ЗМІСТ

ВСТУП

Навчальна дисципліна "Теорія паралельних обчислень" вивчається студентами спеціальності "Програмне забезпечення систем" у IX семестрі. Актуальність вивчення основних понять теорії паралельних обчислень розробниками програмного забезпечення зумовлена наявністю значної кількості практично важливих задач великої розмірності, розв’язання яких потребує паралельного задіювання великої кількості обчислювальних ресурсів [1]. Тому важливою складовою процесу підготовки фахівців сфери IT є вироблення знань та навичок, які стосуються розуміння та використання сучасних методів паралельного програмування.

Посібник складається із чотирьох розділів. У першому розділі наведено основні означення теорії паралельних обчислень, розглянуто паралельну обробку даних і основні характеристики паралельних систем, описано найбільш популярні підходи до класифікації паралельних обчислювальних систем. Завершується розділ описом принципів функціонування та оглядом можливостей глобальної технології GRID. У другому розділі розглянуто основні моделі, які використовуються для опису паралельних систем і використовують поняття графа алгоритму, описано концепцію необмеженого паралелізму та наведено приклади виявлення використання внутрішнього паралелізму у "звичайних" послідовних алгоритмах. У третьому розділі розглянуто програмування з використанням потоків у середовищах Java та .NET Framework. У четвертому розділі наведено завдання для лабораторних робіт.

У посібник не увійшов матеріал, який стосується паралельного програмування у системах MPI та OpenMP. Опис та приклади використання цих систем можна знайти в [2-4] або на ресурсах [10, 12].

1. ПАРАЛЕЛЬНІ ОБЧИСЛЮВАЛЬНІ СИСТЕМИ

1.1. Способи обробки даних в обчислювальних системах

1.1.1. 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. 1.

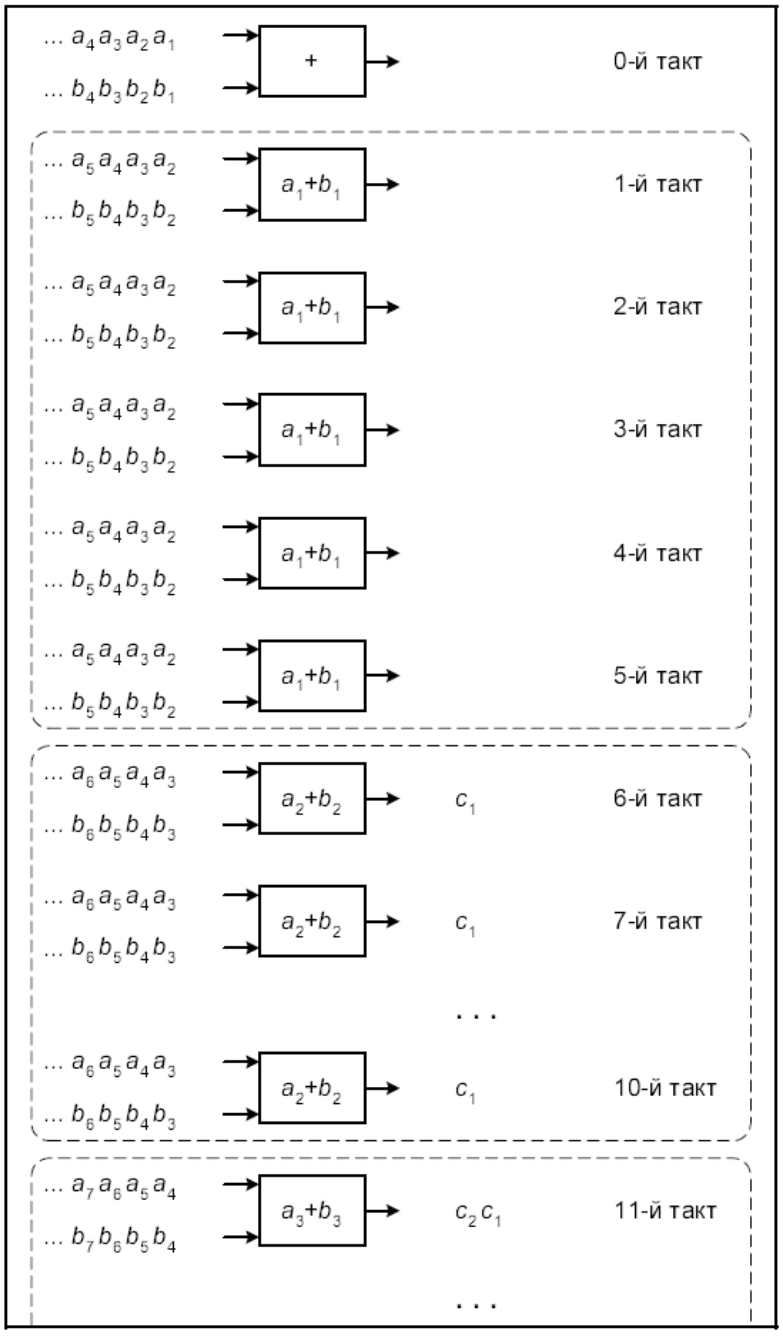

Рис. 1. Додавання векторів на послідовному пристрої з 5-тактовою операцією додавання

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) — тобто маємо подвійне прискорення.

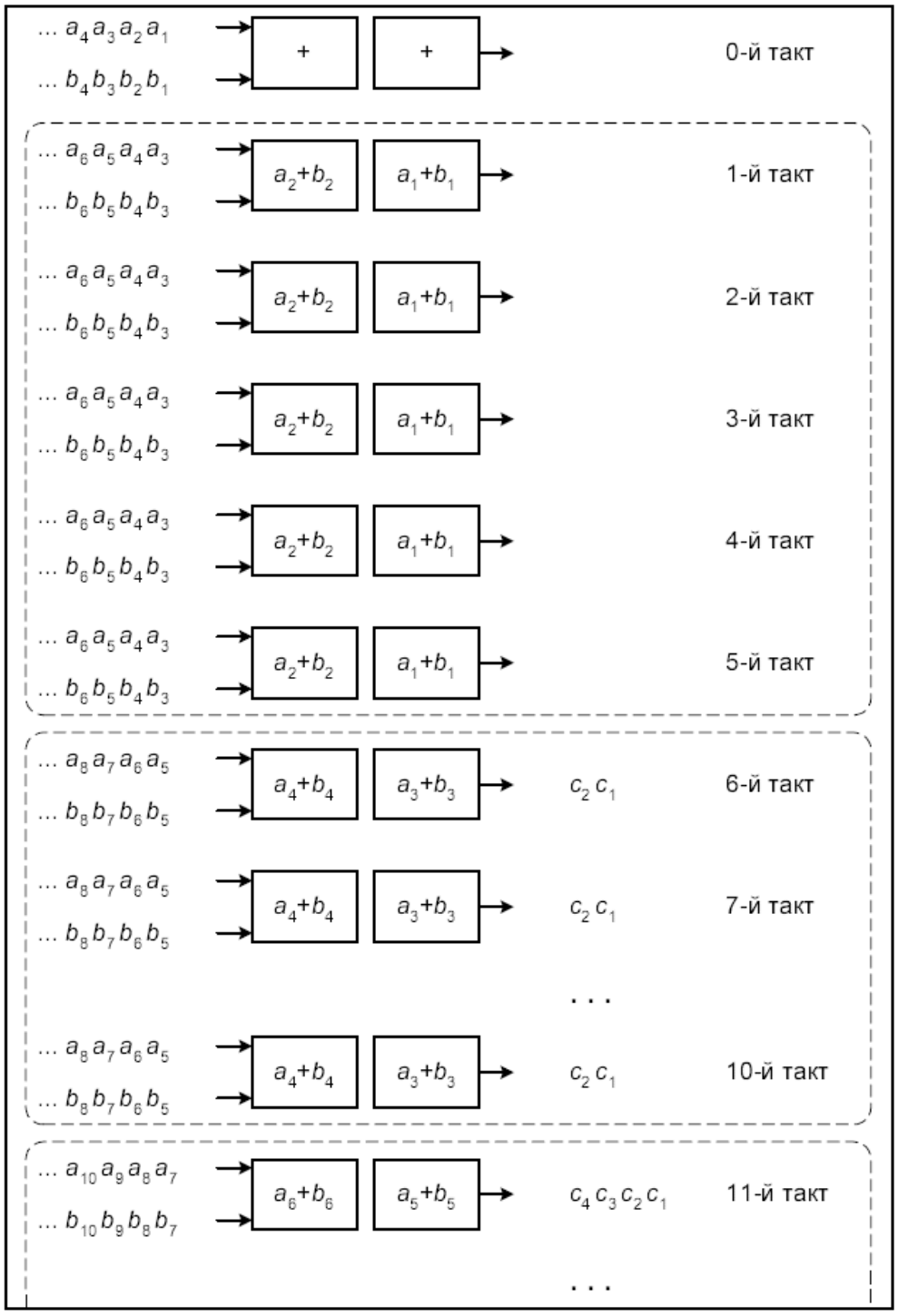

Рис. 2. Паралельне додавання векторів на двох однакових пристроях з 5-тактовою операцією додавання

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.

1.1.2. 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, нормалізацією 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*.

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

Приблизно така сама ситуація буде і у загальному випадку. Якщо конвеєрний пристрій є l-стадійним і обробка даних на кожній стадія триває одиницю часу, то для виконання n послідовних операцій на цьому пристрої потрібно витратити $l+n-1$ одиниць часу. Якщо ж цей пристрій використовувати у послідовному режимі, то час роботи буде рівним $l \times n$. Отже, для великих n отримуємо "прискорення" майже у l разів за рахунок використання конвеєрної обробки даних.

При використанні векторних команд у формулі для часу обробки даних на конвеєрному пристрої додається ще один доданок: $\sigma + l + n - 1$, де σ — це час, необхідний для ініціалізації векторної команди.

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, то залежність продуктивності від довжини вхідних векторів визначається наступним співвідношенням:

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

де τ — це час такту роботи комп'ютера.

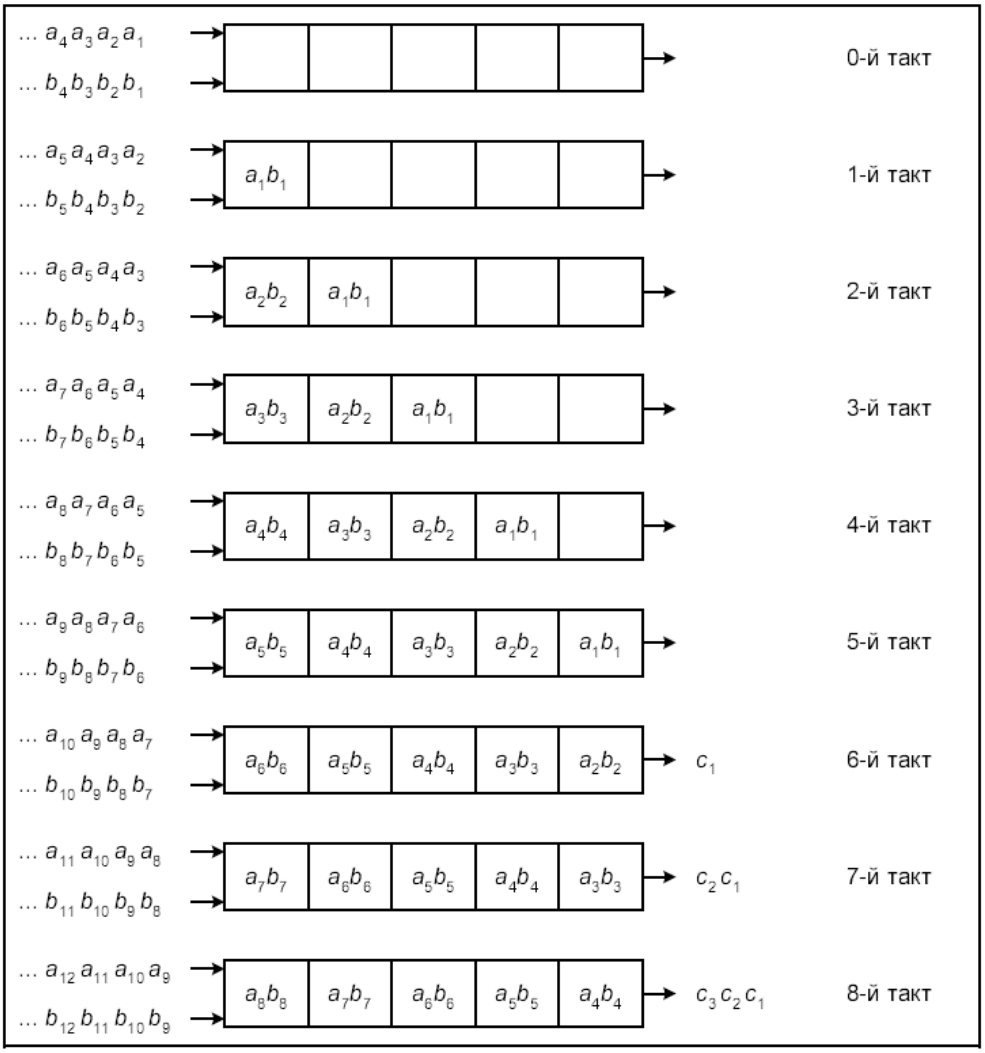

Рис. 3. Знаходження суми **c** = **a** + **b** за допомогою 5-стадійного конвеєрного пристрою

На рис. 4 наведено приблизний вигляд графіка цієї залежності.

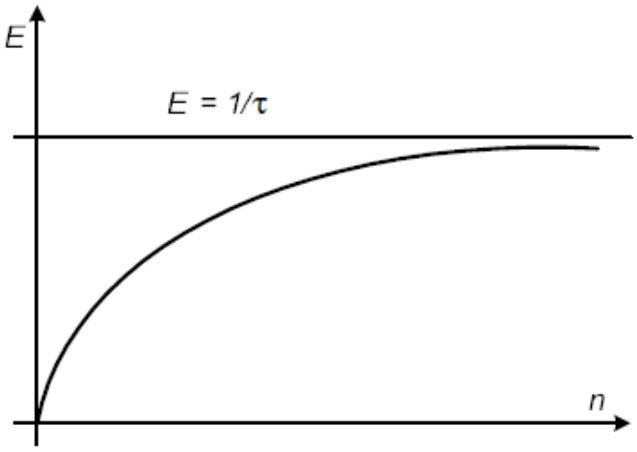

Рис. 4. Залежність продуктивності конвеєрного пристрою від довжини вхідного набору даних

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

1.2. 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 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.

Твердження 2. Якщо система складається із s пристроїв, які мають пікові продуктивності $\pi_1,\ldots,\pi_s$ і працюють із завантаженістю $p_1,\ldots,p_s$, то реальна продуктивність системи r обчислюється за формулою

$$r = \sum_{i=1}^{s} p_i \pi_i . \tag{1}$$

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

$$p = \sum_{i=1}^{s} \alpha_i p_i , \tag{2}$$

де

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

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

$$\sum_{i=1}^{s} \alpha_i = 1,\ \alpha_i \geq 0, \left(i = 1,\ldots,s\right) \tag{3}$$

Тому для завантаженості системи p виконуються нерівності $0 \leq p \leq 1$. Крім того, з (2) та (3) випливає, що завантаженість системи рівна 1 тоді і тільки тоді, коли завантаженості окремих пристроїв системи рівні 1.

Також слід зазначити, що визначення завантаженості системи згідно до (2) узгоджується із формулою реальної продуктивності (1). Дійсно, оскільки пікова продуктивність π системи пристроїв рівна $\pi_1 + \ldots + \pi_s$, то згідно до (1) та (2) справджується рівність

$$r = p\pi . \tag{4}$$

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.

Нехай алгоритм реалізується за час T на обчислювальній системі із s пристроїв, простих або конвеєрних, які мають пікові продуктивності $\pi_1,\ldots,\pi_s$. Припустимо, що $\pi_1 \leq \pi_2 \leq \ldots \leq \pi_s$ і нехай при реалізації алгоритму

система досягає реальної продуктивності r, яка обчислюється за формулою (1). Будемо порівнювати швидкість роботи системи із швидкістю роботи гіпотетичного простого універсального пристрою, який має пікову продуктивність π_s (таку саму, як найшвидший пристрій системи) та здатний виконувати ті самі операції, що й усі ФП системи.

Таким чином, будемо називати відношення $R = r / \pi_s$ прискоренням реалізації алгоритму на даній обчислювальній системі або просто *прискоренням*. Тобто,

$$R = \frac{\sum_{i=1}^{s} p_i \pi_i}{\max\limits_{1 \le i \le s} \pi_i}. \qquad (5)$$

Аналіз формули (5) показує, що прискорення обчислювальної системи, яка складається із s пристроїв, не може перевищувати s і може досягати s тоді і тільки тоді, коли усі пристрої системи мають однакові пікові продуктивності і є цілком завантаженими.

Твердження 3. Якщо система складається із s пристроїв, які мають однакові пікові продуктивності, то

- завантаженість системи рівна середньому арифметичному завантаженості усіх пристроїв;
- реальна продуктивність системи рівна сумі реальної продуктивності усіх пристроїв;
- пікова продуктивність системи у s 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 (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 (можливо із кратними дугами), 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 *графом системи*.

Твердження 4. Якщо система складається із s простих пристроїв, які мають пікові продуктивності $\pi_1,\ldots,\pi_s$, і граф системи є зв'язним, то максимальна продуктивність системи $r_{\max}$ виражається формулою

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

Доведення наведено в [1].

Наслідок 1. В умовах твердження 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

$$p_{\max} = \frac{s \cdot \min\limits_{1 \le i \le s} \pi_i}{\sum\limits_{i=1}^{s} \pi_i};$$

- прискорення системи не перевищує

$$R_{\max} = \frac{s \cdot \min\limits_{1 \le i \le s} \pi_i}{\max\limits_{1 \le i \le s} \pi_i}$$

Наслідок 2 (перший закон Амдала). Продуктивність обчислювальної системи, яка складається із пов'язаних між собою пристроїв, у загальному випадку визначається найменш продуктивним пристроєм.

Наслідок 3. Нехай система утворена простими пристроями і має зв'язний граф. Тоді асимптотична продуктивність системи буде найбільшою, якщо усі пристрої мають однакові пікові продуктивності.

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 рівна m, ширина — q, а всього у алгоритмі виконується N операцій.

Твердження 5. Для системи, яка задовольняє наведені вище умови, максимальне прискорення рівне N / m.

Наслідок 1. Мінімальна кількість пристроїв системи, при якій може бути досягнуто максимально можливе прискорення, рівне 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 (другий закон Амдала). Нехай система складається із s однакових простих універсальних пристроїв. Припустимо, що при виконанні паралельної частини алгоритму усі s пристроїв є цілком завантаженими. Тоді максимальне можливе прискорення рівне

$$R = \frac{s}{\beta s + (1 - \beta)}.$$

Наслідок 3 (третій закон Амдала). Нехай система складається із s однакових простих універсальних пристроїв. При будь-якому режимі роботи її прискорення не може бути більшим за обернену величину до частки послідовних обчислень.

1.3. 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.

1.3.1. 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. 5 (ПР — процесор, ПД — пам'ять даних, УУ — управляючий пристрій).

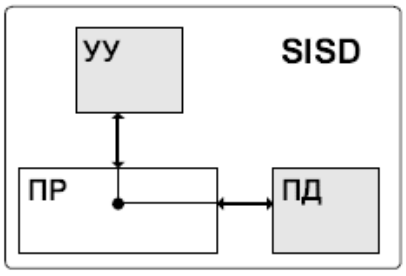

Рис. 5. Клас SISD класифікації Флінна

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. 6).

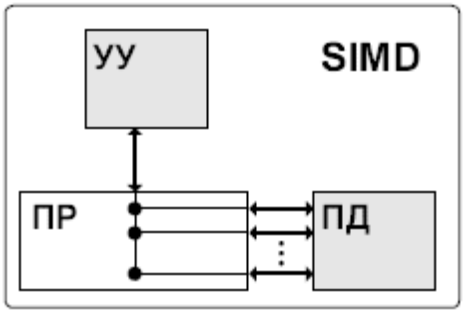

Рис. 6. Клас SIMD класифікації Флінна

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. 7).

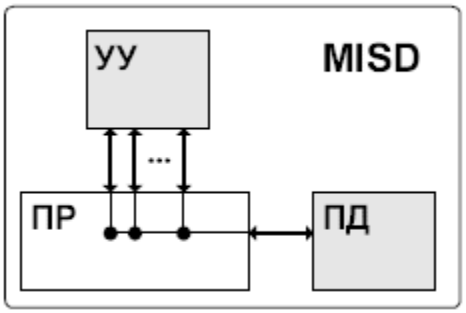

Рис. 7. Клас MISD класифікації Флінна

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. 8).

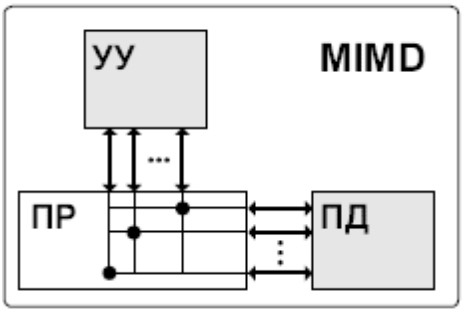

Рис. 8. Клас DISD класифікації Флінна

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].

1.3.2. 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.

1.4. 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. Користувача 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.

На відміну від традиційного комп'ютера 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.

2. ОСНОВНІ ПОНЯТТЯ ТЕОРІЇ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

2.1. 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 навіть 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 недермінованого 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 циклічним мультиграфом. Його ациклічність випливає із того, що у довільних програмах реалізовують лише явні обчислення і ніяка величина не визначається сама через себе.

Твердження 6. 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 5, 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, а довжина усіх шляхів від початкових вершин до вершин індексу k рівна $k-1$. 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 7. 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.

2.2. 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. Розглянемо результати, отримані на її основі.

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".

2.2.1. 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. 9 (початкові вершини символізують ввід даних).

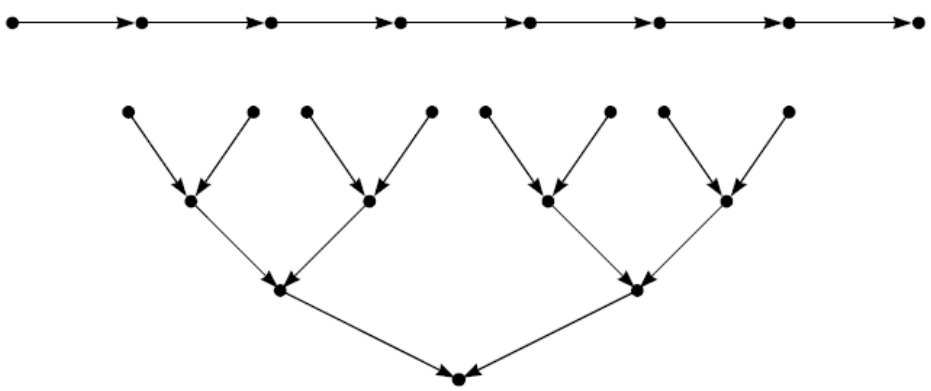

Рис. 9. Послідовний граф та граф алгоритму "здвоєння"

Остання схема очевидним чином розповсюджується на випадок довільного *n*. Для її реалізації потрібно на кожному ярусі виконувати максимально можливу кількість множень пар чисел, які отримані на попередньому ярусі (і не мають спільних множників). В загальному випадку висота паралельної форми рівна $\lceil \log_2 n \rceil$, $\lceil \alpha \rceil$ означає "найближче зверху" до α ціле число. Ця паралельна форма може бути реалізована на $\left\lfloor \frac{n}{2} \right\rfloor$ процесорах, причому ступінь завантаженості процесорів зменшується від ярусу до ярусу. Процес побудови елементів кожного ярусу називається процесом *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 8. 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 *s* — висота алгоритму (без урахуванням вводу даних), то $s \geq \log_p n$.

Якщо деяка задача має *n* вхідних даних, то можна в загальному випадку можна розраховувати на існування алгоритму її розв'язання із висотою не більше, ніж $\log n$. Якщо вдається розробити алгоритм висоти $\log^{\alpha} n$, де $\alpha > 0,$ то такий алгоритм можна вважати достатньо ефективним з точки зору його реалізації на паралельній обчислювальній системі.

2.2.2. 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 $\lceil \log_2 n \rceil$. При цьому процесори використовуються нерівномірно. Усі процесори задіяні тільки на першому кроці. Далі кількість працюючих процесорів зменшується удвічі на кожному кроці.

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.

2.2.3. 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 типові 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 су та добутків чисел.

2.3. 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 2.1. 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 7, паралельна реалізація буде мати такі самі обчислювальні властивості, що й звичайна. Зокрема, чисельно стійкий початковий алгоритм зберігає цю властивість і у паралельній формі. 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.

2.3.1. 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

$$a_{ij} = \sum_{k=1}^{n} b_{ik} c_{kj}, \quad (i, j = 1, \ldots, n). \tag{6}$$

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 (6) є еквівалентними і приводять до одного і того самого результату. Нехай вибрано наступний алгоритм реалізації формул (6):

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

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} + b_{ik} c_{kj}, \quad (i, j, k = 1, \ldots, n), \tag{7}$$

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

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* (як це випливає із формули (7)).

Побудуємо тепер граф алгоритму (7). При побудові графа не будимо враховувати природу елементів матриць *A*, *B* та *C* (можна вважати їх елементами одного і того самого кільця). Вершини графа не можна розташовувати довільним чином. Прийнятний спосіб розташування підказує сам вигляд формул (7). Елементи графу будемо розташовувати у вузлах прямокутної ґратки у тривимірному просторі. Аналізуючи формулу (7) 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. 10, *а*.

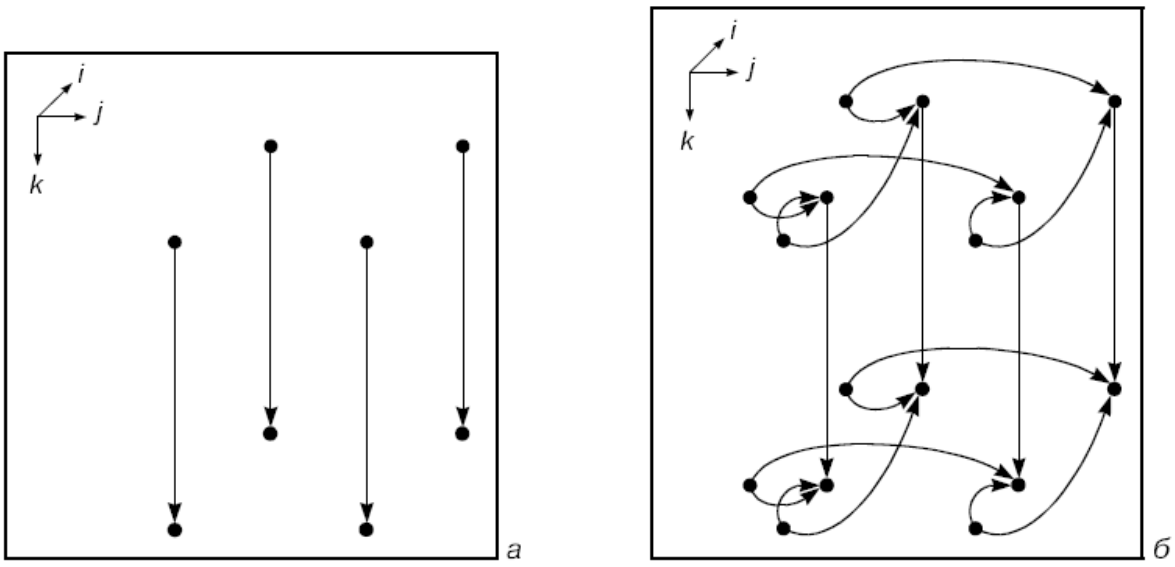

Рис. 10. Граф алгоритму множення матриць

У повному графі присутня множинна розсилка даних. Елемент b_{ik} розсилається по усім вершинам, які мають ті самі значення координат *i* та *k*. Для випадку $n = 2$ відповідні розсилки елементів матриць *B* та *C* наведені на рис. 10, *б*. Наведений приклад також демонструє, як важливо правильно розташовувати вершини графа.

Слід зауважити, що якщо додавання у (6) виконується за схемою здвоєння, то кожний вертикальний ланцюг у графі має бути замінений на дерево, наведене на рис. 9.

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

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

$$x_1 = b_1,\ \ x_i = b_i - \sum_{j=1}^{i-1} a_{ij} x_j,\ \left(2 \le i \le n\right). \tag{8}$$

Цей запис також не визначає алгоритм однозначно, бо не вказано порядок обчислення сум. Розглянемо наступне уточнення процесу (8):

$$x_i^{(0)} = b_i,$$

$$x_i^{(j)} = x_i^{(j-1)} - a_{ij} x_j^{(j-1)},\ \ i = 1,2,\ldots,n,\ j = 1,2,\ldots,i-1, \tag{9}$$

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

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. 11. Верхня кутова вершина відповідає знаходиться у точці (1, 0).

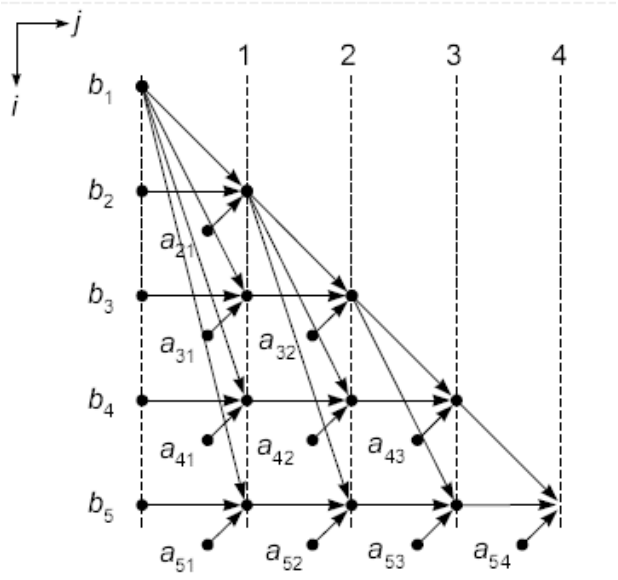

Рис. 11. Граф для алгоритму зворотної підстановки (9) для трикутної системи

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 $n-1$.

Вибір зростаючого по j напрямку додавання у (8), який призвів до алгоритму (9), був зроблений, взагалі кажучи, випадково.

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

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

$$x_i^{(j)} = x_i^{(j+1)} - a_{ij}x_j^{(j)}, \quad i = 1,2,\ldots,n, \; j = 1,2,\ldots,i-1, \tag{10}$$

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

Відповідний граф для випадку $n = 5$ наведено на рис. 12. Тепер верхня кутова вершина розташована у точці $(1, 1)$.

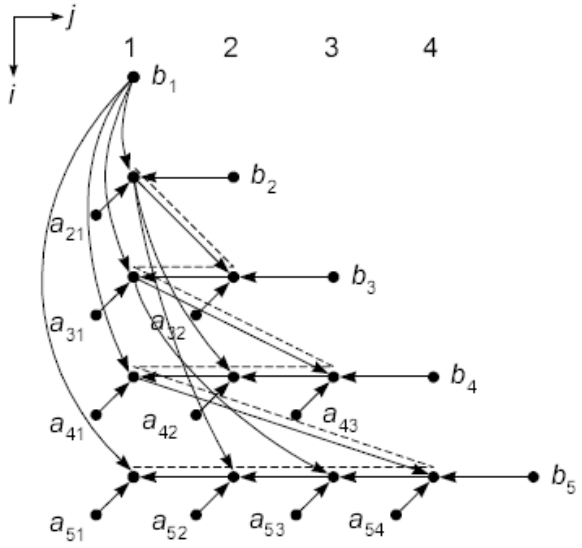

Рис. 12. Граф для алгоритму зворотної підстановки (10) для трикутної системи

Пробуючи розташувати вершини, які відповідають операціям $a - bc$, за ярусами хоча б однієї паралельної форми, приходимо до висновку, що тепер у кожному ярусі завжди може знаходитися тільки одна вершина. Цей факт пояснюється тим, що усі вершини графа на рис. 12 лежать на одному шляху, який позначений на рисунку пунктиром. Тому загальна кількість ярусів алгоритму (10), які містять операції вигляду $a - bc$, завжди рівна $\left(n^2 - n + 2\right)/2$, що набагато більше за число $n-1$ — кількість ярусів для відповідних операцій у алгоритмі (9).

Отриманий результат є досить несподіваним. Обидва алгоритми (9) та (10) призначені для розв'язування тієї самої задачі і розроблені на основі формул (8). 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 (9) можна реалізувати за час, пропорційний n, а алгоритм (10) — 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.

3. 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 Java та C# присутня 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.

3.1. Потокова модель у 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 по відношенню до 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 операційній системі Windows ці потоки ділять між собою час процесора. У інших операційних системах потоки повинні примусово передавати керування своїм "родичам".

3.1.1. 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 добре відомий із теорії міжпроцесної синхронізації 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.

3.1.2. 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, які допомагають керувати потоками. Деякі з них наведені у наступній таблиці

Таблиця 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`

3.1.3. 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, так як виконує різноманітні дії.

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 милісекунди) 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. Група потоку — структура даних, яка керує набором потоків у цілому.

3.1.4. 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 метод `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()`.

3.1.5. 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 літературі по Java [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()
```

3.1.6. Використання методів isAlive() та join()

Існує два способи перевірки завершення виконання потоку. Найпростіше це зробити з використанням методу `isAlive()` класу `Thread`, який описаний наступним чином:

```
final Boolean isAlive()
```

Мето `isAlive()` повертає значення `true`, якщо потік, для якого він був 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`.

3.1.7. Синхронізація

Ключем до синхронізації є концепція монітора. Монітор — це об'єкт, який застосовується як взаємно виключаюче 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 очікуючому 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 має вигляд

```
[Welcome[to synchronized[world]
```

```
]
]
```

У попередньому прикладі три потоки змагаються між собою за завершення виконання метода `call()` одного і того самого об'єкту `Callme`. Така ситуація називається "станом гонки" (race condition).

Проблема пов'язана із тим, що кілька об'єктів одночасно використовують той самий метод `call()` одного і того самого об'єкта.

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. У цьому випадку 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 гонки потоків.

3.1.8. Оператор synchronized

Синхронізація методів у класах не може бути застосована, якщо клас не передбачає багатопотокового доступу або був написаних стороннім розробником, доступ до вихідного коду якого 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);
    }
}
```

3.1.9. 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 є досяжними 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 приклад програми 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, наведена 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
```

Правильний спосіб написання програми полягає у тому, щоб використати методи `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();
    }
}

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");
    }
}
```

Всередині методу `get()` викликається метод `wait()`. Це призупиняє роботу потока до поки об'єкт класу `Producer` повідомить про те, що дані прочитані.

Схоже функціонування програми можна забезпечити без використання методів `wait()` та `notify()`.

```
    synchronized int get(){
        if(valueSet){
            System.out.println(n + " is received");
            valueSet = false;
        }
        valueSet = false;
        return n;
    }
    synchronized void put(){
        if(!valueSet){
            this.n++;
            valueSet = true;
            System.out.println(n + " is put");
        }
        valueSet = true;
    }
```

3.2. Потокова модель .NET Framework

Потокова модель .NET Framework має багато спільного із потоковою системою Java. Для підключення засобів обробки потоків потрібно використовувати засоби простору імен `System.Threading`. Затримка виконання поточного потоку на задану тривалість реалізована у вигляді виклику статичного методу

```
Thread.Sleep(millisecondsTimeout)
```

Розглянемо приклад програми, у якій у фоновому режимі запускається потік, у якому викликається метод `ThreadProc()`, у тілі якого у "вічному"

циклі значення змінної *s* постійно збільшується на 1. Робота програми закінчується після натиснення клавіші "y" або "Y".

```
class Program
{
    static double s = 0;
    static void Main(string[] args)
    {
        Thread t = new Thread(ThreadProc);
        t.IsBackground = true;
        t.Start();
        char c;
        do
        {
            Console.WriteLine("Input any key to suspend calculation");
            Console.ReadKey();
            t.Suspend();
            Console.WriteLine("Current value of s is {0}", s);
            Console.WriteLine("Exit? (Y/N)");
            c = Convert.ToChar(Console.Read());
            t.Resume();
        } while (c != 'Y' && c != 'y');
    }
    public static void ThreadProc()
    {
        for (; ; )
            s++;
    }
}
```

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);
        }
    }
}
```

3.2.1. 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("{0};a[{1}]={2}", Thread.CurrentThread.Name, i, a[i]);
           Thread.Sleep(1000);
        }
    }
}
```

3.2.2. Потоки з використанням делегатів

Делегати можуть працювати у асинхронному режимі, тобто виконуватися у окремому потоці. Розглянемо приклад обчислення функції факторіал у окремому потоці:

```
public delegate void 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 fact_delegate = new Factorial(ShowFactorial);
    IAsyncResult result = fact_delegate.BeginInvoke(10, CallBack, "message");
   Console.WriteLine("Press any key:");
    Console.ReadLine();
}

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);
}
```

4. ЗАВДАННЯ ДЛЯ ЛАБОРАТОРНИХ РОБІТ

Лабораторна робота №1

1. У наявності є 6-стадійний конвеєрний пристрій, тривалість стадій якого рівна 1, 4, 2, 3, 2 та 1 такти відповідно. З якою максимальною частотою будуть з'являтися результати на виході цього пристрою? За який час можна виконати 500 операцій на пристрої, описаному у попередній задачі, використовуючи: а) послідовний; б) конвеєрний режими виконання?
2. Нехай у наявності є 5-стадійний конвеєрний пристрій, тривалість стадій якого рівна 1, 1, 2, 1, та 3 такти відповідно. З якою максимальною частотою будуть з'являтися результати на виході цього пристрою. За який час можна виконати 500 операцій на пристрої, описаному у попередній задачі, використовуючи: а) послідовний; б) конвеєрний режими виконання?
3. У наявності є 6-стадійний конвеєрний пристрій, тривалість стадій якого рівна 5, 1, 2, 3, 4 та 3 такти відповідно. З якою максимальною частотою будуть з'являтися результати на виході цього пристрою? За який час можна виконати 200 операцій на пристрої, описаному у попередній задачі, використовуючи: а) послідовний; б) конвеєрний режими виконання?
4. У наявності є 6-стадійний конвеєрний пристрій, тривалість стадій якого рівна 4, 3, 6, 2, 4 та 5 тактів відповідно. З якою максимальною частотою будуть з'являтися результати на виході цього пристрою? За який час можна виконати 500 операцій на пристрої, описаному у попередній задачі, використовуючи: а) послідовний; б) конвеєрний режими виконання?
5. У наявності є 5-стадійний конвеєрний пристрій, тривалість стадій якого рівна 3, 5, 1, 6 та 4 такти відповідно. З якою максимальною частотою будуть з'являтися результати на виході цього пристрою? За який час можна виконати 300 операцій на пристрої, описаному у попередній задачі, використовуючи: а) послідовний; б) конвеєрний режими виконання?
6. Конвеєрний пристрій складається із k стадій, які спрацьовують за $n_1,\ldots,n_k$ тактів відповідно. За яку найменшу кількість тактів можна виконати на цьому пристрої m операцій.

Лабораторна робота №2

1. Зобразити граф алгоритму знаходження скалярного добутку двох n-вимірних векторів на обчислювальному пристрої із трьома універсальними процесорами.

2. Зобразити граф алгоритму паралельного обчислення значення виразу $a_1 + a_1 a_2 + a_1 a_2 a_3 + ... + a_1 a_2 \ldots a_7 a_8$ на обчислювальному пристрої із трьома універсальними процесорами.
3. Зобразити граф алгоритму паралельного обчислення значення виразу $a_1\left(a_1 + a_2\right)\left(a_1 + a_2 + a_3\right)\left(a_1 + a_2 + \ldots + a_{16}\right)$ на обчислювальному пристрої із чотирма універсальними процесорами.
4. Побудувати граф алгоритму паралельного обчислення значення виразу

$$a_1 + a_1^2 a_2 + a_1^4 a_2^2 a_3 + ... + a_1^{128} a_2^{64} \ldots a_7^2 a_8$$

на обчислювальному пристрої із двома універсальними процесорами.
5. Побудувати граф алгоритму обчислення значення многочлена у точці за схемою Горнера з використанням концепції необмеженого паралелізму.
6. Побудувати граф алгоритму паралельного обчислення максимального елемента 16-вимірного вектора з використанням концепції необмеженого паралелізму.
7. Побудувати граф алгоритму паралельного обчислення мінімального елемента матриці 8×8 з використанням концепції необмеженого паралелізму.
8. Побудувати граф алгоритму паралельного знаходження номера стовпчика квадратної матриці 8×8, сума елементів якого є найбільшою, з використанням концепції необмеженого паралелізму.
9. Побудувати граф алгоритму паралельного пошуку номера рядка квадратної матриці 4×4, добуток елементів якого є найбільшим, з використанням концепції необмеженого паралелізму.

Лабораторна робота №3

1. Написати на Java програму для порівняння алгоритмів сортування, які працюють паралельно.
2. Написати на C# програму для порівняння алгоритмів сортування, які працюють паралельно.
3. Написати на Java програмну реалізацію бітонічного алгоритму сортування.
4. Написати на C# програмну реалізацію бітонічного алгоритму сортування.
5. Написати на Java програму знаходження розв’язку системи лінійних алгебраїчних рівнянь із використанням паралельних обчислень.
6. Написати на C# програму знаходження розв’язку системи лінійних алгебраїчних рівнянь із використанням паралельних обчислень.

5. РЕКОМЕНДОВАНА ЛІТЕРАТУРА

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

Інформаційні ресурси

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