Лекція 5. Етапи розробки паралельних алгоритмів

План лекції:

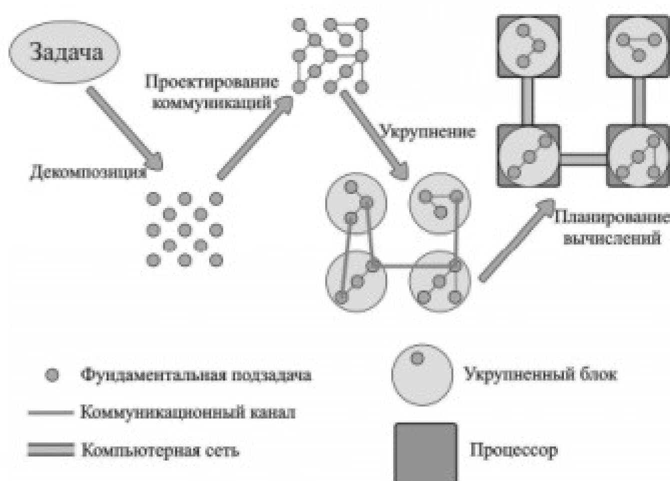
1. Загальна схема розробки паралельного алгоритму
   1.1. Декомпозиція
   1.2. Проектування комунікацій
   1.3. Укрупнення
   1.4. Планування обчислень
2. Паралельні методи матричного множення
   2.1. Постановка задачі
   2.2. Аналіз ефективності
3. Базовий паралельний алгоритм множення матриць
   3.1. Визначення завдань
   3.2. Виділення інформаційних залежностей
   3.3. Масштабування і розподіл підзадач
   3.4. Аналіз ефективності
4. Вихідна проблема — стохастичне вирівнювання
   4.1. Аналіз алгоритму стохастичного вирівнювання
   4.2. Паралельне стохастичне вирівнювання
   4.3. Інші альтернативи

1. Загальна схема розробки паралельного алгоритму

Паралельний алгоритм - це алгоритм, який може бути реалізований по частинах на безлічі різних обчислювальних пристроїв з наступним об'єднанням отриманих результатів і отримання коректного результату.

При розробці паралельних програм виділяють чотири етапи:

- декомпозиція;
- проектування комунікацій;
- укрупнення;
- планування обчислень.



Мал. 5.1. Загальна схема розробки паралельного алгоритму

1.1. Декомпозиція

На цьому етапі завдання аналізується, проводиться оцінка можливості її розпаралелювання. У разі перспективності створення ефективної паралельної програми проводиться розподіл завдання на більш дрібні частини – фрагменти структур даних і фрагменти алгоритму (фундаментальні підзадачі). Такий розподіл необхідний для підвищення максимального паралелізму завдання. Кількість фундаментальних завдань повинна бути, принаймні, на порядок більше планованої кількості використовуваних процесорів, а самі підзадачі в перспективі повинні мати приблизно однаковий розмір. Перетин виділених фрагментів алгоритму повинен бути зведений до мінімуму, щоб уникнути дублювання обчислень (наслідок – зниження ефективності паралельного алгоритму). Декомпозиція повинна бути такою, щоб при збільшенні розміру задачі зростала б кількість фундаментальних підзадач, а не розмір кожної підзадачі.

Розмір фундаментальної підзадачі визначається ступенем деталізації розпаралеленого алгоритму, яка характеризується кількістю операцій в підзадачі. Якщо в алгоритмі можна виділити підзадачі, в яких виконується лише кілька операцій, то говорять про дрібнозернистий паралелізм. Якщо фрагменти алгоритму (фундаментальні підзадачі) визначаються на рівні процедур, в яких кількість арифметичних операцій становить близько 10^3 , то має місце середньоблочний паралелізм. Грубозернистий паралелізм алгоритму характеризується можливістю виділення декількох великих завдань на рівні окремих програм, які можуть виконуватися на комп'ютері з паралельною архітектурою незалежно, що вимагає, як правило, спеціальної організації обчислень.

На цьому етапі розробки програми для паралельних обчислень не враховуються особливості архітектури багатопроцесорної обчислювальної системи.

Розрізняють декомпозицію за даними і функціональну декомпозицію. У першому випадку спочатку фрагментуються дані, з якими зв'язуються операції алгоритму їх обробки, а в другому – спочатку декомпозується алгоритм обробки даних. Прикладами для декомпозиції за даними можуть служити одновимірна, двомірна чи тривимірна декомпозиція перетворюваних за певними правилами елементів масиву, що має три і більше індексів. Одночасне обчислення за послідовним алгоритмом суми, різниці, скалярного і векторного добутків двох заданих векторів розміром n – це приклад функціональної декомпозиції.

1.2. Проектування комунікацій

На цьому етапі встановлюються комунікації (зв'язки) між фундаментальними підзадачами. Визначається комунікаційна модель передачі даних між підзадачами. Вибираються алгоритми і методи комунікацій. При цьому необхідно враховувати типи комунікацій. Так, наявність тільки локальних комунікацій, коли кожна підзадача пов'язана лише з невеликим числом інших підзадач, дає перевагу при побудові паралельної програми порівняно з глобальними комунікаціями (кожна підзадача пов'язана з великим числом інших підзадач). Також простіше будувати паралельну програму, якщо тип комунікацій структурований (комунікації утворюють регулярну структуру, наприклад з топологією «решітка»), статичний (комунікаційна структура не змінюється з часом), синхронний (відправник і одержувач повністю координують передачу даних між собою). Однак використання динамічних комунікацій (комунікаційна структура, що зв'язує підзадачі, змінюється з плином часу) і асинхронних комунікацій (відправник не контролює отримання даних, а одержувач – їх відправку) відкриває нові можливості в створенні ефективних паралельних програм.

При розробці комунікаційної моделі передачі даних між фундаментальними підзадачами необхідно виконувати наступні вимоги:

- у кожної підзадачі кількість комунікацій (зв'язків) з іншими фрагментами алгоритму має бути приблизно однакова;
- об'єм даних, що одночасно передаються по комунікаціям, повинен бути однаковим;
- перевагу у використанні мають локальні комунікації, за якими передача даних здійснюється одночасно;
- по можливості передачу даних краще поєднувати з обчисленнями;
- передача даних по комунікаціям не повинна призводити до неодночасного виконання фундаментальних завдань. На цьому етапі не враховується архітектура суперкомп'ютера.

1.3. Укрупнення

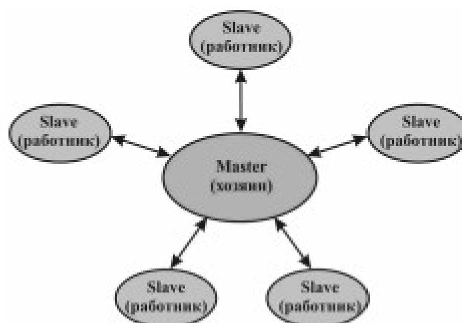
Проводиться об'єднання підзадач в більш великі блоки для підвищення ефективності створюваного алгоритму – щоб, наприклад, знизити комунікаційні витрати або трудомісткість розробки паралельного алгоритму. Враховується архітектура багатопроцесорної системи, для якої розробляється паралельна програма. Кількість укрупнених блоків фундаментальних завдань має відповідати кількості використовуваних процесорів (ядер). Першими кандидатами для об'єднання в укрупнені блоки є фундаментальні підзадачі, які не можуть виконуватися одночасно і незалежно. При

укрупненні іноді вдається знизити комунікаційні витрати за рахунок заміни передачі даних дублюванням обчислень у різних підзадачах або блоках підзадач.

Укрупнення повинно виконуватися таким чином, щоб зберігалися висока масштабованість і продуктивність паралельної програми. Кажуть, що алгоритм або програма є масштабованими, якщо ефективність Ep(n,p) можна утримувати на постійному ненульовому значенні при одночасному збільшенні кількості використовуваних процесорів p і розміру задачі n.

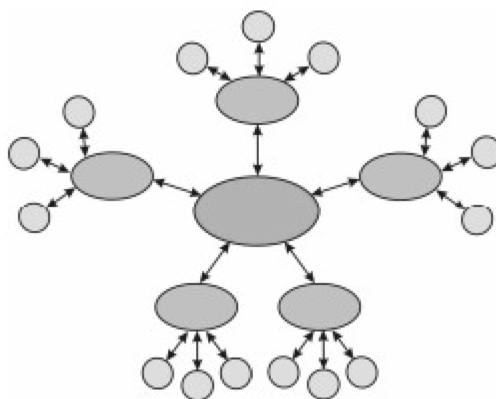
1.4. Планування обчислень

На цьому етапі здійснюється призначення укрупнених підзадач певним процесорам у відповідності з вимогами зниження комунікаційних витрат і забезпечення паралелізму. Особливе значення цей етап має при проведенні обчислень на гетерогенних багатопроцесорних системах, що характеризуються наявністю різних по продуктивності обчислювальних вузлів і компонентів міжпроцесорної мережі. Стратегія розміщення укрупнених блоків завдань на процесорних елементах будується з урахуванням основного критерію – мінімізації часу виконання паралельної програми. Найчастіше використовуються стратегії «господар/працівник», ієрархічні та децентралізовані схеми (мал. 5.2-5.5).



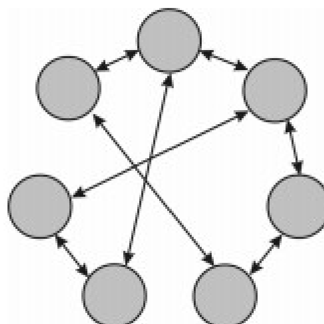
Мал. 5.2. Стратегія «господар/працівник» розміщення укрупнених блоків по шести процесорним елементам

У стратегії «господар/працівник» головний блок (master) відповідає за не перетинаючий розподіл основного обчислювального навантаження (укрупнених блоків підзадач) між процесорами - працівниками (slave), контролювання її виконання і збір розрахункових даних для підготовки основного результату розв'язання задачі (мал. 5.2).



Мал. 5.3. Ієрархічна схема «господар/працівник» розміщення укрупнених блоків по процесорним елементам

В ієрархічній схемі «господар/працівник» обсяг обчислювальної роботи також розділений між процесорами-працівниками, проте кожен з них виступає в якості процесора-«господаря» по відношенню до процесорів-«працівників», розташованих на більш низькому ступені в цій ієрархічній схемі (мал. 5.3). Характер взаємодії між рівнями ієрархії подібний до взаємодії процесорів в стратегії «господар/працівник».



Мал. 5.4. Децентралізована схема розміщення укрупнених блоків по процесорним елементам

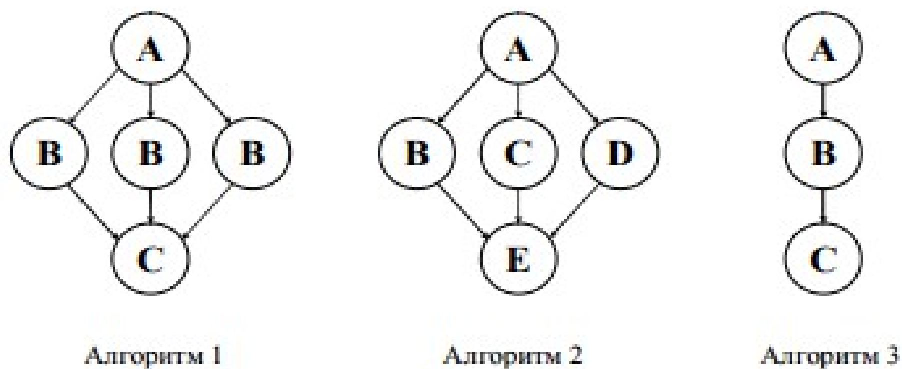
У децентралізованій схемі головний блок відсутній, обчислення в блоках виконуються, дотримуючись певної стратегії (мал. 5.4).

У розглянутій вище схемі розробки паралельних алгоритмів і програм важливу роль в ідентифікації паралелізму алгоритму відіграє аналіз графа алгоритму, який може бути побудований після виконання етапів декомпозиції та проектування комунікацій.

Граф алгоритму являє собою орієнтований граф, що складається з вершин, відповідних певним фрагментам (фундаментальним підзадачам) алгоритму, і спрямованих дуг, що відповідають за передачу даних між ними (мал. 5.5). По дугах (комунікаціях) результати, отримані при виконанні фрагментів алгоритму, передаються в якості аргументів для продовження алгоритму іншими фрагментами. Стрілка з вершини A до

вершини В графа алгоритму означає, що задача А повинна бути виконана перед завданням В. У цьому випадку має місце залежність завдання В від результату виконання завдання А. Якщо завдання А і В не пов'язані між собою, то вони незалежні і можуть виконуватися одночасно (паралельно).

На мал. 5.5 представлені графи деяких алгоритмів. Вершини представляють собою підзадачі або фрагменти алгоритмів. Буква всередині вершини визначає операцію алгоритму. Стрілки означають залежність між підзадачами.



Мал. 5.5. Паралелізм у графах алгоритмів

Граф алгоритму 1 являє паралелізм за даними. Три підзадачі з різними аргументами можуть виконувати операцію В.

Граф алгоритму 2 показує функціональний паралелізм. Підзадачі, які реалізують операції В, С, D, можуть виконуватися одночасно.

Граф алгоритму 3 відповідає лінійним залежностям між операціями А, В, С, які виконуються одна за одною.

2. Паралельні методи матричного множення

2.1. Постановка задачі

Множення матриці А розміру m×n і матриці В розміру n×l призводить до отримання матриці С розміру m×l , кожен елемент якої визначається у відповідності з виразом:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l .$$

Як випливає з цього , кожен елемент результуючої матриці є скалярний добуток відповідних рядка матриці А і стовпця матриці В:

$$c_{ij} = \left(a_i, b_j^T \right), \ a_i = \left(a_{i0}, a_{i1}, \dots, a_{in-1} \right), \ b_j^T = \left(b_{0j}, b_{1j}, \dots, b_{n-1j} \right)^T$$

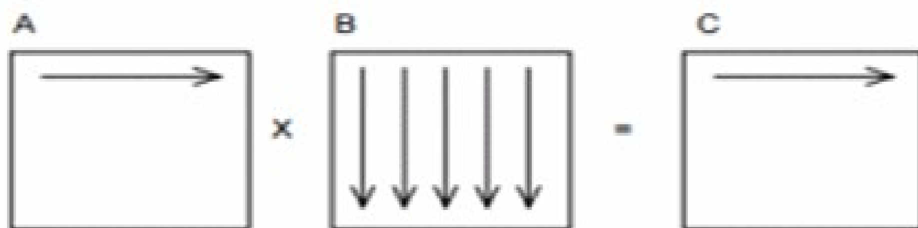
Цей алгоритм передбачає виконання m*n*l операцій множення і стільки ж операцій додавання елементів вихідних матриць. При множенні квадратних матриць розміру n×n

кількість виконаних операцій має порядок $O(n^3)$. Відомі послідовні алгоритми множення матриць, що володіють меншою обчислювальною складністю, але ці алгоритми вимагають певних зусиль для їх освоєння і, як результат, у даній главі при розробці паралельних методів як основи буде використовуватися наведений вище послідовний алгоритм. Також будемо припускати, що всі матриці є квадратними і мають розмір $n \times n$.

Послідовний алгоритм множення матриць реалізується трьома вкладеними циклами:

```
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
for (j=0; j<Size; j++){
MatrixC[i][j] = 0;
for (k=0; k<Size; k++){
MatrixC[i][j] = MatrixC[i][j] +
MatrixA[i][k]*MatrixB[k][j];
}
}
}
```

Цей алгоритм є ітеративним і орієнтованим на послідовне обчислення рядків матриці С. Дійсно, під час виконання однієї ітерації зовнішнього циклу (циклу по змінній і) обчислюється один рядок результуючої матриці (мал. 5.6)



Мал. 5.6. Перша ітерація циклу

На першій ітерації циклу по змінній і використовується перший рядок матриці А і всі стовпці матриці В для того, щоб обчислити елементи першого рядка результуючої матриці С.

Оскільки кожен елемент результуючої матриці є скалярний добуток рядка і стовпця вихідних матриць, то для обчислення всіх елементів матриці розміром n×n необхідно виконати $n^2(2n-1)$ скалярних операцій і витратити час:

$$T_1 = n^2 \cdot \ 2n-1 \ \cdot \tau \quad (1)$$

2.2. Аналіз ефективності

Час виконання алгоритму складається з часу, що витрачається безпосередньо на обчислення, і часу, необхідного на читання даних з оперативної пам'яті в кеш процесора. Час обчислень може бути оцінено з використанням формули (1).

Тепер необхідно оцінити обсяг даних, які необхідно прочитати з оперативної пам'яті в кеш обчислювального елемента у випадку, коли розмір матриць настільки великий, що вони одночасно не можуть бути поміщені в кеш. Для обчислення одного елемента результуючої матриці необхідно прочитати в кеш елементи одного рядка матриці А та одного стовпця матриці В. Для запису отриманого результату додатково потрібно читання відповідного елемента матриці С з оперативної пам'яті. Важливо відзначити, що наведені оцінки кількості даних, що читаються з пам'яті, справедливі, якщо всі ці дані відсутні у кеші. В реальності частина цих даних може вже присутня в кеші, і тоді обсяг переписуваних даних в кеш зменшується.

Розташування даних в кожному конкретному випадку залежить від багатьох величин (розмір кешу, обсягу оброблюваних даних, стратегії заміщення рядків кеша тощо). Детальний аналіз всіх цих моментів є досить скрутним. Можливий вихід в такому випадку полягає в оцінці максимально можливого обсягу даних, що переміщуються з пам'яті в кеш (побудова оцінки зверху). У нашому випадку всього необхідно обчислити n^2 елементів результуючої матриці – тоді, припускаючи, що при обчисленні кожного чергового елемента потрібно прочитати в кеш всі необхідні дані, випливає, що загальний обсяг даних, необхідних для читання з оперативної пам'яті в кеш, не перевищує величини $2n^3 + n^2$.

Таким чином, оцінка часу виконання послідовного алгоритму множення матриць може бути представлена наступним чином:

$$T_1 = n^2(2n-1)\cdot\tau + 64\cdot(2n^3+n^2)/\beta \quad (2)$$

де β є пропускна здатність каналу доступу до оперативної пам'яті (константа 64 введена для обліку факту, що в випадку кеш-промаху з ОП читається кеш-рядок розміром 64 байти).

Якщо окрім пропускної здатності врахувати латентність пам'яті, модель матиме наступний вигляд:

$$T_1 = n^2(2n-1)\cdot\tau + (2n^3+n^2)\ \ \alpha + 64/\beta \quad (3)$$

де α є латентність оперативної пам'яті.

Обидві попередні моделі є моделями на найгірший випадок і дають сильно завищену оцінку часу виконання алгоритму, так як доступ до оперативної пам'яті відбувається не при кожному зверненні до елементу. Як і раніше, введемо в модель (3) величину γ, $0 \le \gamma \le 1$, для завдання частоти виникнення кеш-промахів. Тоді оцінка часу виконання алгоритму матричного множення приймає вигляд:

$$T_1 = n^2(2n-1)\cdot\tau + \gamma(2n^3+n^2)\ \ \alpha + 64/\beta \quad (4)$$

3. Базовий паралельний алгоритм множення матриць

3.1. Визначення завдань

З визначення операції матричного множення випливає, що обчислення всіх елементів матриці C може бути виконано незалежно один від одного. Як результат, можливий підхід для організації паралельних обчислень полягає у використанні в якості базової підзадачі процедури визначення одного елемента результуючої матриці C. Для проведення всіх необхідних обчислень кожна підзадача повинна виконувати обчислення над елементами одного рядка матриці A та одного стовпця матриці B. Загальна кількість одержуваних при такому підході підзадач виявляється рівним n^2 (по числу елементів матриці C).

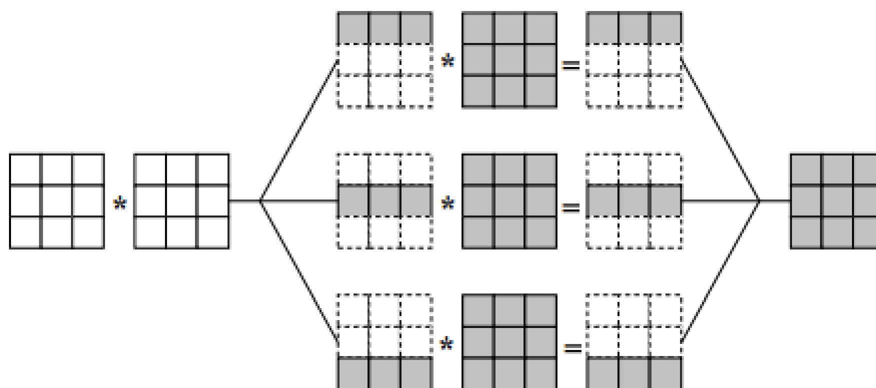
Розглянувши запропонований підхід, можна відзначити, що досягнутий рівень паралелізму є в деякій мірі надлишковим. При проведенні практичних розрахунків кількість сформованих підзадач, як правило, буде перевищувати число наявних обчислювальних елементів (процесорів та/або ядер) і, тим самим, неминучим є етап укрупнення базових завдань. В цьому плані може виявитися корисним агрегація обчислень вже на кроці виділення базових підзадач. Можливе рішення може полягати в об'єднанні в рамках однієї підзадачі всіх обчислень, пов'язаних не з одним, а з декількома елементами результуючої матриці C. Для подальшого розгляду в рамках даного розділу визначимо базове завдання як процедуру обчислення всіх елементів одного з рядків матриці C. Такий підхід призводить до зниження загальної кількості підзадач до величини n.

Для виконання всіх необхідних обчислень базовій підзадачі повинні бути доступні один з рядків матриці A і всі стовпці матриці B. Просте рішення цієї проблеми – дублювання матриці B у всіх підзадачах. Слід зазначити, що такий підхід не призводить до

реального дублювання даних, оскільки розроблюваний алгоритм орієнтований на застосування для обчислювальних систем із загальною пам'яттю, до якої є доступ з усіх використовуваних обчислювальних елементів.

3.2. Виділення інформаційних залежностей

Для обчислення одного рядка матриці необхідно, щоб у кожній подзадаче містилася рядок матриці A і був забезпечений доступ до всіх стовпців матриці B. Спосіб організації паралельних обчислень представлено на мал. 5.7.



Мал. 5.7. Організація обчислень під час виконання паралельного алгоритму множення матриць, заснованого на поділі матриць за рядками

3.3. Масштабування і розподіл підзадач

Виділені базові підзадачі характеризуються однаковою обчислювальною трудомісткістю і рівним обсягом переданих даних. У разі, коли розмір n матриць виявляється більше, ніж число p обчислювальних елементів (процесорів та/або ядер), базові підзадачі можна укрупнити, об'єднавши в рамках однієї підзадачі кілька сусідніх рядків матриці. В цьому випадку вихідна матриця A і матриця - результат C розбиваються на ряд горизонтальних смуг. Розмір смуг при цьому слід вибрати рівним $k=n/p$ (в припущенні, що n кратно p), що дозволить як і раніше забезпечити рівномірність розподілу обчислювального навантаження по обчислювальних елементах.

3.4. Аналіз ефективності

Даний паралельний алгоритм володіє хорошою «локальністю обчислень». Це означає, що дані, які обробляє один з потоків паралельної програми, не змінюються іншим потоком. Немає взаємодії між потоками, немає необхідності в синхронізації. Значить, для того, щоб визначити час виконання паралельного алгоритму, необхідно знати, скільки

обчислювальних операцій виконує кожен потік паралельної програми (обчислення виконуються потоками паралельно) і скільки даних необхідно прочитати з оперативної пам'яті в кеш процесора (доступ до пам'яті здійснюється строго послідовно). Для обчислення одного елемента результуючої матриці необхідно виконати скалярне множення рядка матриці A на стовпець матриці B. Виконання скалярного множення включає (2n–1) обчислювальних операцій. Кожен потік обчислює елементи горизонтальної смуги результуючої матриці, число елементів у смузі становить n^2/p. Таким чином, час, який витрачається на обчислення, може бути визначено по формулі:

$$T_{calc} = (n^2/p)(2n-1)\cdot\tau \quad (5)$$

Для обчислення одного елемента результуючої матриці необхідно прочитати в кеш n+8n+8 елементів даних. Кожен потік обчислює n/p елементів матриці C, однак для визначення повного обсягу переписуваних в кеш даних слід враховувати, що читання даних з оперативної пам'яті може виконуватися тільки послідовно. Як результат, скорочення обсягу переписуваних в кеш даних досягається тільки для матриці B (прочитаний одноразово в кеш стовпець матриці B може використовуватися всіма потоками без повторного читання з пам'яті). Читання рядків матриці A і елементів матриці C в граничному випадку має бути виконано повністю і послідовно. У результаті час роботи з оперативною пам'яттю при виконанні описаного паралельного алгоритму множення матриць може бути визначений у відповідності з наступним співвідношенням:

$$T_{mem} = \; n^3 + n^3/p + n^2 \quad \alpha + 64/\beta \;, \quad (6)$$

де, як і раніше, β є пропускна здатність каналу доступу до оперативної пам'яті, а α – латентність оперативної пам'яті. Отже, час виконання паралельного алгоритму становить:

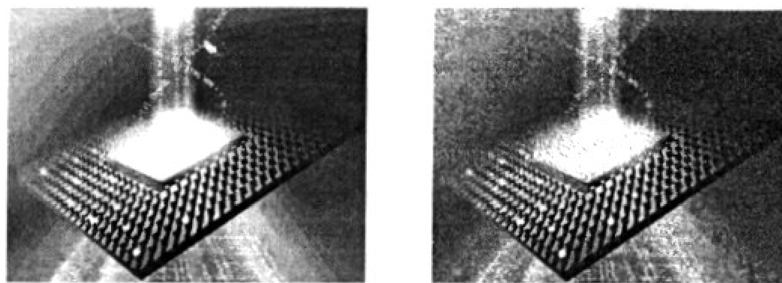
$$T_p = (n^2/p)\;\; 2n-1 \;\;\cdot\tau + \;\; n^3 + n^3/p + n^2 \quad \alpha + 64/\beta \quad (7)$$

Як і раніше, слід врахувати, що частина необхідних даних може бути переміщена в кеш завчасно за допомогою тих або інших механізмів передбачення. Крім того, звернення до даних не обов'язково призводить до кеш-промаху і, відповідно, до читання даних з оперативної пам'яті (необхідні дані можуть перебувати і в кеш пам'яті) Дані фактори можна, як і в попередніх випадках, врахувати за допомогою введення в модель показника частоти кеш промахів:

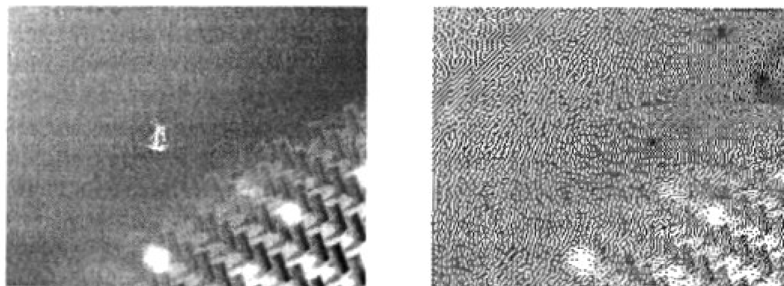
$$T_p = (n^2/p)\;\; 2n-1 \;\;\cdot\tau + \gamma \;\; n^3 + n^3/p + n^2 \quad \alpha + 64/\beta \quad (8)$$

4. Вихідна проблема — стохастичне вирівнювання

Розглянемо алгоритм стохастичного вирівнювання (або розподілу помилки), який використовується у багатьох графічних програмах і програмах обробки зображень. Спочатку метод стохастичного вирівнювання був запропонований Флойдом і Стинбергом як спосіб відтворення цифрових зображень на пристроях, які мають обмежений діапазон кольорів. Надрукувати 8-розрядне монохромне зображення на чорно-білому принтері непросто. Принтер, будучи пристроєм, що має всього два рівня кольору, не може сам надрукувати 8-разрядне зображення. Він повинен імітувати відтінки сірого за допомогою якоїсь апроксимації. Приклад зображення до і після процесу стохастичного вирівнювання показаний на мал. 5.8. Початкове зображення, що складається з 8-разрядних сірих пікселів, показано зліва, а результат після обробки за алгоритмом стохастичного вирівнювання — праворуч. Отримане зображення складається з пікселів тільки двох кольорів: чорного і білого.



Исходное 8-разрядное изображение слева, полученное 2-разрядное справа.
При разрешающей способности
типографского оборудования, с помощью которого напечатана эта книга,
они выглядят похожими



Те же самые изображения, увеличенные до 400 % и обрезанные до 25 %,
чтобы было видно детали. Теперь четко видны следы 2-разрядной
черно-белой визуализации справа и 8-разрядной серой слева.

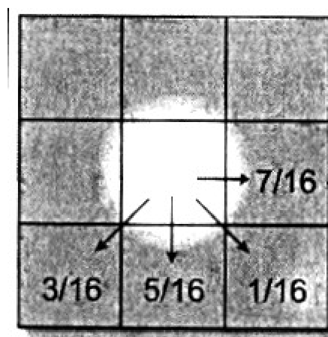
Мал. 5.8. Приклад зображення до і після процесу стохастичного вирівнювання

Базовий різновид алгоритму стохастичного вирівнювання виконується як простий трьохетапний процес.

1. Визначається вихідне значення по вхідному значенню поточного пікселя. На цьому кроці часто використовується квантування або у разі двійкових даних — межі. Для 8-розрядного сірого зображення, яке відтворюється на 1-розрядному вихідному пристрої, всі вхідні значення з діапазону [0,127] повинні зображатися як 0, а всі вхідні значення з діапазону [128,255] — 1.

2. Після того як вихідне значення визначено, код обчислює помилку між тим, що повинно бути зображено на вихідному пристрої, і тим, що фактично відтворюється. В якості прикладу припустимо, що поточне вхідне значення пікселя дорівнює 168. Оскільки воно більше граничного значення (128), то ми отримуємо вихідне значення 1. Це значення записується у вихідний масив. Для обчислення помилки програма повинна спочатку нормалізувати вихідне значення, щоб воно було в тому ж діапазоні, що і вхідне. Тобто в цілях обчислення помилки відтворення вихідний піксел повинен мати нормалізоване значення 0 (якщо вихідний піксель має значення 0) або 255 (якщо вихідний піксель має значення 1). У даному випадку помилка відтворення є різницею між значенням, яке потрібно було вивести (168), і вихідним значенням (255), що становить -87.

3. Нарешті, значення помилки розподіляється з ваговими коефіцієнтами по сусіднім пікселям, як показано на мал. 5.9.



Мал. 5.9. Розподіл помилки по сусіднім пікселям

У даному прикладі для розподілу помилки на сусідні пікселі використовуються вагові коефіцієнти Флойда—Стинберга. Обчислюється 7/16 від помилки і додається до пікселя праворуч від поточного (оброблюваного). 5/16 помилки додається до пікселя в наступному ряду, прямо під поточним. Інші помилки поширюються аналогічно. Хоча годяться і інші вагові схеми, всі алгоритми стохастичного вирівнювання слідують цьому загальному методу.

Цей трьохетапний процес застосовується до всіх пікселів зображення.

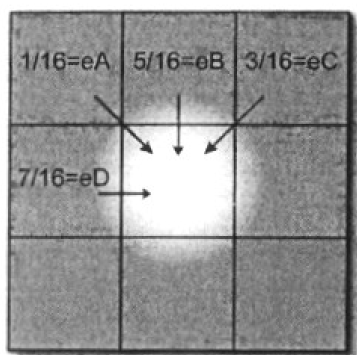
4.1. Аналіз алгоритму стохастичного вирівнювання

На перший погляд можна подумати, що алгоритм стохастичного вирівнювання є за своєю суттю послідовним процесом. Традиційний підхід полягає в поширенні помилок у міру обчислення на сусідні пікселі. В результаті для обчислення значення наступного пікселя необхідно знати значення попереднього. Така залежність означає, що код може обробляти в даний момент часу тільки один піксель. Однак нескладно підійти до вирішення цієї проблеми так, щоб воно найбільше підходило для багатопоточної обробки.

4.2. Паралельне стохастичне вирівнювання

Для перетворення звичайного алгоритму стохастичного вирівнювання до виду, який більше підходить для паралельного рішення, розглянемо різні форми декомпозиції, описані раніше в цій главі. Яка з них підійде до даного випадку? В якості підказки погляньте на мал. 5.10, який ілюструє поширення помилок з дещо іншої точки зору.

В даному випадку ми розглядаємо поширення помилки з точки зору приймаючого пікселя.



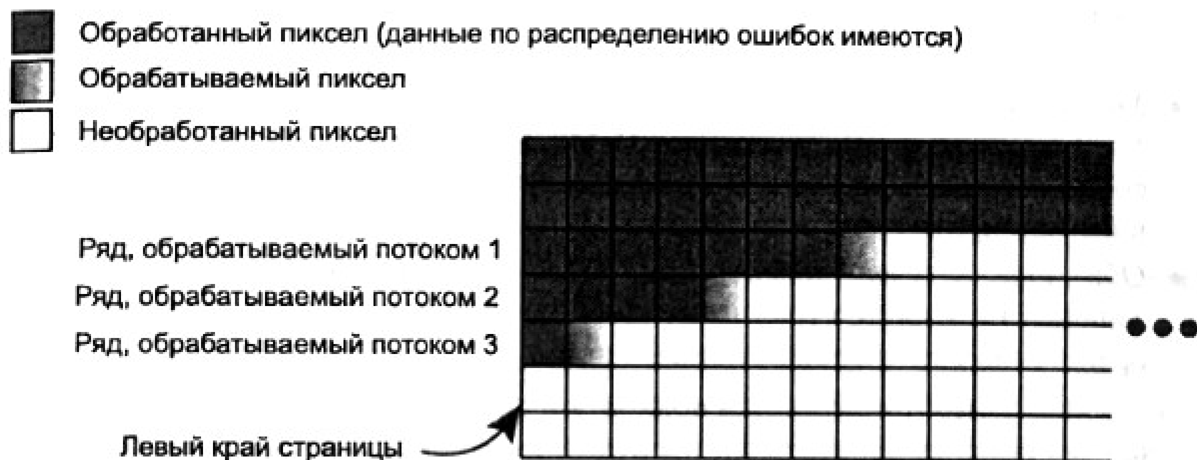
Мал. 5.10. Обчислення помилки в алгоритмі стохастичного вирівнювання з точки зору приймаючого пікселя

З урахуванням того, що піксель не може бути оброблений раніше своїх попередників в алгоритмі, проблема зводиться до того, що у нас є виробник, або в даному випадку кілька виробників, які виробляють дані (значення помилок), які споживач (поточний піксель) використовує для обчислення наступного пікселя. Інформаційний потік помилок, спрямований на поточний піксель, є критичним. Тому проблему, як здається, можна розбити шляхом декомпозиції за інформаційними потоками.

Тепер, коли ми визначили підхід, наступним кроком є визначення найкращого еталона, який може бути застосований для вирішення даної конкретної проблеми. Кожен незалежний програмний потік повинен обробляти рівний об'єм роботи (балансування навантаження). Як слід розподілити роботу? Можна (на основі описаного в попередньому розділі алгоритму) виділити один програмний потік для обробки парних пікселів в даному ряді, а інший — для обробки непарних пікселів того ж ряду. Однак такий підхід неефективний — кожен потік буде блокуватися в очікуванні завершення іншого, в результаті загальна продуктивність може бути нижче, ніж у випадку послідовного вирішення.

Для ефективного розподілу роботи між програмними потоками нам потрібен спосіб зменшення (а в ідеалі — усунення) залежності між пікселями. Для того, щоб можна було обробити піксель, він повинен мати три значення помилок (ці значення позначені на малюнку як eA, eB та eC) з попереднього ряду і одне значення з сусіднього зліва пікселя в

поточному рядку. Таким чином, поточний піксель можна обробити після обробки цих пікселів. Такий порядок диктує реалізацію, при якій кожен потік обробляє окремий ряд даних. Після того як закінчиться обробка перших декількох пікселів ряду, може почати свою роботу програмний потік, який відповідає за обробку наступного ряду. Ця послідовність наведена на мал. 5.11.



Мал. 5.11. Паралельне стохастичне вирівнювання шляхом багатопотокової обробки декількох рядків

Зверніть увагу, що на початку кожного ряду відбувається невелика затримка внаслідок того, що до обробки поточного ряду необхідно підрахувати дані щодо помилок попереднього ряду. Такі типи затримок у реалізаціях виробник-споживач практично неминучі, однак їх вплив можна мінімізувати. Для цього необхідно домогтися гарного розподілу навантаження, щоб кожен програмний потік виконувався як можна більш ефективно. У цьому випадку затримка складе два пікселя перед тим, як можна буде починати виконання наступного потоку. На сторінці розміром 8,5 х 11 дюймів при 1200 точках на дюйм в одному ряду виходить 10 200 пікселів. Затримка в два пікселя в даному випадку несуттєва.

4.3. Інші альтернативи

У попередньому розділі ми запропонували метод стохастичного вирівнювання, коли кожен програмний потік обробляв окремий ряд даних. Однак можна було б розглянути варіант декомпозиції цієї роботи на ще більш дрібні складові. При розподілі завдань між потоками інстинктивно підшукуються незалежні завдання. Найпростіший спосіб добитися паралельності — обробляти окремо кожну сторінку. Взагалі кажучи, кожна сторінка є незалежним набором даних, тобто не має залежностей від інших сторінок. Чому ж ми

запропонували рішення на основі обробки рядів замість того, щоб обробляти окремі сторінки? Цьому є три основні причини:

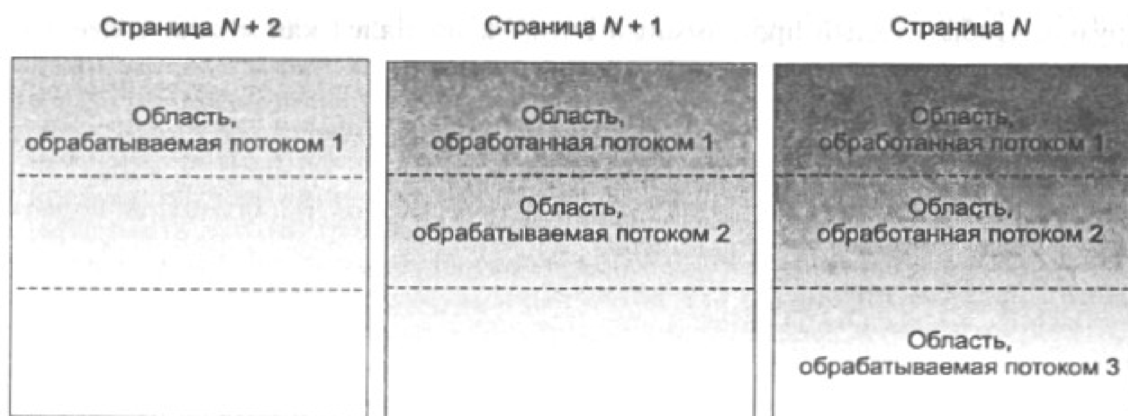
• Зображення може займати кілька сторінок. Обробка по сторінкам накладає обмеження в одне зображення на сторінку, що може не підходити для цього додатка.

• Збільшена витрата пам'яті. Сторінка розміром 8,5 x 11 дюймів при 1200 точках на дюйм займає 131 Мбайт оперативної пам'яті. Доведеться зберігати проміжні результати, а це буде менш ефективно в сенсі використання пам'яті.

• Зазвичай додаток друкує не більше однієї сторінки. Розподіл завдань па рівні сторінок не дало б приросту продуктивності в порівнянні з послідовним кодом.

Можливий комбінований підхід, коли сторінки розбиваються на області і кожна область обробляється окремим програмним потоком, як показано на мал. 5.12.

Зверніть увагу, що потоки повинні обробляти кожен свою сторінку. Це збільшує стартову затримку перед початком виконання потоків. На мал. 5.12, потік 2 перед тим, як почати обробляти дані, має стартову затримку в 1/2 сторінки, в той час як стартова затримка для потоку 3 становить вже 2/3 сторінки. Незважаючи на деякі поліпшення, комбінований підхід має ті ж (описані раніше) обмеження, що і підхід з розбиттям по сторінкам. Для того щоб обійти ці обмеження, слід реалізувати стохастичне вирівнювання на основі рядків.



Мал. 5.12. Паралельне стохастичне вирівнювання для випадку декількох сторінок і декількох програмних потоків