr > 0 || nw > 0) wait(oktowrite);
        nw = nw + 1;
    }

    procedure release_write() {
        nw = nw - 1;
        signal(oktowrite);      # запустити процес-письменник
        signal_all(oktoread); # запустити всі процеси-читачі
    }
}
```

Програма не встановлює порядок чергування процесів-читачів і процесів-письменників. Замість цього дана програма запускає *всі* призупинені процеси і дозволяє стратегії планування процесів визначити, який з них першим отримає доступ до бази даних. Якщо це процес-письменник, то припиняться *всі* запущені процеси-читачі. Якщо ж першим отримає доступ процес-читач, то призупиниться відновлений процес-письменник.

8.2.3. Розподіл ресурсів за схемою «найкоротше завдання»

Умовна змінна за замовчуванням є FIFO-чергою, тому, виконуючи оператор `wait`, процес потрапляє в кінець черги очікування. Оператор пріоритетного очікування `wait(cv, rank)` має призупинені процеси в порядку зростання рангу. Він використовується для реалізації стратегій планування, відмінних від FIFO. Знову розглянемо задачу розподілу ресурсів за схемою «найкоротше завдання», наведену в 7.5.

Для розподілу ресурсів за схемою «найкоротше завдання» потрібні дві операції: `request` та `release`. У наступній програмі наведено монітор, який реалізує розподіл ресурсів згідно стратегії НЗ. Використовується логічна змінна `free` для індикації того, що ресурс вільний, і умовна змінна `turn` для припинення процесів.

Процедури використовують метод передачі умови. Пріоритетний оператор `wait` застосовується для сортування призупинених процесів за часом, протягом якого вони будуть використовувати ресурс. Функція `empty` використовується для перевірки, чи є призупинені процеси. Коли ресурс звільняється, при наявності призупинених процесів запускається той, якому потрібно найменше часу, інакше ресурс позначається як вільний. Якщо процес отримує сигнал, то відмітки про звільнення ресурсу не робиться, щоб інший процес не отримав до нього доступ першим.

```
# Розподіл ресурсів за стратегією НЗ з використанням моніторів

monitor Shortest_Job_Next {
    bool free = true;
    cond turn; # Отримує сигнал, коли ресурс доступний

    procedure request(int time) {
        if (free)
            free = false;
        else
            wait(turn, time);
    }

    procedure release() {
        if (empty(turn))
            free = true;
        else
            signal(turn);
    }
}
```

8.2.4. Сплячий перукар: рандеву

В якості останнього базового прикладу розглянемо ще одну класичну задачу синхронізації: задачу про сплячого перукаря. Вона має практичні застосування, наприклад у плануванні роботи головки дискового накопичувача. Ця задача ілюструє важливість зв’язків типу клієнт-сервер, які часто існують між процесами. Для неї необхідний особливий тип синхронізації, так зване рандеву.

Задача про сплячого перукаря. У тихому містечку є перукарня з двома дверима і декількома кріслами. Відвідувачі входять через одні двері і виходять через іншу. Салон малий, і ходити по ньому може тільки перукар і один відвідувач. Перукар все життя обслуговує відвідувачів. Коли в салоні нікого немає, він спить у своєму кріслі. Коли відвідувач приходить і бачить сплячого перукаря, він будить його, сідає в крісло і спить, поки той зайнятий стрижкою. Якщо перукар зайнятий, коли приходить відвідувач, той сідає в одне з вільних крісел і засинає. Після стрижки перукар відкриває відвідувачеві вихідні двері і закриває її за ним. Якщо є відвідувачі, які очікують, перукар будить одного з них і чекає, поки той сяде в крісло перукаря. Якщо нікого немає, він знову йде спати до приходу наступного відвідувача.

Відвідувачі і перукар є процесами, що взаємодіють в моніторі — перукарні. Відвідувачі — це клієнти, які запитують сервіс (стрижку) у перукаря. Перукар — це сервер, який постійно забезпечує сервіс.

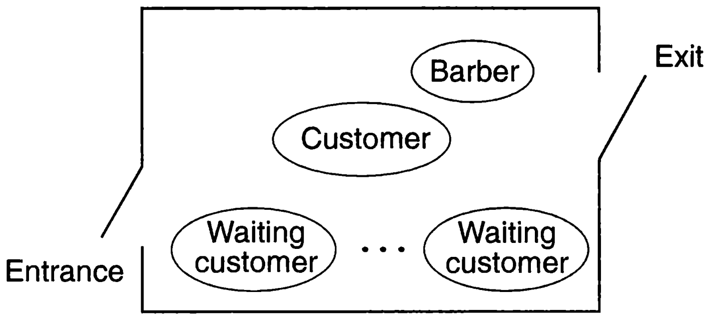

Рис. 8.2. Ілюстрація до задачі про сплячого перукаря

Для реалізації описаних взаємодій перукарню можна змоделювати монітором з трьома процедурами: `get_haircut` (постригтися), `get_next_customer` (покликати наступного) і `finished_cut` (закінчити стрижку). Відвідувачі викликають процедуру `get_haircut`; вихід з неї відбувається після того, як перукар закінчить стрижку даного відвідувача. Перукар циклічно викликає процедуру `get_next_customer`, запрошуючи клієнта в своє крісло, стриже його і випускає з перукарні за допомогою виклику процедури `finished_cut`. Постійні змінні служать для зберігання стану процесів і крісел, в яких процеси сплять. Дії перукаря і відвідувачів необхідно синхронізувати в моніторі. По перше, перукаря і відвідувачеві необхідна зустріч — *рандеву*, тобто перукар повинен дочекатися приходу відвідувача, а відвідувач — звільнення перукаря. Рандеву аналогічне до бар’єра для двох процесів, оскільки для продовження роботи до нього повинні прийти обидві сторони. Однак рандеву відрізняється від бар’єра тим, що перукар може зустрітися з будь-яким з відвідувачів.

По-друге, відвідувачеві необхідно чекати, поки перукар закінчить його стригти, що визначається відкриттям вихідних дверей для відвідувача. Нарешті, перед тим, як закрити вихідні двері, перукар повинен почекати, поки піде відвідувач. Таким чином, перукар і відвідувач проходять через послідовність етапів, що починаються з рандеву.

Найпростіший спосіб визначити подібні етапи синхронізації — використати зростаючі лічильники для запам’ятовування числа процесів, які досягли кожного етапу. У відвідувачів є два важливих етапи: перебування в кріслі перукаря і вихід з перукарні. Для цих етапів будемо використовувати лічильники `cinchair` і `cleave`. Перукар циклічно проходить через три етапи: звільнення від роботи, стрижка і завершення стрижки. Використовуємо для них лічильники `bavail`, `bbusy` і `bdone`. Всі лічильники в початковому стані мають значення нуль. Оскільки процеси проходять свої етапи послідовно, для лічильників виконується наступний інваріант:

```
cinchair >= cleave && bavail >= bbusy >= bdone
```

Щоб забезпечити рандеву відвідувача і перукаря перед початком стрижки, відвідувач не може сідати в крісло перукаря частіше, ніж перукар звільняється від роботи. Крім того, перукар не може починати стрижку частіше, ніж відвідувачі сідають в його крісло. Отже, виконується умова:

```
cinchair <= bavail && bbusy <= cinchair
```

Нарешті, відвідувачі не можуть виходити частіше, ніж перукар завершує стрижку:

```
cleave <= bdone
```

Зростаючі лічильники застосовуються для запам'ятовування етапів, через які проходять процеси, проте їх значення можуть зростати необмежено. Якщо синхронізація залежить тільки від різниці значень лічильників, зростання можна уникнути, змінивши змінні. У задачі є три ключові різниці, для яких виділимо три нові змінні `barber`, `chair` і `open`.

```
barber == bavail - cinchair
chair == cinchair - bbusy
open == bdone - cleave
```

Вони ініціалізуються 0, а під час роботи програми можуть приймати значення 0 або 1. Значення `barber` дорівнює 1, якщо перукар очікує відвідувача і сидить в своєму кріслі. Змінна `chair` має значення 1, якщо відвідувач вже сів в крісло, але перукар ще не зайнятий, а змінна `open` приймає значення 1, коли вихідні двері вже відкриті, але відвідувач ще не вийшов.

```
# Монітор для задачі про сплячого перукаря
monitor Barber_Shop {
    int barber = 0, chair = 0, open = 0;
    cond barber_available; # Отримує сигнал, коли barber > 0
    cond chair_occupied;   # Отримує сигнал, коли chair > 0
    cond door_open;        # Отримує сигнал, коли open > 0
    cond customer_left;    # Отримує сигнал, коли open == 0

    procedure get_haircut() {
        while (barber == 0)
            wait(barber_available);
```

```
            barber = barber - 1;
            chair = chair + 1; signal(chair_occupied);
            while (open == 0) wait(door_open);
            open = open - 1; signal(customer_left);
        }
        procedure get_next_customer() {
            barber = barber + 1;
            signal(barber_available);
            while (chair == 0)
                wait(chair_occupied);
            chair = chair - 1;
        }
        procedure finished_cut() {
            open = open + 1;
            signal(door_open);
            while (open > 0)
                wait(customer_left);
        }
    }
```

У наведеному моніторі ми вперше бачимо процедуру `get_haircut`, що містить два оператора `wait`. Справа в тому, що відвідувач проходить через два етапи: спочатку він чекає, поки не звільниться перукар, потім — поки не закінчиться стрижка.

9. ПРОГРАМУВАННЯ З ВИКОРИСТАННЯМ ПОТОКІВ

Багатозадачність підтримується практично усіма сучасними операційними системами. Існує два типи багатозадачності: багатозадачність, заснована на *процесах* та багатозадачність, заснована на *потоках*.

Багатозадачність на основі використання *процесів* дає змогу одночасно виконувати на комп'ютері декілька програм. При цьому програма є найменшою одиницею, якою може керувати планувальник операційної системи. Кожний процес потребує окремий адресний простір.

Багатозадачність, заснована на *потоках*, вимагає менших витрат обчислювальних ресурсів, оскільки потоки одного процесу використовують спільний адресний простір. Перемикання та комунікації між потоками також потребують значно меншої кількості ресурсів. Мінімальним елементом керованого коду при багатозадачності на основі потоків є потік (thread).

У сучасних мовах програмування вбудована підтримка програмування з використанням кількох потоків. Програма може містити кілька частин, які виконуються одночасно. Наприклад, текстовий редактор може форматувати текст і паралельно друкувати його на принтері. Кожна така частина програми називається потоком і кожний потік задає окремий шлях виконання програми. Тобто, багатопотоковість є спеціалізованою формою багатозадачності.

Підтримка багатопотоковості дає змогу писати ефективні програми за рахунок використання усіх ресурсів процесора. Ще однією перевагою багатопотокового програмування є зменшення часу очікування. Це є важливим для інтерактивних мережевих систем, для яких очікування та простій є звичним явищем. Наприклад, швидкість передачі даних по мережі суттєво нижча за швидкість обробки даних у межах локальної файлової системи, яка у свою чергу значно нижча за швидкість опрацювання даних центральним процесором системи. В однопотокових програмах потрібно очікувати завершення повільних операцій обробки даних. Тому час простою може бути значним. У багатопотокових програмах за цей самий час можна паралельно виконати ряд «швидких» операцій. При цьому багатопотокові програми використовуються як на одно-, так і на багатоядерних системах.

9.1. Потокова модель Java

Уся бібліотека класів Java спроектована таким чином, щоб забезпечити підтримку багатопотоковості. Перевага багатопотоковості полягає у тому, що не використовується механізм циклічного опитування черги подій. Один потік може бути призупинений без зупинки інших.

Потоки існують у кількох *станах*:

1. потік виконується;
2. потік готується до виконання;

3. потік призупинений (з можливістю відновлення);
4. робота потоку відновлена;
5. потік заблокований;
6. потік перерваний (не може бути відновлений).

Java присвоює кожному потоку *пріоритет*, який визначає поведінку цього потоку стосовно інших потоків. Пріоритети потоків задаються цілими числами, які вказують на відносний пріоритет потоку по відношенню до інших потоків. Слід зазначити, що швидкість виконання потоку з низьким пріоритетом *не відрізняється* від швидкості виконання високопріоритетного потоку, якщо потік є єдиним потоком на даний момент. Але пріоритет суттєво впливає на процес переходу від виконання одного потоку до іншого у випадку багатопотокових програм. Цей процес носить назву *перемикання контексту*. Правила перемикання контексту:

1) Потік може *добровільно передати керування*. Для цього можна явно поступитися місцем у черзі виконання, призупинити потік чи блокувати на час виконання вводу-виводу. При цьому усі інші потоки перевіряються і ресурси процесора передаються готовому до виконання потоку з максимальним пріоритетом.
2) Потік може бути *перерваний іншим більш пріоритетним потоком*. У цьому випадку низькопріоритетний потік, який не займає процесор, призупиняється високопріоритетним потоком незалежно від того, що він робить. Цей механізм називається *багатозадачністю* з *витисненням* (або *багатозадачністю з пріоритетом*).

У випадку, коли два потоки із однаковими пріоритетом претендують на те, щоб використати процесор, ситуація ускладнюється. У ОС Windows ці потоки ділять між собою час процесора. У інших операційних системах потоки повинні примусово передавати керування своїм «родичам».

Багатопотоковість надає програмам можливість асинхронної поведінки. Проте як зазначалося у 4-му розділі, у багатьох випадках при спільному використанні даних кількома потоками виникає потреба у синхронізації. Наприклад, при спільному використанні зв'язного списку потрібно передбачити можливість заборони одному потоку змінювати дані цього списку, поки інший потік зчитує елементи цього списку. Для цього у Java використовуються монітори. Неформально можна сприймати монітор як дуже маленьку скриню, у яку в одиницю часу можна «помістити» лише один потік [16]. Як тільки потік «увійшов» у монітор, усі інші потоки повинні чекати, поки потік не вийде із монітора. Таким чином монітор може бути використаний *для захисту спільних ресурсів від одночасного використання* більше ніж одним потоком.

9.1.1. Клас Thread та інтерфейс `Runnable`

Багатопотокова система Java вбудована у клас `Thread`, його методи та доповнюючий його інтерфейс `Runnable`. Клас `Thread` інкапсулює потік виконання. Для того, щоб створити новий потік, потрібно або розширити клас `Thread` шляхом наслідування від нього, або реалізувати у класі інтерфейс `Runnable`. Клас `Thread` визначає ряд методів, деякі з яких наведені у табл. 9.1.

Таблиця 9.1. Методи керування потоками класу `Thread`

Метод	**Призначення**
`getName`	Отримати ім'я потоку
`getPriority`	Отримати пріоритет потоку
`isAlive`	Визначити, чи виконується потік
`join`	Очікувати завершення виконання потоку
`run`	Вхідна точка потоку
`sleep`	Призупинити виконання потоку на заданий інтервал часу
`start`	Запустити потік на виконання викликом його методу `run`

9.1.2. Головний потік

Коли запускається Java-програма, починає виконуватися один потік, який зазвичай називають головним потоком (*main thread*) програми. Головний потік програми:

1) Породжує усі дочірні потоки.
2) Часто повинен бути останнім потоком, який завершує виконання.

Не дивлячись на те, що головний потік створюється автоматично, ним можна керувати за допомогою методів об'єкта класу `Thread`. Для цього потрібно отримати посилання на нього шляхом виклику методу `currentThread()`. Його опис має вигляд:

```
static Thread currentThread()
```

Цей метод повертає посилання на потік, із якого він був викликаний.
Розглянемо наступний приклад:

```
public static void main(String[] args){
    Thread t = Thread.currentThread();
    System.out.println("Current thread: " + t);
    t.setName("My thread");
    System.out.println("Changed thread name: " + t);
    try {
        for (int i = 5; i > 0; i--) {
            System.out.println(i);
```

```
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
```

Посилання на поточний (у даному випадку головний) потік зберігається у змінній `t`. Затримка реалізується шляхом виклику методу `sleep()`, аргументом якого є тривалість затримки у мілісекундах. Використання блоку `try/ catch` *є обов'язковим*, оскільки метод `sleep()` може згенерувати `InterruptedException`. Це може відбутися у випадку, коли деякий потік захоче перервати виконання поточного потоку. Опис методу `sleep()`:

```
static void sleep(long mls) throws InterruptedException
```

При виводі інформації про потік на консоль відображається ім'я потоку, його пріоритет та ім'я групи потоку, до якої відноситься даний потік — структура даних, яка керує набором потоків у цілому.

9.1.3. Реалізація інтерфейсу Runnable

Інтерфейс `Runnable` абстрагує одиницю виконуваного коду. Для реалізації цього інтерфейсу у класі має бути реалізований метод `run()`:

```
public void run()
```

Всередині цього методу потрібно розташувати код, який визначає дії, які мають виконуватися у новому потоці. У методі `run()` можна викликати інші методи, використовувати інші класи, описувати змінні — так само як це робить головний потік. Єдина відмінність — метод `run()` визначає точку входу іншого, паралельного потоку всередині програми. Цей потік завершується тоді, коли `run()` повертає керування.

При реалізації класу користувача використовуються об'єкти класу `Thread`. У класі `Thread` визначено кілька конструкторів. Можна використовувати, наприклад, наступний конструктор:

```
Thread(Runnable об'єкт_потоку, String ім'я_потоку)
```

У цьому конструкторі `об'єкт_потоку` — це екземпляр класу, який реалізує інтерфейс `Runnable`.

Після того як новий потік буде створений, він не запуститься до тих пір, поки не буде викликано метод `start()` класу `Thread`.

Розглянемо наступний приклад:

```
class MyThread implements Runnable{
    Thread t;
```

```
    public MyThread(){
        t = new Thread(this, "My thread demo");
        System.out.println("My thread " + t + " is created");
        t.start();
    }

    public void run() {
        try {
            for (int i = 5; i > 0; i--) {
                System.out.println("My thread " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e){
            System.out.println("My thread interrupted");
        }
        System.out.println("My thread is terminated");
    }
}
public class MyClass {
    public static void main(String[] args){
        new MyThread();
        try {
            for (int i = 0; i < 5; i++) {
                System.out.println("main " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

Усередині конструктора `MyThread()` міститься наступний рядок коду

```
t = new Thread(this, "My thread demo");
```

Передача об’єкта `this` першим аргументом свідчить про те, що новий потік буде викликати метод `run()` об’єкта `this`. Наступний виклик методу `start()` запускає потік на виконання, у результаті чого починає виконуватися цикл `for`, який міститься у тілі методу `run()`.

9.1.4. Створення нащадків класу `Thread`

Клас-нащадок *обов’язково* повинен перевизначити метод `run()`, який є точкою входу для нового потоку. Для запуску потоку на виконання так само потрібно викликати метод `start()`.

Усе вищесказане продемонстровано на наступному прикладі:

```
class NewThread extends Thread{
    NewThread(String name) {
        super(name);
        System.out.println(this + " is started");
    }

    public void run() {
        for (int i = 10; i > 0; i--) {
            System.out.println(getName()+ ", i = " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.printf("%s is terminated\n", getName());
    }
}

public class MyClass {
    public static void main(String[] args){
        NewThread thread1 = new NewThread("thread 1");
        thread1.setPriority(Thread.MIN_PRIORITY);
        NewThread thread2 = new NewThread("thread 2");
        thread2.setPriority(Thread.MAX_PRIORITY);
        thread1.start();
        thread2.start();
        try {
            for (int i = 0; i <= 5; i++) {
                System.out.println("main " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }
    }
}
```

У конструкторі класу `NewThread` спочатку традиційно викликається конструктор суперкласу.

Слід зазначити, що у [16] рекомендується використовувати для потоків підхід з використанням наслідування тоді, коли виникає потреба модифікувати методи класу `Thread`. Тому у більшості «стандартних» випадків використання потоків реалізація інтерфейсу `Runnable` є достатньою.

Пріоритет потоку задається за допомогою методу

```
final void setPriority(int рівень)
```

Значення рівня пріоритету має лежати у межах від `MIN_PRIORITY` до `MAX_PRIORITY`. На даний момент ці значення рівні відповідно 1 та 10. Значення пріоритету по замовчуванню рівне константі `NORM_PRIORITY` (на даний час її значення рівне 5).

Отримати пріоритет потоку можна викликавши метод `getPriority()`:

```
final int getPriority()
```

9.1.5. Метод join()

Існує два способи перевірки завершення виконання потоку. Найпростіше це зробити з використанням методу `isAlive()` класу `Thread`:

```
final Boolean isAlive()
```

Метод `isAlive()` повертає значення `true`, якщо потік, для якого він викликаний, ще виконується.

Крім того, існує метод, який використовується для очікування завершення виконання потоку — метод `join()`.

```
final void join() throws InetrruptedException
```

Цей метод очікує завершення потоку, для якого він був викликаний. Його назва відображає концепцію, згідно до якої викликаючий потік очікує, поки заданий потік приєднається до нього. Також можна вказувати максимальний час очікування (у мілісекундах). Якщо, наприклад, додати до попередньої програми після рядка коду

```
thread2.start();
```

наступний фрагмент

```
try {
    thread1.join(2000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
System.out.println("thread.isAlive());
```

то у `main` виникне двох секундне очікування завершення потоку `thread1`, і тільки потім почнеться виконання наступних операторів.

9.1.6. Синхронізація

Ключем до синхронізації є концепція монітора, наведена у попередньому розділі. Монітор — це об’єкт, який часто застосовується як взаємно виключне блокування, або *м'ютекс* (див. підрозділ 4.1). Коли потік робить запит на блокування, то кажуть, що він входить у монітор. Усі інші потоки, які намагаються увійти у заблокований монітор, будуть призупинені до тих пір, поки перший не вийде із монітора.

Синхронізація у Java заснована на тому, що об'єкти мають власні, асоційовані із ними неявні монітори. Для цього потрібно просто вказати ключове слово `synchronized` при описі методу. Щоб вийти із монітора і передати керування об'єктом іншому потоку, власник монітора просто передає керування із синхронізованого методу.

Розглянемо приклад, у якому продемонстровано проблеми, які можуть виникати при відсутності синхронізації. У класі `Callme` описаний єдиний метод, який виводить параметр рядок у квадратних дужках, причому закриваюча дужка виводиться із секундною затримкою.

Конструктор класу `Caller` приймає посилання на об'єкти класів `String` та `Callme`, якими ініціалізуються поля об'єкта, та створює і запускає новий потік та викликає метод `run()` цього потоку. У цьому методі викликається метод `call()` відповідного об'єкта `Callme`. Нарешті у методі `main()` створюється один об'єкт `Callme`, який передається у якості параметру конструктора трьох різних об'єктів `Caller` разом із відповідним повідомленням.

```
class Callme{
    void call(String message){
        System.out.print("[" + message);
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e){
            System.out.println("Interrupted");
        }
        finally {
            System.out.println("]");
        }
    }
}
class Caller implements Runnable{
    String message;
    Callme target;
    Thread t;
    Caller(String message, Callme target) {
        this.message = message;
        this.target = target;
```

```
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.call(message);
    }
}
public class Sample {
    public static void main(String[] args){
        Callme target = new Callme();
        Caller obj1 = new Caller("Welcome", target);
        Caller obj2 = new Caller("to synchronized", target);
        Caller obj3 = new Caller("world", target);
        try {
            obj1.t.join();
            obj2.t.join();
            obj3.t.join();
        }
        catch (InterruptedException e){
            System.out.println("Interrupted");
        }
    }
}
```

Вивід програми може мати вигляд

```
[Welcome[to synchronized[world]
]
]
```

У попередньому прикладі три потоки змагаються між собою за виконання метода `call()` одного і того самого об’єкта `Callme`. Така ситуація, як було зазначено у параграфі 6.4.3, називається «станом гонитви».

Для виправлення попередньої програми, звичайно, можна було запускати потоки наступним чином:

```
Callme target = new Callme();
try{
    Caller obj1 = new Caller("Welcome", target);
    obj1.t.join();
    Caller obj2 = new Caller("to synchronized", target);
    obj2.t.join();
    Caller obj3 = new Caller("world", target);
    obj3.t.join();
}
```

```
    catch (InterruptedException e){
    }
```

Але у такий спосіб ми просто очікуємо завершення кожного потоку і сумісне використання ресурсів відсутнє. Вихід із ситуації — *синхронізація* методу `call()` шляхом опису його з використанням ключового слова `synchronized`:

```
class Callme{
    synchronized void call(String message){
    ...
```

Це дозволить уникнути доступу інших потоків до цього методу. Вивід програми має вигляд:

```
    [Welcome]
    [to synchronized]
    [world]
```

Як тільки *потік входить у синхронізований метод* екземпляра, жодний інший потік *не може* викликати цей метод (для того самого екземпляра). Це дозволяє уникати гонитви потоків.

Синхронізація методів у класах не може бути застосована, якщо клас не передбачає багатопотокового доступу або доступ до вихідного коду класу відсутній. В таких випадках для синхронізації потрібно помістити виклик методів класу у блок `synchronized`.

Оператор `synchronized` має наступну форму:

```
    synchronized(об'єкт) {
        // оператори, які підлягають синхронізації
    }
```

Цей оператор гарантує те, що виклик методу об'єкта відбудеться тоді, коли потік увійде у монітор об'єкта. Розглянемо альтернативну версію попереднього прикладу. Для синхронізації достатньо модифікувати метод `run()` класу `Caller`.

```
    public void run() {
        synchronized (target){
            target.call(message);
        }
    }
```

Приклад 9.1. Програма паралельного обчислення скалярного добутку векторів.

```
// файл DotProduct.java
import java.util.ArrayList;
import java.util.List;
```

```
public class DotProduct implements Runnable {
    private final float[] a;
    private final float[] b;
    private final int from;      // portion start (inclusive)
    private final int to;        // portion end (exclusive)
    public final Thread thread;
    private static Double sum = 0d;
    static List<DotProduct> calculators = new ArrayList<DotProduct>();

    public DotProduct(float[] a, float[] b, int from, int to) {
        this.a = a;
        this.b = b;
        this.from = from;
        this.to = to;
        thread = new Thread(this);
        calculators.add(this);
        thread.start();
    }

    public void run() {
        double s = 0;
        for (int i = from; i < to; i++) {
            s += a[i] * b[i];
        }
        synchronized (DotProduct.sum){
            DotProduct.sum += s;
        }
    }
    public static void joinAll() {
        for (var calculator:calculators) {
            try {
                calculator.thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void reset(){
        sum = 0d;
        if(calculators != null)
            calculators.clear();
    }

    synchronized public static double getResult(){
        return sum;
```

```
    }
}
// файл Main.java
import java.util.Arrays;
import static java.lang.System.out
public class Main {
    public static void main(String[] args) {
        final int SIZE = 260_000_000;
        var a = new float[SIZE];
        Arrays.fill(a, 1);
        var b = a;
        // Serial mode:
        var start = System.currentTimeMillis();
        var sum = 0.0;
        for (int i = 0; i < SIZE; i++) {
            sum += a[i] * b[i];
        }
        var duration = System.currentTimeMillis() - start;
        out.printf("Serial mode\n\tresult: %f, time: %dms\n", sum, duration);
        // Parallel mode:
        final int THREAD_COUNT = 4;
        var portion = SIZE / THREAD_COUNT;
        start = System.currentTimeMillis();
        DotProduct.reset();
        var calculators = new DotProduct[THREAD_COUNT];
        int from = 0;
        for (int i = 0; i < THREAD_COUNT - 1; i++) {
            calculators[i] = new DotProduct(a, b, from, from + portion);
            from += portion;
        }
       // handle the rest
        calculators[THREAD_COUNT - 1] = new DotProduct(a, b, from, SIZE);
        DotProduct.joinAll();
        var durationPar = System.currentTimeMillis() - start;
        out.printf("Parallel mode\n\tresult: %f, time: %dms, speedup:
%f", DotProduct.getResult(), durationPar, (float) duration / durationPar);
    }
}
```

У лістингу вказано код двох файлів з класами: `DotProduct` — клас для паралельного обчислення скалярного добутку та `Main` — клас для тестування ефективності паралельної реалізації. При цьому основна задача ділилися на `THREAD_COUNT` підзадач приблизно однакової розмірності. Для синхронізації у методі `run` застосовувався монітор, описаний з використанням оператора `synchronized`. Для того, щоб гарантувати коректне завершення обчислень, використовувався статичний метод `joinAll` класу `DotProduct`, у якому активні потоки зберігалися у статичному списку `calculators`.

9.1.7. Комунікація між потоками

У багатьох задачах блокування ресурсів є недостатнім. Часто вимагається забезпечення можливості комунікації між потоками.

Використання потоків дає змогу уникнути опитування, яке раніше використовувалося для перевірки виконання умов і організовувалося у вигляді циклу.

Механізм міжпотокових комунікацій Java реалізований з використанням фінальних методів `wait( )`, `notify()` та `notifyAll( )` класу `Object`. Ці методи *доступні в усіх класах*, *можуть викликатися лише у синхронізованому контексті* і мають наступні властивості:

- Метод `wait()` змушує викликаючий потік віддати монітор і призупинити виконання до тих пір, поки який-небудь інший потік не увійде у той самий монітор та не викличе метод `notify()`.
- Метод `notify()` відновлює роботу потоку, який викликав метод `wait()` *того самого об'єкта*.
- Метод `notifyAll()` відновлює роботу усіх потоків, які викликали метод `wait()` того самого об'єкта. Одному з потоків надається доступ до об'єкта.

Додаткові форми методу `wait()` дозволяють вказувати час очікування.

Приклад 9.2. Задача «виробник-споживач».

Ця задача розглядалася у підрозділі 5.6. Нехай в програмі використовуються класи `Q` — черга, `Producer` — об'єкт-потік, який додає елементи до черги, `Consumer` — об'єкт потік, який вибирає елементи з черги. Розглянемо спочатку програму без використання між потокових комунікацій.

```
class Q{
    int n;
    synchronized int get(){
        System.out.println(n + " is received");
        return n;
    }
    synchronized void put(int n){
        this.n = n;
        System.out.println(n + " is put");
    }
}
```

```
class Producer implements Runnable{
    Q q;
    public Producer(Q q){
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while (true)
            q.put(i++);
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        while (true)
            q.get();
    }
}
public class MyClass {
    public static void main(String[] args){
        Q q = new Q();
        Producer producer = new Producer(q);
        Consumer consumer = new Consumer(q);
        System.out.println("Press Ctrl+C to stop");
    }
}
```

Не дивлячись не те, що методи `put()` та `get()` синхронізовані, програма працює неправильно, оскільки споживач може отримати той самий елемент (у даному випадку — ціле число) кілька разів, а попередні елементи — жодного разу. Можливий результат виконання програми наведений нижче:

```
1 is put
2 is put
2 is received
2 is received
3 is put
4 is put
4 is received
```

Правильне функціонування програми можна досягти модифікувавши клас `Q` з використанням методів `wait()` та `notify()` для передачі повідомлень в обох напрямках:

```
class Q{
    int n;
    boolean valueSet = false;
    synchronized int get(){
        while (!valueSet)
            try {
                wait();
            }
            catch (InterruptedException e){
                System.out.println("Interrupted");
            }
        System.out.println(n + " is received");
        valueSet = false;
        notify();
        return n;
    }
    synchronized void put(int n){
        while (valueSet)
            try {
                wait();
            }
            catch (InterruptedException e){
                System.out.println("Interrupted");
            }
        this.n = n;
        valueSet = true;
        System.out.println(n + " is put");
        notify();
    }

}
```

Всередині методу `get()` викликається метод `wait()`. Це призупиняє роботу потоку поки об'єкт класу `Producer` повідомить про те, що дані прочитані.

9.2. Паралельне програмування засобами .NET Framework

Потокова модель .NET Framework має багато спільного із потоковою системою Java. Для обробки потоків потрібно використовувати засоби простору імен `System.Threading` [17].

Для спрощення у фрагментах програм, написаних на C#, ми будемо використовувати код верхнього рівня (Top-level statements), який є доступним, починаючи з C# 9 [21].

Приклад 9.3. Розглянемо передачу параметрів у методи, які виконуються у потоках та використання фонових потоків.

```
using System;
using System.Threading;

var t1 = new Thread(ThreadProc);
var t2 = new Thread(ThreadProc);
t2.IsBackground = true;
t1.Start(("First thread", 6, 2000));
t2.Start(("Second thread", 20, 1000));
Console.WriteLine("Input any key to exit");
Console.ReadKey();

void ThreadProc(Object param)
{
    if (param is (string message, int count, int delay))
        for (int i = 0; i < count; i++)
        {
            Console.WriteLine($"{message} output #{i + 1}");
            Thread.Sleep(delay);
        }
}
```

У C# наявна можливість передавати у потік не об’єкти класів, як ми робили це Java-коді, а окремі методи, які можуть бути екземплярами одного з типів-делегатів:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object? obj);
```

Для того, щоб передати параметр у метод, який виконується у потоці, використовується параметризована версія методу `Start` класу `Thread`. У попередньому прикладі ми передавали 3 параметри у `ThreadProc`. Для цього вони були об’єднані у кортеж, до якого потім застосовувалася операція `is` для перевірки типу та декомпозиції на складові. Шляхом встановлення властивості `IsBackground` потік `t2` був пертворений у *фоновий*. Тому виконання програми завершується після того, як завершиться потік `t1` та користувач натисне довільну клавішу, не зважаючи на те, продовжується виконання потоку `t2 чи ні`. Зауважимо, що на відміну від Java дочірні потоки, які породжуються у .NET головним потоком, не є фоновими. Тому виконання `t1` до кінця є гарантованим.

9.2.1. Клас `Task`

Починаючи від .NET Framework 4, рекомендується проводити розробку паралельних програм з використанням засобів Task Parallel Library (`TPL`) — бібліотеки розпаралелювання задач, яка є набором типів та API, зосереджених у просторах імен `System.Threading` та `System.Threading.Tasks`. Застосовуючи `TPL`, паралелізм в програму можна втілити двома основними способами: паралелізмом даних та паралелізмом задач [17].

В основу `TPL` покладено клас `Task`. Елементарна одиниця виконання інкапсулюється в `TPL` засобами класу `Task`, а не `Thread`. Клас `Task` відрізняється від класу `Thread` тим, що він є абстракцією, що відповідає асинхронній операції. А в класі `Thread` інкапсулюється потік виконання. На системному рівні потік як і раніше залишається елементарною одиницею виконання, яку можна планувати засобами операційної системи. Але відповідність екземпляра об'єкта класу `Task` і потоку виконання не обов'язково є взаємно-однозначною.

Для створення нових задач в класі `Task` передбачено кілька конструкторів. Ось найпростіший з них:

```
public Task (Action action)
```

де `action` позначає точку входу в код, який буде виконуватися у задачі, `Action` — делегат, визначений у просторі імен `System`:

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

Таким чином, точкою входу повинен бути метод, що не приймає ніяких параметрів і не повертає ніяких значень.

Дуже часто виникає потреба передавати параметри у методи, які запускаються у задачах. Для підтримки цього використовується наступна форму конструктора:

```
public Task (Action<object?> action, object? state)
```

Як тільки задача створена, її можна запустити на виконання, викликавши її метод `Start()`. Нижче наведена одна з його форм:

```
public void Start()
```

Слід мати на увазі, що за замовчуванням завдання виконується в фоновому потоці. Отже, при завершенні створює потоку завершується і сама задача. Тому обов'язково необхідно переконатися, що задача завершила своє виконання. Це робиться за допомогою методу `Wait()`, який очікує завершення виконання задачі, на якій він викликається:

```
public void Wait()
```

Є також перезавантаження цього методу, у яких можна вказувати максимальний час очікування:

```
public void Wait(int millisecondsTimeout)
```

Ще однією перевагою класу `Task` над `Thread` є можливість отримувати назовні результат із методів, які запускаються у задачах. Для цього використовується параметризована версія класу: `Task<TResult>`, де `TResult` — тип результату, який повертається. Їй відповідає конструктор:

```
public Task (Func<TResult> function)
```

де `Func<TResult>` визначається наступним чином:

```
public delegate TResult Func<out TResult>()
```

Після завершення виконання задачі це значення можна отримати за допомогою властивості `Result` об'єкта задачі.

Приклад 9.4. Розглянемо паралельний запуск метода, який обчислює суму елементів свого параметра-масиву.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

var array1 = new double[100];
var array2 = new double[100];
var random = new Random();
for (int i = 0; i < array1.Length; i++)
{
    array1[i] = random.NextDouble();
    array2[i] = random.NextDouble();
}
var tasks = new List<Task<double>>()
{
    new(GetSum, array1),
    new(GetSum, array2),
    new(GetSum, null)
};

foreach (var task in tasks)
    task.Start();
foreach (var task in tasks)
    task.Wait();

for (int i = 0; i < tasks.Count; i++)
    Console.Write($"Task #{i} result: {tasks[i].Result}\n");

double GetSum(Object param)
{
    var sum = double.NaN;
```

```
        if (param is double[] array)
        {
            sum = array.Sum();
            Thread.Sleep(1000);
        }
        return sum;
    }
```

У програмі створюється список задач `tasks` для запуску методу `GetSum`, кожна з яких запускається у циклі `foreach`. Другий цикл `foreach` використовується для очікування завершення усіх задач.

Розглянемо деякі додаткові можливості класу `Task`. Статичний метод `Run`:

```
public static Run (Func<Task?> function)
```

дає змогу одночасно створювати та запускати задачу. Наприклад, створити та запустити той самий список задач можна було таким чином:

```
var tasks = new List<Task<double>>
{
    Task<double>.Run(() => GetSum(array1)),
    Task<double>.Run(() => GetSum(array2)),
    Task<double>.Run(() => GetSum(null))
};
```

Статичний метод

```
public static void WaitAll(params Task[] tasks)
```

дає змогу дочекатися завершення виконання усіх задач, які у нього передаються. Наприклад, дочекатися завершення `tasks[0]` та `tasks[2]` можна так:

```
Task.WaitAll(tasks[0], tasks[2]);
```

Якщо потрібно дочекатися завершення усіх задач колекції, то зручніше використати наступну модифікацію:

```
Task.WaitAll(tasks.ToArray());
```

9.2.2. М'ютекс

Нагадаємо, що м'ютекс — це взаємно виключний синхронізуючий об'єкт, призначений для того, щоб гарантувати, що у кожний момент часу спільний ресурс буде використовуватися ексклюзивно (тільки одним потоком).

М'ютекс реалізований у класі `System.Threading.Mutex`. Він має кілька конструкторів. Нижче наведено два найбільш уживані:

```
public Mutex()
public Mutex(bool initiallyOwned)
```

Перший створює м'ютекс, яким ніхто не володіє. При виклику другого із параметром `true` м'ютексом заволодіває потік, у якому викликається конструктор.

Для того, щоб отримати м'ютекс, в коді програми треба викликати метод `WaitOne()` для цього м'ютекса. Метод `WaitOne()` успадковується класом `Mutex` від класу `Thread.WaitHandle`. Нижче наведена його найпростіша форма:

```
public bool WaitOne()
```

Метод `WaitOne()` очікує до тих пір, поки не буде отриманий м'ютекс, для якого він був викликаний. Тобто, цей метод блокує виконання викликаючого потоку, поки не стане доступним вказаний м'ютекс. Він завжди повертає значення `true`. М'ютекс звільняється викликом методу `ReleaseMutex()`:

```
public void ReleaseMutex()
```

Це дає змогу іншим потокам отримати даний м'ютекс. Якщо потік вже отримав і ще не звільнив м'ютекс і повторно викликає метод `WaitOne()`, то він не заблоковується.

Схема використання м'ютекса:

```
Mutex myMutex = new Mutex();
// .......................
myMutex.WaitOne(); // чекати отримання м'ютекса
// Отримати доступ до ресурса
myMutex.ReleaseMutex(); // звільнити м'ютекс
```

9.2.3. Бар'єр

Клас `System.Threading.Barrier` дає змогу кільком задачам паралельно працювати із алгоритмом, використовуючи кілька фаз. Кожний учасник виконується, поки його код не досягне точки бар'єра. Бар'єр означає закінчення одного етапу роботи. Коли учасник досягає бар'єра, його виконання блокується, поки усі учасники не досягнуть цього самого бар'єра. Коли всі учасники досягнуть бар'єра, при необхідності можна почати наступний етап. Конструктори:

```
Barrier(Int32)
Barrier(Int32, Action<Barrier>)
```

Числовий параметр — кількість учасників, які мають зібратися біля бар'єра. Додатковий параметр — делегат, який буде викликатися після кожної фази.

У роботі з бар'єром використовується метод

```
SignalAndWait()
```

Він повідомляє, що учасник досяг бар'єра та очікує досягнення бар'єра іншими учасниками. Властивість `CurrentPhaseNumber` вказує номер поточної фази бар'єра.

Приклад 9.5. Потоки `t1` та `t2` випадковим чином переставляють слова у масиві `words1` та `words2`, відповідно. Обчислити кількість кроків, потрібних для того, щоб конкатенація слів у масивах дала потрібну фразу `solution`.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

var words1 = new [] {"brown", "jumped", "the", "fox"};
var words2 = new [] {"dog", "lazy", "the", "over"};
var solution = "the brown fox jumped over the lazy dog.";

bool success = false;

var barrier = new Barrier(2, (b) =>
{
    var sb = new StringBuilder();
    sb.AppendJoin(' ', words1);
    sb.Append(' ');
    sb.AppendJoin(' ', words2);
    sb.Append('.');

    Console.CursorLeft = 0;
    Console.Write($"Current phase: { b.CurrentPhaseNumber }");
    if (String.CompareOrdinal(solution, sb.ToString()) == 0)
    {
        success = true;
        Console.Write($"\n{b.CurrentPhaseNumber} attempts needed");
    }
});

var task1 = Task.Run(() => Solve(words1));
var task2 = Task.Run(() => Solve(words2));
Task.WaitAll(task1, task2);

// Both arrays would be solved in the same phase.
void Solve(IList<string> words)
{
    while (success == false)
    {
        var random = new Random();
        for (int i = words.Count - 1; i > 0; i--)
        {
            int Index = random.Next(i + 1);
            (words[i], words[Index]) = (words[Index], words[i]);
        }
        // Stop here to examine results of all tasks activity.
```

```
        // Post-phase delegate in the Barrier constructor is used.
        barrier.SignalAndWait();
    }
}
```

9.2.4. Семафор

Семафор надає одночасний доступ до спільного ресурсу не одному, а кільком потокам. Семафор керує доступом до загального ресурсу, використовуючи лічильник. Якщо значення лічильника більше за нуль, то доступ до ресурсу дозволений. Якщо значення рівне нулю, то доступ заборонений.

Семафор реалізований класом `System.Threading.Semaphore`. Найпростіша форма конструктора класу:

```
public Semaphore(int initialCount, int maximumCount)
```

де `initialCount` — це початкове значення лічильника семафора, тобто кількість початкових дозволів; `maximumCount` — максимальне можливе значення лічильника, тобто максимальна кількість дозволів, які може дати семафор.

Семафор використовується таким чином, як і м’ютекс. Для отримання доступу до ресурсу в коді викликається метод `WaitOne()` для об’єкта семафора, який очікує до тих пір, поки не буде отриманий семафор, для якого він викликається. Він блокує виконання викликаючого потоку до тих пір, поки семафор не надасть дозвіл на доступ до ресурсу. Семафор звільняється викликом методу `Release()`. Є дві форми цього методу:

```
public int Release()
public int Release(int releaseCount)
```

В першій формі `Release()` звільняє один дозвіл, а в другій — `releaseCount` дозволів. В обох формах метод повертає кількість дозволів, які існували до звільнення.

Якщо потік вже отримав доступ до семафора і ще не звільнив свій доступ, а потім повторно викликає метод `WaitOne()`, то на відміну від м’ютекса він може і не отримати доступ (якщо всі дозволи вже вичерпані).

9.2.5. Монітор

У *статичному* класі `Monitor`, який знаходиться у просторі імен `System.Threading` визначено ряд методів для керування синхронізацією. Наприклад, для отримання блокування об’єкта викликається метод `Enter()`, а для зняття блокування — метод `Exit()`. Нижче наведені загальні форми цих методів:

```
public static void Enter(object obj)
public static void Exit(object obj)
```

де obj позначає синхронізований об'єкт. Якщо ж об’єкт недоступний, то після виклику методу Enter() викликаючий потік очікує до тих пір, поки об’єкт не стане доступним (на практиці методи Enter() та Exit() застосовуються рідко, оскільки оператор lock автоматично забезпечує еквівалентні засоби синхронізації потоків).

В класі Monitor описані кілька форм методу TryEnter():

```
public static bool TryEnter(object obj)
public static bool TryEnter(object obj, int milliseconds)
```

Цей метод повертає true, якщо викликаючий потік отримує блокування для об’єкта obj (відразу, або на протязі часу, вказаного за допомогою параметру timeout), а інакше вони повертають false. За допомогою методу TryEnter() можна реалізувати альтернативний варіант синхронізації потоків, якщо необхідний об’єкт тимчасово недоступний. Розглянемо приклад:

```
var lockObj = new Object();
int timeout = 500;
if (Monitor.TryEnter(lockObj, timeout)) {
   try {
      // The critical section.
   }
   finally {
      // Ensure that the lock is released.
      Monitor.Exit(lockObj);
   }
}
else {
   // The lock was not acquired.
}
```

В класі Monitor визначені методи Wait(), Pulse() та PulseAll(), які дають змогу перевести / вивести потоки із стану очікування та забезпечують комунікацію між потоками. Ці методи можуть викликатися тільки з заблокованого фрагмента блоку (після входу у монітор). Вони застосовуються наступним чином. Коли виконання потоку має призупинитися до виконання певної умови, він викликає метод Wait(). В результаті потік переходить в стан очікування, а *блокування з відповідного об’єкта знімається*, що дає можливість використовувати цей об’єкт в іншому потоці. Надалі очікуючи потік активується, коли інший потік увійде в аналогічний стан блокування, і викликає метод Pulse() або PulseAll(). При виклику методу Pulse() поновлюється виконання першого потоку, що очікує своєї черги на отримання блокування. А виклик методу

`PulseAll()` сигналізує про зняття блокування всім очікуючим потокам. Нижче наведено дві найбільш вживані форми методу `Wait()`:

```
public static bool Wait(object obj)
public static bool Wait(object obj, int milliseconds)
```

У першій формі очікування триває аж до повідомлення про звільнення об'єкта, а в другій — як до повідомлення про звільнення об'єкта, так і до закінчення періоду часу, на який вказує кількість мілісекунд. В обох формах `obj` позначає об'єкт, звільнення якого очікується.

Нижче наведені загальні форми методів `Pulse()` і `PulseAll()`:

```
public static void Pulse(object obj)
public static void PulseAll(object obj)
```

Якщо методи `Wait()`, `Pulse()` і `PulseAll()` *викликаються з коду, що знаходиться за межами синхронізованого коду, то генерується* `SynchronizationLockException`!

Слід зазначити, що засіб для блокування вбудовано в мову C#. Завдяки цьому всі об'єкти можуть бути синхронізовані. Синхронізація організовується за допомогою ключового слова `lock`. Нижче наведена загальна форма блокування:

```
lock(lockObj)
{
    // оператори, що синхронізуються
}
```

де `lockObj` позначає посилання на синхронізований об'єкт (якщо потрібно синхронізувати тільки один оператор, то фігурні дужки не потрібні). Оператор `lock` є замінником пари `Monitor.Enter(lockObj)` та `Monitor.Exit(lockObj)`. Блокується той об'єкт, який є синхронізованим ресурсом. Треба мати на увазі, що об'єкт, що блокується не має бути загальнодоступним, бо у протилежному випадку він може бути заблокований з іншого, неконтрольованого в програмі фрагмента коду і надалі взагалі не розблокується. У минулому для блокування об'єктів дуже часто застосовувалася конструкція `lock(this)`. Але вона придатна тільки в тому випадку, якщо `this` є посиланням на закритий об'єкт. У зв'язку з можливими програмними і концептуальними помилками, до яких може привести конструкція `lock(this)`, застосовувати її більше не рекомендується. Замість неї краще створити закритий об'єкт, щоб потім заблокувати його.

Приклад 9.6. Демонстрація синхронізації на прикладі керування доступом до методу `SumIt()`, який обчислює суму елементів цілочисельного масиву.

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

var array = Enumerable.Range(1, 5).ToArray();
var sa = new SumArray();
var taskCount = 10;
var tasks = new Task[taskCount];
for (int i = 0; i < taskCount; i++)
{
    tasks[i] = Task.Run(() => sa.SumIt(array));
}
Task.WaitAll(tasks);

class SumArray
{
    int sum;
    readonly object _lockOn = new (); // object to lock
    public int SumIt(int[] nums)
    {
        lock (_lockOn) // lock entire method
        {
            sum = 0;
            for (int i = 0; i < nums.Length; i++)
            {
                sum += nums[i];
                var id = Task.CurrentId;
                Console.Write($"Current sum for task {id}: {sum}\n");
                Task.Delay(10).Wait(10); // allow the switch
            }
            return _sum;
        }
    }
}
```

Незважаючи на всю простоту і ефективність блокування коду методу, як показано в наведеному вище прикладі, такий засіб синхронізації виявляється придатним далеко не завжди. Припустимо, що потрібно синхронізувати доступ до методу класу, який був створений кимось іншим і сам не синхронізований. Вихідний код класу недоступний. Як же тоді синхронізувати об'єкт такого класу? На щастя, ця проблема розв'язується досить просто: доступ до об'єкта може бути заблокований з зовнішнього коду по відношенню до даного об'єкта, для чого достатньо вказати цей об'єкт в операторі `lock`. Нижче наведений відповідний рядок коду, в якому здійснюється подібне блокування :

```
lock(sa) {answer = sa.SumIt(a);}
```

Об'єкт `sa` має бути закритим для успішного блокування.

9.2.6. Застосування подій

Для синхронізації в С # передбачений ще один тип об'єкта: подія. Існують два різновиди подій: встановлюються в початковий стан вручну і автоматично. Вони підтримуються в класах `ManualResetEvent` та `AutoResetEvent` відповідно. Ці класи є похідними від класу `EventWaitHandle`, що знаходиться на верхньому рівні ієрархії класів, і застосовуються в тих випадках, коли один потік чекає появи деякої події в іншому потоці. Як тільки така подія настає, другий потік повідомляє про нього перший потік, дозволяючи тим самим відновити його виконання. Нижче наведені конструктори класів:

```
public ManualResetEvent(bool initialState)
public AutoResetEvent(bool initialState)
```

Якщо в обох формах параметр `initialState` має значення `true`, то про подію спочатку повідомляється. А якщо він має логічне значення `false`, то про подію спочатку не буде повідомлено.

Використання подій дуже просте. Так, для події типу `ManualResetEvent` порядок застосування наступний. Потік, що очікує деяку подію, викликає метод `WaitOne()` для об'єкта, що пов'язаний із цією подією. Якщо об'єкт події знаходиться в *сигнальному стані*, то відбувається негайне повернення з методу `WaitOne()`. В іншому випадку виконання викликаючого потоку призупиняється доти, поки не буде отримано повідомлення про подію. Як тільки подія відбудеться в іншому потоці, цей потік *встановить* об'єкт події в сигнальний стан, викликавши метод `Set()`. Метод `Set()` треба розглядати як такий, що повідомляє про те, що подія відбулася. Після встановлення об'єкта події в сигнальний стан відбувається негайне повернення з методу `WaitOne()`, і перший потік продовжує своє виконання. А в результаті виклику методу `Reset()` об'єкт події повертається в несигнальний стан.

Подія типу `AutoResetEvent` відрізняється від події типу `Manual-ResetEvent` лише способом переведення у початковий стан. Якщо для події типу `ManualResetEvent` об'єкт події залишається в сигнальному стані доти, поки не буде викликаний метод `Reset()`, то для події типу `AutoResetEvent` об'єкт автоматично переходить в несигнальний стан, як тільки потік, що очікує цю подію, отримає повідомлення про нього і відновить своє виконання. Тому, якщо застосовується подія типу `AutoResetEvent`, то викликати метод `Reset()` необов'язково.

Приклад 9.7. Застосування подій.

```
// Потік повідомляє, що подія передана його конструктору
class MyThread {
    public Thread Thrd;
    ManualResetEvent mre;
    public MyThread(string name, ManualResetEvent evt) {
        Thrd = new Thread(this.Run);
        Thrd.Name = name;
        mre = evt;
        Thrd.Start();
    }
    // Точка входу в потік
    void Run(){
        Console.WriteLine("Всередині потоку" + Thrd.Name);
        for(int i = 0; i < 5; i ++) {
            Console.WriteLine(Thrd.Name);
            Thread.Sleep(500);
        }
        Console.WriteLine(Thrd.Name + "завершено!");
        mre.Set(); // Повідомити про подію
    }
}
class ManualEventDemo {
    static void Main () {
        var evtObj = new ManualResetEvent(false);
        var mtl = new MyThread("Потік події 1", evtObj);
        Console.WriteLine("Основний потік очікує подію.");
        // Очікувати повідомлення про подію.
        evtObj.WaitOne();
        Console.WriteLine("Основний потік отримав" +
            "Повідомлення про подію від першого потоку.");
        // Установити об'єкт події в початковий стан.
        evtObj.Reset();
        mtl = new MyThread("Потік події 2", evtObj);
        // Очікувати повідомлення про подію.
        evtObj.WaitOne();
        Console.WriteLine("Основний потік отримав" +
            "Повідомлення про подію від другого потоку.");
    }
}
```

Подія передається безпосередньо конструктору класу `MyThread`. Коли завершується метод `Run()` класу `MyThread`, він викликає для об'єкта події метод

`Set()`, встановлює цей об'єкт в сигнальний стан. Якби не об'єкт події, то всі потоки виконувалися б одночасно, а результати їх виконання виявилися б заплутаними. Якщо об'єкт `AutoResetEvent` використовувався би замість об'єкта типу `ManualResetEvent`, то викликати метод `Reset()` в методі `Main()` не довелося б.

9.2.7. Клас `Interlocked`

Клас `Interlocked` є альтернативою до інших засобів синхронізації у випадках, коли потрібно тільки змінити значення змінної, спільної для кількох потоків. Методи, доступні в класі `Interlocked`, гарантують, що їх дія буде виконуватися як єдина, неперервна операція. Це означає, що ніякої синхронізації в даному випадку взагалі не вимагається. У класі `Interlocked` надаються статичні методи для додавання двох цілих значень, інкремента та декремента цілих чисел, порівняння і встановлення значень об'єкта, обміну об'єктами і отримання 64-розрядного значення. Всі ці операції виконуються неподільно.
У наведеному нижче прикладі демонструється застосування методів `Add()` та `Increment()`:

```
public static int Increment(ref int location)
public static int Add(ref int location, int value)
```

де `location` — це відповідна змінна, яка змінюється. Ці методи повертають змінене значення.

Приклад 9.8. Визначити, як багато потрібно згенерувати випадкових чисел від 0 до 1000 (включно), для того, щоб отримати 50000 значень, які рівні середині діапазону (тобто, 500). Три задачі запускаються для розпаралелення обчислень. Метод `Increment` використовується для коректного оновлення лічильника `midpointCount`. Монітор використовується для захисту генератора випадкових чисел.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

const int LOWER_BOUND = 0;
const int UPPER_BOUND = 1001;

var lockObj = new Object();
var rnd = new Random();
var totalCount = 0;
var totalMidpoint = 0;
var midpointCount = 0;
```

```
var tasks = new List<Task>();
// Start three tasks.
for (int ctr = 0; ctr <= 2; ctr++)
    tasks.Add(Task.Run(() =>
    {
        int midpoint = (UPPER_BOUND - LOWER_BOUND) / 2;
        int value = 0;
        int total = 0;
        int midpt = 0;

        do
        {
            lock (lockObj)
            {
                value = rnd.Next(LOWER_BOUND, UPPER_BOUND);
            }
            if (value == midpoint)
            {
                Interlocked.Increment(ref midpointCount);
                midpt++;
            }
            total++;
        } while (Volatile.Read(ref midpointCount) < 50000);

        Interlocked.Add(ref totalCount, total);
        Interlocked.Add(ref totalMidpoint, midpt);

        var s = String.Format("Task {0}:\n", Task.CurrentId) +
                String.Format("Random Numbers: {0:N0}\n", total)+
                String.Format("Midpoint values: {0:N0} ({1:P3})", midpt,
                ((double) midpt) / total);
        Console.WriteLine(s);
    }));
Task.WaitAll(tasks.ToArray());
Console.WriteLine();
Console.WriteLine("Total midpoint values:  {0,10:N0} ({1:P3})",
    totalMidpoint, totalMidpoint / ((double) totalCount));
Console.WriteLine("Total number of values: {0,10:N0}", totalCount);
```

Приклад виводу:

```
Task 1:
   Random Numbers: 16,989,455
   Midpoint values: 17,022 (0.100%)
Task 3:
```

```
      Random Numbers: 16,859,438
      Midpoint values: 16,792 (0.100%)
   Task 2:
      Random Numbers: 16,052,344
      Midpoint values: 16,186 (0.101%)

   Total midpoint values:      50,000 (0.100%)
   Total number of values: 49,901,237
```

9.2.8. Асинхронні делегати

Делегати можуть працювати у асинхронному режимі, тобто виконуватися у окремому потоці шляхом виклику його методу BeginInvoke, який повертає значення типу IAsyncResult. Це значення можна використовувати для очікування завершення делегата та знаходження результату виклику методу, який викликається асинхронно. Для досягнення цього потрібно передати відповідне значення у якості єдиного параметру методу EndInvoke.

Приклад 9.9. Асинхронний виклик методу і передача параметрів у нього.

```
using System;
using System.Threading;

// Print out the ID of the executing thread.
var id = Thread.CurrentThread.ManagedThreadId;
Console.WriteLine($"Main() invoked on thread {id}");
// Invoke Add() on a secondary thread.
BinaryOp b = Add;
var asyncResult = b.BeginInvoke(2, 3, null, null);
// Do other work on primary thread...
Console.WriteLine("Doing more work in Main()!");
// Obtain the result of the Add() method when ready.
var answer = b.EndInvoke(asyncResult);
Console.WriteLine($"2 + 3 is {answer}.");
Console.WriteLine("Press any key");
Console.ReadKey();

int Add(int x, int y)
{
    Console.WriteLine("Add() invoked on thread {0}.",
        Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(5000);
    return x + y;
}

public delegate int BinaryOp(int x, int y);
```

Розглянемо приклад обчислення функції факторіал у окремому потоці:

```
using System;
using System.Threading;

var watch = new Stopwatch();
watch.Start();
Func<int, int> func = SleepingFactorial;

var asyncResult = func.BeginInvoke(10, CallBack, "10!");
Console.WriteLine("Calculations are proceeding...");
Console.WriteLine($"Result: {func.EndInvoke(asyncResult)}");
watch.Stop();
Console.WriteLine($"Call duration: {watch.Elapsed.Seconds}s");

static void CallBack(IAsyncResult asyncResult) =>
    Console.WriteLine($"{asyncResult.AsyncState} finished");

static int SleepingFactorial(int number)
{
    int fact = 1;
    for (int i = 2; i < number; i++)
    {
        fact *= i;
        Thread.Sleep(500);
    }
    return fact;
}
```

Вивід:

```
Calculations are proceeding...
Result: 362880
10! finished
Call duration: 4s
```

9.2.9. Багатозадачність з використанням процесів

При програмуванні на C# можна використовувати і багатозадачність на основі процесів. У цьому випадку замість запуску іншого потоку в одній і тій же програмі одна програма починає виконання іншої. При програмуванні на C# це робиться за допомогою класу `Process`, визначеного в просторі `System.Diagnostics`. Для запуску процесів можна скористатися методом `Start()` класу `Process`:

```
public static Process Start(string fileName)
```

де `fileName` позначає конкретне ім’я файлу, який повинен виконуватися.

Коли створений процес завершується, слід викликати метод `Close()`, щоб звільнити пам’ять, виділену для цього процесу:

```
public void Close()
```

Процес може бути перерваний двома способами. Якщо процес є GUI-додатком Windows, то для переривання такого процесу викликається метод

```
public bool CloseMainWindow()
```

Цей метод посилає процесу повідомлення, що пропонує йому зупинитися. Він повертає логічне значення `true`, якщо повідомлення отримано, і логічне значення `false`, якщо програма не має графічного інтерфейсу або головного вікна. Метод `CloseMainWindow()` служить тільки для запиту зупинки процесу. Якщо додаток проігнорує такий запит, то він не буде перерваний.

Для безумовного переривання процесу треба викликати метод `Kill()`:

```
public void Kill()
```

Методом `Kill()` слід користуватися акуратно, так як він призводить до неконтрольованого переривання процесу. Будь-які незбережені дані, пов'язані з процесом, який переривається, будуть, швидше за все, втрачені.

Для того щоб організувати очікування завершення процесу, можна скористатися методом `WaitForExit()`. Нижче наведено дві його форми:

```
public void WaitForExit()
public bool WaitForExit(int milliseconds)
```

У першій формі очікування триває до тих пір, поки процес не завершиться, а в другій формі — тільки протягом зазначеного часу. В другому випадку `WaitForExit()` повертає `true`, якщо процес завершився, і `false`, якщо він все ще виконується.

Приклад 9.10. Створення, очікування і закриття процесу.

```
using System;
using System.Diagnostics;

var newProc = Process.Start("wordpad.exe");
Console.WriteLine("Новий процес запущений");
newProc.WaitForExit();
newProc.Close(); // звільнити виділені ресурси
Console.WriteLine("Новий процес завершено");
```

При виконанні цієї програми запускається WordPad, і на екрані з'являється повідомлення. Потім програма очікує закриття WordPad. Після закінчення роботи WordPad на екрані з'являється заключне повідомлення.

10. ТЕХНОЛОГІЯ OPENMP

10.1. Основні характеристики OpenMP

Одним з найбільш популярних засобів паралельного програмування, заснованих на традиційних мовах програмування, є технологія OpenMP [1–3]. За основу береться послідовна програма, а для створення її паралельної версії використовується набір директив, функцій та змінних середовища. При цьому паралельна програма буде коректно працювати на різних комп'ютерах з розподіленою пам'яттю, які підтримують OpenMP API.

Розробкою стандарту займається некомерційна організація OpenMP ARB (Architecture Review Board), в яку увійшли представники компаній-розробників SMP-архітектури та ПЗ. OpenMP підтримує роботу з мовами Fortran та C/C++.

OpenMP реалізує паралельні обчислення за допомогою багатопотоковості. Головний (*master*) потік створює набір «підпорядкованих» (*slave*) потоків, між якими розподіляється задача. Потоки виконуються паралельно на багатопроцесорній машині, причому кількість процесорів не обов'язково має бути більше або рівна за кількість потоків.

Компілятор з підтримкою OpenMP визначає макрос `_OPENMP`, який може використовуватися для умовної компіляції окремих блоків паралельної програми і визначений у форматі `yyyymm`, де `yyyy` та `mm` — цифри року та місяця прийняття стандарту OpenMP, який підтримується компілятором. Наприклад, для компілятора з підтримкою стандарту OpenMP 3.0 значення `_OPENMP` рівне `200805`. Приклад перевірки підтримки OpenMP у системі:

```
#include <stdio.h>
int main(){
    #ifdef _OPENMP
    printf("OpenMP is supported!\n");
    #endif
}
```

10.1.1. Модель паралельної програми

Розпаралелювання в OpenMP виконується за допомогою спеціальних директив, які додаються до тексту програми. Використовується SPMD-модель (Single Program Multiple Data), яка передбачає, що для усіх паралельних потоків використовується один і той самий код.

Програма починається з послідовної області — спочатку робочим є лише один потік, при вході в паралельну область породжуються інші потоки, між якими розподіляється виконання частини коду. Після завершення паралельної області всі потоки, крім головного, завершуються, і починається нова послідовна область і т. д. Паралельні області можуть бути вкладеними одна в одну. Для написання ефективної програми необхідно, щоб усі потоки, які беруть участь в обробці програми, були рівномірно завантажені. Важливим моментом є синхронізація доступу до спільних даних для уникнення конфліктів при одночасному доступі. Цьому призначена значна частина функціональності OpenMP.

10.1.2. Директиви та функції

Значна частина функціональності OpenMP реалізується за допомогою директив компілятора. Директиви OpenMP на мові С оформлюються вказівками препроцесора, які починаються з `#pragma omp`. Ключове слово `omp` використовується для того, щоб уникнути випадкового співпадання імен директив OpenMP з іншими іменами. Формат директиви:

```
#pragma omp directive-name [опція[[,] опція]...]
```

Усі директиви OpenMP можна поділити на 3 категорії:

- визначення паралельної області;
- розподіл роботи;
- синхронізація.

Кожна директива може мати кілька додаткових атрибутів (опцій).

Для використання функцій OpenMP в до програми треба підключити файл `omp.h`. Функції визначення параметрів мають пріоритет над відповідними змінними середовища.

В OpenMP передбачені функції роботи із системним таймером. Функція

```
double omp_get_wtime()
```

повертає астрономічний час у секундах, який пройшов з деякого моменту в минулому. Різниця значень функції показує тривалість роботи програми між двома викликами. Функція

```
double omp_get_wtick(void)
```

повертає роздільну здатність таймера у секундах (міру точності таймера).

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    double start_time, end_time, tick;
    start_time = omp_get_wtime();
    end_time = omp_get_wtime();
    tick = omp_get_wtick();
    printf("Duration %lf\n", end_time-start_time);
    printf("Precision %lf\n", tick);
}
```

10.2. Директива `parallel`

У момент запуску програми породжується єдиний головний потік, який починає виконання програми з першого оператора. Основний потік виконує усі послідовні області програми. При вході у паралельну область породжуються додаткові потоки.

Паралельна область задається за допомогою директиви `parallel` [2, 18]:

```
#pragma omp parallel [опція[[,] опція]...]
```

Можливі опції:

- `if(умова)` — виконання паралельної області за умовою. Якщо умова не виконується, то директива не спрацьовує та продовжується виконання програми в попередньому режимі;
- `num_threads(цілочисловий вираз)` — явна вказівка кількості потоків; за замовчуванням вибирається значення, встановлене за допомогою функції `omp_set_num_threads()`, або значення змінної `OMP_NUM_THREADS`;
- `private(список)` — задає список змінних, для яких створюються локальні копії в кожному потоці, *початкове значення яких не визначено*;
- `firstprivate(список)` — задає список змінних, для яких створюються локальні копії в кожному потоці; локальні копії ініціалізуються значеннями цих змінних у головному потоці;
- `shared(список)` — задає список змінних, спільних для усіх потоків;
- `reduction(оператор: список)` — задає оператор та список спільних змінних; для кожної змінної створюються локальні копії у кожному потоці, які ініціалізуються 0 — для адитивних операцій, 1 — для мультиплікативних); над локальними копіями змінних після виконання усіх паралельних ділянок виконується заданий оператор з переліку: `– +, *, –, &, |, ^, &&, ||`.

При вході в паралельну область породжуються нові `OMP_NUM_THREADS – 1` потоків, кожний з яких отримує свій унікальний номер, причому породжуючий потік отримує номер 0 і стає основним потоком групи («майстром»). Інші потоки отримують номери від 1 до `OMP_NUM_THREADS - 1`. Кількість потоків, які виконують дану паралельну область, залишається незмінним до моменту виходу із області. *При виході з паралельної області проводиться неявна синхронізація* та знищуються усі потоки, крім породжуючого. Директива `parallel` *не передбачає розподіл виконання коду паралельної області між потоками*. Фактично, якщо не використовуються додаткові директиви (`sections`, `single` або `for`), то код паралельної області виконується кожним потоком у повному обсязі.

Розглянемо приклад використання директиви `parallel`:

```
    printf("Serial scope 1\n");
    #pragma omp parallel num_threads(5)
    {
        printf("Parallel scope\n");
    }
    printf("Serial scope 2\n");
```

Кожний потік отримує локальну копію змінної `count` із значенням 0. Потім кожний потік збільшує значення власної копії `count` на 1 та його виводить. На виході з паралельної області додаються значення змінних `count` по усім потокам.

Перед запуском програми кількість потоків паралельній області можна задати, вказавши значення змінної середовища `OMP_NUM_THREADS`.

10.2.1. Функції керування кількістю потоків

Функція `omp_get_num_procs()` повертає кількість процесорів, доступних на момент виклику. У процесі функціонування програми можна дізнатися чи змінити кількість потоків за допомогою функцій, наведених у табл. 10.1.

Таблиця 10.1. Функції керування кількістю потоків

Функція	Опис
`void omp_set_num_threads(int num)`	Встановлює кількість потоків. Діє до кінця програми. Впливає на ті паралельні області, при описі яких не використано параметр `num_threads`
`int omp_get_num_threads()`	Повертає кількість потоків в активній області
`int omp_get_max_threads()`	Якщо кількість потоків попередньо задана функцією `omp_set_num_threads`, то повертається це число, інакше — кількість логічних процесорів
`int omp_get_thread_num()`	Повертає номер поточного потоку

Приклад 10.1. Демонстрація функції `omp_set_num_threads`.

```
omp_set_num_threads(2);
#pragma omp parallel num_threads(3)
{
    printf("Parallel scope 1\n");
}
#pragma omp parallel
{
    printf("Parallel scope 2\n");
}
```

Перше повідомлення виводиться трьома потоками, друге — двома.

Приклад 10.2. Використовуючи `num_threads` задати 8 потоків для паралельної області. Вивести номер поточного потоку та загальну кількість створених потоків і максимальну кількість потоків (тільки для головного потоку).

```
#pragma omp parallel num_threads(8)
{
    int num = omp_get_thread_num();
    printf("Current thread number is %d\n", num);
    if(num == 0){
        printf("Max threads: %d\n", omp_get_max_threads());
        printf("Threads num: %d\n", omp_get_num_threads());
    }
}
```

10.2.2. Директиви `single` та `master`

Якщо в паралельній області якась ділянка коду має бути виконана лише один раз, то використовується директива `single`.

```
#pragma omp single [опція [[,] опція]...]
```

Приклад 10.3. Використання директиви `single`.

```
#pragma omp parallel num_threads(3)
{
    int n = omp_get_thread_num();
    printf("First message from thread %d\n", n);
    #pragma omp single
        printf("Exclusive code only in thread %d\n", n);
    printf("Last message from thread %d\n", n);
}
```

Після виконання виділеної директивою `single` ділянки коду *відбувається неявна синхронізація паралельних потоків*: їх подальше виконання відбувається лише тоді, коли вони всі досягнуть даної точки коду. Опція `nowait` дає змогу потокам продовжити їх виконання без очікування інших. Наприклад,

```
int m = 0;
#pragma omp parallel num_threads(3)
{
    int n = omp_get_thread_num();
    # pragma omp single nowait
    {
        printf("Exclusive code only in thread %d\n", n);
        m = 2;
    }
    printf("Parallel code in thread %d, m = %d\n", n, m);
}
```

Директива `master` виділяє фрагмент коду, який має виконуватися тільки головним потоком. Інші потоки просто пропускають цю ділянку. Ця директива *не припускає неявної синхронізації*. Наприклад,

```
int n;
#pragma omp parallel num_threads(3) private(n)
{
    n = 1;
    #pragma omp master
        n = 2;
    printf("First value of n: %d\n", n);
    #pragma omp master
        n = 3;
    printf("Second value of n: %d\n", n);
}
```

10.3. Модель даних OpenMP

Модель даних в OpenMP передбачає наявність як загальної спільної для усіх потоків ділянки пам'яті, так і локальної пам'яті для кожного потоку. В OpenMP змінні в паралельних областях програми поділяються на два основні класи:

- `shared` (спільні — загальнодоступні для усіх потоків)
- `private` (локальні — кожний потік бачить свій екземпляр змінної).

Спільна змінна завжди існує в одному екземплярі. Оголошення локальної змінної викликає породження окремого екземпляру даної змінної для кожного потоку, зміна значень якого у потоці ніяк не впливає на екземпляри у інших потоках.

Якщо кілька потоків одночасно записують значення спільної змінної без виконання синхронізації або якщо один потік читає значення змінної, а інший — записує (без синхронізації), то виникає ситуація «гонитви даних» (*data race*), при якій результат виконання програми непередбачуваний.

За замовчуванням, *усі змінні, породжені за межами паралельної області, при вході в цю область вважаються спільними* (`shared`), а *змінні, визначені всередині паралельної області — локальними* (`private`). Режим доступу також можна явно вказати з використанням опцій `private`, `shared`, `firstprivate` в директивах OpenMP.

Приклад 10.4. Використання `firstprivate`.

```
int n = 10;
printf("n in serial scope (begin): %d\n", n);
```

```
#pragma omp parallel firstprivate(n), num_threads(5)
{
    printf("Value of n (enter): %d\n", n);
    n = omp_get_thread_num();
    printf("Value of n (exit): %d\n", n);
}
printf("n in serial scope (end): %d\n", n);
```

10.4. Паралельні цикли

Якщо в паралельній області зустрічається оператор циклу, то, згідно до загального правила, він буде виконуватися окремо усіма потоками. Для розподілу ітерацій циклу між різними потоками використовують директиву

```
#pragma omp for [опція [[,] опція]...]
```

Формат паралельних циклів на мові C/C++:

```
for(i = інваріант циклу; i {<,>,=,<=,>=} інваріант; i {+,-}= інваріант)
```

Ці вимоги потрібні для того, щоб у OpenMP можна було при вході в цикл точно визначити кількість ітерацій. Локальна ітеративна змінна *обов'язково має бути цілочислового типу* і *не має змінюватися у тілі циклу*. У тілі циклу *не можна використовувати інструкції* `break` або `continue`.

Якщо директива паралельного виконання стоїть перед вкладеними циклами, то вона діє тільки на самий зовнішній цикл.

Приклад 10.5. Паралельне обчислення поелементної суми двох векторів.

```
int a[10], b[10], c[10];
for(int i = 0; i < 10; i++)
{
    a[i] = i;
    b[i] = i*i+1;
}
#pragma omp parallel num_threads(5)
{
    int n = omp_get_thread_num();
    #pragma omp for
    for(int i = 0; i < 10; i++)
    {
        c[i] = a[i] + b[i];
        printf("c[i] = %d is from thread %d\n",c[i],n);
    }
}
```

Якщо паралельна секція починається із циклу, то можна використовувати скорочену форму опису:

```
#pragma omp parallel for [опція [[,] опція]...]
```

Розпаралелювання допускається лише для циклів `for`. *Цикли* `while, do` *не розпаралелюються*.

Крім того, у циклі *не має бути залежностей, коли чергова ітерація залежить від попередньої*, тобто не має бути виразів вигляду $a[i+1] = f(a[i])$.

Приклад 10.6. Обчислення суми елементів масиву з використанням каскадного алгоритму (схеми здвоєння).

```
const int N = 100;
int a[N];
for(int i = 0; i < N; i++)
    a[i] = i+1;
int j = 0; // j is difference between summand
for(j = 1; j <= N/2; j *= 2){
    #pragma omp parallel for
    for(int i = 0; i < N-j; i += 2*j)
        a[i] += a[i+j];
}
if(j != N/2 && j < N)
    a[0] += a[j];
cout << "Sum is " << a[0] << '\n';
```

10.4.1. Накопичення значень

В багатьох обчислювальних циклічних алгоритмах необхідно рахувати значення сум, добутків або інших агрегатних функцій. Для того, щоб реалізувати накопичення, в циклах в OpenMP використовується опція `reduction(оператор: список)`. Допустимі операції вказані в пункті 10.1.2. При використані опції `reduction` кожний потік фактично отримує свою копію даних та виконує необхідну операцію над своїми даними (без блокування). Після виконання циклу (чи паралельного блоку іншого типу) виконується операція, вказана у якості параметра `reduction`.

Приклад 10.7. Обчислення $(2n+1)!! = 1 \cdot 3 \cdot \ldots \cdot (2n+1)$ та $(2n)!! = 2 \cdot 4 \cdot \ldots \cdot (2n)$.

```
const int N = 16;
int p1 = 1, p2 = 1;
#pragma omp parallel for reduction(*: p1, p2)
for(int i = 1; i <= N; i += 2){
    p1 *= i;
    p2 *= i+1;
}
printf("p1 = %d, p2 = %d\n", p1, p2);
```

10.4.2. Розподіл навантаження між потоками.

Зазвичай кількість потоків менша за кількість ітерацій циклу. В такому випадку за замовчуванням навантаження рівномірно розподіляється між потоками. Для цього кількість ітерацій ділиться з остачею на кількість потоків. Частка — кількість ітерацій на один потік. Ітерації, які відповідають остачі, діляться між потоками починаючи з 0-го. Наприклад, якщо треба виконати 18 ітерацій на 8 потоках, то потоки з номерами 0 та 1 виконують по 3 ітерації, а усі інші потоки — по 2 ітерації.

Цей спосіб розподілу може бути неефективний у випадку, коли тривалості ітерацій сильно залежать від їх номеру. Наприклад, якщо перші ітерації виконується значно довше, ніж інші, то потік з номером 0 може завершитися значно пізніше, ніж інші потоки. Для зміни способу розподілу ітерацій між потоками використовується опція

```
schedule(type[, chunk])
```

де параметр `type` може приймати такі значення:

- `static` — блоково-циклічний розподіл; розмір блоку – `chunk`. Перший блок з `chunk` ітерацій виконує потік 0, другий блок — наступний і т. д. до останнього потоку, а далі розподіл знову розпочинається з потоку 0. Якщо `chunk` не вказано, то використовується розподіл за замовчуванням.
- `dynamic` — динамічний розподіл з фіксованим розміром блоку: спочатку кожний потік отримує `chunk` ітерацій (за замовчуванням `chunk = 1`), той потік, який закінчив виконання своєї порції ітерацій, отримує першу вільну порцію із `chunk` ітерацій і т. д.
- `guided` — динамічний розподіл ітерацій, при якому розмір порції зменшується з деякого початкового значення до величини `chunk` (за замовчуванням `chunk = 1`) пропорційно кількості ще не розподілених ітерацій, поділеній на кількість потоків. Розмір початкового блоку залежить від реалізації.

На рис. 10.1–10.3 наведено діаграму розподілу навантаження між потокам.

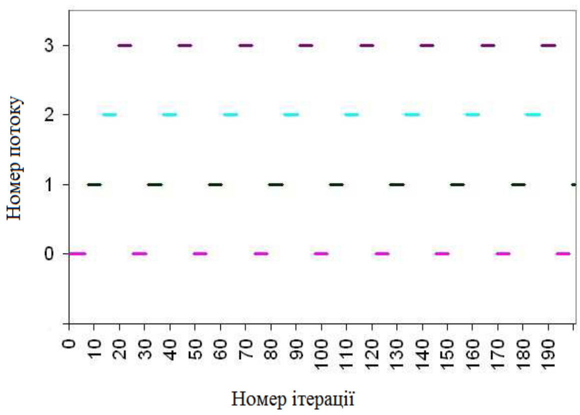

Рис. 10.1. Розподіл ітерацій в режимі `(static, 6)`

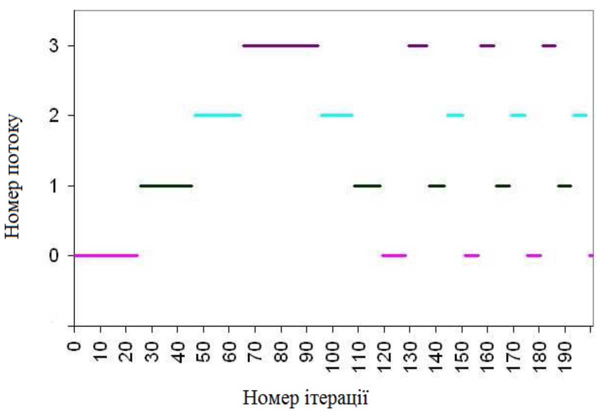

Рис. 10.2. Ітерації в режимі `(guided, 6)`

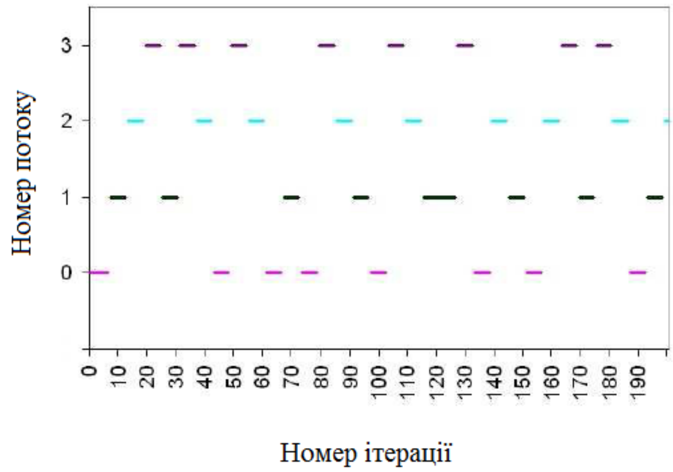

Рис. 10.3. Розподіл ітерацій в режимі (dynamic, 6)

На рис. 10.2. видно, що в режимі guided розмір порції зменшується від 24 до 6.

10.5. Паралельні секції

Директива sections використовується для задання скінченного (неітеративного) паралелізму:

```
#pragma omp sections [опція [[,] опція]...]
```

Директива section задає ділянку коду всередині секції sections для виконання одним потоком. Перед першою ділянкою коду в блоці sections директива section не обов'язкова. *Які саме потоки будуть задіяні для виконання секції, наперед невідомо*.

Наступний приклад ілюструє застосування директиви sections. Спочатку три потоки, які виконують відповідні секції, виведуть повідомлення зі своїм номером, а потім всі потоки надрукують повідомлення зі своїм номером.

```
int n;
#pragma omp parallel private(n)
{
    n = omp_get_thread_num();
    #pragma omp sections
    {
        #pragma omp section
            printf("first section in thread %d\n", n);
        #pragma omp section
            printf("second section in thread %d\n", n);
        #pragma omp section
            printf("third section in thread %d\n", n);
    }
    printf("Parallel scope in thread %d\n", n);
}
```

Приклад 10.8. Обчислення суми $a[0]a[1]+a[1]a[2]+\ldots+a[5]a[6]+a[6]a[0]$ у системі із трьома процесорами.

```
double a[7];
for(int i = 0; i < 7; i++)
    a[i] = i+1;
double r = 0, r1, r2, r22, r3;
#pragma omp parallel sections
{
    r1 = a[1]+a[6];
    # pragma omp section
        r2 = a[1]+a[3];
    # pragma omp section
        r3 = a[3]+a[5];
}
#pragma omp parallel sections
{
    r1 = r1*a[0];
    # pragma omp section
        r2 = a[2]*r2;
    # pragma omp section
        r3 = a[4]*r3;
}
#pragma omp parallel sections
{
    r1 = r1 + r2;
    # pragma omp section
    r22 = a[5]*a[6];
}
r1 += r3;
r = r1 + r22;
printf("r = %f\n", r);
```

10.6. Синхронізація

Низка директив OpenMP призначена для синхронізації роботи потоків [1].

10.6.1. Бар'єр

Найпоширеніший спосіб синхронізації в OpenMP — бар'єр, який описується за допомогою директиви `barrier`:

```
#pragma omp barrier
```

Потоки, що виконують поточну паралельну область, дійшовши до цієї директиви, зупиняються і чекають, поки всі потоки не дійдуть до цієї точки програми, після чого розблоковуються і продовжують працювати далі. Наступний приклад демонструє застосування директиви `barrier`. Виведення повідомлень 1 та 2 можуть чергуватися в довільному порядку, а вивід повідомлення 3 обов'язково йде після двох попередніх.

```
#pragma omp parallel
{
    printf("Message 1\n");
    printf("Message 2\n");
    #pragma omp barrier
    printf("Message 3\n");
}
```

Слід зазначити, що неявні бар'єри автоматично додаються у кінці областей, описаних за допомогою директив `parallel`, `for`, `sections` та `single`. У останніх трьох випадках цього можна уникнути за допомогою `nowait`.

10.6.2. Директива **`ordered`**

Директива `ordered` визначає блок всередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть в послідовному циклі. Синтаксис:

```
#pragma omp ordered
```

Блок операторів відноситься до самого внутрішнього з циклів, а в паралельному циклі повинна бути задана опція `ordered`. Потік, який виконує першу ітерацію циклу, виконує операції даного блоку. Потік, який виконує будь-яку наступну ітерацію, має спочатку дочекатися виконання всіх операцій блоку усіма потоками, які виконують попередні ітерації.

Наступний приклад ілюструє директиву `ordered` і опцію `ordered`. Цикл `for` позначений як `ordered`. Усередині тіла циклу йдуть два виведення — одне поза блоком `ordered`, а друге — всередині нього. В результаті перше виведення виходить невпорядкованим, а друге — в строгому порядку по зростанню номера ітерації.

```
int i, n;

#pragma omp parallel private (i, n)
{
    n = omp_get_thread_num();
    #pragma omp for ordered // ordered is clause here
```

```
    for (i = 0; i < 15; i++)
    {
        printf("Thread %d, iteration %d\n", n, i+1);
        #pragma omp ordered  // ordered directive
        printf("Ordered: thread %d, iteration %d\n", n, i+1);
    }
}
```

10.6.3. Критичні секції

За допомогою директив `critical` оформляється критична секція програми:

```
#pragma omp critical [ім'я]
```

У кожен момент часу в критичній секції може перебувати не більше одного потоку. Якщо критична секція вже виконується якимось потоком, то всі інші потоки, які хочуть увійти у критичну секцію, будуть заблоковані, поки потік, що увійшов у критичну секцію, не закінчить виконання даної критичної секції. Як тільки цей потік вийде з критичної секції, то один з заблокованих на вході потоків увійде в неї, а інші заблоковані потоки продовжать очікування.

Все неіменовані критичні секції умовно асоціюються з одним і тим самим ім'ям. Усі критичні секції, що мають одне і те саме ім'я, розглядаються як єдина секція, навіть якщо вони знаходяться в різних паралельних областях.

У наступному прикладі зміна `n` оголошена поза паралельною областю, тому за замовчуванням вона є спільною. Критична секція дозволяє розмежувати доступ до змінної `n`. Кожний потік по черзі присвоїть `n` свій номер і потім надрукує отримане значення.

```
int n;
#pragma omp parallel num_threads(20)
{
    #pragma omp critical
    {
        n = omp_get_thread_num();
        cout << "thread " << n << '\n';
    }
}
```

Якби у попередньому прикладі не було використано директиву `critical`, результат виконання програми був би непередбачуваний. З директивою `critical` порядок виведення результатів може бути довільним, але це завжди буде набір одних і тих же чисел від `0` до `OMP_NUM_THREADS-1`. Подібного результату можна було б досягти іншими способами, наприклад, оголосивши змінну `n` локальною, тоді кожний потік працював би зі своєю копією цієї змінної.

Однак у виконанні цих фрагментів різниця є суттєвою. Якщо є критична секція, то в кожен момент часу фрагмент буде оброблятися лише якимось одним потоком. Інші потоки, навіть якщо вони вже підійшли до цієї точки програми і готові до роботи, будуть очікувати своєї черги. Якщо критичної секції немає, то всі потоки можуть одночасно виконати дану ділянку коду. З одного боку, критичні секції надають зручний механізм для роботи з спільними змінними. З іншого боку, користуватися ним потрібно обережно, оскільки вони додають послідовні ділянки коду в паралельну програму.

Приклад 10.9. Знаходження найменшого елемента масиву з використанням подвійної перевірки умови (див підрозділ 5.3).

```
const int N = 1000;

int a[N];
for(int i = 0; i < N; i++)
    a[i] = 2*rand() - RAND_MAX;
int min = RAND_MAX;

#pragma omp parallel for shared(min)
    for(int i = 0; i < N; i++)
        if(a[i] < min){
            #pragma omp critical
            {
                if(a[i] < min){
                    min = a[i];
                }
            }
        }
printf("min = %d", min);
```

10.6.4. Директива `atomic`

Найчастіше при використанні критичних секцій на практиці в них лише оновлюють спільні змінні. Наприклад, якщо змінна `sum` є загальної і оператор вигляду `sum = sum + expr` знаходиться в паралельній області програми, то при одночасному виконанні цього оператора кількома потоками можна отримати некоректний результат. Щоб уникнути такої ситуації можна скористатися механізмом критичних секцій або спеціально передбаченої для таких випадків директивою `atomic`.

Ця директива *стосується лише до одного, наступного за нею оператора*, у якому проводиться обчислення значення деякого виразу, який має мати одну з наступних форм:

```
x binop= expr
x++
```

```
++x
x--
--x
```

На час виконання оператора, який йде за директивою, блокується доступ до змінної `x` всім запущеним в даний момент потокам, крім того, який виконує операцію. Атомарною є тільки робота із змінною в лівій частині оператора присвоювання, при цьому обчислення в правій частині не зобов'язані бути атомарними. Наведемо приклад застосування директиви `atomic`.

```
int s = 0;
#pragma omp parallel num_threads(20)
{
    int n = omp_get_thread_num();
    #pragma omp atomic
        s += (n+1);
}
printf("s = %d\n", s);
```

Для того, щоб запобігти одночасної зміни кількома потоками значення змінної `s`, використовується директива `atomic`.

10.6.5. Замки (блокування)

Один з варіантів синхронізації в OpenMP реалізується через механізм замків або блокувань (`locks`). У якості замків використовуються цілочислові змінні (розмір повинен бути достатнім для зберігання адреси). Ці змінні повинні використовуватися тільки як параметри примітивів синхронізації. Замок може перебувати в одному з трьох станів: неініціалізований, розблокований або заблокований. Розблокований замок може бути захоплений деяким потоком. При цьому він переходить в заблокований стан. Потік, який захопив замок, і тільки він може його звільнити, після чого замок повертається в розблокований стан.

Є два типи замків: прості замки і множинні замки. Множинний замок може багаторазово захоплюватися одним потоком перед його звільненням, в той час як простий замок може бути захоплений тільки один раз. Для множинного замку вводиться поняття коефіцієнта захоплення. Спочатку він встановлюється в нуль, при кожному наступному захопленні збільшується на одиницю, а при кожному звільненні зменшується на одиницю. Множинний замок вважається розблокованим, якщо його коефіцієнт захоплення дорівнює нулю [18].

Для ініціалізації простого або множинного замку використовуються відповідно функції `omp_init_lock()` і `omp_init_nest_lock()`:

```
void omp_init_lock(omp_lock_t* lock);
void omp_init_nest_lock(omp_nest_lock_t* lock);
```

Функції

```
void omp_destroy_lock(omp_lock_t* lock);
void omp_destroy_nest_lock(omp_nest_lock_t* lock);
```

використовуються для переведення простого або множинного замку в неініціалізований стан.

Для захоплення замку використовуються функції:

```
void omp_set_lock(omp_lock_t* lock);
void omp_set_nest_lock(omp_nest_lock_t* lock);
```

Потік, що викликав цю функцію, чекає звільнення замку, а потім захоплює його. Замок при цьому переводиться в заблокований стан. Якщо множинний замок вже захоплений даним потоком, то потік не блокується, а коефіцієнт захоплення збільшується на 1.

Для звільнення замку використовуються функції

```
void omp_unset_lock (omp_lock_t * lock);
void omp_unset_nest_lock (omp_lock_t * lock);
```

Їх виклик звільняє простий замок, якщо він був захоплений викликаючим потоком. Для множинного замку коефіцієнт захоплення зменшується на одиницю. Якщо коефіцієнт дорівнює нулю, замок звільняється. Якщо після звільнення замку є потоки, заблоковані на операції захоплення замка, замок буде відразу ж захоплений одним з таких очікуючих потоків.

Наступний приклад ілюструє застосування технології замків.

```
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel num_threads(10)
{
    int n = omp_get_thread_num();
    omp_set_lock(&lock);
    printf("Protected section %d starts\n", n);
    Sleep(5);
    printf("Protected section %d finishes\n", n);
    omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
```

Для перевірки можливості захоплення замку (без блокування у випадку неможливості захоплення) використовуються функції:

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

Ці функції пробують захопити вказаний замок. Якщо це вдалося, то для простого замка функція повертає 1, а для множинного замку — новий коефіцієнт захоплення. Якщо замок захопити не вдалося, в обох випадках повертається 0.

Наступний приклад ілюструє використання функції `omp_test_lock()`.

```
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel num_threads(4)
{
    int n = omp_get_thread_num();
    while(!omp_test_lock(&lock)){
        printf("Protected section is closed for thread %d\n", n);
        Sleep(2);
    }
    printf("Protected section %d starts\n", n);
    Sleep(5);
    printf("Protected section %d finishes\n", n);
    omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);
```

10.6.6. Директива `flush`

Оскільки в сучасних паралельних обчислювальних системах може використовуватися складна структура і ієрархія пам’яті, користувач повинен мати гарантії того, що в необхідні йому моменти часу всі потоки будуть бачити єдиний узгоджений образ пам’яті. Саме для цього призначена директива `flush`:

```
#pragma omp flush [(список)]
```

Виконання даної директиви передбачає, що значення всіх змінних (змінних зі списку), які тимчасово зберігаються в регістрах і кеш-пам'яті поточного потоку, будуть занесені в основну пам’ять; всі зміни значень змінних, зроблені потоком під час роботи, стануть видимі іншим потокам; якщо якась інформація зберігається в буферах виведення, то буфери будуть скинуті і т.п. При цьому операція проводиться тільки з даними потоку, у якому відбувся виклик, а дані, що змінювалися іншими потоками, не чіпаються. Неявно `flush` без параметрів присутній в директиві `barrier`, на вході і виході областей дії директив `parallel`, `critical`, `ordered`, на виході областей розподілу робіт, у викликах функцій роботи із замками (`omp_set_lock()` і т. п.). Крім того, `flush` викликається для змінної, яка бере участь в операції, асоційованій з директивою `atomic`.

10.7. Приклади використання OpenMP

Приклад 10.10. Обчислення числа π. Скористаємося формулою

$$\pi / 4 = \operatorname{arctg} 1 = \int_0^1 \frac{1}{1+x^2} dx .$$

Для обчислення визначеного інтегралу скористаємося формулою середніх прямокутників:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f\left(a + \left(i + \tfrac{1}{2}\right)h\right) \cdot h, \; h = \frac{b-a}{n}.$$

Розглянемо послідовну та паралельну версії програми:

```
double pi_calc(int n){
    double s = 0;
    double h = 1.0 / n;
    for(int i = 0; i < n; i++){
        double x = h * (i + 0.5);
        s += h / (1 + x * x);
    }
    s *= 4;
    return s;
}
double pi_calc_parallel(int n){
    double s = 0;
    double h = 1.0 / n;
    #pragma omp parallel for reduction(+:s)
        for(int i = 0; i < n; i++){
            double x = h * (i + 0.5);
            s += h / (1 + x * x);
        }
    s *= 4;
    return s;
}
void main()
{
    int n = 100000;
    double start = omp_get_wtime();
    double pi = pi_calc(n);
    double finish = omp_get_wtime();
    printf("Pi: %e. Duration in serial mode: %e\n", pi, finish-start);
    start = omp_get_wtime();
    pi = pi_calc_parallel(n);
    finish = omp_get_wtime();
    printf("Pi: %e. Duration in parallel mode: %e\n", pi, finish-start);
}
```

Приклад 10.11. Обчислення добутку двох матриць.

```
double a[N][N], b[N][N], c[N][N];
int main()
{
    int i, j, k;
    double t1, t2;
    for (i = 0; i < N; i++) // initialization
        for (j = 0; j < N; j++)
            a[i][j] = b[i][j] = i * j;

    #pragma omp parallel for private(i, j, k)
    for(i = 0; i < N; i++){
        for(j = 0; j < N; j++){
            c[i][j] = 0.0;
            for(k = 0; k < N; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Коцовський В. М. Технології розподілених систем та паралельних обчислень. Частина I: Методичний посібник. Ужгород: Видавництво УжНУ "Говерла", 2017. 51 с.
2. Коцовський В. М. Технології розподілених систем та паралельних обчислень. Частина II: Методичний посібник. Ужгород: Видавництво УжНУ "Говерла", 2017. 76 с.
3. Коцовський В. М. Теорія паралельних обчислень. Частина I: Методичний посібник. Ужгород: Видавництво УжНУ "Говерла", 2019. 51 с.
4. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.
5. Коцовський В. М. Теорія паралельних обчислень. Частина II: Методичний посібник. Ужгород: Видавництво УжНУ "Говерла", 2019. 52 с.
6. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. М.: Издательский дом "Вильямс", 2003. 512 с.
7. Качко Е. Г. Параллельное программирование: Учебное пособие. Харьков: "Форт", 2011. 528 с.
8. Антонов А. С. Параллельное программирование с использованием технологии OpenMP. М.: Изд-во МГУ, 2009. 77 с.
9. Воеводин В. В. Математические основы параллельных вычислений. – М.: МГУ, 1991. 345 с.
10. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ, 2-е издание. М.: "Вильямс", 2005. 1296 с.
11. Оленев Н. Основы параллельного программирования в системе MPI. М.: Вычислительный центр им. А. А. Дородицына РАН, 2005. 81 с.
12. Сердюк Ю. П. Введение в параллельное программирование на языке MC#. Переславль-Залесский: Институт программных систем РАН, 2007. 51 с.
13. Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. 232 с.
14. Коцовський В. М. Технології розподілених систем та паралельних обчислень: Методичні матеріали до лабораторних робіт. Ужгород: Видавництво УжНУ «Говерла», 2020. 32 с.
15. Коцовський В. М. Основи дискретної математики: навчальний посібник. Ужгород: ПП «АУТДОР-ШАРК», 2020. 128 с.
16. Шилдт Г. Java. Полное руководство. М.: ООО "И.Д. Вильямс", 2012. 1104 с.
17. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0, 5-е изд. М.: "Вильямс", 2011. 1392 с.

Інформаційні ресурси

18. OpenMP Architecture Review Board. URL: http://www.openmp.org
19. C++ documentation. URL: http://docs.microsoft.com/en-us/cpp/
20. C++ documentation. URL: http://www.cplusplus.com
21. C# documentation. URL: http://docs.microsoft.com/en-us/dotnet/csharp
22. JDK 17 Documentation. URL: https://docs.oracle.com/en/java/javase/17/
23. Message passing interface. URL: http://www.mpiforum.org
24. Параллельное программирование с использованием технологии OpenMP. URL: http://parallel.ru/tech/tech_dev/OpenMP/examples/
25. Development of distributed computing. URL: http://www.gridforum.org
26. CUDA Toolkit Documentation v11.5.1. URL: https://docs.nvidia.com
27. Сайт електронного навчання ДВНЗ "УжНУ". Теорія паралельних обчислень. URL: https://e-learn.uzhnu.edu.ua/course/view.php?id=2737