МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

РОЛЬЩИКОВ В.Б.

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Конспект лекцій

Одеса
Одеський державний екологічний університет
2016

УДК 681.518
Р68

Рекомендовано методичною радою Одеського державного екологічного університету Міністерства освіти і науки України як конспект лекцій (протокол №7 від 31.03. 2016 р.)

Рольщиков В. Б.
Технології розподілених систем та паралельних обчислень: конспект лекцій. Одеса, Одеський державний екологічний університет, 2016. 155 с.

В конспекті лекцій з курсу «Технології розподілених систем та паралельних обчислень» розглянуті основні питання архітектури систем паралельних обчислень. Паралельні обчислення є перспективною (і дуже привабливою) областю застосування обчислювальної техніки і являють собою складну науково-технічну область діяльності. Тим самим, знання сучасних тенденцій розвитку комп'ютерів і апаратних засобів для досягнення паралелізму, уміння розробляти моделі, методи й програми паралельного рішення завдань обробки даних варто віднести до числа важливих кваліфікаційних характеристик сучасного фахівця із прикладної математики, інформатиці й обчислювальній техніці.

Конспект лекцій призначений для студентів четвертого курсу Одеського державного екологічного університету, які навчаються за кваліфікаційним рівнем «бакалавр» та спеціальністю 07.05010101 «Інформаційні управляючі системи та технології (за галузями)».

ISBN 978-966-186-096-3

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ І УМОВНИХ ПОЗНАЧЕНЬ

АЛП – арифметико-логічний пристрій

БУП – блок управління пам’яттю

ГіБ – гібібайт

Гіб – гібібіт

Гіб/с – гібібіт у секунду

Гібі – приставка, яка замінює множник до деякої одиниці вимірювання, кратна ступеню двійки і дорівнює 2^{30}

Демон – вартовий процес, який постійно знаходиться у ОЗП (див. нижче) і відслідковує появу деякої наперед визначеної події

Квод – обробний обчислювальній блок у паралельних системах типу COMA (див. нижче) з пам’яттю, яка притягує (Attraction Memories)

КіБ – кібібайт

Кіб – кібібіт

Кібі – приставка, яка замінює множник до деякої одиниці вимірювання, кратна ступеню двійки і дорівнює 2^{10}

КОП – код операції

Мебі – приставка, яка замінює множник до деякої одиниці вимірювання, кратна ступеню двійки і дорівнює 2^{20}

Мегафлоп – 10^6 флопів (див. нижче)

МіБ – мебібайт

Міб – мебібіт

Міб/с – мебібіт у секунду

МОК – мультипроцесорний обчислювальній комплекс

ОЗП – оперативний запам’ятовуючий пристрій

ОС – операційна система

ПВВ – пристрій введення/виведення

Петафлопс – 10^{15} флопів (див. нижче) у секунду

ПЗ – програмне забезпечення

ПЗП – постійний запам’ятовуючий пристрій

ППД – процесор передачі даних

РАН – Російська академія наук

Стаб – оболонка для локальної процедури або локального методу у системах віддаленого програмування (калька з англ. stub – заглушка)

Тебі – приставка, яка замінює множник до деякої одиниці вимірювання, кратна ступеню двійки і дорівнює 2^{40}

Терафлоп – 10^{12} флопів (див. нижче)

Тіб/с – тебібіт у секунду

Флоп – операція з рухомою крапкою (калька з англ. абревіатури flop – floating point operation)

Флопс – операцій з рухомою крапкою у секунду (калька з англ. абревіатури flops – floating point operation per second)

ANSI – Національний інститут стандартів США (American National Standard Institute)

CC-NUMA – мультипроцесори архітектури NUMA (див. нижче) з погодженою кеш-пам’яттю – Coherent Cache NUMA

COMA – мультипроцесори з архітектурою з доступом тільки до кеш-пам’яті – Cache Only Memory Access

CORBA – технологічний стандарт написання розподілених додатків з відповідною йому інформаційною технологією, загальна архітектура брокера об’єктних запитів – Common Object Request Broker Architecture

COW – мультикомп’ютер на основі комп’ютерів з’єднаних звичайною локальною обчислювальною мережею, кластер робочих станцій – Cluster Of Workstations

DDR SDRAM – синхронна динамічна пам'ять із довільним доступом і по-

двоєною швидкістю передачі даних – Double Data Rate Synchronous Dynamic Random Access Memory

DSM – технологія програмної емуляції спільно використовуваної пам’яті, розподілена, спільно використовувана пам’ять – Distributed Shared Memory

IDL – спеціальна мова визначення і опису інтерфейсу зв’язку в технології віддалених викликів процедур – Interface Definition Language

MESI – протокол когерентності кешування зі зворотним записом, має чотири стани: Modified – рядок кеш змінений, Exclusive – тільки один кеш містить рядок, Shared – кілька кешів містять рядок, Invalid – у рядку знаходяться недійсні дані

MIMD – паралельна обчислювальна система, яка має багато потоків команд і багато потоків даних – Multiple Instruction stream Multiple Data stream

MISD – уявна, не існуюча архітектура комп’ютера, в якій функціонує кілька потоків команд і один потік даних – Multiple Instruction stream Single Data stream

mpC – клон мови програмування C для гетерогенних кластерів робочих станцій – multi processor C

MPI – бібліотека примітивів середнього рівня для створення паралельних програм за технологіями розпаралелювання задач і даних, інтерфейс передачі повідомлень– Message-Passing Interface

MPP – архітектура мультикомп’ютерів на базі спеціалізованих комп’ютерів з високошвидкісною мережею міжз’єднань, процесори з масовим паралелізмом – Massively Parallel Processors

NC-NUMA – мультипроцесори архітектури NUMA (див. нижче) без

кэшування – No Caching NUMA

NFS – власна файлова система операційної системи – Native File System

NORMA – паралельні комп’ютери без доступу до віддалених модулів пам’яті, але з встановленою системою DSM – NO Remote Memory Access

NOW – мережа робочих станцій на основі звичайних комп’ютерів у звичайній локальній мережі – Network of Workstations (інша назва кластер робочих станцій COW)

NUMA – архітектура мультипроцесорів з неоднорідним доступом до пам’яті – NonUniform Memory Access

PVM – частина програмного забезпечення відкритих операційних систем для проведення паралельних обчислень на неоднорідних кластерах, паралельна віртуальна машина – Parallel Virtual Machine

RMI – технологія розподілених обчислень мовою Java за допомогою викликів віддалених методів – Remote Method Invocation

RPC – технологія розподілених обчислень мовами C і Fortran за допомогою викликів віддалених процедур – Remote Procedure Call

SIMD – архітектура паралельних комп’ютерів, у яких за допомогою одного потоку команд обробляється кілька потоків даних – Single Instruction stream Multiple Data stream

SISD – архітектура комп’ютерів з одним потоком команд і одним потоком даних – Single Instruction stream Single Data stream (звичайний одно процесорний комп’ютер фон Неймана)

SMP – мультипроцесор у якого всі процесори рівноправні (симетричний) і мають однаковий доступ до всіх пристроїв си-

стеми – Symmetric Multiprocessor

SPMD – парадигма паралельного програмування, коли одна програма обробляє кілька потоків даних – Single Program Multiple Data

Sun UltraSPARC – мікропроцесор фірми Sun Microsystems

UMA – архітектура мультипроцесорів, в яких всі процесори мають однорідний, з однаковим часом, доступ до будь-якого модуля пам’яті – Uniform Memory Access

1 ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ

Розв’язання складних проблем «великого виклику» можливостям сучасної науки й техніки: моделювання клімату, генної інженерії, проектування інтегральних схем, аналіз забруднення навколишнього середовища, створення лікарських препаратів і всіх подібних завдань, неможливе без:

- організації паралельних обчислень, коли в той самий момент виконується одночасно кілька операцій обробки даних;
- паралельного узагальнення традиційної – послідовної технології вирішення задач на комп’ютері;
- прискорення процесу розв’язання обчислювальної задачі при розподілі застосовуваного алгоритму на інформаційно незалежні частини й організація виконання кожної частини обчислень на різних процесорах.

Всім цим проблемам присвячений перший модуль цього конспекту. Але говорити про них не має ніякого сенсу, якщо проблеми вирішуються у відриві від розуміння принципів роботи апаратного забезпечення, яке призначено для їхнього вирішення, від знання структури й властивостей програмного забезпечення, яке застосовується для розпаралелювання процесів.

Даний модуль курсу лекцій саме й присвячений розгляду питань *розробки апаратного* забезпечення і принципам *побудови програмного забезпечення* паралельних обчислень.

1.1 Побудова паралельних обчислювальних систем

Проблема створення високопродуктивних обчислювальних систем відноситься до числа найбільш складних науково-технічних завдань сучасності. Її вирішення можливо тільки при всілякій концентрації зусиль багатьох талановитих учених і конструкторів, передбачає використання всіх останніх досягнень науки й техніки та вимагає значних фінансових інвестицій.

1.1.1 Загальні питання створення паралельних систем

Коли розроблювачі приступають до створення нової комп'ютерної системи паралельної дії, вони, у першу чергу повинні вирішити три виникаючі при цьому проблеми:

1. зі скількох процесорних елементів, якого типу й розміру буде складатися розроблювальний комп'ютер;
2. скільки модулів пам'яті, якого типу й розміру встановити у системі;
3. яким чином будуть взаємодіяти процесорні елементи з елементами пам'яті.

У сьогодення в розробках можуть використовуватися процесорні елементи всіляких типів – від щонайменших АЛП до повних центральних процесорів. Розмір одного процесора може варіюватися в межах від невеликої частини мікросхеми до кубічного метра електроніки. Очевидно, що якщо процесорний елемент являє собою частину мікросхеми, то можна помістити в комп'ютер величезну кількість таких елементів (наприклад, мільйон). Якщо процесорний елемент являє собою цілий комп'ютер зі своєю пам'яттю й пристроями введення/виведення, цифри будуть набагато меншими, хоча були сконструйовані такі системи з кількістю до 10000 процесорів. Слід зазначити, що дотепер застосування паралельних комп'ютерів не одержало настільки широкого поширення, як це очікувалося. Однієї з можливих причин подібної ситуації була донедавна висока вартість високопродуктивних систем. Сучасна тенденція побудови паралельних обчислювальних комплексів з типових конструктивних елементів (мікропроцесорів, мікросхем пам'яті, комунікаційних пристроїв), масовий випуск яких освоєний промисловістю, знизила вплив цього фактора, і в даний момент практично кожний споживач може мати у своєму розпорядженні багатопроцесорні обчислювальні системи досить високої продуктивності. Зараз комп'ютери паралельної дії *конструюються з частин, які випускаються серійно*. Розробка комп'ютерів паралельної дії часто залежить від того, які функції виконують ці частини і які обмеження на-

кладаються на них.

Системи пам’яті часто розділяються на модулі, які працюють незалежно друг від друга, щоб кілька процесорів одночасно могли здійснювати доступ до пам’яті. Ці модулі можуть бути маленького розміру (кілька десятків кібібайтів) або великого розміру (мебі- й навіть гібібайти). Конструктивно вони можуть розташовуватися поруч із процесорами, або на іншій платі. Динамічна пам’ять працює набагато повільніше центральних процесорів, тому для підвищення швидкості доступу до пам’яті звичайно використовуються різні схеми кеш-пам’яті. Може бути два, три й навіть чотири рівні кеш-пам’яті.

Хоча існують найрізноманітніші процесори і системи пам’яті, системи паралельної дії різняться в основному тим, як з’єднані різні частини. Схеми взаємодії можна розділити на дві категорії: ***статичні*** й ***динамічні***. У *статичних схемах* компоненти безпосередньо зв’язуються один з одним певним чином. Як приклади статичних схем можна привести зірку, кільце й ґрати. У *динамічних схемах* усі компоненти приєднані до перемикальної схеми, яка може трасувати повідомлення між компонентами. У кожній з цих схем є свої позитивні якості й недоліки.

Комп’ютери паралельної дії можна розглядати як набір мікросхем, які з’єднані одна з одною певним чином. Це перший підхід. При іншому підході виникає питання, які саме процеси виконуються паралельно. Тут існує кілька варіантів. Деякі комп’ютери паралельної дії одночасно виконують кілька незалежних завдань. Ці завдання ніяк не пов’язані один з одним і не взаємодіють. Типовий приклад – комп’ютер, що містить від 8 до 64 процесорів, який являє з себе велику систему UNIX з розподілом часу, з якою можуть працювати тисячі користувачів. До цієї категорії потрапляють системи обробки транзакцій, які використовуються в банках (наприклад, банківські автомати), на авіалініях (наприклад, системи резервування квитків) і у великих web-серверах. Сюди ж відносяться незалежні прогони моделюючих програм, при яких використовується кілька наборів параметрів.

Інші комп'ютери паралельної дії виконують єдине завдання, що складається з декількох одночасних процесів. Як приклад можна привести програму гри в шахи, що аналізує дані позиції на дошці, породжує список можливих ходів із цих позицій, а потім породжує паралельні процеси, щоб проаналізувати кожну нову ситуацію одночасно. Тут паралелізм потрібний не для того, щоб обслуговувати велику кількість користувачів, а щоб прискорити розв'язання однієї задачі.

Далі йдуть машини з високим ступенем конвеєризації або з великою кількістю АЛП, які обробляють одночасно один потік команд. У цю категорію потрапляють суперкомп'ютери зі спеціальним апаратним забезпеченням для обробки векторних даних. Тут вирішується одне головне завдання, і при цьому всі частини комп'ютера працюють разом над одним аспектом цього завдання (наприклад, різні елементи двох векторів підсумуються паралельно).

Ці три приклади різняться по так званому ***ступеню деталізації***. У багатопроцесорних системах з розподілом часу блок паралелізму досить великий – ціла користувальницька програма. Паралельна робота великих частин програмного забезпечення практично без взаємодії між цими частинами називається ***паралелізмом на рівні великих структурних одиниць***. Діаметрально протилежний випадок (при обробці векторних даних) називається ***паралелізмом на рівні дрібних структурних одиниць***.

Термін *ступінь деталізації* застосовується по відношенню до алгоритмів і програмного забезпечення, але в нього є прямий аналог в апаратному забезпеченні. Системи з невеликим числом великих процесорів, які взаємодіють по схемах з низькою швидкістю передачі даних, називаються системами з ***непрямим*** (***слабким***) ***зв'язком***. Їм протиставляються системи з ***безпосереднім*** (***тісним***) ***зв'язком***, у яких компоненти звичайно менші за розміром, розташовані ближче друг до друга і взаємодіють через *спеціальні комунікаційні мережі* з високою пропускною здатністю. У більшості випадків задачі з паралелізмом на рівні великих структурних одиниць найкраще вирішуються в системах зі слабким зв'язком, а задачі з паралелізмом на рівні дрібних струк-

турних одиниць найкраще вирішуються в системах з безпосереднім зв’язком. Однак існує безліч різних алгоритмів і безліч різноманітного програмного й апаратного забезпечення. Розмаїтість ступеня деталізації й можливості різного ступеня зв’язності систем призвели до різноманіття архітектур, які й будуть розглянуті нижче.

1.1.2 Мультипроцесори і мультикомп’ютери

У будь-якій системі паралельної обробки процесори, які виконують різні частини одного завдання, повинні якось взаємодіяти один з одним, щоб обмінюватися інформацією. Для організації такого обміну було запропоновано й реалізоване дві розробки:

− мультипроцесори;

− мультикомп’ютери.

У першій розробці всі процесори розділяють загальну фізичну пам’ять, як показано на рис. 1.1 а). Така система називається ***мультипроцесором*** або ***системою зі спільно використовуваною пам’яттю***.

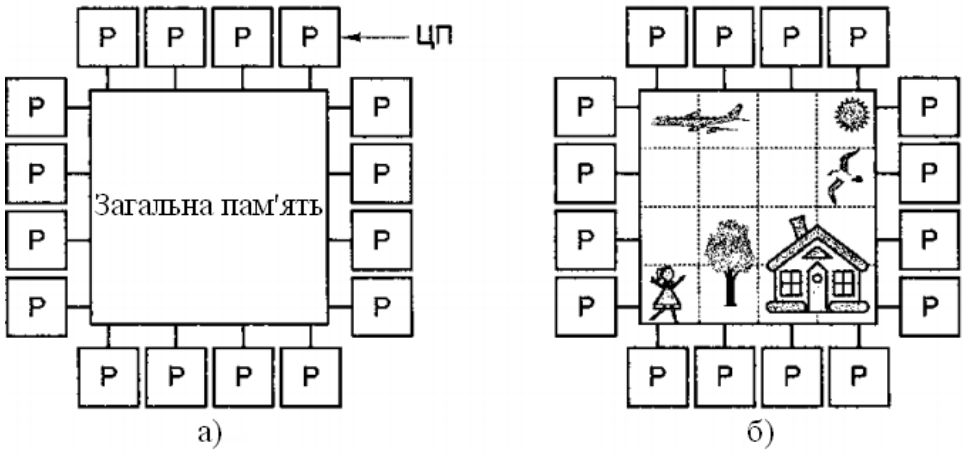


Рисунок 1.1 − Схематичне представлення системи зі спільно використовуваною пам’яттю

а) мультипроцесор з 16 процесорів, що розділяють загальну пам’ять

б) 16 фрагментів зображення, які аналізуються окремими процесорами

Мультипроцесорна модель розповсюджується на програмне забезпечення. Всі процеси, що працюють разом на мультипроцесорі, можуть поділяти один віртуальний адресний простір, відображений у загальну пам’ять. Будь-який процес може зчитувати слово з пам’яті або записувати слово у пам’ять за допомогою звичайних команд читання/запису. Два процеси можуть обмінюватися інформацією, якщо один з них буде просто записувати дані у пам’ять, а інший буде зчитувати ці дані.

Завдяки такій можливості взаємодії двох і більше процесів мультипроцесори досить популярні. Дана модель є зрозумілою програмістам і може бути застосованою до широкого кола задач. Прикладом такої задачі може слугувати програма, що обробляє якесь зображення (рис. 1.1 б). У пам’яті зберігається одна копія зображення. Кожний з 16 процесорів запускає один процес, якому приписана для аналізу одна з 16 секцій. Якщо процес виявляє, що один з його об’єктів переходить через границю секції, цей процес просто переходить слідом за об’єктом у наступну секцію, зчитуючи слова цієї секції. У прикладі деякі об’єкти обробляються відразу декількома процесами, тому наприкінці буде потрібно деяка координація, щоб визначити кількість будинків, дерев і літаків.

У другому типі паралельної архітектури кожний процесор має свою власну пам’ять, доступну тільки цьому процесору. Така розробка називається ***мультикомп’ютером*** або ***системою з розподіленою пам’яттю***. Така система умовно зображена на рис. 1.2 а). Мультикомп’ютери звичайно є системами зі слабким зв’язком (але це не обов’язково). Ключова відмінність мультикомп’ютера від мультипроцесора полягає в тому, що кожний процесор у мультикомп’ютері має свою власну локальну пам’ять, до якої цей процесор може звертатися, виконуючи команди читання/запису, але ніякий інший процесор не може одержати доступ до цієї пам’яті за допомогою тих самих команд. Таким чином, мультипроцесори мають один фізичний адресний простір, поділюваний всіма процесорами, а мультикомп’ютери мають окремий фізичний адресний простір для кожного центрального процесора.

Оскільки процесори в мультикомп’ютері не можуть взаємодіяти один з одним просто через загальну пам’ять, то тут є необхідним інший механізм взаємодії. Процесори посилають один одному *повідомлення* через мережу міжз’єднань. При відсутності пам’яті спільного використання в апаратному забезпеченні, у системі передбачається певна структура програмного забезпечення. У мультикомп’ютері неможливо мати один віртуальний адресний простір, з якого всі процеси можуть зчитувати інформацію й у яке всі процеси можуть записувати інформацію. Наприклад, якщо процесор (у верхньому лівому куті) на рис. 1.2 б) виявляє, що частина його об’єкта потрапляє у пам’ять іншого процесора (літак перемістився), він не може просто зчитати потрібну йому інформацію, як це було б у мультипроцесорі. Для одержання необхідних даних йому потрібно зробити щось інше.

Рисунок 1.2 − Умовне представлення мультикомп’ютера

а) – 16 процесорів, кожний зі своєю власною пам’яттю

б) – зображення розподілене між 16 ділянками пам’яті

Зокрема, йому потрібно якось визначити, який процесор утримує необхідні йому дані, і послати цьому процесору повідомлення із запитом копії даних. Потім процесор блокується до одержання відповіді. Коли процесор-

хазяїн даних отримує повідомлення, програмне забезпечення повинне проаналізувати його й відправити назад необхідні дані. Коли процесор, який запросив дані, дістає відповідне повідомлення, програмне забезпечення розблокується і продовжує роботу.

У мультикомп'ютері для взаємодії між процесорами часто використовуються деякі примітиви посилки й одержання повідомлення. Тому програмне забезпечення мультикомп'ютера має більше складну структуру, чим програмне забезпечення мультипроцесора. При цьому основною проблемою стає правильний поділ даних і розумне їхнє розміщення. У мультипроцесорі розміщення частин не впливає на правильність виконання задачі, хоча може вплинути на продуктивність. Таким чином, мультикомп'ютер програмувати набагато складніше, ніж мультипроцесор.

Складність у створенні програмного забезпечення мультикомп'ютерів жодним чином не обмежує їхнє використання, навпроти кількість систем такого типу у світі постійно зростає. Цей факт пояснюється тим, що набагато простіше й дешевше побудувати великий мультикомп'ютер, ніж мультипроцесор з такою же кількістю процесорів. Реалізація загальної пам'яті, яка поділяється кількома сотнями процесорів, – це досить складне завдання, а побудувати суперкомп'ютер, що містить до десяти тисяч і навіть більше процесорів, відносно легко.

Таким чином, виникла дилема: мультипроцесори складно будувати, але «легко» програмувати, а мультикомп'ютери «легко» будувати, але важко програмувати. Тому розроблювачі апаратного забезпечення стали вживати спроби створення гібридних систем, які відносно легко конструювати й відносно легко програмувати. Це призвело до усвідомлення того, що спільну пам'ять можна реалізовувати по-різному, і в кожному випадку будуть якісь переваги й недоліки. Практично всі дослідження в ділянці архітектур з паралельною обробкою спрямовані на створення гібридних форм, які об'єднують у собі переваги обох архітектур. Тут важливо отримати таку систему, яка буде здатна до розширення, тобто буде продовжувати справно працювати при додаванні все

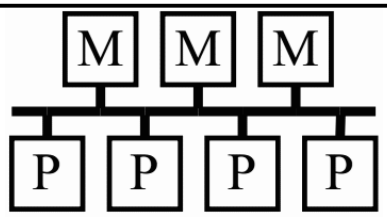
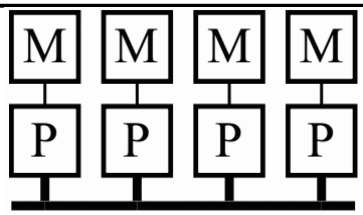
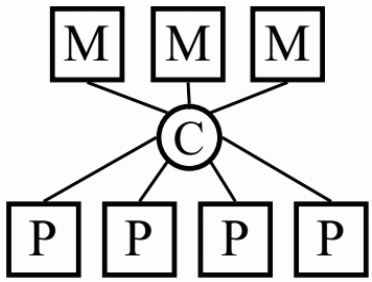
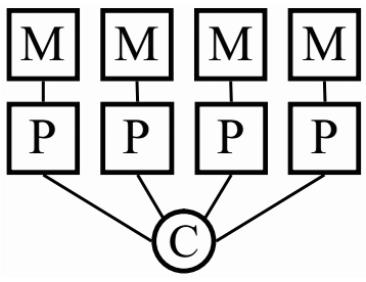
нових і нових процесорів.

1.1.3 Мережі міжз’єднань паралельних систем

Вище говорилося про те, що в мультикомп’ютері процесори зв’язані між собою деякою мережею міжз’єднань. Але необхідно особливо підкреслити, що й *мультипроцесори теж містять мережу міжз’єднань*, яка зв’язує окремі модулі пам’яті, останні також повинні бути зв’язані і з процесорами.

У теперішній час існує цілий спектр різних архітектур як мультипроцесорів, так і мультикомп’ютерів, причому кожна із цих категорій може бути підрозділена на додаткові категорії на основі архітектури тій мережі, яка їх з’єднує. У табл. 1.1 наведено структурні схеми основних архітектур обох категорій суперкомп’ютерів. На схемах використані такі позначення: М – модуль пам’яті, Р – процесорний блок, С – комутуюча мережа міжз’єднань.

Таблиця 1.1 − Структурні схеми основних архітектур мультипроцесорів і мультикомп’ютерів

	Мультипроцесор	Мультикомп’ютер
Шинна архітектура		
Комутована архітектура		

Основна причина подібності комунікаційних зв’язків у мультипроцесорі і у мультикомп’ютері полягає в тім, що в обох випадках застосовується пе-

редача повідомлень. Навіть в однопроцесорній машині, коли процесору потрібно зчитати або записати слово, він установлює певні лінії на шині й чекає відповіді. Ця дія являє собою те ж саме, що й передача повідомлень: ініціатор надсилає запит і чекає відповіді. У великих мультипроцесорах взаємодія між процесорами і віддаленою пам'яттю майже завжди полягає в тому, що процесор посилає у пам'ять повідомлення, так званий пакет, який запитує певні дані, а пам'ять посилає процесору відповідний пакет.

У зв'язку зі сказаним, у першу чергу необхідно розглянути структуру й роботу мереж міжз'єднань тому що вони є одним з основних елементів архітектури обох типів суперкомп'ютерів.

Мережі міжз'єднань можуть складатися максимум з п'яти компонентів:

- центральні процесори;
- модулі пам'яті;
- інтерфейси;
- канали зв'язку;
- комутатори.

Процесори і модулі пам'яті встановлюються стандартних типів з тих, що наявні на ринку на момент розробки суперкомп'ютера й ні чим не відрізняються від процесорів звичайних персональних комп'ютерів. Розгляд роботи цих пристроїв не входить у завдання даного курсу лекцій. Інтерфейси – це пристрої, які вводять і виводять повідомлення із центральних процесорів і модулів пам'яті. У багатьох розробках інтерфейс являє собою мікросхему або плату, до якої приєднується локальна шина кожного процесора і яка може передавати сигнали процесору і локальній пам'яті (якщо така є). Часто усередині інтерфейсу міститься програмувальний процесор зі своїм власним ПЗП, що належить тільки цьому процесору. Звичайно інтерфейс здатний зчитувати й записувати інформацію в різні модулі пам'яті, що дозволяє йому переміщати блоки даних.

Канали зв'язку – це канали, по яких переміщаються біти інформації.

Канали можуть бути електричними або оптико-волоконними, послідовними (шириною 1 біт) або паралельними (шириною більше 1 біта). Кожний канал зв'язку характеризується ***максимальною пропускною здатністю*** (це максимальне число бітів, яке він здатний передавати в секунду). Канали можуть бути *симплексними* (передавати біти тільки в одному напрямку), *напівдуплексними* (передавати інформацію в обох напрямках, але не одночасно) і *дуплексними* (передавати біти в обох напрямках одночасно).

Комутатори – це пристрої з декількома вхідними й декількома вихідними портами. Коли на вхідний порт приходить пакет, деякі біти в цьому пакеті використовуються для вибору номера вихідного порту, до якого буде переспрямований пакет. Розмір пакета може становити 2 або 4 байти, але може бути й значно більше (наприклад, 8 КіБ).

При розробці і аналізі мережі міжз'єднань важливо враховувати кілька ключових моментів. По-перше, це *топологія* (тобто спосіб розташування компонентів). По-друге – *принцип роботи* системи перемикання каналів і *технологія* здійснення зв'язку між ресурсами. По-третє – *алгоритм вибору маршруту* для доставки повідомлень між пунктами передачі й одержання пакетів. Має сенс розглянути кожний із цих трьох пунктів.

Топологія мережі міжз'єднань визначає, як розташовані канали зв'язку й комутатори. Цілком очевидно, що топології зображують у вигляді графів, у яких дуги відповідають каналам зв'язку, а вузли – комутаторам (рис. 1.3). З кожним вузлом у мережі (у графі) зв'язаний певний ряд каналів зв'язку. У математиці число дуг пов'язаних з вершиною називають ***ступенем вузла***, у техніці частіше використовується термін ***коефіцієнт розгалуження***. Чим більше коефіцієнт розгалуження, тим більше варіантів маршрутів передачі даних і тем вище стійкість мережі до відмов. Якщо кожний вузол містить k дуг і з'єднання зроблене правильно, то можна побудувати мережа міжз'єднань так, щоб вона залишалася *повнозв'язаною*, навіть якщо будуть ушкоджені k-1 каналів зв'язку.

Наступна властивість мережі міжз'єднань – це її *діаметр*. ***Діаметром*** графа називається найбільша *відстань* між будь-якими парами вершин. ***Відстанню*** між двома вершинами графа називається число ребер у найкоротшому шляху від одного вузла до іншого. Щоб знайти діаметр графа спочатку знаходять найкоротші шляхи між всіма парами вершин. Найбільша довжина кожного із цих шляхів і є діаметр графа. Діаметр мережі визначає найбільшу затримку при передачі пакетів від одного процесора іншому або від процесора до пам'яті, оскільки кожне пересилання через канал зв'язку займає певну кількість часу. Чим менше діаметр, тим вище продуктивність. Велике значення також має середня відстань між двома вузлами, оскільки від нього залежить середній час передачі пакета.

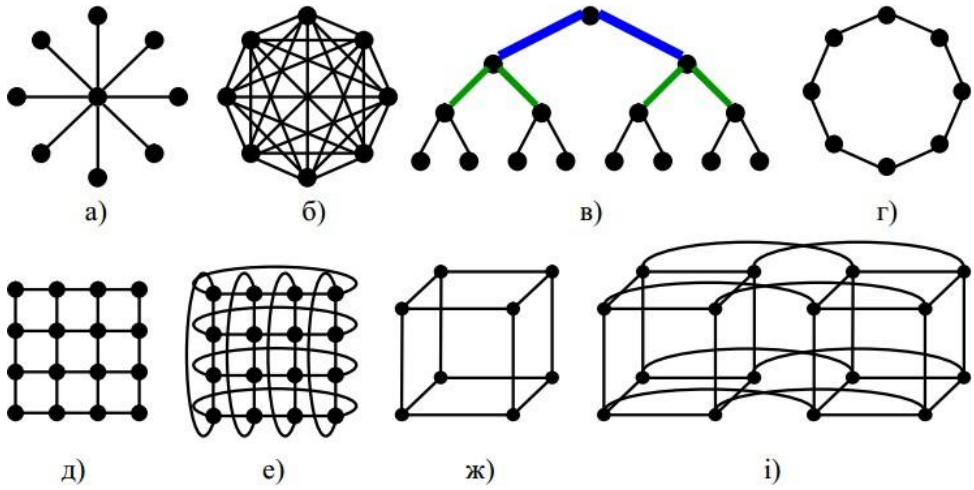


Рисунок 1.3 − Різні топології мереж міжз'єднань
вершини відповідають комутаторам мережі, дуги – каналам зв'язку
процесори й модулі пам'яті не показані
а) – топологія зірка, б) – повне міжз'єднання (full interconnect), в) – дерево,
г) – кільце, д) – ґрати, е) – подвійний тор, ж) – топологія куб, і) – гіперкуб

Ще одна важлива властивість мережі міжз'єднань – це її ***пропускна здатність***, тобто кількість даних, яке вона здатна передавати в секунду. Дуже важлива характеристика – *бісекційна пропускна здатність*. Щоб обчисли-

ти її значення, потрібно подумки поділити мережу міжз'єднань на дві рівні (з погляду числа вузлів) незв'язані частини шляхом видалення низки дуг з графа. Потім потрібно обчислити загальну пропускну здатність дуг, які були вилучені. Існує безліч способів поділу мережі міжз'єднань на дві рівні частини. ***Бісекційна пропускна здатність*** – мінімальна із всіх можливих. Якщо між двома частинами багато взаємодій, то в найгіршому разі загальна пропускна здатність не перевищує бісекційної пропускної здатності. На думку багатьох розроблювачів, бісекційна пропускна здатність – це найважливіша характеристика мережі міжз'єднань. Тому при розробці мережі прагнуть зробити бісекційну пропускну здатність максимальною.

Мережі міжз'єднань характеризуються ще і їхньою *вимірністю*. ***Вимірність*** визначається по кількості можливих варіантів переходу з вихідного пункту в пункт призначення. Якщо існує тільки один шлях з кожного вихідного пункту в кожний кінцевий пункт, то мережа визначається як ***нульвимірна***. Якщо існують два можливих варіанти напрямку пакета, то мережа визначається як ***одновимірна***. Якщо пакет може бути відправлений по двох напрямних осях, то говорять, що мережа ***двовимірна*** й т.д.

На рис. 1.3 наведено декілька найпоширеніших топологій мережі. На рисунку зображені тільки канали зв'язку – дуги та комутатори – вершини. Модулі пам'яті і процесори приєднуються до комутаторів через інтерфейси. На рис. 1.3 а) зображена найпростіша нульвимірна конфігурація *зірка*, де процесори й модулі пам'яті прикріплюються до зовнішніх вузлів, а перемикання робить центральний вузол. У великій системі з такою мережею центральний комутатор буде головним критичним елементом, що повністю обмежує продуктивність системи. З погляду стійкості до відмови це також дуже невдала розробка, оскільки збій або відмова центрального комутатора приводить до виходу з ладу всієї системи.

Рис. 1.3 б) представляє ще одну нульвимірну топологію – *повне міжз'єднання* (full interconnect). Тут кожний вузол безпосередньо пов'язаний

з кожним наявним вузлом. У такій розробці пропускна здатність між двома секціями максимальна, діаметр мінімальний, а стійкість до відмови дуже висока. На наведеному прикладі, навіть при втраті шести каналів зв'язку система однаково буде повністю взаємозалежною. Однак для k вузлів потрібно $k(k-1)/2$ каналів, а це зовсім неприйнятно для розгалужених мереж з більшою кількістю вузлів k.

На рис. 1.3 в) зображена третя нульвимірна топологія – *дерево*. Тут основна проблема полягає в тому, що пропускна здатність між секціями дорівнює пропускної здатності каналів. У такій структурі найбільший потік даних спостерігається у верхівки дерева, тому верхні вузли стають перешкодою для підвищення продуктивності. Звичайно ця проблема вирішується шляхом збільшення пропускної здатності каналів у кореня дерева. Наприклад, канали у вузлів на найбільш низькому ярусі мережі виконуються із пропускною здатністю b, вузли наступного рівня – роблять із пропускною здатністю $2b$, а кожний канал на ще більш верхньому рівні – із пропускною здатністю $4b$ і так далі. Така схема називається *товстим деревом* (fat tree).

Кільце (рис. 1.3 г) – це вже одновимірна топологія, оскільки кожний відправлений пакет може піти праворуч або ліворуч. *Ґрати* або *сітка* (рис. 1.3 д) являє собою двовимірну топологію, що застосовується в багатьох комерційних системах. Вона відрізняється регулярністю і є застосовною до систем великого розміру, а діаметр її дорівнює кореню квадратному від числа вузлів, що дозволяє проводити розширення системи при незначному збільшенні діаметра. *Подвійний тор* (рис. 1.3 е) є різновидом ґрат. Це ґрати, у яких з'єднані краї. Топологія характеризується більшою стійкістю до відмови й меншим діаметром, ніж звичайні ґрати, оскільки тепер між двома протилежними вузлами лише дві транзитних ділянки.

Топологія *куб* (рис. 1.3 ж) – це правильна тривимірна топологія. На рисунку зображений куб $2\times2\times2$, але в загальному випадку він може бути $k\times k\times k$. І нарешті, на рис. 1.3 і) показана вже чотиривимірна топологія – *гіперкуб*,

отриманий із двох тривимірних кубів, у яких всі вузли одного пов'язані з відповідними вузлами іншого. Можна зробити п'ятивимірний куб, з'єднавши разом чотири чотиривимірних куби. Щоб одержати 6 вимірів, потрібно продублювати п'ятивимірний куб і з'єднати відповідні вузли й т.д.; n-*вимірний* куб називається ***гіперкубом***. Ця топологія використовується в багатьох комп'ютерах паралельної дії, оскільки її діаметр має в лінійну залежність від вимірності. Інакше кажучи, діаметр – це логарифм з основою 2 від числа вузлів, тому 10-вимірний гіперкуб має 1024 вузлів, але діаметр дорівнює всього 10, що викликає дуже незначні затримки при передачі даних. Слід відзначити, що ґрати 32×32, які також містять 1024 вузла, мають діаметр 62, це більш ніж у шість разів перевищує діаметр гіперкуба. Однак чим менше діаметр гіперкуба, тим більше розгалуження й число каналів і, отже, вище вартість системи. Проте, у системах з високою продуктивністю найчастіше використовується саме гіперкуб. У табл. 1.2 наведені основні характеристики різних топологій для деяких мереж міжз'єднань (зірка, повне міжз'єднання, кільце, ґрати й гіперкуб). У таблицю внесені такі характеристики мережі, як її діаметр, розрахункова бісекційна пропускна здатність, стійкість до відмови. У таблиці не вказується вимірність графа, але додана характеристика, яка є досить значимою з погляду розробки – вартість мережі.

Таблиця 1.2 − Основні характеристики деяких топологій

Топологія	Діаметр	Бісекційна пропускна здатність	Стійкість до відмови	Вартість
Повне міжз'єднання	1	$p^{2/4}$	(p-1)	$p(p$-1)/2
Зірка	2	1	1	(p-1)
Кільце	$\lfloor p/2 \rfloor$	2	2	p
Гіперкуб	$\log_2 p$	$p/2$	$\log_2 p$	$p \log_2 p/2$
Ґрати	$\approx 2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$

1.1.4 Питання комунікації і маршрутизації

Як уже вказувалося, елементами мережі міжз'єднань є комутатори і канали, які їх з'єднують (шини, кабелі, проводи). Завдання комутатора полягає в прийомі пакетів,які поступають з мережі на будь-який вхідний порт, перенаправлення їх на відповідні вихідні порти і відправлення в канали мережі з цих вихідних портів.

Існує кілька стратегій перемикання, які здійснюються комутаторами при комутації потоків інформації:

1) комутація каналів;
2) комутація із проміжним зберіганням;
   - з буферуванням на вході;
   - з буферуванням на виході;
   - загальне буферування;
3) комутація без буферування пакетів;
4) «wormhole routing» (червоточина).

На рис. 1.4 наведено приклад невеликої мережі міжз'єднань із чотирма комутаторами. На рисунку кожний комутатор має по 4 вхідних і вихідних порти. Кожний комутатор може містити до декількох центральних процесорів і схеми з'єднання (на рисунку вони не показані).

Кожний вихідний порт пов'язаний із вхідним портом іншого комутатора через послідовний або паралельний канал (штрихові лінії на малюнку). Послідовні канали передають інформацію з одного біта одноразово. Паралельні канали призначені для пересилання декількох бітів відразу. Додатково канали зв'язку доповнюються спеціальними сигналами для керування каналом. Паралельні канали характеризуються більш високою продуктивністю, ніж послідовні канали з такою ж тактовою частотою, але в них виникає проблема ***розфазіровки даних*** – явище, коли окремі біти можуть при розповсюдженні у каналі випереджати або відставати від інших. Явище обумовлюється фізичними законами й геометрією каналу зв'язку, воно з трудом піддається

виправленню, і тому паралельні лінії зв’язку коштують набагато дорожче.

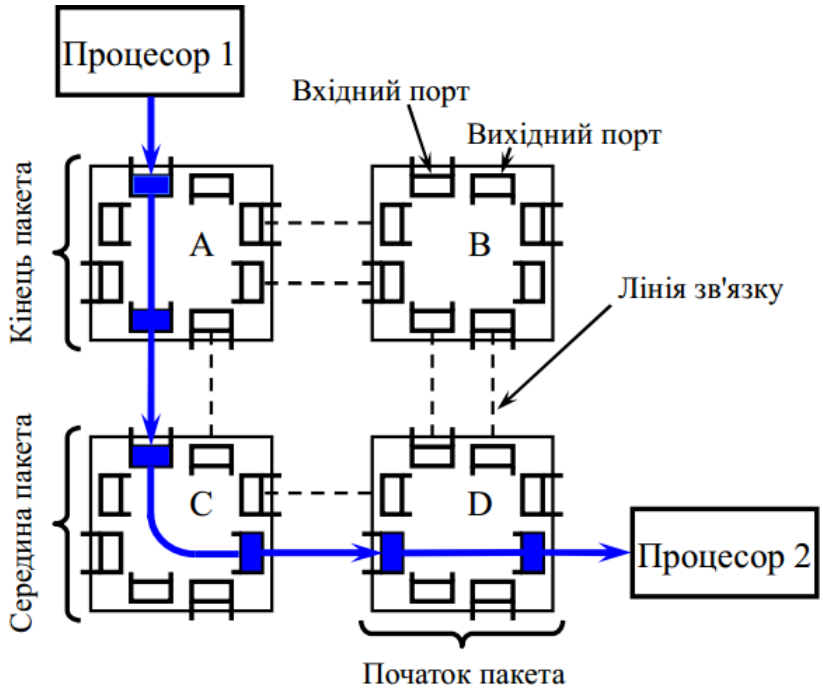


Рисунок 1.4 − Приклад ділянки мережі типу ґрати із чотирма комутаторами

На прикладі рисунка можна розглянути стратегії перемикання в таких мережах міжз’єднань. Перша з названих стратегій – ***комутація каналів***. По цій стратегії попередньо перед посилкою пакета, весь шлях від початкового пункту до кінцевого резервується заздалегідь. Із цією метою спочатку посилається спеціальний пакет невеликого розміру, він то й резервує всі вхідні і вихідні порти від джерела до приймача. Оскільки всі порти й буфери викликані заздалегідь, то в процесі передачі основного пакета він може на повній швидкості без яких-небудь колізій переміщатися від вихідного пункту через всі комутатори до пункту призначення. На рис. 1.4 показано комутацію каналів, де резервується канал від процесора 1 до процесора 2 через комутатори *A*, *C* і *D* (лінія зі стрілками). Тут резервуються три вхідних і три вихідних порти.

Друга стратегія – ***комутація із проміжним зберіганням***. У цій стратегії комутації не потрібно попереднього резервування елементів каналу передачі пакета. Весь пакет цілком посилається з вихідного пункту до першого

комутатора, де він може зберігатися до моменту звільнення необхідних ресурсів мережі. Так у розглянутому прикладі процесор 1 відправляє весь пакет процесору 2. Досягши спочатку комутатора *A* пакет, при необхідності, зберігається всередині нього. Потім цей пакет переміщається в комутатор *C* і, нарешті, весь пакет цілком переміщається в комутатор *D*. На останньому етапі пакет доходить до пункту призначення – до процесора 2. Треба ще раз особливо підкреслити, що *ніякого попереднього резервування* ресурсів *не потрібно*.

Комутатори із проміжним зберіганням повинні мати деякий буфер для зберігання отриманих пакетів, оскільки, в той час коли пункт відправник посилає пакет, потрібний йому вихідний порт комутатора може бути зайнятий передачею іншого пакета. Якби не було буферування, то пакети, які надходять до комутатора і вимагають зайнятого у цей момент вихідного порту, пропадали б. Тому, як це було зазначено вище, ця технологія комутації має три модифікації, які відрізняються методом буферування пакетів. При ***буферуванні на вході*** один або кілька буферів зв'язуються з кожним вхідним портом у формі звичайної черги типу FIFO. Якщо пакет на початку черги не можна передати через зайнятість потрібного вихідного порту, цей пакет просто чекає своєї черги.

Однак якщо пакет очікує, коли звільниться вихідний порт, то пакет, що йде за ним, теж не може передаватися, навіть якщо потрібний йому вихідний порт вільний. Така ситуація називається ***блокуванням початку черги***.

Описана проблема усувається за допомогою методу ***буферування на виході***. У цій моделі буфери зв'язуються з вихідними, а не із вхідними портами. Інформація пакета, при надходженні і при необхідності бути збереженою, накопичується в буфері, який пов'язаний з потрібним вихідним портом. Блокування початку черги не відбувається тому, що при накопиченні першого пакета на одному з вихідних портів, наступний пакет спрямовується на потрібний йому вільний вихідний порт.

Однак, і при буферуванні на вході, і при буферуванні на виході з кожним портом зв'язана певна кількість буферів і, якщо обсяг буферів є недоста-

тнім для зберігання всіх пакетів, то деякі пакети доведеться відкидати. Щоб розв'язати цю проблему, використовують ***загальне буферування***, при якому один буферний пул динамічно розподіляється по портах у міру необхідності. Однак така схема вимагає більш складного керування, щоб стежити за буферами, і дозволяє одному зайнятому з'єднанню захопити всі буфери, залишивши інші з'єднання ні з чим. Крім того, кожний комутатор повинен уміщати найбільший пакет і, навіть, кілька пакетів максимального розміру, а для цього буде потрібно посилити вимоги до пам'яті й знизити максимальний розмір пакета.

Хоча метод комутації із проміжним зберіганням досить гнучкий і ефективний, виникає проблема зростаючої затримки при передачі даних по мережі міжз'єднань. Явище пов'язане з тим, що якщо для переміщення пакета по одній транзитній ділянці потрібен час *t* нс, то для переміщення пакета від процесора 1 до процесору 2 потрібно скопіювати його 4 рази (в A, в C, в D і в процесор 2). Але наступна передача не може початися, поки не закінчиться попередня, тому сумарна затримка по мережі становить 4*t*.

Вихід із цієї ситуації, можна знайти застосовуючи наступну модель комутації – ***комутацію без буферування пакетів***. Така комутація має місце в якійсь гібридній мережі міжз'єднань, яка поєднує в собі комутацію каналів і комутацію пакетів. У цьому випадку, наприклад, кожний пакет можна розділити на невеликі частини. Як тільки перша частина надходить у комутатор, її можна відразу направити в наступний комутатор, навіть якщо частини пакета, яки залишилися, ще не прибули до цього комутатора.

Такий підхід відрізняється від комутації каналів тим, що ресурси заздалегідь не резервуються. Однак в цьому разі, можлива конфліктна ситуація в змаганні за право володіння ресурсами (портами і буферами). При *комутації без буферування* пакетів, якщо перший блок пакета не може рухатися далі то частина пакета, яка залишилася, продовжує надходити в комутатор. У найгіршому разі ця схема перетворюється в комутацію із проміжним зберіганням.

Ще один метод комутації «***wormhole routing***» (***червоточина***), загалом кажучи, являє собою різновид комутації без буферування. При цьому типі

комутації, якщо перший блок не може рухатися далі, у вихідний пункт передається сигнал зупинити передачу, і пакет обривається, будучи розтягнутим на два й більше комутатори. Це явище відображене на рис. 1.4 коли початок пакета перебуває на комутаторі D, середина на комутаторі C і кінець розташовується в комутаторі A. При звільненні необхідних ресурсів, з'являється можливість подальшого просування пакета. У мережі великого діаметра це явище може виникнути неодноразово.

Крім вирішення проблем з комутацією пакетів у мережі міжз'єднань з вимірністю один і вище необхідно вирішити проблему ***маршрутизації*** – вибору шляху, по якому виконувати передачу пакетів від одного вузла до іншого. Ця проблема виникає тому, що в таких мережах існує безліч можливих маршрутів передачі інформації. Правило, що визначає, яку послідовність вузлів повинен минути пакет при прямуванні від вихідного пункту до пункту призначення, називається ***алгоритмом вибору маршруту***.

Гарні алгоритми вибору маршруту є доконче необхідними, оскільки часто вільними виявляються декілька шляхів. Гарний алгоритм допомагає рівномірно розподілити навантаження по каналах зв'язку, щоб повністю використати існуючу в наявності пропускну здатність. Крім того, алгоритм вибору маршруту допомагає уникати тупикових ситуацій у мережі міжз'єднань. Взаємне блокування виникає в тому випадку, якщо при одночасній передачі декількох пакетів ресурси є викликаними таким чином, що жоден з пакетів не може просуватися далі й всі вони блокуються навічно. Виникає так звана ***тупикова ситуація***.

Схематично приклад *тупикової ситуації* в мережі з комутацією каналів наведений на рис. 1.5. Тупикова ситуація може виникати й у мережі з комутацією типу буферування пакетів, але графічно це явище легше представити в мережі з комутацією каналів. Тут кожний процесор намагається послати пакет процесору, що перебуває напроти нього по діагоналі (P1 $\rightarrow$ P3, P2 $\rightarrow$ P4, P3 $\rightarrow$ P1, P4 $\rightarrow$ P2). Кожний з них зміг зарезервувати вхідний і вихідний порти свого найближчого комутатора (P1 – A, P2 – C, P3 – D і P4 – B), а також один вхідний порт наступного комутатора. Але він змушений зупинитися й

очікувати на необхідному йому вихідному порту другого комутатора, наприклад, P1 на вихідному порту комутатора C, оскільки порт уже зайнятий під потреби процесора P2. Очікування триває до моменту звільнення цього порту. Якщо всі чотири процесори починають цей процес комутації одночасно, то всі вони блокуються на вихідному порту другого від себе комутатора й у результаті мережа зависає.

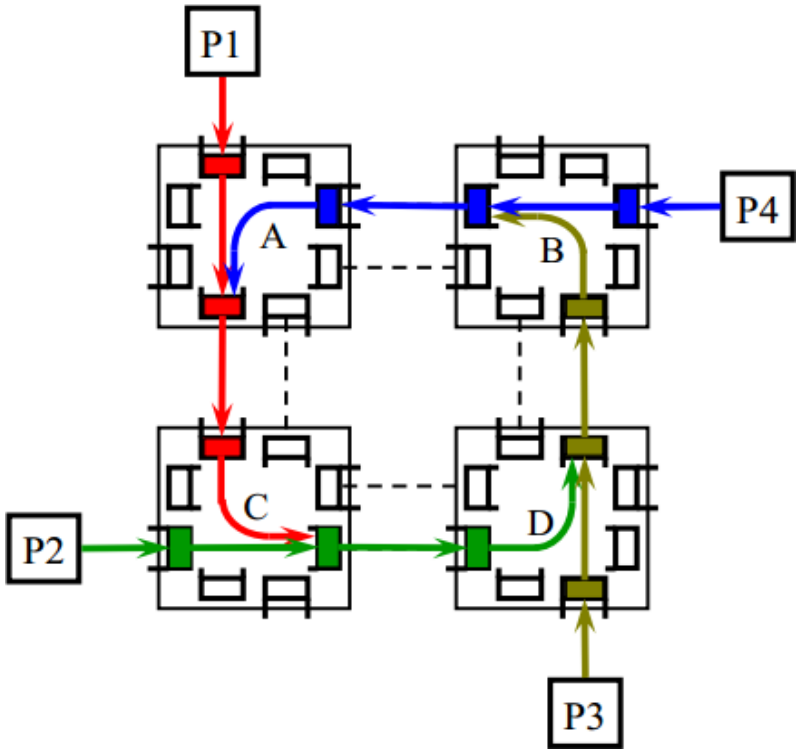


Рисунок 1.5 − Тупикова ситуація в мережі з комутацією каналів

Алгоритми вибору маршруту можна розділити на дві категорії: *маршрутизація від джерела* та *розподілена маршрутизація*. При ***маршрутизації від джерела*** джерело визначає весь шлях по мережі заздалегідь. Цей шлях виражається списком з номерів портів, які потрібно буде використовувати в кожному комутаторі на шляху до пункту призначення. Якщо шлях проходить через *k* комутаторів, то перші *k* байтів у кожному пакеті будуть містити *k* номерів вихідних портів, по одному байту на кожний порт. Коли пакет доходить до першого комутатора, то перший байт пакета відсікається й використовується для визначення номера вихідного порту. Частина пакета, яка залишила-

ся, потім направляється у відповідний порт. Після кожної транзитної ділянки пакет стає на 1 байт коротше, показуючи новий номер порту, якій потрібно вибрати наступного разу.

При ***розподіленій маршрутизації*** кожний комутатор сам вирішує, у який порт відправити той або інший прихожий пакет. Якщо ніякий з наявних маршрутів не має переваги перед іншими й вибір однаковий для кожного пакета, спрямованого до того ж самого кінцевого пункту, то маршрутизація є ***статичною***. Якщо ж комутатор при виборі маршруту бере до уваги, наприклад, трафік, що тече, то маршрутизація є ***адаптивною***.

Популярним алгоритмом маршрутизації, що застосовується для прямокутних ґрат з будь-яким числом вимірів і в якому ніколи не виникає тупикових ситуацій, є ***просторова маршрутизація***. Відповідно до цього алгоритму пакет спочатку переміщається уздовж осі *X* до потрібної координати, а потім уздовж осі *Y* до потрібної координати й т.д. (залежно від кількості вимірів). На рис. 1.6 наведений приклад просторової маршрутизації для куба 5×5×5. Для доставки пакета із точки (1,1,1) у точку (3,2,4), пакет спочатку повинен перемістится в точку (3,1,1) по осі *X* через точку (2,1,1), потім у точку (3,2,1) по осі *Y* і, у завершення, по осі *Z* через точки (3,2,2), (3,2,3) у кінцеву точку (3,2,4), як це показано на рисунку. Такий алгоритм запобігає тупиковим ситуаціям.

Винятковою метою створення комп'ютера паралельної дії є *підвищення продуктивності* у порівнянні з однопроцесорним комп'ютером для виконання складних обчислювальних робіт у різних галузях науки й техніки за розумний проміжок часу. Якщо ця мета не досягнута, то ніякого сенсу в побудові комп'ютера паралельної дії немає. Понад усе, ця мета повинна бути досягнута при найменших витратах. Паралельний комп'ютер, що працює у два рази швидше, ніж однопроцесорний, але коштує, наприклад, в 50 разів дорожче, не буде користуватися особливим попитом. Тому питання продуктивності, пов'язані зі створенням архітектур паралельних комп'ютерів перебувають у центрі уваги розроблювачів таких систем.

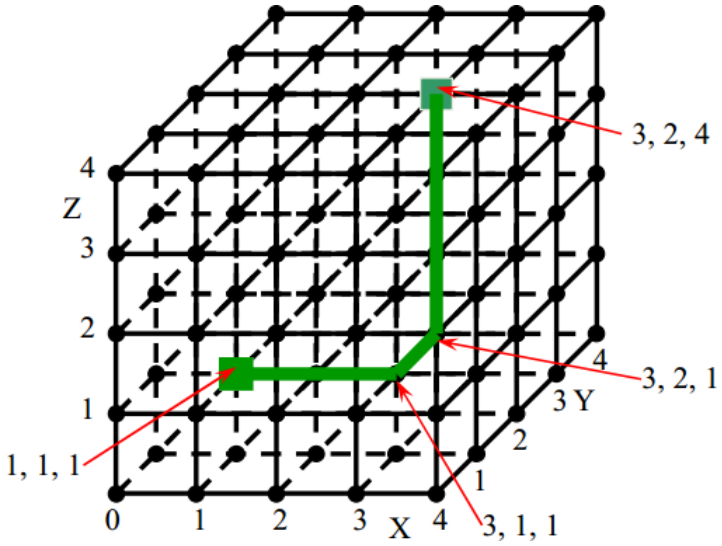


Рисунок 1.6 − Приклад просторової маршрутизації на кубі

1.1.5 Апаратна і програмна метрики паралельних комп’ютерів

В апаратному забезпеченні найбільший вплив на продуктивність паралельних комп’ютерів оказує швидкість роботи процесорів, пристроїв введення/виведення і мережі. Швидкість роботи процесорів і пристроїв введення/виведення така ж сама, як і в однопроцесорній машині, тому ключовими параметрами у паралельній системі є ті, які пов’язані з мережами міжз’єднань. Таким чином, ***метрика апаратного забезпечення*** як міра, яка дозволяє одержати чисельне значення збільшення продуктивності системи, в основному визначається властивостями комутуючої мережі. Серед параметрів, що характеризують властивості мережі з погляду швидкодії, виділяються такі ключові параметри, як *час очікування* й *пропускна здатність мережі*.

Повний час очікування – це час, який є потрібним на те, щоб процесор відправив пакет інформації і одержав відповідь. Якщо пакет посилається у пам’ять, то час очікування – це час, що потрібно на читання й запис слова або блоку слів. Якщо пакет посилається іншому процесору, то час очікування – це час, який є потрібним на міжпроцесорний зв’язок для пакетів даного розміру. Безперечно, інтерес становить час очікування для пакетів мінімального розміру (як правило, для одного слова або невеликого рядка кеш-пам’яті).

Час очікування складається з декількох факторів. Для мереж з комутацією каналів, мереж із проміжним зберіганням і мереж без буферування пакетів характерним є різний час очікування. Для комутації каналів час очікування становить суму часу ***установлення*** й часу передачі. Для установлення схеми потрібно вислати пробний пакет, щоб зарезервувати необхідні ресурси, а потім передати назад повідомлення про це. Після цього можна асемблювати основний пакет даних. Коли пакет готовий, його можна передавати на повній швидкості, тому якщо загальний час установлення становить T_S, розмір пакета дорівнює p біт, а пропускна здатність мережі – b біт у секунду, то час очікування при проходженні пробного пакета в одну сторону складе $T_S + p/b$. Якщо схема дуплексна й ніякого часу установлення на відповідь не потрібно, то мінімальний час очікування для передачі пакета розміром в p біт і одержання відповіді розміром в p біт становить $T_S + 2p/b$ секунд.

При пакетній комутації немає необхідності посилати пробний пакет у пункт призначення заздалегідь, але однаково потрібним є якийсь час установлення, T_A, на компонування пакета. Тут час передачі в одну сторону також становить $T_A + p/b$, але за цей час пакет доходить тільки до першого комутатора. При проходженні через сам комутатор утворюється деяка затримка, T_D, а потім відбувається перехід до наступного комутатора й т.д. Час T_D складається із часу обробки й затримки в черзі (очікування звільнення вихідного порту). Якщо є n комутаторів, то загальний час очікування в одну сторону становить $T_A + n(p/b + T_D) + p/b$, де останній доданок відбиває копіювання пакета від останнього комутатора у пункт призначення.

Час очікування в одну сторону для комутації без буферування пакетів і “червоточини” у найкращому разі буде наближатися до $T_A + p/b$, оскільки тут немає пробних пакетів для установлення схеми й немає затримки, обумовленої проміжним зберіганням. По суті, цей час початкового установлення для компонування пакета плюс час на передачу бітів. Варто було б ще додати затримку на розповсюдження сигналу, але вона звичайно є незначною.

Наступна характеристика метрики апаратного забезпечення – *пропускна здатність*. Багато програм паралельної обробки, особливо в природничих

науках, переміщують величезну кількість даних, тому число байтівів, що система здатна переміщати в секунду, має дуже велике значення для продуктивності. Існує кілька показників пропускної здатності. Один з них – пропускна здатність між двома секціями (*бісекційна пропускна здатність*) і її обмежуючий вплив на роботу мережі були розглянуті раніше. Інший показник – ***сумарна пропускна здатність*** – обчислюється шляхом підсумовування пропускної здатності всіх каналів зв'язку. Це число показує *максимальне число біт*, яке можна передати по мережі одночасно. Ще один важливий показник – ***середня пропускна здатність*** кожного процесора. Якщо кожний процесор здатний посилати в мережу, наприклад, тільки 1 Міб/с, то від мережі, що має пропускну здатністю між секціями в 100 Гіб/с, особливої користі не буде, навіть навпаки, висока вартість такої мережі зробить систему значно дорожче. Максимальна швидкість взаємодії процесорів буде обмежена вже не мережею, а тим, скільки даних може видавати кожний процесор.

На практиці наблизитися до теоретично можливої пропускної здатності дуже важко. Пропускна здатність скорочується з багатьох причин. Наприклад, кожний пакет завжди містить деякі службові сигнали й дані: це компонування, побудова заголовка, відправлення. При відправленні 1024 пакетів по 4 байти кожний ніколи не буде досягнута така ж сама пропускна здатність, що й при відправленні 1 пакета розміром відразу 4096 байтів. Але, на жаль, для досягнення малого часу очікування краще використовувати маленькі пакети, оскільки більші надовго блокують лінії зв'язку і комутатори. У результаті *виникає конфлікт* між досягненням низького часу очікування й високої пропускної здатності. Для одних прикладних завдань перше важливіше, ніж друге, для інших – навпаки. Важливо знати, що завжди можна збільшити пропускну здатність шляхом додавання й розширення каналів передачі даних, але не можна простим вкладенням матеріальних засобів знизити час очікування. Тому краще спочатку зробити час очікування якнайменше, а вже потім піклуватися про пропускну здатність.

З міркувань загального характеру випливає, що найпростіший спосіб збільшення продуктивності комп'ютера – введення надмірності функціона-

льних пристроїв (*багатопроцесорності*). У цьому випадку можна досягти прискорення процесу вирішення обчислювальної задачі при поділі застосовуваного алгоритму на інформаційно незалежні частини і організація виконання кожної частини обчислень на різних процесорах. Здавалося б, що подібний підхід дозволяє виконувати необхідні обчислення з меншими витратами часу, а можливість одержання максимально-можливого прискорення обмежується тільки числом наявних процесорів і кількістю «незалежних» частин у виконуваних обчисленнях. Однак при додаванні процесорів у систему може статися так, що зі збільшенням числа комп'ютерів продуктивність може зменшитися. Це явище значною мірою знову ж є зв'язаним із властивостями мережі міжз'єднань. Тому додавати процесори в систему потрібно таким чином, щоб при цьому не обмежувати підвищення продуктивності. Система, до якої можна додавати процесори й одержувати відповідно до цього більшу продуктивність, називається ***розширюваною***.

Існування нерозширюваних і розширюваних систем можна продемонструвати на наступному прикладі (рис. 1.7). Елементарні розрахунки показують, що комп'ютер, який містить чотири процесори, які з'єднані загальною шиною (рис. 1.7 а), *не є* розширюваною системою. Дійсно, якщо покласти, що пропускна здатність шини в такій конфігурації становить *b* Міб/с, то середня пропускна здатність на один процесор дорівнює *b*/4 Міб у секунду. Після ж розширення комп'ютера в чотири рази додаванням ще дванадцяти процесорів (рис. 1.7 б) без зміни пропускної здатності шини, середня пропускна здатність на один процесор зменшиться в чотири рази й складе *b*/16 Міб/с. Зменшення середньої пропускної здатності одного процесора й, отже, зменшення продуктивності комп'ютера в цілому, говорить про те, що система не є розширюваною.

Для комп'ютера із чотирма процесорами, але з мережею міжз'єднань типу ґрати (рис. 1.7 в) аналогічні міркування дають зовсім протилежний результат. У такій топології при додаванні нових процесорів обов'язково додаються й нові канали зв'язку, тому при розширенні системи сумарна пропускна здатність на кожний процесор не знизиться, як це було у випадку із загаль-

ною шиною. До розширення комп'ютера при пропускній здатності одного каналу *b* Міб/с, сумарна пропускна здатність на один процесор так само становила *b* Міб/с (4 процесори, 4 канали – 4/4 *b*). Після ж збільшення кількості процесорів у чотири рази (рис. 1.7 г) ця величина складе вже 24/16 *b* (16 процесорів, 24 канали), тобто зросте в півтора рази, що підтверджує властивість розширюваності системи.

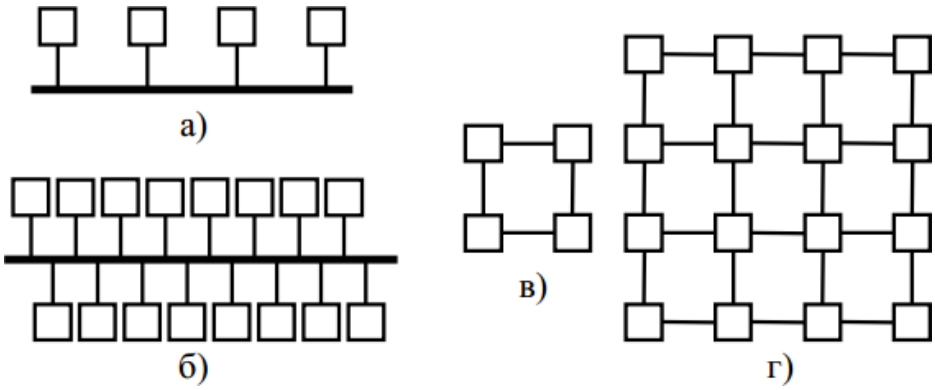


Рисунок 1.7 − Нерозширювана і розширювана топології мереж паралельного комп'ютера (квадратиками позначені процесори)
а) топологія загальна шина до розширення, б) та ж мережа після розширення
в) мережа ґрати до розширення, г) розширений комп'ютер з мережею ґрати

Але при розширенні системи негативний вплив на її продуктивність може зробити збільшення часу очікування, яке, як вказувалося вище, є прямопропорційним до діаметра мережі. При цьому додавання процесорів до мережі із шиною організацією абсолютно не впливає на час очікування при відсутності трафіку, тому що не збільшується діаметр: як дорівнювався 1 так і залишається після розширення. А от при збільшенні кількості процесорів в *k* раз для мережі на рис. 1.7 в) її діаметр зростає приблизно в $\sqrt{k}$ раз (див. табл. 1.2), тому й час очікування в найгіршому разі зросте так само. Наприклад, для мережі з 400 процесорів (20×20) діаметр дорівнює 38, а для комп'ютера з 1600 (40×40) процесорами – 78, тобто при збільшенні числа процесорів в 4 рази діаметр, а отже, і середній час очікування виросли при-

близно у два рази.

В ідеальному випадку розширювана система при додаванні нових процесорів *повинна зберігати ту саму середню пропускну здатність* на кожний процесор і *постійний середній час очікування*. На практиці збереження достатньої пропускної здатності на кожний процесор здійснюється доволі просто, але час очікування завжди росте зі збільшенням розміру. Найбільш оптимальною з погляду на незначне зростання часу очікування є топологія гіперкуба. У такій мережі час очікування росте логарифмічно зі збільшенням кількості процесорів.

Турбота розроблювачів про діаметр мережі пояснюється тим, що великий час очікування часто є неприпустимим для продуктивності систем при роботі з дрібномодульними додатками і з додатками із середнім розміром модуля. У таких програмах, коли виникає необхідність у даних, яких зараз немає в локальній пам'яті, на їхнє одержання витрачується істотна кількість часу, і чим більше система, тим більшою виходить затримка. Ця проблема є характерною не тільки для мультикомп'ютерів, але й для мультипроцесорів, оскільки в обох випадках фізична пам'ять розділена на незмінні, розташовані на великій площі модулі.

Для скорочення або, принаймні, приховання часу очікування існують і застосовуються кілька різних технологій. Перша технологія – це ***копіювання даних***. Якщо копії блоку даних можна зберігати в декількох місцях, то можна збільшити швидкість доступу до цих даних. Одним з можливих варіантів цієї технології є *використання кеш-пам'яті*, коли одна або кілька копій блоків даних зберігаються близько до того місця, де вони можуть знадобитися. Інший варіант – *зберігати кілька рівноправних копій*. На противагу асиметричним відносинам первинності/вторинності, які спостерігаються при використанні кеш-пам'яті, це копії з рівним статусом. У випадку, коли зберігається кілька рівноправних копій, Виникає безліч проблем з їхнім розміщенням. Необхідно якимось чином постійно відслідковувати, хто й коли створив ці копії й куди помістив їх. Запропоновані і реалізуються самі різні варіанти вирі-

шення проблеми – від динамічного розміщення блоків даних на вимогу апаратного забезпечення до навмисного їх розміщення у пам'яті під час завантаження директив компілятора. У всіх випадках головним питанням є ***узгодженість керування***.

Друга технологія – так звана ***випереджаюча вибірка***. Елемент даних можна викликати ще до того, як він знадобиться. Це дозволяє сховати час очікування й розділити процеси виклику даних й виконання, і коли цей елемент даних стає потрібний, він вже буде доступний. Випереджаюча вибірка може бути *автоматичною*, а може *контролюватися* програмою. У кеш-пам'ять завантажується не тільки потрібне слово, а й весь рядок кеш-пам'яті цілком, і інші слова із цього рядка теж можуть знадобитися в майбутньому. Аналогічним образом у звичайних однопроцесорних системах працюють контролери повільних дискових пристроїв введення/виведення.

Процесом випереджаючої вибірки, можна керувати і явним чином. Коли компілятор довідується, що йому будуть потрібні які-небудь дані, він може видати явну команду, щоб одержати ці дані, і видає він цю команду заздалегідь із таким розрахунком, щоб одержати потрібні дані вчасно. Така стратегія вимагає, щоб компілятор мав повні знання про машину і її синхронізацію, а також контролював, куди поміщаються всі дані. Спекулятивні команди завантаження працюють найкраще, коли абсолютно точно відомо, що ці дані будуть потрібні. Помилка через відсутність сторінки при виконанні команди для гілки програми, що в остаточному підсумку не використовується, дуже невигідна.

Третя технологія – це ***багатопоточна обробка***. У більшості сучасних систем підтримується мультипрограмування, при якому кілька процесів можуть працювати одночасно (або створювати ілюзію паралельної роботи на основі розподілу часу). Якщо переключення між процесами можна робити досить швидко, наприклад, надаючи кожному з них його власну схему розподілу пам'яті й апаратних регістрів, то коли один процес блокується й очікує прибуття даних, апаратне забезпечення може швидко переключитися на ін-

ший процес. У граничному випадку процесор виконує першу команду з потоку 1, другу команду з потоку 2 і т.д. Таким чином, процесор завжди буде зайнятий, навіть при тривалому часі очікування в окремих потоках.

Деякі машини автоматично переключаються від процесу до процесу після кожної команди, щоб сховати тривалий час очікування.

Четверта технологія – ***використання неблокуючих записів***. Звичайно при виконанні команди запису даних у пам'ять, процесор чекає, поки вона не закінчиться, і тільки після цього продовжує роботу. При наявності неблокуючих записів, починається операція з пам'яттю, але програма однаково продовжує роботу. Продовжувати роботу програми при виконанні команди завантаження трохи складніше (без гарантовано отриманих даних, не можна продовжити роботу), але навіть це є можливим, якщо застосовувати виконання зі зміною послідовності.

В 1967 році Джин Амдал (Gene Amdahl) – фахівець із архітектури комп'ютерів сформулював правило, що зв'язує властивості апаратного забезпечення паралельного комп'ютера з його продуктивністю. Згідно із цим правилом *продуктивність* обчислювальної системи, що складається зі зв'язаних між собою пристроїв, у загальному випадку *визначається самим непродуктивним* її *пристроєм*. Пізніше в літературі, присвяченій питанням архітектури комп'ютерів, деякі автори це правило стали називати ***першим законом Амдала***. Цей закон справедливий для будь-яких складових суперкомп'ютерів: мереж міжз'єднань, процесорів, модулів пам'яті...

Розглянута метрика апаратного забезпечення й перший закон Амдала показує, на що здатне апаратне забезпечення. Але користувачів зовсім не цікавить, як улаштований і як працює паралельний комп'ютер, які пристрої входять у його склад, за рахунок яких інженерних рішень досягається збільшення його продуктивності. Для користувачів важливо зовсім інше – наскільки швидше вони одержать результати роботи своєї програми на комп'ютері паралельної дії в порівнянні з однопроцесорним комп'ютером. Таким чином, у ***метриці програмного забезпечення*** на перший план виходить такий показ-

ник як ***коефіцієнт прискорення***, що показує наскільки швидше працює програма в *n*-процесорній системі в порівнянні з однопроцесорною системою. Результати звичайно ілюструються графіком залежності збільшення швидкості обчислень від кількості процесів у системі (рис. 1.8). Наведені графіки демонструють, як поводяться різні паралельні програми, які працюють на мультикомп'ютері, що складається з 64 процесорів Pentium Pro. Сімейство кривих на графіку показує, як зростає швидкість роботи тієї або іншої програми залежно від кількості процесорів *k*, які включені у роботу. Для порівняння, на графіку суцільною похилою прямою показане ідеальне, прямопропорційне до кількості процесорів *k*, підвищення швидкості роботи програми.

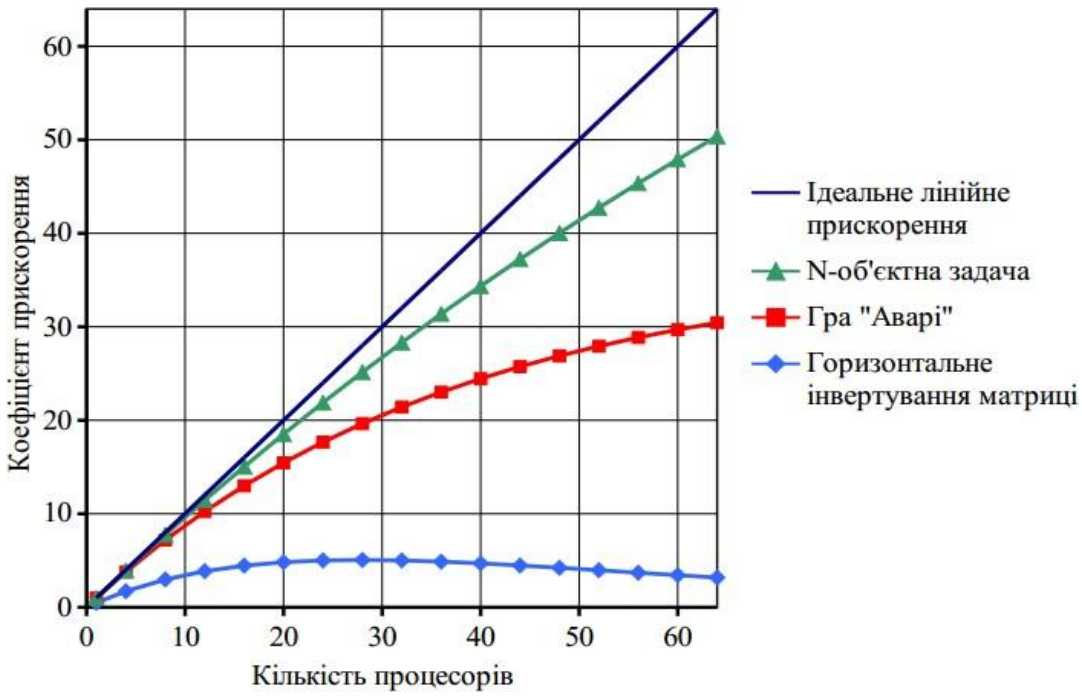


Рисунок 1.8 − Залежність коефіцієнта прискорення від кількості процесів у системі для різних програм

З низки різних причин (недосконаліть алгоритму, невідповідність архітектури комп'ютера розв'язуваному завданню…) тільки дуже і дуже незначна кількість реальних програм можуть показати майже ідеальний збіг із цією прямопропорційною залежністю. Але є досить велика кількість програм,

які досягають значного підвищення швидкості і демонструють досить задовільні результати по наближенню до ідеалу. Прикладом такої програми може слугувати N-об’єктна задача, швидкість роботи якої стрімко збільшується при додаванні нових процесорів. Програма «Аварі» (африканська логічна гра, у дечому подібна до гри в нарди) прискорюється цілком задовільно, але вже спостерігається значне відхилення коефіцієнта прискорення від лінійного закону. А от задача інвертування матриць, яка доволі часто зустрічається в практиці моделювання різних фізичних процесів, не може бути прискореною більш ніж у п’ять разів, і навіть зі збільшенням кількості процесорів коефіцієнт прискорення починає зменшуватися.

Є низка причин, по яких практично неможливо досягти ідеального підвищення швидкості, але однієї з основних є те, що всі програми містять деяку принципово послідовну частину, вони часто мають фазу ініціалізації, вони звичайно повинні зчитувати необхідні дані й збирати результати роботи процесорів. Велика кількість процесорів тут абсолютно не допомагає.

Крім закону, який визначає зв’язок властивостей апаратного забезпечення із продуктивністю паралельного комп’ютера, Джин Амдал сформулював ще один дуже важливий закон (***другий закон Амдала***). Цей закон вже визначає залежність прискорення виконання програми на суперкомп’ютері від властивостей самої програми. У законі стверджується наступне.

Якщо система складається з однакових простих універсальних пристроїв і при виконанні паралельної частини алгоритму всі пристрої завантажені повністю, то *максимально можливе прискорення* $S(n)$ *дорівнює*:

$$S(n) = \frac{1}{\square + \frac{1-\square}{n}}, \qquad (1.1)$$

де n – кількість процесорів у системі;

β – частка послідовної частини програми.

Вираз (1.1) може бути отриманий шляхом наступних міркувань.

Якщо для одержання результату в програмі необхідно виконати кількість операцій N, причому деяка частина операцій із цієї загальної кількості – $N_{ПОСЛ}$ може бути виконана винятково в послідовному режимі. Очевидно, що кількість операцій, які остаються для виконання вже у паралельному режимі складає, $N_{ПАР} = N - N_{ПОСЛ}$. Якщо одна операція виконується за час t, то на однопроцесорній машині програма буде виконана за час $T_{ОП} = Nt$. На багатопроцесорному комп'ютері з n процесорами час виконання програми $T_{МП}$ буде складатися із двох частин: часу виконання, який відповідає послідовній частині – $T_{ПОСЛ} = N_{ПОСЛ}t$ і часу виконання для паралельної частини (вона буде виконуватися відразу на всіх процесорах) – $T_{ПАР} = N_{ПАР}t/n = (N - N_{ПОСЛ})t/n$. Тим самим загальний час роботи програми вже буде дорівнюватися величині $T_{МП} = T_{ПОСЛ} + T_{ПАР} = (N_{ПОСЛ} + (N - N_{ПОСЛ})/n)t$. А коефіцієнт прискорення:

$$S(n) = \frac{T_{ОП}}{T_{МП}} = \frac{Nt}{\left(N_{ПОСЛ} + \frac{N - N_{ПОСЛ}}{n}\right)t}.$$

Після скорочення чисельника й знаменника останнього виразу на Nt і позначення $N_{ПОСЛ}/N = \beta$, одержується вираз для другого закону Амдала (1.1).

Очевидно, що у випадку, коли $\beta = 0$ (у програмі немає послідовної частини) цілком можливо одержати лінійне підвищення швидкості, але для $\beta > 0$ ідеальне підвищення швидкості неможливо.

Дане зауваження характеризує одну із самих серйозних проблем в галузі паралельного програмування (алгоритмів без певної частки послідовних команд практично не існує). Однак часто частка послідовних дій характеризує не можливість паралельного вирішення задач, а послідовні властивості застосовуваних алгоритмів. Як результат, частка послідовних обчислень може бути істотно знижена при виборі більш відповідних для розпаралелювання алгоритмів.

З отриманої залежності коефіцієнта прискорення від кількості процесо-

рів у системі випливає слідство, яке іноді називають ***третім законом Амдала***. Цей закон стверджує, що якщо система складається із простих однакових універсальних пристроїв, то *при будь-якому режимі роботи* її *прискорення не може перевершити зворотної величини до частки послідовних обчислень*.

Дійсно, якщо у виразі (1.1) спрямувати n до нескінченності то виходить: $\lim_{n\to\infty} S(n) = 1/\square$. На рис. 1.9 представлено залежності коефіцієнта прискорення від кількості процесорів при різних долях послідовної частини в програмах.

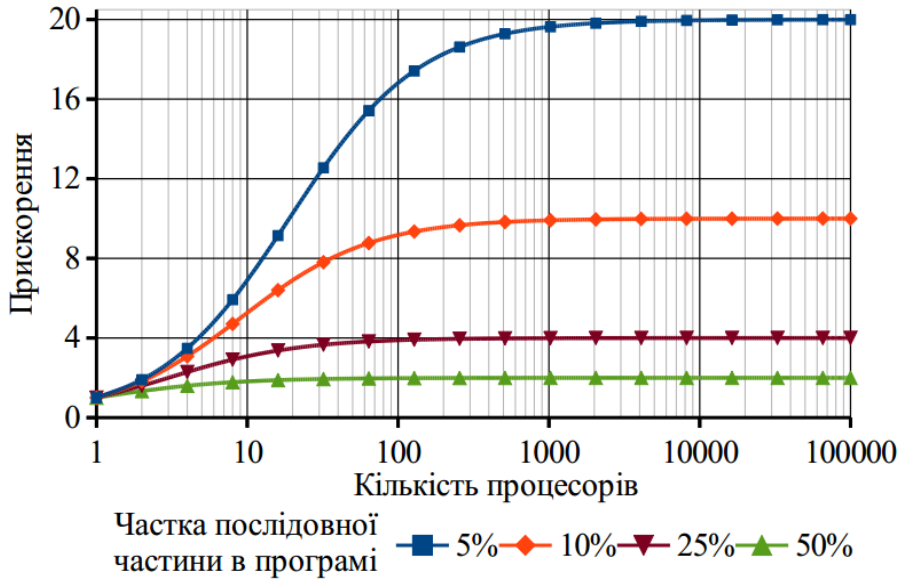


Рисунок 1.9 − Ілюстрація третього закону Амдала

Наявність послідовної частини в програмі не є єдиною причиною, яка унеможливлює ідеальне підвищення швидкості. Певну роль у цьому грає й час очікування в комунікаціях, обмежена пропускна здатність мережі, і недоліки алгоритмів. Тому вводиться уточнений, так званий, ***мережний закон Амдала***, що враховує пересилання даних по мережі при виконанні програми:

$$S(n) = \frac{1}{\square + \frac{1-\square}{n} + \square},$$

де γ – частка часу, що витрачена на пересилання даних відносно до повного часу роботи програми.

1.1.6 Принципи розробки програмного забезпечення паралельних систем

Таким чином, навіть при наявності досить великої кількості процесорів, не всі програми можна написати так, щоб ефективно використовувати всі ці процесори, непродуктивні витрати для запуску їх всіх можуть бути дуже значними. Крім того, багато відомих алгоритмів важко піддати паралельній обробці, тому дуже часто доводиться використовувати який-небудь субоптимальний алгоритм. Для багатьох прикладних задач виявляється цілком прийнятним одержати прискорення в n разів, навіть якщо для цього буде потрібним $2n$ або навіть більше процесорів.

Оскільки функціонування суперкомп'ютерів (як і будь-яких комп'ютерів) без належного системного програмного забезпечення (ПЗ) неможливо має сенс розглянути основні принципи організації такого ПЗ. Тим більше, що розроблювачі апаратного забезпечення повинні враховувати особливості програмного забезпечення і погоджувати з ним архітектурні особливості систем, які розробляються.

Існує чотири основних підходи до розробки програмного забезпечення для паралельних комп'ютерів. Перший підхід – *додавання спеціальних бібліотек чисельного аналізу* до звичайних послідовних мов програмування. Наприклад, бібліотечна процедура для інвертування великої матриці або для розв'язання низки диференціальних рівнянь з частинними похідними може бути викликана з послідовної програми, після чого вона буде виконуватися на паралельному процесорі, а програміст навіть не буде знати про існування паралелізму. Недолік цього підходу полягає в тому, що паралелізм може застосовуватися тільки в декількох процедурах, а основна частина програми залишиться послідовною.

Другий підхід – *додавання спеціальних бібліотек*, що містять примітиви комунікації й керування. Тут програміст сам створює процес паралелізму і управляє їм, використовуючи додаткові примітиви.

Наступний крок – *додавання декількох спеціальних конструкцій до існуючих мов програмування*, які, наприклад, дозволяють легко породжувати нові паралельні процеси, виконувати повторення циклу паралельно або виконувати арифметичні дії над всіма елементами вектора одночасно. Цей підхід широко використовується, і на теперішній час існує дуже багато мов програмування до яких були включені елементи паралелізму.

Четвертий підхід – *створення зовсім нової мови* спеціально для паралельної обробки. Очевидна перевага такої мови полягає в тім, що вона дуже добре підходить для паралельного програмування на якому-небудь *певному* суперкомп’ютері. Але недолік її в тому, що програмісти повинні вивчати нову мову. Більшість нових паралельних мов імперативні (їхні команди змінюють змінні станів), але деякі з них функціональні, логічні або об’єктно-орієнтовані.

Існує безліч бібліотек, розширень мов і нових мов, які створені спеціально для паралельного програмування, і вони дають найширший спектр можливостей, тому їх дуже важко класифікувати. Але у загальному випадку можна виділити п’ять ключових парадигм, які формують основу програмного забезпечення для комп’ютерів паралельної дії:

- моделі керування;
- ступінь розпаралелювання процесів;
- обчислювальні парадигми;
- методи комунікації;
- базисні елементи синхронізації.

Моделі керування – одна з найважливіших серед названих парадигм. Відповідно до цієї парадигми визначається, скільки буде потоків керування в роботі ПЗ – один або декілька. Розрізняють дві моделі керування. У *першій моделі* існує одна програма й один лічильник команд, але кілька наборів да-

них. Кожна команда виконується над всіма наборами даних одночасно різними обробними елементами.

Така модель програмування має дуже велике значення для апаратного забезпечення. По суті, це значить, що кожний обробний елемент – це АЛП і пам'ять, без схеми декодування команд. Замість цього один центральний блок викликає команди й повідомляє всім АЛП, що робити далі.

Альтернативна модель передбачає кілька потоків керування, кожний з яких має власний лічильник команд, регістри і локальні змінні. Кожний потік керування виконує свою власну програму над своїми даними, при цьому він час від часу може взаємодіяти з іншими потоками керування. Існує безліч варіацій цієї ідеї, і в сукупності вони формують основну модель для паралельної обробки. Власне кажучи, ця модель використовувалася коли в першому модулі курсу розглядалися приклади різних паралельних програм.

Парадигма ***ступеня розпаралелюваниня процесів*** характеризує рівні паралелізму керування. На найнижчому рівні *елементи паралелізму містяться в окремих машинних командах*. Програмісти звичайно не знають про існування такого паралелізму – він управляється компілятором або апаратним забезпеченням.

На *більш високому рівні* відбувається перехід до *паралелізму на рівні блоків*, що дозволяє програмістам самим контролювати, які висловлювання будуть виконуватися послідовно, а які – паралельно.

Присутнім є паралелізм на рівні ще більш *великих структурних одиниць*, коли можна викликати процедуру й не змушувати програму, яка її викликала, чекати завершення цієї процедури. Це означає, що програма, яка викликала процедуру, і сама викликана процедура будуть працювати паралельно. Якщо програма, яка викликає процедуру, здійснює це у циклі при кожному його проходженні і не чекає завершення викликаних процедур, то створюється велика кількість паралельних процедур, які працюють одночасно.

Інша форма паралелізму – *створення або породження для кожного процесу декількох потоків*, кожний з яких працює в межах адресного просто-

ру цього процесу. Кожний потік має свій лічильник команд, свої регістри й стік, але розділяє весь інший адресний простір і всі глобальні змінні з іншими потоками. Потоки працюють незалежно друг від друга, можливо навіть на різних процесорах. В одних системах операційна система має повну інформацію про всі потоки й здійснює планування потоків. В інших системах кожний користувальницький процес сам виконує планування потоків і управляє потоками, а операційній системі про це невідомо.

Нарешті, паралелізм на *рівні ще більш великих структурних одиниць* – це кілька незалежних процесів, які разом працюють над вирішенням спільної задачі. На відміну від потоків незалежні процеси не розділяють загальний адресний простір, тому задача повинна бути розділена на досить великі частини, по одній на кожний процес.

Обчислювальні парадигми застосовуються для *структуризації роботи* паралельних програм, які містять велику кількість потоків або незалежних процесів. Існує безліч таких парадигм, деякі з них, які, мабуть, можна визначити як основні, представлені на рис. 1.1.

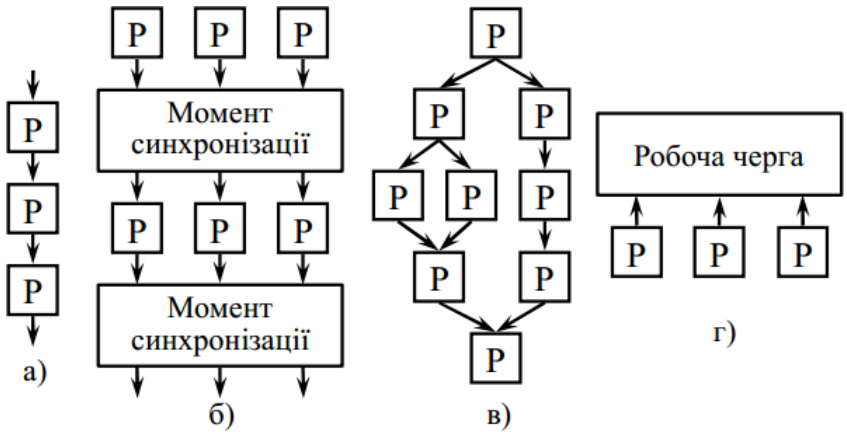


Рисунок 1.10 − Логічна структура деяких обчислювальних парадигм
а) конвеєр процесів, б) фазоване обчислення,
в) «розділяй і володарюй», г) завдання в пулі потоків. (Р – процеси)

По-перше, *парадигма SPMD* (Single Program Multiple Data – одна про-

грама, кілька потоків даних, на рис. Рисунок 1.10 не наведена). У цьому випадку, хоча система й складається з декількох незалежних процесів, вони всі виконують ту саму програму, але над різними наборами даних. Тільки зараз, на відміну від моделей керування, всі процеси виконують ті самі обчислення, але кожний у своєму просторі.

По-друге – *конвеєр процесів* (рис. 1.10 а). Дані надходять у перший процес, що трансформує їх і передає другому процесу для читання й т.д. Якщо потік даних довгий (наприклад, якщо це відеозображення), всі процесори можуть бути зайняті одночасно. Так працюють конвеєри в системі UNIX. Вони можуть працювати як окремі процеси паралельно на мультикомп'ютері або мультипроцесорі.

Наступна парадигма – *фазоване обчислення* (рис. 1.10 б), коли робота поділяється на фази, наприклад, повторення циклу. Під час кожної фази кілька процесів працюють паралельно, але якщо один із процесів закінчить свою роботу, він повинен чекати доти, поки всі інші процеси не завершать свою роботу, і тільки після цього починається наступна фаза.

Четверта парадигма – «*розділяй і володарюй*», зображена на рис. 1.10 в), у цій парадигмі запускається один процес, що потім породжує інші процеси й передає їм частину роботи.

Останній приклад – *завдання в пулі потоків* (replicated-worker) демонструється на рис. 1.10 г). Тут існує центральна черга, і робітники-процеси одержують завдання з цієї черги і виконують їх. Якщо завдання породжує нове завдання, воно додається до центральної черги. Щораз, коли робочий процес завершує виконання поточного завдання, він одержує з черги наступне.

Якщо програма розділена на частині, які працюють паралельно, вони повинні мати можливість взаємодіяти один з одним, для цього і призначені ***методи комунікації.*** Взаємодію можна здійснити одним із двох способів: за допомогою загальних змінних і за допомогою передачі повідомлень. У першому випадку всі процеси мають *доступ до загальній логічній пам'яті* й взаємодіють, зчитуючи й записуючи інформацію в цю пам'ять. Наприклад, один

процес може встановити змінну, а інший процес може прочитати її.

У мультипроцесорі змінні можуть розділятися між декількома процесами за допомогою відображення однієї й тієї ж сторінки віртуальної пам'яті в адресний простір кожного процесу. Потім загальні змінні можна зчитувати й записувати за допомогою звичайних машинних команд читання/запису. Навіть у мультикомп'ютері без розподіленої фізичної пам'яті можливим є логічний розподіл змінних, але зробити це трохи складніше. Існує також можливість розподілу одного адресного простору на мультикомп'ютері, а також розбиття на сторінки у мережі. Таким чином, процеси можуть взаємодіяти між собою через загальні змінні, як у мультипроцесорах, так і в мультикомп'ютерах.

Альтернативний підхід – взаємодія через *передачу повідомлень*. У даній моделі використовуються примітиви посилки/прийому повідомлення. Один з процесів виконує посилку повідомлення, оголошуючи інший процес як пункт призначення. Як тільки другий процес виконує прийом, повідомлення копіюється в адресний простір одержувача.

Існує безліч варіантів передачі повідомлень, але всі вони зводяться до застосування двох примітивів посилки й прийому. Вони звичайно реалізуються як системні виклики. Наступне питання при передачі повідомлень – *кількість одержувачів*. Найпростіший випадок – один відправник і один одержувач (передача повідомлень типу точка-точка). Однак у деяких випадках потрібно відправити повідомлення всім процесам (широкомовлення) або деякому визначеному набору процесів (мультимовлення).

У мультипроцесорі передачу повідомлень легко реалізувати шляхом простого копіювання даних від відправника до одержувачів. Таким чином, можливості фізичної пам'яті спільного використання (мультипроцесор/мультикомп'ютер) і логічної пам'яті спільного використання (взаємодія через загальні змінні/передача повідомлень) ніяк не зв'язані між собою. Всі чотири комбінації мають сенс і можуть бути реалізованими. Вони наведені в табл. 1.3.

Таблиця 1.3 − Комбінації спільного використання фізичної і логічної пам’яті

Фізична пам’ять (апаратне забезпечення)	Логічна пам’ять (програмне забезпечення)	Приклади
Мультипроцесор	Поділювані змінні	Обробка зображень
Мультипроцесор	Передача повідомлень	Використання буферів пам’яті, каналів
Мультикомп’ютер	Поділювані змінні	DSM, Linda, Orca і т.д.
Мультикомп’ютер	Передача повідомлень	PVM або MPI

Паралельні процеси повинні не тільки взаємодіяти, але й за допомогою якихось ***базисних елементів синхронізації*** погоджувати свої дії. Якщо процеси використовують загальні змінні, потрібно бути впевненим, що поки один процес записує що-небудь у загальну структуру даних, ніякий інший процес не зчитує цю структуру. Інакше кажучи, потрібно деяка форма взаємного виключення, щоб кілька процесів не могли використовувати ті самі дані одночасно.

Існують різні базисні елементи, які можна використовувати для взаємного виключення. Це семафори, блокування, м’ютекси й критичні секції. Всі вони дозволяють процесу монопольно використовувати якийсь ресурс (загальну змінну, пристрій введення/виведення й т.п.), і при цьому ніякі інші процеси доступу до цього ресурсу не мають. Якщо отримано дозвіл на доступ, процес може використовувати цей ресурс. Якщо другий процес запитує дозвіл, а перший усе ще використовує цей ресурс, доступ не буде заборонений доти, поки перший процес не звільнить ресурс.

У багатьох паралельних програмах існує й такий тип примітивів (базисних елементів), які блокують всі процеси доти, поки не завершиться якась визначена фаза роботи (див. рис. 1.10 б). Найпоширенішим примітивом подібного роду є бар’єр. Коли процес зустрічає бар’єр, він блокується доти, поки всі процеси не підійдуть до бар’єра. Коли останній процес зустрічає бар’єр,

всі процеси одночасно звільняються й продовжують роботу.

1.1.7 Класифікація паралельних комп’ютерів

З урахуванням всіх можливостей апаратного забезпечення й задоволень всіх вимог ПЗ була запропонована й побудована безліч різних видів паралельних комп’ютерів. Тому з’явилася потреба в *класифікації комп’ютерів паралельної дій*. Створити повну, тобто таку, яка враховує весь спектр особливостей комп’ютерів паралельної дії, класифікацію практично дотепер не вдалося. Найчастіше використовується класифікація Флінна (Flynn), яка фактично є дуже грубим наближенням. У табл. 1.4 наведена ця класифікація, а на рис. 1.11 показана узагальнена структура комп’ютерів, що відповідає класифікації.

Таблиця 1.4 − Типи комп’ютерів за класифікацією Флінна

Потоки команд	Потоки даних	Назва	Приклади
1	1	SISD	Класична машина фон Неймана
1	Багато	SIMD	Векторний суперкомп′ютер, масивно-паралельний процесор
Багато	1	MISD	Не існує
Багато	Багато	MIMD	Мультипроцесор, мультикомп’ютер

В основі класифікації лежать два поняття: потоки команд і потоки даних. Потік команд відповідає лічильнику команд. Система з *n* процесорами має *n* лічильників команд і, отже, *n* потоків команд.

Потік даних складається з набору операндів. У багатопроцесорній системі може існувати безліч таких наборів – по одному на процесор.

Потоки команд і даних у деякому ступені незалежні, тому існує чотири різні комбінації (табл. 1.4). SISD (Single Instruction stream Single Data stream – один потік команд, один потік даних) – це самий звичайний класичний послідовний комп’ютер фон Неймана. Він містить один потік команд і один потік

даних і може виконувати тільки одну дію в кожний конкретний момент часу.

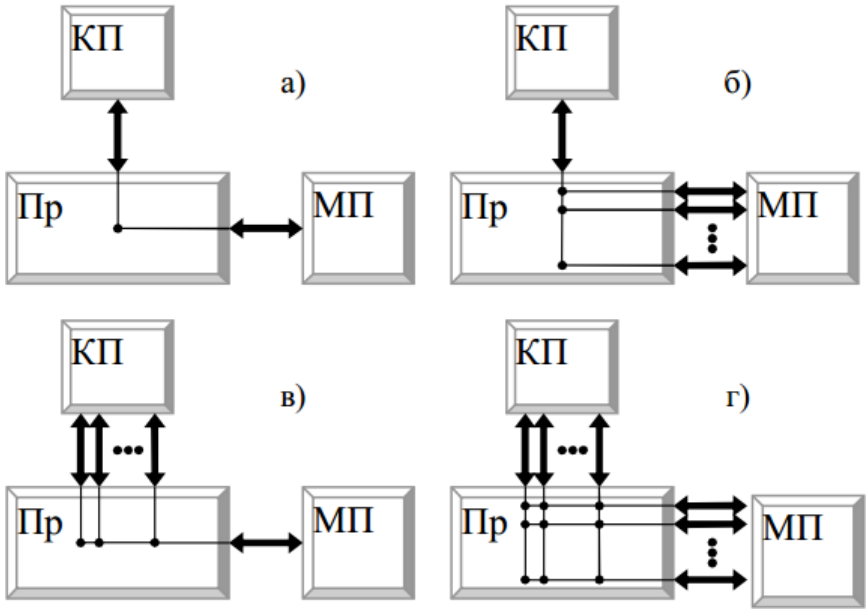


Рисунок 1.11 − Узагальнена структура комп’ютерів
за класифікацією Флінна а) SISD, б) SIMD, в) MISD, г) MIMD.
КП – керуючий пристрій, Пр – процесор, МП – модулі пам’яті

Машини SIMD (Single Instruction stream Multiple Data stream – один потік команд, кілька потоків даних) містять один блок керування, що видає по одній команді, але при цьому є декілька АЛП.

Машини MISD (Multiple Instruction stream Single Data stream – кілька потоків команд, один потік даних) – дещо дивна категорія. Тут кілька команд оперують одним набором даних. Важко сказати, чи існують такі машини. Однак дехто з авторів різних комп’ютерних видань вважає за машини типу MISD машини з конвеєрами.

Остання категорія – машини MIMD (Multiple Instruction stream Multiple Data stream – кілька потоків команд, кілька потоків даних). У цьому випадку кілька незалежних процесорів працюють як частина великої системи. У цю категорію попадає більшість паралельних процесорів. І мультипроцесори, і мультикомп’ютери – це машини MIMD.

Оскільки в цей час, коли розроблена і експлуатується безліч супер-

комп’ютерів різних архитектур, класифікація Флінна не охоплює всього різноманіття систем. Більше докладна, заснована на класифікації Флінна, сучасна класифікація наведена на рис. 1.12. У цій класифікації комп’ютери SIMD поділяються на дві підгрупи. У першу підгрупу потрапляють суперкомп’ютери, які оперують векторами, виконуючи ту саму операцію над кожним елементом вектора. У другу підгрупу попадають комп’ютери у яких головний блок керування посилає команди до декількох незалежних АЛП.

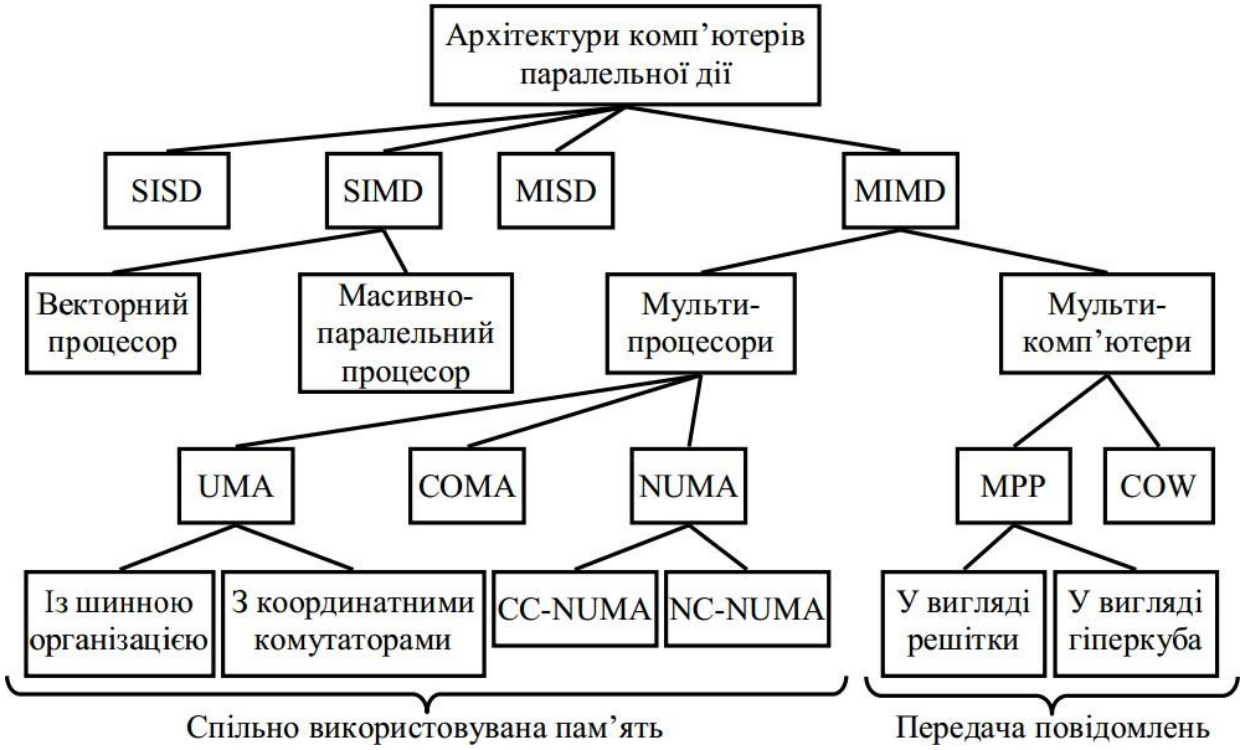


Рисунок 1.12 − Уточнена класифікація комп’ютерів паралельної дії

Комп’ютери категорії MIMD підрозділяються на мультипроцесори (системи з пам’яттю спільного використання) і мультикомп’ютери (системи з передачею повідомлень). Існує три типи мультипроцесорів. Вони відрізняються друг від друга способом реалізації пам’яті спільного використання. Вони називаються UMA (Uniform Memory Access – архітектура з однорідним доступом до пам’яті), NUMA (NonUniform Memory Access – архітектура з неоднорідним доступом до пам’яті) і COMA (Cache Only Memory Access – архітектура з доступом тільки до кеш-пам’яті). У комп’ютерах типу UMA кожний

процесор має той самий час доступу до будь-якого модуля пам’яті. Іншими словами, кожне слово пам’яті можна зчитати з тією же швидкістю, що й будь-яке інше слово пам’яті. Якщо це технічно неможливо, найшвидші звертання до пам’яті вповільнюються, щоб відповідати самим повільним, тому програмісти не побачать ніякої різниці. Це й означає ***однорідний доступ***. Така однорідність робить продуктивність передбачуваною, а цей фактор є дуже важливим для написання ефективної програми.

Мультипроцесор категорії NUMA, навпроти, не має цієї властивості. Звичайно є такі модулі пам’яті, які розташовані ближче до того чи іншого процесора, і доступ до цих модулів пам’яті відбувається набагато швидше, ніж до інших – ***неоднорідний доступ***. З погляду продуктивності, для таких систем є дуже важливим де розміщуються програма й дані. Машини COMA теж з неоднорідним доступом, але з іншої причини. Докладніше кожний із цих трьох типів будуть розглянуті пізніше.

У другу підкатегорію систем MIMD потрапляють мультикомп’ютери, які на відміну від мультипроцесорів не мають пам’яті спільного використання на архітектурному рівні. Інакше кажучи, операційна система в процесорі мультикомп’ютера не може одержати доступ до пам’яті, яка відноситься до іншого процесора, просто шляхом виконання команди читання з пам’яті. Йому доводиться відправляти повідомлення й чекати відповіді. Саме здатність операційної системи зчитувати слово з віддаленого модуля пам’яті за допомогою звичайної команди читання і відрізняє мультипроцесори від мультикомп’ютерів. Як було сказано вище, навіть у мультикомп’ютері користувальницькі програми можуть звертатися до інших модулів пам’яті за допомогою команд читання/запису, але цю ілюзію створює операційна система, а не апаратне забезпечення. Різниця незначна, але дуже важлива. Тому що мультикомп’ютери не мають прямого доступу до віддалених модулів пам’яті, вони іноді називаються системами NORMA (NO Remote Memory Access – без доступу до віддалених модулів пам’яті).

Мультикомп’ютері можна розділити на дві категорії. Перша категорія

містить багато спеціальних процесорів, які мають власні модулі пам'яті. Це, так звані, системи MPP (Massively Parallel Processors – процесори з масовим паралелізмом) – дорогі суперкомп'ютери, які складаються з великої кількості процесорів, зв'язаних високошвидкісною комунікаційною мережею.

Друга категорія мультикомп'ютерів включає звичайні робочі станції, які зв'язуються за допомогою вже наявної межовій технології з'єднання. Ці досить прості машини називаються NOW (Network of Workstations – мережа робочих станцій) або COW (Cluster of Workstations – кластер робочих станцій).

З метою розуміння відмінностей у структурах і принципах дії систем SIMD і MIMD, які ввійшли в класифікацію, має сенс розглянути ключові конструктивні особливості цих розробок.

1.1.8 Матричні і конвеєрні паралельні системи

Комп'ютери SIMD (Single Instruction Stream Multiple Data Stream – один потік команд, кілька потоків даних) використовуються для розв'язання наукових і технічних задач із векторами й масивами. Такий комп'ютер містить один блок керування, що виконує команди по одній, але кожна команда оперує відразу декількома елементами даних. Два основних типи комп'ютерів SIMD – це *масивно-паралельні процесори* (array processors) і *векторні процесори* (vector processors).

Сама ідея ***масивно-паралельних процесорів*** (деякі джерела називають такі системи ***матричні процесори***) була вперше запропонована приблизно піввіку тому (60-ти роки минулого століття). Але пройшло десь близько десяти років до моменту створення першого екземпляру з такою архітектурою. З тих пір різними компаніями було розроблено й створено кілька типів комерційних масивно-паралельних процесорів, але у сьогоденні ці комп'ютери не дуже популярні.

У масивно-паралельному процесорі міститься один блок керування, що передає сигнали, щоб запустити кілька обробних елементів, як це показано на

рис. 1.13. Кожний обробний елемент складається із процесора або вдосконаленого АЛП і, як правило, невеликої локальної пам’яті.

Хоча всі масивно-паралельні процесори відповідають цій загальній моделі, вони можуть відрізнятися друг від друга в деяких моментах.

Рисунок 1.13 − Типова структура масивно-паралельного (матричного) процесора

По-перше, відмінність може бути в *структурі обробного елемента*. Вона може бути різної – від надзвичайно простій до надзвичайно складній. Найпростіші обробні елементи – 1-бітні АЛП. У такій машині кожний АЛП одержує два 1-бітних операнди зі своєї локальної пам’яті плюс біт зі слова стану програми (наприклад, біт переносу). Результат операції – 1 біт даних і декілька бітів прапорців. Щоб зробити додавання двох цілих 32-бітних чисел, блоку керування потрібно транслювати команду 1-бітного додавання 32 рази. Якщо на одну команду витрачається, наприклад, 600 нс, то для додавання цілих чисел буде потрібно 19,2 мкс, тобто виходить повільніше, ніж у перших IBM PC. Але при наявності 65536 обробних елементів можна одержати більше трьох мільярдів додавань у секунду при часі додавання 300 пікосекунд.

З іншого боку обробним елементом може бути 8-бітний АЛП, 32-бітний АЛП або ще більш потужний пристрій, здатний виконувати операції з рухомою крапкою. У деякому ступені, вибір типу обробного елемента залежить від цілей застосування машини. Операції із рухомою крапкою можуть знадобиться для складних математичних розрахунків (хоча при цьому істотно скоротиться число обробних елементів), але, наприклад, для цілей інформаційного пошуку вони не потрібні.

По-друге, відрізнятися можуть схеми *зв'язків обробних елементів* один до одного. Тут застосовуються практично всі топології, які були розглянуті раніше (рис. 1.3). Найчастіше використовуються прямокутні ґрати, оскільки вони підходять для завдань із матрицями й відображеннями і добре застосовні до більших розмірів, тому що з додаванням нових процесорів автоматично збільшується пропускна здатність.

По-третє, розходження можуть торкнутися *локальної автономії*, який можуть володіти обробні елементи. Блок керування повідомляє, яку команду потрібно виконати, але кожний обробний елемент, на основі деяких локальних даних, наприклад, на основі бітів коду умови процесор, може вибирати виконувати йому цю команду або ні. Така особливість надає процесору значну гнучкість.

Другий тип машини SIMD – ***векторний процесор***. Він більш популярний на ринку. Комп'ютери такого типу домінували в області наукових досліджень протягом десятиліть. Популярність цих комп'ютерів обумовлена тим, що в програмах для наукових розрахунків і модельних експериментів постійно потрібна швидка обробка великих обсягів даних, при цьому присутня безліч таких виразів, як:

```
for(i = 0; i < n; i++) a[i] = b[i] + c[i];
```

де *a*, *b* і *c* – це вектори, причому найбільше часто це числа із плаваючою крапкою. Цикл наказує комп'ютеру скласти *i*-ті елементи векторів *b* і *c*, і зберег-

ти результат в *i*-му елементі масиву *a*. У циклі елементи складаються послідовно один за однім, але послідовність має значення тільки у випадку великої різниці в порядках чисел, коли можлива втрата точності обчислень через округлення результатів розрахунків, в більшості ж випадків послідовність не має ніякого значення.

На рис. 1.14 схематично зображено векторний АЛП. Такий АЛП на вході одержує два *n*-елементних вектори й обробляє відповідні елементи паралельно, оперуючи всіма елементами одночасно. У результаті утворюється новий вектор. Вхідні й вихідні вектори можуть зберігатися в пам’яті або в спеціальних векторних регістрах.

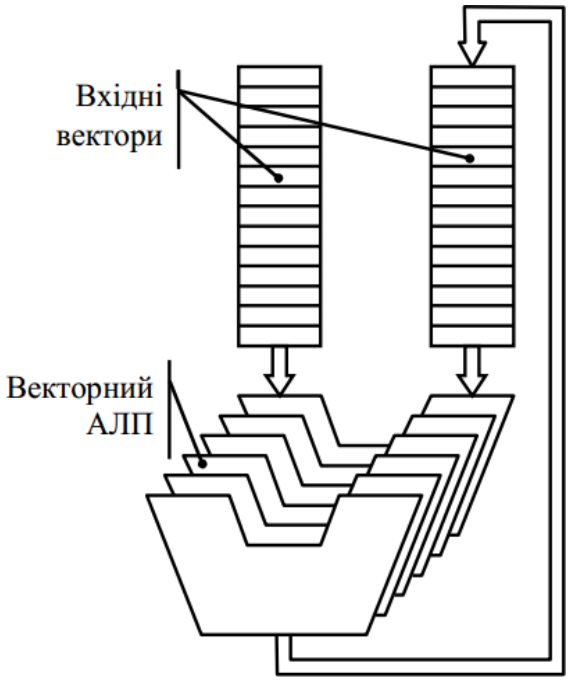


Рисунок 1.14 − Векторний АЛП

Застосування векторних комп’ютерів не обмежується тільки векторними операціями, вони застосовуються й для скалярних і для змішаних векторно-скалярних операцій. Основні типи векторних операцій можливих у такому АЛП наведені в табл.1.5. Перша з них – f_1 виконує ту або іншу операцію, наприклад, обчислення квадратного кореня або значення косинуса над кожним елементом одного вектора. Друга – f_2, на вході одержує вектор, а на виході

видає скалярне значення. Типовий приклад – підсумовування всіх елементів масиву. Третя – f_3, виконує бінарну операцію над двома векторами, наприклад додавання відповідних елементів. Нарешті, четверта – f_4, з'єднує скалярний операнд із векторним. Типовим прикладом може служити множення кожного елемента вектора на константу. Іноді швидше переробити скалярний операнд у вектор, кожне значення якого дорівнює скалярному операнду, а потім виконати операцію над двома векторами.

Таблиця 1.5 − Комбінації векторних і скалярних операцій

Операція	Приклади
$A_i = f_1(B_i)$	Обчислення косинуса, квадратного кореня, …
Скаляр = $f_2(A)$	Підрахунок суми елементів, знаходження мінімуму
$A_i = f_3(B_i, C_i)$	Додавання, вирахування векторів
$A_i = f_4$(скаляр, B_i)	Множення елементів B_i на константу

Практично всі звичайні операції з векторами можуть провадитися з використанням цих чотирьох основних форм. Наприклад, щоб одержати скалярний добуток двох векторів, потрібно спочатку перемножити відповідні елементи векторів – операція f_3, а потім скласти отримані результати – f_2.

Хоча суперкомп'ютери за такою схемою й користуються популярністю, але економічно вони дуже не вигідні. Створення комп'ютера з 64 високошвидкісними АЛП обходиться занадто дорого.

Частіше на практиці векторні процесори сполучаються з ***конвеєрами***. Операції із рухомою крапкою досить складні. Вони вимагають виконання декількох кроків, а для виконання будь-якої багатокрокової операції краще використовувати конвеєр.

У табл. 1.6 наведена послідовність операцій при вирахуванні числа $9{,}212\times10^{11}$ з $1{,}082\times10^{12}$. Звичайно числа з рухомою крапкою представляються в так званій *нормалізованій формі запису*. Нормалізоване число повинне мати мантису, яка більша або дорівнює одиниці, але менша за 10. У прикладі обид-

ва вихідні операнди представлені у нормалізованій формі і результат теж повинен бути нормалізований.

Таблиця 1.6 − Вирахування чисел із плаваючою крапкою

Номер кроку	Назва кроку	Значення	
1	Виклик операндов	$1{,}082\times10^{12}$;	$9{,}212\times10^{11}$
2	Вирівнювання експоненти	$1{,}082\times10^{12}$;	$0{,}9212\times10^{12}$
3	Вирахування	$0{,}1608\times10^{12}$	
4	Нормалізація результату	$1{,}608\times10^{11}$	

Щоб від одного числа з рухомою крапкою відняти інше число з рухомою крапкою, спочатку потрібно перетворити їх так, щоб їхні експоненти мали те саме значення. У прикладі можна або перетворити зменшуване в $10{,}82\times10^{11}$, або перетворити від'ємник у $0{,}9212\times10^{12}$. Загалом кажучи, будь-яке подібне перетворення досить ризикована операція. Збільшення експоненти може призвести до зникнення значущих розрядів мантиси, а зменшення експоненти може викликати переповнення мантиси. Зникнення значущих розрядів мантиси менш небезпечно, оскільки в цьому випадку число можна округлити нулем. Тому частіше вибирається зменшення експоненти. Привівши обидві експоненти до 12, отримуються значення, які показані в табл. 1.6 на кроці 2. Потім виконується вирахування й на закінчення виконується нормалізація результату.

Описану технологію конвеєризації можна застосовувати до циклу *for*, наведеному вище. Табл. 1.7 пояснює роботу конвеєризованого суматора з рухомою крапкою. У кожному циклі на першій стадії викликається пара операндів. На другій стадії менша експонента перетворюється так, щоб відповідати більшій. На третій стадії виконується операція підсумовування, а на четвертій стадії нормалізується результат. Таким чином, у циклі, починаючи з четвертої ітерації, на виході конвеєра одержується один готовий результат операції.

Таблиця 1.7 − Пояснення роботи конвеєризованого суматора з рухомою крапкою

Крок	Цикл						
	1	2	3	4	5	6	7
Виклик операндів	B_1C_1	B_2C_2	B_3C_3	B_4C_4	B_5C_5	B_6C_6	B_7C_7
Вирівнювання експоненти		B_1C_1	B_2C_2	B_3C_3	B_4C_4	B_5C_5	B_6C_6
Вирахування			B_1–C_1	B_2–C_2	B_3–C_3	B_4–C_4	B_5–C_5
Нормалізація результату				B_1–C_1	B_2–C_2	B_3–C_3	B_4–C_4

Істотне розходження між використанням конвеєра для операцій над векторами й використанням його для виконання звичайних команд – відсутність переходів при роботі з векторами. Кожний цикл використовується повністю, і ніяких порожніх циклів немає.

1.1.9 Мультипроцесорні системи, їх архітектура і принципи дії

Як виявляється з рис. 1.12, ***системи MIMD*** підрозділяються на дві основні категорії: мультипроцесори й мультикомп'ютери. Як уже вказувалася раніше *мультипроцесор* – це комп'ютерна система, що містить кілька процесорів і один адресний простір, видимий для всіх процесорів. На мультипроцесорі запускається одна копія операційної системи з одним набором таблиць, у тому числі таблицями, що стежать за тим які сторінки пам'яті зайняті, а які вільні. Коли процес блокується, його процесор зберігає свій стан у таблицях операційної системи, а потім переглядає ці таблиці для знаходження іншого процесу, який потрібно запустити. Саме *наявність одного відображення й відрізняє* мультипроцесор від мультикомп'ютера.

Мультипроцесори, як і всі комп'ютери, повинні містити пристрої введення/виведення (диски, мережні адаптери й т.п.). В одних мультипроцесорах тільки деякі визначені процесори мають доступ до пристроїв введення/виведення, в інших системах кожний процесор має доступ до будь-якого

пристрою. Якщо всі процесори мають однаковий доступ до всіх модулів пам'яті і всіх пристроїв введення/виведення і кожний процесор є взаємозамінним з іншими процесорами, то така ***система*** називається ***SMP*** (Symmetric Multiprocessor – ***симетричний мультипроцесор***). Оскільки SMP системи найбільш поширені, то надалі розглядається тільки саме такий тип систем.

Незважаючи на те, що у всіх мультипроцесорах процесорам надається відображення загального використовуваного адресного простору, часто поряд із цим є безліч модулів пам'яті, кожний з яких містить яку-небудь частину фізичної пам'яті. Процесори й модулі пам'яті з'єднуються за допомогою *складній комунікаційній мережі*. Кілька процесорів можуть намагатися зчитати слово з пам'яті, а в цей же самий час кілька інших процесорів можуть намагатися записати те ж саме слово, і деякі повідомлення можуть бути доставлені не в тім порядку, у якому вони були відправлені. Якщо ще врахувати існування численних копій деяких блоків пам'яті (наприклад, у кеш-пам'яті) і, якщо не вжити певних заходів, то в системі створюється абсолютно незрозуміла хаотична ситуація. Позбутися виникаючих неоднозначностей у розташуванні даних можна тільки яким-небудь значеннєвим семантичним упорядкуванням операцій запису/читання в системі.

Семантику пам'яті можна розглядати як контракт між програмним і апаратним забезпеченням системи. Якщо програмне забезпечення підтримує певні правила, то й пам'ять має можливість видавати цілком визначені результати. Ці правила називаються ***моделями погодженості пам'яті***. Була запропонована й розроблена безліч таких моделей.

Сувора погодженість – найпростіша модель. У такій моделі при будь-якому зчитуванні з будь-якої комірки пам'яті завжди вертається значення самого останнього запису до неї. Але таку модель на практиці можна реалізувати тільки у випадку, коли є один монолітний модуль пам'яті, який обслуговує всі запити просто в міру їх надходження (першим надійшов – першим оброблений), без кеш-пам'яті й без дублювання даних. Але цей процес дуже сильно сповільнює роботу пам'яті.

Наступна модель називається ***погодженістю за послідовністю***. Тут

при наявності декількох запитів на читання й запис порядок обслуговування всіх запитів обумовлюється винятково апаратним забезпеченням, але всі процесори спостерігають ту саму послідовність запитів.

Процесорна погодженість – більше програшна модель, але зате її легше реалізувати на великих мультипроцесорах. Вона має дві властивості:

1) всі процесори *сприймають записи* будь-якого процесора *в тім порядку, у якому вони починаються*;
2) всі процесори *бачать записи* в будь-яке слово пам'яті *в тому ж порядку, у якому вони відбуваються*.

Ці два правила дуже важливі. Перше затверджує, що якщо який-небудь процесор починає операції запису в комірки пам'яті в певному порядку, то всі інші процесори бачать ці записи тільки в тому ж порядку й ніколи жоден інший процесор не може побачити іншу послідовність. Друге правило потрібно для того, щоб кожне слово в пам'яті мало недвозначне значення після того, як процесор зробив кілька записів у це слово, а потім зупинився. Усі інші процесори повинні сприймати тільки останнє значення.

Більш складний випадок, коли не один, а, наприклад, два процесори одночасно починають операції запису. У цьому випадку інші процесори, які виконують операції зчитування, можуть спостерігати довільну послідовність дій цих двох процесорів, але так, що *ніколи не порушується послідовність у діях кожного* з них.

У наступній моделі, ***слабкої погодженості***, записи, які зроблені одним процесором, сприймаються строго один по одному. Але тут для усунення неоднозначностей у порядку спостереження записів іншими процесорами, у пам'яті містяться змінні синхронізації або операція синхронізації (апаратний аналог бар'єрної синхронізації потоків). Коли виконується синхронізація, всі незакінчені записи завершуються, і жоден новий запис не може початися, поки не будуть завершені всі старі записи й не буде зроблена синхронізація. Синхронізація приводить пам'ять у стабільний стан, коли не залишається ніяких незавершених операцій. Самі операції синхронізації погоджені за послідовністю, тобто якщо вони викликаються декількома процесорами, вибира-

ється деякий чіткий порядок, причому всі процесори сприймають той самий порядок.

При слабкій погодженості час розділяється на послідовні періоди, розмежовані моментами синхронізації. Таким чином, за допомогою операцій синхронізації програмне забезпечення може вносити порядок у послідовність подій, але це якогось зайвого часу.

Слабка погодженість – не дуже ефективний метод, оскільки він вимагає обов'язкового завершення всіх операцій пам'яті й затримує виконання нових операцій доти, поки старі не будуть завершені. При ***вільній погодженості*** справи йдуть набагато краще, оскільки тут використовується деякий апаратний *аналог критичних секцій* у ПЗ. Якщо процес виходить за межі такої критичної області, це ні яким чином не означає, що всі записи повинні негайно завершитися. Потрібно тільки, щоб всі записи були завершені до того, як будь-який процес знову ввійде в цю критичну область.

У цій моделі операція синхронізації розділяється на дві різні операції. Щоб зчитати або записати загальну змінну, програмне забезпечення процесора повинне виконати операцію одержання монопольного доступу до загальних поділюваних даних. Потім процесор може використовувати ці дані за своїм розсудом (зчитувати або записувати їх). По закінченні використання комірки пам'яті процесор виконує операцію звільнення над змінною синхронізації. Операція звільнення не вимагає завершення незакінчених записів, але сама вона не може бути завершена, поки не закінчаться всі раніше початі записи. Понад усе, нові операції пам'яті можуть починатися відразу ж.

Коли починається наступна операція монопольного захоплення ресурсу, виробляється перевірка, чи всі попередні операції звільнення завершилися. Якщо ні, то операція захоплення припиняється доти, поки не будуть зроблені всі операції звільнення. Таким чином, якщо наступна операція захоплення з'являється через досить тривалий проміжок часу після останньої операції звільнення, то їй не потрібно чекати, і вона може ввійти в критичну область без затримки. Така схема трохи складніша, ніж слабка погодженість, але вона має істотну перевагу: тут *не потрібно затримувати виконання команд так*

часто, як при слабкій погодженості.

На рис. 1.15 наведені структурні схеми мультипроцесорів архітектури ***UMA SMP із шинною організацією***.

В основі найпростіших мультипроцесорів лежить мережа міжз'єднань з топологією загальної шини, як показано на рис. 1.15 а). Два або більше процесори і один або кілька модулів пам'яті використовують для взаємодії ту саму шину. Якщо процесору потрібно зчитати слово з пам'яті, він спочатку перевіряє вільна шина чи ні. Якщо шина вільна, процесор поміщає адресу потрібного слова на шину, встановлює кілька сигналів управління й чекає, коли пам'ять помістить на шину потрібне слово.

Якщо шина зайнята, процесор просто чекає, коли вона звільниться. Із цією розробкою зв'язана одна проблема. При наявності двох або трьох процесорів доступ до загальної шини є цілком керованим, але при наявності 32 або 64 процесорів виникають труднощі. Продуктивність системи буде повністю обмежуватися пропускною здатністю шини, а значна кількість процесорів буде простоювати більшу частину часу.

Щоб розв'язати цю проблему, кожному процесору додають кеш-пам'ять, як це показано на рис. 1.15 б). Кеш-пам'ять конструктивно може перебувати усередині мікросхеми процесора, поруч із мікросхемою процесора, на платі процесора. Припустимими є й комбінації цих варіантів. Оскільки тепер зчитувати слова можна з кеш-пам'яті, завантаження загальної шини буде менше, і система зможе підтримувати більшу кількість процесорів.

Ще одна можливість – розробка, у якій кожний процесор має не тільки кеш-пам'ять, але й власну локальну пам'ять, до якої він одержує доступ через свою локальну шину (рис. 1.15 в). Щоб оптимально використовувати таку конфігурацію, компілятор повинен поміщати у локальні модулі пам'яті весь текст програми, константи, інші дані, які призначені тільки для читання, стеки й локальні змінні. Загальна поділена пам'ять використовується *тільки для загальних змінних*. У більшості випадків таке розумне розміщення сильно скорочує кількість даних, які передаються по шині, і не вимагає активного втручання з боку компілятора.

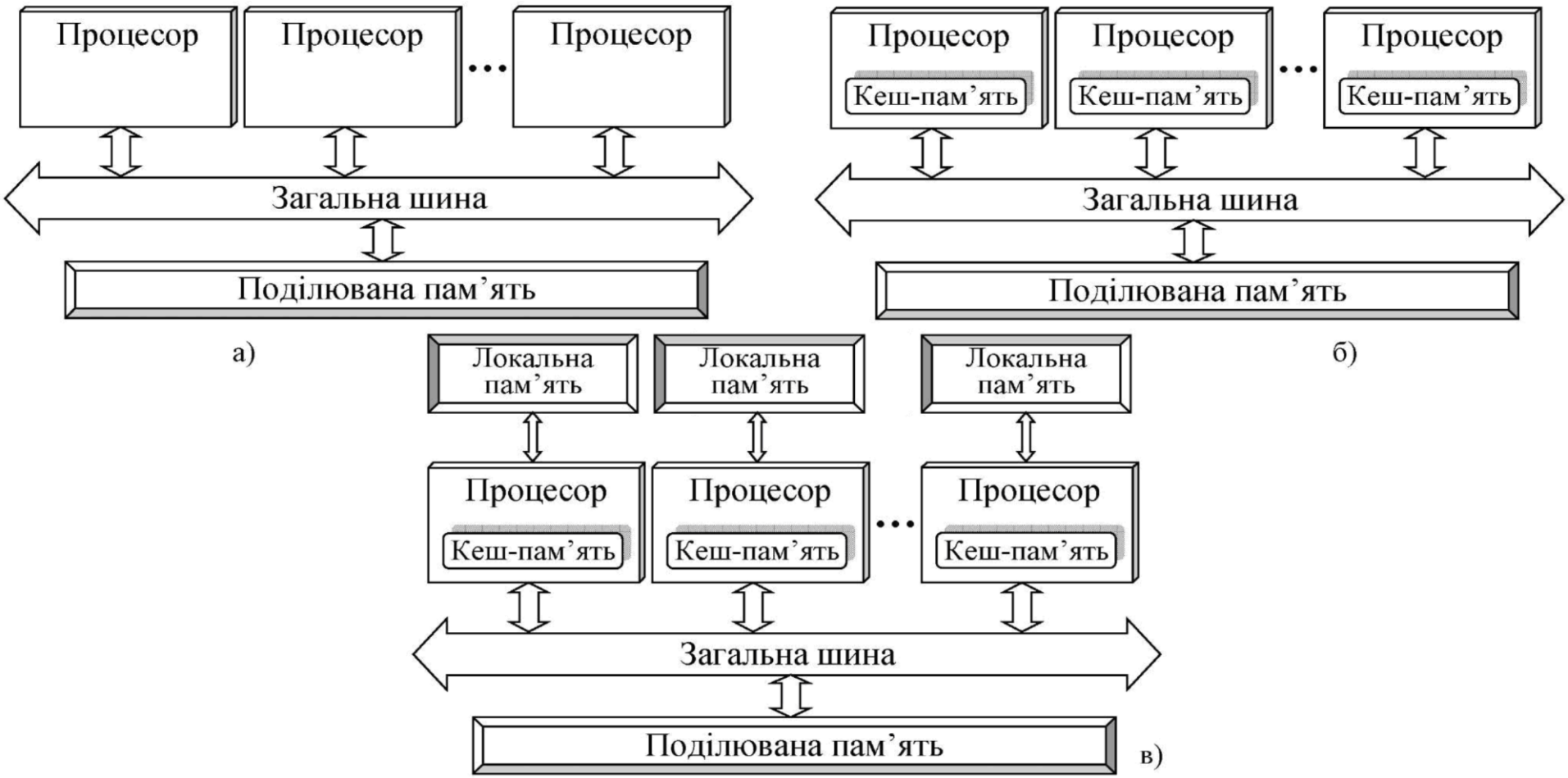


Рисунок 1.15 − Типи мультипроцесорів UMA SMP із загальною шиною

а) найпростіша конфігурація, б) конфігурація з кеш-пам’яттю,

в) повна конфігурація з кеш-пам’яттю і блоками локальної пам’яті

Якщо при наявності тільки спільно поділюваної пам’яті зміна, яка зроблена будь-яким процесором, є видимою всіма іншим процесорами системи, то включення в систему блоків кеш-пам’яті призводить до необхідності використання ***механізму відстеження змін даних у кеш-пам’яті***. Ця необхідність пов’язана з тим, що в блоках кеш-пам’яті, які належать різним процесорам, можуть одночасно зберігатися копії того самого слова з основної пам’яті. І якщо один із власників такої копії зробить її зміну, то всі інші власники копії втрачають право на її використання, оскільки копія стає недійсною.

Виникаюча проблема, яку називають ***несуперечністю кешів,*** дуже важлива. Якщо її не розв’язати, не можна буде використовувати кеш-пам’ять, і кількість процесорів, приєднаних до загальної шини, прийдеться скоротити до двох-трьох. За нашого часу запропонована безліч різних вирішень проблеми. Хоча всі алгоритми, які називають ***протоколами когерентності кешування***, різняться в деяких деталях, всі вони *не дозволяють* одночасно мати різні варіанти одного й того ж рядка в різних блоках кеш-пам’яті.

У всіх випадках контролер кеш-пам’яті розробляється таким чином, щоб кеш-пам’ять мала змогу перехоплювати запити на шині і контролювати всі звернення до неї від інших процесорів і інших блоків кеш-пам’яті й вживати відповідних заходів для вирішення виникаючих проблем. Ці пристрої називаються ***кеш-пам’яттю з відстеженням*** (snooping caches або snoopy caches), оскільки вони відслідковують шину. Набір правил, які виконуються кеш-пам’яттю, процесорами й основною пам’яттю для запобігання появі різних варіантів даних у декількох блоках кеш-пам’яті, формує ***протокол когерентності кешування***.

Найпростіший протокол когерентності кешування називається ***наскрізним кэшуванням***. При роботі контролера кеш-пам’яті з використанням цього протоколу, можливі чотири випадки, всі вони наведені в табл. 1.8. Якщо процесор намагається зчитати слово, якого немає у нього в кеш-пам’яті, контролер кеш-пам’яті завантажує рядок, який містить це слово, з основної пам’яті.

Основна пам’ять, що надала рядок, у цьому протоколі завжди є оновленою. Надалі інформація може зчитуватися з кеш-пам’яті.

Таблиця 1.8 − Можливі стани контролера кеш-пам’яті при роботіі з протоколом наскрізного кешування

Подія	Дії	
	Локальна кеш-пам’ять	Віддалена кеш-пам’ять
Промах при читанні	Виклик даних з основної пам’яті в кеш	Відслідковується, дій з пам’яттю не виробляється
Влучення при читанні	Використання даних з локальної кеш-пам’яті	Не відслідковується, ніяких дій не виконується
Промах при запису	Відновлення даних в основної пам’яті	Перевіряється наявність слова у себе, ніяких дій не виконується
Влучення при запису	Відновлення кеш-пам’яті, запис слова в основну пам’ять	Оголошення елемента свого кеш і основної пам’яті недійсним

У випадку промаху кеш-пам’яті при запису, слово, що було змінено, записується тільки в основну пам’ять. Рядок, що містить потрібне слово, не завантажується в кеш-пам’ять. У випадку результативного звертання до кеш-пам’яті при запису кеш оновляється й додатково слово записується в основну пам’ять. *Суть протоколу полягає в тому, що в результаті всіх операцій запису записуване слово обов’язково проходить через основну пам’ять, щоб інформація в основній пам’яті завжди була актуальною*.

Можливі різні варіації цього основного протоколу. Наприклад, при успішному запису кеш, який відслідковує операцію, просто оголошує недійсним рядок, що містить дане слово. З іншого боку, замість того щоб оголошувати слово недійсним, можна прийняти нове значення і оновити кеш-пам’ять. По суті, оновлення кеш-пам’ять – це те ж саме, що визнати слово недійсним, а потім зчитати потрібне слово з основній пам’яті. У всіх кеш-протоколах потрібно зробити вибір між стратегією відновлення й стратегією з визнанням даних недійсними. Ці протоколи працюють по-різному. Повідомлення про

відновлення несуть корисне навантаження, отже, вони більші за розміром, ніж повідомлення про недійсність, натомість вони можуть запобігти подальшим промахам кеш-пам’яті.

Інший варіант – завантаження кеш-пам’яті, яка відслідковує дію при промахах. Таке завантаження ніяк не впливає на правильність виконання алгоритму. Воно впливає тільки на продуктивність. Якщо ймовірність того, що щойно записане слово незабаром буде записано знову, є високою, то вибирається стратегія завантаження кеш-пам’яті при промахах запису – ***політика заповнення по запису***. Якщо ймовірність мала, краще не оновляти кеш-пам’ять у випадку промаху при записі. Якщо дане слово незабаром буде зчитуватися, воно однаково буде завантажено після промаху при зчитуванні, і нема рації завантажувати його у випадку промаху при записі.

Як і більшість простих вирішень, наскрізне кешування не дуже ефективно. Кожна операція запису повинна передаватися в основну пам’ять по шині, а при великій кількості процесорів це важко. Тому були розроблені інші протоколи. Всі вони характеризуються одною загальною властивістю: *не всі записи проходять безпосередньо через основну пам’ять*. Замість цього при зміні рядка у кеш-пам’яті усередині кеш-пам’яті встановлюється біт, що вказує, що рядок у кеш-пам’яті правильний, а в основній пам’яті – ні. У підсумку цей рядок потрібно буде записати в основну пам’ять, але перед цим у кеш-пам’ять можна зробити багато записів. Такий тип протоколу називається ***протоколом зі зворотним записом***.

Один з популярних протоколів зі зворотним записом називається ***MESI***, по перших буквах назв чотирьох станів кеш – Modified, Exclusive, Shared і Invalid. Протокол MESI використовується в процесорах, у тому числі й у процесорах Intel, для відстеження шини. Кожний елемент кеш-пам’яті може перебувати в одному з наступних чотирьох станів:

1) Invalid – елемент кеш-пам’яті містить недійсні дані;
2) Shared – кілька кешів можуть містити даний рядок, основна пам’ять оновлена;

3) Exclusive – ніякий інший кеш не містить цей рядок, основна пам’ять оновлена;
4) Modified – елемент дійсний; основна пам’ять недійсна, копій елемента не існує.

У табл. 1.9 наведений приклад взаємодії трьох процесорів, які позначені P1, P2 і P3 при роботі за протоколом MESI. У таблиці недійсні дані в кеш-пам’яті процесорів позначені як XXXX. При завантаженні системи всі елементи кеш-пам’яті позначаються як недійсні (крок 0 у табл. 1.9). При першому зчитуванні з основної пам’яті потрібний рядок викликається в кеш-пам’ять, наприклад, процесора P2 і позначається як **E** (Exclusive), оскільки це єдина копія в кеш-пам’яті (крок 1 у табл. 1.9). При наступних зчитуваннях процесор використовує цей рядок і *не використовує шину*. Процесор P2 може викликати той же рядок і помістити його у свою кеш-пам’ять (крок 3), але при відстеженні вихідний власник рядка – P1 довідається, що він вже не одноосібний власник, і повідомить, що в нього є копія. Обидві копії позначаються станом **S** (Shared). При наступних читаннях рядків кеш-пам’яті у стані **S** процесор не використовує шину й *не змінює стан* елемента.

Якщо ж який-небудь процесор, наприклад, P2 спробує зробити запис у рядок кеш-пам’яті, що перебуває в стані **S**, то він поміщає сигнал про недійсність на шину повідомляючи що потрібно відкинути всі копії. Відповідний рядок переходить у стан **M** (Modified), а інші в стан **I** (Invalid) – це крок 4. Рядок *не записується* в основну пам’ять. Якщо ж записуваний рядок перебуває в стані **E**, то ніяких змін не виконується, оскільки інших копій у системі немає.

Якщо у одержаному стані процесор P3 спробує зчитати цей же рядок (крок 5), то P2 передасть на шину сигнал, щоб процесор 3 почекав. Потім P2 записує рядок назад у пам’ять (крок 6) і тим самим дозволяє P3 викликати з пам’яті копію рядка й позначити обидві копії як **S** (крок 7). Якщо на кроці 8 P2 знову зробить запис у рядок, то система знову перейде в стан **IMI.**

Таблиця 1.9 − Взаємодія трьох процесорів (P1, P2, P3) з використанням протоколу MESI

Крок	Дія	Дані вкеш-пам’яті процесора			Дані в	Стан		
		P1	P2	P3	пам’яті	P1	P2	P3
0	Вихідний стан	XXXX	XXXX	XXXX	0xA5A5	I	I	I
1	P1 читає слово з пам’яті у свою кеш-пам’ять	0xA5A5	XXXX	XXXX	0xA5A5	E	I	I
2	P1 читає слово зі своєї кеш-пам’яті	0xA5A5	XXXX	XXXX	0xA5A5	E	I	I
3	P2 читає слово з пам’яті у свою кеш-пам’ять	0xA5A5	0xA5A5	XXXX	0xA5A5	S	S	I
4	P2 пише слово у свою кеш-пам’ять	0xA5A5	0xD2D2	XXXX	0xA5A5	I	M	I
5	P3 намагається читати слово з пам’яті (заборона на	0xA5A5	0xD2D2	XXXX	0xA5A5	I	M	I
6	P2 зберігає свій кэш в основній пам’яті	0xA5A5	0xD2D2	XXXX	0xD2D2	I	M	I
7	P3 одержує дозвіл на читання й читає слово	0xA5A5	0xD2D2	0xD2D2	0xD2D2	I	S	S
8	P2 робить запис у свою кеш-пам’ять	0xA5A5	0xB4B4	0xD2D2	0xD2D2	I	M	I
9	P1 намагається записати слово в кэш (заборона на за-	0xA5A5	0xB4B4	0xD2D2	0xD2D2	I	M	I
10	P2 зберігає свій кэш у пам’яті	0xA5A5	0xB4B4	0xD2D2	0xB4B4	I	M	I
11	P1 одержав дозвіл на запис у кэш і робить її	0xC3C3	0xB4B4	0xD2D2	0xB4B4	M	I	I

Врешті на кроці 9 тепер і P1 намагається зробити запис у слово в цьому ж рядку. P2 знову, як і раніше забороняє таку дію й зберігає свій кэш в основній пам'яті (крок 10 у табл. 1.9), даючи дозвіл P1 виконати запис. У такому стані, оскільки запис виробляється у фактично некешований рядок, можливі два варіанти запису. Якщо в системі застосовується політика *write-allocate*, рядок буде записаний в кеш-пам'ять і система перейде в стан **MII** (крок 11 у таблиці). Якщо політика *write-allocate* не застосовується, то запис буде зроблений безпосередньо в основну пам'ять, а система перейде у вихідний стан **III**, аналогічно кроку 0 у таблиці.

Навіть при всіх можливих оптимізаціях *використання тільки однієї шини обмежує розмір мультипроцесора* UMA до 16 максимум до 32 процесорів. Щоб одержати більший розмір, потрібно мати інший тип комунікаційної мережі. Найпростіша схема з'єднання *n* процесорів з *k* блоками пам'яті – використання ***координатного комутатора*** (рис. 1.16). Координатні комутатори використовуються протягом багатьох десятиліть (свій початок вони беруть від звичайних телефонних комутаторів) для з'єднання групи вхідних ліній з рядом вихідних ліній довільним образом.

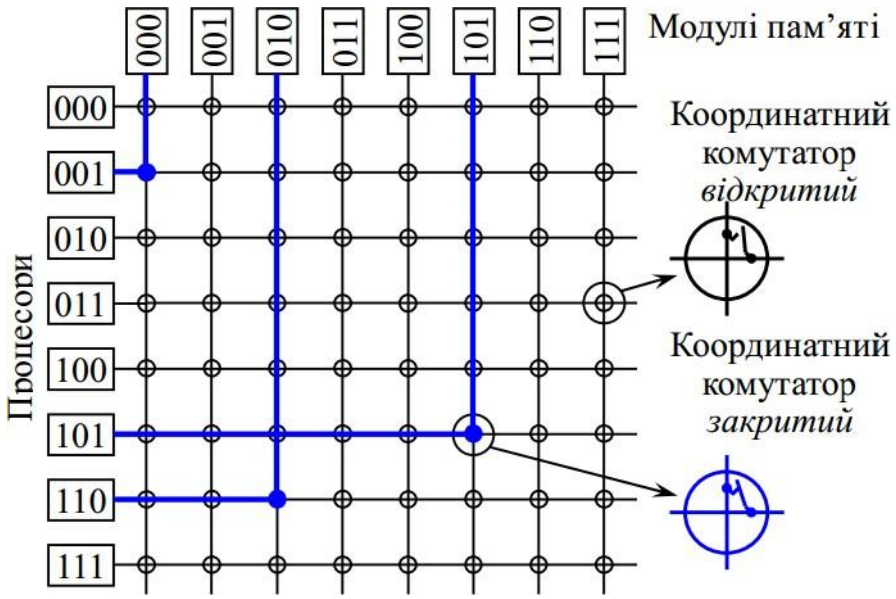


Рисунок 1.16 − Координатний комутатор 8×8

У кожному перетинанні горизонтальної (вхідної) і вертикальної (вихідної) лінії знаходиться перемикач (crosspoint), який можна замкнути або розімкнути залежно від того, потрібно з'єднати горизонтальну й вертикальну лінії або ні. На наведеному рисунку показані три замкнуті вузли (позначені зафарбованими кружечками) завдяки чому встановлюється зв'язок між парами процесор/пам'ять із номерами (001/000), (101/101) і (110/010) одночасно. Можлива безліч і інші комбінації. Для зображеної мережі число комбінацій дорівнює вісім факторіал (~40000).

Оскільки в такому координатному комутаторі практично використовується просторова маршрутизація (рис. 1.6), то мережа міжз'єднань не блокує пересилань даних. Це означає, що процесор завжди буде мати можливість зв'язатися з потрібним йому блоком пам'яті, навіть якщо якась лінія або вузол уже зайняті. Понад усе, в цьому разі ніякого попереднього планування не потрібно. Навіть якщо вже попередньо встановлено сім довільних зв'язків, завжди можна зв'язати процесор, що залишився, з вільним блоком пам'яті.

Не кращою властивістю координатного комутатора є те, що число вузлів росте як n^2. При наявності 1000 процесорів і 1000 блоків пам'яті необхідно мати мільйон вузлів. Проте, координатні комутатори широко використовуються в системах середніх розмірів.

В основі архітектури мультипроцесорів UMA з ***багатоступінчастими мережами*** лежить невеликий комутатор з декількома двонаправленими входами й виходами, з можливістю перемикання з будь-якого входу на будь-який вихід і у зворотному напрямку. Застосовуються комутатори з різною кількістю входів/виходів, прикладом може служити найпростіший комутатор 2×2 (рис. 1.17 а). Цей комутатор містить два входи й два виходи. Повідомлення, що приходять на кожну із вхідних ліній, можуть перемикатися на будь-яку вихідну лінію. Повідомлення, які передаються по мережі з такими комутаторами, можуть містити до чотирьох частин (рис. 1.17 б). У полі «Модуль» вміщується номер вузла, якому передається повідомлення. Поле «Адреса» визначає адресу даних у модулі пам'яті. У полі «Код операції» (КОП) утриму-

ється вказівка на виконувану операцію – запис або читання даних. Додаткове поле «Значення» може слугувати для поміщення переданого операнду. Біти вмісту поля «Модуль» визначають, на яку з ліній виходу – верхню або нижню – потрібно відправити повідомлення.

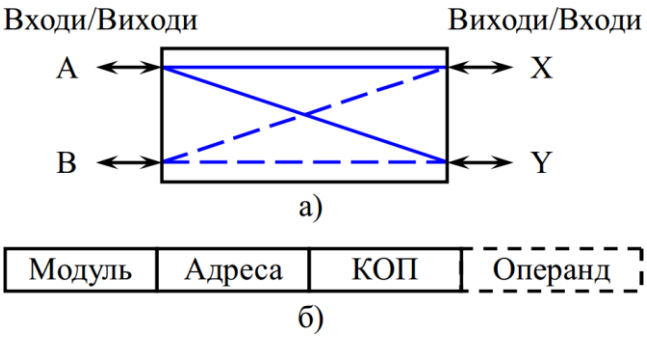


Рисунок 1.17 − Принцип дії комутатора багатоступінчастої мережі
а) структурна схема комутатора 2×2, б) формат повідомлення

Для одержання багатоступінчастих мереж комутатори 2×2 можна компонувати різними способами. Один з можливих варіантів – мережа Omega представлена на рис. 1.18. Тут, використовуючи 12 комутаторів, з'єднані 8 процесорів з 8 модулями пам'яті. Для n процесорів і n модулів пам'яті необхідна наявність $\log_2 n$ рівнів по $n/2$ комутаторів на кожному рівні, тобто всього $(n/2)\log_2 n$ комутаторів. Така кількість значно менше, ніж n^2 вузлів координатного комутатора, особливо для більших n.

Принцип роботи мережі Omega, можна пояснити в такий спосіб. Коли, наприклад, процесору 011 потрібно зчитати слово з модуля пам'яті 110, процесор посилає повідомлення зі значенням у полі «КОП», що відповідає операції читання слова з адреси в полі «Адреса». У полі «Модуль» міститься номер необхідного модуля пам'яті – 110. Коли повідомлення надходить на комутатор якого-небудь рівня 1, 2 або 3, комутатор виділяє старший біт поля «Модуль» і по ньому довідується про подальший напрямок розповсюдження команди. Потім виділений біт поля відкидається й на наступний рівень приходить повідомлення з полем «Модуль» зменшеним на один (старший) роз-

ряд. Якщо значення виділеного біта дорівнює 0, повідомлення передається на верхній вихід комутатора, а якщо 1 – на нижній.

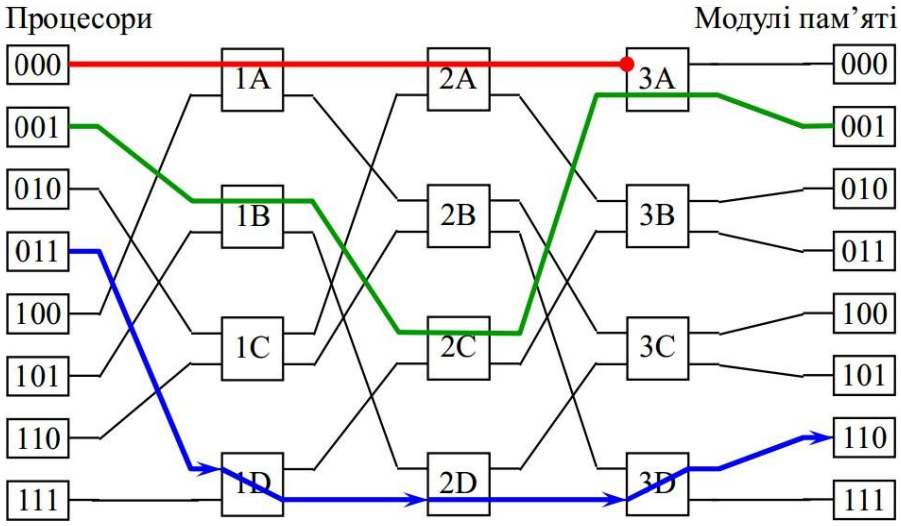


Рисунок 1.18 − Структура мережі Omega

У прикладі, на комутаторі першого рівня 1D старший біт дорівнює 1, і повідомлення відправляється через нижній вихід до комутатора другого рівня 2D. На цьому комутаторі поле «Модуль» уже дорівнює 10 і старший біт знову дорівнює 1, тобто повідомлення знову через нижній вихід відправляється до комутатору третього рівня 3D. Нарешті, на третьому рівні останній біт, що залишився, дорівнює 0 і, отже, повідомлення переходить на верхній вихід і прибуває до пам'яті з номером 110. Шлях, який пройшло повідомлення, на рис. 1.18 показаний стрілками.

Як тільки повідомлення пройде через всі рівні мережі, у полі «Модуль» встановлюється адреса процесора, який запросив виконання операції (у прикладі 011). Модуль пам'яті в поле «Операнд» поміщає слово, яке розташовано в комірці з адресою «Адреса», і сформоване в такий спосіб повідомлення відправляється назад процесору. Тільки тепер, при проходженні комутаторів на рівях мережі, напрямок передачі визначається за значенням молодшого біту поля «Модуль», який потім відкидається. Для описуваного прикладу зворотній путь буде наступним – нижній вихід в 3D (1), нижній вихід в 2D (1) і,

відповідно, верхній вихід в 1D (0).

Якщо під час пересилання цих повідомлень, процесору 001 знадобиться записати слово в модуль пам’яті 001, то буде виконаний аналогічний процес передачі через верхній вихід комутатора 1B, верхній вихід 2C і нижній 3A (рис. 1.18 жирна лінія). Оскільки ці два запити використовують зовсім різні комутатори, лінії зв’язку і модулі пам’яті, вони можуть виконуватися паралельно.

Але якщо процесору 000 одночасно із цим знадобиться доступ до модуля пам’яті 000. Його запит вступить у конфлікт із запитом процесора 001 на комутаторі 3A (рис. 1.18 лінія із крапкою на кінці). Одному з цих процесорів доведеться почекати. На відміну від координатного комутатора, мережа Omega – це мережа з блокуванням, не всякий набір запитів може передаватися одночасно. Конфлікти можуть виникати при використанні тієї самої лінії зв’язку або того самого комутатора, а також між запитами, спрямованими до пам’яті, і відповідями, що виходять з пам’яті.

Як уже говорилося, кількість процесорів у мультипроцесорах UMA з одною шиною звичайно є невеликою й обмежується декількома десятками процесорів, а для координатних мультипроцесорів або мультипроцесорів з комутаторами потрібно дороге апаратне забезпечення, і вони незначно більші за розміром. Щоб одержати понад 100 процесорів була запропонована архітектура мультипроцесорів ***NUMA*** (*NonUniform Memory Access – з неоднорідним доступом до пам’яті*). Як і мультипроцесори UMA, вони забезпечують єдиний адресний простір для всіх процесорів, але доступ до локальних модулів пам’яті відбувається швидше, ніж до віддалених. Отже, всі програми для систем UMA без змін працюють на машинах NUMA, але продуктивність виявляється гіршою, ніж на машині UMA з тією же тактовою частотою.

Машини NUMA мають три ключові характеристики, якими всі вони володіють і які в сукупності відрізняють їх від інших мультипроцесорів:

1) існує один адресний простір, видимий для всіх процесорів;
2) доступ до віддаленої пам’яті виконується з використанням команд читання/запису;

3) доступ до віддаленої пам’яті відбувається повільніше, ніж до локальної пам’яті.

Якщо час доступу до віддаленої пам’яті не схований наявністю кеш-пам’яті, то така система називається ***NC-NUMA*** (*No Caching NUMA* – ***NUMA без кэшування***). Якщо присутні погоджені кеші, то система називається ***CC-NUMA*** (*Coherent Cache NUMA* – ***NUMA з погодженою кеш-пам’яттю***).

Програмісти часто називають її ***апаратною DSM*** (*Distributed Shared Memory – розподілена спільно використовувана пам’ять*), оскільки вона по суті подібна із програмної DSM, але реалізується в апаратному забезпеченні з використанням сторінок маленького розміру.

На рис. 1.19 наведено приклад однієї з найпростіших структур комп’ютера NC-NUMA із двома рівнями шин. Система складається з набору процесорів (Пр), кожний із власною пам’яттю (ОЗП), звертання до якої виробляється по локальній шині. Крім того, процесори зв’язані один з одним системною шиною. Коли запит пам’яті приходить у блок управління пам’яттю (БУП), виробляється перевірка й визначається, чи перебуває потрібне слово в локальній пам’яті. Якщо так, то запит відправлявся по локальній шині. Якщо ні, то запит направлявся по системній шині до вузла, що містить дане слово. Природно, друга операція займає набагато більше часу, чим перша. Виконання програми з вилученої пам’яті займає приблизно в 10 разів більше часу, чим виконання тієї ж програми з локальної пам’яті.

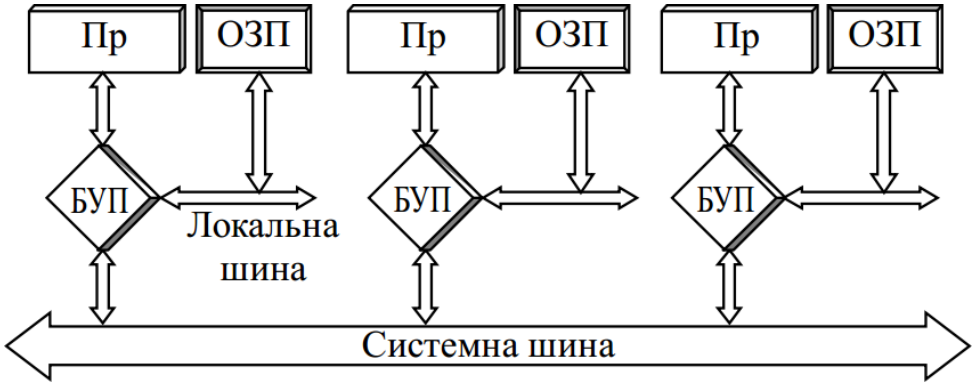


Рисунок 1.19 − Машина NUMA із двома рівнями шин
(Пр – процесор, ОЗП – оперативний запам’ятовуючий пристрій,
БУП – блок управління пам’яттю)

Погодженість пам’яті в комп’ютерах NC-NUMA гарантована через відсутність кеш-пам’яті. Кожне слово пам’яті перебуває тільки в одному місці, тому немає ніякої небезпеки появи копії із застарілими даними: тут взагалі немає копій. Має велике значення, у якій саме фізичній пам’яті перебуває та або інша сторінка віртуальної пам’яті, оскільки від цього залежить продуктивність. У системах NC-NUMA використовується дуже складне програмне забезпечення для переміщення сторінок, з метою максимального збільшення продуктивності.

Звичайно існує сторожовий процес («демон»), так званий сторінковий сканер, що запускається кожні кілька секунд. Він стежить за статистикою використання сторінок і переміщає їх таким чином, щоб поліпшити продуктивність. Якщо сторінка виявиться в неправильному місці, сторінковий сканер перетворить її таким чином, щоб наступне звертання до неї викликало помилку через відсутність сторінки. Коли відбувається така помилка, приймається рішення про те, куди помістити цю сторінку, можливо, в іншу пам’ять, з якої вона була взята раніше. Для запобігання пробуксовки існує правило, яке говорить, що якщо сторінка була поміщена в те або інше місце, вона повинна залишатися в цьому місці якийсь заздалегідь визначений час ΔT. Була розглянута безліч алгоритмів, але жоден з них не працює краще інших при будь-яких обставинах.

Мультипроцесори NC-NUMA зі структурою, зображеної на рис. 1.19, погано розширюються, оскільки в них немає кеш-пам’яті. Щораз переходити до віддаленої пам’яті, щоб одержати доступ до слова, якого немає в локальній пам’яті, дуже невигідно: це сильно знижує продуктивність. Однак з додаванням кеш-пам’яті потрібно додати спосіб сумісності кешів. Тут знов може допомогти стандартний спосіб – спосіб відстеження системної шини. Технічно це зробити нескладно, але верхня межа такої системи – передача даних від 64-х процесорів.

Самий популярний підхід для побудови більших мультипроцесорів це створення систем NUMA з ***погодженою кеш-пам’яттю*** – ***CC-NUMA***. Найчастіше такі системи будуються як мультипроцесори на основі каталогу (рис. 1.20). Основна ідея складається в збереженні бази даних, що повідомляє, де саме перебуває кожний рядок кеш-пам’яті і який в неї стан. При звертанні до рядка кеш-пам’яті процесор з бази даних витягає інформацію про те, де перебуває цей рядок і змінювався він чи ні. Оскільки звертання до бази даних відбувається

на кожній команді, що звертається до пам’яті, база даних зберігається у високошвидкісному спеціалізованому апаратному забезпеченні, яке здатне видавати відповідь на запит за незначну частку циклу шини.

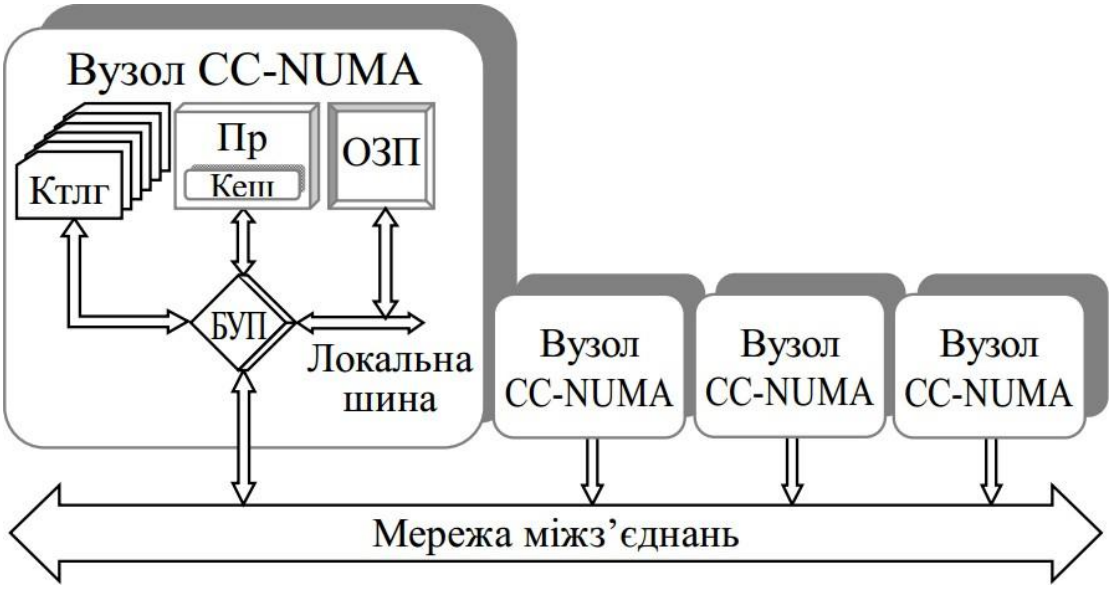


Рисунок 1.20 − Мультипроцесор на основі каталогу
(Ктлг – апаратний каталог бази даних, решта позначень як і на рис. 1.19)

Найкраще принцип дії мультипроцесора на основі каталогу («Ктлг» на рис. 1.20) розглянути на прикладі не дуже потужного паралельного комп’ютера, що складається з 256 вузлів. Кожний вузол містить процесор і ОЗП обсягом 16 МіБ, яке зв’язано з ним через локальну шину. Таким чином, загальний обсяг поділюваної пам’яті становить 2^{32} байтів. Вона розділена на 2^{26} рядків кеш-пам’яті по 64 байта кожний. Пам’ять статично розподілена по вузлах: 0 ÷ 16МіБ у вузлі 0, 16МіБ ÷ 32МіБ – у вузлі 1 і т.д. Вузли зв’язані швидкісною мережею міжзв’язків. Топологія мережі може бути у вигляді ґрат, гіперкуба або будь-якою іншою. У кожному вузлі містяться 2^{18} елементів каталогу для рядків кеш-пам’яті розміром у 64 байта, створюючи власну пам’ять каталогу обсягом 2^{24} байтів. Для спрощення розгляду можна припустити, що рядок може втримуватися максимум в одній кеш-пам’яті.

Коли, наприклад, процесор (Пр) з 20-го вузла виконує операцію завантаження слова з кешованого рядка, то спочатку команда передається в блок керування

пам’яттю (БУП), який перетворює її у фізичну адресу. Нехай ця фізична адреса дорівнює 0x24000108. Тут же адреса розбивається на три частини, як показано на рис. 1.21 а). Вісім старших біт задають адресу вузла, до якого відбувається звернення, наступні 18 вказують на номер рядка кеш-пам’яті, останні шість бітів – це зміщення слова усередині рядка. Для зазначеної вище фізичної адреси ці три частини в десятковій системі числення дорівнюють: вузол – 36, рядок – 4 і зміщення – 8. Кожному номеру рядка в каталозі (рис. 1.21 б) відповідає один запис, у якому вказується присутність рядка де-небудь у кеш-пам’яті (біт присутності) і конкретна адреса вузла, у кеш-пам’яті якого перебуває цей рядок. Блок керування пам’яттю бачить, що слово пам’яті, до якого здійснюється звернення, перебуває у вузлі 36, а не у вузлі 20, тому він надсилає запит через мережу у вузол 36, довідується, чи є 4-ий рядок у кеш-пам’яті, і якщо так, то де саме.

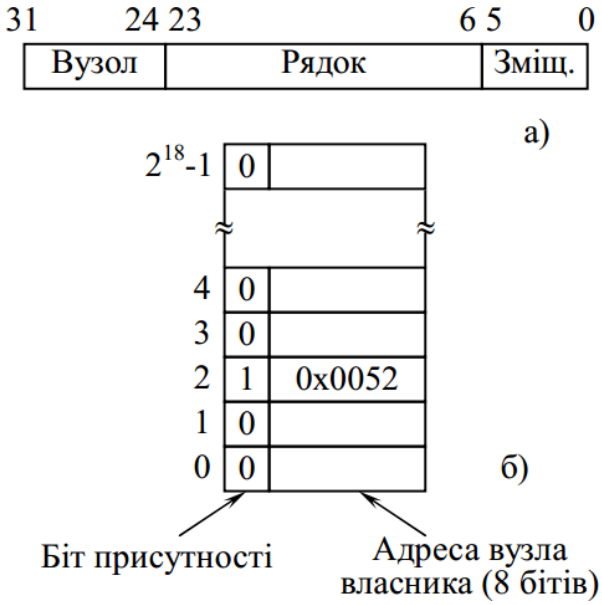


Рисунок 1.21 − Принцип роботи каталогу

а) формат 32 бітної адреси пам’яті, б) структура каталогу

Коли запит прибуває у вузол 36, він направляється в апаратне забезпечення каталогу. Апаратне забезпечення індексує таблицю з 2^{18} елементів (один елемент на кожний рядок кеш-пам’яті) і витягає елемент з індексом чотири. З рис. 1.21 б) видно, що цей рядок відсутній в кеш-пам’яті (біт присутності дорівнює 0), тому апаратне забезпечення викликає рядок 4 з локального ОЗП і від-

правляє його у вузол 20. Потім четвертий елемент каталогу оновляється, щоб показати, що цей рядок тепер перебуває в кеш-пам'яті у вузлі 20.

Якщо ж звернення відбудеться до рядка, що перебуває в кеш-пам'яті, наприклад до рядка 2 з того ж вузла 36, то з каталогу в цьому вузлі видно (див. рис 1.21 б), що рядок зараз перебуває в кеш-пам'яті у 82-у вузлі. У цьому випадку апаратне забезпечення оновляє елемент каталогу 2, щоб повідомити, що рядок тепер переміщений до вузла 20, а потім посилає повідомлення у вузол 82, щоб рядок з нього був переданий у вузол 20, і оголошує недійсною його кеш-пам'ять.

Є інтересним обчислити, скільки пам'яті займають каталоги. Кожний вузол містить 16 МіБ ОЗП і 2^{18} 9-бітних елементів для спостереження за цим ОЗП. Таким чином, непродуктивні витрати каталогу складають приблизно 9×2^{18} бітів від 16 МіБ або близько 1,76%, що цілком припустимо. Навіть якщо довжина рядка кеш-пам'яті становить не 64, а 32 байта, непродуктивні витрати складуть усього 4%. Якщо довжина рядка кеш-пам'яті дорівнює 128 байтів, непродуктивні витрати будуть нижче 1%.

Очевидним недоліком цієї розробки є те, що рядок може бути кешований тільки в одному вузлі. Щоб рядок можна було кешувати у декількох вузлах, потрібним буде якийсь спосіб їхнього знаходження (наприклад, щоб повідомляти недійсними або обновляти їх при запису). Можливі різні варіанти.

Одна з можливостей – надати кожному елементу каталогу додатково *k* часток для визначення інших вузлів, що дозволить зберігати кожний рядок у декількох блоках кеш-пам'яті (припустимо до *k* різних вузлів). *Друга можливість* – замінити номер вузла його бітовим відображенням – один біт на вузол. Тут немає обмежень на кількість копій, але істотно зростають непродуктивні витрати. Каталог, який містить 256 бітів для кожного 64-байтного (512-бітного) рядка кеш-пам'яті, вносить вже непродуктивні витрати більші за 50%. *Третя можливість* – зберігати в кожному елементі каталогу 8-бітне поле і використовувати це поле як заголовок зв'язного списку, який пов'язує всі копії рядка кеш-пам'яті разом. При такій стратегії потрібний додатковий простір у кожному вузлі для покажчиків зв'язного списку. Крім того, необхідно переглядати зв'язний список повністю, у разі виникнення потреби знайти всі копії. Кожна із трьох стратегій

має свої переваги й недоліки. На практиці *використовуються всі три стратегії.*

Комп'ютери NUMA і CC-NUMA мають один великий недолік: звертання до віддаленої пам'яті відбуваються набагато повільніше, ніж звертання до локальної пам'яті. У комп'ютерах CC-NUMA цей недолік у значній мірі маскується завдяки використанню кеш-пам'яті. Однак якщо кількість необхідних віддалених даних сильно перевищує місткість кеш-пам'яті, промахи будуть відбуватися постійно й продуктивність стане дуже низкою.

Навпроти, комп'ютери UMA мають дуже високу продуктивність, але обмежені в розмірах і досить дорого коштують. Комп'ютери NUMA можуть розширюватися до великих розмірів, але в них потрібно ручне або напівавтоматичне розміщення сторінок віртуальної пам'яті, але ця дія не завжди проходить вдало. Справа в тому, що дуже важко завбачити, де які сторінки можуть знадобитися, і крім того, сторінки важко переміщати через їх великий розмір. Машини CC-NUMA можуть працювати з дуже низькою продуктивністю, якщо великій кількості процесорів потрібно багато віддалених даних. Так чи інакше, кожна із цих розробок має істотні недоліки.

Однак існує процесор, у якому всі ці проблеми вирішуються за рахунок того, що основна пам'ять кожного процесора використовується як кеш-пам'ять. Така розробка називається ***COMA*** (*Cache Only Memory Access – **доступ тільки до кеш-пам'яті***).

COMA – це архітектура, яка конкурує з CC-NUMA, спроектована з аналогічними цілями, але має цілком іншу реалізацію. Замість того щоб розподіляти частини пам'яті й підтримувати їхню когерентність, як це робиться в CC-NUMA, COMA-вузол не має пам'яті, а тільки великі кеші під назвою *Attraction Memories – **пам'ять, яка притягує***, у кожному обробному блоці – ***кводі*** (quad). Така архітектура передбачає відсутність основної пам'яті у вузлах. Для значень даних, які розташовані у «ні домашніх» комірках пам'яті, на вузлі працює одна загальна для всіх кводів копія ОС, а когерентність на вузлі підтримується спеціальним апаратним обладнанням. Ця особливість компенсується наявністю в кожному вузлі кеш великого обсязу. Дана архітектура накладає більш жорсткі обмеження на міжвузлові комунікації, тому що кожний

кеш-промах вимагає звертання через мережу до основній пам'яті за актуальними даними. Однак використання основної пам'яті як великої кеш-пам'яті збільшує частоту успішних звернень до кешів і, таким чином, підвищує продуктивність.

На жаль, нічого ідеального не існує. У системі COMA з'являються дві нових проблеми:

Перша проблема зв'язана з наступним. Після трансляції віртуальної адреси у фізичну, може виявитися, що потрібного рядка немає в апаратній кеш-пам'яті. У такому випадку дуже важко визначити, чи є взагалі цей рядок в основній пам'яті. Апаратне забезпечення тут не допомагає, оскільки кожна сторінка складається з великої кількості окремих рядків кеш-пам'яті, які переміщаються в системі незалежно друг від друга. Навіть якщо відомо, що рядок відсутній в основній пам'яті, все ж таки важко визначити, місце його знаходження. У цьому випадку не можна навіть запитати власний вузол, оскільки вузла як такого в системі немає.

Було запропоновано кілька вирішень цієї проблеми. По-перше, можна *ввести додаткове апаратне забезпечення*, яке буде стежити за тегом кожного рядка кеш-пам'яті. Тоді блок керування пам'яттю може порівнювати тег потрібного рядка із тегами всіх рядків кеш-пам'яті, поки не буде виявлений збіг.

Інше рішення – *відображати сторінки повністю*, але при цьому не вимагати присутності всіх рядків кеш-пам'яті. У цьому випадку апаратному забезпеченню знадобиться бітове відображення для кожної сторінки, де один біт для кожного рядка вказує на присутність або відсутність цього рядка. У цій схемі, що називається ***простою схемою COMA***, якщо рядок присутній, він повинен перебувати в правильній позиції на відповідній сторінці. Якщо рядок відсутній, то будь-яка спроба використати його викликає переривання за відсутністю, а це дозволяє програмному забезпеченню знайти потрібний рядок і звернутися до нього. Таким чином, система буде шукати тільки ті рядки, які дійсно перебувають у віддаленій пам'яті.

Ще одне рішення – *надати кожній сторінці власну машину* (ту, яка містить елемент каталогу даної сторінки, а не ту, у якій перебувають дані). По-

тім можна відправити повідомлення у власну машину, щоб знайти місце розташування даного рядка.

Можна також запропонувати й таке рішення – *організувати пам'ять у вигляді дерева* й здійснювати пошук в напрямку до кореня, поки не буде виявлений потрібний рядок.

Друга проблема пов'язана *з унеможливленням видалення останньої копії*. Як і в машині CC-NUMA, рядок кеш-пам'яті може перебувати одночасно в декількох вузлах. Якщо відбувається промах кеш-пам'яті, рядок потрібно звідкись викликати, а це значить, що його потрібно відкинути. У випадку ж якщо обраний рядок виявляється останньою копією його не можна відкидати.

Одне з можливих рішень – *повернення до ідеї каталогів* й за їх допомогою перевіряти, чи існують інші копії. Якщо так, то рядок можна безбоязно відкидати. Якщо ні, то копію потрібно перемістити куди-небудь ще.

Інше рішення – *позначити одну з копій* кожного рядка кеш-пам'яті *як головну копію* й ніколи її не відкидати. При такому підході не потрібна перевірка каталогу.

На додаток до сказаного необхідно відзначити, що хоча апаратні засоби COMA і здатні компенсувати недоліки алгоритмів ОС по розподілу пам'яті й диспетчеризації її, але для COMA-систем потрібним є внесення змін у підсистему віртуальної пам'яті ОС і, додатково до плати кеш-пам'яті, замовлення плати пам'яті (когерентного з'єднання).

На закінчення розгляду подібних систем слід зазначити, що на сьогодення лідером у розробці суперкомп'ютерних платформ є фірма IBM, що розвиває кілька потужних систем аж до петафлопс-систем. Так розроблена система Blue Gene/P, що розрахована на перспективну продуктивність рівня петафлопс і вище. Спочатку система була орієнтована на сферу наукових досліджень, однак розширені ресурси пам'яті й кластерні вузли із симетричною багатопроцесорною обробкою (SMP) роблять цю модель досить привабливою для широкого спектра інших прикладних застосувань. Суперкомп'ютери IBM, побудовані на базі новітнього покоління процесорів POWER, займають

лідируючі позиції на ринку високопродуктивних обчислювальних систем, призначених для розв’язання таких комерційних і технічних завдань, як прогнозування погоди, моделювання змін клімату, дослідження нових джерел енергії, проектування в автомобільній і аерокосмічній індустрії. Однак варто сказати, що цей комп’ютер, хоча й має деякі відмітні властивості систем SMP, все-таки відноситься не до мультипроцесорів, а до мультикомп’ютерів, точніше кажучи – це мультикомп’ютер на основі багатьох мультипроцесорів.

1.2 Побудова кластерних систем

Розглянуті вище системи мультипроцесорів, з погляду користувачів, володіють рядом досить корисних властивостей. Основним з них є те, що наявність розділеної пам’яті дозволяє програмам, які написані для мультипроцесора, одержувати доступ до будь-якої комірки пам’яті, не маючи ніякої інформації про внутрішню топологію або схему реалізації пам’яті.

Однак мультипроцесори мають і деякі недоліки. По-перше, мультипроцесори не можна розширити до великих розмірів. Всі спроби збільшити кількість процесорів у системах UMA натрапляють на невиправдане ускладнення і подорожчання апаратного забезпечення. А збільшення кількості процесорів у системах NUMA до двох-трьох сотень, не говорячи вже про тисячі, можливо тільки завдяки значній розбіжності в часі доступу до локальній й віддаленій пам’яті.

Крім того, конфліктні ситуації при звертанні до пам’яті в мультипроцесорах сильно впливають на їхню продуктивність. Якщо 100 процесорів постійно намагаються зчитувати й записувати ті самі змінні, конфліктна ситуація для різних модулів пам’яті, шин і каталогів приводить до різкого зниження продуктивності.

Останнім часом внаслідок цих і інших факторів розроблювачі виявляють величезну зацікавленість до паралельних комп’ютерів з розподіленою пам’яттю, у яких кожний процесор має свою власну пам’ять, до якої інші

процесори не можуть одержати прямий доступ – це мультикомп'ютери. Програми на різних процесорах у мультикомп'ютері взаємодіють одна з одною за допомогою примітивів відправлення/одержання повідомлень. Такий тип взаємодії повністю змінює модель програмування.

Кожний вузол у мультикомп'ютері (рис. 1.22) складається з одного або декількох процесорів (Пр), ОЗП – загального для процесорів тільки даного вузла, диска і/або інших пристроїв введення/виведення (ПВВ), а також процесора передачі даних (ППД). ППД зв'язані між собою через високошвидкісну комунікаційну мережу. Використовується безліч різних топологій, схем комутації й алгоритмів вибору маршруту. Всі мультикомп'ютери подібні в одному: коли програма виконує примітив відправлення повідомлення, ППД одержує повідомлення й передає блок даних до цільової машини (можливо, після попереднього запиту й одержання дозволу).

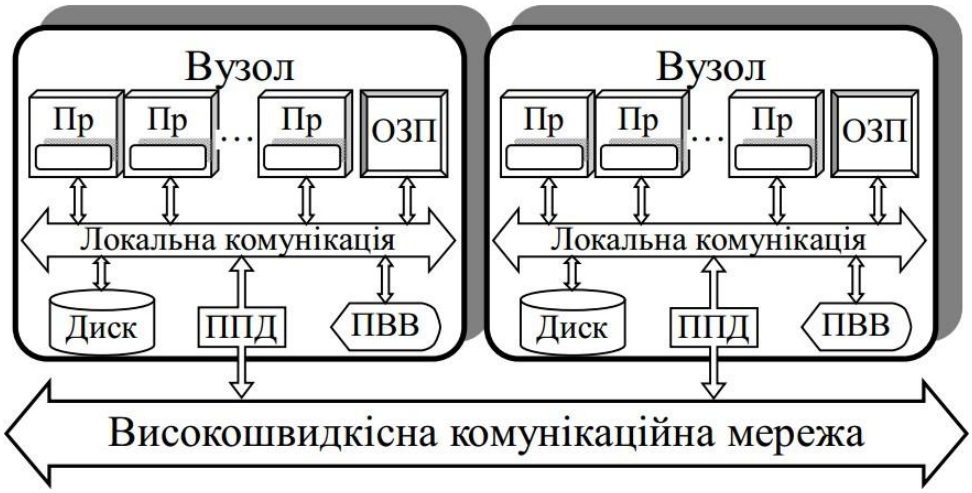
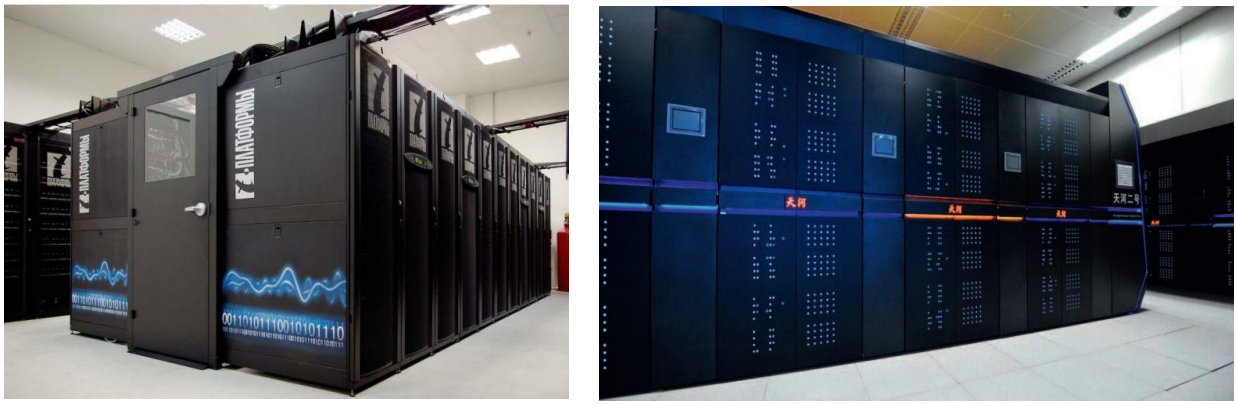


Рисунок 1.22 − Узагальнена структурна схема мультикомп'ютера
Пр – процесор з кеш-пам'яттю, ОЗП – оперативний запам'ятовуючий пристрій, ПВВ – пристрій введення/виведення, ППД – процесор передачі даних

Мультикомп'ютери бувають різних типів і розмірів, тому дуже важко, майже практично неможливо, привести їх докладну класифікацію. Проте, виділяють (див. рис. 1.12) два загальних типи: MPP і COW (NOW).

1.2.1 Системи з масовим паралелізмом

Ще зовсім недавно, років п’ять – сім тому, ***процесори з масовим паралелізмом MPP*** (***Massively Parallel Processors***) асоціювалися з величезними суперкомп’ютерами вартістю в кілька мільйонів доларів. Прикладом подібних комп’ютерів може слугувати наймогутніший російський, один із самих потужних у світі суперкомп’ютер – СКИФ МГУ-Чебышев (рис. 1.23 а), що містить 1250 4-ядерних процесорів Intel Xeon, або лідер світового рейтингу самих потужних комп’ютерів 2014 року – комп’ютер китайського виробництва Tianhe-2 (рис. 1.23 б), із продуктивністю в 33,86 петафлопс. Але сьогодні ця гілка в архітектурі суперкомп’ютерів перекриває весь спектр – від систем, яким потрібні приміщення у сотні квадратних метрів, до офісних систем, габарити яких не перевищують розмірів канцелярської шафи або столу (рис. 1.24).



а) б)

Рисунок 1.23 − Зовнішній вигляд суперкомп’ютерів архітектури MPP
а) стійки-шафи суперкомп’ютера СКИФ МГУ-Чебышев,
б) машинний зал комп’ютера Tianhe-2 китайського виробництва

MPP системи використовуються в різних галузях науки й техніки для виконання складних обчислень, обробки великої кількості транзакцій у секунду, керування великими базами даних, і т.д. Спочатку це були супер-

комп’ютери, які призначалися здебільшого для наукових розрахунків, але зараз багато таких систем знаходять застосування в комерції. У цілому, можна говорити, що мультикомп’ютери MPP витиснули SIMD-машини, векторні суперкомп’ютери й матричні процесори.

а) б)

Рисунок 1.24 − Зовнішній вигляд малогабаритних комп’ютерів типу MPP
а) суперкомп’ютер Aurora Tigon, б) офісна установка HPC Aurora

У більшості комп’ютерів MPP використовуються стандартні процесори. Це можуть бути процесори Intel Pentium, Sun UltraSPARC, IBM RS/6000 і DEC Alpha. Відрізняє мультикомп’ютери *наявність високопродуктивної комунікаційної мережі*, по якій можна передавати повідомлення з низьким часом запізнювання і високою пропускною здатністю. Обидві характеристики (час запізнювання і пропускна здатність) дуже важливі, оскільки повідомлення у більшості невеликі за розміром (менш ніж 256 байтів), хоча при цьому головний внесок у загальний трафік вносять великі повідомлення (більші за 8 КіБ). Мультикомп’ютери MPP поставляються разом з доволі дорогим програмним забезпеченням і бібліотеками.

Ще одна дуже важлива й цікава характеристика MPP – це *величезні обсяги введення/виведення*. За допомогою систем MPP звичайно приходиться обробляти величезні масиви даних, іноді Тебібайти. Ці дані повинні бути розподілені по численних дисках, і їх потрібно передавати між пристроями системи з великою швидкістю.

Нарешті, важливо пам'ятати про ще одну обов'язкову властивість MPP – *високої стійкості до відмов*. При наявності тисяч процесорів кілька несправностей у тиждень неминучі. Припиняти роботу системи через збої в одному із процесорів неприйнятно, особливо якщо очікується, що збої будуть траплятися щотижня. Тому у великих комп'ютерах MPP завжди є спеціалізована апаратна й програмна підтримка постійного моніторингу системи, передбачення, виявлення й виправлення неполадок.

Резюмуючи можна сказати, що системи MPP являють собою просто ряд більш-менш стандартних обчислювальних вузлів, зв'язаних один з одним високошвидкісною комунікаційною мережею. Всі ці елементи були докладно розглянуті вище, і повторюватися не має сенсу, а основних принципів організації комп'ютерів MPP зовсім не багато. Тому більш доцільно розглянути ці основні принципи на конкретному комп'ютері, наприклад із сімейства BlueGene, що випускається фірмою IBM.

Проект BlueGene був задуманий IBM ще в1999 році як суперкомп'ютер для вирішення обчислювальних задач великої складності в області біології. Зокрема, біологи вважають, що функції білка визначаються його тривимірною структурою. Але визначення форми навіть однієї невеликої молекули білка на суперкомп'ютерах того часу зажадало б декількох років обчислень. У людському ж організмі біля півмільйона різних білків, деякі з них винятково складні, і порушення в структурі кожного можуть призводити до серйозних захворювань. Очевидно, що для розрахунку тривимірної структури всіх людських білків потрібно на кілька порядків підвищити обчислювальну потужність.

У листопаді 2001 з'явився й перший замовник першого комп'ютера із

сімейства BlueGene, а вже в 2007 році компанія IBM представила комп'ютер BlueGene другого покоління BlueGene/P.

Метою проекту BlueGene була побудова системи MPP, що не тільки була б найшвидшою, але й найефективнішою відносно показників терафлоп/долар, терафлоп/Вт і терафлоп/м^3. Це змусило IBM відмовитися від застосування найшвидших складових незалежно від їхньої ціни. Було вирішено випустити власний однокристальний процесор, який працює з помірною швидкістю, але має низьке енергоспоживання, щоб на його основі побудувати великий комп'ютер з ефективним розташуванням компонентів. Перший BlueGene/P містив 65536 мікропроцесорів, а його продуктивність сягала 167 терафлоп/с. На момент введення у експлуатацію, цей суперкомп'ютер був шостим по швидкодії у світі й входив до числа лідерів по ефективності споживання потужності: він виконував 371 мегафлоп/Вт. В 2009 році BlueGene/P був доповнений до 294912 процесорів, у результаті чого його обчислювальна потужність сягла 1 петафлоп/с.

Основою системи BlueGene/P є вузол, утворений спеціалізованою мікросхемою, структура якої показана на рис. 1.25. Мікросхема містить у чотири ядра PowerPC 450, які працюють на частоті 850 МГц. PowerPC 450 – це конвеєризований здвоєний суперскалярний процесор, який часто застосовувався у вбудованих системах. У кожному ядрі є пара здвоєних блоків виконання операцій із рухомою крапкою (Floating Point Unit, FPU), що в сумі дозволяє за один цикл виконувати 8 команд із рухомою крапкою. Ці блоки доповнені підтримкою SIMD-команд, які можуть бути корисні при обробці масивів. Але в цілому, процесор не можна зарахувати до рекордсменів з продуктивності.

На мікросхемі підтримуються три рівні кэшування. Кеш першого рівня роздільний, у ньому 32 Кіб приділяється для команд і ще 32 Кіб для даних. Розмір об'єднаного кеш другого рівня складає 2 КіБ. У дійсності, це не стільки кеш-пам'ять, скільки буфери передвибірки. У кеш другого рівня для підтримки погодженості реалізований механізм спостереження один за одним. Третій рівень представлений об'єднаним цілісним кеш обсягом 4 МіБ, що спі-

льно використовується обома кеш другого рівня. Кеші першого рівня всіх чотирьох процесорів також погоджені. Таким чином, коли загальний блок пам’яті присутній відразу в декількох кеш, результати запису в нього одним процесором негайно відображаються в кеш інших процесорів. Звертання до пам’яті, яке викликає кеш-промах на першому рівні й кеш-влучення на другому, обробляється 11 тактів. При кеш-промаху на другому рівні кеш-влучення на третьому обробляється вже 28 тактів. Нарешті, при кеш-промаху на третьому рівні доводиться звертатися до головної пам’яті (DDR SDRAM), на що потрібно близько 75 тактів.

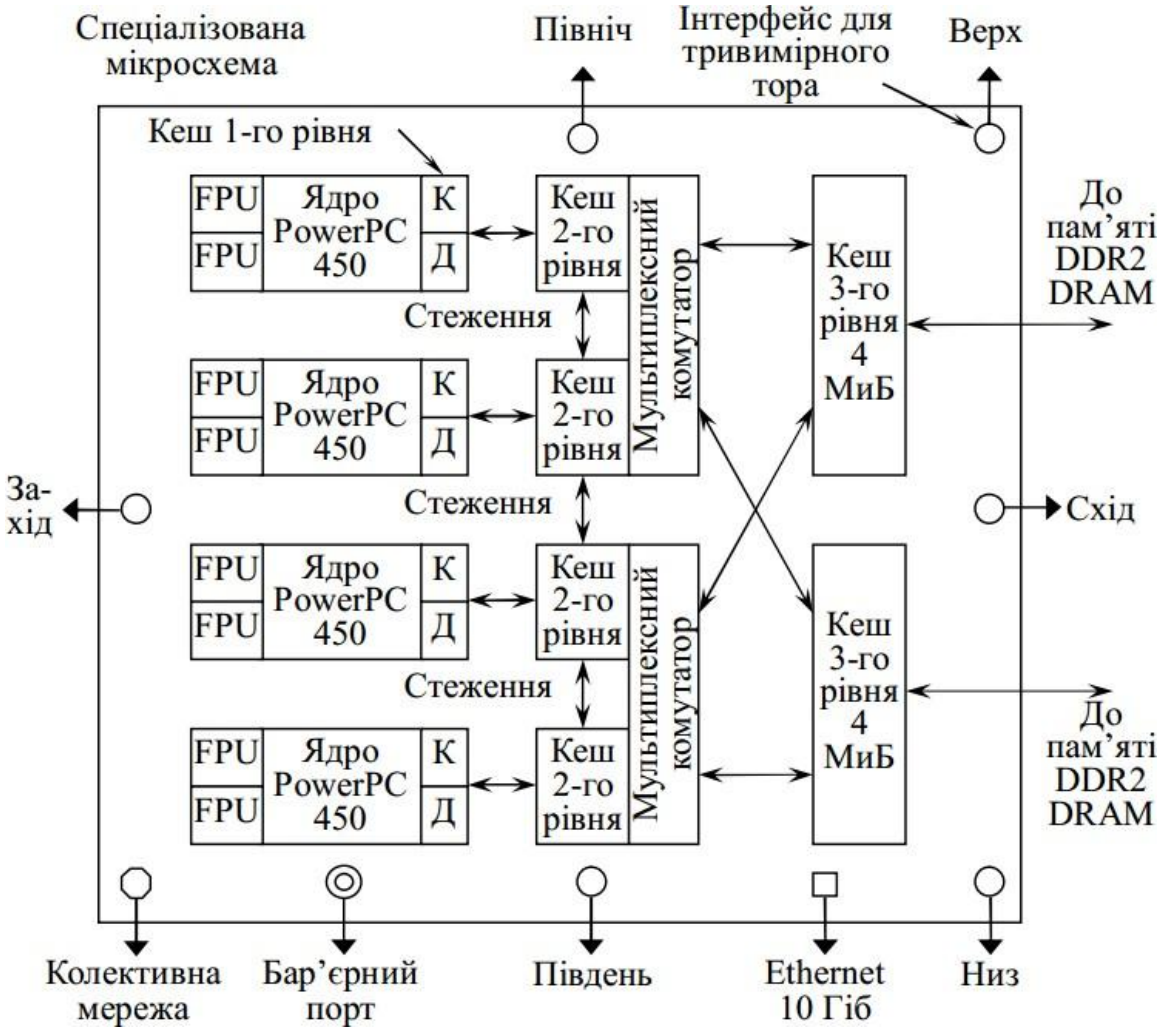


Рисунок 1.25 − Спеціалізований мікропроцесор у системі BlueGene/P

Чотири процесори зв’язані високопродуктивною шиною в мережу з то-

пологією «тривимірний тор», що вимагає шести з'єднань: верх, низ, північ, захід, південь і схід. Крім того, кожний процесор через додатковий порт зв'язується із колективною мережею, яка використовується для розсилання даних між процесорами. Бар'єрний порт прискорює операції синхронізації, надаючи кожному процесору швидкий доступ до спеціалізованій синхронізуючій мережі.

Для більш високого рівня в IBM була розроблена спеціалізована плата, на яку встановлюється мікропроцесор і ОЗП DDR2 обсягом 2 ГіБ. У свою чергу, плати монтуються по 32 плати на панель, яка вбудовується до наступного рівня. Таким чином одержується 32 мікросхеми (тобто 128 процесорів) на панель. Тому що на кожній платі є DRAM обсягом 2 ГіБ, усього на панелі виходить 64 ГіБ пам'яті. На наступному рівні 32 такі панелі вставляються в стійку, у результаті у стійці містяться 4096 процесорів.

Нарешті, вся система, складається з 72 стійок і містить 294912 процесорів. PowerPC 450 обробляє до 6 команд/цикл; таким чином, повна система BlueGene/P теоретично може обробляти за цикл до 1769472 команд. При тактовій частоті 850 Мгц теоретична продуктивність системи сягає 1,504 петафлоп/с. Однак, через конфлікти даних, затримок пам'яті й недостатнього паралелізму, фактична продуктивність при виконанні реальних програм на BlueGene/P сягає значення близько 1 петафлоп/с.

Система являє собою мультикомп'ютер у тому розумінні, що жоден із процесорів не має безпосереднього доступу до всієї пам'яті, якщо не вважати 2 ГіБ власної пам'яті на платі. Ні в однієї пари процесорів немає загальної пам'яті. Крім того, не підтримується виклик сторінок на вимогу, оскільки для розміщення сторінок немає локальних дисків. Замість цього в системі є 11524 вузла введення/виведення, які з'єднуються з дисками й іншими периферійними пристроями.

Незважаючи на виняткові розміри системи, вона досить проста й у ній не використовуються які-небудь особливі технології, за винятком, хіба що надзвичайно щільного розміщення вузлів. Тому що основними цілями були

надійність і доступність, досить ретельно були спроектовані системи живлення, охолодження, кабельні системи й т.п., все це дозволило підняти середній час наробітки на відмову до 10 днів.

Для підключення всіх мікросхем потрібна високопродуктивна мережа міжз'єднань, яка добре масштабується. Як топологія був обраний тривимірний тор розміром 72×32×32. Таким чином, кожній мікросхемі потрібні 6 ліній зв'язку: 2 для сусідів, логічно розташованих зверху й знизу, 2 для сусідів з півночі й півдня, 2 для сусідів із заходу й сходу (див. відповідні позначення на рис. 1.25). Конструктивно, кожна стійка на 1024 вузла утворює тор розміром 8×8×16. Пари сусідніх стійок з'єднуються в тор розміром 8×8×32. Чотири пари стійок з одного ряду утворюють тор розміром 8×32×32, і нарешті, всі 8 рядів дають тор розміром 72×32×32.

Таким чином, всі з'єднання є двоточечні й працюють на швидкості 3,4 Гіб/с. Тому що від кожного з 73728 вузлів ідуть три лінії зв'язку, по однієї на кожний вимір, загальна пропускна здатність системи становить 752 Тіб/с. Якщо вважати, що середній обсяг книг з комп'ютерних наук становить 750-800 щільнозаповнених сторінок, то по такій мережі за секунду можна передати вміст 2,5 млн таких книг.

Взаємодія в тривимірному торі підтримується у формі ***віртуальної наскрізної маршрутизації*** (virtual cut through routing). Цей підхід у чомусь нагадує комутацію із проміжним зберіганням, за винятком того, що перед подальшим просуванням по лінії зв'язку пакети цілком не зберігаються. Як тільки черговий байт пакета прибуває на транзитний вузол, він передається уздовж маршруту далі, не чекаючи одержання всього пакета. Допускається як динамічна (адаптивна), так і статична (фіксована) маршрутизація. Для реалізації віртуальної наскрізної маршрутизації на мікросхемі є кілька спеціалізованих пристроїв.

На додаток до основного тривимірного тору, що забезпечує обмін даними, є й інші комунікаційні мережі. Друга (колективна) мережа має дерево-

подібну структуру. У системах з високим ступенем паралелізму, таких як BlueGene/P, для виконання багатьох операцій потрібує участі всіх вузлів, прикладом може слугувати підсумовування чисел методом здвоювання або пошук найменшого значення в масиві. При такому підході в кореневий вузол потрапляє лише необхідний мінімум інформації.

Деякі алгоритми вимагають поетапного виконання, коли кожний вузол, закінчивши свій етап, не переходить до наступного, а очікує, поки той же етап закінчать всі інші. Третя – особлива *бар'єрна мережа* дозволяє програмно задавати ці етапи й припиняти обчислення на всіх процесорах, що завершили свій етап раніш за інших. Коли всі процесори завершують свій етап, обчислення тривають далі. Та ж сама бар'єрна мережа використовується для переривань.

Четверта й п'ята мережі використовують технологію Ethernet 10 Гіб. Одна з них з'єднує вузли введення/виведення з файловими серверами, що не входять у систему BlueGene/P, а також з Інтернетом; інша використовується для налагодження системи.

На кожному обчислювальному і комунікаційному вузлі працює спеціалізована мала операційна система, що підтримує одного користувача й один процес. Процес може мати до чотирьох програмних потоків, по одному на кожний процесор у вузлі. Ця проста структура була обрана через її високу продуктивність і надійність.

Для підвищення надійності прикладна програма може створити точку збереження, викликавши відповідну бібліотечну процедуру. Після того як у мережі закінчиться передача всіх ще не переданих повідомлень, можна створити глобальну точку збереження, щоб при збою системи завдання можна було б запустити з цієї точці, а не з самого початку. Вузли введення/виведення працюють під керуванням традиційної ОС Linux і підтримують багатозадачність.

За даними на 2015 рік уже розроблений і випускається комп'ютер третього поколінням – BlueGene/Q. Ця система містить 18 процесорів на мікро-

схему й підтримує паралельну багатопоточність. Ці дві особливості значно збільшують продуктивність системи в кількості команд на цикл. Система може сягати продуктивності до 20 петафлоп/с.

При всіх своїх перевагах, системи MPP і великі суперкомп'ютери, і офісні малогабаритні системи володіють одним, але істотним недоліком – досить високою вартістю.

1.2.2 Кластери робочих станцій

Інший варіант мультикомп'ютерів хоча і значно не такий продуктивний, як системи MPP, але й набагато-набагато дешевший за них – це системи COW (Cluster of Workstations – ***кластер робочих станцій***) або NOW (Network of Workstations – ***мережа робочих станцій***). Як правило, кластер складається з декількох сотень, а може навіть тисяч зв'язаних звичайною локальною мережею персональних комп'ютерів або робочих станцій, з підключенням до мережі через звичайну мережну плату. Розходження між MPP і кластером таке ж, як між мейнфреймом і персональним комп'ютером. В обох є процесор, ОЗУ, диски, операційна система й т.д. Але в мейнфреймі все це (за винятком, мабуть, операційної системи) працює набагато швидше, і через це застосовуються й управляються вони зовсім по-різному. Те ж саме можна сказати про MPP і кластери.

Ще кілька років назад взаємодія між елементами, які утворюють MPP, відбувалася набагато швидше, ніж між машинами, з яких складається кластер. Однак з розвитком мережних технологій і з появою на ринку високошвидкісних мереж ця відмінність почала поступово згладжуватися. Головна ж перевага системи COW над MPP полягає в тім, що COW повністю складається з доступних компонентів, які можна легко купити. Ці компоненти випускаються великими партіями. Крім того, вони існують на ринку із жорсткою конкуренцією, через яку продуктивність зростає, а ціни падають. Імовірно, системи COW у багатьох галузях поступово витиснуть MPP, подібно до того,

як персональні комп'ютери витиснули більші обчислювальні машини, які застосовуються тепер тільки в спеціалізованих областях. Основною нішею для систем MPP залишаться дорогі суперкомп'ютери, у яких головне – продуктивність, а питання вартості не мають вирішального значення.

Існує безліч різних видів COW, але домінують два з них: *централізовані* й *децентралізовані*. ***Централізовані системи*** COW являють з себе кластер робочих станцій або персональних комп'ютерів, змонтованих у великий блок в одному приміщенні (чим не MPP). Іноді вони компонуються більш компактно, ніж звичайно, щоб скоротити фізичні розміри й довжину мережних кабелів. Як правило, ці машини гомогенні й не мають ніяких периферичних пристроїв, крім мережних плат і, можливо, дисків. Гордон Белл (Gordon Bell), розроблювач PDP-11 і VAX, назвав такі машини «*автономними робочими станціями*» оскільки вони не мають власників.

Децентралізована система COW складається з робочих станцій або персональних комп'ютерів, які розкидані по будинку або по території деякої установи. Більшість із них простоює багато годин на протязі доби, особливо вночі. Звичайно вони зв'язані через локальну мережу. Вони гетерогенні й мають повний набір периферійних пристроїв, хоча кластер, який має тисячу маніпуляторів типу миша нічим не краще за кластер взагалі без мишей. Найважливіша відмінність децентралізованої системи COW від локальної обчислювальної мережі, яка з'єднує користувальницькі машини, не має ніякого відношення до апаратного забезпечення й полягає в тому, як використовуються комп'ютери в системі. У локальній мережі користувачі працюють із *персональними* машинами й використовують їх тільки для своєї роботи. Децентралізована ж система COW, навпроти, є загальним ресурсом, якому користувачі можуть доручити роботу, що вимагає декількох процесорів для її виконання. Але справа в тому, що комп'ютери, які входять у децентралізований кластер мають власників, кожний з яких душі не чує у своєму ПК і не занадто лояльно ставиться до того, що хтось сторонній намагається використовувати їх комп'ютера для цілей обчислення, які «не стосуються справ». Якщо ж вико-

ристовувати для організації кластера тільки машини, які є бездіяльними на даний момент, обов'язково виникає потрібність у якомусь механізмі міграції завдань, щоб звільнити машину, коли вона знадобиться своєму власникові. Хоча проблема міграції завдань цілком розв'язувана, рішення вимагає додаткового ускладнення програмного забезпечення

Крім того щоб система COW могла обробляти запити від декількох користувачів, кожному з яких потрібно кілька процесорів, до програмного забезпечення обов'язково повинен входити *планувальник завдань*. Є кілька алгоритмів роботи такого планувальника.

Найпростіша модель планування передбачає, що заздалегідь відома лише тільки кількість процесорів, яка необхідна для виконання кожної роботи (завдання). В цьому випадку організується звичайна черга типу FIFO («першим увійшов – першим вийшов») і завдання добавляються до неї у порядку надходження (рис. 1.26 а). Коли перше завдання починає виконуватися, відбувається перевірка, чи є достатня кількість процесорів для виконання наступного завдання з черги. Якщо так, то воно теж починає виконуватися й т.д. Якщо ні, то система чекає, поки не визволиться достатня кількість процесорів. На рисунку якась система COW, як приклад, містить 8 процесорів, але вона цілком могла б містити 128 процесорів, розташованих у блоках по 16 процесорів (вийшло б 8 груп процесорів) або в якій-небудь іншій комбінації.

У більш досконало розробленому алгоритмі, задачі, які не відповідають кількості наявних процесорів, пропускаються й з черги береться перше завдання, для якого процесорів достатньо. Щораз, коли завершується виконання якогось завдання, черга із завдань, що залишилися, перевіряється в порядку «першим увійшов – першим вийшов». Результат застосування цього алгоритму зображений на рис. 1.26 б).

Ще більш складний алгоритм вимагає, щоб заздалегідь було відомо, скільки процесорів потрібно для кожного завдання й скільки часу займе його виконання. Маючи у своєму розпорядженні таку інформацію, планувальник завдань може спробувати найкращим чином заповнити прямокутник «проце-

сори/час». Це особливо ефективно, коли завдання представляються на розгляд удень, а виконуватися будуть уночі. У цьому випадку планувальник одержує всю інформацію про завдання заздалегідь і може виконувати їх в оптимальному порядку, як показано на рис. 1.26 в).

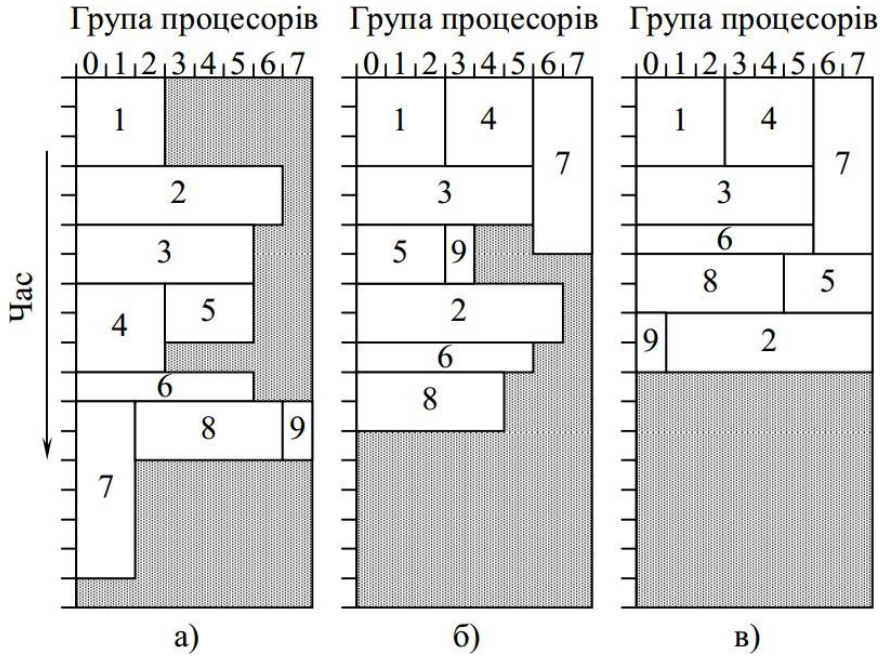


Рисунок 1.26 − Планування роботи в системі COW
а) – FIFO, б) – планування без блокування початку черги,
в) – заповнення прямокутника «процесори/час»

Кластери найчастіше невеликі – у межах від дюжини до декількох сотень комп’ютерів. Проте, можна побудувати дуже великий кластер зі звичайних ПК. Прикладом такого кластера може слугувати кластер, що розробила компанія Google, і архітектуру якого має сенс розглянути докладніше.

Google – популярна система пошуку інформації в Інтернеті. З погляду пошукової системи завдання полягає в тім, щоб проіндексувати і зберегти всю Всесвітню Павутину (а це більше 8 мільярдів сторінок і одного мільярда зображень), а потім знаходити серед збереженої інформації потрібну сторінку за незначні частки секунди, обслуговуючи тисячі запитів у секунду, які цілодобово приходять із всіх кінців світу. На додаток система ніколи не повинна

відключатися, навіть у випадках природних катаклізмів, перебоїв в електроживленні, у роботі мережі, апаратних і програмних збоїв.

Функціонування Google забезпечують безліч інформаційних центрів по усьому світі. Це не тільки уможливлює підміну на випадок раптової повені або землетрусу. При звертанні до адреси www.Google.com аналізується IP-адреса відправника запиту, і далі браузер спілкується вже тільки з найближчим до нього інформаційним центром.

Кожний інформаційний центр підключається до Інтернету як мінімум однією оптоволоконною лінією OC-48 (2,488 Гіб/с), по якій надходять запити й відправляються сформовані відповіді. Крім того, є додаткова лінія OC-12 (622 Міб/с) до резервного постачальника послуг на випадок перерви в роботі основного. Щоб робота могла тривати й при перебоях в електропостачанні, у всіх центрах є джерела безперебійного живлення й аварійні дизельні генератори. Таким чином, під час природних катаклізмів робота Google не порушиться, хоча продуктивність і знизиться.

Для більшого розуміння архітектури кластера Google, корисно познайомитися з механізмом обробки запиту, який надійшов в один з інформаційних центрів. Надійшовши до центру (крок 1 на рис. 1.27), запит переправляється вирівнювачем навантаження до одного з численних оброблювачів запитів (2), а також, паралельно, у систему перевірки правопису (3) і сервер контекстної реклами (4). Паралельно виконується пошук запитаного слова на індексних серверах (5), на яких зберігаються записи мабуть про кожне слово в Мережі.

У кожному такому записі перераховані всі наявні документи, які містять це слово (це можуть бути веб-сторінки, PDF-файли, презентації PowerPoint і т.д.). Посилання в цих списках розташовані відповідно до рейтингу сторінки – параметра, який обчислюється за складною формулою. Принцип обчислення рейтингу тримається в таємниці, але відомо, що велике значення має кількість посилань на сторінку й рейтинги сторінок, які саме посилаються на неї.

Для підвищення продуктивності індекс розбивається на фрагменти, пошук яких ведеться паралельно. Відповідно до цієї ідеї, перший фрагмент містить всі слова з індексу, і кожному слову зіставлені ідентифікатори *n* перших за рейтингом сторінок. Другий фрагмент містить всі слова й ідентифікатори *n* наступних по рейтингу сторінок і т.д. За мірою розростання Мережі, кожний із цих фрагментів можна додатково розділити на кілька частин так, що в першій частині будуть перші *k* слів, у другий – наступні *k* і т.д. Це дозволяє досягати ще більшого паралелізму при пошуку.

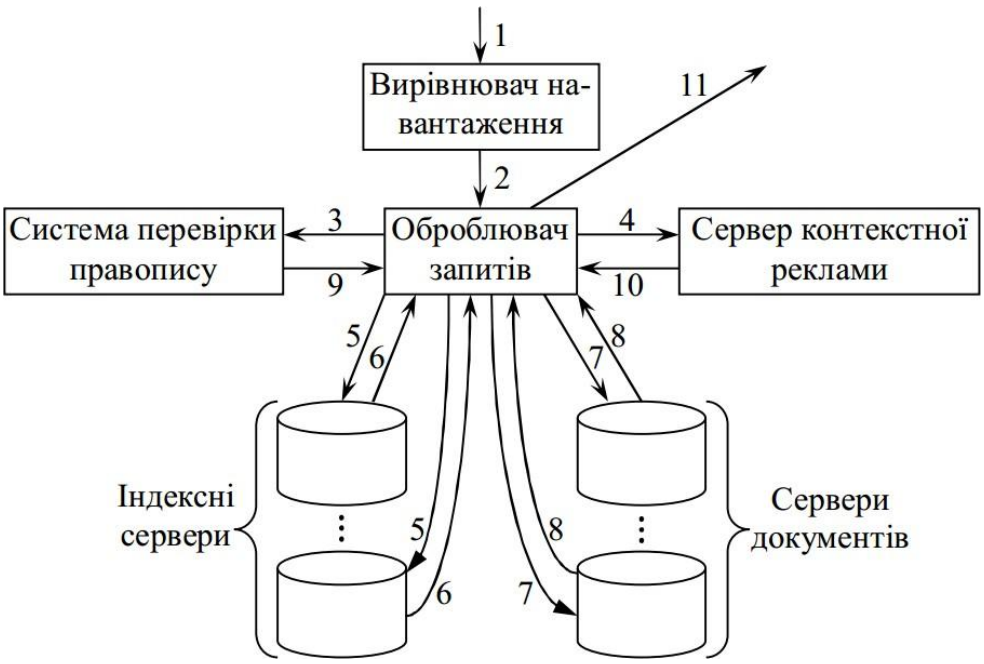


Рисунок 1.27 − Схема обробки запиту в Google

Індексні сервери повертають набори ідентифікаторів документів (6), які потім комбінуються відповідно до логіки запиту. Наприклад:

```
+digita +kapibara +dance
```

При такому запиті на наступний крок потраплять ідентифікатори тільки тих документів, які є у всіх трьох наборах. На цьому кроці Google звертається до самих документів (7), витягаючи з них назви, посилання, а також фрагменти тексту, який оточує запитані слова. Копії багатьох документів Мережі збе-

рігаються на серверах документів всіх інформаційних центрів, на сьогодення їхній обсяг сягає сотень тебібайтів. Для прискорення паралельного пошуку документи також поділяються на фрагменти. У підсумку, хоча для обробки запиту й не потрібно зчитувати весь уміст Мережі (і обробляти десятки тебібайтів індексів), при обслуговуванні рядового запиту все ж таки доводиться «переворушити» не менш 100 МіБ даних.

Після того як результати вертаються оброблювачеві запиту (8), вони поєднуються відповідно до рейтингу сторінок. Додається інформація про можливі помилки правопису, якщо вони виявлені (9), і контекстна реклама (10). Включення в результати запиту тих або інших ключових слів, куплених рекламодавцями (наприклад, «Готель», або «Відеокамера»). Нарешті, результати оформлюються у форматі HTML документу і повертаються користувачеві (11) у вигляді звичайної веб-сторінки.

Усвідомивши весь обсяг роботи, який необхідно виконати для обслуговування навіть простого запиту, і зрозумівши, чому Google знадобився потужний кластер, можна розглянути й архітектуру цього кластера. Більшість компаній, зіштовхуючись із необхідністю підтримувати величезну і надійну базу даних з колосальною кількістю транзакцій, намагаються закупити найшвидше й найнадійніше (тому найдорожче) устаткування, яке наявне на ринку. В Google до цього питання підійшли з точністю до навпаки. Вони купили дешеві персональні комп’ютери із середньою продуктивністю. Об’єднавши ці машини, побудували найбільший у світі кластер зі звичайних компонентів. Головний принцип, який лежить в основі цього рішення, проста – оптимізація відношення ціна/продуктивність. Тобто корені ухваленого рішення лежать в економіці: звичайні персональні комп’ютери досить дешеві. Для висококласних серверів це не так, а для великих мультипроцесорів й тим паче не так. Приміром, продуктивність потужного сервера може в 2-3 рази перевищувати продуктивність середнього ПК, а от коштує він звичайно не в 2-3, а в 5-10 разів більше.

Звичайно ж, дешевий персональний комп’ютер набагато менш надій-

ний, чим кращі моделі серверів, але й сервери іноді «падають». Тому програмне забезпечення Google написане так, щоб надійно працювати на ненадійному апаратному забезпеченні. Маючи у своєму розпорядженні стійке до відмов програмне забезпечення, уже не має великого значення, яка інтенсивність відмов, 0,5 або 2% у рік. Досвід Google показує, що за рік ламаються 2% всіх комп'ютерів. Більш половини відмов викликані жорсткими дисками, наступна кількість відмов пов'язана з блоками живлення, а за ними ідуть мікросхеми пам'яті. Процесори, після обкатування, взагалі не ламаються. У дійсності, основною причиною збоїв є не апаратне, а програмне забезпечення. Тому першою реакцією на помилку є перезавантаження комп'ютера, що у більшості випадків вирішує проблему.

За наявними даними, у ПК, які використовуються в Google, установлений процесор Pentium з тактовою частотою 2 ГГц, 512 МіБ оперативної й 80 ГіБ дискової пам'яті. Комп'ютери, включаючи мікросхему Ethernet, явно не вищого класу, але зате доволі дешеві. Комп'ютери розміщаються в корпусах висотою близько 5-ти сантиметрів і встановлюються в стійки, по 40 штук з двох боків стійки – попереду й позаду. В одній стійці, таким чином, установлюються 80 машин, які підключаються до Ethernet за допомогою комутатора усередині стійки. Всі стійки в одному інформаційному центрі також підключені до Ethernet через комутатор, а для живучості при збоях є 2 надлишкових комутатори.

Структура типового інформаційного центра Google показана на рис. 1.28.

Дані з високошвидкісної оптоволоконної лінії OC-48 надходять на два 128-портових Ethernet-комутатори. До них же підключена й резервна лінія OC-12. Щоб вхідні канали не займали порти Ethernet-комутаторів застосовується спеціальна плата. З кожної стійки виходить чотири Ethernet-лінії, дві до комутатора, який показаний на рисунку ліворуч, і дві праворуч. Завдяки цьому система може пережити відмову кожного із двох комутаторів. Завдяки наявності чотирьох ліній, для втрати зв'язку зі стійкою необхідно, щоб одноча-

сно вийшли з ладу або всі чотири лінії, або дві лінії й комутатор. Маючи пару комутаторів на 128 портів і стійки із чотирма лініями, можна з'єднати в мережу 64 стійки. Якщо вважати, що в стійці 80 комп'ютерів, це сумарно дає 5120 машин, хоча, звичайно ж, ніхто не вимагає, щоб у стійці було саме 80 машин, та й у комутаторів може бути більше 128 портів. Просто така кількість обладнання є характерною для кластера Google.

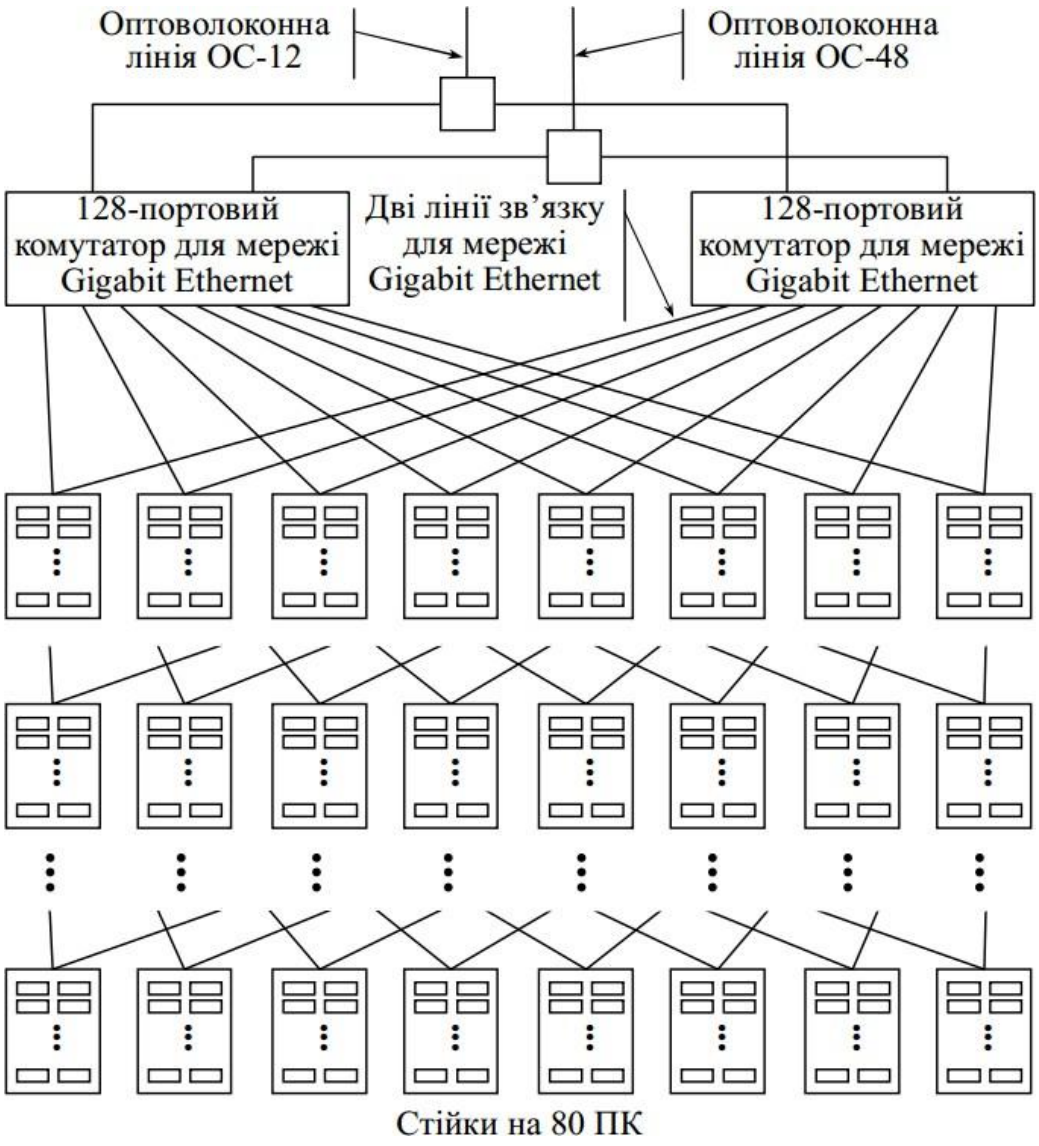


Рисунок 1.28 − Типовий кластер Google

Велике значення має також питоме енергоспоживання на одиницю площі. Типовий персональний комп'ютер споживає близько 120 Вт, що дає 10

кВт на стійку. Щоб обслуговуючий персонал міг установлювати в стійку й витягати зі стійки комп'ютери, для неї потрібно не менш 3 $м^2$ простору.

Таким чином, питоме енергоспоживання становить 3000 $Вт/м^2$. При проектуванні звичайних інформаційних центрів, до розрахунків площі необхідного приміщення закладають питоме енергоспоживання від 600 до 1200 $Вт/м^2$, тому в цьому разі потрібно вживати спеціальних заходів для охолодження.

В Google добре засвоїли три правила створення й використання великих веб-серверів:

1) будь-які компоненти ламаються, і це треба враховувати;
2) для підвищення пропускної здатності й доступності все повинне дублюватися;
3) необхідно оптимізувати співвідношення ціна/продуктивність.

Перший пункт говорить, що програмне забезпечення повинне бути стійким до відмов. Навіть найкраще устаткування рано чи пізно зламається, якщо його багато, і треба програмно враховувати цю можливість. Система такого розміру повинна переживати відмови, навіть якщо відбуваються вони кілька разів у тиждень.

Другий пункт указує на те, що й програмне, і апаратне забезпечення повинні мати надмірність. Це не тільки робить систему стійкою до відмов, але й підвищує пропускну здатність. У випадку Google самі комп'ютери, їхні диски, кабелі, блоки живлення й комутатори багаторазово дублюються. У межах одного центра дублюються фрагменти індексів і документів. Дублюються й самі інформаційні центри.

Третій пункт є наслідком перших двох. Якщо система належним чином реагує на збої, нерозумно купувати дорогі компоненти, такі як RAID-масиви або SCSI-диски. Навіть вони ламаються, а витрачати в десять разів більше, щоб удвічі знизити інтенсивність відмов – погана ідея. Краще купити в десять разів більше встаткування й передбачити можливість відмов. Зрештою, чим більше встаткування, тим вище продуктивність (за умови, що обладнан-

ня працює).

1.3 Способи передавання даних і типи паралелізму

Розробка паралельних процедур неможлива без урахування особливостей архітектури мультипроцесорного обчислювального комплексу (МОК) і алгоритму вирішення конкретного завдання. Однак серед усього різноманіття способів організації паралельних обчислень звичайно виділяють кілька наступних основних типів паралелізму.

Поточний паралелізм визначає незалежне виконання процесів при взаємодії через загальну пам'ять. Кожний процес виконується так, ніби він працює один. При цьому, якщо в якийсь момент часу t_i один процес привласнює змінній x деяке значення a, то в будь-який інший момент t_j, перед яким немає звертання до x, інший процес зчитує саме значення a. Цей тип паралельності характерний для мультипроцесорів із загальною пам'яттю.

Як приклад цього типу паралелізму можна розглянути наступне завдання: необхідно обчислити $X_k = k*X_k + k$, при $k=1,\ldots,n$

На однопроцесорному комп'ютері цей розрахунок може бути виконаний за допомогою циклу:

```
for(k=1; k<=n; k++) X(k) = k * X(k) + k;
```

На двопроцесорній системі це можна зробити за допомогою наступних двох циклів, які працюють паралельно і другий цикл затриманий відносно першого на один такт (за такт приймається середній час виконання однієї арифметичної операції):

```
for(k=1; k<=n; k++) X(k) = k * X(k);     //1
for(k=1; k<=n; k++) X(k) = X(k) + k;     //2
```

У першому випадку буде потрібно $2n$ тактів роботи, у другому $n+1$ такт.

Автономний паралелізм відрізняється від поточного паралелізму тим, що всі змінні локалізовані у своїх процесах. Виконання процесів відбувається

незалежно, і якщо один процес виконує присвоювання *x* = *a*, а інший зчитує *x*, то ці *x* різні. Для обміну даними в цьому випадку використовуються спеціальні процедури обміну. Цей паралелізм орієнтований на мулитикомпьютери з розділеною пам'яттю.

Одночасний паралелізм використовується для виконання циклів команд над великими обсягами даних. При цьому цикл реалізується в такий спосіб. Спочатку у всіх процесах зчитується інформація для виконання перших команд, і одночасно у всіх процесах виконуються ці команди. Потім одночасно розсилаються отримані дані, але при цьому необхідно забезпечити, щоб два процеси не робили засилання в ту саму змінну. Після цього дії повторюються для наступної команди. Цей паралелізм найбільш пристосований для застосування на матричних суперкомп'ютерах.

Паралелізм із синхронізацією може бути використаний у тих випадках, коли тіло паралельного циклу також є циклом, але послідовним. При цьому, у кожному паралельному процесі може бути виконана, незалежно від інших процесів, деяка послідовність операцій («***кортеж***»). Після цього потрібно деяка синхронізація операцій, які виконуються у процесах, без синхронізації забезпечити правильний результат неможливо.

Прикладом може слугувати наступний фрагмент:

```
DO I=1,16
  X(I) = X(6*I - 27)
ENDDO
```

Всі операції цього циклу не можна виконати незалежно. Наприклад, 5-а ітерація залежить від 3-ої. Тоді їх можна виконувати кортежами по 4:

```
X(1) = X(-21)  X(5) = X(3)   X(9) = X(27)   X(13) = X(51)
X(2) = X(-15)  X(6) = X(9)   X(10) = X(33)  X(14) = X(57)
X(3) = X(-9)   X(7) = X(15)  X(11) = X(39)  X(15) = X(63)
X(4) = X(-3)   X(8) = X(21)  X(12) = X(45)  X(16) = X(69)
```

Операції в кожному кортежі можуть виконуватися паралельно, після чого потрібна синхронізація, тобто обмін даними, і виконується наступний кор-

теж.

Ексклюзивний паралелізм може бути використаний у тих випадках, коли всі або деяка частина обчислень може бути виконана в довільному порядку. Він найбільш характерний для завдань авторегулювання.

Очевидно, що всі перераховані види паралелізму можуть використовуватися в різних несуперечливих сполученнях.

1.4 Комутація й синхронізація в розподілених системах

Всі комп'ютери в розподіленої системі зв'язані між собою комунікаційною мережею. Комунікаційні мережі складаються з комунікаційних комп'ютерів, зв'язаних між собою комунікаційними лініями, які забезпечують транспортування повідомлень. Звичайно використовується процедура ***store-and-forward,*** коли наступні одне за одним повідомлення передаються від одного комп'ютера до іншого із проміжним зберіганням.

У розподілених обчислювальних системах використовуються також традиційні способи комутації: комутація пакетів і комутація каналів.

Як було сказано раніше, комутація каналів вимагає резервування ліній на час усього сеансу взаємодії двох пристроїв.

Розглянута вище, пакетна комутація заснована на розбивці повідомлень у пункті відправлення на порції (пакети), посилці пакетів за адресою призначення, і складанню повідомлення з пакетів у пункті призначення. При цьому лінії використовуються ефективніше, повідомлення можуть передаватися швидше, але при цьому потрібно виконати дії по розбивці й складанню повідомлень.

Синхронізація в розподілених системах здійснюється на основі визначення для всіх подій відношення попередування. По суті, вона полягає у визначенні черговості доступу процесів до поділюваних ресурсів.

Паралельний процес, який володіє поділюваним ресурсом, виключає для інших процесів можливість одночасного з ним звертання до тих же ре-

сурсів. Це називається ***взаємовиключенням***.

Говорять, що процес перебуває на своїй критичній ділянці, якщо він робить звертання до поділюваних ресурсів. Для забезпечення ***взаємовиключення*** необхідно при входженні одного процесу у свою критичну ділянку, не допускати можливості такого входження для всіх інших пов'язаних з ним процесів. Звичайно, при цьому інші процеси можуть продовжувати своє виконання, але без входу у свої критичні ділянки.

Кожний процес повинен максимально швидко проходити свою критичну ділянку, звільняючи доступ до ресурсу іншим процесам. Саме тому слід виключати блокування й зациклення процесу усередині критичної ділянки.

Існує безліч алгоритмів синхронізації. Нижче наведений опис деяких з них.

Для багатьох алгоритмів розподіленої обробки даних потрібно, щоб один із процесів реалізовував функції координатора, виконуючи синхронізацію процесів, це так званий ***алгоритм із процесом-координатором***. При цьому дуже часто буває неважливо, який саме процес обраний як координатор. Можна вважати, що звичайно вибирається процес із самим більшим унікальним номером.

Всі процеси запитують у координатора дозвіл на вхід у критичну ділянку й чекають цього дозволу. Координатор обслуговує запити в порядку надходження (*FIFO*). Одержавши дозвіл, процес входить у критичну ділянку. При виході з неї процесор сповіщає про це координаторові.

Недоліки цього алгоритму є звичайними для алгоритму централізованого типу. Це проблеми при перевантаженні координатора повідомленнями або при його відмові.

В іншому алгоритмі – ***алгоритмі із круговим маркером*** всі процеси утворюють логічне коло з попередньо визначеним порядком опитування процесів. По колу циркулює маркер, який дає право на вхід процесу у критичну ділянку. Одержавши маркер (за допомогою повідомлення точка-точка) процес або входить у критичну ділянку (якщо він чекав дозволу), або переправляє

маркер далі. Після виходу із критичної секції маркер переправляється далі, повторний вхід у секцію при тому ж маркері не дозволяється.

До недоліків алгоритму відносяться:

- складність виявлення втрати маркера й необхідність його регенерації;
- припинення будь-якого процесу в системі приводять до збою алгоритму. Однак відновлення виробляється доволі просто з використанням квитанцій, які дозволяють виявити припинений процес у момент передачі маркера. Припинений процес виключається при цьому з логічного кола.

1.5 Програмування паралельних обчислень на неоднорідних мережах комп'ютерів на мові mpC

Чисельні методи у випадку багатопроцесорних систем повинні проектуватися як системи паралельних і взаємодіючих між собою процесів, які допускають виконання на незалежних процесорах. Застосовувані алгоритмічні мови й системне програмне забезпечення повинні забезпечувати створення паралельних програм, організовувати синхронізацію й взаємовиключення асинхронних процесів і т.п.

На сьогодні мережі комп'ютерів – кластери є самою доступною й розповсюдженою паралельною архітектурою й, найчастіше, необхідне прискорення вирішення тих або інших завдань може бути досягнуто не шляхом придбання нового могутнішого комп'ютера, а за рахунок використання обчислювального потенціалу вже наявних комп'ютерів, пов'язаних з допомогою сучасного мережного устаткування.

Але на відміну від суперкомп'ютерів, мережі по своїй природі гетерогенні й складаються з різноманітних комп'ютерів найчастіше різної продуктивності, зв'язаних у мережу іноді неоднорідним мережним устаткуванням, що надає різну швидкість обміну даними між різними процесорними вузлами. Тому продуктивність паралельних обчислень із використанням звичайних

засобів програмування для SMP і MPP на такому кластері обмежується першим законом Амдала. Пов'язане це з тим, що паралельні програми розподіляють дані, обчислення й комунікації без урахування розходжень у продуктивності процесорів і комунікаційних каналів.

Для усунення цього недоліку наприкінці 90-х років минулого століття в Інституті системного програмування РАН була розроблена паралельна мова програмування, яка спеціально спроектована для програмування неоднорідних мереж, а також створена підтримуюча її система програмування. Ця мова, названа *mpC*, являє собою розширення стандарту ANSI для мови *C*.

Основна ідея, яка лежить в основі мови *mpC*, полягає в наданні користувачеві мовних конструкцій, що дозволяють визначати абстрактну неоднорідну паралельну машину, найбільш підходящу для виконання алгоритму. Інформація про таку абстрактну машину разом з інформацією про фізичну паралельну систему використовується системою програмування *mpC* для забезпечення ефективного виконання відповідної програми на цій фізичній паралельній системі.

Вся семантика мови *mpC* будується на понятті ***обчислювального простору***, який визначається як деяке доступне для керування безлічі віртуальних процесорів різної продуктивності, з'єднаних каналами зв'язку з різною швидкістю передачі.

При цьому основним поняттям *mpC* є поняття ***мережного об'єкта*** або просто ***мережі***. *Мережа є областю обчислювального простору*, яка може бути використана для обчислення виразів і виконання операторів і складається з тих самих віртуальних процесорів і каналів комунікації.

Розміщення й звільнення мережних об'єктів в обчислювальному просторі виконується в манері, яка аналогічна розміщенню і звільненню екземплярів об'єктів класів у пам'яті в мові C++. Концептуально, створення нової мережі ініціюється яким-небудь процесором із уже існуючої мережі. Цей процесор називається ***батьківським процесом створюваної мережі***. Батько завжди належить створюваній мережі. Єдиним виділеним процесором, від

початку й до кінця виконання програми, є заздалегідь визначений віртуальний ***хост-процесор***.

Будь-який мережний об'єкт, оголошений у програмі, має тип. Тип специфікує кількість, типи й продуктивності процесорів, каналів зв'язку між процесорами і їхню швидкість, а також батька мережі. Наприклад, оголошення (фрагмент 1.1):

Фрагмент коду 1.1

```
/* Рядок 1 */ nettype Rectangle {
/* Рядок 2 */   coord I = 4;
/* Рядок 3 */   node { I >= 0: I + 1; };
/* рядок 4 */   link {
/* Рядок 5 */     I > 0: [I] <-> [I-1];
/* Рядок 6 */     I == 0: [I] <-> [3];
/* Рядок 7 */   };
/* Рядок 8 */   parent [0];
/* Рядок 9 */ }
```

уводить мережний тип з ім'ям *Rectangle*, що відповідає мережам, які складаються із чотирьох віртуальних процесорів різної продуктивності, зв'язаних у прямокутник ненаправленими каналами зі звичайною швидкістю.

Тут рядок 1 містить заголовок оголошення мережного типу. Він вводить ім'я мережного типу.

Рядок 2 містить оголошення системи координат, до якої прив'язуються процесори. Він вводить цілочисельну координатну змінну *I* зі значеннями від 0 до 3.

Рядок 3 містить оголошення процесорних вузлів, тим самим прив'язуючи процесори до системи координат і повідомляючи їхні типи й продуктивності. Рядок 3 відповідає наступному предикату:

> *Для всіх I<4 якщо I>=0, то віртуальний процесор з відносною продуктивністю, обумовленою величиною I+1*, прив'язується *до точки з координатою [I]*.

Вираз *I*+1 називається ***специфікатором продуктивності***. Більше число задає більшу продуктивність. У наведеному прикладі 0-й віртуальний процесор у два рази повільніший ніж 1-й, у три рази повільніший ніж 2-й і в чотири рази повільніший ніж 3-й віртуальний процесор. Для будь-якої мережі такого типу ця інформація дозволяє компіляторові приписати кожному віртуальному процесору його вагу, яка нормалізована щодо батьківського вузла мережі.

Рядки 4-7 специфікують канали зв'язку між процесорами. Рядок 5 відповідає предикату:

Для всіх I<4 якщо I>0 то є ненаправлений канал зі звичайною швидкістю, що зв'язує процесори з координатами [I] і [I-1],

а рядок 6 відповідає предикату:

Для всіх I<4 якщо J==0 то є ненаправлений канал зі звичайною швидкістю, що зв'язує процесори з координатами [I] і [3].

За замовчуванням якщо канал між двома процесорами не специфікований явно, то це означає наявність між ними каналу з мінімальною для даної мережі швидкістю.

Рядок 8 містить оголошення батька й указує, що батьківський процесор має координату [0] у створюваній мережі.

Маючи оголошення мережного типу, можна оголосити ідентифікатор мережного об'єкта цього типу. Наприклад, оголошення:

```
net Rectangle rl;
```

уводить ідентифікатор *rl* мережного об'єкта типу *Rectangle.*

Поняття розподіленого об'єкта вводиться в стилі мов C* і Dataparallel C. За визначенням, об'єкт даних, розподілений по деякій області обчислювального простору, *складається зі звичайних* (нерозподілених) *об'єктів* одного типу, називаних ***компонентами розподіленого об'єкта даних*** і розміщених у

процесорних вузлах цієї області таким чином, що *кожний процесорний вузол містить один й тільки один компонент*. Наприклад, наступні оголошення:

Фрагмент коду 1.2

```
net Rectangle r2;
int [*]Derror, [r2]Da[10];
float [host]f, [r2:I<2]Df;
repl [*]di;
```

уводять:

- цілочисельну змінну *Derror*, розподілену по всьому обчислювальному простору;
- масив *Da* з десяти цілих, розподілений по мережі *r2*;
- нерозподілену речовинну змінну *f*, що належить тільки віртуальному хост-процесору;
- дійсну змінну *Df*. розподілену по підмережі мережі *r2* (по двох вузлах);
- цілую змінну *di*, розмазану по всьому обчислювальному просторі.

За визначенням, розподілений об’єкт є розмазаним, якщо всі його компоненти рівні між собою.

Поняття розподіленого значення вводиться аналогічно поняттю розподіленого об’єкта даних.

Крім простого мережного типу, можна оголошувати параметризоване сімейство мережних типів, називане ***топологією або параметризованим мережним типом***. Наприклад, оголошення (фрагмент 1.3)

Фрагмент коду 1.3

```
/* Рядок 1 */ nettype Ring (n, p[n]) {
/* Рядок 2 */   coord I=n;
/* Рядок 3 */   node {
/* Рядок 4 */     I>=0: p[I];
/* Рядок 5 */   };
/* Рядок 6 */   link {
```

```
/* Рядок 7 */     I>0: [I]<->[I-1];
/* Рядок 8 */     I==0: [I]<->[n-1];
/* Рядок 9 */   };
/* Рядок 10*/   parent [0];
/* Рядок 11*/ }
```

уводить топологію з ім'ям *Ring*, що відповідає мережам, які складаються з *n* віртуальних процесорів, зв'язаних у кільце за допомогою ненаправлених каналів з нормальною швидкістю.

Заголовок оголошення (рядок 1) уводить параметри топології *Ring*, а саме, цілий параметр *n* і векторний параметр *p*, який складається з *n* цілих. Відповідно, координатна змінна *I* пробігає значення від 0 до *n*-1, рядок 4 відповідає предикату «*Для всіх I<n якщо I>=0, то віртуальний процесор з відносною продуктивністю, специфікованій значенням p[I], прив'язується до точки з координатами [I]*» і т.п.

Маючи оголошення топології, можна оголосити ідентифікатор мережного об'єкта підходящого типу. Наприклад, фрагмент

```
repl [*]m,[*]n[100];
/* Обчислення m, n[0], n[m-1] */
net Ring(m, n) rr;
```

уводить ідентифікатор *rr* мережного об'єкта, чий тип визначається повністю тільки під час виконання програми. Мережа *rr* складається з *m* віртуальних процесорів. Відносна продуктивність і-*го* віртуального процесора визначається значенням *n*[*i*].

Мережний об'єкт характеризується класом обчислювального простору, який виділений для нього і визначає час його життя. Область обчислювального простору може виділятися статично або динамічно. Обчислювальний простір під статичну мережу приділяється один раз. Будучи створеною, мережа існує до кінця виконання програми. Новий екземпляр мережі, описаної як динамічна, створюється при кожному вході в блок, у якому мережа описана, і знищується при кожному виході із блоку.

Нижче наведена проста *mpC* програма (фрагмент 1.4) множення двох

щільних квадратних матриць *X* і *Y*, яка використовує кілька віртуальних процесорів, кожний з яких обчислює частину рядків результуючої матриці *Z*.

Фрагмент коду 1.4

```
/* 1 */ #include <stdio.h>
/* 2 */ #include <stdlib.h>
/* 3 */ #include <mpc.h>
/* 4 */ #define N 1000
/* 5 */ void [host] Input(), [host] Output ();
/* 6 */ nettype Star(m, n[m]) {
/* 7 */   coord I=m;
/* 8 */   node { I>=0: n[I]; };
/* 9 */   link { I>0: [0]<->[I]; };
/*10 */ };
/*11 */ void [*] main()
/*12 */ {
/*13 */    double [host] x[N][N], [host] в[N][N],
                  [host] z[N][N];
/*14 */    repl int nprocs;
/*15 */    repl double *powers;
/*16 */    Input(x, y);
/*17 */    MPC_Processors(&nprocs, &powers);
/*18 */    {
/*19 */       repl int ns[nprocs];
/*20 */       MPC_Partition_1b(nprocs,powers,ns,N);
/*21 */       {
/*22 */          net Star(nprocs, ns) w;
/*23 */          int [w]myn;
/*24 */          myn=([w]ns) [I coordof myn];
/*25 */          {
/*26 */             repl int [w] i, [w] j;
/*27 */             double [w] dx[myn][N],
                  [w] dy[N][N], [w] dz[myn][N);
/*28 */             dy[] = y[];
/*29 */             dx[] =::x[];
/*30 */             for(i=0; i<myn; i++)
/*31 */               for (j=0; j<N; j++)
/*32 */                 dz[i][j]=[+](dx[i][]*
                    (double[*][N:N])(dy[0]+j)[]);
/*33 */             z[]::=dz[];
/*34 */          }
/*35 */       }
/*36 */       Output(z);
/*37 */    }
/*38 */ }
```

Програма містить у собі 5 функцій; *main*, наведену вище, *Input* і *Output*,

які визначені в інших вихідних файлах, і бібліотечні функції *MPC_Processors* і *MPC_Partition_1b*. Функції *Input* і *Output* оголошуються в рядку 5, а функції *MPC_Processors* і *MPC_Partition_1b* оголошуються у файлі mpC.h.

У загальному випадку. *mpC* допускає використання 3-х класів функцій. У цьому прикладі використовуються функції всіх трьох видів: *main* відноситься до класу ***базових функцій***. *Input* і *Output* до класу ***мережних функцій*** і *MPC_Processors* і *MPC_Partition_1b* до класу ***вузлових функцій***.

Виклик базової функції завжди є колективним виразом (тобто він обчислюється на всьому обчислювальному просторі одночасно; ніякі інші обчислення не можуть виконуватися паралельно з обчисленням колективного виразу). Його аргументи, якщо такі є, або належать хост-процесору, або розподілені по всьому обчислювальному простору, а повертається значення (якщо воно є), яке розподілено по всьому обчислювальному простору. На відміну від інших видів функцій, базова функція може містити визначення мереж. У рядку 11 конструкція *[*]*, яка поміщена перед ідентифікатором функції *main*, указує, що оголошується ідентифікатор базової функції.

Вузлова функція може бути повністю виконана на будь-якому одному процесорі обчислювального простору. У ній можуть створюватися тільки локальні об’єкти даних віртуального процесора, на якому вона викликається, і крім них можуть ще використовуватися компоненти зовнішніх об’єктів даних, що належать цьому процесору. Оголошення ідентифікатора вузлової функції не вимагає ніяких додаткових специфікаторів. З погляду *mpC* всі звичайні функції мови С є вузловими.

У загальному випадку, *мережна функція* викликається й виконується на деякій області обчислювального простору, і її аргументи й значення, яке повертається, також розподілені по цій же області. Дві мережні функції можуть виконуватися паралельно, якщо області, на яких вони викликані, не перетинаються. Функції *Input* і *Output* являють собою найпростішу форму мережної функції, яка може бути викликана тільки на статично визначеній області обчислювального простору. Вони оголошені в рядку 5 як мережні функції, які

можуть бути викликані й виконані тільки на віртуальному хост-процесорі (це задається конструкцією *[host]*, яка поміщена безпосередньо перед ідентифікаторами функцій). Тому виклики функції в рядках 16 і 36 виконуються на віртуальному хост-процессоре.

Рядки 11-38 містять визначення функції *main*. Рядок 13 містить визначення масивів *x*, *y* і *z*, які належать віртуальному хост-процессору.

Рядок 14 визначає цілу змінну *nprocs*, розмазану по всьому обчислювальному простору. Її розподіл визначається правилом умовчання без допомоги конструкції *[*]*. У загальному випадку, розподіл за замовчуванням задається розподілом найменшого блоку, який охоплює відповідне оголошення й має явно визначений розподіл. Визначення в рядку 14 міститься в тілі функції *main*, для якої явно заданий розподіл по всьому обчислювальному простору.

Рядок 15 визначає розподілену по всьому обчислювальному простору змінну *powers* типу покажчик на дійсне число. Оголошення визначає, що всі розподілені об’єкти даних, на які вказує *powers*, є розмазаними.

У рядку 17 бібліотечна вузлова функція *MPC_Processors*, повертаючи число фізичних процесорів і їхньої продуктивності, викликається на всьому обчислювальному просторі. Таким чином, відразу після цього виклику розмазана змінна *nprocs* містить число фізичних процесорів, а розмазаний масив *powers* містить їхні продуктивності.

Рядок 19 визначає динамічний цілий масив *ns*, розмазаний по всьому обчислювальному простору. Усі компоненти масиву складаються з однакового числа елементів *nprocs*.

У рядку 20 бібліотечна вузлова функція *MPC_Partition_1b* викликається на всьому обчислювальному просторі. Ґрунтуючись на продуктивностях фізичних процесорів, ця функція обчислює, скільки рядків результуючої матриці буде обчислюватися кожним з фізичних процесорів. Таким чином, відразу після цього виклику *ns*[*i*] містить число рядків, яке обчислюється *i*-им фізичним процесором. *MPC_Partition_1b* розбиває дане ціле (N у наведеному вище виклику) на частині відповідно до заданої пропорції.

У рядку 22 визначається автоматична мережа *w*, що складається з *nprocs* віртуальних процесорів. Відносна продуктивність *i*-го віртуального процесора визначається значенням *ns*[*i*]. Таким чином, тип цієї мережі визначається повністю тільки в період виконання. Ця мережа, яка виконує інші обчислення й обміни даними, визначається таким чином, що більш могутній віртуальний процесор отримує більшу кількість рядків матриць для обчислення. Система програмування *mpC* буде забезпечувати оптимальне відображення віртуальних процесорів, які утворюють мережу *w*, у безліч процесів, які надають обчислювальний простір. Таким чином, у точності один процес із процесів, які виконуються на кожному з фізичних процесорів, буде залучений до множення матриць, і чим могутніше буде відповідний фізичний процесор, тим більше число рядків він буде обчислювати.

Рядок 23 визначає змінну *myn*, розподілену по *w*.

Результатом бінарної операції *coordof* у рядку 24 буде ціле значення, розподілене по простору *w*, кожний компонент якого дорівнює значенню координати *I* віртуального процесора, якому цей компонент належить. Операнд праворуч операції присвоювання не обчислюється, а використовується для специфікації області обчислювального простору. Слід відмітити, що координатна змінна *I* інтерпретується як ціла змінна, яка розподілена по області обчислювального простору. Таким чином, після виконання оператора в рядку 24 кожний компонент *myn* містить число рядків результуючої матриці, що обчислюється віртуальним процесором, якому вона належить.

Рядок 26 визначає цілі змінні *i* і *j* розмазані по мережі *w*.

Рядок 27 визначає три масиви, розподілених по мережі *w*. Тип *dy* визначений статично як масив з *N* масивів з *N* дійсних чисел. Тип *dx* і *dz* динамічно визначається як масив з *myn* масивів з *N* дійсних чисел. Слід відзначити, що вимірність *myn* масивів *dx* і *dz* не однакова для різних компонентів цих масивів.

Рядок 28 містить незвичний унарний постфіксний оператор []. Справа в тому, що, строго говорячи, *mpC* є розширенням векторного розширення ANSI

C, яке називається C[], у якому уведене поняття вектора як упорядкованої послідовності значень деякого одного типу. На відміну від масиву, вектор є не об’єктом даних, а новим видом значення. Зокрема, значенням масиву є вектор. Оператор [] був уведений для доступу до масиву як цілому. Він має операнд типу «масив» і блокує перетворення операнда у покажчик. Таким чином, *y*[] позначає масив *y* як один об’єкт, а *dy*[] позначає розподілений масив *dy* як один об’єкт.

Оператор у рядку 28 розсилає матрицю *Y* від батька мережі *w* всім віртуальним процесорам цієї мережі. У результаті кожний компонент розподіленого масиву, на який указує *dy*, буде містити цю матрицю. У загальному випадку, якщо операнд ліворуч оператора присвоювання розподілений по деякій області обчислювального простору *R*, а значення операнда праворуч належить деякій області обчислювального простору, який охоплює *R* і присвоєння може бути зроблене без перетворення типів, то виконання оператора полягає в посилці значення правого операнда кожному віртуальному процесору області *R*, де воно привласнюється відповідному компоненту операнда ліворуч.

Оператор у рядку 29 розсилає матрицю *X* від віртуального хост-процесора по всіх віртуальних процесорах мережі *w*. У результаті кожний компонент *dx* містить відповідну порцію матриці *X*.

У загальному випадку першим операндом чотиримісної операції =:: повинен бути масив, розподілений по деякій області *R*, що складається з *NP* віртуальних процесорів. Необов’язкові другий і третій операнди – нерозподілені й належать одному процесору. Другий операнд повинен або вказувати на початковий елемент *NP*-елементного цілого масиву, або на *NP*-елементний цілий масив. Третій операнд повинен або вказувати на покажчик на початковий елемент *NP*-елементного цілого масиву, значення *i*-го елемента якого не повинне бути більше кількості елементів *i*-й компоненти першого операнда, або вказувати на такий масив. Четвертий операнд повинен бути масивом, значення елементів якого можуть бути без перетворення типу привласнені будь-

якому елементу будь-якого компонента першого операнда.

Виконання *e*1=*e*2:*e*3:*e*4 полягає у вирізанні *NP* підмасивів (можливо таких, які перекриваються) з масиву *e*4 і посилці значення *i*-го підмасиву *i*-му віртуальному процесору у області *R*, де воно привласнюється відповідному компоненту розподіленого масиву *e*1. Зміщення *i*-го підмасиву щодо початкового елемента масиву *e*4 задається значенням *i*-го елемента масиву *e*2, а його довжина – значенням *i*-го елемента масиву *e*3.

Якщо *e*3 указує на порожній покажчик (який має значення *NULL*), то операція виконується, так ніби **e*3 указує на початковий елемент *N*-елементного цілого масиву, значення *i*-го елемента якого дорівнює довжині *i*-го компонента розподіленого масиву *e*1. Навіть, у цьому випадку такий масив дійсно створюється в результаті виконання операції, а покажчик на його початковий елемент привласнюється **e*3.

Аналогічно, якщо *e*2 указує на порожній покажчик (який має значення *NULL*), то операція виконується, так ніби **e*2 указує на початковий елемент *N*-елементного цілого масиву, значення нульового елемента якого дорівнює 0, а значення *i*-го елемента дорівнює сумі значення його (*i*-1)-го елемента й значення *i*-го елемента масиву *e*3. Точно так само, у цьому випадку масив створюється, а покажчик на його початковий елемент привласнюється **e*2.

Другий і третій операнди операції можуть бути опущені. У цьому випадку операція виконується, так ніби *e*2 і *e*3 указують на *NULL*. Єдине розходження полягає в тім, що *NP*-елементні масиви, які створені в ході виконання операції, звільняються.

Таким чином, *dx*[]=::*x*[] у рядку 29 призводить до поділу *N*-елементного масиву *x* масивів з *N* дійсних чисел на *nprocs* підмасивів таким чином, що довжина *i*-го підмасиву дорівнює значенню компонента *myn*, який належить *i*-му віртуальному процесору, тобто значенню [*w*:*I*==*i*]*myn*. Ця ж операція буде виконана швидше, якщо використати іншу форму *dx*=:&*ns*:*x*, що дозволить уникнути додаткових і зайвих комунікацій і обчислень, які необхідні для формування опушених операндів.

Асинхронний (такий, що потребує комунікацій) оператор у рядках 30-32 паралельно обчислює відповідну порцію результуючої матриці *Z* на кожному з віртуальних процесорів мережі *w*.

І, нарешті, оператор у рядку 32 збирає ці порції на віртуальному хост-процесорі, формуючи результуючий масив *z*, за допомогою оператора складання ::=. Ця чотиримісна операція відповідає операції розсилання й виконує подібним чином зворотні комунікаційні операції.

Система програмування *mpC* включає компілятор, систему підтримки часу виконання, бібліотеку й командний користувальницький інтерфейс.

Компілятор виконує трансляцію вихідної програми на *mpC* в ANSI C програму зі звертаннями до функцій системи підтримки часу виконання. При цьому використовується або модель SPMD, або модель квазі-SPMD. У першому випадку всі процеси паралельної програми виконують однаковий код. При другому підході вихідний *mpC* файл транслюється у два різних файли – один для хост-процесора й другий для інших.

Система підтримки часу виконання управляє обчислювальним простором на цільовій обчислювальній машині з розподіленою пам’яттю й забезпечує передачу повідомлень. Для цього система повністю інкапсулює комунікаційний пакет стандарту MPI і забезпечує незалежність компілятора від конкретної цільової платформи.

Командний користувальницький інтерфейс включає команди для створення віртуальної обчислювальної машини й виконання *mpC* програм на ній. При створенні віртуальної паралельної машини її топологія (число й продуктивності процесорів, характеристики комунікаційних зв’язків між процесорами) визначається автоматично.

Перша реалізація системи програмування *mpC* з’явилася наприкінці 1996 року. Вона є вільно розповсюджуваним ПЗ й доступна в Internet. Використовується система, в основному, для проведення наукових розрахунків на мережах робочих станцій і ПК. Основне коло розв’язуваних завдань – це множення матриць, розв’язання проблеми N тіл, задачі лінійної алгебри, чи-

сельне інтегрування, моделювання видобутку нафти, розрахунки міцності будівельних конструкцій і багато чого іншого. Система *mpC* дозволяє розробляти переносимі модульні паралельні програми, яки значно прискорюють розв'язання як регулярних, так і нерегулярних завдань на неоднорідних мережах. Крім того, *mpC* дозволяє вирішувати нерегулярні завдання на однорідних мережах набагато швидше, ніж за допомогою традиційних засобів.

1.6 Засоби підтримки паралельних обчислень

Строго кажучи, велика розмаїтість архитектур паралельних систем передбачає таку ж саму розмаїтість систем програмування, які забезпечують створення програм для різних способів здійснення паралелізму (конвеєрні обчислення, багатопроцесорні системи, мультиком'ютери й т.п.). Проте, інваріантність створюваних паралельних програм може бути забезпечена завдяки використанню *типових програмних засобів* підтримки паралельних обчислень (програмних бібліотек PVM, MPI і ін.).

1.6.1 Програмний інтерфейс паралельної віртуальної машини

Програмне забезпечення ***PVM*** (Parallel Virtual Machine – ***паралельна віртуальна машина***) надає уніфіковані структури, за допомогою яких паралельні програми можуть розроблятися ефективним способом для існуючого обладнання. PVM дозволяє групі гетерогенних комп'ютерних систем, які мають несумісні комп'ютерні архітектури, сприйматися як одна паралельна віртуальна машина.

Обчислювальна модель PVM є простою й доволі узагальненою. Користувач пише свою програму у вигляді групи взаємозалежних завдань. Завдання одержують доступ до ресурсів PVM за допомогою бібліотеки підпрограм зі стандартизованим інтерфейсом. Ці підпрограми дозволяють ініціювати й завершити завдання в мережі, а також забезпечити зв'язок між завданнями і їхню синхронізацію. Примітиви обміну повідомленнями PVM орієнтовані на

гетерогенні операції, які включають строго визначені конструкції для буферування й пересилання даних. Ці примітиви використовуються як комунікаційними конструкціями для передачі й прийому структур даних, так і високорівневими примітивами широкомовної передачі, бар'єрної синхронізації й глобального підсумовування.

У будь-якій точці виконання взаємозалежних додатків будь-яке можливе завдання може запускати або зупиняти інші завдання, додавати або видаляти комп'ютери з віртуальній машини. Кожний процес може взаємодіяти і/або синхронізуватися з будь-яким іншим.

Метою створення проекту PVM було дослідження проблематики й розробка рішень в області гетерогенних паралельних обчислень. Головною же метою системи PVM є забезпечення можливості спільного використання групи комп'ютерів для взаємозалежних або паралельних обчислень.

При розробці PVM був врахований ряд вимог, які пред'являються до подібних до систем. У першу чергу – це система комп'ютерів, яка конфігурується користувачем, причому до системи можуть динамічно додаватися нові й віддалятися непотрібні комп'ютери. У систему можуть бути включені як однопроцесорні машини, так і апаратне забезпечення комп'ютерів з поділюваною і розподіленою пам'яттю.

Врахована також прозорість доступу до обладнання – прикладні програми бачать апаратне забезпечення не просто як групу віртуальних обчислювальних елементів, але можуть враховувати можливості і специфіку машин у системі, переміщаючи окремі завдання на найбільш підходящі для їхнього вирішення комп'ютери.

Одиницею паралелізму в PVM є завдання – це *незалежний послідовний потік керування*, що може бути або комунікаційним, або обчислювальним. PVM не містить карти зв'язків процесів, причому складені завдання можуть виконуватися на одному процесорі.

Групи обчислювальних завдань, виконують свою частину додатка використовуючи декомпозицію даних, функцій або гібридну, взаємодіють поси-

лаючи повідомлення один одному й приймаючи їх.

І, як вже відзначалося, у системі підтримується гетерогенність машин, мереж і додатків, також врахована можливість підтримки мультипроцесорів.

Система PVM складається із двох частин. Перша частина – це *демон*, який поміщається на всі комп'ютери, що створюють віртуальну машину. Він розроблений так, щоб будь-який користувач із достовірним логином міг інсталювати його на машину. Користувачеві щоб запустити додаток PVM, необхідно створити віртуальну машину. Після цього додаток запускається з будь-якого термінала на кожному з комп'ютерів. Декілька користувачів можуть конфігурувати віртуальні машини, які перекриваються, кожний користувач може послідовно запустити кілька додатків PVM.

Друга частина системи – це бібліотека підпрограм інтерфейсу PVM. Вона містить функціонально повний набір примітивів, які необхідні для взаємодії між завданнями додатка. Ця бібліотека містить підпрограми, які викликаються користувачем для обміну повідомленнями, породження процесів, координування завдань і модифікації віртуальної машини.

Обчислювальна модель PVM базується на припущенні, що додаток складається з декількох завдань. Кожне завдання відповідальне за частину обчислювального навантаження додатка. Іноді додаток розпаралелюється за функціональним принципом (*паралелізм завдань*), тобто кожне завдання виконує свою функцію, наприклад: введення, породження, обчислення, виведення, відображення. Більш часто зустрічається *паралелізм даних*. PVM підтримує кожний із цих методів окремо або в комплексі. Залежно від функцій завдання можуть виконуватися паралельно й мати потребу в синхронізації або у обміні даними.

PVM підтримує мови програмування C, C++ і Фортран (є розробки для Java). Цей набір мовних інтерфейсів узятий за основу у зв'язку з тим, що переважна більшість цільових додатків написані на C і Фортран.

Прив'язка мов C і C++ до користувальницького інтерфейсу PVM реалізована у вигляді функцій, які підпорядковуються загальноприйнятим підхо-

дам, використовуваним більшістю С-систем. На додаток до цього використовуються макровизначення для системних констант і такі глобальні змінні як *errno* і *pvm_errno*. Прикладні програми, написані на С і С++, одержують доступ до функцій бібліотеки PVM шляхом компонування з бібліотекою (libpvm3.a) зі стандартного дистрибутива.

Прив'язка до мови Фортран реалізована переважно у вигляді підпрограм, ніж у вигляді функцій. Такий підхід застосовується з тієї причини, що деякі компілятори не можуть правильно реалізувати інтерфейс між С- і Фортран-функціями. Безпосереднім наслідком з цього є те, що для кожного виклику бібліотеки PVM на Фортрані, для повернення результуючого статусу в програму, яка зробила виклик, уводиться додатковий аргумент. Бібліотечні підпрограми уніфіковані для розміщення уведених даних у буфери повідомлення і їхнього відновлення, вони мають додатковий параметр для відображення типу даних. Крім цих розходжень (і різниці в стандартних префіксах при виклику функцій: *pvm_* – для С і *pvmf_* – для Фортрану), можлива взаємодія між двома мовними прив'язками. Інтерфейси PVM на Фортрані реалізовані у вигляді бібліотечних контейнерів, які викликають потрібні функції на С. Таким чином, додатки на Фортрані додатково вимагають компонування з бібліотекою-надбудовою (libfpvm3.a).

У системі всі завдання PVM ідентифікуються за допомогою цілочисельного ***ідентифікатора завдання*** (task identifier – *TID*). З їхньою допомогою передаються й приймаються повідомлення. Ідентифікатори унікальні в межах внутрішньої віртуальної машини, що підтримується локальним демоном. PVM містить кілька підпрограм, які повертають значення *TID*, тим самим даючи можливість користувальницькому додатку ідентифікувати інші завдання в системі.

У деяких додатках доцільне створення груп завдань, а також бувають випадки, коли користувачеві зручніше визначати свої завдання по номерах від 0 до (*p*-1), де *p* – кількість завдань. Із цією метою PVM підтримує концепцію іменованих користувачем груп. При цьому завдання входить у групу і їй при-

власнюється «випадковий» номер у групі. Будь-яке завдання PVM може ввійти до складу будь-якої групи або покинути її в довільний момент часу, не інформуючи про це інші завдання в групі. Групи можуть перекриватися, а завдання можуть розсилати широкомовні повідомлення, адресовані тим групам, у яких вони не перебувають. Для використання кожної із групових функцій програма повинна бути зкомпонована зі спеціальною бібліотекою libgpvm3.a.

Загальна парадигма для програмування додатків за допомогою PVM виглядає в такий спосіб. Користувач пише одну або кілька послідовних програм на C, C++ або Фортрані, у які вбудовані виклики бібліотеки PVM. Кожна програма породжує завдання, що реалізує додаток. Ці програми компілюються за правилами кожної архітектури в системі комп'ютерів, і в результаті виходять об'єктні файли, які поміщаються на доступних машинах системи. Для виконання додатка користувач запускає одну копію завдання (*провідне* або *ініціюче* завдання) вручну на одному з комп'ютерів. Цей процес послідовно породжує інші завдання PVM. В остаточному підсумку, виходить група активних завдань, які оперують локально й обмінюються повідомленнями один з одним для вирішення проблеми. Це типовий сценарій, але вручну можна запускати будь-яку кількість завдань. Як уже згадувалося, завдання взаємодіють шляхом прямого обміну повідомленнями за допомогою ідентифікації визначеними системою прихованими *TID*.

Базову концепцію програмування PVM можна проілюструвати стандартною програмою «Hello World» (фрагмент 1.5). Ця програма запускається вручну. Після виведення на екран свого ідентифікатора завдання (отриманого за допомогою *pvm_mytid*()) вона породжує копію іншої програми за назвою «hello_other», використовуючи функцію *pvm_spawn*(). Успішне породження змушує програму виконати прийом повідомлення з блокуванням, за допомогою *pvm*_recv(). Після прийому повідомлення програма виводить на екран повідомлення, яке надіслане до неї абонентом і його ідентифікатор завдання. Уміст буфера витягується з повідомлення функцією *pvm_upksrt*(). Заключний виклик *pvm_exit* завершує програму.

Фрагмент коду 1.5

```
#include "pvm3.h"
main()
{
  int cc, tid, msgtag;
  char buf[100];
  printf("Це програма \t%x\n", pvm_mytid());
  cc = pvm_spawn("hello_other", (char**)0, 0, "", 1, &tid);
  if (cc == 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("Повідомлення \t%x: %s\n", tid, buf);
  } else
    printf("Неможливо запустити hello_other\n");
  pvm_exit();
}
```

Код породжуваної програми наведений на фрагменті 1.6. Її першою дією є одержання ідентифікатора провідного завдання викликом *pvm_parent*(). Потім вона визначає власне ім’я комп’ютера й передає його провідній програмі, використовуючи послідовність із трьох викликів: *pvm_initsend*() – для ініціалізації буфера передачі; *pvm_pkstr*() – для розміщення рядка, навмисно уведеного в архітектурно-незалежному стилі, у буфері передачі; *pvm_send*() – для його пересилання до запитуючого процесу, впізнаному за допомогою *ptid*, з призначенням повідомленню номера 1 (змінна *msgtag*).

Фрагмент коду 1.6

```
#include "pvm3.h"
main()
{
  int ptid, msgtag;
  char buf[100];
  ptid = pvm_parent();
  strcopy(buf, "hello, world from");
  msgtag = 1;
  gethostname(buf + strlen(buf), 64);
  msgtag = 1;
  pvm_initsend(PvmDataDefault);
  pvm_pkstr(buf);
  pvm_send(ptid, msgtag);
```

```
  pvm_exit();
}
```

Всеосяжна природа концепції віртуальної машини, а також її простий, але функціонально повний програмний інтерфейс, забезпечили системі PVM широке визнання, у тому числі й у науковому співтоваристві, пов'язаному з високошвидкісними обчисленнями.

1.6.2 Бібліотека передачі повідомлень MPI

Наступний пакет для програмування мультикомп'ютерів, а також й мультипроцесорів – це бібліотека процедур MPI (Message-Passing Interface – ***інтерфейс передачі повідомлень***). MPI набагато складніша, ніж PVM. Пакет містить набагато більше бібліотечних викликів і набагато більше параметрів для кожного виклику. Перша версія MPI, що зараз називається MPI-1, була доповнена другою версією, MPI-2, в 1997 році, в 2012 році з'явився стандарт MPI-3, який поки не одержав широкого розповсюдження.

MPI-1, на відміну від PVM, ніяк не пов'язана зі створенням процесів і керуванням процесами. Створювати процеси повинен сам користувач за допомогою локальних системних викликів. Після створення процеси організуються в групи, які вже не змінюються. Саме із цими групами й працює MPI.

В основі MPI лежать 4 поняття: *комунікатори, типи переданих даних, операції комунікації* й *віртуальні топології.*

Комунікатор – це група процесів плюс їхній контекст. ***Контекст*** – це мітка, яка ідентифікує що-небудь (наприклад, фазу виконання). У процесі відправлення й одержання повідомлень контекст може використовуватися для того, щоб незв'язані повідомлення не заважали одне одному.

Повідомлення можуть бути різних типів: символьні, целочисленные (short, regular і long integers), зі звичайною й подвоєною точністю, з рухомою крапкою й т.д. Можна утворити нові типи повідомлень із уже існуючих.

MPI підтримує безліч операцій комунікації. Нижче наведена операція, що використовується для відправлення повідомлень:

```
MPI_Send(buffer, count, data_type, destination, tag, communicator).
```

Цей виклик відправляє вміст буфера (*buffer*) з елементами визначеного типу (*data_type*) у пункт призначення – *destination* (це просто індекс у списку процесів з певної групи). Параметр *count* – це кількість елементів у буфері. Поле *tag* позначає повідомлення; одержувач може сказати, що він буде приймати повідомлення тільки з даним тегом. Останнє поле – *communicator* показує, у яку групу процесів входить цільовий процес. Відповідний виклик для одержання повідомлення такий:

```
MPI_Recv(&buffer, count, data_type, source, tag, communicator, status).
```

У ньому повідомляється, що одержувач шукає повідомлення визначеного типу – *data_type* від вказаного джерела – *source* з вказаним тегом – *tag*.

MPI підтримує чотири основних типи комунікації. *Перший тип* – ***синхронний***, у ньому відправник не може почати передачу даних, поки одержувач не викличе процедуру *MPI_Recv*. *Другий тип* – ***комунікація з використанням буфера***. Обмеження для першого типу тут не існує. *Третій тип* – ***стандартний***. Він залежить від реалізації й може бути або синхронним, або з буфером. *Четвертий тип – подібний до першого*. В цьому разі відправник вимагає, щоб одержувач був доступний для комунікації, але без перевірки факту одержання/відправлення повідомлення. Кожний із цих примітивів буває двох видів: які *блокують* процес доти поки не завершиться операція комунікації і таки, що *не блокують* процеси, це в сумі дає 8 можливостей відправлення повідомлення. Одержання повідомлення може бути *тільки у двох варіантах*: з блокуванням і без блокування.

MPI підтримує колективну комунікацію – *широкомовлення*, *розподіл* і *збирання даних*, *обмін даними*, *агрегацію* й *бар'єр*. При будь-яких формах колективної комунікації всі процеси в групі повинні робити цей виклик, причому із сумісними параметрами. Якщо цього зробити не вдається, виникає помилка. Наприклад, процеси можуть бути організовані у вигляді дерева, у якому значення передаються від листів до кореня, підкоряючись певній обробці на кожному кроці (це може бути додавання значень або находження максимуму). Це типові форми колективної комунікації.

Останнє основне поняття в MPI – *віртуальна топологія*, коли процеси можуть бути організовані в дерево, кільце, ґрати, тор і т.д. Така організація процесів забезпечує спосіб найменування каналів зв'язку й полегшує комуні-

кацію.

Специфікація MPI-1 навмисне, з міркувань доцільності, не торкалася кількох «важких» питань, ці питання були відкладені до пізніших специфікацій.

MPI-2 була значною переробкою MPI-1 з додаванням нової функціональності та виправлень. Основні сфери додання нової функціональності в MPI-2:

- *динамічні процеси* – розширення, що усуває статичну модель процесів MPI і забезпечує процедури для створення нових процесів після запуску завдання;
- *односторонні комунікації* – додає процедури для односпрямованих комунікацій, а також включає спільну пам'ять операцій (покласти/отримати) та дистанційне накопичення операцій;
- *розширені колективні операції* – дозволяє використання колективних операцій з зовнішніми комунікаторами;
- *зовнішні інтерфейси* – визначає процедури, які надають розробникам рівень поверх MPI, наприклад, дебагери і профайлери;
- *додаткові мовні прив'язки* – описує прив'язки до C++ і описує застосування мови Фортран-90;
- *паралельне* введення/виведення – описує MPI-підтримку паралельного введення/виведення.

Стандарт MPI-3 який був прийнятий зовсім недавно містить значні розширення функціональності не тільки MPI-1, а й MPI 2, включаючи:

- *неблокуючі колективні операції* – дозволяє завданням, що працюють колективно, виконувати операції без блокування, тім самим, можливо, поліпшуючи продуктивність;
- *нові односторонні операції зв'язку* – покращене керування різними моделями пам'яті;
- *локальні групи* – розширює розподілені графові і декартові топології

процесів додатковою комунікаційною потужністю;

- *зв'язування з Фортраном-2008* – розширено зв'язування з Фортраном-90;
- *MPIT інструмент інтерфейсу* – новий інструмент інтерфейсу, який дозволяє користувачеві відкривати певні внутрішні змінні, лічильники та інше (інструмент оптимізації та підвищення продуктивності);
- *узгоджені проби* – виправлена стара помилка в MPI-2, коли ніхто не міг виконати операцію *MPI_Probe* для повідомлень у богатопотоковому середовищі.

Слід відзначити, що серед користувачів описаних SDK (PVM і MPI) постійно їдуть суперечки. Прихильники PVM, стверджують, що цю систему простіше вивчати й легше використовувати. Прихильники MPI, стверджують, що MPI виконує більше функцій і, крім того, вона стандартизована, що підтверджується офіційними документами.

1.7 Комунікаційні, колективні, глобальні обчислювальні операції над розподіленими даними

У розподілених системах не існує пам'яті, безпосередньо доступної процесам, що працюють на різних комп'ютерах, тому комунікація процесів може здійснюватися тільки шляхом передачі повідомлень через мережу.

Повідомлення – це блок інформації, відформатований процесом-відправником таким чином, щоб він був зрозумілий процесу-одержувачу. У загальному випадку повідомлення складається із заголовка, звичайно фіксованої довжини, і набору даних певного типу змінної довжини. У заголовку, як правило, утримуються наступні елементи:

- адреси – символи, які унікально визначають процеси відправника й одержувача. Адресне поле в такий спосіб складається із двох частин - адреси процесу-відправника й адреси процесу-одержувача. Адреса

кожного процесу може, у свою чергу, мати деяку структуру, або являти собою просто якийсь унікальний номер, який дозволяє знайти потрібний процес у системі, що складається з великої кількості комп'ютерів.

– порядковий номер повідомлення, що є його ідентифікатором. Використовується для ідентифікації загублених повідомлень і дублікатів повідомлень у випадку відмов у мережі.
– структурована інформація, що складається в загальному випадку з декількох частин: полів типу даних, довжини даних і значення даних (тобто самих даних). Поле типу даних визначає, наприклад, що дані є цілим числом або ж являють собою рядок символів (це поле може також містити покажчик на дані, які зберігаються десь поза даним повідомленням). Друге поле визначає довжину переданих у повідомленні даних (звичайно в байтах), тобто розмір наступного поля повідомлення. Повідомлення може включати кілька елементів, що складаються з описаних трьох полів.

Система передачі повідомлень дозволяє процесам взаємодіяти за допомогою досить простих примітивів. У найпростішому випадку системні засоби забезпечення зв'язку можуть бути зведені до двох основних комунікаційних примітивів, перший – *send* (відправити) для посилки повідомлення, інший – *receive* (одержати) для одержання повідомлення. Надалі на їхній базі будуються могутніші засоби комунікацій.

Незважаючи на концептуальну простоту примітивів *send* і *receive*, існують різні варіанти їхньої реалізації, від правильного вибору яких залежить ефективність роботи комп'ютера. При виборі реалізації примітивів *send* і *receive* необхідно враховувати наступні аспекти обчислювальної системи: кількість одержувачів повідомлення один або кілька; гарантується доставка повідомлень або ні; необхідність відправникові дочекатися відповіді на своє повідомлення; реакція відправника й одержувача на відмови мережі комунікації

й багато чого іншого.

Комунікаційні примітиви діляться на операції ***з блокуванням*** – ***синхронні*** і ***асинхронні*** – ***без блокування***. При виконанні комунікаційних примітивів завершення запитаної операції в загальному випадку залежить від роботи не тільки локального, але й віддаленого вузла.

При використанні примітива *send з блокуванням* процес, що видав запит на його виконання, припиняється до моменту одержання повідомлення про те, що приймач одержав відправлене повідомлення. Виклик примітива *receive з блокуванням* припиняє визивний процес до моменту, коли він одержить повідомлення. При використанні примітивів *без блокування*, *send* і *receive* керування вертається до визивного процесу негайно, відразу після виклику функції. Перевагою цієї схеми є паралельне виконання визивного процесу й процедур передачі повідомлення.

При використанні примітива *без блокування – receive* для оповіщення процесу-одержувача про те, що повідомлення прийшло й поміщене в буфер застосовується базовий *примітив – test* (перевірити), за допомогою якого процес-одержувач може аналізувати момент приходу повідомлення.

Якщо при взаємодії двох процесів обидва примітиви *send* і *receive* виконуються з блокуванням, говорять, що процеси взаємодіють ***синхронно***, у противному випадку взаємодія вважається ***асинхронною***.

У порівнянні з асинхронною взаємодією синхронна простіше і її легше реалізувати. Вона також надійніше, тому що гарантує процесу-відправникові, який відновив своє виконання, що його повідомлення було отримано. Головний же недолік – обмежений паралелізм і можливість виникнення тупикових ситуацій, коли приймаючий процес виявляється неготовим обробити повідомлення. Для усунення недоліку використовуються пересилання повідомлень з буферуванням, тобто повідомлення для наступної обробки зберігається в буфері.

У випадку синхронної комунікації буфер найчастіше вибирають розмі-

ром в одне повідомлення, тому що процес-відправник не може послати наступне повідомлення, не одержавши підтвердження про прийом попередні. Для асинхронних примітивів буферування є бажаним, але можуть виникати ситуації з переповненням буфера й, тим самим, із втратою повідомлень. Для зниження ймовірності втрат повідомлень використовується механізм керування потоком повідомлень на зразок застосовуваного в протоколі TCP.

Системи програмування для організації комунікацій з буферуванням надають спеціальний примітив *create_buffer* для створення буферів повідомлень. Ця функція викликається перед тим, як відправляти або одержувати повідомлення за допомогою примітивів *send* і *receive.*

При відправленні й одержанні повідомлень для адресації процесів застосовується числовий ідентифікатор, що має унікальне значення в межах обчислювального вузла. Цей ідентифікатор просто однозначно вказує на конкретний процес, що працює в даній паралельній обчислювальній системі.

Крім всіх цих примітивів системи передачі повідомлень підтримують засновані на них примітиви ***широкомовлення*** й ***мультимовлення***. Перша процедура відправляє повідомлення всім процесам у групі, друга посилає повідомлення тільки деяким процесам, що входять у деякий список.

Синхронізація між процесами найчастіше здійснюється за допомогою процедури ***бар'єра*** (*Barrier*). Коли процес викликає цю процедуру, він блокується до тех. пор, поки наперед визначена кількість інших процесів не досягне бар'єра й вони не викличуть цю ж процедуру.

Деякі системи, наприклад MPI, підтримують колективну комунікацію з розподілом і збором даних, з обміном даними й агрегацію. При будь-яких формах колективної комунікації всі процеси в групі повинні робити виклик відповідної функції. Дуже корисною властивістю систем передачі даних є можливість створення ***віртуальної топології.***

Зі сказаного вище можна зробити кілька основних висновків.

По-перше, єдиною по-справжньому важливою відмінністю розподіле-

них систем є використовуваний ними спосіб взаємодії між процесами – у розподілених системах взаємодія процесів може здійснюватися тільки шляхом передачі повідомлень.

По-друге, всі можливі типи комунікацій у системах передачі повідомлень працюють на основі двох основних комунікаційних примітивів – *send* і *receive*.

У свою чергу, основними характеристиками комунікаційних примітивів є:

- спосіб адресації;
- наявність синхронізації;
- спосіб буферування;
- ступінь надійності доставки повідомлення.

1.8 Моделі віддаленого виклику процедур та віддаленого застосування методів

Для полегшення організації розподілених обчислень розроблені засоби для передачі керування й даних через мережу, які засновані на механізмі передачі керування й даних усередині програми на одній машині.

1.8.1 Організація віддаленого виклику процедур RPC

Одним з таких засобів є ***віддалений виклик процедур*** – ***RPC*** (Remote Procedure Call). Хоча цей засіб і оснований на механізмі виклику звичайних локальних процедур, але реалізація віддалених викликів істотно складніша реалізації локальних.

У першу чергу складності пов’язані з тим, що визивна й викликувана процедури виконуються на різних машинах і мають різні адресні простори. Це створює проблеми при передачі параметрів і результатів виконання. Відсутність поділюваної пам’яті призводить до того, що параметри RPC не повинні містити покажчиків на комірки пам’яті і значення параметрів повинні

якось копіюватися з одного комп'ютера на іншій.

Віддаленість вносить додаткові проблеми. У реалізації RPC беруть участь як мінімум два процеси – по одному в кожній машині. У випадку якщо один з них аварійно завершиться, можуть виникнути наступні ситуації:

- при аварії визивної процедури, віддалено викликані процедури стають «осиротілими»;
- при аварійному завершенні віддалених процедур стають «знедоленими батьками» визивні процедури, які безрезультатно очікують відповіді від віддалених процедур.

Крім того, існує ряд проблем, пов'язаних з неоднорідністю мов програмування й операційних середовищ.

Ідея, покладена в основу RPC, полягає в тому, щоб виклик віддаленої процедури по можливості виглядав так само, як і виклик локальної процедури. Тобто необхідно зробити механізм RPC прозорим, щоб визивна процедура не знала, що викликувана процедура перебуває на іншій машині, і навпаки.

Коли викликувана процедура є віддаленою, у бібліотеку процедур замість локальної процедури поміщається інша версія процедури, називана ***клієнтським стабом*** (*stub* – заглушка) (рис. 1.29). На віддаленому комп'ютері (сервері процедур), поміщається оригінальний код викликуваної процедури, а також ще один стаб, називаний ***серверним стабом***. Призначення клієнтського й серверного стабів – організувати передачу параметрів викликуваної процедури й повернення значення визивної процедурі через мережу. Для передачі даних через мережу стаби використовують засоби підсистеми обміну повідомленнями, тобто примітиви *send* і *receive*. Іноді у підсистемі обміну повідомленнями виділяється програмний модуль, що організує зв'язок стабів із примітивами передачі повідомлень, називаний ***модулем часу виконання RPC*** (RPC Runtime).

Подібно до локальної процедури, клієнтський стаб викликається шляхом звичайної передачі параметрів через стек, потім відбувається формування

повідомлення, яке містить ім’я викликуваної процедури і її параметри.

Ця операція називається ***впакуванням параметрів***. Після впакування клієнтський стаб звертається до примітива *send* для передачі сформованого повідомлення віддаленому комп’ютеру, на який поміщена реалізація оригінальної процедури. Одержавши з мережі повідомлення, ядро ОС віддаленого комп’ютера викликає серверний стаб, який повинен попередньо викликати примітив *receive*. Серверний стаб розпаковує параметри виклику, наявні в повідомленні, і звичайним образом викликає оригінальну процедуру, передаючи їй параметри через стек. Після закінчення роботи процедури серверний стаб упаковує результат роботи й за допомогою примітива *send* передає повідомлення по мережі клієнтському стабу, а той звичайним образом повертає результат і керування визивній процедурі.

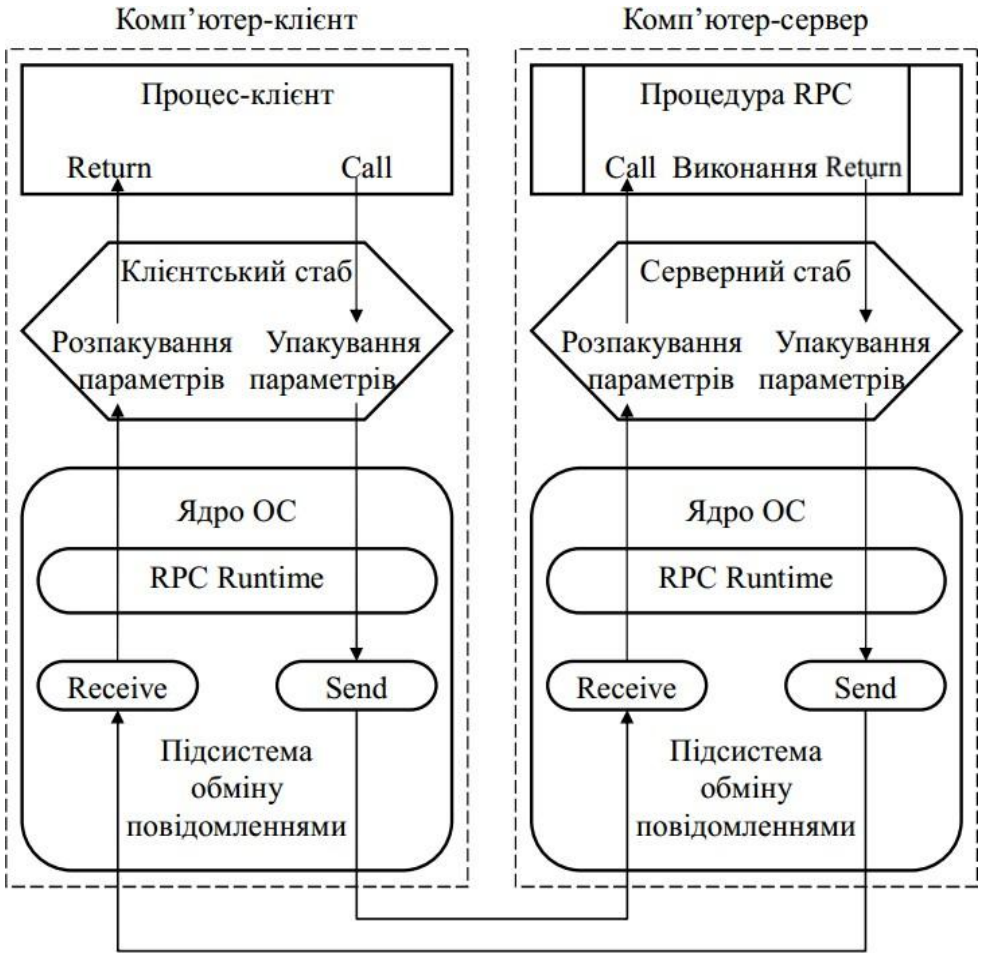


Рисунок 1.29 − Виконання віддаленого виклику процедури

Стаби можуть генеруватися або вручну, або автоматично. У першому випадку програміст використовує для генерації ряд допоміжних функцій, які йому надає розроблювач засобів RPC. При цьому способі програміст одержує більшу свободу у виборі способу передачі параметрів виклику й у застосуванні тих або інших примітивів передачі повідомлень, однак цей спосіб пов'язаний з більшим обсягом ручної праці.

Автоматичний спосіб заснований на застосуванні спеціальної ***мови визначення інтерфейсу*** (Interface Definition Language – ***IDL***). За допомогою цієї мови програміст описує інтерфейс між RPC-клієнтом і RPC-сервером. Опис включає список імен процедур, виконання яких клієнт може запросити від сервера, а також список типів аргументів і результатів цих процедур.

Після того як опис інтерфейсу складений програмістом, він компілюється спеціальним IDL-компілятором, який виробляє вихідні модулі клієнтських і серверних стабів для зазначених в описі процедур, а також генерує спеціальні *.h*-файли з описом типів процедур і їхніх аргументів.

Крім компіляції стабів, необхідно ще встановити відповідність між RPC-клієнтом і RPC-сервером. Процедура завдання такої відповідності зветься ***зв'язуванням*** (*binding*). Методи зв'язування, застосовувані в різних реалізаціях RPC, відрізняються:

- способом завдання сервера, з яким зв'язується клієнт;
- способом виявлення процесом зв'язування мережної адреси (місця розташування) необхідного сервера;
- стадією, на якій відбувається зв'язування.

У найбільш простому випадку *ім'я або адреса* RPC-сервера *задається в явній формі*, як аргумент клієнтського стабу або програми-сервера, що реалізує інтерфейс визначеного типу. Наприклад, у якості такого аргументу можна використовувати IP-адресу комп'ютера й номер порту TCP/UDP, через який приймається повідомлення процедур. Основний недолік такого підходу – *відсутність гнучкості й прозорості*.

Динамічне зв'язування вимагає зміни способу іменування сервера. Най-

більш гнучким є використання ім’я RPC-інтерфейсу, що складається із двох частин:

- типу інтерфейсу;
- екземпляра інтерфейсу.

Тип інтерфейсу визначає всі характеристики інтерфейсу, крім його місця розташування. Це ті ж характеристики, що вказуються в описі для IDL-компілятора. Частину, яка описує екземпляр інтерфейсу, повинна точно задати мережна адреса сервера, який підтримує даний інтерфейс. Друга частина ім’я інтерфейсу може бути опущена.

Динамічне зв’язування іноді називають ***імпортом/експортом інтерфейсу***: клієнт імпортує інтерфейс, а сервер його експортує.

У тому випадку, коли важливим є тільки тип інтерфейсу, виявлення необхідного сервера з потрібним екземпляром інтерфейсу може бути реалізований двома способами:

- с використанням широкомовлення;
- с використанням централізованого агента зв’язування.

Ці два способи забезпечують пошук мережного ресурсу будь-якого типу за його ім’ям.

До недоліків динамічного зв’язування можна віднести додаткові накладні витрати – витрати часу на експорт і імпорт інтерфейсів.

1.8.2 Розподілені об’єктні моделі RMI

При розробці мови програмування Java, як мови мережної взаємодії, необхідно було передбачити механізм для виконання розподілених обчислень подібний до механізму RPC. Такий механізм дійсно був включений у систему за назвою ***RMI*** (Remote Method Invocation – ***виклик віддалених методів***).

Java RMI є механізмом, що дозволяє програмі Java, яка працює на одному комп’ютері, викликати метод об’єкта, розташованого на іншому комп’ютері. RMI являє собою реалізацію об’єктної моделі розподіленого про-

грамування подібної до CORBA, але більше простої і пристосованої до мові Java.

Синтаксис виклику віддаленого методу виглядає як звичайний виклик методу Java. При виклику віддаленого методу йому можуть бути передані аргументи, обчислені в контексті локальної машини, і він може повертати довільні значення, обчислені в контексті віддаленої машини. З погляду програміста, виконуюча система RMI працює з методами віддалених об'єктів зовсім прозоро.

У деяких відносинах Java RMI є більш загальним механізмом, чим CORBA, оскільки, для більш повного забезпечення об'єктно-орієнтованої семантики, здатна використовувати такі функції Java, як серіалізація об'єктів і динамічне завантаження класів.

В RMI мінімальний обсяг інформації, що повинен бути заздалегідь відомий клієнтові й серверу надається загальним віддаленим інтерфейсом. Він визначає «протокол» високого рівня, по якому машини будуть спілкуватися один з одним.

Віддалений інтерфейс є звичайним інтерфейсом Java, який повинен розширювати інтерфейс-маркер *java.rmi.Remote*. Оскільки загальнодоступні члени віддаленого об'єкта визначаються за допомогою інтерфейсу Java, то конструктори, статичні методи й поля, що змінюються, не можуть бути доступні віддалено (в інтерфейсах Java не можуть існувати такі члени). Всі методи віддаленого інтерфейсу повинні бути оголошені з можливістю генерації виключення типу *java.rmi.RemoteException*.

Наприклад, файл *MessageWriter.java* може містити таке визначення віддаленого інтерфейсу (фрагмент 1.7):

Фрагмент коду 1.7

```
import java.rmi.*;
public interface MessageWriter extends Remote {
  void writeMessage(String s) throws RemoteException;
}
```

Цей інтерфейс визначає єдиний метод віддаленого об'єкта – *writeMessage*().

Як уже згадувалося, інтерфейс *java.rmi.Remote* є інтерфейсом-маркером. Такий інтерфейс не оголошує методів або полів, але для системи RMI його розширення в деякому класі передбачає наявність у об'єктів цього класу властивостей характерних для віддалених об'єктів.

Вимога до всіх віддалених методів вкидати виключення типу *RemoteException* була свідомим вибором розробників системи RMI. RMI робить віддалені виклики синтаксично схожими на виклики локальних методів. На практиці ж, механізм віддалених викликів не може усунути проблеми, яка властива розподіленим обчисленням – несподіваного збою в мережі або на віддаленій машині. Таким чином, примушення програміста обробляти віддалені виключення допомагає вирішити проблему своєчасного виявлення й часткового виправлення таких відмов.

Віддалений об'єкт є екземпляром класу, який реалізує віддалений інтерфейс. Найчастіше це клас також розширює бібліотечний клас *java.rmi.server.UnicastRemoteObject*. Цей клас містить конструктор, що виконує експорт об'єкта в системі RMI при його створенні, таким чином, роблячи об'єкт видимим для зовнішнього миру. Звичайно програмісти не мають справу із цим класом явно, але від нього породжуються класи віддалених об'єктів.

При розширенні класу прийнято щоб ім'я субкласу віддаленого об'єкта відповідало імені інтерфейсу реалізованому в класі, але з додаванням суфікса «*Impl*». Як і домовленість про ідентифікатори типів і методів, таке правило створення імені необов'язково, але бажано. Наприклад, файл із описом класу, який реалізує інтерфейс *MessageWriter.java*, може виглядати в такий спосіб (фрагмент 1.8):

Фрагмент коду 1.8

```
import java.rmi.*;
import java.rmi.server.*;
public class MessageWriterImpl extends UnicastRemoteObject
```

```
                                    implements MessageWriter {
  public MessageWriterImpl() throws RemoteException {}
  public void writeMessage(String s) throws RemoteException
    { System.out.println(s); }
}
```

Для компіляції класів, які реалізують можливість віддалених викликів, використовується не стандартний компілятор *javac*, а спеціальний компілятор *rmic*, наприклад у такий спосіб:

```
myhome$ rmic -keep MessageWriterImpl
```

Наявності відкомпільованого файлу класу, який описує віддалений об’єкт, ще не достатньо, необхідне ще створення фактичних програм для клієнта й сервера, які використовують цей клас.

У загальному випадку доводиться виконувати досить великий обсяг адміністраторської роботи, коли необхідно зробити доступними файли класів у системі й установити менеджери безпеки. Але якщо дуже строгих вимог до захисту файлів не пред’являється і можна довіряти коду локальних класів, то забезпечення спільного доступу до файлів для клієнта й сервера здійснюється, наприклад, просто через загальні каталоги NFS. При цьому відпадають обидві проблеми, немає необхідності в публікації файлів і немає необхідності встановлювати менеджер безпеки.

Програма, що запускається на сервері, як мінімум повинна виконати дві дії:

– створити вилучений об’єкт на локальному сервері імен;
– опублікувати віддалене посилання на цей об’єкт із якимсь зовнішнім ім’ям (у прикладі – «*MessageWriter*»).

Ці дві дії можуть описуватися, наприклад таким кодом класу (фрагмент 1.9):

Перший оператор методу *main* саме й створює об’єкт, а звертання до методу *Naming.rebind*() поміщає посилання в реєстрі RMI на сервері, який працює на локальному вузлі (тобто, вузлі, де запускається програма

HelloServer).

Фрагмент коду 1.9

```
import java.rmi.*;
public class HelloServer {
  public static void main(String [] args) throws Exception
    {
       MessageWriter server = new MessageWriterImpl();
       Naming.rebind("messageservice", server);
    }
}
```

Клієнтські програми можуть одержати посилання на віддалений об’єкт із реєстру шляхом виклику методу *Naming.lookup*().

Таким чином, клас клієнта може мати наступний опис (фрагмент 1.10):

Фрагмент коду 1.10

```
import java.rmi.*;
public class HelloClient {
  public static void main(String [] args) throws Exception
    {
      MessageWriter server = (MessageWriter) Naming.lookup(
         "rmi://myhome.csit.fsu.edu/messageservice");
      server.writeMessage(“Hello, other world”);
    }
}
```

Ця програма також виконує дві дії:

– від локального сервера імен одержує посилання на віддалений об’єкт із зовнішнім ім’ям «*MessageWriter*»;

– викликає метод віддаленого об’єкта *writeMessage*() на сервері.

Як параметр у метод *Naming.lookup*() передається URL сервера з тегом протоколу «*rmi*».

У розглянутому прикладі передбачається, що серверна програма, зі створеним віддаленим об’єктом, запущена з домашнього каталогу «*myhome*» на локальному сервері, і об’єкт зареєстрований у реєстрі RMI за замовчуван-

ням на порту 1099 на цьому комп'ютері.

Компіляція файлів *HelloServer* і *HelloClient* виконується звичайним образом компілятором *javac*, наприклад, так:

```
myhome$ javac HelloServer
yourhome$ javac HelloClient
```

Для нормальної роботи програм необхідно або забезпечити клієнтові й серверу спільний доступ до поточного каталогу, або скопіювати всі файли з іменами *MessageWriter*.class* у поточний каталог клієнта «*yourhome*».

Результати роботи програми з викликом віддаленого методу показані на рис. 1.30.

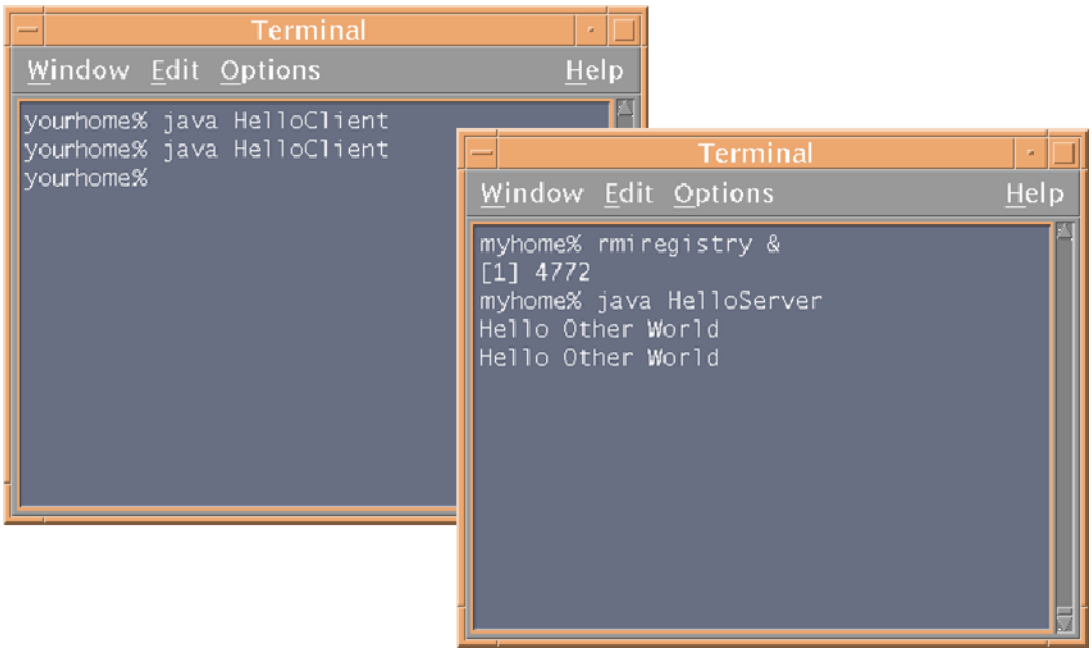


Рисунок 1.30 – Результат виклику методу віддаленого об'єкта

Виклик методу віддаленого об'єкта здійснюється й виглядає, як виклик методу звичайного локального об'єкта. Але механізм роботи віддаленого виклику складнішій. Аргументи, які передаються в метод, самі по собі можуть бути об'єктами довільної складності й повинні бути якимсь образом зібрані в повідомлення, придатні для пересилання по мережі. Аналогічним образом протилежному напрямку по мережі повинні бути відправлені результати ро-

боти методу або об’єкт виключення.

Потужним засобом розподілених об’єктних моделей RMI є те, що посилання на інші віддалені об’єкти можуть бути передані як аргументи методів і вертатися ними як реакція на повідомлення.

Одержавши єдине посилання з реєстру RMI на віддалений об’єкт, клієнт може одержати посилання на інші віддалені об’єкти, які вертаються методами першого. При використанні цього засобу є одне обмеження. Якщо аргумент, який передається до віддаленого методу, або результат, що повертається їм, сам є об’єктом класу, який реалізує інтерфейс *Remote*, то в оголошенні методу повинен вказуватися тип віддаленого інтерфейсу, а не класу, що його реалізує. Спроба явного приведення типу цього посилання до типу класу реалізації віддаленого об’єкта приводить до вкидання виключення типу *ClassCastException*.

Таке поводження посилань свідчить про те, що посилання на віддалені об’єкти, насправді не є посиланнями на об’єкти, які працюють на віддалених віртуальних машинах. Насправді вони є Java посиланнями на якісь локальні об’єкти, які реалізують ті ж віддалені інтерфейси, що й пов’язані з ними віддалені об’єкти. Локальні об’єкти посилаються на об’єкти, які є екземплярами класу заглушки – стабу.

Кожному класу вилученого об’єкта відповідає клас стабу, який реалізує ті ж самі віддалені інтерфейси. Екземпляр класу стабу є необхідний для кожного клієнта. Віддалені виклики на боці клієнта насправді є локальними викликами методу об’єкта класу заглушки.

Як видно, механізм RMI дуже схожий на механізм виклику віддалених процедур. Тут, так само як і в RPC аргументи методу і значення, які повертаються з методу (або виключення), повинні бути перетворені у представлення, яке може бути переданим по мережі. Таке перетворення виконує об’єкт стабу, на машині клієнта. На боці сервера працює система часу виконання, яка виконує прослуховування запитів, що надходять на відповідні IP порти, робить їхню диспетчеризацію й перенаправлення відповідним локальним віддаленим

об’єктам (рис. 1.31). У цілому завдання перетворення й серіализації об’єктів Java є не досить тривіальним завданням.

Ілюзія простоти віддаленого виклику методів досягається завдяки «розумним» бібліотекам і відносно простому компілятору *rmic*. Компілятор є основою технології RMI і генерує додатковий код для класів, які реалізують вилучені інтерфейси. Цей додатковий код дозволяє віддалено викликати методи.

Рисунок 1.31 – Механізм віддаленого виклику методів

Вхідним потоком компілятора *rmic* слугує вихідний код класу що реалізує віддалений інтерфейс *Remote*. У потік на виході вміщується вихідний код нового класу заглушки, який реалізує ті ж інтерфейси віддалених викликів, що й вхідний клас. Методи нового класу містять код для відправлення аргументів і одержання результатів роботи віддаленого об’єкта, мережна адреса

якого зберігається в екземплярі заглушки.

Виклик компілятора *rmic* у командному рядку, як зазначено вище, для компіляції класу *MessageWriterImpl* (ключ *-keep* передбачає збереження коду вихідного класу) приводить до створення коду класу заглушки на зразок такого: (фрагмент 1.11)

Фрагмент коду 1.11

```
public final class MessageWriterImpl_Stub extends
         java.rmi.server.RemoteStub implements MessageWriter,
         java.rmi.Remote {
  public MessageWriterImpl_Stub(java.rmi.server.RemoteRef ref)
    { super(ref); }
  public void writeMessage(java.lang.String $param_String_1)
                throws java.rmi.RemoteException {
    try { ref.invoke(this, $method_writeMessage_0,
                  new java.lang.Object[] {$param_String_1},
                  4572190098528430103L); }
  }
}
```

Клас стабу містить успадковане поле *ref* об’єкта типу *RemoteRef*. По суті, клас заглушки являє собою просто оболонку для цього віддаленого посилання. Віддалені методи викликаються за допомогою методу *invoke*() цього об’єкта. При цьому передається масив об’єктів класу *Object*, який містить оригінальні аргументи, на виході він також повертає об’єкт класу *Object*. Крім того передаються аргументи для ідентифікації конкретного методу, який буде викликаний на сервері.

По суті оболонка слугує для забезпечення безпеки типів під час компіляції. Фактична ж робота виконується бібліотечними класами, які заздалегідь нічого не знають про типи часу компіляції.

Об’єкти, передані як аргументи в метод *invoke*() повинні бути перетворені для передачі по мережі. Java для перетворення об’єктів (і груп об’єктів) до зовнішнього представлення, яке згодом може бути прочитане й відновлено на будь-який JVM, надає спеціальний механізм серіалізації об’єктів. Інструментом такого механізму слугує реалізація ще одного порожнього інтерфейсу

– *Serializable*. Клас об’єкта, переданого по мережі за допомогою бібліотечного методу *writeObject*() (власне цей метод і здійснює серіалізацію), повинен реалізовувати інтерфейс *Serializable*. Якщо ця умова не підтримується, то генерується виключення типу, *NotSerializableException*.

Багато (мабуть більшість) допоміжних класів стандартної бібліотеки Java можуть виконувати серіалізацію. Масиви серіалізуються, якщо цю властивість мають всі їхні елементи.

У загалі будь-яке об’єктне значення аргументу або результату віддаленого методу повинні реалізовувати інтерфейс *Remote* або *Serializable*. Якщо аргумент або результат реалізує *Remote*, він може бути ефективно переданий віддалено за посиланням. Якщо він реалізує *Serializable*, він передається й копіюється по мережі.

У принципі все, про що говорилося вище, повністю приховано від користувача усередині реалізації RMI. На практиці, для успішного створення додатка RMI, досить знати тільки основні фундаментальні основи цього механізму. Необхідно знати про існування об’єктів стабу, і про основи роботи об’єкта серіалізації. Програміст повинен знати, що посилання на віддалені об’єкти, як правило, виникають шляхом створення об’єкта заглушки на сервері, а потім стаб передається до реєстру й клієнтів у послідовній формі.

ПЕРЕЛІК РИСУНКІВ

ПЕРЕЛІК ТАБЛИЦЬ

АЛФАВІТНИЙ ПОКАЖЧИК

Навчальне електронне видання

РОЛЬЩИКОВ ВАДИМ БОРИСОВИЧ

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Конспект лекцій

Видавець і виготовлювач
Одеський державний екологічний університет
вул. Львівська, 15, м. Одеса, 65016
тел./факс: (0482) 32-67-35
E-mail: info@odeku.edu.ua
Свідоцтво суб'єкта видавничої справи
ДК № 5242 від 08.11.2016