МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

О. В. Русанова, О.В.Корочкін

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КОМПЮ'ТЕРНИХ СИСТЕМ

ПРОГРАМУВАННЯ ТА КОМПІЛЯЦІЯ

Навч. посібник до кредитного модуля «Програмне забезпечення комп'ютерних систем-1.Програмування та компіляція» для студентів магістерської освітньої програми «Комп'ютерні системи та мережі», за спеціальністю123 – "Комп'ютерна інженерія"

Київ
КПІ ім. Ігоря Сікорського
2020

УДК681.3.06
Н72

Рецензент:
О.В.Тарасенко-Клятченко, канд. техн. наук, доц.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор
В. П. Сімоненко, доктор техн. наук, проф.,
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Русанова О.В., Корочкін О.В.
Н72 Програмне забезпечення комп'ютерних систем. Програмування та компіляція. [Електронний ресурс] /, Русанова О.В.,О.В.Корочкін. – Електронні текстові дані (1 файл: 1,8 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 95с.

Розглянуті особливості розробки програмного забезпечення паралельних комп'ютерних систем для підвищення їх реальної продуктивності, труднощі, що виникають при цьому та шляхи їх подолання. Викладені і проаналізовані підходи до програмування, які застосовуються для паралельних систем з різною архітектурою. Розглянуті мовні засоби для синхронних та асинхронних паралельних процесів. Наведено особливості алгоритмів компіляції для різних комп'ютерних систем. Викладені алгоритми автоматичного розпаралелювання програм. Особлива увага приділяється питанням розпаралелювання та векторизації циклів. Посібник може бути використаний студентами вищих учбових закладів при вивченні курсу «Програмне забезпечення комп'ютерних систем», а також при виконанні магістерських дисертацій присвячених удосконаленню програмного забезпечення для паралельних комп'ютерних систем.

УДК681.3.06

Програмне забезпечення комп'ютерних систем. Програмування та компіляція: Навч. посібник до кредитного модуля «Програмне забезпечення комп'ютерних систем-1.Програмування та компіляція» для студентів магістерської освітньої програми «Комп'ютерні системи та мережі», за спеціальністю123 –"Комп'ютерна інженерія"/Автори Русанова О.В., Корочкін О.В.. – К.: КПІ імені Ігоря Сікорського, 2020. – 95 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 6 від 31.01.2020 р.) за поданням Вченої ради факультету інформатики та обчислювальної техніки (протокол № 4 від 25.11.2019 р.)

Навчальне видання

Програмне забезпечення комп'ютерних систем. Програмування та компіляція

Навч. посібник до кредитного модуля «Програмне забезпечення комп'ютерних систем-1.Програмування та компіляція» для студентів магістерської освітньої програми «Комп'ютерні системи та мережі», за спеціальністю123 – "Комп'ютерна інженерія"

Укладачі:
Корочкін Олександр Володимирович, канд. техн. наук, доц.
Русанова Ольга Веніамінівна, канд. техн. наук, доц.
Відповідальний редактор: В.П.Сімоненко, д.т.н., проф.
Рецензент: Тарасенко-Клятченко О.В. , канд. техн. наук, доц.

Зміст

1. Вступ

Паралельна обробка інформації створює передумови для суттєвого підвищення продуктивності засобів обчислювальної техніки. Тому останні десятиріччя пов'язані з швидким розвитком паралельних комп'ютерних систем(ПКС). Зараз у світі існує велика кількість архітектурних рішень ПКС і потреба у їх використанні весь час зростає. Сьогодні однією з найскладніших проблем розвитку ПКС є підвищення ефективності їх використання при вирішенні задач користувача. У рамках даного кредитного модуля розглядаються основні напрямки підвищення ефективності ПКС, які пов'язані з досконалим використанням методів та засобів програмування та компіляції для ПКС із різними архітектурами.

1.1 Класифікація комп'ютерних систем

Перш за все розглянемо найважливіші характеристики комп'ютерних систем (КС):

1. ***Синхронність.***

Якщо в КС усі процесори працюють під управлінням єдиного машинного такту, то такі системи відносяться до класу синхронних. У разі відсутності єдиного машинного такту - системи асинхронні. З точки зору програмного забезпечення КС ця характеристика є однією з найважливіших.

2. ***Зернистість*** (рівень паралелізму).

Розрізняють системи:

- крупнозернисті;
- середньо зернисті;
- дрібно зернисті.

3. ***Зв'язність.***

Розрізняють системи:

- слабкозв'язані;
- зв'язані;
- сильнозв'язані.

Ця характеристика (її тип) визначається за швидкістю передачі даних між процесорами.

4. ***Тип пам'яті :***

- розподілена (загальна);
- роздільна (локальна);
- роздільно-розподілена.

5. ***Спосіб управління :***

- загальне управління;
- розподілене управління.

6. ***Масштабованість.***

Це здатність до нарощування процесорів в системі. Існують КС з масштабованістю:

- обмеженою;
- необмеженою.

7. ***Симетричність систем.***

Існує декілька визначень симетричності систем. Відповідно до першого з них у симетричній системівсі процесори мають рівні права на доступ до пам'яті. Відповідно до другого визначення, у симетричній системі всі процесори рівноправні з точки зору виконання системних і користувацьких завдань, а в несиметричній - одна частина процесорів призначена для виконання системних, а інша частина - для користувацьких завдань. Згідно до третього визначення розрізняють симетричні та несиметричні топології систем. У симетричних топологіях усі процесори мають одне й те саме значення середньої відстані до інших процесорів системи, а у несиметричних такі відстані можуть мати різні значення.

Розглянемо класифікацію сучасних комп'ютерних систем (КС), представлену на рис. 1.1. Різні типи КС розрізнятимемо за наступними ознаками:

- Тип зв'язаності.
- Тип управління.
- Тип паралельної обробки.
- Організація пам'яті.
- Тип взаємозв'язку.
- Тип топології.

По типу зв'язаності розрізнятимемо три типи КС : паралельні КС(сильно зв'язані); кластерні мультикомпьютерні системи (зв'язані) і розподілені системи (слабкозв'язані).

Паралельні КС - це комплекс апаратних і програмних засобів, призначених для ефективного виконання конкретного користувацького завдання. До складу ПКС входить множина, як правило, ідентичних процесорів, або процесор (процесори) конвеєрного типу, в якому (-их) виконується одночасна (паралельна) обробка інформації. У ПКС може застосовуватися спільна (розподілена), роздільна або розподілено-роздільна оперативна пам'ять (ОП), взаємодія між процесорами можлива або через ОП, або за допомогою спеціальних каналів. Також є загальні канали введення-виведення (КВВ), зовнішня пам'ять, як загальна, так і роздільна. Метою ПКС є підвищення реальної користувацької продуктивності системи.

Кластерні мультикомпьютерные системи (КМС) - це комплекс, що складається з множини комп'ютерів, об'єднаних високошвидкісними комунікаційними засобами, є єдиним цілим для операційної системи (ОС),

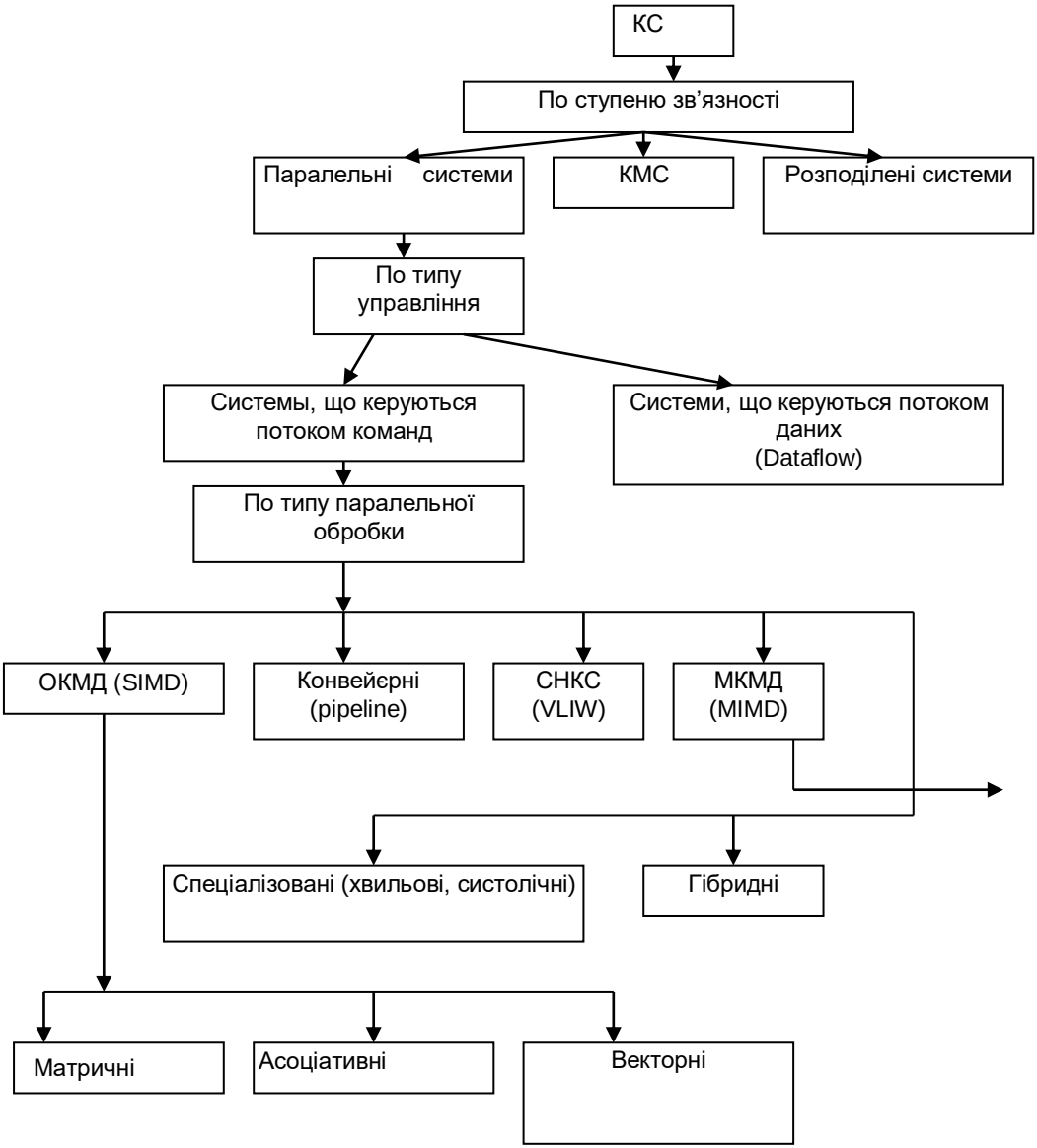
системного програмного забезпечення, прикладних програм і користувачів. Як і ПКС, кластерні системи передусім призначені для ефективного виконання конкретного обчислювального завдання, тобто для підвищення реальної призначеної для користувача продуктивності. По суті вони є досконалішимпродовженням багатомашинних комплексів 60-х років XX-го століття. Нині найбільш поширеними комунікаційними засобами кластерних систем є Gigabit Ethernet та Infiniband, що відрізняються ціною і технічними характеристиками.

Розподілені системи (РС) - це комплекс апаратних і програмних засобів, призначених для ефективного одночасного виконання безлічі обчислювальних завдань. Апаратні засоби є множина комп'ютерів (часто різних по продуктивності), об'єднаних між собою за допомогою або мережевих засобів зв'язку, або високошвидкісних комутаторів, вживаних в кластерних системах. Можна вважати, що ПКС і КМС є поодинокими випадками РС, оскільки вони призначені для ефективного виконання єдиного обчислювального завдання, в той час, як РС - для ефективного виконання множини обчислювальних завдань. Таким чином, основною метою РС є підвищення системної продуктивності системи. Прикладом сучасних РС є обчислювальні **Grid системи**, які представляють собою множину різних за архітектурою і географічно віддалених КС, що об'єднані між собою за допомогою засобів Інтернету.

За типом управління ПКС можуть бути розділені на два типи:

- Системи, що керуються потоком даних.
- Системи, що керуються потоком команд.

Системи, що керуються потоком даних (Dataflow systems) складаються з множини ідентичних процесорів, із загальним управлінням і загальною (розподіленою) оперативною пам'яттю. **Dataflow (**потокові) системи мають обмежену масштабованість через конфлікти, що виникають у спільній пам'яті. Ці системи призначені для реалізації дрібнозернистого паралелізму (на рівні операцій). Розпаралелювання в цих системах здійснюється спочатку, під час компіляції, а потім - динамічно в процесі виконання програми. Виконання команд і відповідних операцій здійснюється в той момент часу, коли для них готові операнди, тобто дані диктують порядок виконання операцій. Потокові системи відносяться до класу асинхронних. Перевагою цих систем є простота програмування, оскільки на етапі програмування відсутня проблема розпаралелювання. На сьогодні архітектура **Dataflow** систем використовується у сигнальних процесорах **(DSP)**, мережевих маршрутизаторах, прискорювачах нейромереж. Прикладом останніх є проект Triton компанії WaveComputing [16].


КС
По ступеню зв'язності
Паралельні системи
КМС
Розподілені системи
По типу управління
Системы, що керуються потоком команд
Системи, що керуються потоком даних (Dataflow)
По типу паралельної обробки
ОКМД (SIMD)
Конвейєрні (pipeline)
СНКС (VLIW)
МКМД (MIMD)
Спеціалізовані (хвильові, систолічні)
Гібридні
Матричні
Асоціативні
Векторні

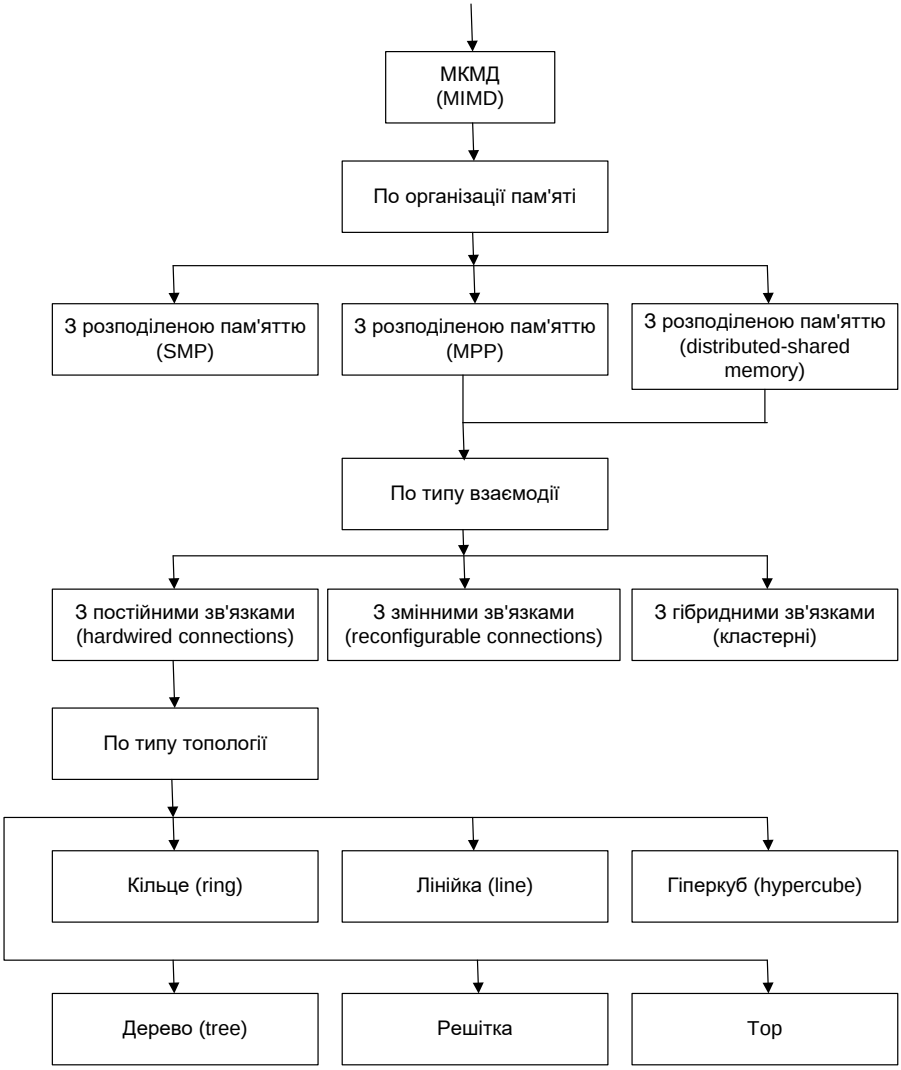


Рис. 1.1. Класифікація комп'ютерних систем

За типом паралельної обробки, ПКС, керовані потоком команд, можна розділити на наступні класи:

- **ОКМД**(одиночний потік команд множинний потік даних) (SIMD - single instruction multiple data).
- **Конвеєрні** (Pipeline).
- **Системи з наддовгим командним словом** (**СНКС**) (VLIW - very large instruction words).
- **МКМД**(множинний потік команд множинний потік даних) (MIMD - multiple instruction multiple data).
- **Спеціалізовані**(проблемно-орієнтовані, хвильові, систолічні).
- **Гібридні.**

Системи класу ОКМД - це синхронні ПКС, призначені, як правило, для реалізації дрібнозернистого рівня паралелізму. До них відносяться матричні, асоціативні і векторні ПКС.

Матричні ПКС - це КС, що складаються з великої кількості універсальних процесорів, призначених для повнорозрядної або для однорозрядної обробки інформації. Працюють усі процесори під управлінням єдиного пристрою управління, тобто в один і той же момент часу усі процесори виконують одну й ту ж саму операцію над різними даними. Ці системи орієнтовані на реалізацію природного паралелізму. Вони відносяться до систем з розділеною (локальною) пам'яттю, а значить процесори можуть бути об'єднані у різні топології з необмеженою масштабованістю. Найбільш поширеними топологиями для матричних систем є кільце, решітка і тор. Сьогодні такі архітектури використовуються для графічних процесорів та прискорювачів нейромереж.

Асоціативні ПКС відрізняються від матричних тим, що в якості оперативної пам'яті використовується асоціативна пам'ять.

Векторні системи будуються на основіодного або множини векторних процесорів. У свою чергу векторний процесор включає набір одно функціональних конвеєрів, які використовують загальну, або розподілену пам'ять. Такий тип оперативної пам'яті впливає на обмеженість масштабованості подібних процесорів. Якщо векторна система складається із множини векторних процесорів із роздільною пам'яттю, то масштабованість таких систем необмежена. Векторні системи орієнтовані на реалізацію як природного паралелізму, так і паралелізму суміжних операцій, а у разі багатопроцесорних варіантів векторних систем, що складаються з безлічі векторних процесорів, то і на реалізацію середньо та крупнозернистого паралелізму. При побудові суперкомп'ютерів векторні процесори використовують такі компанії, як Cray, Nec (векторний чіп SX-Aurora), Fujitsuта ін. Одним із цікавих прикладів є чіп A64FX компанії Fujitsu, представлений у 2018 р. На його

базі розробляються суперкомп'ютери Post-K (планується випуск у 2021 р.) та CrayCS500 із екзафлопною продуктивністю [17-19].

Конвеєрні системи побудовані на базі багатофункціональних конвеєрних пристроїв. Така система може містити одне або набір таких пристроїв. Це синхронні системи. Конвеєрні системи відносяться до систем із загальною, або роздподіленою пам'яттю і отже мають обмежену масштабованість. Вони орієнтовані на паралелізм суміжних операцій, паралелізм мікроопераций, природний паралелізм(дрібнозернисті види паралелізму).

Конвеєрні системи можна розділити на три класи:

- Конвеєр з постійним тактом. Тривалість такту визначається по найскладнішій функції, яка може бути виконана в цьому конвеєрі (наприклад ділення).
- Конвеєр із статичною перебудовою такту. Тривалість такту визначається перед виконанням обчислювальної функції, тобто статично. Такий конвеєр у фіксований момент часу працює як одно функціональний, не дивлячись на те, що він є апаратно багатофункціональним. Перебудова конвеєра з однієї функції на іншу здійснюється тільки після його повного звільнення. У цей же момент перевизначається і тривалість його такту.
- Конвеєр з динамічною перебудовою такту. Він є багатофункціональним, тривалість його такту визначається по найскладнішій функції з тих, які виконуються в даний момент в конвеєрі, тобто перебудова такту відбувається динамічно. Такі процесори є потенційно найбільш продуктивними, але і найбільш дорогими.

Конвеєрні архітектури сьогодні використовуються для побудови конвеєрів команд, а також для проблемно-орієнтованих систем.

Системи з наддовгим командним словом (СНКС) складаються з набору універсальних процесорів, орієнтованих на реалізацію паралелізму суміжних операцій (дрібнозернистий паралелізм). Ці системи відносяться до класу синхронних. Вони використовують спільну або розподілену оперативну пам'ять і загальний пристрій управління. Система працює із спеціальними наддовгими командами, що представляють собою множини (вектор) команд (для кожного процесора - своя команда). Використовується спеціальна широкоформатна оперативна пам'ять. Розміри команди в залежності від кількості процесорів у системі можуть досягати сотень і тисяч розрядів. СНКС є системою з динамічною перебудовою такту (тривалість такту визначається по найскладнішій команді, що входить у вектор команд). Можливе неефективне використання процесорних ресурсів через різні по складності і відповідно,

за часом виконання команди, що входять в одне і те ж командне слово. Проте, якщо використовуються процесори з RISC архітектурою, то цей недолік усувається. Мають обмежену масштабованість. СНКС архітектура знайшла своє використання у чіпах IntelItanium, Ельбрусі-3. Останнім прикладом є 16-ядерний VLIW процесор компанії ViaCenTaur із вбудованим блоком штучного інтелекту (2019 р.) [20] .

МКМД системи - це багатопроцесорні системи, що складаються з множини універсальних та ідентичних процесорів з розподіленим управлінням, мають різну організацію пам'яті (роздільну, спільну (розподілену), роздільно-розподілену). Ці системи орієнтовані на паралелізм незалежних гілок і паралелізм незалежних завдань (крупно та середньо зернистий паралелізм). Вони відносяться до класу асинхронних.

МКМД системи із спільною (розподіленою) пам'яттю (shared memory) називають також SMP (symmetric multiprocessor) системами, оскільки в цій системі усі процесори мають рівні права на доступ до пам'яті. У SMP системах використовується три варіанти топологій, таких як:

- Шинна.
- Багатошинна комутація.
- За допомогою матричних комутаторів.

Основним недоліком цих систем є обмежена масштабованість, пов'язаних із конфліктами по пам'яті. На практиці число процесорів в SMP системах не перевищує 64. Перевагою подібних систем є швидкий обмін між процесорами та відсутність проблем маршрутизації.

МКМД системи з розділеною пам'яттю (distributed memory) називають також MPP (massively parallel processing) системами. Ці системи не мають обмежень на масштабування і тому нині є найбільш перспективними з точки зору зростання продуктивності. Сьогодні список найбільш продуктивних систем світу TOP-500 складається з MPP та кластер них систем. Основні недоліки цих систем пов'язані з надзвичайною складністю їх ефективного використання. Необхідно вирішувати складні проблеми створення прикладного і системного програмного забезпечення, що включають виконання задач розпаралелювання, ефективного паралельного програмування з урахуванням синхронізації обміну повідомленнями, маршрутизації, розподілу ресурсів і тому подібне. Від якості рішення перерахованих завдань залежить значення реальної продуктивності системи.

МКМД з роздільно-розподіленою пам'яттю (distributed-shared memory)(DSM) - це гібрид SMP і MPP систем. DSM -системы є архітектурою, в якій пам'ять фізично розділена, але логічно (програмно) загальнодоступна. Така ідеологія підтримується в NUMA (non - uniform memory access) системах. Масштабованість цих систем обмежується об'ємом адресного простору, можливостями апаратури підтримки

когерентності кеш пам'яті і можливостями ОС по управлінню великим числом процесорів. На справжній момент, максимальне число процесорів в NUMA -системах складає 256 (Origin2000).

Системи, в яких пам'ять фізично розділена ***за типом взаємозв'язку*** можна розділити на три групи, :

- Системи з постійними (статичними) зв'язками.
- Системи із змінними (динамічними) зв'язками.
- Системи з гібридними зв'язками (постійні+змінні зв'язки).

У системах з постійними зв'язками зв'язок між процесорами здійснюється через канали (link). У цьому випадку формується статична конфігурація. Недоліками такої конфігурації є жорсткі вимоги до елементної бази - наявність множини каналів і автономних контроллерів введення/виведення. Інакше можлива побудова тільки шинної топології. Системи, що використовують статичні зв'язки розрізняють ***по типу топології*** . На рис.1.1 представлені основні типові топології подібних систем: лінійка, кільце, дерево, решітка, гіперкуб, тор.

У системах із змінними зв'язками використовуються різні засоби комутації, такі як однокаскадні(single - stage), багатокаскадні (multistage) і матричні(crossbar). В цьому випадку формується динамічна конфігурація (реконфігурована система). Із перерахованих комутаторів найбільш продуктивними і одночасно дорогими є матричні комутатори. Проте вони є найбільш використовуваними нині.

У системах із гібриднимизв'язками процесори зв'язуються як через канали, так і через комутатори. У цьому випадку часто формується так звана кластерна конфігурація. Подібні системи можуть складатися з ієрархії груп (кластерів) процесорів.Зазвичай усередині групи процесорів можуть використовуватись статичні зв'язки, а кластери між собою зв'язуються через комутатори.

Гібридні ПКС - це системи, які включають себе комбінацію структурних рішень, таких як векторна, потокова, МКМД, СНКС і конвеєрна організації.

1.2 Основні етапи виконання обчислювальних завданьу комп'ютерних системах

Розглянемо шість основних етапів підготовки до виконання обчислювальних завдань в КС:

1. Фізична постановка завдання.
2. Математична постановка завдання.
3. Вибір (розробка) паралельного чисельного методу.
4. Програмування.

5. Компіляція.

6. Організація паралельних процесів у рамках операційної системи (ОС).

Найкращий результат адаптації обчислювального завдання до обчислювальних засобів досягається тоді, коли робота проводитися комплексно, і з урахуванням архітектури КС. Проте, на практиці роботи виконуються, як правило, послідовно з вимушеними ітеративними повтореннями деяких етапів робіт. Саме цим і можна пояснити появу в 70-х роках гіпотези Мінського (рис. 1.2.), відповідно до якої, продуктивність КС із зростанням кількості процесорів росте, не за лінійним, а за логарифмічним законом. Причинами такого зростання є:

- Об'єктивна причина, пов'язана з паралельно-послідовноюприродою користувацької задачі і, як наслідок пересилкою даних між послідовними ділянками завдання.
- Суб'єктивна причина, пов'язана з якістю виконання основних етапів виконання задачі і пересилок даних.

КС 70-х років підтверджували цю гіпотезу, оскільки виконання вище перерахованих етапів проводилося чисто послідовно, ОС таких систем практично не зазнавали змін, їх продуктивність дійсно різко падала. Для отримання підвищеної реальної користувацької продуктивності необхідно виконувати етапи 3 - 6 комплексно. Усі ці етапи мають бути виконані з урахуванням специфіки архітектури і особливостей КС, (число процесорів, об'єм пам'яті і т. д.), оскільки тип КС визначає ефективність тих або інших чисельних методів рішення задачі, вимоги до мов програмування і відповідно спосіб програмування, функції трансляторів, вимоги до конфігурацій і функцій ОС.

Так, наприклад, для МКМД систем програміст повинен створити таку програму, при реалізації якої протягом усього часу існувало би*n* або більше незалежних процесів (де *n* - число процесорних елементів (ПЕ) у КС), причому найкращим варіантом задач є слабка зв'язаність процесів. Більше того, користувач може (якщо дозволяють конструкції мови програмування) створювати файли конфігурації, який передбачає розподіл процесів по ПЕ. Для цього він повинен, з урахуванням інтервалів часу виконання окремих процесів на ПЕ, оптимізувати графік робіт для скорочення часу вирішення завдань. Це складне завдання по організації обчислень є практично нездійсненним у разі великого числа процесорів (для МРР систем). Таку задачу доцільно вирішувати у рамках ОС або спеціального системного програмного забезпечення, виконувати аналіз і планування алгоритмів на КС. Реалізація цього завдання - одна з головних проблем ОС КС для отримання високої реальної, призначеної для користувача, продуктивності.

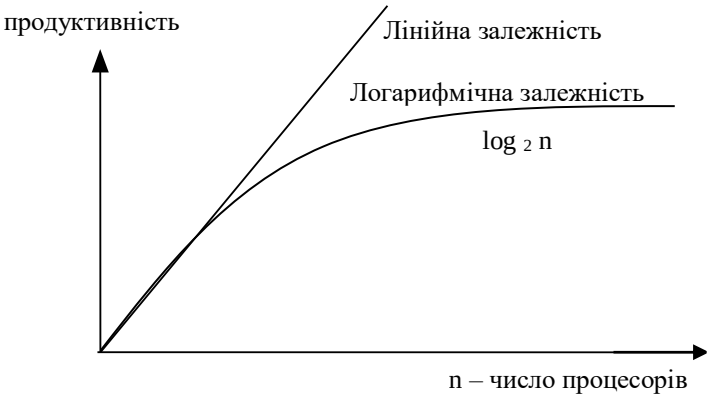


Рис.1.2.Гіпотеза Мінського.

Дещо простіше завдання (щодо програмування) для користувачів таких систем, як конвеєрні, СНКС і Dataflow. Вони можуть використовувати традиційні мови програмування такі ж, як і однопроцесорні КС. Тобто тут простіше виконується 4 етап, але ускладнюється 5 етап, оскільки компіляція повинна включати функцію автоматичного розпаралелювання обчислень.

Зрозуміло, що продуктивність КС зростає пропорційно зростанню завантаженості обчислювальних засобів. Комплексне виконання усіх етапів веде до підвищення завантаженості, проте, насправді, 100% зайнятість реально не може бути досягнута. Завжди присутні певні часові втрати, пов'язані із забезпеченням правильності ходу обчислень, а також з необхідністю обміну даними між процесорами. Тому подальше підвищення завантаженості КС можливо за рахунок виконання множини обчислювальних завдань, якщо одне завдання не може повністю завантажити усі процесори. Таким чином, максимальна реальна продуктивність системи може бути досягнута за рахунок комплексного ефективного виконання 3-6 етапів робіт, а також при необхідності виконання множини завдань.

З усього сказаного ясно, що з одного боку ПЗ КС значно впливає на реальну користувацьку продуктивність КС, а з іншої - типи архітектури КС впливають на їх програмне забезпечення.

Так матричні, векторні КС, SMP і MPP системи мають істотний вплив на паралельні мови програмування і прикладні програми (а конвеєрні, потокові - ні). Векторні, матричні, конвеєрні, потокові, SMP і MPP системи

мають істотний вплив на компілятори, а SMP і MPP системи - на операційні системи (у них найскладніше вирішуються питання синхронізації, диспетчеризація, маршрутизації і тому подібне). У векторних, матричних, конвеєрних, синхронних системах на етапі компіляції і програмування здійснюється адаптація обчислень до КС, розмірність завдання приводиться до розмірності системи.

Таким чином, основною метою ПЗ КС є забезпечення максимальної користувацької продуктивності за рахунок оптимального рішення перерахованих вище проблем. Критерій оптимізації - мінімальний час рішення обчислювальних завдань. Це може бути досягнуто за рахунок створення ефективних методів і засобів розпаралелювання обчислювальних алгоритмів, програмування, компіляції, а також розподілу обчислювальних робіт по процесорах. Перераховані методи і засоби складають основу для вдосконалення мовних засобів, компіляторів і ОС КС.

У даному учбовому посібнику спочатку ми розглядатимемо особливості мов програмування, що описують паралельні алгоритми. Тут, ми будемо розглядати специфіку програмування, продиктовану особливостями різної архітектури КС. Далі розглянемо особливості компіляції длярізних типів ПКС.

2. Мовні засоби опису паралельних процесів

2.1 Основні підходи до створення паралельних мов програмування

Поява паралельних КС привела до ускладнення програмування для цих систем. Алгоритми і структури даних, які розробляються для традиційних однопроцесорних систем, не можуть бути сліпо перенесені на КС паралельної архітектури без втрати ефективності у використанні процесорів і обмеженого підвищення продуктивності. Таким чином, виникла задача створення нових паралельних алгоритмів і мов програмування, що дозволяють описувати паралельні процеси. Проте, до моменту створення паралельних КС вже була величезна кількість ефективних послідовних чисельних алгоритмів і ПЗ, що реалізують ці алгоритми і повністю відмовитися від них було б не розумно. Тим паче, що для деяких архітектур паралельних КС передбачається автоматичне розпаралелювання програм у процесі їх виконання.

Існує два шляхи програмування.

- Адаптація програм, написаних для традиційних комп'ютерів до архітектури ПКС.

- Створення нових мовних засобів паралельного програмування.

У першому випадку функція розпаралелювання виконується у процесі компіляції. У цьому випадку шляхом моделювання досліджуються наявні програми (для виконання певного обчислювального завдання) з метою перевірки конфліктів і рівня пристосування цих програм до паралельної обробки на заданій архітектурі. Тут здійснюється перевірка залежностей між елементами програми і тим самим здійснюється розпаралелювання. У результаті здійснюється вибір оптимальної програми для заданої архітектури ПКС. Як правило, таке моделювання здійснюється у процесі компіляції програми. Таким чином, у даному випадку ми спостерігаємо тісний зв'язок реалізації 4 і 5 етапів (програмування і компіляція програми) робіт для ПКС. Тут результат 4-го етапу впливає на результат 5 і навпаки, тобто 4 і 5 етапи виконуються спільно до тих пір, поки не буде знайдена оптимальна програма і її паралельне представлення (у процесі компіляції). Проблеми автоматичного розпаралелювання послідовних програм ми розглядатимемо пізніше у другій частині курсу, коли вивчатимемо проблеми компіляції програм для паралельних КС. Приклади застосування такого програмування - це конвеєри, системи з наддовгим командним словом, Dataflow системи.

У другому випадку для усіх інших систем, існує три підходи до створення мов паралельного програмування :

1. Розширення традиційних мов засобами паралельної обробки (паралельний Сі, паралельний Fortran, паралельний Паскаль та ін.).

2. Створення паралельних мов, орієнтованих на конкретну архітектуру ПКС (ОККАМ, ВЕКТОР, Эль- 76 для Ельбрусу), конструкції яких могли б ефективно транслюватися в систему команд конкретних типів ПКС.

Створення універсальних мов паралельного програмування, які не орієнтовані на конкретну архітектуру КС (Ada, Modula).

Перший підхід широко застосовується, завдяки наступним перевагам:

- найбільш простий у застосуванні, оскільки такі мови легко освоюються, будучи розширеннями уже відомих і широко використовуваних мов програмування.
- дешевий підхід, оскільки є можливість використання раніше розробленого програмного забезпечення.

Проте при реалізації цього підходу відомі мови необхідно доповнити новими засобами, такими як опис паралельних процесів, методи синхронізації доступу до роздільного ресурсу, методи синхронізації при обміні повідомленнями. Такі засоби, не властиві ідеології створення традиційних мов програмування, можуть бути не ефективними, механізми синхронізації, як правило, не автоматизовані, що веде до зниження надійності та складності програм. Таким чином, необхідність збереження

концепцій використовуваної мови, часто заважає, а в деяких випадках ускладнює створення мовних засобів паралельного програмування.

Переваги спеціалізованих мов на базі другого підходу:

- висока ефективність програмного коду;
- повна відповідність структури паралельної програми розмірності і архітектурі системи.

Такі мови дозволяють розробляти найбільш ефективні програми. Проте занадто дорого для кожної архітектури створювати вузькоспеціалізовані мови, з низькою ефективністю перенесення програм на інші типи ПКС і необхідністю вивчення безлічі спеціалізованих мов користувачами.

Універсальні паралельні мови у рамках третього підходу мають наступні переваги:

- висока ефективність перенесення програми з однієї паралельної архітектури на іншу (змінюють тільки файл конфігурації, розподіл по процесорах, підлаштування під топологію);
- можливість введення нових конструкцій, що підвищують ефективність мови програмування і надійність написання на ній програм (в порівнянні з першим підходом).

Недоліками таких мов є:

- неперевіреність нових концепцій;
- частенько складність створення ефективних об'єктних програм при компіляції (у порівнянні з другим підходом);
- необхідність вивчення користувачем не лише синтаксису мови, але і закладених в нього концепцій.

Перший підхід був реалізований для більшості паралельних КС : ASC SYSTEM, ПС- 3000, трансп'ютерні системи. Другий підхід був реалізований при створенні мов "Вектор " для ПС- 2000, Эль- 76 для Ельбрусу, Оккам для трансп'ютерних систем . Яскравим представником третього підходу є ADA, Modula - 2, CSP, PLITS, SR, Jawa.

Розглянемо вплив структури і архітектури КС на мовні засоби паралельного програмування за допомогою таблиці 2.1.

Таблиця2.1 Застосування мовних засобів паралельного програмування.

Тип ПКС	Мовні засоби				
	Синхронна паралельна обробка	Асинхронна паралельна обробка			
	Операції над масивами	Опис процесів	Ініціалізація і завершення паралельних процесів та їх синхронізація	Синхронізація доступу до ресурсів	Обмін повідомленнями і їх синхронізація
Векторн. і матричн. КС	+	-	-	-	-
МКМД системизспільною(розподіл.) пам'яттю	- +	+	+	+	-
МКМД з роздільною пам'яттю	- +	+	+	+ -	+

2.2 Специфічні проблеми паралельного програмування

Аналізуючи наведені у таблиці 1.1 нові мовні засоби, перерахуємо специфічні для паралельного програмування поняття і проблеми.

У традиційному програмуванні, на відміну від паралельного, програмі, як правило, відповідає єдиний процес. У паралельному програмуванні програмі може відповідати безліч процесів (кожен процесс-це, наприклад, гілка алгоритму).

Пересилки можуть здійснюватися через оперативну пам'ять (SMP) або за допомогою пересилки повідомлень (MPP).

Якщо в SMP системах різні процеси використовують одні і ті ж дані, то в цьому випадку існує поняття критичної ділянки (області) програми, яка повинна виконуватися з винятковим правом доступу до спільних даних, на які є посилання в цій програмі. Процес, який готується увійти в

критичну область, може бути затриманий, якщо будь-який інший процес в цей час знаходитися в критичній області. Одним із засобів організації критичної області є семафор. Крім того, дані одних процесів необхідно передавати іншим. При цьому існує таке поняття, як очікування події (тобто зміна стану семафора).

У МРР системах при обміні даними за допомогою пересилки повідомлень слід координувати виконання декількох гілок. При цьому існує поняття бар'єру. Існують бар'єри декількох типів. При використанні бар'єру одного типу передбачається, що усі процеси деякої групи повинні досягти певної точки своєї програми (бар'єру), перш ніж будь-якому з них буде дозволено рушити далі. Варіантом організації бар'єру може бути однониткова критична секція, що йде за бар'єром, з однієї гілки, яку виконує будь-який (але тільки один) процес із групи процесів. Модернізація цього типу бар'єру полягає у здатності тільки останнього з прибулих до бар'єру процесу виконувати критичну секцію з однієї гілки.

Недоліками пересилок є виникнення оборотних і безповоротних блокувань (тупік). Оборотне блокування (Livelock) означає ситуацію, коли два і більше паралельних процесів зациклюються, тобто обмінюються повідомленнями, не виконуючи при цьому ніякої корисної обчислювальної роботи. Процеси знаходяться в такому стані до тих пір, поки не вичерпають виділені ним ресурси. Безповоротне блокування (Deadlock) - це ситуація, коли два або більш процеси чекають події, яка повинна статися у процесі, що знаходиться у тому ж заблокованому стані.

Для паралельних програм можлива ситуація недетермінованості результату. Це означає, що при одних і тих же початкових даних паралельна програма може давати різні результати. Це може відбуватися через розкид часів виконання гілок програми, що виникає в умовах так званих перегонів. Неправильна синхронізація призводить до недетермінованості результатів програми.

Розглянемо ситуацію недетермінованості, що виникає при виконанні наступної програми на Fortran

```
Do 100 I=1,N
CREATE CALC(I)
100 CONTINUE
```

У кожному процесі CALC використовується формула (B-A)/N*(I - 1)+A

Коли така програма виконувалася на традиційному комп'ютері, проблем не було, але при використанні ПКС, результати були кожного разу різні, скільки б разів не виконували програму. Більше того, жодна відповідь не була правильна. Чому це сталося? З'ясувалося, що причиною некоректної поведінки програми був неумисний розподіл даних, який

виникав в операторові CREATE. У FORTRAN усі параметри передаються по посиланню, а це означає, що адреса змінної I передається при кожному виклику підпрограми CALC і таким чином, стає загальним для усіх N процесів і початкового процесу. При створенні кожного процесу відбувається звернення до I, проте, в той же самий час, початковий процес виробляє зміну змінної I як частину циклу DO. Залежно від точності синхронізації і планування процесів, будь-який процес міг отримати або призначене для нього I або більш пізніше I . Для усунення цієї помилки відкоригуємо:

```
DO 100 I=1,N
IARG(I)=(I)
CREATECALC(IARG(I))
100 CONTINUE
```

Тут ми вводимо додатковий масив IARG, елементами якого будуть усі значення I . Коли вони будуть заповнені, будуть створені підпрограми CALC зі своїми аргументами.

Таким чином, розробка ПЗКС пов'язана з рішенням наступних проблем :

1. Виключення взаємних блокувань і даремних обмінів повідомленнями.
2. Захист від небажаних умов змагання.
3. Уникнення створення занадто великого числа паралельних процесів.
4. Виявлення завершення програми простим способом.

Перерахуємо також низку нових запитань, що виникають при проектуванні паралельних програм :

1. Розмір окремих компонент (наприклад, гілок) паралельної програми.
2. Співвідношення розмірності завдання і системи (відображення віртуальних процесорів на фізичні)
3. Який спосіб синхронізації застосовувати краще?
4. Які дані слід зробити загальними для усіх процесів?
5. Як гарантувати детермінованість результатів?
6. Як розділити загальні дані на частини, щоб забезпечити найбільш ефективне використання устаткування паралельних КС?
7. Якщо необхідно, виконати розподіл по процесорах і маршрутизацію.
8. Введення-виведення даних.

Критерії оцінки паралельних програм.

1. Коефіцієнт прискорення.
2. Коефіцієнт завантаженості або ефективність роботи ПКС (максимальне число зайнятих процесорів при вирішенні задачі).
3. Витрати на синхронізацію.
4. Мінімізація часу на обмін даними.
5. Відсоток простою системи.
6. Часові витрати на пересилки і маршрутизацію.
7. Масштабованість програми.
8. Вплив розміру завдання на прискорення.

2.3 Мовні засоби опису синхронних паралельних процесів

Операції над масивами як засіб укрупнення об'єктів, над якими виконуються обчислення, з'явилося в мовах програмування задовго до створення векторних і матричних систем. Однак тільки з появою цих систем, операції над масивами у мовах програмування стали засобом прискорення обчислень. У традиційних мовах програмування типи даних описуються як скаляри або як індексована множина скалярів, що названі масивами. Це засоби для зручного і скороченого написання програм, що мають у своєму складі набір ідентичних операцій над різними даними.

Існує три методи опису масивів для паралельних КС.

1. Масиви, як послідовні об'єкти. У цьому випадку масиви описуються як послідовність скалярів. При зверненні до імені масиву за замовчуванням здійснюється послідовна обробка його елементів. Для паралельної обробки елементів масиву в тілі програми вказується набір індексів, по яких і здійснюється паралельна обробка. Приклад такого методу : розширений FORTRAN для системи STAR - 100.

2. Масиви, як паралельні об'єкти даних (альтернативний першому методу). Обробка по усіх елементах цього масиву виконується паралельно (за замовчуванням). Цей підхід є найбільш загальним і розглядається як варіант у будь-якому стандарті на обробку масивів. Тут усі масиви описуються як особливі об'єкти даних заданої розмірності. Будь-яке звернення до масиву матиме на увазі увесь масив. Проте цей підхід не відміняє використання масиву, як набору послідовних даних, коли це необхідно, для цього використовують механізм індексації для пониження рангу за допомогою вибірки. Наприклад, якщо заданий тривимірний масив **A**, то ранг дорівнює трьом і будь-яке звернення до імені **A** має на увазі паралельне звернення до усіх його елементів. Індексація будь-якої розмірності може бути розглянута як операція вибірки, яка знижує ранг

масиву **A** на одиницю. Якщо індексувати усіх три розмірності, то з **A** вибирається скалярна величина або об'єкти даних нульового рангу. Цей підхід був запропонований в розширеному FORTRAN для CRAY-1.

3. Масиви, як суміш послідовних і паралельних об'єктів. Це компромісний варіант. У цьому випадку обмежується число розмірностей і загальна кількість елементів масиву. Обмеженість масиву даних повністю залежать від фізичної структури КС, на якій буде виконується обробка цього масиву. Якщо розмірність масивів менше або дорівнює розмірності системи, обробка їх елементів здійснюється паралельно. У разі, коли розмірність масиву перевищує розмірність системи, виконується паралельно-послідовна обробка. Цей підхід був адаптований для багатьох матричних КС, де розмірності масивів, для яких можна здійснювати паралельну обробку, відповідає розміру і конфігурації самої КС. Цей підхід реалізований в мовах (ACTUS, CFD та ін.) для ILLIAC - 4. Так, мова CFD оперує одновимірними масивами з 64-х елементів, які перетворяться в матрицю для процесорів ILLIAV, - 4. Цей же підхід реалізований в DAP - FORTRAN, де паралельне звернення до масиву можна виконувати або по першій, або по першій і другій розмірностях. І ця розмірність повинна відповідати розміру КС DAP.

2.3.1 Розміщення масивів у пам'яті матричних КС

Як відомо, матричні КС (МКС) відносяться до систем з роздільною або локальною пам'яттю. Розміщення масивів в пам'яті цих систем впливає на показник реальної продуктивності (призначеною для користувача, продуктивності, при рішенні конкретної задачі). При виконанні матричних операцій стандартними способами розміщення елементів матриць у пам'яті МКС є їх розміщення по рядках або по стовпцях. Для паралельного виконання більшості матричних завдань необхідно здійснити одночасну вибірку як рядків матриць, так і їх стовпців. Згадані способи розміщення не забезпечують цієї вимоги. При розміщенні матриць по стовпцях можлива паралельна вибірка елементів рядків і послідовна вибірка елементів стовпців, а при розміщенні по рядках навпаки - паралельна вибірка елементів стовпців і послідовна вибірка елементів рядків. Для одночасної вибірки як стовпців, так і рядків матриці зручним способом є так званий "скошений" метод, наведений нижче.

ПЕ0	ПЕ1	ПЕ2	ПЕ4
$\mathbf{a_{11}}$	a_{12}	a_{13}	a_{14}

a_{24}	$\mathbf{a_{21}}$	a_{22}	a_{23}
a_{33}	a_{34}	$\mathbf{a_{31}}$	a_{32}
a_{42}	a_{43}	a_{44}	$\mathbf{a_{41}}$

Цей спосіб дозволяє мінімізувати час виконання матричних завдань. Для забезпечення його працездатності у системі має бути передбачена можливість використання фігурної адресації, або структура матричної системи повинна мати вигляд, представлений на рис. 2.1.

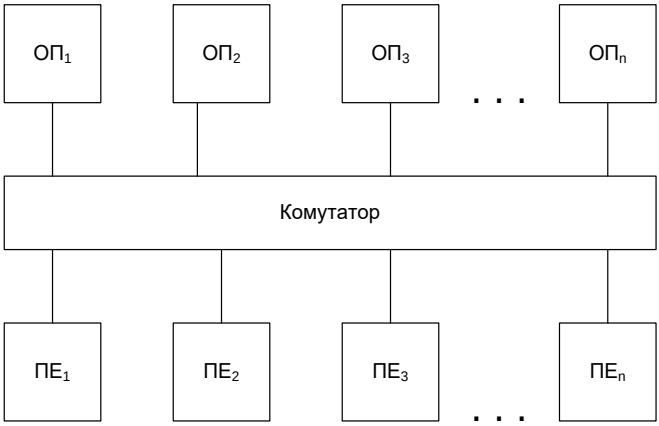


Рис. 2.1. Структура матричної КС

Важливим чинником підвищення користувацької продуктивності є мінімізація часу, що витрачається на пересилку даних. Зменшення цього часу можливо за рахунок адаптації алгоритму обчислень до структури МКС. Розглянемо приклад виконання завдання перемноження двох матриць, тобто. C=A*B, де A,B,C - матриці розмірністю n*n, а n≤64у системі ПС - 2000 (рис. 2.2.). У цьому випадку

$$c_{i,k} = \sum_{j=1}^{n} a_{i,j} * b_{j,k} \ \ (i,k = 1,2,\dots,n)(1)$$

Відомо, що у системі ПС- 2000 мінімізація часу, що витрачається на обмін даними, пов'язана з використанням локальних пересилок по

швидких регулярних каналах (РК). Використання ж пересилок по послідовному магістральному каналу (МК) призводить до збільшення часу пересилок даних.

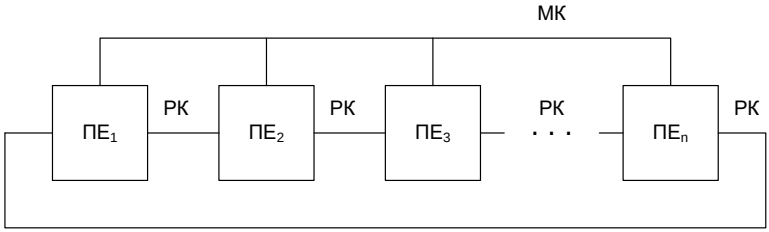


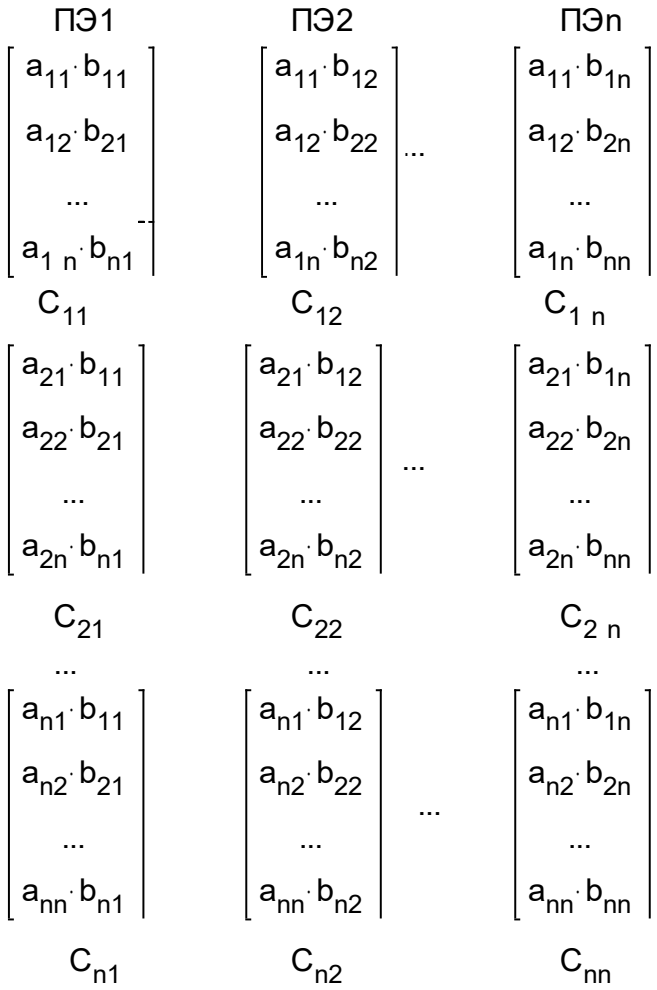
Рис.2.2. Структура системи ПС-2000

Будемо розміщувати елементи матриць A, B і C у пам'яті ПЕ по стовпцях, тобто в першому ПЕ - 1-ші стовпці матриць, у другому ПЕ - 2-гі стовпчі матриць і т.д.

ПЕ1 ПЕ2 ПЕn

$$\begin{pmatrix} a_{11} & b_{11} & c_{11} \\ a_{21} & b_{21} & c_{21} \\ \vdots & \vdots & \vdots \\ a_{n1} & b_{n1} & c_{n1} \end{pmatrix} \quad \begin{pmatrix} a_{12} & b_{12} & c_{12} \\ a_{22} & b_{22} & c_{22} \\ \vdots & \vdots & \vdots \\ a_{n2} & b_{n2} & c_{n2} \end{pmatrix} \quad \begin{pmatrix} a_{1n} & b_{1n} & c_{1n} \\ a_{2n} & b_{2n} & c_{2n} \\ \vdots & \vdots & \vdots \\ a_{nn} & b_{nn} & c_{nn} \end{pmatrix}$$

Тоді виконуючи обчислення за формулою (1), отримаємо наступну схему обчислень:



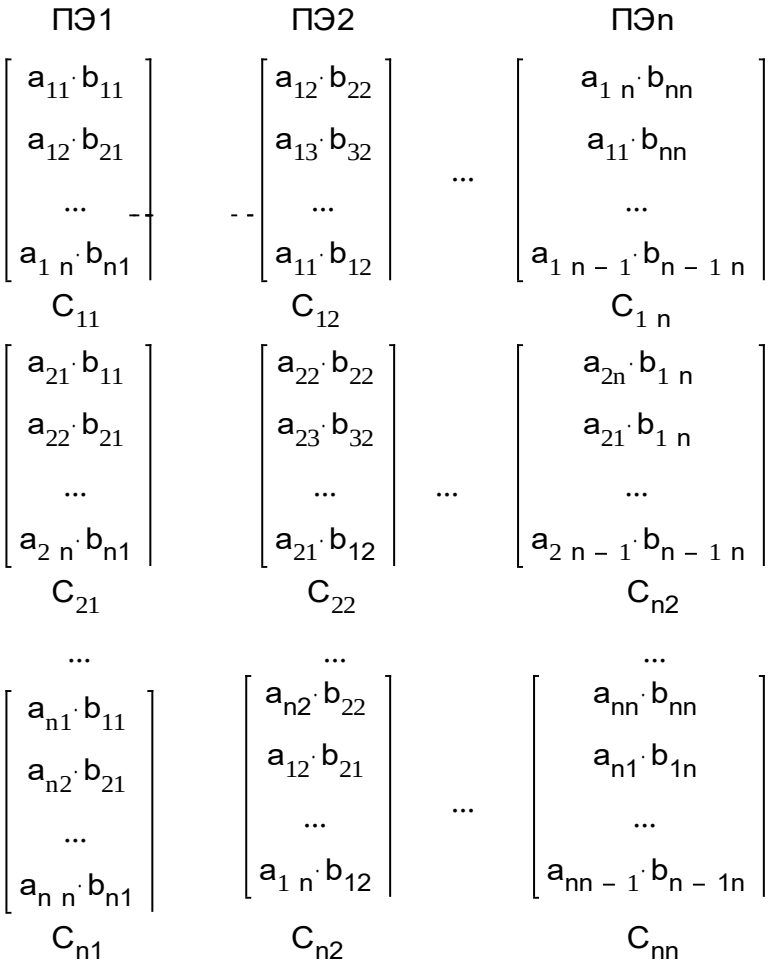
Використання даної схеми обчислень, з урахуванням розміщення матриць в ПЕ, призводить до великих часових витратах на обмін даних між ПЕ. Тут використовуються як швидкі РК, так і повільний МК. З метою зменшення витрат часу на обмін даними співвідношення (1) перетворимо до вигляду:

$$C_{i,k} = \sum_{j=1}^{n} a_{i,t} * b_{t,k} \qquad (2)$$

де $t = (r + j) \bmod n$ (*r* - номер ПЕ, *r= 1,2,...,n*).

Це призводить до циклічного зсуву на r позицій часткових добутків при їх додаванніу кожному r-ому ПЕ.

Схема обчислень із використанням співвідношення (2) має наступний вигляд:



Таким чином, друга схема обчислень відрізняється від першої порядком зберігання елементів стовпців матриць в ПЕ, а також порядком обчислення часткових добутків, що призводить до мінімізації часу пересилки даних за рахунок використання тільки регулярних каналів і за рахунок зменшення загальної кількості пересилок.

2.3.2 Способи вибірки об'єктів масивів

При матричних обчисленнях можливі ситуації, коли потрібне звернення не до усього масиву, а тільки до його частини. Наприклад, вибіркою з масиву може бути вектор- рядок, вектор- стовпець, діагональ матриці, матриця в тривимірному масиві. Можуть знадобитися вибірки і більш складних конфігурацій, у тому числі з використанням непрямої адресації. Розглянемо різні способи вибірки елементів масиву :

1. Вибірка об'єктів пониженого рангу.

Якщо заданий масив **A** розмірності (чи рангу) *n*, то індексація по будь-якій з розмірності, знижує ранг на одиницю.

A(*,*); A - паралельна вибірка усіх елементів.

A(I, *); A(I,)- паралельна вибірка і -го рядка;

A(*, J): A(,J) - паралельна вибірка j -го стовпця.

A(I,J) - звернення до одного елементу.

2. Вибірка діапазону значень.

У даному випадку замість пониження рангів об'єктів масиву зменшується розмір цього об'єкту. Тому діапазон повинен специфікувати піднабір усього діапазону розмірності. Можливі наступні варіанти специфікацій :

а) Відрізки допустимої області зміни індексу по відповідному виміру, який може задаватися нижньою і верхньою межею зміни індексу. У цьому випадку крок зміни індексу дорівнює 1.

Наприклад, A(1:3, 2:7).

б) Послідовність діапазону зміни індексів із заданим кроком. Крок зміни індексів може бути заданий різними способами. Найбільш відомі з них наведені у прикладах 1 і 2. Приклад 1: A(1:5:2, 2:8:2). У даному прикладі по кожному виміру масиву вказується нижня межа, верхня межа і крок зміни індексу. Таким чином,у цьому прикладі відбувається вибірка елементів масиву **A** з наступними значеннями стовпців: 2,4,6,8 і рядків: 1,3,5. Приклад 2: A(1:(2)5, 2:(2)8). У даному випадку крок зміни індексів зазначено у дужках. У цьому прикладі виконується вибірка тих же елементів масиву **A**, що і в прикладі 1. Відмінність полягає лише у синтаксисі.

Крім того можливі два способи формування індексів:

- **Декартовий:** заздалегідь визначається швидкість зміни індексів. Нехай швидше змінюється лівий індекс.

Наприклад, маємо **A**– матрицю з 10 * 10 елементів, тоді **A** (1:3,5:9) - підмасив, що складається з 15 елементів: A (1:5), A (2:5), A (3:5), A (1:6), ... A (3:9),

- **Синхронний:** рівна швидкість зміни індексів по всіх вимірах. **A**<1:10, 1:10>-задається головна діагональ,**A**<1:9, 2:10>-задається нижня діагональ матриці, розташована під головною діагоналлю матриці.

3. ***Вибірка елементів масиву за допомогою цілочисельних векторів.***

Тут використовується непряма адресація, яка задається, як правило, за допомогою спеціальних векторів. Нехай задані цілочисельні масиви IVi JV виду

IV→ 1,1,1,3 JV→2,1,3

також заданий двовимірний масив **A** виду

$$\mathbf{A} \rightarrow \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ k & l & m & n \\ o & p & r & s \end{bmatrix}$$

$$\text{Тоді } A(IV, *) = \begin{bmatrix} a & b & c & d \\ a & b & c & d \\ a & b & c & d \\ k & l & m & n \end{bmatrix}$$

$$A(*, IV) = \begin{bmatrix} a & a & a & c \\ e & e & e & g \\ k & k & k & m \\ o & o & o & r \end{bmatrix}$$

$$A(IV, JV) = \begin{bmatrix} b & a & c \\ b & a & c \\ b & a & c \\ l & k & m \end{bmatrix}$$

При синхронній вибірці:

A<IV, *>→abcn
A<*, IV>→aekn

4. Вибірка за допомогою булевих масивів.

Для вибірки елементів використовуються логічні масиви. Вибірка здійснюється залежно від значення істинності логічних елементів заданого масиву. Таким чином, і -й елемент може бути вибраний з одновимірного масиву за допомогою індексації його одновимірним булевим масивом зі значенням "істинно" в і-му елементі. Аналогічно можна використовувати такий логічний масив для вибірки рядка або стовпця з двовимірного масиву. Нехай заданий логічний масив виду

$$LV \rightarrow 0\ 0\ 1\ 0$$

Розглянемо приклади вибірки за допомогою масиву LV із матриці **A,** описаної вище:

A(LV,*) → k l m n
A(* , LV) → c g m r
A(LV,LV) →m.

Окрім вибірки масиву за допомогою булевого вектора, у мовах паралельного програмування для синхронних КС існує аналог оператора умовного оператора при виконанні векторної операції. Цей оператор визначає необхідність виконання операції над тим або іншим елементом масиву залежно від результату попередніх обчислень. Для вибірки за допомогою булевих масивів існує оператор *IF*. У мові *Fortran* він позначається як *IFA* (щоб не плутати з класичним оператором IF) і представляється у вигляді *IFA(L) e,* де *L* - умова, *e* - виконавчий оператор. Для виконання цього оператора масиви, що входять в *L* і *e*, мають бути однієї і тієї ж розмірності. Наприклад, нехай *A, B, C, D, F* -масиви однакової розмірності. У операторі *IFA(A.LT.B) C=D+F* операція складання елементів масивів *D* і *F* та привласнення отриманої суми відповідним елементам *C* буде виконуватись тільки у тих випадках, коли для відповідних елементів масивів *A* і *B* логічний вираз *A.LT.B* істинно, тобто, якщо *a34<b34*, то *c34=d34+f34*, якщо ж *a34 >=b34*, то значення *c34* не зміниться.

2.3.3 Функції відповідності розмірності масивів

Основним правилом при паралельній обробці масивів є те, що два операнди повинні мати однаковий ранг і одну й те саму область у

відповідних розмірностях. Обмеживши представлення операндів таким чином, можна здійснити набагато більший контроль за появою помилок як під час компіляції, так і під час роботи. Для забезпечення цього обмеження, у мовах використовуються конструкції, які надають можливість стискання, розширення або переформатування масивів із метою їх відповідності. Організація такої відповідності покладається на програміста. Розглянемо детальніше функції отримання відповідності розмірності масивів.

У мовах, що передбачають паралельну обробку масивів, мають бути:

1. Функції **стискання** (пониження рангу). Як було показано вище, ранг об'єкту масиву може бути знижений за допомогою індексації або вибірки. Іншим способом, при якому він може бути знижений, являється повторне використання двійкового оператора між усіма елементами в одній розмірності масиву. Він, безумовно, може бути описаний на мові як послідовність операцій, проте це не дає можливості використання паралелізму для пониження рангу. Наприклад, сума з N елементів може бути отримана при паралельному виконанні $\log_2 N$. кроків. Наведемо перелік найбільш популярних функцій, що використовують пониження рангу, де кожна з них має два параметри, масив і розмірність, по якій буде виконано пониження : SUM(A, k), PROD(A, k), AND(A, k), OR(A, k), MIN(A, k), MAX(A, k). Так SUM(A, k), означає функцію додавання елементів масиву A k-ої розмірності.

2. Функції **розширення** (підвищення рангу). Часто для досягнення відповідності необхідно підвищити ранг об'єкту. Підвищення рангу необхідно у двох випадках: 1) для операцій між скалярами і векторами. 2) для операцій між векторами і матрицями. У першому випадку, як правило, функція пониження рангу не застосовується. Мови допускають послаблення правил відповідності для операцій скаляр- вектор, тобто, є можливість довільно змішувати скаляри і масиви. У другому випадку функції розширення зводяться до повторення вектора або як рядок, або як стовпчик, щоб добитися відповідності з матрицею. Прикладом оператора підвищення рангу є *XPND(A, k, N),* в якому *A* - ім'я вектора*; k* - розмірність по якій використовується повторення (1 - по рядку, 2 - по стовпчику); *N* - число повторень.

Нехай існує масив A→abcd. Тоді:

$$XPND(A, 1, 3) \rightarrow \begin{matrix} a & b & c & d \\ a & b & c & d \\ a & b & c & d \end{matrix}$$

$$XPND(A, 2, 3) \rightarrow \begin{matrix} a & a & a \\ b & b & b \\ c & c & c \\ d & d & d \end{matrix}$$

3. Функція **переформатування** масивів. Ці функції використовуються тоді, коли потрібна відповідність між масивами різної розмірності, але які містять однакове, загальне число елементів (прикладом є алгоритми швидкого перетворення Фур'є).

Для виконання цієї функції в мові *Fortran* використовується два оператори: *DIMENSION і MAP.*

Оператор *DIMENSION* визначає динамічний діапазон масиву, а оператор *MAP* використовується для зміни форми масиву. Наприклад, нехай є лінійний масив (вектор) **A**, що складається з *N* елементів. Необхідно переформатувати його в двовимірний масив. Фрагмент такої програми можна представити наступним чином:

```
DIMENSION A(N)
.
.
.
MAP A(N/2, 2).
```

У даному прикладі показано, що початковий вектор **A** з *N* елементів за допомогою оператора *MAP* перетворений в двомірний масив *N/2*2.* Таке перетворення коректне, коли *N* - парне. Підкреслимо, що обов'язковою умовою такого перетворення є одне і теж число елементів у початкового і результуючого масивів.

2.3.4 Мова Parallaxis

Основні характеристики

Parallaxis - це машинно-незалежна мова, створена на базі *Modula - 2*. Програми на мові *Parallaxis* призначені для роботи на SIMD системах, кластерних, робочих станціях або однопроцесорних комп'ютерах (таких, як SIMD системи MP- 1 і MP- 2, кластерні станції, які використовують PVM, Intel Paragon, однопроцесорні версії для майже усіх UNIX систем : SUN SPARC - станцій (ОС Solaris), DEC -станціях, HP 9000, IBM RS 6000, IBM PC з LINUX. ПЗ *Parallaxis* охоплює компілятори для паралельних і однопроцесорних КС, дебагери, безліч прикладного ПЗ для різних

застосувань, особливо для обробки зображень. Конструкції мови орієнтовані на паралельну обробку завдань з природним паралелізмом. Основна особливість мови *Parallaxis* - це програмування на абстрактному рівні на віртуальній структурі SIMD -системи. Таким чином, кожна програма включає окрім опису алгоритму завдання також опис віртуальної SIMD -моделі системи.

SIMD -модель включає основний процесор (попередньої обробки) Фон-Неймановської архітектури, призначений для скалярної обробки і матрицю процесорних елементів (ПЕ) для векторної обробки (рис. 2.3.). Матриця процесорів є множина ідентичних ПЕ. Основний процесор зв'язаний з ПЕ за допомогою шини. Зв'язки між ПЕ можуть бути довільними і задаються у програмі. Крім того, у програмі можна задавати множину віртуальних систем. Їх число залежить від кількості і структур масивів, що обробляються у програмі. Структури SIMD -моделей абсолютно не залежать від фізичної структури системи. Перед виконанням відбувається відображення віртуального масиву на фізичну структуру комп'ютерної системи. Якщо число віртуальних ПЕ більше, ніж число фізичних ПЕ, то кожен фізичний ПЕ відповідає за множину віртуальних ПЕ і оперативна пам'ять цих віртуальних ПЕ вважається єдиною.

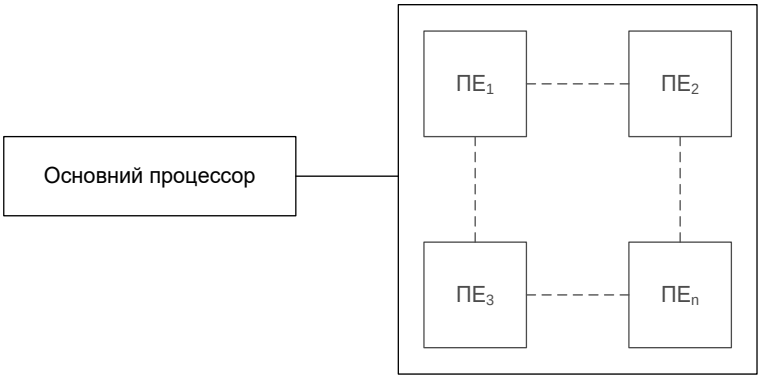


Рис.2.3. SIMD–модель системи

Структура програми

Програма складається з головного модуля (який у свою чергу може викликати безліч інших модулів) і має наступний вигляд:

```
MODULE < ім'я >
    <опис конфігурації віртуальної системи>
    <опис скалярних і векторних даних>
```

```
BEGIN
    <введення векторних і скалярних даних>
    <оператори Parallaxis>
    <виведення результатів>
END.
```

Опис кофігурації

Оператор опису конфігурації призначений для визначення імені, кількості та розташування ПЕ у віртуальній системі за аналогією з операторами опису масивів і в загальному вигляді представляється наступним чином:

CONFIGURATION <ім'я віртуальної системи><кількість і розташування ПЕ>

Розглянемо приклади операторів опису конфігурації:

```
CONFIGURATION list[1..4096]
```

У наведеному операторі описана конфігурація системи з ім'ям list, що представляє собою вектор ПЕ (розмірність один) з 4096 елементів.

```
CONFIGURATION grid[1..64][1..64]
```

У наведеному операторі описана конфігурація системи з ім'ям grid, що представляє собою матрицю ПЕ (розмірність два), яка включає 64 рядки і 64 стовпця.

```
CONFIGURATION hyper[0..1][0..1][0..1]
```

У наведеному операторі описана система з ім'ям hyper представляє собою гиперкубічну конфігурацію ПЕ розмірністю три. У даному випадку число елементів дорівнює восьми з номерами: 000; 001; 010; 011; 100; 101; 110; 111.

Опис зв'язків між ПЕ

Оператор опису зв'язків між ПЕ віртуальної системи у загальному вигляді може бути представлений таким чином:

```
CONNECTION <ім'я зв'язку>: <опис зв'язків>.
```

При описі зв'язків використовуються символи ”→” і “↔”. Перший з них означає однонаправлений зв'язок, а другий - двонаправлений зв'язок.

Існує чотири типи зв'язків:

1) Один в один.
2) Один в множину.
3) Бінарні зв'язки.
4) Багатовимірні (гиперкубічні).

Розглянемо приклади опису різних зв'язків у віртуальних системах.

```
CONNECTION north : grid[i,j]→grid[i-1,j];
CONNECTION east : grid[i,j]→grid[i,j+1];
CONNECTION west : grid[i,j]→grid[i,j-1];
CONNECTION south : grid[i,j]→grid[i+1,j];
```

У наведених вище операторах описуються однонаправлені зв'язки конфігурації grid будь-якої розмірності у чотирьох напрямках (тип зв'язку - один в один).

```
CONNECTION brd : gride[i,1]→grid[i, 2..32];
```

У наведеному прикладі описуються однонаправлені зв'язки конфігурації grid першого елемента кожного рядка з елементами 2-32 цього ж рядка (тип зв'язку - один у множину).

Для опису бінарних зв'язків розглянемо конфігурацію віртуальної системи, що складається з 15 ПЕ і організованою у вигляді бінарного дерева:

```
CONFIGURATION tree [1..15];
```

Тоді оператори опису бінарних двонаправлених зв'язків мають такий вигляд:

```
CONNECTION l child : tree[i] ↔tree[2*i];
CONNECTION r child : tree[i]↔tree[2*i+1];
```

Розглянемо приклад опису двонаправлених багатовимірних (гиперкубічних) зв'язків для конфігурації hyper, представленої вище:

```
CONNECTION axis[1] : hyper[i,j,k]↔hyper[(i+1) mod 2, j,k];
CONNECTION axis[2] : hyper[i,j,k]↔hyper[i, (j+1) mod 2, k];
CONNECTION axis[3] : hyper[i,j,k]↔hyper[i,j,(k+1) mod 2];
```

Опис векторних і скалярних даних

У даній мові існує два типи даних:

• Скалярні;

• Векторні.

Скалярні дані розташовуються на керуючому ПЕ. При описі векторних даних закладається зв'язок з ПЕ. Вони розподіляються між ПЕ віртуального масиву. Таким чином, структура векторних даних повинна відповідати структурі віртуального масиву. Оператори опису векторних даних має наступний вигляд:

VAR<ім'я>: <ім'я конфігурації> OF <тип даних>;

Розглянемо приклад оператор опису векторних даних:

VAR Y : hyper of REAL;

У наведеному операторі описується тривимірний масив даних Y типу REAL, розташованих у ПЕ віртуального гіперкуба.

Векторні дані для різних конфігурацій можуть одночасно існувати в одній і тій же програмі.

Основні функції мови Parallaxis

Існує набір функцій, що дозволяють визначити позицію заданого ПЕ у віртуальному масиві і певну інформацію про віртуальний масив. Перерахуємо деякі з них:

1. DIM (config, axis). Ця функція має два аргументи: ім'я конфігурації та номер розмірності. Вона повертає позицію кожного віртуального ПЕ всередині даної розмірності.

2. LOWER (config, axis). Ця функція має ті ж аргументи, що і попередня і повертає min значення даної розмірності масиву.

3. UPPER (config, axis). Ця функція має ті ж аргументи, що і попередня і повертає max значення даної розмірності масиву.

4. LEN (config, axis). Ця функція з тими ж агрумент, що і попередні. Вона повертає номер віртуального ПЕ в масиві.

5. LEN (config). Ця функція визначає загальне число ПЕ, що складають дану конфігурацію.

6. ID (config). Ця функція визначає номер кожного ПЕ в масиві.

Паралельні операції

Звернення по імені означає паралельне виконання операцій над елементами векторних даних. Так, якщо x, y і z описані як векторні дані однієї і тієї ж конфігурації, то оператор

$$x := y + z;$$

означає одночасне додавання значень y і z та привласнення отриманого результату x для всіх елементів цих даних.

Для того, щоб виконати операцію над обмеженим числом елементів векторних даних може бути застосований умовний оператор.

Приклад 1: VAR X, Y, Z tree of REAL; - масиви.

IF Z <> 0.0 then X: = Y / Z; - ця операція паралельна над частинами X, Y, Z. (ПЕ з нульовими Z будуть пасивними)

END;

У даному прикладі нульові значення Z обмежують виконання операція X: = Y / Z над усіма масивами даних. Ця операція буде виконуватися тільки для тих елементів X, Y, Z, де Z приймає ненульові значення.

Приклад 2: VAR X, Y, Z tree of REAL; - масиви.

IF Z <> 0.0 then X: = Y / Z; - ця операція паралельна над частинами X, Y, Z. (ПЕ з нульовими Z не будуть виконувати цю операцію)

ELSE

X: = Y + 10; - це паралельна операція над частинами X, Y, Z. (ПЕ з ненульовими Z не будуть виконувати цю операцію)

.

END;

У даному прикладі масиви даних X, Y і Z діляться на дві активні групи: операція X: = Y / Z виконується для тих елементів X, Y, Z, де Z приймає ненульові значення, а операція X: = Y + 10 для елементів X , Y, Z, де Z приймає нульові значення. Тут ми бачимо нетрадиційне виконання оператора умовного переходу.

Функції згортки

Дані функції виконуються тільки над векторними даними, в результаті яких формується скаляр. У загальному вигляді такі функції можна представити таким чином:

<scalar>: = REDUCE. (sum, product, min, max, and, or, first, last) (<ім'я векторного даного>);

де sum - сума значень елементів векторних даних; product - добуток значень елементів векторних даних; min і max - відповідно мінімальне і

максимальне значення елементів векторних даних; and і or - логічні функції "І" і "АБО" значень елементів векторних даних;, first і last - відповідно значення першого і останнього активних ПЕ.

Оператори COMMUNICATION

Умові Parallaxis існує два види операторів комунікації:

- Regular communication (Регулярнізв'язки).
- General communication (Універсальнізв'язки).

При використанні першого виду операторів передбачається використання стандартних зв'язків між ПЕ, які описані в операторі конфігурації. При використанні другого виду не передбачається обов'язкове використання зв'язків, описаних у конфігурації. Тут застосовується непряма адресація (використовуються адреси - «куди переслати» і «звідки отримати»). Операції пересилань можуть бути використані як для передачі даних при виконанні прикладної задачі, так і для переупорядкування масиву даних (для здійснення різних варіантів їх вибірки).

Регулярні зв'язки

Існують такі варіанти операторів:

- MOVE. <Ім'я комунікації>;
- SEND. <Ім'я комунікації>; або
- RECEIVE. <Ім'я комунікації>;

Зазначимо, що MOVE і RECEIVE є функціями, а SEND - процедура. При використанні функції MOVE обидва віртуальних ПЕ - джерело даних і приймач даних повинні знаходитися в активному стані, в той час, як при використанні функції RECEIVE і процедури SEND в активному стані повинен знаходитись лише один віртуальний ПЕ, відповідно, для RECEIVE - ПЕ-приймач, а для SEND - ПЕ-джерело.

Приклади регулярних зв'язків:

```
CONFIGURATION linear[1..5];
CONNECTION right : linear[i]→linear[i+1];
VAR X, Y : linear of INTEGER;
IF ID(linear)<>2 THEN Y := MOVE.right(X); --ID – ідентифікатор процесора.
```

Проаналізуємо приклад, зображений на рис.2.4. У операторі IF за допомогою функції ID визначаються активні віртуальні ПЕ (усі, крім другого ПЕ). Таким чином визначаються неактивні віртуальні ПЕ для конфігурації джерела (X) і приймача даних (Y). У разі, коли і-й елемент

ПЕ-джерела не визначено (наприклад, 0-й, який не існує або 2-й - неактивний), то перезапис проводиться не з і-го в (і +1)-й, а з (і +1)-го в (і +1)-й. 2-й елемент ПЕ-приймача є незмінним.

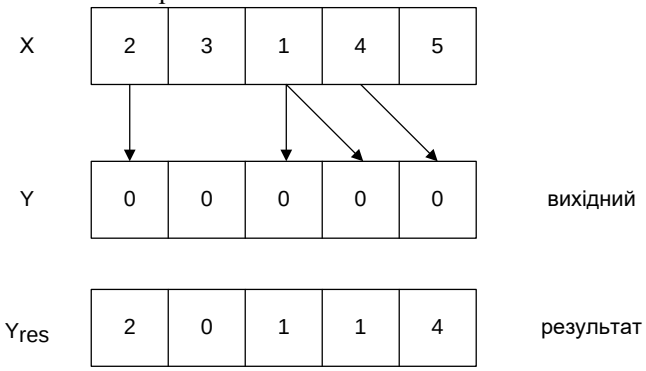


Рис.2.4. Застосуванняфункції MOVE

```
CONFIGURATION linear[1..5];
CONNECTION right : linear[i]→linear[i+1];
VAR X, Y : linear of INTEGER;
IF ID(linear)<>2 THEN Y := RECEIVE.right(X); --ID – ідентифікатор процесора.
```

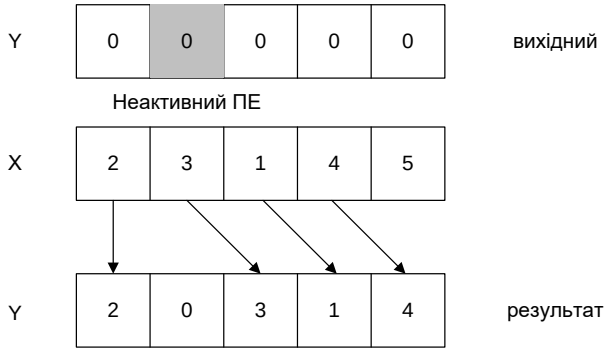


Рис.2.5.Застосування функції RECEIVE

У прикладі, зображеному на рис.2.5., в операторі IF за допомогою функції ID визначаються активні віртуальні ПЕ для конфігурації приймача даних (Y). Оскільки 2-й елемент є неактивним, то його значення

залишається незмінним, а інші значення елементів змінюються, як зазначено на рис.2.5.

```
CONFIGURATION linear[1..5];
CONNECTION right : linear[i]→linear[i+1];
VAR X, Y : linear of INTEGER;
IF ID(linear)<>2 THEN SEND.right(X,Y); --ID – ідентифікатор процесора.
```

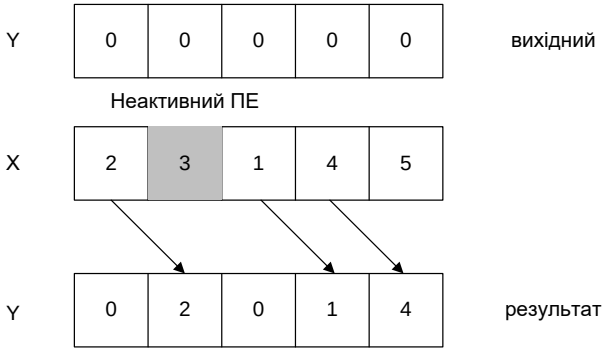


Рис.2.6.Застосування процедури SEND

У прикладі, зображеному на рис.2.6., в операторі IF за допомогою функції ID визначаються активні віртуальні ПЕ для конфігурації джерела даних (X). Оскільки 2-й елемент є неактивним, то значення 3-го елемента масиву Y залишається незмінним, незмінним залишається також значення 1-го елемента Y, а інші значення елементів змінюються, як зазначено на рис.2.6.

Універсальні зв'язки

Для організації універсальних зв'язків можуть бути використані два види операцій:

- SEND.<<index>>
- RECEIVE.<<index>>

де - index - це масив, в якому записуються адреси віртуальних ПЕ-джерел або ПЕ-приймачів даних.

Приклади реалізації універсальних зв'язків:

```
CONFIGURATION linear[1..5];
CONNECTION right linear[i]→linear[i+1];
```

VAR X, Y, IA: linear if INTEGER;
Y := RECEIVE.<<IA>>(X); -- Y отримує з X те, що лежить за адресою IA.

CONFIGURATION linear[1..5];
CONNECTION right linear[i]→linear[i+1];
VAR X, Y, IA: linear if INTEGER;
SEND.<<IA>>(X,Y); -- послати з X в Y і розмістити за адресою IA.

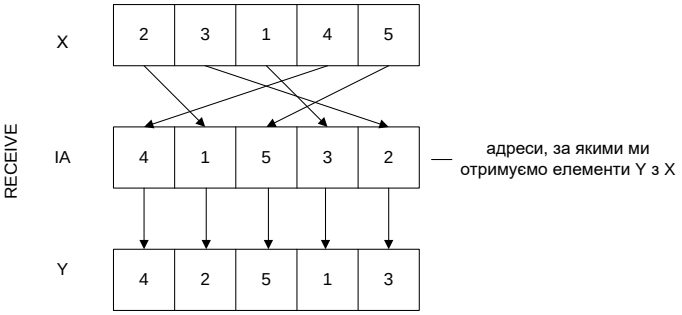


Рис.2.7. Приклад застосування операції RECEIVE

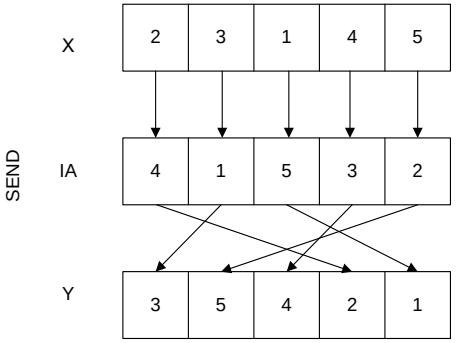


Рис.2.8. Приклад застосування операції SEND

2.4 Мовні засоби опису асинхронних паралельних процесів

Існує безліч прикладних задач, які реалізуються за допомогою алгоритмів, що допускають паралельну обробку, але погано векторизованих. До таких завдань відносяться деякі задачі моделювання

складних систем, таких як моделі нафтових родовищ та інші. Ці завдання реалізуються в системах класу МКМД, які, як відомо, відносяться до асинхронного типу. У асинхронної КС, як уже згадувалося раніше, не існує загального синхронізуючого сигналу (машинного такту системи). Тому в цих системах необхідно виконувати синхронізацію програмними засобами. Розглянемо основні причини, що викликають необхідність синхронізації:

- обмін інформацією між гілками алгоритму (паралельними процесами);
- доступ до загальних ресурсів;
- завершення виконання гілок (якщо при виконанні паралельної програми допускається створення і знищення паралельних гілок, то при завершенні виконання гілки необхідна синхронізація).

2.4.1 Опис паралельних процесів

Кожен процес, який може бути виконаний паралельно з іншим (іншими) процесами, може бути описаний як завдання, процедура, підпрограма, функція, блок або навіть оператор (у ОККАМ), які мають спеціальну ознаку, що вказує на те, що дана частина програми може виконуватися одночасно з деякими іншими частинами цієї програми. Такою ознакою може бути службове слово:

- PARBEGIN;
- COBEGIN;
- PSUBROUTINE;
- PAR;
- PARPROCEDURE.

Так, наприклад, у мові ОККАМ застосовується службове слово PAR, яке означає, що наступні за ним перераховані процеси будуть виконуватися одночасно.

2.4.2 Програмні засоби ініціалізації та завершення паралельних процесів

Як вже згадувалося, в МКМД системах обробляється потік асинхронних процесів. У зв'язку з цим, необхідні мовні засоби

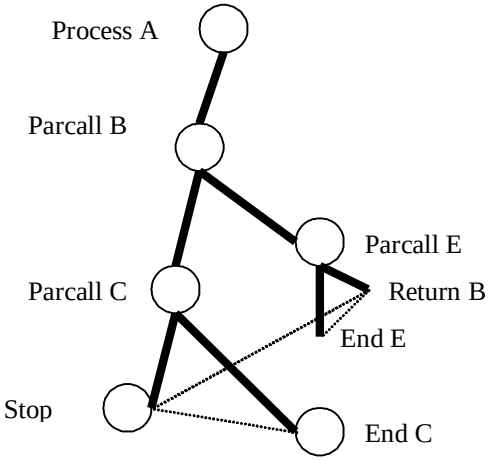


Рис.2.9. Застосування конструкції PAR CALL

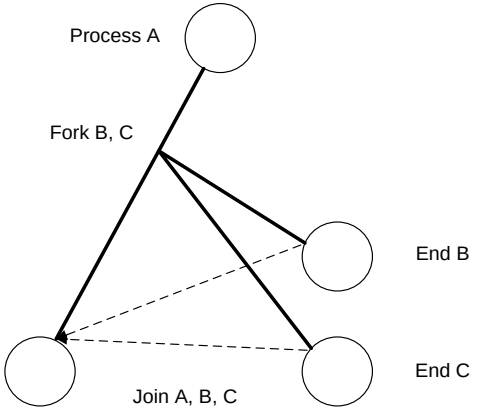


Рис.2.10. Застосування конструкції FORK.

ініціалізації процесів, а також їх завершення (для того, щоб знати, коли ініціювати подальші процеси). Ці засоби близькі (синтаксично) до мовних засобів звернення до підпрограм або процедур у даній мові. У мовах паралельного програмування використовується три типи конструкцій для ініціалізації та завершення паралельних процесів:

1. Породження одиночного процесу, внаслідок чого процес, що викликає, а також процес, який викликають виконуються одночасно (тобто паралельний процес розглядається як підпрограма, але виконується паралельно з програмою, що викликається). У цьому випадку для ініціалізації процесу використовується оператор, що включає службове слово PAR CALL, після якого йде ім'я процесу і параметри (у разі необхідності). Для завершення процесів можуть використовуватися оператори RETURN <ім'я процесу> або END <ім'я процесу>. Перший з них застосовується при завершенні процесів, які у свою чергу породжували ще які-небудь процеси (є одночасно тим, що викликає та, якого викликають), а другий при завершенні процесів, що не породжували ніяких процесів, тобто є тільки таким, якого викликали. Якщо процес був останнім у задачі, то при його завершенні використовується оператор STOP. Приклад застосування даної конструкції зображений на рис.2.9.

2. Одночасна ініціалізація множини процесів одним викликаючим процесом. У цьому випадку при ініціалізації процесів використовується оператор FORK <імена викликаючих процесів>, які будуть виконуватися до тих пір, поки не завершиться якийсь із них (за допомогою оператора END <ім'я процесу>) або ці процеси не об'єднаються (за допомогою оператора JOIN <імена об'єднуваних процесів>). Приклад застосування даної конструкції зображений на рис. 2.10.

3. Одночасне породження і завершення паралельних процесів. Це більш жорстка конструкція в порівнянні з попереднім варіантом. Для реалізації даної конструкції використовуються оператори COBEGIN <список процесів> - для ініціалізації та COEND <список процесів> - для завершення процесів. Приклад застосування даної конструкції зображений на рис. 2.11.

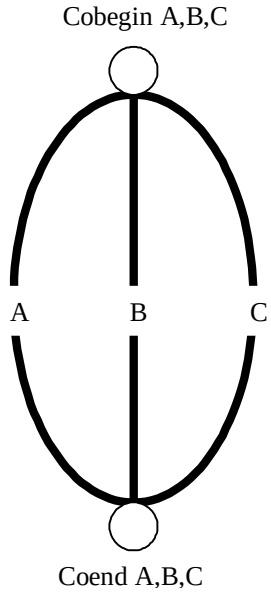


Рис. 2.11. Застосування конструкції COBEGIN

Слід зазначити, що перший і другий типи конструкцій передбачають асинхронну ініціалізацію і завершення паралельних процесів, у той час як третій тип передбачає синхронну ініціалізацію і завершення процесів.

Оператори RETURN, JOIN, COEND є фактично синхронізуючими примітивами типу "семафор". Процес, який породжує асинхронні паралельні процеси, перевіряють за допомогою семафорів,чи закінчилося виконання породжених ним процесів, і якщо ні, то чекає їх завершення, а якщо вони закінчилися, то породжуючий процеси також може завершитися.

Крім того, в паралельному Pascal передбачені так звані керуючі вирази. Вони дають можливість безпосередньо описувати порядок виконання асинхронних паралельних процесів. Синтаксис виразу має вигляд:

PATH <керуючий вираз> END

Керуючий вираз містить імена процесів і визначає порядок їх виконання. Список імен, що розділяються ";" означає послідовне виконання процесів. Список імен, що розділяються "," означає паралельне виконання перерахованих процесів. "n [керуючий вираз]" означає

можливість ініціювання зазначених процесів не більше, ніж n разів до їх завершення. "[Керуючий вираз]" означає необмежену кількість разів ініціювання перерахованих процесів до їх завершення.

Приклади:

PATH A, B, C, D - ці процеси виконуються паралельно.

PATH A; B; C; D - ці процеси виконуютьсяпослідовно.

PATH n [A; B; C; D] - ця група процесів буде ініційована n раз (циклічно).

Ініціалізація процесів супроводжується постановкою їх у чергу на виконання. Якщо в системі на момент ініціалізації процесів є вільні ресурси, то диспетчер ОС призначає процеси на процесори, тобто ініціалізація є фізичною. Якщо ж вільних процесорів немає, то процеси очікують звільнення ресурсів у черзі. Вибір вільного процесора для процесу засобами ОС, часто не призводить до оптимального рішення усієї задачі. Тому в деяких мовах паралельного програмування (наприклад, CI і OCCAM) передбачені засоби призначення процесів на ресурси користувачами, які можуть призвести до отримання більш високої реальної користувальницької продуктивності системи. Такі процедури у програмах називаються файлами конфігурації.

Розглянемо приклад файлу конфігурації на мові OCCAM:

PAR PLACED - говорить про те, що буде зазначено місце виконання паралельних процесів.

PROCESSOR 0 T4-тип процесора;

[послідовність процесів, виконуваних на процесорі 0]

PROCESSOR 1 T4

[послідовність процесів, виконуваних на процесорі 1].

Підіб'ємо підсумок: Перші три конструкції PARCALL, FORK, COBEGIN створюють деревовидні структури (ці оператори ставлять готові до виконання процеси у чергу, з якої вони вибираються по мірі звільнення процесорів).

Керуючі вирази в конструкціях [керуючий вираз] генерують циклічне виконання процесів. Але кожен знову ініційований процес повинен мати своє ім'я. Це ім'я йому дає ОС, яке програмісту невідомо, і якщо сталася помилка, її важко знайти. Це приклад того, що дуже великі мовні можливості в управлінні процесами мають свої недоліки.

2.4.3 Мовні засоби синхронізації доступу до загальних ресурсів

Послідовність операторів процесу, які претендують на використання деякого роздільного (загального) ресурсу, називають критичним інтервалом (областю, зоною, ділянкою). Програміст, створюючий

програму, повинен стежити за появою критичних інтервалів і забезпечувати такі умови, щоб до критичного інтервалу мав доступ тільки один процес, а під час його дії іншим процесам не дозволяли б доступ до загальних змінних. Така синхронізація при зверненні до загальних ресурсів зводиться до вирішення так званої задачі взаємного виключення.

При роботі з критичною ділянкою встановлюються такі правила:

1. Якщо критична ділянка вільна, процес може увійти в нього без будь-яких затримок.

2. Коли процес знаходиться всередині критичної ділянки, інші процеси, які намагаються увійти в ту ж критичну ділянку, затримуються на його вході.

3. Якщо процес залишає ділянку і є процеси, що претендують на неї, один з них отримує доступ на вхід.

4. Дисципліна вибору процесу з черги, які очікують входу, залежитьвід вимог, що пред'являються до системи, або від заданого пріоритету.

У якості загальних ресурсів найчастіше використовується спільна пам'ять (для ПКС із загальною пам'яттю) або її банки (у разі ПКС з розподіленою пам'яттю). Але взагалі загальні ресурси можуть бути будь-якими.

Існують наступні механізми, які застосовуються при вирішенні задачі взаємного виключення:

- прапорці;
- семафори;
- умовні критичні ділянки;
- монітори.

Розглянемо суть першого механізму. У кожному процесі, який претендує на загальний ресурс вводиться змінна, так званий прапорець. Процесам дозволяється зчитування всіх прапорців, але змінювати значення лише свого прапорця. Основу даного механізму становить правило: процес отримує доступ до загального ресурсу (входить у критичну ділянку) тоді, коли він "виставив" свій прапорець, у той час, як прапорці всіх інших процесів не "виставлені". Відповідно, після завершення роботи у критичній ділянці даний процес "знімає" свій прапорець і звільняє дорогу для інших процесів. Такий механізм працює добре до тих пір, поки кілька процесів одночасно не підходять до критичної ділянки. Тоді вони одночасно "виставляють" свої прапорці. Кожен перевіряє прапорці сусідів і виявляє, що критичний інтервал зайнятий. Виникає режим deadlock. Це головний недолік подібного механізму. Боротьба з цим недоліком можлива за рахунок введення випадкового запізнювання між виставлянням та зняттям прапорця в різних процесах. У загальному випадку такий механізм вимагає значних витрат ресурсів, як з точки зору обсягу оперативної пам'яті, так і

малоефективного використання часу процесора, зайнятого опитуванням стану прапорців інших процесів. Перевагою цього механізму є його простота.

Розвиток механізму прапорців пов'язано з введенням механізму семафорів, запропонованим Дейкстрою. Семафор - це невід'ємна змінна, для якої визначено дві операції P (пропустити) і V (звільнити). Зазвичай семафор S приймає два значення 0 і 1 та називається двійковим. Операція P на семафорі S виконується наступним чином. Перевіряється значення S. Якщо S> 0, тобто S = 1, то P (S) - істинно і це означає пропуск процесу в критичну ділянку і S = S-1. Таким чином, якщо в цей момент буде зроблена спроба увійти у цю критичну ділянку, то P (S) для інших процесів буде false(0) і вхід до критичної ділянки буде затриманий. Наприкінці критичної ділянки виконується операція V, яка змінює значення семафора S = S +1 і тим самим відкриває можливість входу у критичну область іншого процесу.

Недоліками семафорів є погана структуризація і виразність. У цьому механізмі не передбачені ніякі спеціальні конструкції виділення загальних (розподілених) даних. Тому програміст може включити використання загальних даних, як в критичному інтервалі, так і поза ним, що призводить до помилок. Крім цього виникають помилки, пов'язані з пропуском однієї з операцій (P або V) чи включення P в один семафор, а V в іншій - це призводить до порушення принципу взаємного виключення. Ведення семафорів погіршує прозорість тексту програми, ускладнює налагодження і можливу подальшу модифікацію.

Певним поліпшенням семафорів можна вважати запропоновані В. Хансеном засоби виділення критичних інтервалів у програмі. Програмісту необхідно лише зазначити ці зони, а компілятор автоматично вводить необхідні семафори і необхідні операції, регулюючи взаємне виключення. Знімаючи з програміста певне навантаження, ця технологія все ж таки залишає за ним процедуру виділення критичних інтервалів по тексту. Таким чином, при роботі з семафорами висока ймовірність помилки, яку важко знайти і виправити.

Для підвищення надійності механізмів взаємного виключення були запропоновані два типи конструкцій: умовні критичні ділянки і монітори.

При використанні підходу умовних критичних ділянок загальні ресурси (дані) представлені змінними, які мають спеціальний оператор опису SHARED. У контексті мови Pascal даний оператор має вигляд:

VARA: SHAREDT

де A - ідентифікатор змінної;

T – тип змінної.

Процес може використовувати загальні змінні тільки всередині оператора, так званим умовним критичним інтервалом, який має такий вигляд:

REGION A WHEN B DO S
де B - умовний вираз;
S - оператор (-ри) критичного ділянки.

Таким чином процес може увійти в критичну ділянку лише, якщо B - істинно, інакше цей процес відкладається і затримується до тих пір, поки інший процес не завершить своїх обчислень у критичній ділянці. Затримані процеси утворюють чергу, повторні входження в умовні критичні ділянки реалізуються оператором REGION. Умовні критичні ділянки «розкидані» по всій програмі. Тому простежити за ходом використання поділюваних ресурсів і переконатися у правильності їх використання досить складно. Так само важко правильно визначити значення B.

Ця складність долається мовною конструкцією централізуючою управління асинхронними процесами, яка отримала назву монітора. Ідея монітора полягає у створенні механізму, який би відповідним чином уніфікував взаємодію паралельних процесів по синхронізації, спільним даним і програмам, які обробляють ці дані. Монітор захищає дані і доступ до них може бути здійснений тільки за допомогою програми, включеної у тіло монітора. У моніторі є два типи процедур: вхідні та вихідні. Звернення до монітора відбувається через вхідні процедури. Звернення до внутрішніх процедур можливо тільки з тіла монітора. Монітор гарантує, що в кожен момент часу процедури монітора може використовувати тільки один процес. Цей процес називається процесом в моніторі. Решта процесів, які звертаються в цей момент до монітора, затримуються і ставляться в чергу. Монітор «відторгає» від процесів усі їх критичні інтервали, пов'язані з даними ресурсами, перетворюючи їх у свої процедури.

Звернення до монітора (виклик монітора) проводиться за допомогою вказівки імені процедури та імені монітора. Таке звернення доступно всім процесам в тому сенсі, що будь-який з процесів може спробувати викликати будь-яку програму, але тільки один з них може увійти в процедуру, інші ж повинні чекати, коли буде завершений попередній виклик. Процедури монітора можуть містити тільки змінні, локалізовані у тілі монітора. Це обмеження застерігає від помилок, які виникають при використанні семафорів.

Синтаксично опис монітора починається зі слова MONITOR. Потім описуються процедури і дані. Функціонально монітор об'єднує деякі дані, у тому числі типу «подія», що використовуються паралельними процесами, і всі оператори обробки цих даних, в єдиний блок.

Змінна типу «подія» використовується для управління доступом до ресурсів, що розділяються. У моніторах вона називається «умовна змінна». До змінної типу «подія» створюється черга процесів, що очікують виконання певної події. Над цією змінною дозволені операції двох типів: WAIT і SIGNAL. Оператор WAIT затримує виконання процесу, що викликав монітор, і відкриває доступ процесу до монітора. Виконання затриманого процесу може бути ініційоване оператором SIGNAL іншого процесу. Оператор SIGNAL виконується наступним чином. Якщо черга до змінної «подія» не порожня, то з черги вибирається один з процесів і ініціюється його виконання, а якщо черга порожня, то не виробляється ніякої дії.

Приклад монітора:

```
RESOURSE: MONITOR
        BEGIN LOGICAL: BUSY;
        CONDITIONAL X;-змінна, типу «подія»
        PROC ACQUIRE;-захват ресурсу
        BEGIN IF BUSY THEN WAIT X;
        BUSY: = TRUE;
        END;
        PROC RELEASE;-звільнення ресурсу
        BRGIN BUSY: = FALSE;
        SIGNAL X;
        END;
        BUSY: = FALSE;
        END;
```

У моніторі може бути введено логічна умова у вигляді умовного оператора WAIT (L), де L - булевський вираз. Вираз WAIT (L) затримує ініціалізацію процесу доти, поки значенне L стане істинним.

Третій підхід синхронізації доступу до ресурсів, може бути реалізований у керуючому виразі типу PATH керуючим виразом END.

Так, запис PATH P1 END означає, що якщо кілька процесів стоять у черзі до процедури P1, то один з них отримає до неї доступ. Після завершення виконання P1 за першим зверненням до неї P1 отримує доступ другий процес і т.д. Можливі комбінації керуючих виразів цього типу з іншими. Запис {P1} означає що P1 може бути виконана кількома процесами одночасно. Якщо один процес розпочав виконання P1, то інший може отримати до неї доступ без затримки. Цей підхід досить спеціалізований і використовується тількиу мовах, де реалізовані конструкції PATH.

2.4.4 Мовні засоби синхронізації повідомлень для МКМД-систем з роздільною пам'яттю

У системах класу МКМД з роздільною пам'яттю (МРР) для організації обміну повідомленнями потрібні нові мовні засоби і серйозна системна підтримка.

Особливість обміну інформацією у даному випадку полягає в тому, що тут діють дві автономні сторони: відправник (відправники) і адресат (адресати). Причому моменти часу, коли процеси готові до відправлення і прийому можуть не збігатися. Тому виникає необхідність синхронізації.

Існує три основних способи синхронізації пересилання повідомлень:

1) Слабко зв'язані системи (loosly coupled).

Обмін повідомленнями проводиться через «поштову скриньку» у вигляді буферного накопичувача.

2) Сильно-зв'язані системи (tightly coupled).

Обмін повідомленнями проводиться в окремих точках синхронізації («рандеву»).

3) Повністюпов'язанісистеми (completely couple).

Існує єдине централізоване синхронізуюче джерело (send, receive). Централізована синхронізація необхідна, коли відбувається спілкування трьох і більше процесів.

У слабкозв’язаних системах в обміні беруть участь два процеси і використовується асинхронна модель пересилання даних. Сенс такої моделі полягає в наступному. Коли готові дані в процесі-джерелі даних, вони передаються відразу ж або в процес-приймач, якщо він готовий до прийому даних, або в проміжну буферну пам'ять («поштова скринька»), після чого процес-джерело продовжує свою роботу незалежно від тогочи передані дані процесу-приймачу. Процес-приймач приймає дані у момент готовності іне інформує процес-джерелопро отримання даних. Прикладом використання є мова ВВС.

Перевага: висока ефективність використання процесорних ресурсів.

Недоліки:

1. Наявність буферної пам'яті для тимчасового зберігання повідомлень, обсяг якої може досягати великих розмірів із зростанням числа процесорів КС.

2. Низька надійність доставки повідомлень, тому що процес-приймач не інформує процес-джерело про свій стан. Можлива ситуація, коли процес-джерело закінчує своє існування до прийому його повідомлення процесом-приймачем, і, взагалі, може ніколи не дізнатися про те, чи прийнято його повідомлення приймачем.

У сильно зв'язаних системах так само, як і в слабкозв’язаних, використовується два процеси, які беруть участь в обміні, але буферна

пам'ять відсутня. Для синхронізації використовується механізм «рандеву». Даний підхід забезпечує синхронну модель обміну. Приклади використання - мови ADA, OCCAM. Сенс механізму «рандеву» полягає в наступному. При взаємодії двох процесів, процес-джерело зупиняється до тих пір, поки процес-приймач не буде готовий до прийому. Коли процес-приймач готовий до прийому, він посилає процесу-джерелу сигнал готовності, після чого відбувається пересилання даних. Далі процес-приймач інформує джерело про прийом його повідомлення (службове слово - reply message), після чого процес-джерело і процес-приймач продовжують свою роботу. Здійснюється постійна перевірка надійності доставки. Очікування ведуть до зниження ефективності роботи процесорів. Однак для мультипрограмного режиму це лише відносне зниження продуктивності.

Переваги:

1) Висока надійність доставки повідомлень.

2) Відсутність необхідності додаткових ресурсів (буферної пам'яті).

Недолік: можлива низька ефективність використання окремих процесорних ресурсів.

Для подолання зазначених недоліків, має сенс об'єднати перший і другий підходи. В якості буферної пам'яті можна організувати додатковий проміжний процес, який буде приймати і передавати дані, і він завжди буде готовий до таких дій (рис.2.12.). Таким чином, програміст вводить буферну пам'ять, але використовує механізм «рандеву».

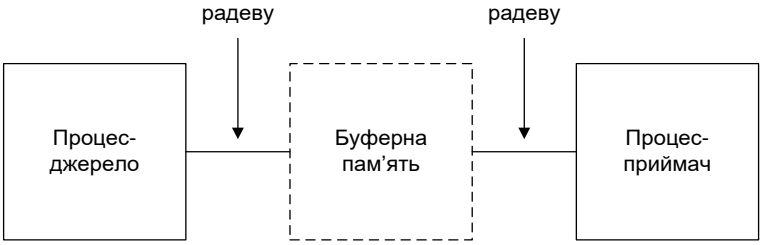


Рис.2.12. Комбінований спосіб передачі повідомлень

У повністю пов'язаних системах одночасно обмінюються даними безліч процесів. Один процес повинен опитати всі процеси, чи готові вони до обміну (вони повинні бути готові одночасно). Механізм заснований на механізмі «рандеву». Якщо процеси перебувають на різних процесорах, то потрібен механізм маршрутизації.

Сильно зв'язані системи є найбільш застосованими у даний час. Розглянемо спеціальні мовні конструкції, необхідні для відправлення і

прийому інформації в таких системах. Існує два основних оператора пересилання і прийому, які містять відповідні службові слова мови, адреси відправника та адресата, і список змінних, значення яких пересилаються з одного процесу в інший:

SEND <набір значень> TO <адресу процесу-приймача> (цей оператор знаходиться в процесі-джерелі)
RECEIVE <набір значень> FROM <адресу процесу-джерела> (цей оператор знаходиться в процесі-приймачі)

Розглянемо оператори пересилання повідомлень, які використовуютьсяу мові OCCAM. У даному випадку, замість ключових слів SEND і RECEIVE використовуються символи "!" та "?", а для визначення адрес процесу-приймача і процесу-джерела використовується ім'я логічного каналу. Для організації кожного обміну у мові OCCAM використовується логічний канал з унікальним ім'ям, яке застосовується тільки одним процесом-відправником і одним процесом-приймачем.

Оператор SEND має наступний вигляд:
<ім'я логічного каналу>! <набір значень>
Оператор RECEIVE має такий вигляд:
<ім'я логічного каналу>? <змінна>

Приклад: Нехай нам необхідно передати змінній a, яка знаходиться в одному процесі значення b +5, що знаходиться в іншому процесі. Тоді оператори пересилання даних мають наступний вигляд:

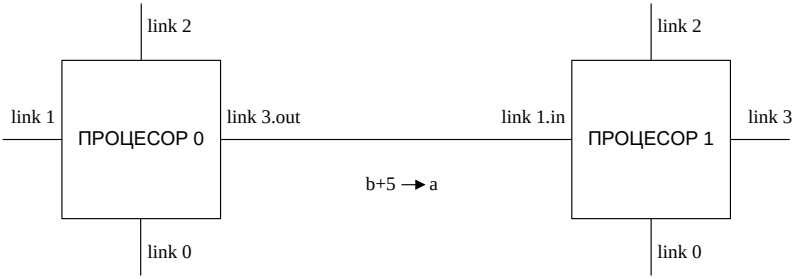
```
chan1   !       b+5.
chan1   ?       a.
```

У наведеному прикладі для обміну даними використовується ім'я логічного каналу chan1.

Щоб пересилання даних було реалізоване фізично, слід, перш за все, визначити на яких процесорах виконуються процеси 1 і 2. Можливі два варіанти: обидва процеси виконуються на одному й тому ж процесорі або процеси виконуються на різних процесорах. При реалізації першого варіанту обмін відбувається через оперативну пам'ять процесора і для цього достатньо перерахованих вище двох операторів. При реалізації ж другого варіанту обмін здійснюється по фізичних каналах, що з'єднують процесори. Тому, крім перерахованих операторів, у файл конфігурації необхідно включити оператори відповідності логічних каналів фізичним.

Приклад:

```
PLACED      F     PAR
PROCESSOR   0     T4
PLACE chan1 AT    link3.out
PROCESSOR   0     T4
PLACE chan1 AT    link1.in
```



2.5 Висновок

Усі розглянуті нами мовні конструкції зводяться до опису паралельних процесів і матричних обчислень. Паралельні процеси є реалізацією паралелізму незалежних гілок, а матричні обчислення є одним з окремих випадків реалізації природного паралелізму. Іншим прикладом природного паралелізму в програмах є паралельне виконання циклічних ділянок, однак, на рівні створення паралельних програм найчастіше спеціальних засобів паралельного виконання циклічних ділянок немає.

Прикладом використання паралельно заданих циклів є VS FORTRAN (розширення FORTRAN77) для IBM3096 (двухпроцесорна КС з векторною обробкою, загальною ОП, кожен має СОЗУ). Паралельний цикл задається наступним чином:

```
@DO
.
.
.
@ENDDO;
```

Вимогою є незалежність усіх ітерацій та наступні умови:

- Індекси циклів повинні бути цілочисельними;
- Не повинно бути переходів з тіла циклу і всередину тіла;

- Початкове значення індексу повинно бути менше кінцевого значення індексу.

Як правило, не використовується ніяких мовних засобів опису паралелізму суміжних операцій. Таким чином, розпаралелювання циклів і програм для КС, що реалізують паралелізм суміжних операцій, виконується на етапі компіляції.

3. Особливості компіляції програм для КС

3.1 Традиційні етапи компіляції

Компіляція - це перетворення програми на вхідній мові в еквівалентну послідовність машинних команд.

Розглянемо основні етапи процесу компіляції. Процес компіляції ділять на чотири логічно окремих етапи:

1. Лексичний аналізатор.
2. Синтаксичний аналізатор.
3. Семантичний аналізатор
4. Генерація коду.

Логічно компілятор можна поділити на дві частини - синтез і аналіз. У першій частині розпізнаються конструкції абстрактної програми і перевіряють їх правильність, а в другій частині проводиться побудова еквівалентної вихідної програми у машинних кодах. Реалізація цих етапів у вигляді послідовності кроків називається переглядами.

Лексичний аналіз генерує побудову по вхідному тексту програми лексичної згортки. Лексичний аналізатор на вході має довгий ланцюжок символів (вхідна програма) і повинен перетворити її у послідовність внутрішніх кодувань і набір таблиць лексем, що утворюють лексичну згортку. Лексеми - це ланцюжки літер, що утворюють конструкції мови (ідентифікатори, знаки операцій, службові слова, роздільники і т. п.). Таким чином, лексична згортка складається з набору лексем у єдиному форматі, які представляються за допомогою дескриптора, що містить два типи інформації: тип лексеми (ідентифікатор, службове слово) та місцезнаходження даної лексеми (вхід в таблицю ідентифікаторів, номер у таблиці службових слів і т. п .).

Наступним етапом є синтаксичний аналіз, який здійснює виявлення в лексичній згортці понять (ланцюжки лексем - арифметичні вираження, оператори присвоювання, умовний оператор та ін.) та їх структуру, а також перевірку, чи задовольняє ця структура синтаксису мови.

Третім етапом є семантичний аналіз, де перевіряється смислова правильність програми. Іншими словами, семантичний аналіз - це

знаходження значень атрибутів (набір характеристик) для понять програми (лексем і типів лексем), у результаті чого будується атрибутне дерево програми.

Останнім етапом є отримання програми у машинних кодах по вхідний програмі (яка представлена як атрибутне дерево).

3.2 Особливості компіляції при використанні різних способів програмування для паралельних КС

Як зазначалося раніше, існують два шляхи програмування для паралельних КС:

- Розробка нових паралельних програм.
- Адаптація програм, написаних для однопроцесорних (одноядерних) комп'ютерів.

При використанні першого шляху перед компілятором стоїть завдання адекватного перенесення паралелізму з мови високого рівня на машинну мову. Саме адекватного, так як «підгонка» паралелізму до архітектури КС відбувається на етапі програмування. Стадії лексичного і синтаксичного аналізів при компіляції, як для матричних та векторних, так і для МКМД систем можуть бути організовані традиційним способом. Необхідний "додаток" для обробки паралельних операторів складається з процедур розпізнавання деяких додаткових конструкцій, що веде до появи нових типів семантичних підпрограм. Як було зазначено вище, практично у всіх паралельних мовах програмування або відсутні конструкції для розпаралелювання циклічних ділянок программ, або існуючі конструкції можуть бути використані для розпаралелювання найпростіших циклів. Тому додатковою функцією компіляторів практично всіх мов паралельного програмування є векторизація (для матричних і векторних КС) або розпаралелювання (для МКМД систем) циклів.

Другий шлях передбачає розпаралелювання не у процесі програмування, а у процесі компіляції. Таким чином, у цьому випадку компілятори для будь-яких типів КС, крім традиційних етапів, повинні виконувати також функцію автоматичного розпаралелювання програм. Другий шлях програмування теоретично може бути використаний для будь-яких типів структур КС. Практично цей спосіб програмування використовується для Dataflow, VLIW і конвеєрних КС. Для перерахованих систем компілятор повинен забезпечувати автоматичне розбиття програми на незалежні операції (паралелізм суміжних операцій).

3.3 Способи розпаралелювання процесу компіляції

Для прискорення самого процесу компіляції, для паралельних КС при будь-якому способі програмування можуть застосовуватися алгоритми розпаралелювання процесу компіляції.

Існує три способи розпаралелювання процесу компіляції:

1. Конвеєризація процесу компіляції з використанням традиційних послідовних алгоритмів. У компіляторі виділяються незалежні процеси, пов'язані чергами часткових результатів, переданих від одного процесу до іншого (лексичний, синтаксичний, семантичний аналізатори та генерація коду). Кожен з цих процесів може бути виконаний на окремому процесорі (шарі конвеєра). Даний підхід доцільно використовувати у багатопроцесорних системах з розподіленою (загальною) пам'яттю, тоді як у МКМД системах з роздільною пам'яттю виникають великі втрати на пересилання часткових результатів.

2. Паралельне виконання окремих етапів компіляції(розробка нових паралельних алгоритмів компіляції). Для систем з векторними процесорами ці алгоритми є досить перспективними. Існують алгоритми лексичного аналізу, що використовують векторні операції. Крім цього, векторні операції можуть використовуватися на етапі синтаксичного аналізу. У цьому випадку здійснюється паралельне розпізнавання всіх заданих синтаксичних конструкцій, наприклад на першому етапі - усі роздільники, на другому - усі ідентифікатори і т. п. Семантичний аналіз і генерацію коду розпаралелити складно. Для реалізації даного підходу може знадобитися безліч копій програм.

3. Паралельна компіляція частин програми. Такий підхід зводиться до одночасної компіляції різних операторів або груп операторів з подальшою «зшивкою» відкомпільованих частин у загальну програму. Даний спосіб має сенс особливо для програм із модульною структурою. Основна складність його застосування - забезпечення достатньо ефективної синхронізації при об'єднанні паралельно зкомпільованих модулів.

3.4 Автоматичне розпаралелювання програм

Спочатку для виявлення різних типів паралелізму розглянемо типові елементи розпаралелювання у програмах. Перелічимо основні елементи і відповідно рівні розпаралелювання:

- розпаралелювання арифметичних виразів (на рівні операцій);
- розпаралелювання лінійних ділянок програм (на рівні операторів);

- розпаралелювання розгалужених ділянок програм (на рівні операторів);
- розпаралелювання та векторизація циклів (відповідно на рівні ітерацій і векторних операцій);
- розпаралелювання універсальних програм (на рівні гілок).

Розпаралелювання на кожному з перерахованих рівнів має свої особливості і складності. Найбільш формалізованими елементами розпаралелювання є арифметичні вирази і лінійні ділянки, тому що в даному випадку мається повна визначеність, як з точки зору розмірів (число операторів і відповідних їм операцій) елементів розпаралелювання, так і з точки зору рівнів розпаралелювання. Складніше вирішити питання розпаралелювання для розгалужених ділянок програм і циклів, оскільки з'являється деяка неоднозначність. Так, число операторів і операцій (розмір), що реально використовуються у розгалужених ділянках залежить від вхідних даних програми і тому неоднозначно, число ітерацій циклів також не завжди відомо заздалегідь (цикли з while). І, нарешті, найменше піддається формалізації розпаралелювання програм по гілках, оскільки саме поняття гілка - неоднозначне за розміром (це просто послідовність операторів). Більше того програма може бути розгалуженою і тоді є невизначеність ще й з точки зору елемента розпаралелювання. Таким чином, якість автоматичного розпаралелювання залежить як від характеристик (структури) послідовної програми (чи є в ній цикли, розгалуження), так і від типу структури КС, для якої необхідно виконати розпаралелювання (а значить від типу паралелізму: дрібнозернистий паралелізм суміжних операцій, природний або середньо чи крупно зернистий паралелізм на рівні гілок програми).

Завдання автоматичного розпаралелювання для КС, орієнтованих на паралелізм суміжних операцій, зводиться до розпаралелювання програм до рівня операцій.

Завдання автоматичного розпаралелювання для КС, орієнтованих на природний паралелізм, зводиться до розпаралелювання циклів.

І, нарешті, завдання автоматичного розпаралелювання для КС, орієнтованих на паралелізм незалежних гілок, зводиться до розпаралелювання циклів і до розбиття програми на незалежні асинхронні процеси. Остання функція має бути присутня у компіляторі тільки у випадку застосування традиційних мов програмування, у випадку ж використання паралельних мов - дана функція відсутня.

Результати сказаного відображені у наступній таблиці.

Тип КС	Функції компілятора, що забезпечують автоматичне розпаралелювання програм		
	Розпаралелювання лінійних ділянок і арифметичних виразів	Векторизація і розпаралелювання циклів	Розбиття програм на гілки (асинхронні процеси)
Dataflow, VLIW, Конвеєрні	+	–	–
Векторні та матричні	–	+	–
МКМД	–	+	+ –

3.4.1 Розпаралелювання лінійних програм

Відомо, що лінійна програма - це програма без розгалужень. Основними поняттями при розпаралелюванні лінійних програм є інформаційний граф і граф паралельної форми. Інформаційний граф містить інформацію про оператори (операції), що виконуються і про залежність між операторами за даним, але не містить інформацію про порядок виконання і залежності по пам'яті. Всю цю інформацію має граф паралельної форми.

Вхідними даними для розпаралелювання є запис програми у будь-якому вигляді, наприклад у вигляді схеми алгоритму. Перетворення схеми алгоритму в граф паралельної форми і є завданням розпаралелювання. Формально це завдання вирішити нескладно. Існує два необхідних і достатніх умов для паралельного виконання операторів:

1. Інформаційна незалежність (незалежність за даними). Два оператора вважаються незалежними за даними, якщо жоден з них не формує дані, які використовуються іншим в якості початкових (вхідних).
2. Незалежність по пам'яті. Два оператора a і b вважаються незалежними по пам'яті, якщо для них виконується умова:

$$Out(a) \cap In(b) = 0 \,\&\, In(a) \cap Out(b) = 0 \,\&\, Out(a) \cap Out(b) = 0$$

де *In* - вхідний множина даних оператора;
Out- вихідна множина даних оператора.

Розглянемо приклад. Нехай задана схема алгоритму (рис.3.1.). Для цієї схеми алгоритму інформаційний граф і граф паралельної форми мають вигляд, представлений на рис.3.2.

Оптимізація результату розпаралелювання пов'язано із зменшенням кількості дуг (зв'язків) у графах. Розглянемо способи зменшення числа зв'язків у наведених графах. Так, в інформаційному графі кількість зв'язків можна зменшити лише за рахунок застосування іншого алгоритму рішення даної задачі. У графі паралельної форми може бути зменшено число зв'язків з пам'яті за допомогою наступних двох підходів.

Перший спосіб пов'язаний з перетворенням лінійної програми за рахунок збільшення обсягу оперативної пам'яті.Для наведеного прикладу зробимо наступне перетворення: замінимо a:=φ_3(b) на d:=φ_3(b). Тоді схема алгоритму програми буде мати вигляд, представлений на рис.3.3., а відповідні їй інформаційний граф і граф паралельної форми на рис.3.5.

Другий спосіб пов'язаний з перетворенням лінійної програми до наведеної форми, в якій зменшення кількості зв'язків по пам'яті досягається без збільшення його обсягу. Для розглянутого прикладу перетворення до наведеної форми можна виконати заміною a:=φ_3(b) на b:=φ_3(b). Схема алгоритму у наведеній формі має вигляд, представлений на рис.3.4. Відповідні наведеній схемі алгоритму інформаційний граф і граф паралельної форми мають такий же вигляд, як і в результаті першого способу перетворення програми (рис.3.5.)

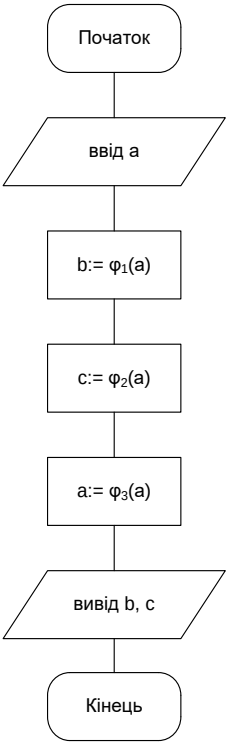
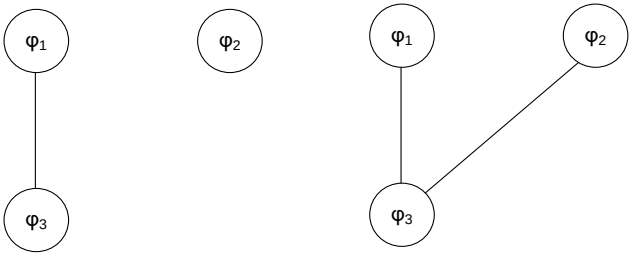


Рис.3.1. Приклад схеми алгоритму



інформаційний граф
(зв'язків може бути меньше)

граф паралельної форми
(зв'язків може бути більше)

Рис.3.2. Інформаційний граф і граф паралельної форми

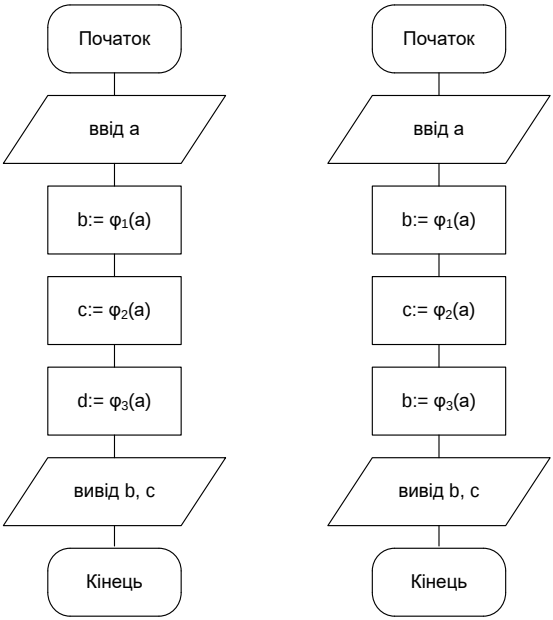


Рис.3.3. Схема алгоритму після першого способу перетворення

Рис.3.4. Схема алгоритму після другого способу перетворення

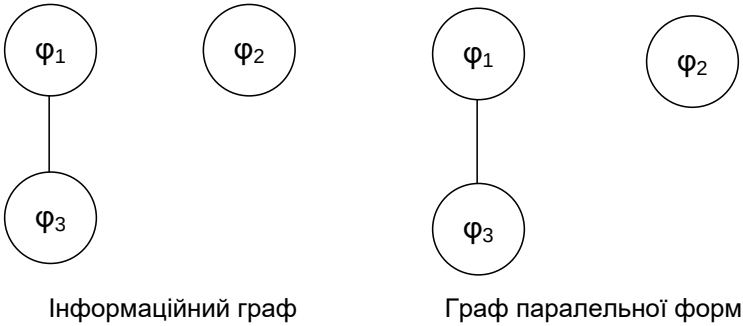
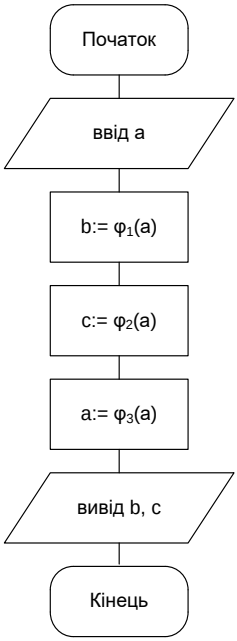


Рис.3.5. Інформаційний граф і граф паралельної форми, відповідно до перетворених схем алгоритму.

Розглянемо приклад реалізації алгоритму розпаралелювання лінійної програми, заснованого на побудові та перетворенні матриць.

Приклад: Нехай задана лінійна програма.



Out		**In**
B	=	**φ_1(a)**
C	=	**φ_2(a)**
A	=	**φ_3(b)**

Алгоритм розпаралелювання полягає в наступному:

1. *Початок.*
2. *Побудова матриці послідовності виконання - C.*
3. *Побудова матриці суміжності, в якій описані інформаційні зв'язки - A.*
4. *Побудова матриці вхідних і вихідних даних - In, Out.*
5. *Побудова матриці графа паралельної форми - B на підставі перерахованих вище матриць.*
6. *Кінець.*

C	**A**	**In**	**Out**

φ_1	φ_2	φ_3
1		
1	**1**	

	φ_1	φ_2	φ_3
φ_1			
φ_2			
φ_3	**1**		

	a	**b**
φ_1	**1**	
φ_2	**1**	
φ_3		**1**

	a	**b**	**c**
φ_1		**1**	
φ_2			**1**
φ_3	**1**		

Формуємо результуючу матрицю B на підставі вихідних матриць:

$\mathbf{B}_{ij} = \left(\mathbf{B}_{ij} \cup IO_{ij}\right) \cap C_{ij}$ где **IO$_{ij}$ –** функція зв'язку по пам'яті між і-им і j-им елементом.

IO$_{ij}$= In$_i$∩Out$_j$= 0 &Out$_i$∩In$_j$ = 0 &Out$_i$∩Out$_j$ = 0

$IO_{ij} = In_i \cap Out_j = 0 \;\&Out_i \cap In_j = 0 \;\&Out_i \cap Out_j = 0$ (у мова незалежності по пам'яті)

B

	φ_1	φ_2	φ_3
φ_1			
φ_2	**0**		
φ_3	**1**	**1**	

Дана матриця B відповідає графу паралельної форми (рис.3.2).

3.4.2 Розпаралелювання розгалужених програм

Існує два підходи розпаралелювання розгалужених ділянок програм:

1. Перетворення розгалуженої ділянки програми до лінійного вигляду з наступним розпаралелюванням її, як лінійної.

2. Розпаралелювання розгалуженої ділянки програми по операторам.

Перший підхід пов'язаний із укрупненням схеми алгоритму з метою її приведення до лінійної програми. У цьому випадку може бути використаний механізм «гамаків».

Гамаком називається такий підграф g графа G, для якого обов'язково існують дві вершини: вхідна A і вихідна B, що володіють такими властивостями:

1. Всі дуги з A ведуть у гамак.
2. Будь-який шлях уB гамак веде через A.
3. Всі дуги уB ведуть з гамака.
4. Будь-який шлях, що йде з гамака у зовні, проходить через B.

Гамак цікавий тим, що його можна стягнути в одну вершину, не порушуючи відносин суміжності між іншими вершинами графа, проте - це повна відмова від паралельної обробки.

Приклад:

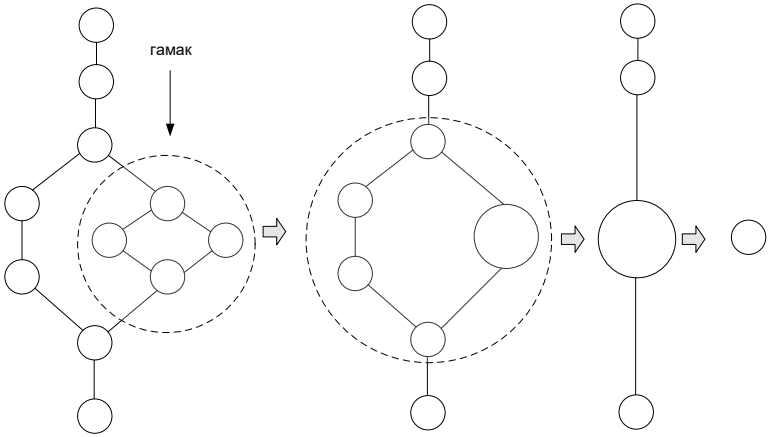


Рис.3.6. Послідовність застосування механізму гамаків для розгалуженої програми

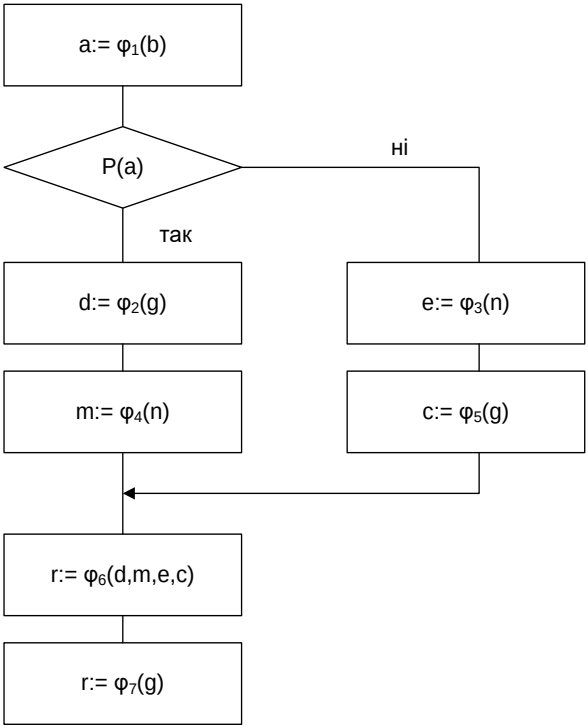
У результаті застосування механізму гамаків (рис.3.6.) отримуємо лінійну програму з операторами різної складності. Далі цю програму можна розпаралелити способом зазначеним вище. Перевага такого підходу полягає в однозначності одержуваного результату. Недоліками даного підходу є складність визначення оптимальних меж гамаків, кількості ітерацій застосування механізму гамаків, а також різна складність одержаних операторів (елементів паралельної обробки), яка може призвести до низької ефективності використання процесорних ресурсів.

Другий підхід пов'язаний з розпаралелюванням розгалуженої програми по операторам без будь-якого попереднього її перетворення. На відміну від лінійних програм, для паралельного виконання операторів розгалуженої програми, необхідно і достатньо дотримання трьох таких умов:

1. Незалежність за даними.
2. Незалежність по пам'яті.
3. Незалежність по управлінню, тобто оператори не повинні бути пов'язані між собою оператором перевірки умови.

Розглянемо алгоритм реалізації даного підходу розпаралелювання розгалуженої програми, заснованого на побудові та перетворенні матриць.

Нехай є розгалужена ділянка алгоритму.



Алгоритм розпаралелювання полягає в наступному:

1. *Початок.*
2. *Побудова матриці послідовності виконання - C.*
3. *Побудова матриці суміжності, в якій описані інформаційні зв'язки - S.*
4. *Побудова матриці вхідних і вихідних даних - I і O.*
5. *Побудова матриці логічних (по управлінню) зв'язків - L.*
6. *Побудова матриці неповного паралелізму P 'на основі матриць C, I, O, S.*

$$IO_{ij} = In_i \cap Out_j = 0 \;\&\; Out_i \cap In_j = 0 \;\&\; Out_i \cap Out_j = 0$$

Дана матриця об'єднує залежності за даними і по пам'яті.

7. *Побудова матриці графа паралельної форми - R на базі матриць L, P ', C.*
8. *Кінець.*

	C							
	φ_1	P	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7
φ_1								
p	1							
φ_2	1	1						
φ_3	1	1						
φ_4	1	1	1					
φ_5	1	1		1				
φ_6	1	1	1	1	1	1		
φ_7	1	1	1	1	1	1	1	

	S							
	φ_1	p	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7
φ_1								
p	1							
φ_2								
φ_3								
φ_4								
φ_5								
φ_6			1	1	1	1		
φ_7								

	I							
	b	a	g	N	d	e	M	c
φ_1	1							
p		1						
φ_2			1					
φ_3				1				
φ_4				1				
φ_5			1					
φ_6					1	1	1	1
φ_7			1					

	O					
	A	d	e	m	c	r
φ_1	1					
p						
φ_2		1				
φ_3			1			
φ_4				1		
φ_5					1	
φ_6						1
φ_7						1

	L							
	φ_1	p	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7
φ_1								
p								
φ_2		1						
φ_3		1						
φ_4		1						
φ_5		1						
φ_6								
φ_7								

	P'							
	φ1	p	φ2	φ3	φ4	φ5	φ6	φ7
φ1	0	0	0	0	0	0	0	0
p	1	0	0	0	0	0	0	0
φ2	0	0	0	0	0	0	0	0
φ3	0	0	0	0	0	0	0	0
φ4	0	0	0	0	0	0	0	0
φ5	0	0	0	0	0	0	0	0
φ6	0	0	1	1	1	1	0	0
φ7	0	0	0	0	0	0	1	0

	R							
	φ1	p	φ2	φ3	φ4	φ5	φ6	φ7
φ1								
p	1							
φ2		1						
φ3		1						
φ4		1						
φ5		1						
φ6			1	1	1	1		
φ7							1	

Результуючій матриці R відповідає граф, зображений на рис.11.2.2.а. Даний граф паралельної форми включає вершину P, якій відповідає операція перевірки умови. Це означає, що ми отримали деякий об'єднаний граф паралельної форми, який включає множину реальних підграфів (залежно від вхідних даних). Такий граф неоднозначний з точки зору визначення та оцінки таких його характеристик, як число вершин, ширина ярусів, критичний шлях, загальна трудомісткість виконання, зв'язність та ін.Така неоднозначність є **недоліком** даного способу розпаралелювання. З іншого боку, цей спосіб легко піддається формалізації і в цьому його **перевага**.

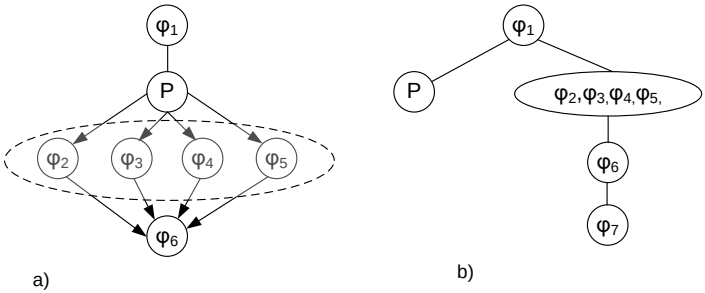


Рис.3.7. Результуючі графи паралельної форми

Певним удосконаленням виконання розгалужених ділянок програм є паралельне виконання логічно пов'язаних операторів (рис.3.7.б). Це означає, що одночасно з перевіркою умови організовується виконання всіх, залежних від нього гілок обчислень. Коли ж результат перевірки умови стає відомим, виконання непотрібних гілок призупиняється. Таким чином,

може бути досягнуто значне підвищення реальної продуктивності КС при виконанні розгалужених програм. Звичайно ж це можливо за рахунок додаткової апаратної та програмної підтримки обчислювальних засобів. Такий підхід отримав поширення у даний час. Слід, однак, зауважити, що він не вписується в класичні поняття паралелізму (див. необхідні і достатні умови паралельного виконання операторів у розгалужених програмах).

3.4.3 Розпаралелювання і векторизація циклічних ділянок програм

Основна робота, яку виконує будь-який комп'ютер - обробка циклів та інших повторюваних ділянок. Це пояснюється тим, що ациклічні ділянки обчислюються один раз, а фрагменти всередині циклів повторюються сотні і тисячі разів. Тому питома вага таких фрагментів в обчислювальних завданнях дуже висока і розпаралелювання циклічних ділянок має особливе значення. Залежно від типів КС застосовується розпаралелювання або векторизація циклів.

Розглянемо основні відмінності розпаралелювання та векторизації (рис.3.8.). Розпаралелювання застосовується для асинхронних МКМД систем і передбачає одночасне виконання процесорами різних ітерацій (проходів) циклу. Векторизація застосовується для синхронних ОКМД систем і передбачає перетворення циклу в послідовність векторних команд, причому КС, для якої проводиться векторизація, повинна мати відповідний набір векторних команд. Як розпаралелювання, так і векторизація можливі при дотриманні деяких умов і обмежень на структуру циклічних ділянок. Перш ніж їх перерахувати, розглянемо перешкоди до розпаралелювання і векторизації циклів.

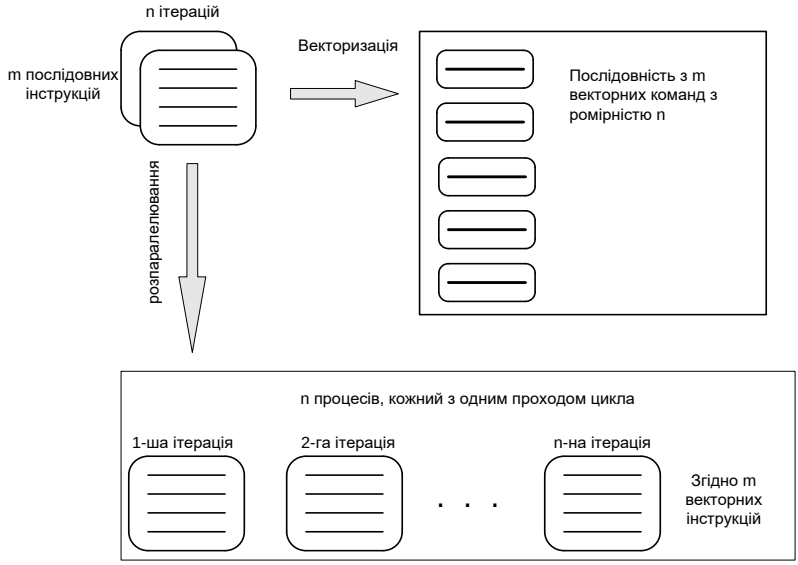


Рис.3.8. Розпаралелювання і векторизація

Перешкоди до розпаралелювання і векторизації циклів

Розглянемо різні конструкції, які ускладнюють або навіть блокують процес розпаралелювання та векторизації. Іншими словами, деякі з нижче перерахованих конструкцій "фундаментально" не розпаралелювані і (або) не підлягають векторизації, а деякі є важкими для розпаралелювання та (або) векторизації з точки зору аналізу і необхідного перетворення.

Наявність операторів умовного переходу в тілі циклу.

Процес розпаралелювання в даному випадку може проводиться безперешкодно, у той час як процес векторизації може стримуватися циклами, які містять оператори IF і передачу управління. Для того, щоб такі цикли можна було векторизувати, у компіляторах використовуються два методи:

1. Метод маскування.
2. Метод скорочених векторів.

Приклад: Розглянемо цикл з умовним оператором на мові FORTRAN

```
      DO 100 I = 1,N
      IF (A(I).NE.0) B(I) = C(I) + 10
100   CONTINUE
      END.
```

Якщо застосувати до цього циклу метод маскування, то отримаємо таку програму:

```
VECTOR CODE
LENGTH
    N      COMPARE A(1:N), 0  M(1:N)        – маска
    N      ADD C(1:N), 10     B(1:N)        WHERE  M(1:N)
           (векторнікомандидовжиною N)
```

Спочатку, за допомогою операції порівняння векторів, формується розрядний вектор-маска. У наступній команді виконується векторна операція під маскою, тобто складаються тільки ті елементи, для яких у відповідних розрядах маски стоять одиниці. В обох командах зберігається розмірність N.

У *методі з використанням скорочених векторів* головну роль грає операція стискання, яка виконується під маскою. Існує два способи стискання. Спочатку розглянемо програму, отриману в результаті застосування першого способу:

```
N     COMPAREA(1:N), 0    M(1:N)
N      COMPRESS C(1:N),M(1:N) CC(1:N*d)
N*d    ADD CC(1:N*d), 10         BB(1:N*d)
N      DECOMPRESS BB(1:N*d), M(1:N)    B(1:N)
```

де N - розмірність команд (довжина); d - питома вага «1» в масці;

$$(0 \le d \le 1)$$

Знову спочатку виконується порівняння векторів, у результаті чого формується маска. Потім слідує команда стискання, що дає тимчасовий скорочений вектор, тобто такий, в якому залишені елементи, відповідні одиницям у векторі-масці. Далі над скороченим вектором виконується основна операція, результатом якої є також тимчасовий вектор у стислому вигляді. Процес завершується перетворенням стисненого вектора-результату в нормальний вигляд (розмірності N) за допомогою операції розширення, яка виконується під керуванням тієї самої маски. Тепер розглянемо програму, отриману в результаті другого способу стискання:

```
N     COMPAREA(1:N), 0    M(1:N)
N     GENERATE 1:N I(1:N)
N     COMPRESS I(1:N),M(1:N) II(1:N*d)
```

```
N*d  GATHER C(II(1:N*d)) CC(1:N*d)
N*d  ADD CC(1:N*d), 10          BB(1:N*d)
N*d  SCATTER BB(1:N*d)   B(II(1:N*d))
```

Спочатку генерується вектор-маска. Потім формується вектор індексів, елементами якого є числа 1,2, ..., N, де N –максимальний розмір вектора. Третій крок полягає в тому, що цей вектор індексів стискається за допомогою сформованої раніше маски і перетворюється в тимчасовий скорочений вектор, елементи якого являють собою індекси елементів вхідного вектора, що задовольняють умові. Потім вхідний вектор за допомогою операції складання, виконується під управлінням скороченого індексного вектора, перетворюється у тимчасовий вектор, як це було в попередньому прикладі. На наступному кроці над стислим таким чином вектором виконується основна операція. Нарешті, стислий вектор-результат після операції розсипання, який також виконується під управлінням індексного вектора, перетворюється в нормальний вигляд.

Який з двох способів, пов'язаних з використанням скорочених векторів, ефективніше, залежить від довжин векторів, з якими працюють команди, які використаніу тому чи іншому циклі. Метод маскування дозволяє обходитися меншим числом команд, але кожен з них оперує з векторами довжиною N. В обох випадках використання методу скорочених векторів число команд більше, але основна операція проводиться над вектором довжиною N*d ($0 \le d \le 1$), де d - відносне число одиниць у векторі-масці. Накладні витрати (тимчасові), пов'язані з обробкою скорочених векторів, залежать від того, як реалізовані операції стиснення даних. У першому випадку довжина кожного вектора, що бере участь в операції стискання, дорівнює N. У другому випадку для формування вектора індексів використовуються три операції, кожна над векторами довжини N, але наступні операції збірки і розсипання виконуються над векторами, що мають довжину N * d.

Таким чином, метод скорочених векторів вигідніше тоді, коли у векторі-масці є невелике число одиниць і число операцій стискання меньше, порівняно з числом арифметичних операцій.

У КС CRAY X-MP, 4-MP, 2, 3, Fujitsu (VP-100, VP-200), NEC (SX-1, SX-2) використано метод скорочених векторів.

У КС ETA Systems (ETA-10), Control Data (Cyber-205) використаний метод маскування.

Залежності по даним у циклах.

З точки зору розпаралелювання та векторизації існує три групи залежностей у циклах:

1.Залежності, що не перешкоджають векторизації і розпаралелюванню.

2. Залежності, що перешкоджають векторизації і розпаралелюванню. Для їх усунення необхідна трансформація тіла циклу.

3. Рекурсивні залежності, які векторизацію і паралельне виконання циклу роблять неможливим.

Розглянемо приклади, що ілюструють залежності кожної з перерахованих груп.

Приклади залежностей, які не перешкоджають векторизації і розпаралелювання:

a) Залежність за даними всередині однієї і тієї ж ітерації циклу.

```
FOR i:=1 TO n DO
  A(i):=C(i);
  B(i):=A(i);
END;
```

Як видно з наведеного прикладу, така залежність не перешкоджає як векторизації, так і розпаралелюванню.

b) Залежності за даними між операторами, які виконуються на різних ітераціях циклу.

```
FOR i:=2 TO n DO
  A(i):=C(i);
  B(i):=A(i-1);
END;
```

У даному випадку векторизація можлива, а розпаралелювання ні, оскільки існує зв'язок попередньої і наступної ітерацій.

c) Залежності за даними між операторами для вкладених циклів.

```
FOR i:=1 TO n DO
  FOR j:=2 TO n DO
    A(i,j):=C(i,j);
    B(i,j):=A(i,j-1);
  END;
END;
```

У даному прикладі по зовнішньому циклу (по i) розпаралелювання і векторизація можливі, а по внутрішньому (за j) векторизація можлива, а розпаралелювання ні.

```
FOR i:=2 TO n DO
  FOR j:=2 TO n DO
     A(i,j):=C(i,j);
     B(i,j):=A(i-1,j-1);
```

```
END;
END;
```

У наведеному прикладі векторизація можлива, а розпаралелювання по внутрішньому циклу можливе, а по зовнішньомуні.

Приклади залежностей, що перешкоджають векторизації і розпаралелюванню:

а) Залежності за даними між операторами, виконуваними на різних ітераціях циклу (для одиночного циклу).

```
FOR i:=1 TO n-1 DO
  A(i):=C(i);
  B(i):=A(i+1);
END;
```

У даному прикладі є залежність між поточною та наступною ітераціями циклу, тобто між і та і +1 ітераціями циклу, яка перешкоджає як його розпаралелюванню, так і векторизації. Однак векторизація може стати можливою, якщо виконати трансформацію такого циклу шляхом переупорядкування його операторів:

```
FOR i:=1 TO n-1 DO
  B(i):=A(i+1);
  A(i):=C(i);
END;
```

Тепер векторизація можлива.

Розглянемо приклад, коли існує залежність по пам'яті (циркуляційна):

```
FOR i:=1 TO n-1 DO
  A(i):=B(i);
  B(i):=A(i+1);
END;
```

У наведеному прикладі трансформація циклу шляхом тільки переупорядкування операторів на дасть бажаного результату. У даному випадку проблема розриву залежності може бути вирішена за допомогою введення додаткового масиву.

```
FOR i:=1 TO n-1 DO
  D(i):=B(i);
  B(i):=A(i+1);
  A(i):=D(i);
END;
```

Введення додаткового масиву **D** робить можливим векторизацію даного циклу.

а) Залежності за даними між операторами, що виконуються на різних ітераціях для вкладених циклів.

```
FOR i:=1 TO n DO
  FOR j:=1 TO n DO
    A(i+1,j):= A(i,j)+10;
```

```
END;
END;
```

Розглянемо даний приклад докладніше Нехай існує масив А виду

$$A = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Тоді згідно з наведеним вище циклом нові елементи масиву А формуються в такому порядку:

$$A(2,1) \to A(1,1) + 10 = 11 \quad (1)$$
$$A(3,1) \to A(2,1) + 10 = 21 \quad (2)$$
$$A(2,2) \to A(1,2) + 10 = 12 \quad (3)$$
$$A(3,2) \to A(2,2) + 10 = 22 \quad (4)$$
$$A(2,3) \to A(1,3) + 10 = 13 \quad (5)$$
$$A(3,3) \to A(2,3) + 10 = 23 \quad (6)$$

Аналізуючи отриману послідовність операторів, можна зробити висновок, що спочатку можуть виконуватися одночасно (1), (3) і (5) оператори, а потім (2), (4) і (6) оператори. Іншими словами, повна векторизація і розпаралелювання неможливі, однак можлива часткова по j. Для цього замінимо місцями заголовки циклу FOR і отримаємо:

```
FOR j:=1 TO n DO
  FOR i:=1 TO n DO
    A(i+1,j):= A(i,j)+10;
END;
END;
```

Рекурсивні залежності в циклах.

Розглянемо приклад рекурсивної залежності в циклі:

```
FOR j:=2 TO n DO
  A(j):= A(j-1)+10;
END;
```

За своєю природою такі цикли є «чисто» послідовними, тому не векторизуються і не розпаралелюються.

Основні умови векторизації і розпаралелювання циклічних ділянок програм

Усі існуючі методи векторизації і розпаралелювання працюють при дотриманні наступних умов і обмежень:

1. У циклі не повинно бути неприпустимих залежностей за даними або по пам'яті.
2. Цикл повинен бути тісно-гніздовим, тобто всі оператори DO завершуються одним оператором і між операторами DO не повинно бути ніяких інших операторів.
3. Мінімальні і максимальні значення параметрів циклів повинні бути цілочисельними виразами, а крок (нарощення) повинен бути цілою константою.
4. У тілі циклу не повинно бути операторів введення / виведення, передачі управління з циклу, виклику підпрограм і функцій.
5. Не повинна застосовуватися нелінійна і непряма індексація масивів у тілі циклу.

Методи розпаралелювання та векторизації циклів

Методи розпаралелювання та векторизації циклів мають різну ступінь універсальності і орієнтовані на різні типи КС. Так, метод гіперплощин орієнтований на багатопроцесорні векторно-конвеєрні та МКМД КС, метод координат - на матричні і векторні КС а метод пірамід - на МКМД системи. Розглянемо кожен з цих методів.

Метод гіперплощин Загальна постановка задачі розпаралелювання циклів полягає у наступному. Нехай є тісно-гніздовий цикл виду:

```
FOR  I1 = 1 TO r1 DO
  FOR I2 = 1 TO r2 DO
    FOR In = 1 TO rn DO
      T(I1, I2,…, In) -тіло циклу представляє собою
функцію від індексів.
    END;
  END;
END;
```

де $\mathbf{r_k}$, k=1,n - кінцеві значення індексів $\mathbf{I_k}$,.

Уся множина ітерацій такого циклу називається простором ітерацій. Завдання розпаралелювання циклу ставиться як завдання розбиття простору ітерацій на такі підпростори (підмножини), де ітерації кожного з них можуть бути виконані одночасно, при цьому зберігається порядок інформаційних зв'язків вхідного циклу. Між підпросторами дії виконуються послідовно. Вирішення цього завдання розпадається на два.

Перше завдання - виявлення залежностей (інформаційних і по пам'яті) між ітераціями, друге - збірка на основі виявлених залежностей ітерацій в окремі гілки.

Пояснимо сенс методу гіперплощин на прикладі. Нехай дано гніздо циклу виду:

```
FOR  I = 1 TO l DO
 FOR  J = 2 TO m DO
  FOR K = 2 TO n DO
   X(J,K) = f((J+1,K), X(J,K+1), X(J-1,K), X(J,K-1))
  END;
END;
END.
```

Кожен елемент матриці X є функцією від 4-х сусідніх елементів. Елемент X (J, K) перевизначається l разів.

Простір ітерацій цього гнізда циклів трьохмірний і всі ітерації є цілочисельними точками паралелепіпеда [1, l] * [1, m] * [1, n] (див. рис.3.9.).

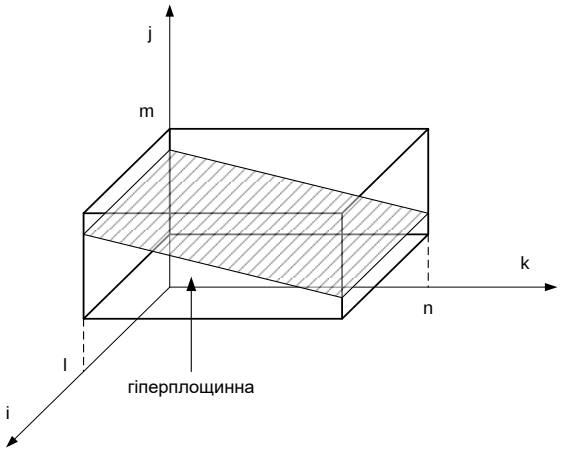


Рис.3.9. Геометричне уявлення простору ітерацій, що застосовується в методі гіперплощин

Метою даного методу є пошук у просторі ітерацій такого сімейства гіперплощин, тобто у нашому випадку звичайних двовимірних площин, щоб точки ітерацій, що лежать на ній, задовольняли б умовам незалежності. Ідеальним варіантом є знаходження площин, паралельних

одній з осей координат. Однак, аналіз залежностей точок X показує, що в таких площинах ми не можемо одночасно прораховувати множину точок X, які входять до них (між ними існує інформаційна залежність). Таким чином, необхідно знайти які-небудь похилі площини (гіперплощини), які б у результаті покрили всі точки нашого паралелепіпеда (простору ітерацій). За допомогою спеціального математичного апарату на підставі залежності між координатами виводяться рівняння площин. Циклам з різною структурою відповідають різні рівняння.

Для нашого випадку візьмемо площину, яка визначається рівнянням:

$$2I + J + K = \text{const}$$

Значення const повинно змінюватися до тих пір, поки не будуть знайдені всі точки простору ітерацій.

У зв'язку з введенням рівняння площини повинні бути введені нові індексні змінних у гніздо циклів.

$$I' = 2I + J + K$$
$$J' = I'-2J'-K'$$
$$K' = K$$

Далі перевизначємо наші координати:

```
FOR  I' = 6 TO 2l + m + n DO
FOR J', K' DO PAR (паралельнодляусіх J', K')
X(J',K') = f(X(J'+1,K'), X(J',K'+1), X(J'-1,K'),
X(J,K'-1))
  END;
END.
```

У цьому випадку на кожному кроці зовнішнього циклу I '(для заданої гіперплощини), що змінюється від 6 до 2l + m + n, паралельно виконуються операції над усіма елементами масиву, що входять в дану гіперплощину. Іншими словами зовнішній цикл "перебирає" все гіперплощини, що перетинають простір ітерацій [1, l] * [1, m] * [1, n], а всередині кожного звернення цього циклу всі ітерації, що належать площині, виконуються одночасно і незалежно. Мінімальне значення I 'визначається підстановкою мінімальних значень I, J, K у рівняння I' = 2I + J + K. Множина ітерацій, що належать кожній площині, визначається перебором координат I, J, K із заданої множини, що задовольняє значеннями констант гіперплощин. Так, наприклад, площині 2I + J + K = 6 належить єдиний набір координат

I	**J**	**K**
1	**2**	**2**

і, відповідно, єдина ітерація: при **I = 1** визначається **X (2,2).**

Наступній площині **2I + J + K = 7** належить два набору координат

I	J	K
1	2	3
1	3	2

і, відповідно, дві незалежні ітерації при I = 1: X (2,3) і X (3,2), а площини 2I + J + K = 10 вже належать вісім різних наборів координат

I	J	K
1	2	6
1	6	2
1	3	5
1	5	3
1	4	4
2	2	4
2	4	2
2	3	3

що відповідає одночасному виконанню восьми ітерацій: при I = 1: X (2.6); X (6.2); X (3.5); X (5.3); X (4.4) і при I = 2: X (2.4); X (4.2); X (3.3).

Метод координат. Даний метод є модифікацією методу гіперплощин і дає ефективні результати для векторних і матричних систем. Сутність його полягає у тому, щоб використовуючи перестановки операторів тіла циклу і перейменування деяких змінних, домогтися можливості розташування гіперплощини паралельно одній з координат.

У цьому методі всі оператори тіла циклу виконуються строго послідовно, але кожен з них виконується не над одним елементом масиву, а над усім масивом або його частиною (тобто перетвориться у векторну команду (оператор)). Проілюструємо це на прикладі. . Нехай заданий послідовний цикл

```
FOR i = 2 TO m DO
  FOR j = 1 TO n DO
    x(i,j) = y(i,j) + z(i);
    z(i) = y(i-1,j);
    y(i,j) = x(i+1,j);
  END;
  END.
```

Для того, щоб застосувати метод координат, необхідні наступні перетворення гнізда циклу: поміняти між собою місцями два оператора заголовка циклу (1 і 2 оператори), ввести додатковий вектор **u (i)** для зберігання **x (i +1, j)**, поміняти місцями оператори **z (i) і y (i, j)**. У результаті отримаємо такий цикл

```
FOR j = 1 TO n DO
  FOR i ∈ 2 TO m синхронно DO
    u(i)= x(i+1,j)
    x(i,j) = y(i,j) + z(i);
    y(i,j) = u(i);
    z(i) = y(i-1,j);
  END;
END.
```

Цей цикл векторизується по і, тобто кожен з операторів є векторною операцією для і ∈ 2, m. Даний цикл виконується n раз (j = 1, ..., n).

Метод пірамід. Метою даного методу розпаралелювання циклів є повне виключення пересилань у КС. Тому метод пірамід орієнтований на МКМД системи з роздільною пам'яттю, в яких синхронізація та обмін даними між обчислювальними гілками призводить до великих часових витрат. Прикладом його застосування є компілятор для КС «Ельбрус».

У відповідності з цим методом розпаралелювання циклу зводиться до його перетворення в множину автономних гілок. У даному випадку гілка – це послідовність ітерацій. Уся необхідна інформація для кожної гілки обчислюється всередині неї.

Найпростіший метод формування автономних гілок полягає в наступному. У циклі вибираються усі результуючі ітерації, тобто ітерації, що виробляють деякі дані, які не використовуються ніякими іншими далі в тілі циклу. Кожна ітерація служить основою для окремої паралельної гілки і є останньою в цій гілці. Потім до кожної з них приєднуються всі ітерації, які безпосередньо формують для неї вхідні дані і т.д. до тих пір, поки кожна гілка не міститиме всю «історію» обчислень результуючої ітерації (перегляд знизу вгору). Таким чином, ми можемо побудувати повністю автономні гілки (без пересилань), проте деякі ітерації при такому підході будуть дублюватися у різних гілках (рис.3.10.). Тому даний метод є ефективним тоді, коли втрати часу на синхронізацію і обмін даними перевищують час на дублювання обчислень. Крім того, сформовані гілки ітерацій можуть бути різної довжини, а значить трудомісткості, що може негативно позначитися на ефективності використання процесорних ресурсів. Тому для підвищення ефективності використання процесорних ресурсів після породження множини гілок, деякі з них (короткі) можуть

бути склеєні (штучно за допомогою компілятора) для того, щоб довжина усіх гілок могла бути однаковою.

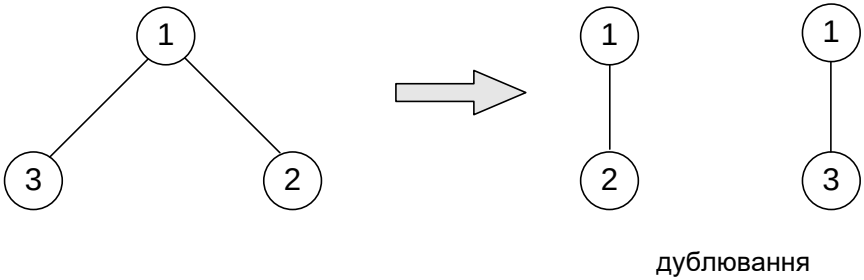


Рис. 3.10. Перетворення циклу внаслідок застосування методу пірамід

3.4.4 Розбиття програми на асинхронні паралельні процеси

Перш за все слід зазначити, що, якщо методи розпаралелювання та векторизації циклів широко використовуються при побудові компіляторів, то методи автоматичного розпаралелювання на асинхронно виконувані процеси (рівень незалежних гілок) представляють в основному теоретичний інтерес.

У загальному випадку програма може бути представлена як послідовність циклів, або розгалужена програма, що включає обчислювальні оператори і циклічні ділянки.

У першому випадку суть розбиття програми на асинхронні паралельно виконувані процеси полягає в наступному. Кожен цикл (із заданої множини m) розпаралелюється за допомогою методу пірамід, у результаті чого формуються асинхронні гілки (рис.3.11.).

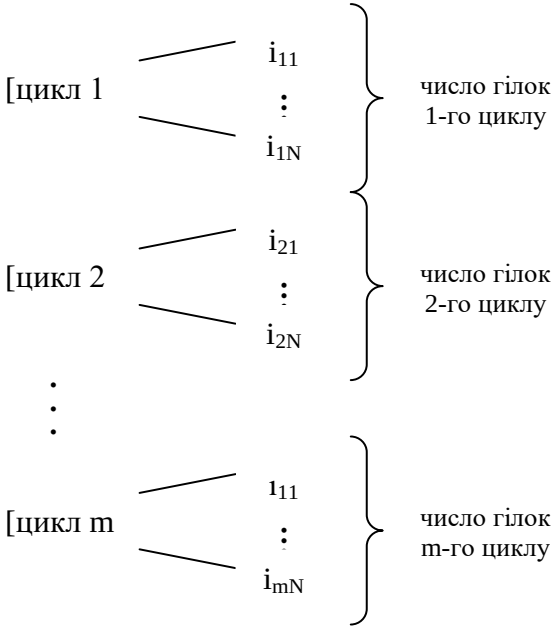


Рис.3.11. Послідовність циклів після застосування методу пірамід

Далі формується граф паралельної форми, з числом ярусів, рівним кількості послідовних циклів. На кожному і-му ярусі міститься число вершин, рівне кількості гілок і-ого циклу (рис.3.12.).

У другому випадку для розбиття програми на асинхронні паралельні процеси може бути застосований метод гамаків, описаний раніше. Відмінною особливістю даного варіанту є те, що елементом програми можуть бути не тільки окремі оператори, а й циклічні ділянки, які включаються до гілки з допомогою методу гамаків (рис.3.13.). Зрозуміло, що метод гамаків, як і метод пірамід, не дає гарантії отримання оптимального розбиття на гілки (критерій оптимізації - мінімальний час виконання програми), тому що існує безліч варіантів об'єднання в гамаки. Для того щоб наблизити отриманий результат розбиття до оптимального, доцільно перетворити отриманий граф програми шляхом об'єднання деяких гілок (процесів) з метою вирівнювання їх трудомісткості, а також наближення кількості гілок до значення, яке кратне числу процесорних елементів. При цьому всі процеси, що виконуються до кожного з поєднуваних процесів, повинні бути виконані до моменту початку виконання об'єднаних гілок. Аналогічно усі процеси, які виконувалися

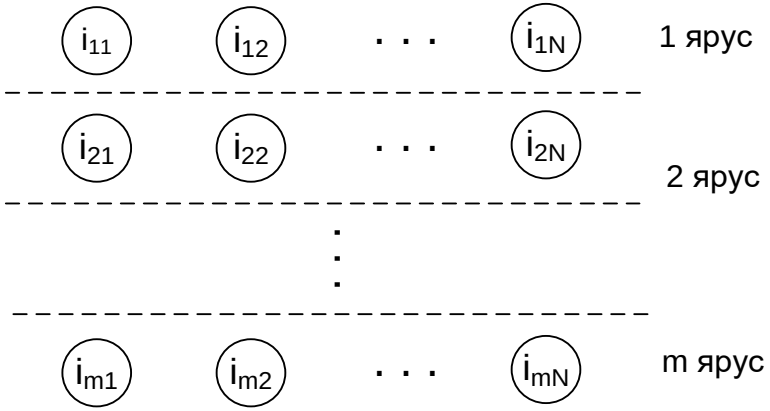


Рис.3.12. Граф паралельної форми, відповідний програмі, яка складається з послідовності циклів на рис.3.11.

після кожного з поєднуваних процесів, можуть починати своє виконання лише після реалізації об'єднаного процесу. Об'єднання двох процесів в один призводить до скорочення сумарного часу роботи на час *t*реалізації команд створення й об'єднання процесів. Необхідно генерувати такі процеси програми, щоб для їх реалізації на кінцевому числі процесорів *Q* витрачався мінімальний час.

Іншими словами, необхідно знайти *Q* об'єднань гілок приблизно рівної тривалості, в які входять усі вхідні гілки.

Для визначення оптимальної структури програми у загальному випадку запропонований евристичний алгоритм, що дозволяє за допомогою ітеративного методу синтезувати структуру програми з досить гарним наближенням до мінімального часу виконання.

У поставленій задачі з'являється можливість зміни структури графа, в залежності від якої, змінюється і час виконання графа. Сутьність запропонованого алгоритму полягає у перевірці доцільності об'єднання сусідніх процесів і в ітеративному процесі їх об'єднання. Резерв часу для кожного процесу визначається як різниця між пізнім і раннім термінами початку виконання процесів без зміни загального часу виконання графа. Об'єднання можливо створювати для гілок, які є вихідними або вхідними для однієї вершини.

Об'єднання проводиться, якщо резерв часу однієї гілки більше часу виконання іншої гілки, з якою вона об'єднується. Процес триває ітеративно до тих пір, поки зменшується час виконання програми.

На рис. 3.14. і рис.3.15. представлені фрагменти графів програм і умови об'єднання гілок, для яких може розглядатися описаний вище алгоритм.

Розглянемо фрагмент графа на рис.3.14. Нехай для 2-й вершини ранній термін *(tpc2),* пізній термін *(tnc2)* і її вага *(W2)* відповідно рівні: *tpc2 '= i; tnc2' = j; W2 = Z2*, а для 3-й вершини: *tpc3 '= k; tnc3 '= l; W3 = Z3* Тоді для об'єднаної вершини ранній, пізній терміни її виконання, а також вага відповідно визначаються, як: *tpc2 +3 = max {i, k}; tnc2 +3 = min {j, l} ; W2 +3 = Z2 + Z3.* Об'єднання має сенс, якщо резерв часу об'єднаної вершини не перевищує її сумарну вагу, тобто *tnc2 +3 - tpc2 +3 ≤ Z2 + Z3.* Аналогічним чином визначаються ранні, пізні терміни виконання вершин, а також умови їх об'єднання для фрагмента графа, зображеного на рис.3.15. Розглянемо приклад об'єднання гілок фрагмента програми на рис. 3.16. У даному випадку об'єднання можливе, оскільки*tnc2 +3 - tpc2 +3 = 8-3 = 5, W2 +3 = 5*, тобто резерв в 5 тактів достатній для виконання додаткових тактів об'єднаної вершини.

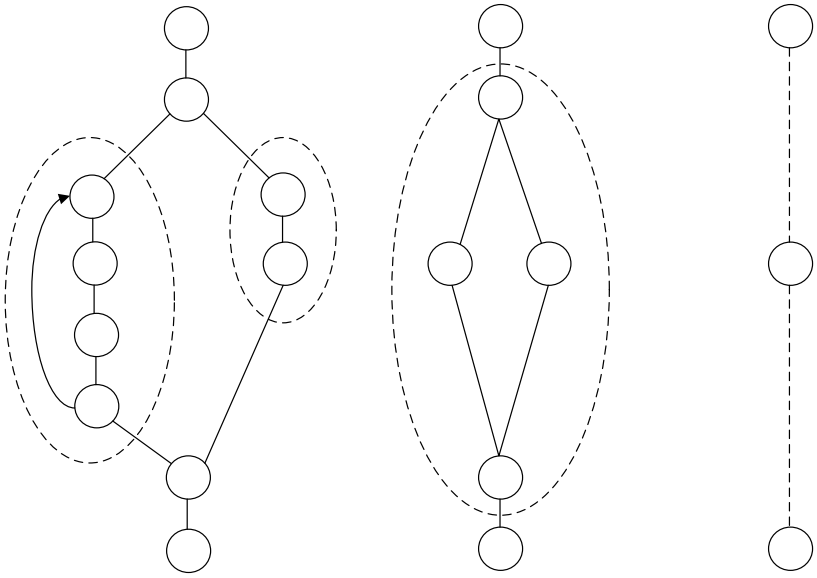


Рис.3.13. Перетворення розгалуженої програми з циклами до лінійної форми за допомогою методу гамаків

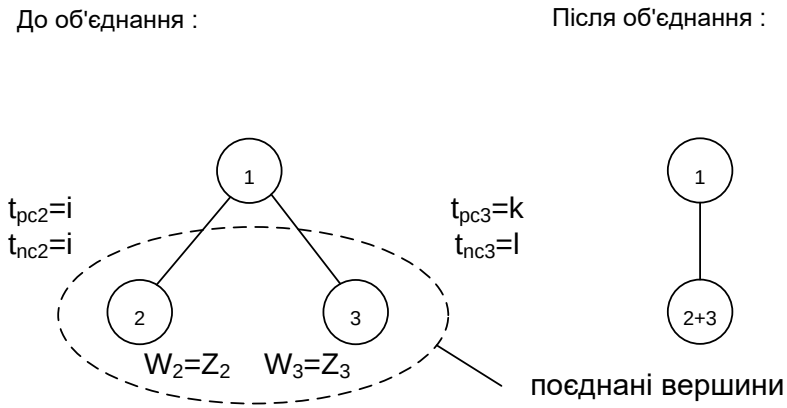


Рис.3.14. Фрагмент графа програми до і після об'єднання

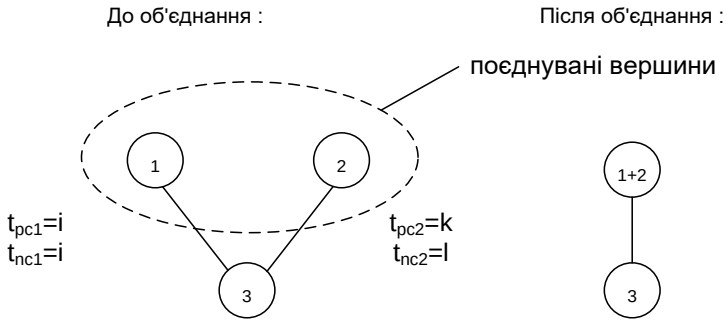


Рис.3.15. Фрагмент графа програми до і після об'єднання

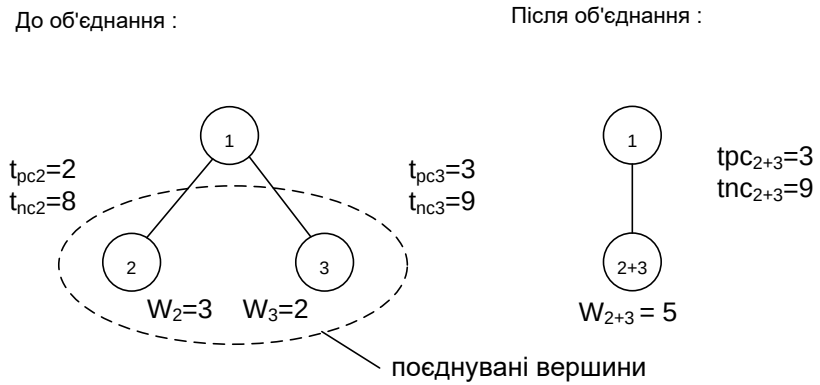


Рис.3.16. Приклад фрагмента графа програми з можливістю об'єднання гілок

3.4.5 Розпаралелювання програм для КС, що керуються потоком даних (Dataflow)

Граф потоку даних

Як зазначалося вище, в КС, керованих потоком даних (Dataflow системах) порядок виконання операторів у програмі визначається динамічно (виконуються ті оператори, для яких підготовлені дані). У кожен момент часу здійснюється перевірка умов готовності операторів до роботи і виконання тих з них, які задовольняють умовам готовності. Такий спосіб управління дозволяє зберегти при складанні програми і легко використовувати при виконанні природний паралелізм, властивий будь-якій задачі. При цьому повністю виключається проблема синхронізації, оскільки виконуються завжди тільки ті оператори, для яких вже закінчився процес обчислення необхідних вхідних даних. Такий підхід сильно відрізняється від традиційного, коли звичайні програми вважаються за замовчуванням послідовними і які використовують «природний» порядок виконання команд. Для Dataflow систем використовується протилежний підхід - будь-яка програма вважається за замовчуванням паралельною, незважаючи на те, що для складання таких програм застосовуються традиційні (послідовні) мови програмування. Розпаралелювання ж проводиться динамічно, спочатку на стадії компіляції, а потім у процесі виконання програм.

У Dataflow системах в якості моделі обчислень використовується граф потоку даних. Вершинам такого графа відповідають команди (і відповідні

їм операції), а ребрам - дані для них. У такій моделі зв'язки між командами програми здійснюються тільки через дані. Операції, зазначені у вершинах, виконуються тільки над даними на входах вершин і не залежать від сусідніх вершин. Ребра графа мають позначення, що відповідають значенням даних. Ці значення даних являють собою проміжний результат обчислень, який ще не був використаний. Поява необхідних для виконання даної команди операндів призводить до виконання цієї команди. Для графа потоку даних немає поняття арифметичного виразу, програми лінійної або з розгалуженням, а є потік команд унарних або бінарних. Граф будується зліва направо. Верхній вхід - лівий операнд, нижній вхід - правий операнд. Перед початком виконання програми визначені тільки вхідні дані та константи. Команда стає активною, коли усі ребра, що входять у відповідну даній команді вершину графа, мають значення. Команда стає активізованою, коли хоча б одне з ребер, що входять у вершину, має значення. Команда неактивна, коли жодне з вхідних ребер не має значення.

Розглянемо приклад графа потоку даних. Нехай є система, що складається з двох лінійних алгебраїчних рівнянь виду.

$$a_{11} * x_1 + a_{12} * x_2 = b_1$$
$$a_{21} * x_1 + a_{22} * x_2 = b_2$$

У разі, якщо

$$\Delta = a_{11} * a_{12} - a_{12} * a_{21} \neq 0$$

то дана система має рішення

$$x_1 = \frac{1}{\Delta} * (a_{22} * b_1 - a_{12} * b_2)$$
$$x_2 = \frac{1}{\Delta} * (a_{11} * b_2 - a_{21} * b_1)$$

якому відповідає схема алгоритму на рис.3.17.

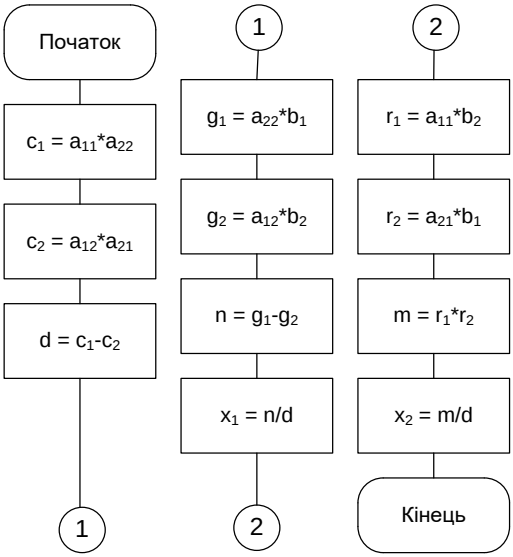


Рис.3.17. Схема алгоритму розв'язання системи з двох лінійних алгебраїчних рівнянь

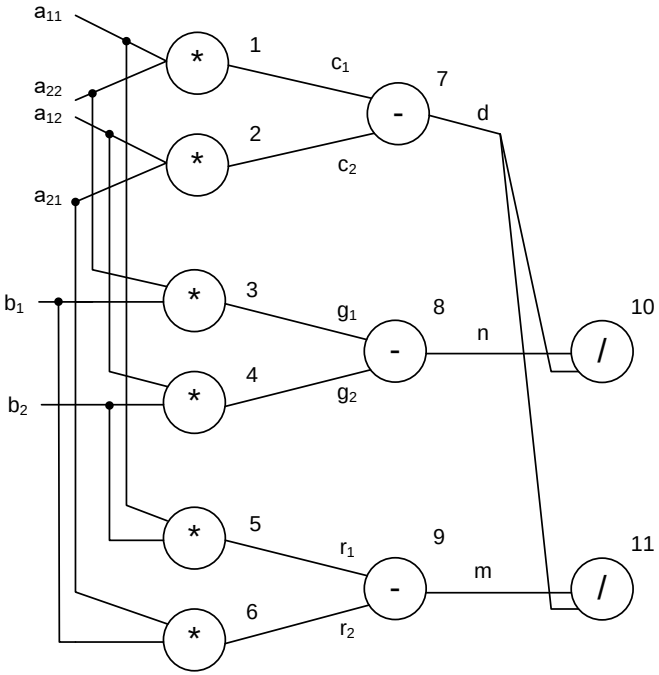


Рис 3.18. Граф потоку даних, відповідний рішенню системи двох лінійних алгебраїчних рівнянь

Формати команд і даних для Dataflow систем

Розглянемо структуру команд і даних в Dataflow системі, що реалізує програми, представлені у вигляді графа потоку даних. Як правило, такі команди складаються з чотирьох або шести полів (рис.3.19.).

КОП	ПА	АДР	ПО

КОП	ПА	АДР1	ПО	АДР2	ПО

Рис. 3.19. Формати команд для Dataflow системи

Поле КОП містить код операції, поле ПА - ознака активності заданої команди (0 - неактивна, 1 - активна або активізована). Поле АДР містить адресу (адреси) подальшою команди (команд), поле ПЗ - ознака черговості

операндів (0 - лівий операнд, 1 - правий операнд). Наприклад, для команд, показаних на рис.3.18., команда 1 буде виглядати наступним чином:

*	1	7	0

а команда 7 буде мати вигляд:

–	0	10	1	11	1

Значення полів АДР1 і АДР2 -10 і 11 - це номери наступних (за командою 7) команд.

Формат даних має наступний вигляд:

РЗ	АДР	ПО

Поле РЗ містить результат виконання команди, а поля АДР і ПО, відповідно, адресу команди, в якій використовується це дане і ознака черговості операндів. Таким чином, дане, обчислене командою 1, виглядатиме так:

$a_{11} * a_{22}$	7	0

Виконання команди здійснюється у два етапи: на першому етапі по вхідним значенням даних формується результат. При цьому в полі АДР і ПО результату поміщається зміст полів АДР і ПО команди, а в полі РЗ поміщається результат виконання вказаної у полі КОП операції. Ознака активності в команді встановлюється в стан 0 (команда стає неактивною). На другому етапі здійснюється виклик і передача значення результату команді (командам), адреса (и) якої (-их) вказано (-і) в полі АДР.

Наприклад, до виконання команда 3 (рис.3.18.) виглядає так:

*	0	8	0

а результат команди відсутній (не має значення). Після виконання команди 3 з'явиться результат

$a_{22} * b_1$	8	0

До виконання команда 7 виглядає так:

-	0	10	0	11	0

У результаті виконання команди 7 формуються дані (результати):

$a_{11} * a_{22}$	10	0

$a_{11} * a_{22}$	11	0

Структура Dataflow системи

Розглянемо один із можливих варіантів структурних рішень Dataflow систем (рис.3.20.).

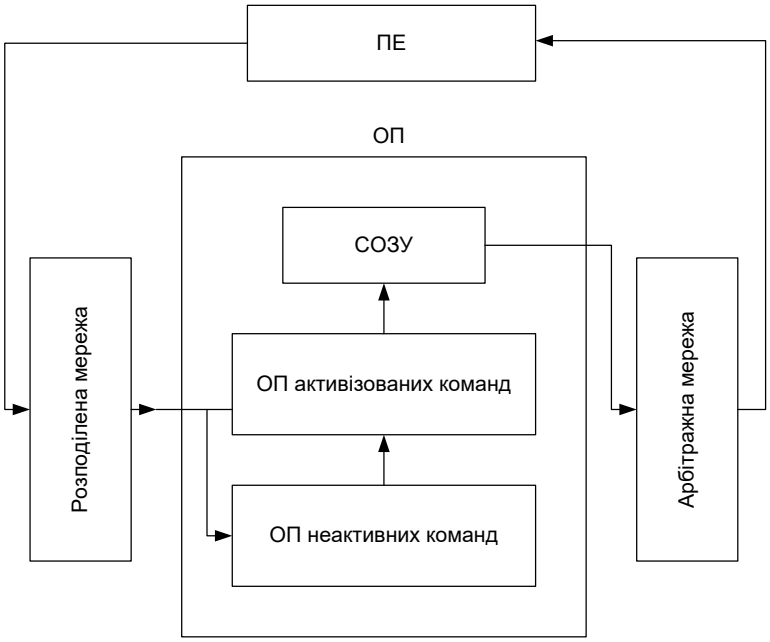


Рис.3.20. Структура системи, керованої потоком даних

Оперативна пам'ять (ОП) такої системи складається з трьох компонент: ОП неактивних команд, для яких жоден операнд не готовий; ОП активізованих команд, для яких готовий тільки один операнд і СОЗУ

для зберігання активних команд (готові всі операнди). Власне операції виконуються за допомогою набору процесорних елементів (ПЕ). Зв'язок між ОП і ПЕ здійснюється за допомогою двох мереж пересилань: арбітражна мережа служить для передачі пакетів готових команд ПЕ, а розподілена мережа - для пересилання результатів у ті командні комірки ОП, в яких ці результати використовуються вже як операнди. Початкове заповнення компонент оперативної пам'яті проводиться за результатами компіляції програми. Спочатку виконуються команди, що залежать тільки від вхідних даних і констант. Подальше виконання програми здійснюється наступним чином: після обчислення результату чергової команди спочатку відбувається звернення до командних комірок активізованих команд. В якості ключа пошуку в даній асоціативної пам'яті виступає значення поля адреси виконаної команди. У разі, якщо дана команда знаходиться в пам'яті активізованих команд, здійснюється її передача разом з необхідними для її виконання операндами в СОЗУ. У разі відсутності цієї команди в пам'яті активізованих команд, проводиться її пошук в ОП неактивних команд і подальше її переміщення в пам'ять активізованих команд.

Максимальна продуктивність системи визначається швидкодією і числом ПЕ, а також пропускними здатностями пам'яті і мереж пересилань.

Компіляція програм для Dataflow систем

Компіляція програм для Dataflow системи означає перетворення програми на традицій мовіу послідовність потокових команд, представлених у вигляді графа потоку даних. Такий процес компіляції складається з наступних етапів:

1) За вхідним текстом отримання програми утетрадному вигляді (стандартна компіляція).

2) Формування таблиць операндів.

3) По таблиці операндів отримання послідовності команд у форматі Dataflow систем.

Розглянемо етапи компіляції програми для Dataflow системи на прикладі рішення системи лінійних алгебраїчних рівнянь. Схема алгоритму даної задачі представлена на рис.3.17. , а її граф потоку даних - на рис.3.18.

Послідовності команд у тетрадному вигляді має наступний вигляд:

№ ком.	КОП	Операнд 1	Операнд 2	Результат
1	введення	-	-	a_{11}
2	введення	-	-	a_{12}
3	введення	-	-	b_1
4	введення	-	-	a_{21}

5	введення	-	-	a_{22}
6	введення	-	-	b_2
7	*	a_{11}	a_{22}	c_1
8	*	a_{12}	a_{21}	c_2
9	*	a_{22}	b_1	g_1
10	*	a_{12}	b_2	g_2
11	*	a_{11}	b_2	r_1
12	*	a_{21}	b_1	r_2
13	–	c_1	c_2	D
14	–	g_1	g_2	N
15	–	r_1	r_2	M
16	/	N	d	x_1
17	/	M	d	x_2
18	виведення	x_1		
19	виведення	x_2		

Формування таблиці операндів проводиться таким чином. Для кожного ідентифікатора операнда здійснюється запис номера команди, що використовує цей операнд, у відповідне поле таблиці операндів. Заповнена таким чином таблиця має наступний вигляд:

Операнд	Ком. 1	ОЧ	Ком. 2	ОЧ
a_{11}	7	0	11	0
a_{12}	8	0	10	0
a_{21}	8	1	12	0
a_{22}	7	1	9	0
b_1	9	1	12	1
b_2	10	1	11	1
c_1	13	0		
c_2	13	1		
g_1	14	0		
g_2	14	1		
r_1	15	0		
r_2	15	1		
n	16	0		
d	16	1	17	1
m	17	0		
x_1	18	0		
x_2	19	0		

Формування послідовності команд у форматі Dataflow систем проводиться таким чином. По ідентифікаторах результатів здійснюється звернення у таблицю операндів і поля, що містять номери команд, переписуються в рядок програми, що розглядається. Наприклад, у команді 13 стоїть ідентифікатор результату d. Звертаємося до таблиці операндів з цим ідентифікатором. У таблиці операндів у рядку, що містить d, знаходяться номери команд 16 і 17. Ці номери переписуємо в рядок команди 13 після ідентифікатора d. Таким чином, відкомпільований текст програми має наступний вигляд:

№ ком.	КОП	ОА	Адреса 1	ОЧ	Адреса 2	ОЧ
1	введення	1	7	0	11	0
2	введення	1	8	0	10	0
3	введення	1	9	1	12	1
4	введення	1	8	1	12	0
5	введення	1	7	1	9	0
6	введення	1	10	1	11	1
7	*	1	13	0		
8	*	1	13	1		
9	*	1	14	0		
10	*	1	14	1		
11	*	1	15	0		
12	*	1	15	1		
13	–	0	16	1	17	1
14	–	0	16	0		
15	–	0	17	0		
16	/	0	18	0		
17	/	0	19	0		

Як видно з отриманої таблиці, команди 1-12 - активні, а 13-17 - неактивні. Активні команди можуть бути виконані паралельно на різних процесорах Dataflow системи. Таким чином, у процесі компіляції виконується початковий етап розпаралелювання програми, який потім динамічно продовжується під час її виконання.

Література

1. El-Rewini H., T.G.Lewis Distributed and Parallel Computing.-Manning Publications Co.,1998.-447 p.
2. Корочкін О.В. Багатоядерне програмування на мові Ада. Навч.посібн.Част. І.(англ. мовою).- Київ, КПІ ім. І.Сікорського, 2018.- 226 с
3. Ахо А., Р. Сети, Дж.Ульман Компиляторы: принципы, технологии и инструменты,-М.:Изд. Дом “Вильямс”, 2001.-768 с.
4. Жуков І.. Корочкін О.В. Паралельні та розподілені обчислення. Навч.посібн..- Київ, «Корнійчук» 2014.- 226 с.
5. Бройнль Томас. Паралельне програмування: Початковий курс: Навч.посібник.-К.:Вища школа,1997.-358 с.
6. Вальковский В., В.Е.Котов, А.Г.Марчук, Н.Н.Миренков. Элементы параллельного программирования.-М.:Радио и связь,1983.-240 с.
7. Корнеев В.В. Параллельные вычислительные системы.-М.: “Нолидж”, 1999.-320 с.
8. СуперЭВМ. Аппаратная и программная организация / Под ред. С.Фернбаха.-М.:Радио и связь,1991.-320 с.
9. Сырков Б.Ю., С.В.Матвеев. программное обеспечение мультитранспьютерных систем.-М.: “ ДИАЛОГ-МИФИ ”, 2002.-224 с.
10. Транспьютеры, Архитектура и программное обеспечение / Под ред. Г.Харпа.-М.:Радио и связь,1993.-304 с.
11. Трахтенгерц Э. Программное обеспечение параллельных процессов.М.:Наука,1987. - 272 с.
12. Хокни Р., Джессхоуп К. Параллельные ЭВМ. Архитектура, программирование и алгоритмы.-М.:Радио и связь,1995.-358 с.
13. Abbas A., Grid Computing: A Practical Guide to Technology and Applications. - Charles River Media, Hingham, MA, -2004.
14. Albaharis J. Threading in C#. – O’Reilly Media,Inc., 2006.
15. Lester B., The Art of Parallel Programming. - 2nd ed., 1st World Publishing, Fairfield, IA, 2006.
16. http://www.silicon-russia.com/2019/04/15/triton-skolkovo-robotics-ai/
17. М.Кузьминский Векторные процессоры против акселераторов//Открытые системы.СУБД.-2018.-№1 http://www.osp.ru/os/2018/01/13053934
18. http://www.habr.com/ru/
19. http://servernews.ru/997596
20. https://servernews.ru/999379/?utm_source=hw