ДВНЗ "Ужгородський національний університет"
Факультет інформаційних технологій
Кафедра інформаційних управляючих систем та технологій

В. М. Коцовський

Теорія паралельних обчислень

Конспект лекцій

Частина І

Ужгород – 2019

Зміст

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.

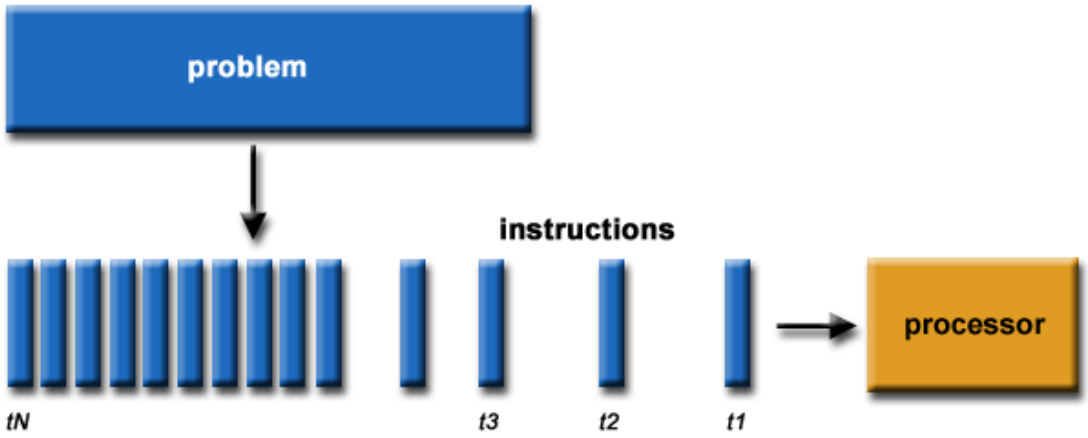


Рис. 1.1. Схема послідовних обчислень

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

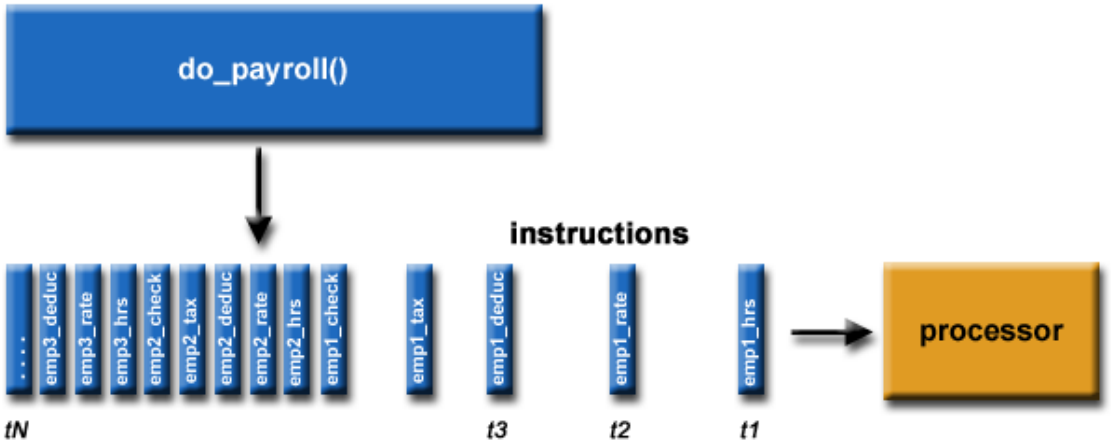


Рис. 1.2. Схема обчислення зарплати

1.2. Паралельні обчислення

Паралельні обчислення (Parallel Computing) — 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 паралельних обчисленнях:

- Один комп'ютер з кількома процесорами.
- Довільну кількість комп'ютерів з'єднаних у мережу.

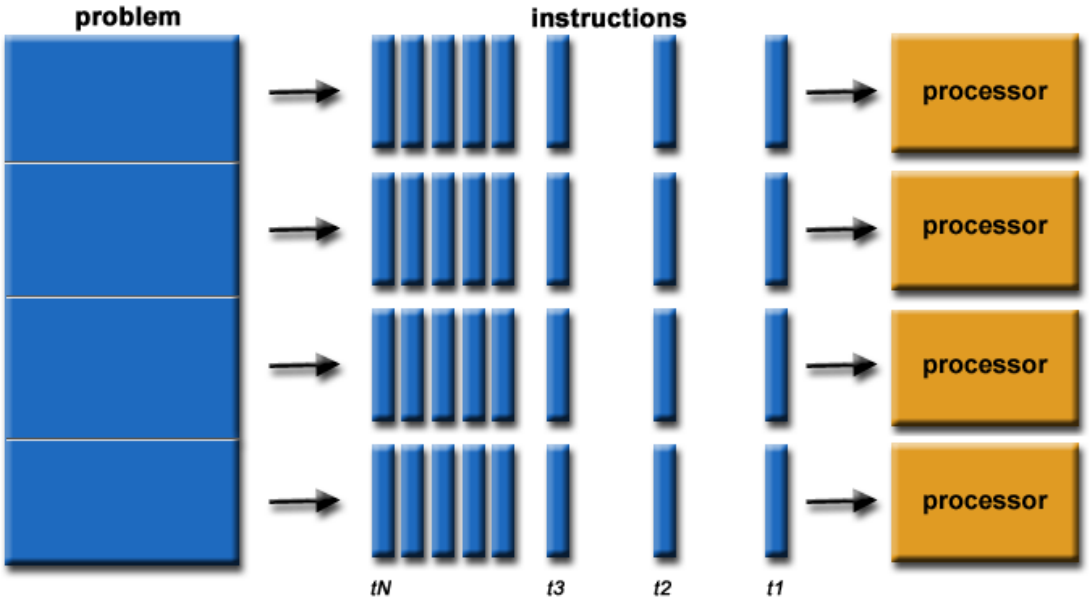


Рис. 1.3. Схема паралельних обчислень

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

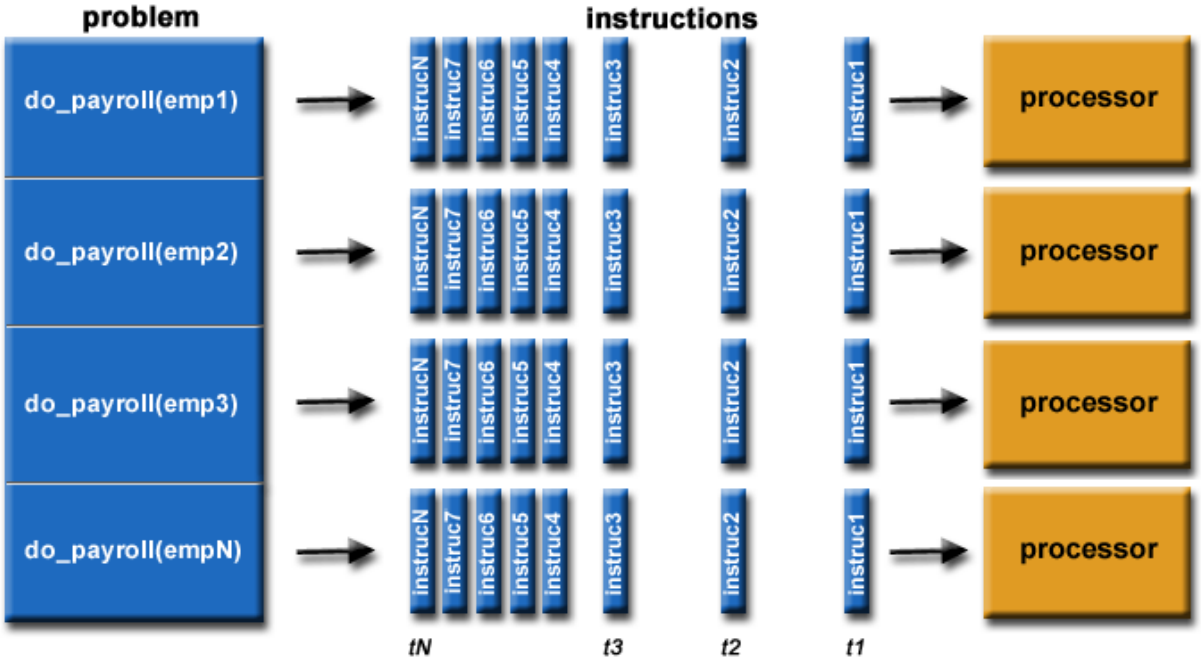


Рис. 1.4. Схема паралельного обчислення зарплати

1.3. Засоби для здійснення паралельних обчислень

Засоби, які дають змогу втілити парадигму паралелізму, можна 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 і т.д.)
  - Багатоядерність чи багатопроцесорність (Multiple execution units/cores).
  - Підтримка використання потоків (Multiple hardware threads)

Будова обчислювального чіпу IBM BG/Q з 18 ядрами (PU) та 16 кеш-модулями 2-го рівня (L2) наведена на рис. 1.5.

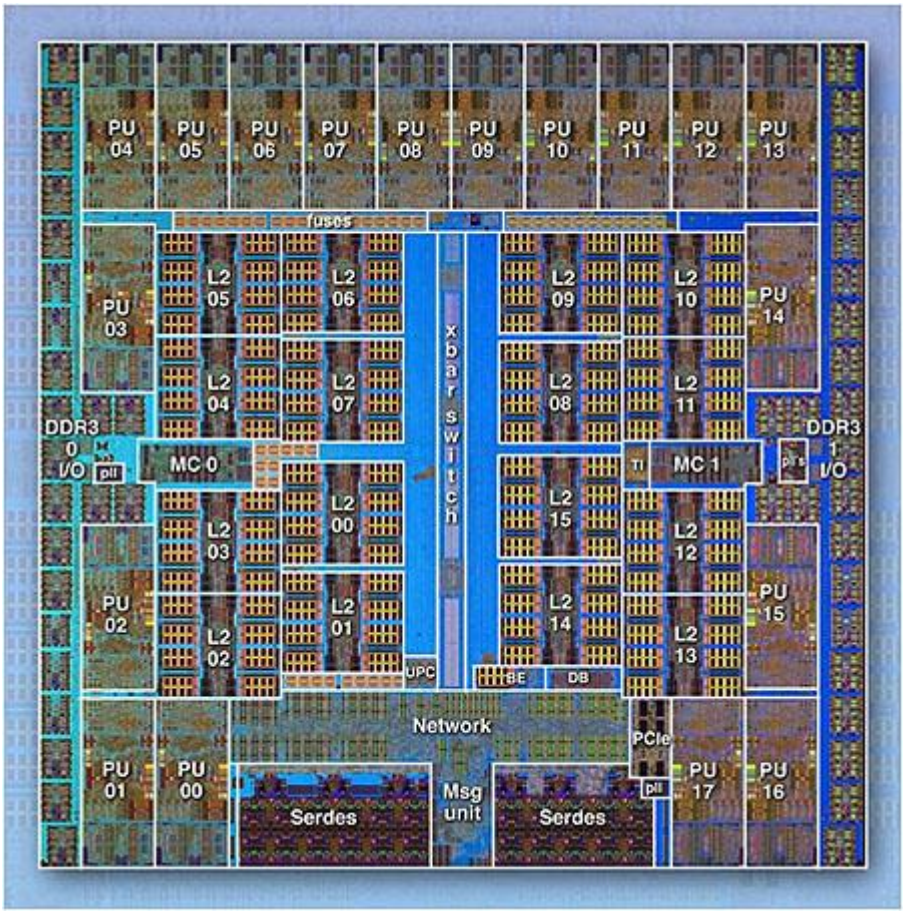


Рис. 1.5. Обчислювальний чіп IBM BG/Q

- Кластери, які складаються із кількох робочих станцій, з’єднаних мережею (див рис. 1.6).

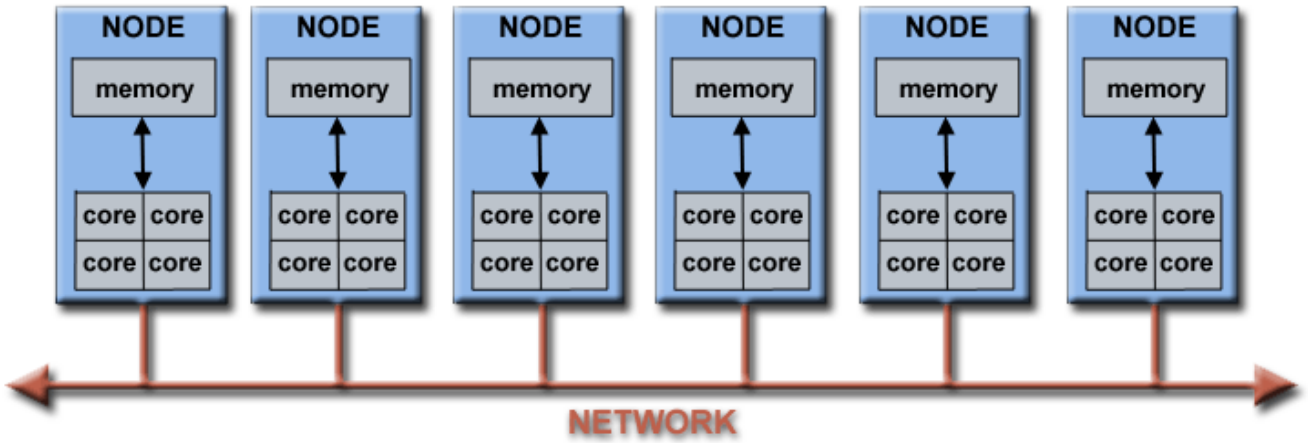


Рис. 1.6. Схема кластера

Наприклад, на рис. 1.7 наведено схему типового кластера для паралельних обчислень, які використовуються в LLNL (Lawrence Livermore National Laboratory). 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

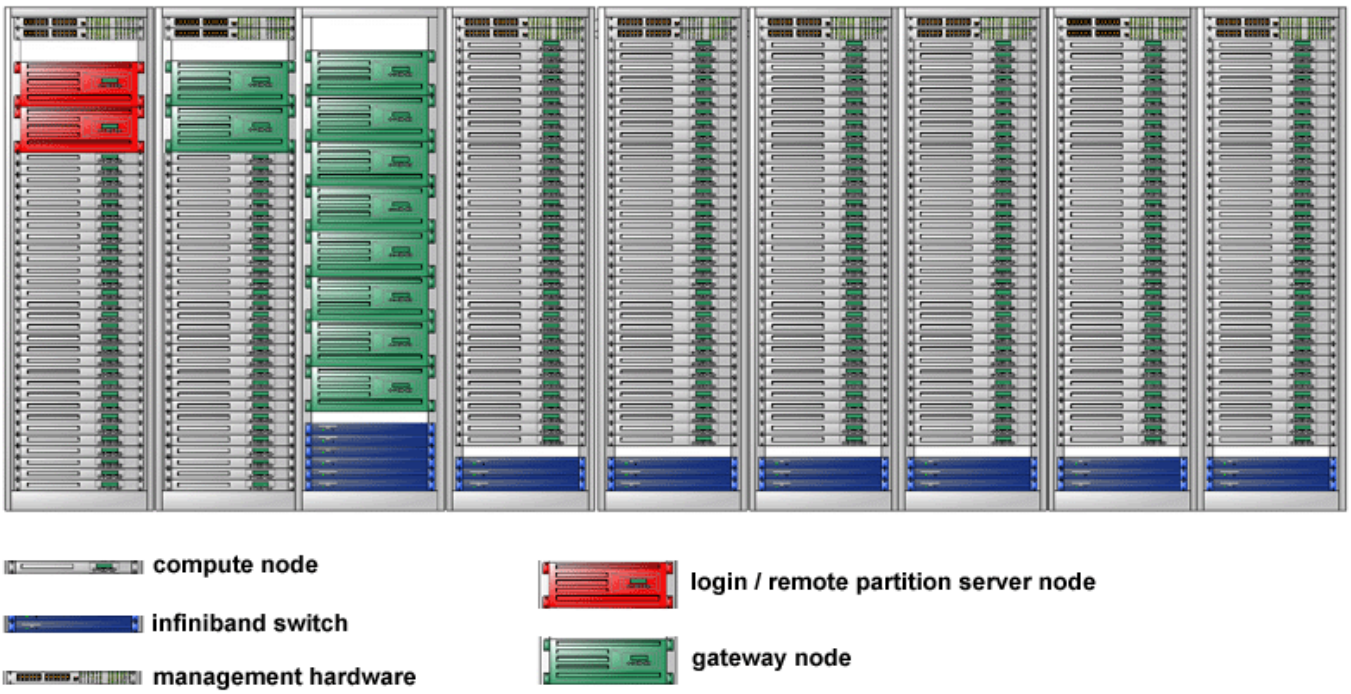
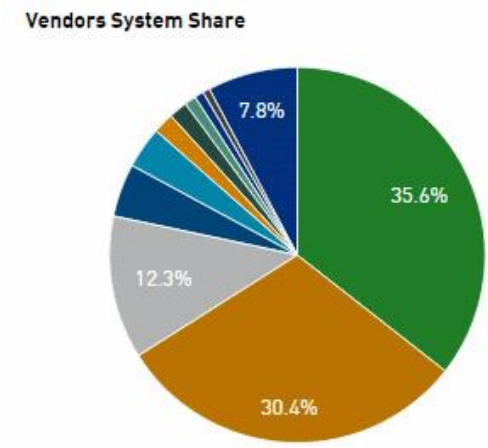
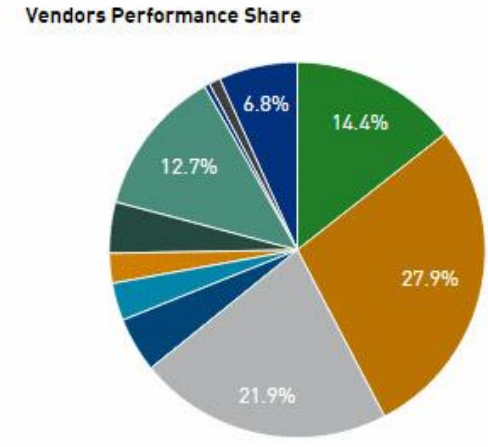


Рис. 1.7. Схема будови кластера LLNL

Дані про найбільші у світі паралельні комп’ютери (суперкомп’ютери), переважна більшість яких є кластерами, наведено на рис. 1.8 (джерело — сайт [Top500.org](Top500.org)).





VENDORS	COUNT	SYSTEM SHARE (%)
HP	179	35.8
IBM	153	30.6
Cray Inc.	62	12.4
SGI	23	4.6
Bull	18	3.6
Dell	9	1.8
Fujitsu	8	1.6
NUDT	5	1
RSC Group	4	0.8
Atipa	3	0.6
NEC	3	0.6
MEGWARE	3	0.6
T-Platforms	2	0.4
Itautec	2	0.4
Hitachi	2	0.4
Oracle	2	0.4
Self-made	2	0.4
ClusterVision	2	0.4
Dawning	2	0.4
NEC/HP	1	0.2
Acer Group	1	0.2
PEZY Computing / Exascaler Inc.	1	0.2
HP/WIPRO	1	0.2
Inspur	1	0.2
Niagara Computers, Supermicro	1	0.2
Clustervision/Supermicro	1	0.2
AMD, ASUS, FIAS, GSI	1	0.2
Xenon Systems	1	0.2
Netweb Technologies	1	0.2
IPE, Nvidia, Tyan	1	0.2
Adtech	1	0.2
Intel	1	0.2
NRCPCET	1	0.2
Supermicro	1	0.2
Hitachi/Fujitsu	1	0.2

Рис. 1.8. Ринок суперкомп’ютерів

1.5. Актуальність використання паралельних обчислень

Важливість дослідження та використання паралельних обчислень зумовлена наступними причинами:

а) *Явища у реальному світі відбуваються паралельно* (The Real World is Massively Parallel).

Тому паралельні обчислення 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. 1.9.

Рис. 1.9. 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

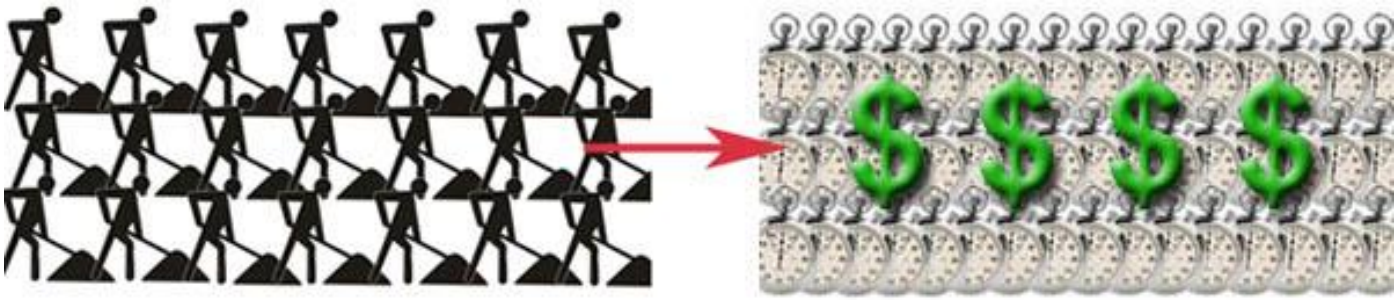


Рис. 1.10. Економічний ефект від використання паралельних обчислень

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.

Рис. 1.11. Деякі складні задачі біології та астрономії

Ще один приклад: Web-пошуковики, які виконують мільйони транзакцій щосекунди, та задачі прогнозу погоди. Область розв’язку (атмосфера) розбивається на окремі про-

сторові 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 х 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.

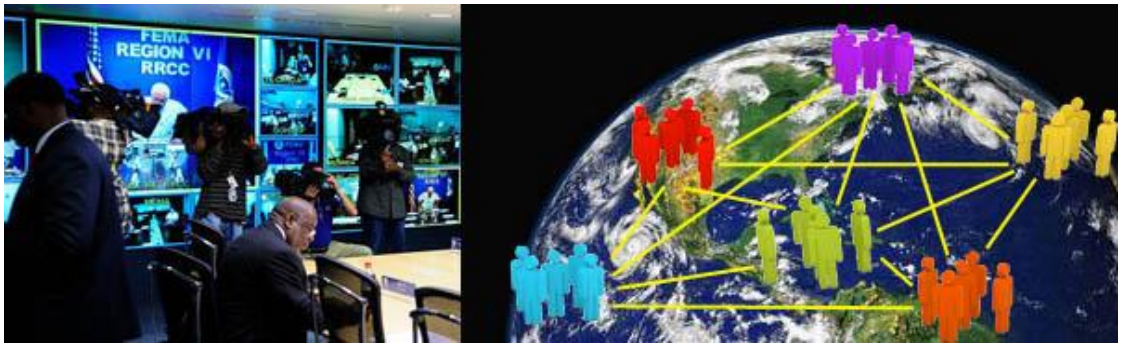


Рис. 1.12. Забезпечення одночасної роботи

е) Використання глобальних ресурсів:

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

Наприклад, SETI@home (setiathome.berkeley.edu) використовує понад 1,5 мільйонів користувачів із усього світу.

Рис. 1.13. Використання нелокальних ресурсів

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».

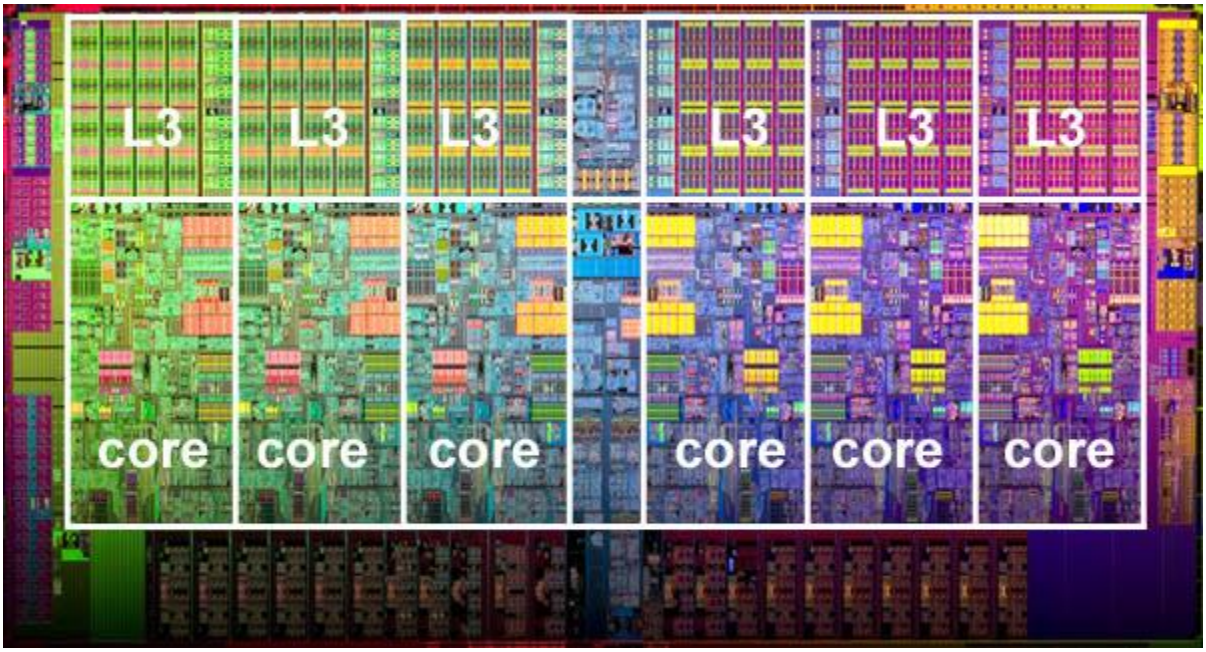


Рис. 1.14. Процесор Intel Xeon з 6 ядрами та 6 модулями L3-кеша.

1.6. Перспективи використання паралельних обчислень

Згідно до прогнозів, наведених у [1], 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. 1.15 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.

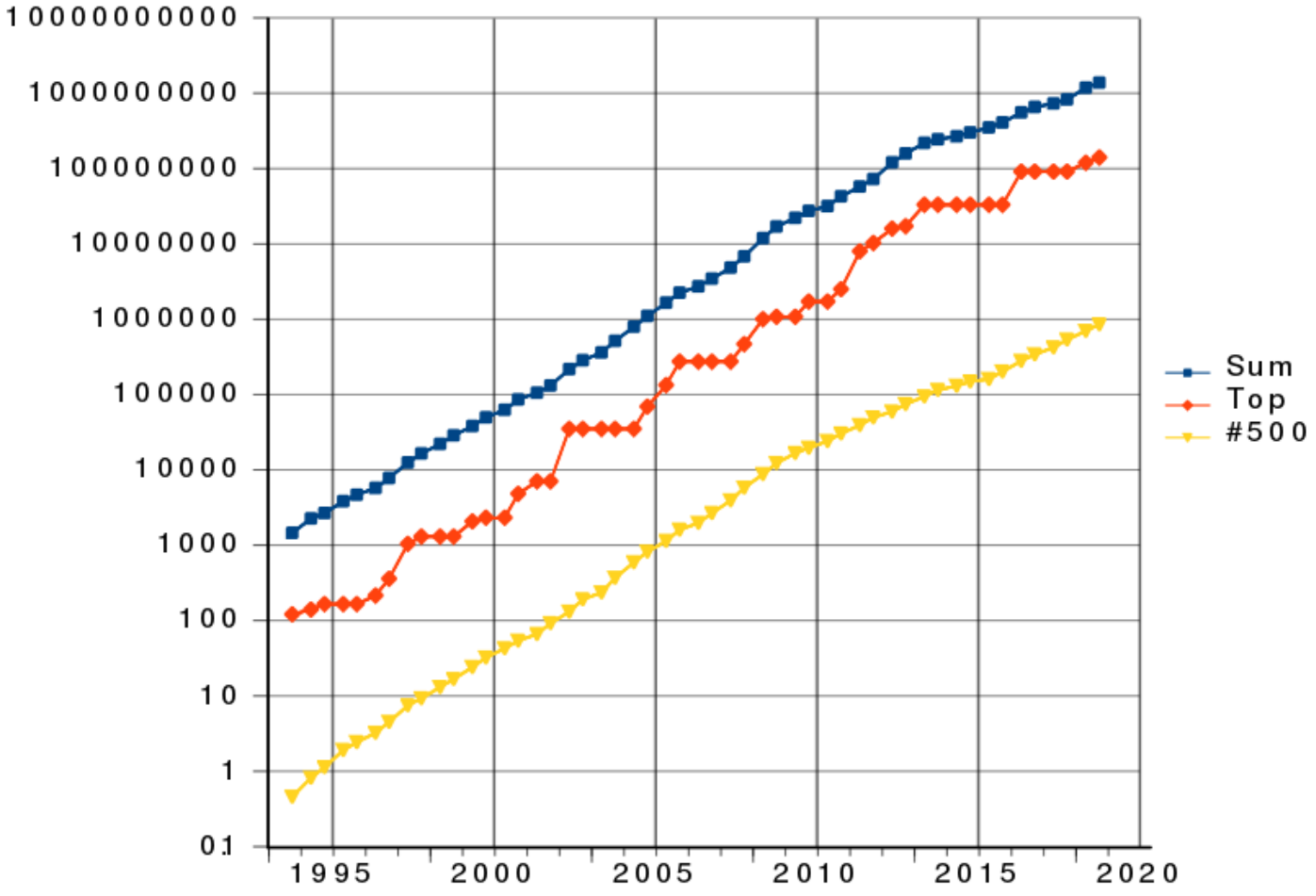


Рис. 1.15. Зростання продуктивності суперкомп’ютерів (у Terflops за даними Top500.org).

Станом на червень 2018 року рейтинг Top500 мав такий вигляд (за тестом LINPACK benchmarks):

1. Summit (США) — 122,3 петафлопс
2. Sunway TaihuLight (КНР) — 93,0
3. Sierra (США) — 71,6
4. Tianhe-2A (КНР) — 61,4
5. AI Bridging Cloud Infrastructure (Японія) — 19,9.

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. Він охолоджується системою, в якій циркулює 15150 літрів очищеної води. Summit — перший комп'ютер, який подолав межу екаор (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 наукових та інженерних задач у таких галузях (див. рис. 1.16):

- 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.

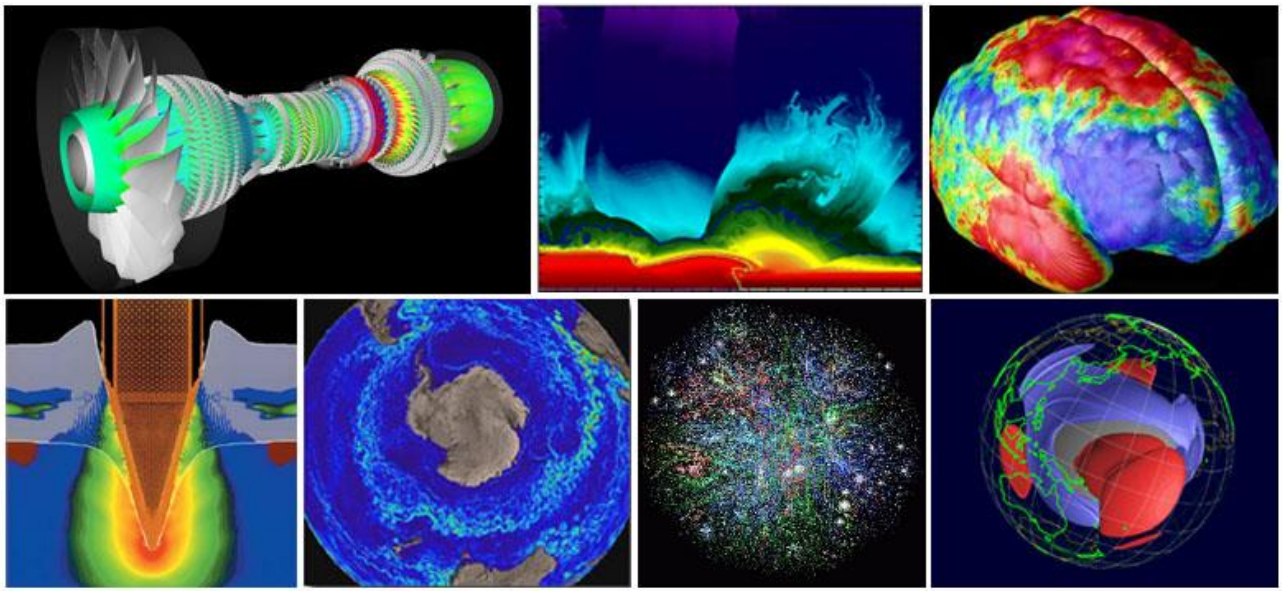


Рис. 1.16. Сфери застосування паралельних обчислень

б) Індустрія та комерція.

У наш час індустріальні та комерційні застосування є основним драйвером розвитку інформаційних технологій. Паралельні обчислення використовуються при розв’язуванні задач (див. рис. 1.17):

- розвідування корисних копалин;
- фінансове та економічне моделювання;
- обробки великих БД, Big Data, Data Mining;
- Web-пошук, web-торгівля;
- Обробка медичних зображень та діагнозування;
- Обробка графіки, мультимедіа та віртуальна реальність;
- Мережева співпраця;
- Військова справа, розробка зброї.

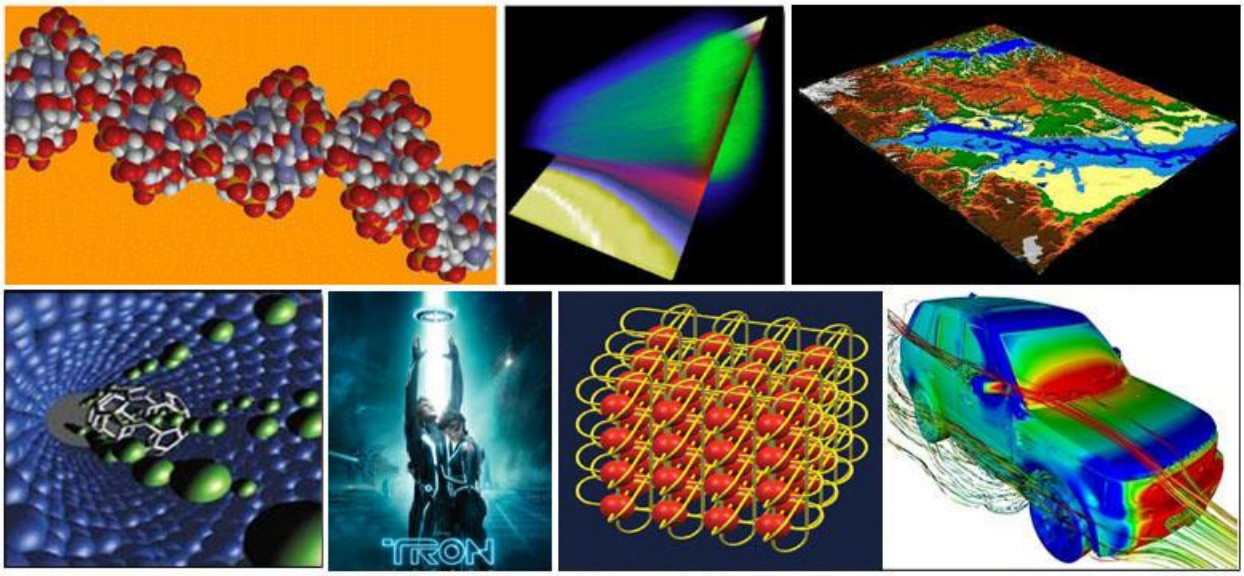


Рис. 1.17. Сфери застосування паралельних обчислень.

2. ПАРАЛЕЛЬНІ ОБЧИСЛЮВАЛЬНІ СИСТЕМИ

2.1. Рівні розпаралелювання

Існують різні підходи до визначення типів паралелізму [1, 2].

2.1.1. Перший підхід до класифікації паралелізму

Класифікація паралельності за рівнями, що відрізняються показниками абстрактності розпаралелювання задач, наведена в табл. 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 (див. рис. 2.2), то воно синхронно розпаралелюється дуже просто тому, що кожному процесору відповідає один елемент матриці-результату.

$$\begin{pmatrix} 1 & 1 \\ 3 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 1 \\ 2 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 5 & 4 \end{pmatrix}$$

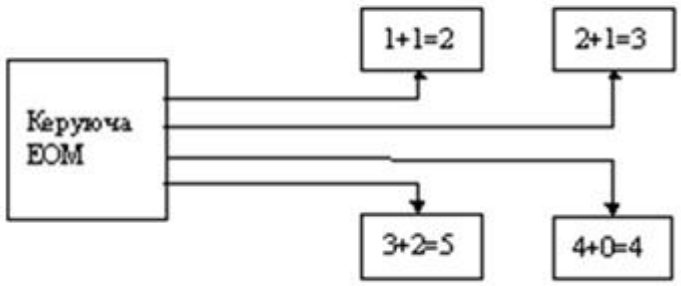


Рис. 2.2. Розпаралелювання при додаванні матриць на рівні формул

При застосуванні n^2 процесорних елементів можна одержати суму двох матриць порядку n^2 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 [2] 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 суперскаляр-

ними 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 C 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. Розвиток процесу обчислень схематично наведено на рис. 2.3.

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 (рис. 2.4) — 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.

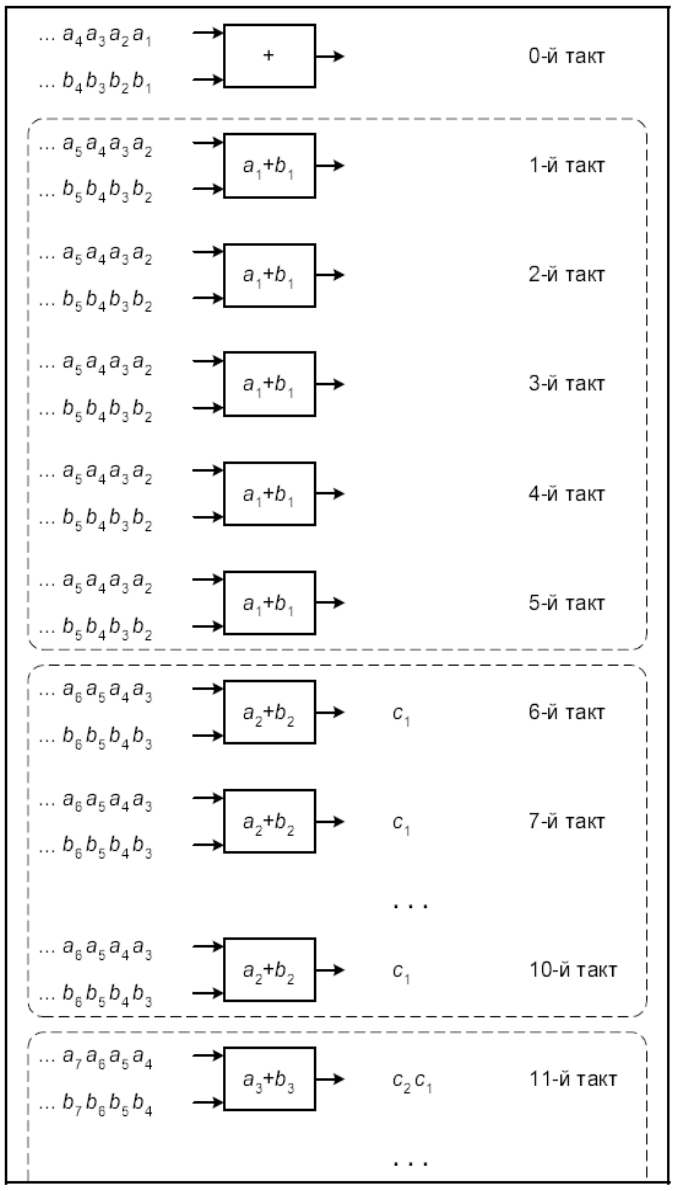


Рис. 2.3. Додавання векторів на послідовному пристрої

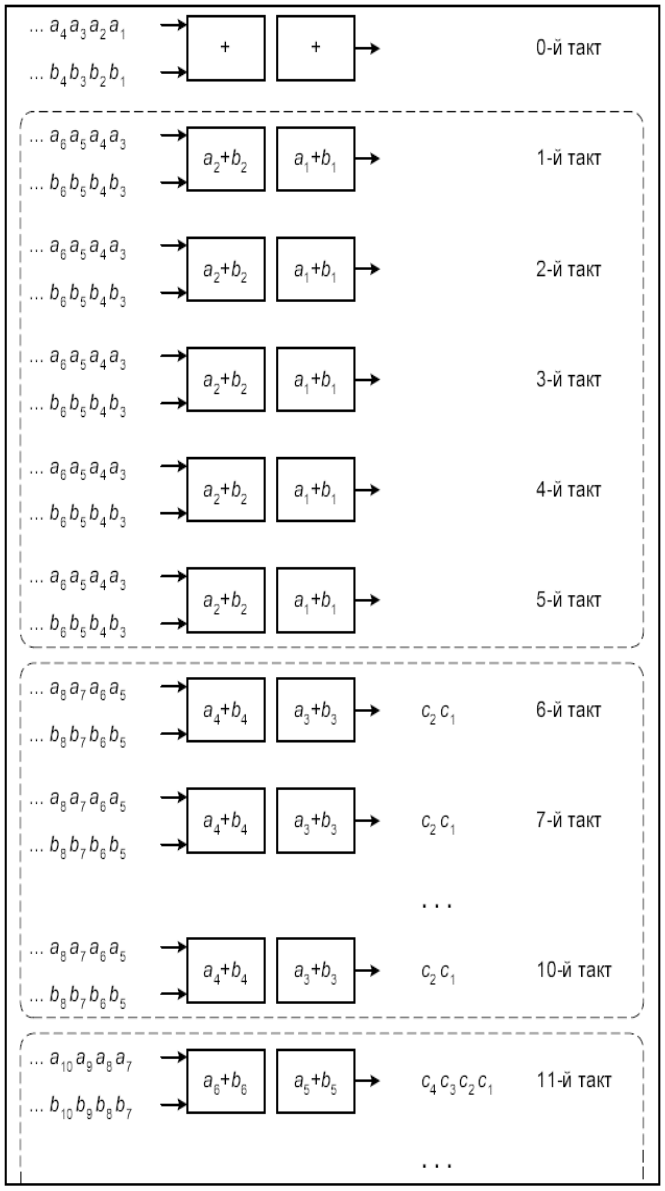


Рис. 2.4. Паралельне додавання векторів на двох однакових пристроях

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. 2.5), Pentium 4 — конвеєр з 35 стадій.

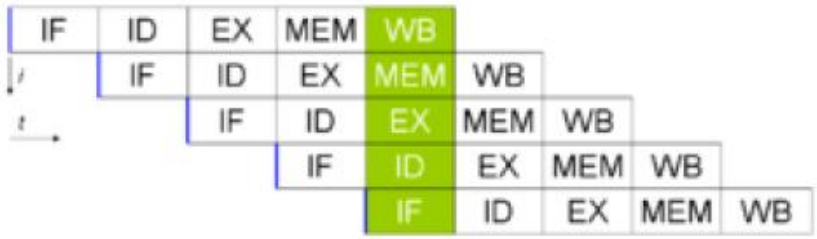


Рис. 2.5. Стандартний п'ятикроковий конвеєр в машині RISC.

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. 2.6). 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.

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 $\sigma+l+n-1$.

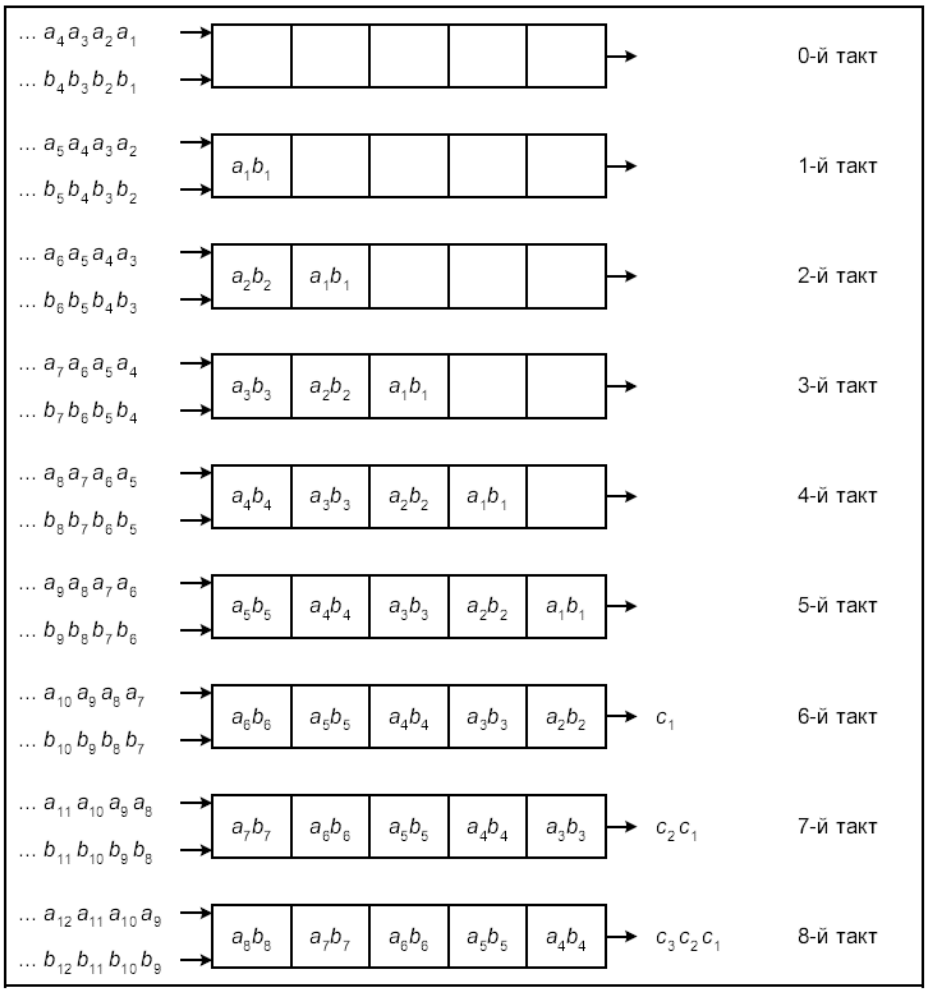


Рис. 2.6. Знаходження суми **c** = **a** + **b** за допомогою 5-стадійного конвеєрного пристрою

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:

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

де τ — це тривалість такту роботи комп'ютера.

На рис. 2.7 наведено приблизний вигляд графіка цієї залежності.

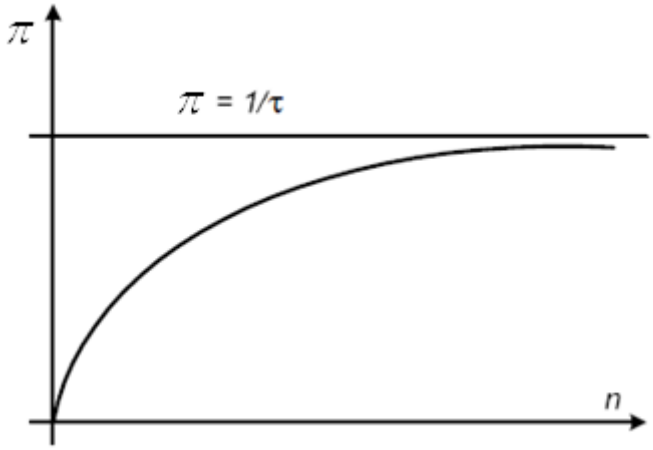


Рис. 2.7. Залежність продуктивності конвеєрного пристрою від довжини вхідного набору даних

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

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 $S=\lim_{n\to\infty}S(n)$ визначається за формулою

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

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 $\frac{1}{t_{\max}\cdot\tau}$.

Задача 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
   a. у послідовному режимі;
   b. 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.

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

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

Тому для завантаженості системи *p* виконуються нерівності $0 \leq p \leq 1$. 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

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

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,

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

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 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 [1]. *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* $\dfrac{l \cdot \min\limits_{1 \le i \le l} \pi_i}{\sum\limits_{i=1}^{l} \pi_i}$;
- *прискорення системи не перевищує* $\dfrac{l \cdot \min\limits_{1 \le i \le l} \pi_i}{\max\limits_{1 \le i \le l} \pi_i}$

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.

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* $(S_l < 1/\beta)$.

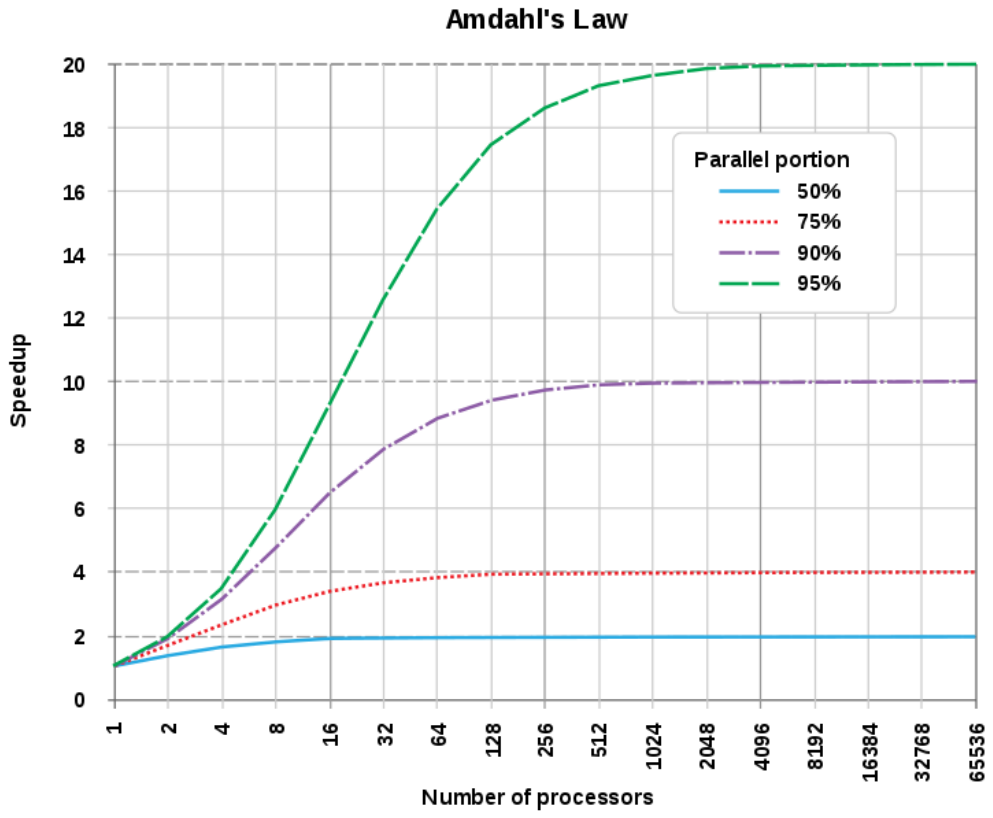


Рис. 2.8. Графік залежності прискорення від кількості пристроїв системи

Другий та третій закони Амдала проілюстровані на рис. 2.8.

Для системи, яка складається із l однакових простих пристроїв величина $E_l = \frac{S_l}{l}$ називається *ефективністю системи*. Легко переконатися, що для розглядуваних систем *ефективність співпадає із завантаженістю p*. Максимальні значення прискорення та ефективності — l та 1.

Задача 2.2. Граф системи ФП наведений на рис. 2.9. Відомі 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) Завантаженість системи.
3) Реальну продуктивність системи.
4) Прискорення системи.

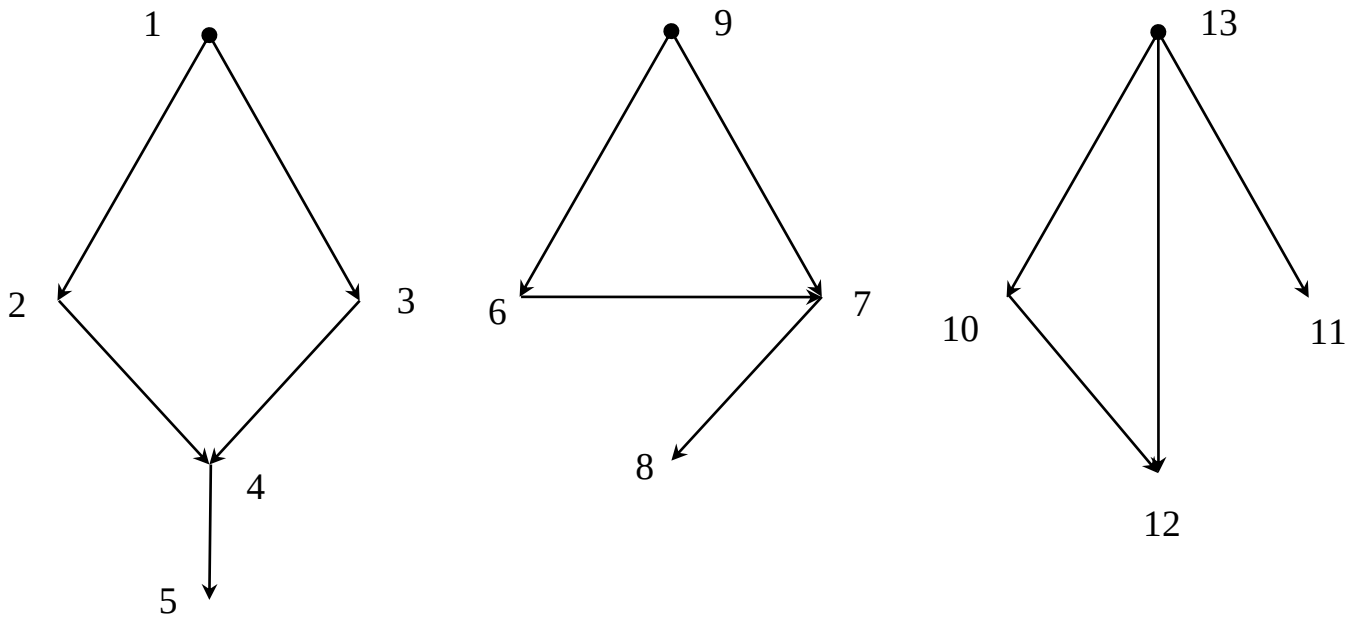


Рис. 2.9. Граф системи ФП

Задача 2.3. За допомогою 2-го закону Амдала визначити максимальне можливе прискорення і ефективність системи, яка складається з однакових пристроїв і призначена для реалізації алгоритму, граф якого наведений на рис. 2.10.

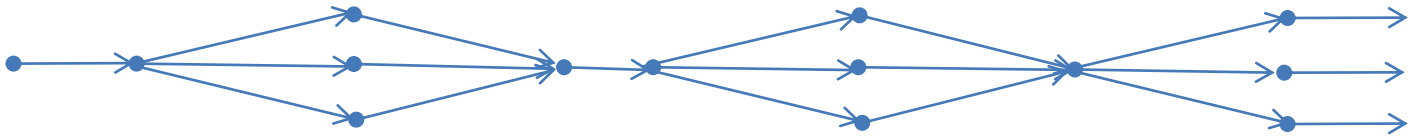


Рис. 2.10. Граф системи

Задача 2.4. Нехай у алгоритмі паралельні обчислення складають 2/3. Визначити:

1) Максимальне можливе прискорення у випадку використання 5 однакових універсальних процесорів;
2) Мінімальну кількість процесорів, використання яких може забезпечити 80% максимально можливого прискорення.

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. 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. 2.11 (ПР — процесор, ПД — пам'ять даних, УУ — управляючий пристрій).

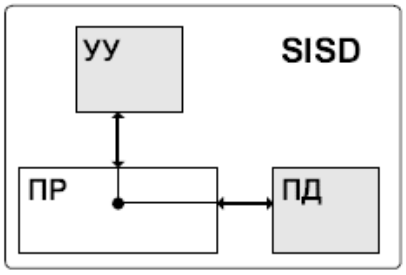


Рис. 2.11. Клас SISD класифікації Флінна

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. 2.12).

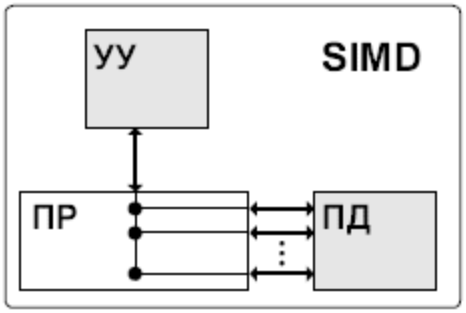


Рис. 2.12. Клас SIMD класифікації Флінна

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. 2.13).

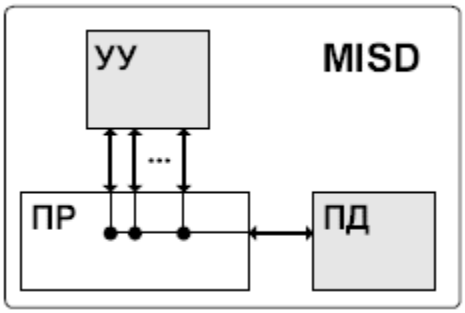


Рис. 2.13. Клас MISD класифікації Флінна

У визначенні мають на увазі, що наявність у архітектурі багатьох процесорів, які опрацьовують один і той самий потік даних. У [1] наведено аргументацію того, що даний клас потрібно вважати порожнім.

MIMD (Multiple Instruction stream / Multiple Data stream) — множинний потік команд та множинний потік даних (див. рис. 2.14).

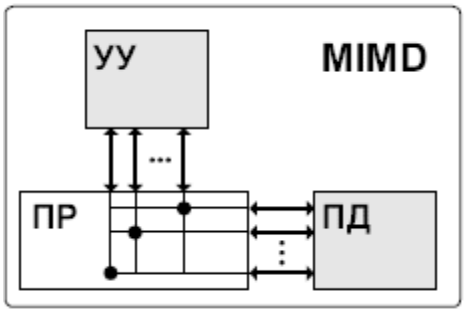


Рис. 2.14. Клас DISD класифікації Флінна

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 та багато інших. 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.

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.

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:

1) Розрядно-послідовні, послівно-послідовні $(n = 1, m = 1)$. У кожний момент часу на таких машинах обробляється тільки один двійковий розряд.
2) Розрядно-паралельні, послівно-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, послівно-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, послівно-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) та багато інших.

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 [1] 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].

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 [11]:

- 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*. 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 є простішим, 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. 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 $s \le n$, що усі вершини графа можна так помітити одним із індексів $1,\ldots,s$, що 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. Група вершин, які мають однакові індекси називається *ярусом* паралельної форми, а кількість вершин у групі — *шириною* ярусу. Кількість ярусів у паралельній формі називається *висотою* паралельної форми, максимальна ширина ярусу — її *шириною*. Відповідні "ботанічні" терміни застосовуються також і безпосередньо до алгоритмів чи програм.

Серед строгих паралельних форм існує форма, у якій вхідні вершини мають індекс 1 і кожна вершина індексу k знаходиться на відстані $k-1$ від деякої початкової вершини. Така форма графа називається *канонічною паралельною формою.* Для заданого графа його канонічна форма *єдина.*

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 містять одну вершину. Така паралельна форма називається лінійною.

Показано [1], що незалежно від того, яка паралельна форма алгоритму реалізується на комп'ютері, результат реалізації буде одним і тим самим.

Твердження 3.2. 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. Тогочасний швидкий розвиток елементної бази підказував дослідникам, що кількість обчислювальних пристроїв у системі невдовзі може стати дуже великим. Відповідна концепція отримала назву *концепції необмеженого паралелізму*.

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

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).

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 "найближче зверху" до α ціле число. 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* [1] (*каскадною схемою* [2]).

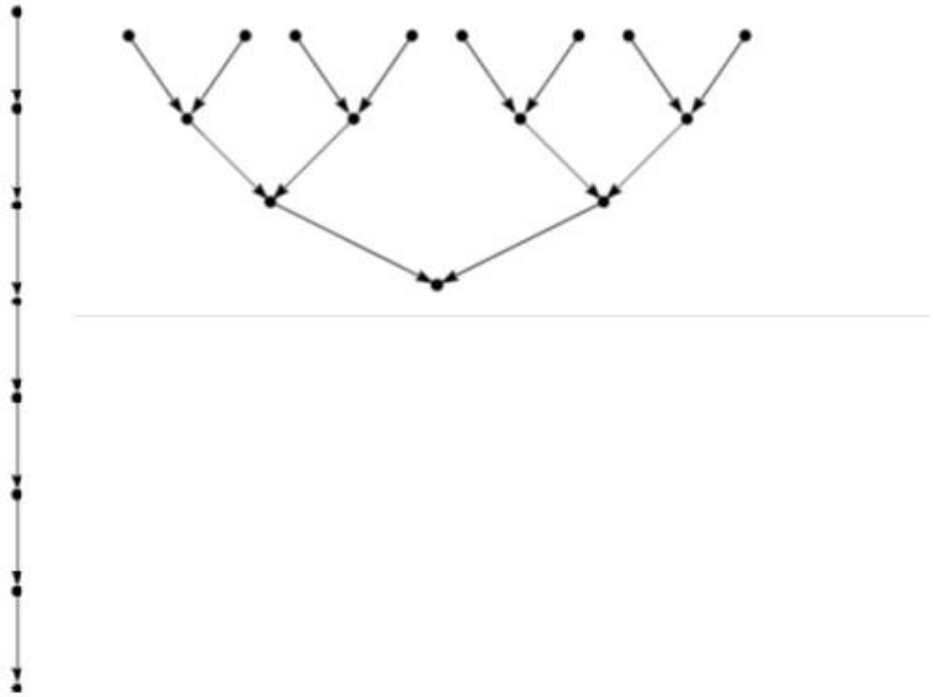


Рис. 3.1. Граф послідовного алгоритму та алгоритму "здвоєння"

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. 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.

Якщо система містить l однакових простих універсальних процесорів, то прискорення реалізації алгоритму на цій системі $S_l(n)$ можна обчислити за формулою [2]

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

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 $1 \le S_l(n) \le l$.

Ефективністю $E_l(n)$ реалізації алгоритму у паралельній системі (завантаженістю) називається відношення прискорення до кількості процесорів системи [2], 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. 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: $S_l(n) \sim l$, $E_l(n) \sim 1$.

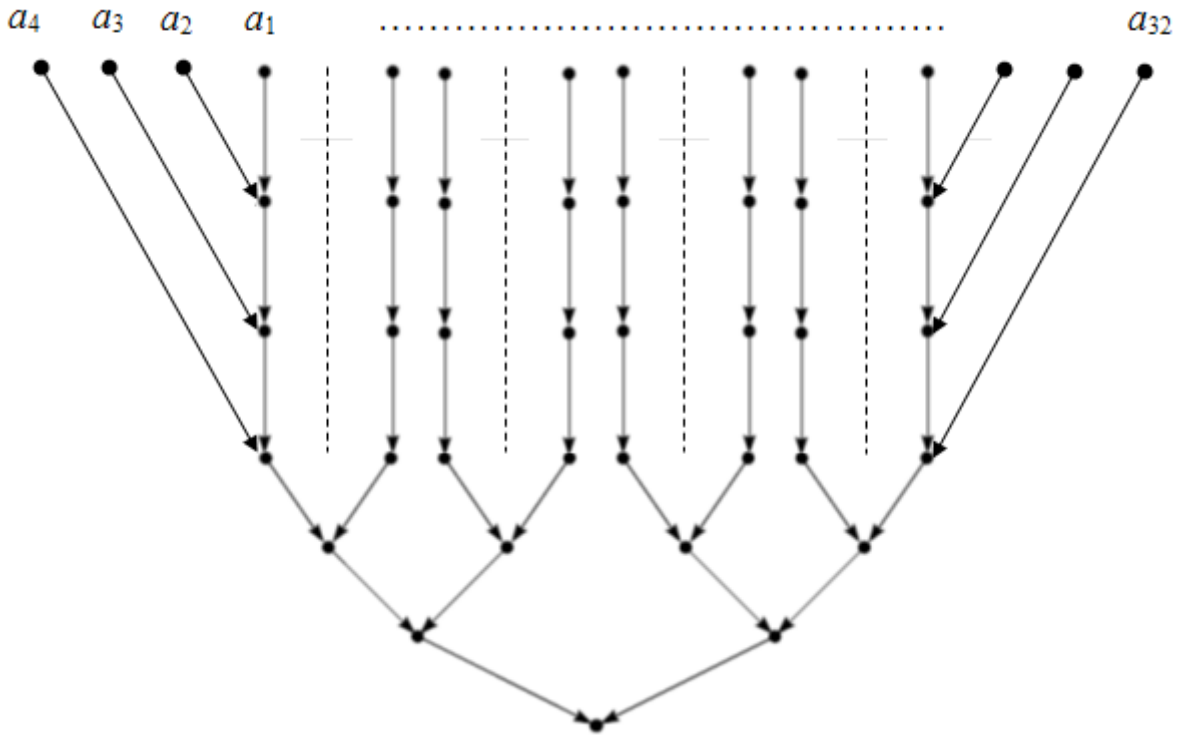


Рис. 3.2. Комбінований алгоритм у випадку n = 32, l = 8.

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 **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.

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.

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, 5].

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

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

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):

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

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

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

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 (i, j, k) буде передаватися результат операції, якій відповідає вершина $(i, j, k-1)$.

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. 3.3, *а*.

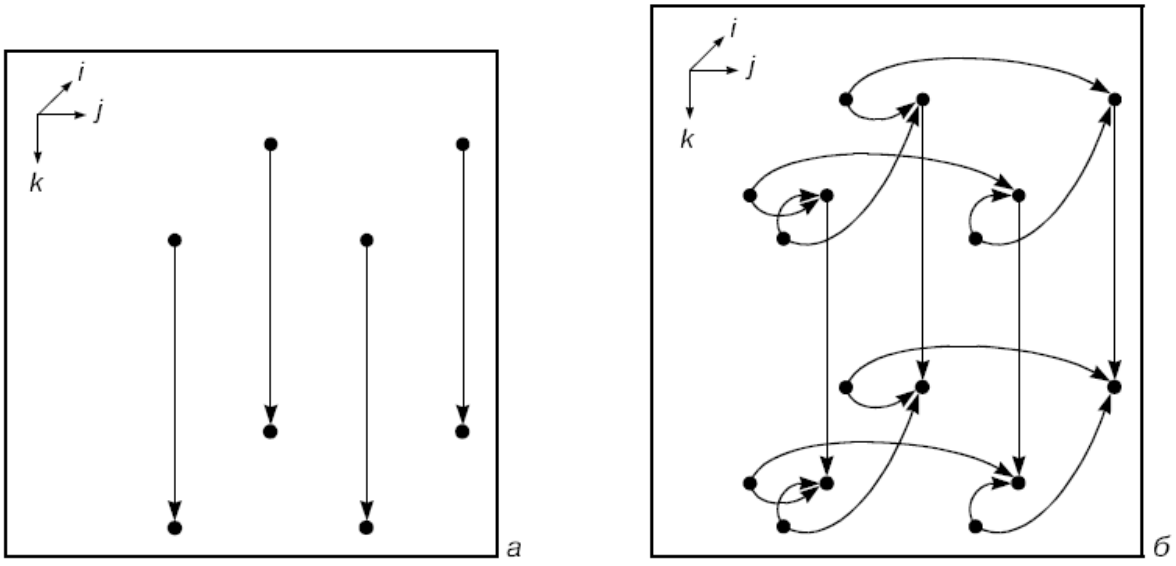


Рис. 3.3. Граф алгоритму множення матриць

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. 3.3, *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

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

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

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

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 для випадку $n = 5$ зображено на рис. 3.4. Верхня кутова вершина знаходиться у точці (1, 0).

На цьому рисунку зображена одна із максимальних паралельних форм. Її яруси помічені пунктиром. Ця паралельна форма стане канонічною, якщо вершини, відповідні за ввід елементів a_{ij}, розташувати у першому ярусі. Загальна кількість ярусів (без урахування вводу) рівна $n-1$.

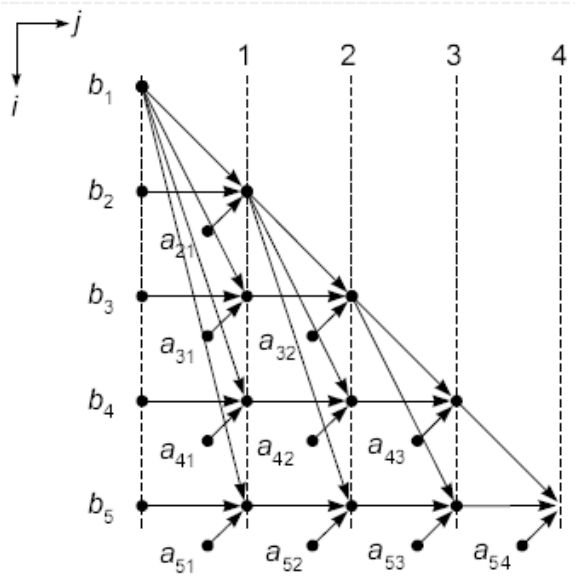


Рис. 3.4. Граф для алгоритму зворотної підстановки (9) для трикутної системи

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:

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

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

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

Відповідний граф для випадку $n = 5$ наведено на рис. 3.5. Верхня кутова вершина розташована у точці (1, 1).

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. 3.5 лежать на одному шляху, який позначений на рисунку пунктиром. Тому загальна кількість ярусів алгоритму (3.5), які містять операції вигляду $a - bc$, завжди рівна $1 + (1 + 2 + \ldots + n - 1) = \left(n^2 - n + 2\right)/2$, що набагато більше за число $n - 1$ — кількість ярусів для відповідних операцій у алгоритмі (3.4).

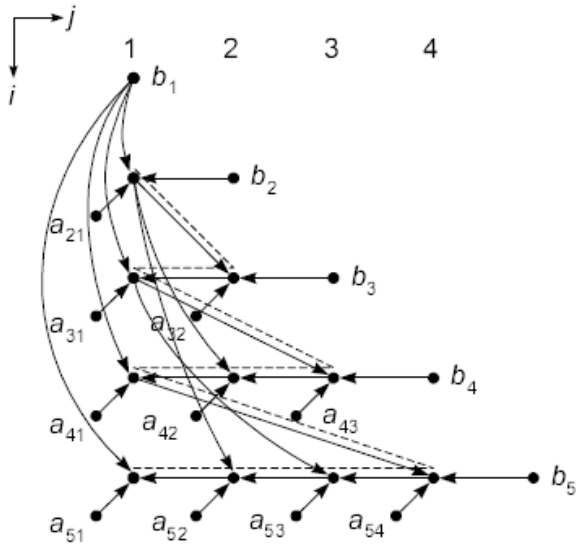


Рис. 3.5. Граф для алгоритму зворотної підстановки (10) для трикутної системи

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. 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 одно потокових 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.

4.1. 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".

4.1.1. Синхронізація

Багатопотоковість надає програмам можливість асинхронної поведінки. Проте 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 використовується механізм, який має назву "монітор". *Монітор* був розроблений Ч. Хоаром. Неформально можна сприймати монітор як дуже маленьку скриню, у яку в одиницю часу можна "помістити" лише один потік. 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.

4.1.2. 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 наступній таблиці

Таблиця 4.1. 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`

4.1.3. 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'єкту 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 мілісекунди) 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.

4.1.4. 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 is 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()`.

4.1.5. Створення нащадків класу Thread

Клас нащадок має *обов'язково* 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 [8] 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 метода

```
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 з використанням методу `getPriority()`:

```
final int getPriority()
```

4.1.6. Використання методів isAlive() та 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 методі `main` виникне двох секундне очікування завершення потоку `thread1`, і тільки потім почнеться виконання відповідного циклу `for`.

4.1.7. Синхронізація

Ключем до синхронізації є концепція монітора. Монітор — це об’єкт, який застосовується як взаємно виключне блокування (mutually exclusive lock — *mutex*), або *мьютекс*. 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 метода.

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 методу `call()` одного і того самого об’єкту `Callme`. Така ситуація називається "станом гонитви" (race condition).

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.

4.1.8. Оператор synchronized

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);
    }
}
```

4.1.9. 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'єкту*.
- Метод `notifyAll()` відновлює роботу усіх потоків, які викликали метод `wait()` того самого об'єкту. Одному з потоків надається доступ до об'єкта.

Додаткові форми методу `wait()` дозволяють вказувати час очікування.

Розглянемо приклад використання методів `wait()` та `notify()`. Розглянемо задачу "постачальник/споживач". 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 невірно, 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.

4.2. Потокова модель .NET Framework

Потокова модель .NET Framework має багато спільного із потоковою системою Java. Для обробки потоків потрібно використовувати засоби простору імен System.Threading.

У наступному прикладі продемонстровано паралельне виконання фонових потоків та передача параметрів при запуску потоків на виконання.

```
class Program
{
    static void Main(string[] args)
    {
        Thread t1 = new Thread(ThreadProc1);
        t1.IsBackground = true;
        t1.Start(7);
        Thread t2 = new Thread(ThreadProc2);
        t2.IsBackground = true;
        t2.Start(10);
        Console.WriteLine("Input any key to exit");
        Console.ReadKey();
    }

    public static void ThreadProc1(Object param)
    {
        for (int i = 0; i < (int)param; i++)
        {
            Console.WriteLine("Output from ThreadProc1");
            Thread.Sleep(3000);
        }
    }
    public static void ThreadProc2(Object param)
    {
        for (int i = 0; i < (int)param; i++){
            Console.WriteLine("Output from ThreadProc2");
            Thread.Sleep(1000);
        }
    }
}
```

4.2.1. Конкурентний доступ та блокування ресурсів

Для блокування ресурсів використовуються блоки коду `lock`, всередині яких розташовують той фрагмент коду, який має виконуватися для кожного потоку окремо. У наступному прикладі потоки спільно використовують масив `a`.

```
static int[] a = {1,2,3,4,5};
static void Main(string[] args)
{
    for (int i = 0; i < 5; i++){
        Thread t = new Thread(new ThreadStart(ThreadProc));
        t.Name = "Thread " + i;
        t.Start();
    }
    Console.WriteLine("Main proc");
}
```

```
public static void ThreadProc()
{
    lock (a)
    {
        for (int i = 0; i < 5; i++){
           Console.WriteLine($"{Thread.CurrentThread.Name};a[{i}]={a[i]}");
           Thread.Sleep(1000);
        }
    }
}
```

4.2.2. Потоки з використанням делегатів

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 метода, який викликається за допомогою делегата (асинхронно). Для досягнення цього потрібно передати відповідне значення у якості єдиного параметру методу `EndInvoke`.

Приклад.

```
public delegate int BinaryOp(int x, int y);
static int Add(int x, int y)
{
   Console.WriteLine("Add() invoked on thread {0}.",
    Thread.CurrentThread.ManagedThreadId);
    Thread.Sleep(5000);
    return x + y;
}

static void Main(string[] args)
{
    // Print out the ID of the executing thread.
    Console.WriteLine("Main() invoked on thread {0}.",
                      Thread.CurrentThread.ManagedThreadId);
    // Invoke Add() on a secondary thread.
    BinaryOp b = new BinaryOp(Add);
    IAsyncResult ar = b.BeginInvoke(10, 10, null, null);
    // Do other work on primary thread...
    Console.WriteLine("Doing more work in Main()!");
    // Obtain the result of the Add()
    // method when ready.
    int answer = b.EndInvoke(ar);
    Console.WriteLine("10 + 10 is {0}.", answer);
    Console.ReadLine();
```

```
}
```

Розглянемо приклад обчислення функції факторіал у окремому потоці:

```
public delegate int Factorial(int number);
static void CallBack(IAsyncResult asyncResult)
{
   string s = (string)asyncResult.AsyncState;
   Console.WriteLine("Asynchronous method finished\nParameter={0}", s);
}

static void Main(string[] args)
{
    Factorial factDelegate = new Factorial(ShowFactorial);
    IAsyncResult result = factDelegate.BeginInvoke(10, CallBack, "message");
    Console.WriteLine("Please wait. Calculations are proceeding...");
    //result.AsyncWaitHandle.WaitOne();
    Console.WriteLine("Calculations terminated. Output result is
        {0}",factDelegate.EndInvoke(result));
}

public static void ShowFactorial(int number)
{
    int fact = 1;
    for (int i = 2; i < number; i++)
    {
        fact *= i;
        Thread.Sleep(500);
    }
    Console.WriteLine("Result from thread: {0}", fact);
}
```

Приклад. Програма паралельного обчислення добутку матриці на вектор.

5. ТЕХНОЛОГІЯ OPENMP

5.1. 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
}
```

5.1.1. Модель паралельної програми

Розпаралелювання в OpenMP виконується за допомогою спеціальних директив, які додаються до тексту програми, а також виклику допоміжних функцій. 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.

5.1.2. Директиви та функції

Значна частина функціональності OpenMP реалізується за допомогою директив компілятора. Директиви OpenMP на мові C 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 (`clause`).

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);
}
```

5.2. 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` [3]:

```
#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 операцій); 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`.

```
#include <stdio.h>
int main()
{
    printf("Serial scope 1\n");
    #pragma omp parallel num_threads(5)
    {
        printf("Parallel scope\n");
    }
    printf("Serial scope 2\n");
}
```

Розглянемо приклад опції `reduction`. Підрахуємо загальну кількість потоків, які використовуються у паралельній області:

```
int main()
{
    int count = 0;
    #pragma omp parallel num_threads(5) reduction(+: count)
    {
        count++;
        printf("count is %d\n", count);
    }
    printf("Threads count is %d\n", count);
}
```

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, які виконуються у паралельній області, можна задати, вказавши значення змінної середовища `OMP_NUM_THREADS`.

5.2.1. Функції керування кількістю потоків

Функція `omp_get_num_procs()` повертає кількість процесорів, доступних для використання на момент виклику. У процесі функціонування програми можна дізнатися чи змінити кількість потоків за допомогою наступних функцій:

Таблиця 5.1. Функції керування кількістю потоків

Функція	Опис
`void omp_set_num_threads(int num)`	Встановлює кількість потоків. Дія 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 задана функцією `omp_set_num_threads`, то повертається це число, інакше — кількість логічних процесорів
`int omp_get_thread_num()`	Повертає номер поточного потоку

Приклад. Демонстрація функції `omp_set_num_threads`.

```
int main()
{
    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. 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).

```
int main()
{
    #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 num is %d\n",
omp_get_max_threads());
            printf("Threads num is %d\n",
omp_get_num_threads());
        }
    }
}
```

Можливий вивід:

```
Current thread number is 0
Current thread number is 1
Current thread number is 6
Max threads num is 2
Current thread number is 2
Current thread number is 3
Current thread number is 5
Current thread number is 7
Threads num is 8
Current thread number is 4
```

5.2.2. 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.

```
    #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.

```
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.

Приклад.

```
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);
}
```

5.3. 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`). Змінні, визначені всередині паралельної області, за замовчуванням є локальними (`private`). Режим доступу також можна явно вказати з використанням опцій `private`, `shared`, `firstprivate` в директивах OpenMP.

Приклад.

```
int main()
{
    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);
}
```

5.4. 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. 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("value c[i] = %d is obtained in the 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. 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;
```

```
    for(j = 1; j <= N/2; j *= 2){// j is difference
between summand
        #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';
```

5.4.1. 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 5.1.2. 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. 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);
```

5.4.2. 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 розподілений ітерацій, поділеному на кількість потоків. Розмір початкового блоку залежить від реалізації.

На рис. 5.1–5.3 наведено діаграму розподілу навантаження між потокам.

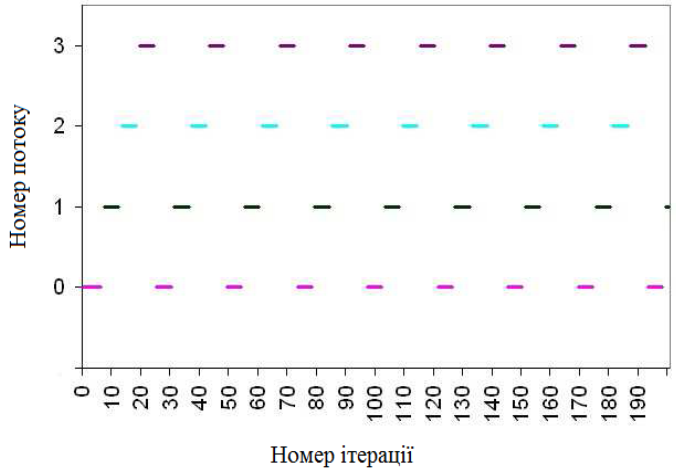


Рис. 5.1. Розподіл ітерацій в режимі `(static, 6)`

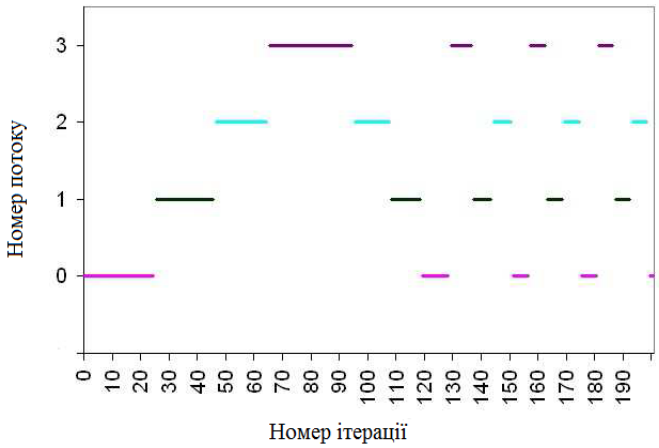


Рис. 5.2. Ітерації в режимі `(guided, 6)`

На рис. 5.2. видно, що в режимі `guided` розмір порції зменшується від 24 до 6.

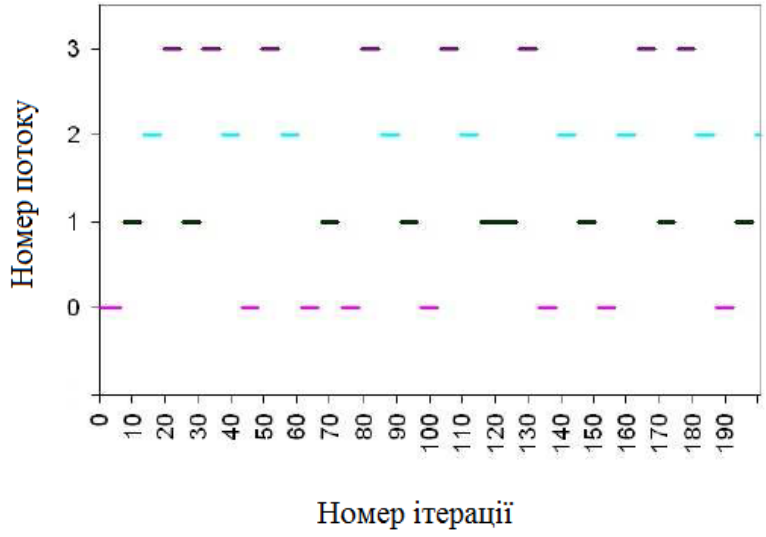


Рис. 5.3. Розподіл ітерацій в режимі `(dynamic, 6)`

5.5. 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);
}
```

Розглянемо опцію `lastprivate(список)` — змінним, наведеним у списку, присвоюється результат, отриманий у останній секції;

У наступному прикладі змінна `n` описана як `lastprivate`. Три потоки, які виконують секції, присвоюють своїй локальній копії `n` різні значення. Після виходу з області `sections` значення `n` з останньої секції присвоюється локальним копіям у всіх потоках, тому всі потоки надрукують число 3. Це ж значення збережеться для змінної `n` і в послідовній області.

```
int n = 0;
#pragma omp parallel num_threads(4)
{
    #pragma omp sections lastprivate(n)
    {
        #pragma omp section
        {
            n = 1;
        }
        #pragma omp section
        {
            n = 2;
        }
        #pragma omp section
        {
            n = 3;
        }
    }
    printf("Value of n for thread %d: %d\n",
omp_get_thread_num(), n);
}
printf("Value of n in the serial scope: %d\n", n);
```

Приклад. 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, 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
    r2 = a[5]*a[6];
}
r1 += r3;
r = r1 + r2;
printf("r = %f\n", r);
```

5.6. Синхронізація

Цілий набір директив в OpenMP призначений для синхронізації роботи потоків.

5.6.1. Бар'єр

Найпоширеніший спосіб синхронізації в OpenMP — бар'єр, який оформляється 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`.

5.6.2. 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, повинна 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 is directive here
          printf("Ordered: thread %d, iteration %d\n", n, i+1);
    }
}
```

5.6.3. 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. Всі 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 кожна нитка працювала б 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.

```
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);
```

5.6.4. 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`.

Дана *директива відноситься 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`.

5.6.5. 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 (nesting count). 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.

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);
```

5.6.6. 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`.

5.7. Приклади використання OpenMP

Приклад 1. Обчислення числа π. Скористаємося формулою

$$\pi / 4 = \text{arctg}1 = \int_0^1 \frac{1}{1+x^2}\,dx\,.$$

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 is %e. Duration of serial computation: %e\n", pi,
finish-start);
    start = omp_get_wtime();
    pi = pi_calc_parallel(n);
    finish = omp_get_wtime();
    printf("Pi is %e. Duration of parallel computation: %e\n", pi,
finish-start);
}
```

Приклад 2. Обчислення добутку двох матриць.

```
double a[N][N], b[N][N], c[N][N];
```

```
int main()
{
    int i, j, k;
    double t1, t2;
    // initialization
    for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    a[i][j] = b[i][j] = i*j;
    t1 = omp_get_wtime();
    // calculation
    #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];
        }
    }
    t2 = omp_get_wtime();
    printf("Time = %lf\n", t2-t1);
}
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 608 с.
2. Качко Е. Г. Параллельное программирование: Учебное пособие. — Харьков: "Форт", 2011. — 528 с.
3. Антонов А. С. Параллельное программирование с использованием технологии OpenMP. — М. Изд-во МГУ, 2009. — 77 с.
4. Воеводин В. В. Математические основы параллельных вычислений. – М.: МГУ, 1991. – 345 с.
5. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ, 2-е издание. – М.: "Вильямс", 2005. — 1296 с.
6. Шилдт Г. Java. Полное руководство. — М.: ООО "И.Д. Вильямс", 2012. — 1104 с.
7. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0, 5-е изд. — М.: "Вильямс", 2011. — 1392 с.
8. Оленев Н.Основы параллельного программирования в системе MPI. — М.: Вычислительный центр им. А.А. Дородицына РАН, 2005 — 81 с.
9. Сердюк Ю. П. Введение в параллельное программирование на языке MC#. Переславль-Залесский: Институт программных систем РАН, 2007. — 51с.
10. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. — 232 с.

Інформаційні ресурси

11. OpenMP Architecture Review Board (**http://www.openmp.org/**)
12. http://www.gridforum.org
13. http://www.mpiforum.org
14. http://parallel.ru/tech/tech_dev/OpenMP/examples/