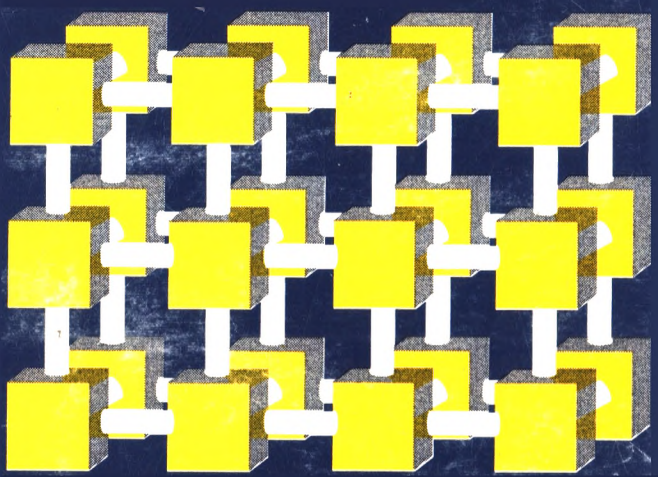
Томас Бройнль

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ

Початковий курс



Навчальне видання

Thomas BRÄUNL

Parallele Programmierung

Eine Einführung

Vieweg
Braunschweig/Wiesbaden
1993

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ

Початковий курс

Томас Бройнль

ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ

Початковий курс

Рекомендовано Міністерством освіти України як навчальний посібник для студентів вищих закладів освіти, які навчаються за спеціальностями "Комп'ютерні інтелектуальні системи та мережі", "Програмне забезпечення обчислювальної техніки та автоматизованих систем"

Київ
"Вища школа"
1997

УДК 519.68=30=03=83(07)

ББК 32.973-01я73
Б 88

Переклав з німецької *В.А. Святний*
Вступне слово *Андреаса Ройтера*

Бройнль Томас

Б88 Паралельне програмування: Початковий курс: Навч. посібник / Вступ. слово А.Ройтера; Пер. з нім. В.А.Святного. – К.: Вища шк., 1997. – 358 с.: іл.
ISBN 5–11–004717–0.

Текст складається з чотирьох розділів. Після основ паралельних обчислень викладено два головних види паралельного програмування – "звичайне" асинхронне та синхронне "масивно паралельне" (паралельне за даними) програмування з тисячею та більше процесорів. Четвертий розділ присвячено іншим паралельним моделям, які не можуть бути віднесені до двох зазначених видів програмування, а також питанням автоматичного розпаралелювання та векторизації, аналізу ефективності паралельних систем.

Для студентів вищих закладів освіти, які навчаються за спеціальностями "Комп'ютерні інтелектуальні системи та мережі", "Програмне забезпечення обчислювальної техніки та автоматизованих систем". Може бути корисним для спеціалістів, що цікавляться проблематикою паралельних обчислювальних систем.

Б $\frac{2404010000-014}{211-97}$ ББК 32.973-01я73

ISBN 5–11–004717–0

Вступне слово

Паралельні обчислювальні системи – це надзвичайно актуальні теми як у науці, так і в галузі розроблення програмних продуктів. Надії та сподівання, пов'язані з паралельними комп'ютерними архітектурами, мають різноманітний характер. Одні сідають за паралельну систему тому, що тільки вона може задовольнити потребу проблем комп'ютеризації наукових розрахунків в неосяжній обчислювальній продуктивності. Це стосується проблем у галузях струмової механіки, оптимізації складних систем, прогнозу погоди, моделювання найрізноманітніших технічних і природних процесів – задачі, що часто узагальнюються поняттям "grand challenges". Ці проблеми поєднує те, що швидкість обчислень, яка потрібна для вирішення проблеми реальних розмірів за прийнятний інтервал часу, не може бути досягнута у векторних машинах традиційної структури і тільки декомпозиція загальної проблеми на незалежно оброблювані блоки і паралельне вирішення їх на відповідних обчислювальних структурах обіцяє шанси на успіх. Інші групи користувачів не мають жодної потреби в обчислювальній надпродуктивності, але все ж таки цікавляться паралельними обчислювачами, бо вони чекають, що їхні завдання в галузях банківської справи, оброблення зображень тощо можуть бути розв'язані завдяки розпаралелюванню на багатьох малих, дешевих обчислювачах економічніше, ніж на відповідно більшій і набагато дорожчій великій обчислювальній машині. Ці групи користувачів, таким чином, чекають від паралельних засобів кращого співвідношення між ціною та продуктивністю. Паралельні обчислювальні системи мають, природно, й інші потенційні переваги, а саме: зумовлену дуже високою редундантністю компонентів підвищену дійову готовність, кращі показники в реакції системи на запити завдяки динамічному розподілу

завантаження і т. ін. Ці аспекти відіграють у загальній проблемі побіжну роль, тому і тут ми не будемо їх враховувати. Ми зосередимося тільки на двох названих вище перевагах: на принципово будь-якому збільшенні обчислювальної спроможності завдяки сумісному включенню в роботу відповідної кількості обчислювальних вузлів і на скороченні грошових витрат шляхом заміни великої ЕОМ мікропроцесорами, що працюють паралельно. Дискусії в цій галузі часто ведуться в такому дусі, що ніби для реалізації названих переваг нічого, крім правильного балансу між швидкістю процесорів, розмірами локально доступної пам'яті, смугою частот пропускання мережі міжпроцесорних зв'язків та тривалістю обміну повідомленнями між двома будь-якими процесорами, не потрібно. Все це, безумовно, являє собою цікаві й до останнього часу дуже складні технічні проблеми, про що свідчать численні концепції, які застосовуються в різних прототипах і зразках паралельних систем.

Поряд із цим часто не звертають належної уваги на те, що перехід від ЕОМ традиційної побудови – чи то векторна машина для чисельних задач чи велика машина для комерційних застосувань – до паралельної системи потребує значно більшого, ніж просто заміни машини і нового перекладу всіх наявних програм. Це означає передусім перехід від звичайного послідовного до паралельного програмування, і цей крок відповідно до сучасного стану техніки тільки частково може бути виконаний з комп'ютерною підтримкою. Більша частина цієї роботи має виконуватися самими користувачами, а це означає, що, якщо користувач хоче мати шанси на успіх, то необхідно зайнятися грунтовним вивченням усіх проблем, пов'язаних з паралельними комп'ютерними архітектурами і їх застосуванням. Це може бути реалізовано тільки за умови, що різні державні програми підтримки розробок і впровадження паралельних обчислювальних структур як на національному, так і на міжнародному рівнях поставлять собі за мету поряд з чисто технічними засобами забезпечити підтримку освітніх програм для користувачів наявних обчислювальних архітектур.

Сьогодні, а також у недалекому майбутньому властивості паралельної системи не можуть бути дані програмісту в абстрактному вигляді в такій мірі, щоб він міг працювати так, як у звичайному своєму середовищі. Хто хоче цілком використати переваги паралельного комп'ютера, той повинен передусім добре розуміти принципи розпаралелювання. Це тим більше важливо, що в багатьох випадках потрібно не тільки перепрограмувати наявні алгоритми для паралельного використання їх, а передусім розробити для вирішуваної проблеми нові алгоритми, які в достатній мірі були б придатні до розпаралелювання.

Пропонованій книзі пана доктора Бройнля, на щастя, притаманне розуміння вказаних проблем. На відміну від багатьох інших навчальних посібників на тему "паралельний комп'ютер" книга не концентрується на апаратних аспектах; хоч автор і дає короткі відомості про найважливіші варіанти архітектур, принципи обміну інформацією між процесорами і т.ін., він все ж обмежується тим, щоб читачеві, який цікавиться технікою паралельних систем, дати основні ідеї, яким чином процесори, що працюють автономно, взаємодіють між собою, обмінюються даними і які при цьому виникають переваги та недоліки. Книга також не зосереджується на розпаралелюванні числових проблем; з цього питання є інші повнозмістовні стандартні посібники. Праця пана доктора Бройнля ставить за мету подати проблеми і принципи програмування в середовищі, де вирішення будь-якої проблеми здійснюється кооперацією деякої кількості процесів або процесорів, що працюють автономно. Це потребує розуміння певних фундаментальних механізмів комунікації та синхронізації разом з формальними засобами описування їх, визначення коректності. Далі на їхній основі можуть бути визначені програмні примітиви, які майже гарантують взаємне невтручання процесів у захищені

структури даних, забезпечують безумовне виконання повних паралельних послідовностей процесів тощо. Ці аспекти пан доктор Бройнль показує передусім на прикладі перетворення таких програмних примітивів у різних мовах програмування, які призначені для прямого відображення паралельності. На більш високому рівні абстракцій перебувають різноманітні програмні моделі реалізації паралельності, які застосовуються в різних машинних архітектурах або мовах з відповідними програмними середовищами. Найсуттєвішим у сучасному стані дискусій є розподіл на асинхронну паралельність, що притаманна MIMD-архітектурам і на синхронну паралельність, що презентується SIMD- або SPMD-системами. Тут пан Бройнль досить докладно описує перетворення програмної моделі в різні мови програмування, ілюструє застосування програмних моделей на цілому ряді ретельно відібраних прикладів. В останньому розділі дискутується цілий ряд безумовно важливих аспектів паралельності, таких як підходи до мов програмування, що забезпечують автоматичне розпаралелювання, функції паралельності в нейронних мережах, а також постійно актуальні питання точного визначення продуктивності або її підвищення за рахунок застосування паралельних систем. Цей останній розділ особливо заслуговує на увагу з двох причин: по-перше, автор займається тут темами, які дуже часто ігноруються в книгах, присвячених паралельним ЕОМ. Це стосується передусім паралельної обробки інформації в системах банків даних та автоматичного розпаралелювання в таких мовах, як SQL. По-друге, в підрозділі про оцінювання продуктивності паралельних систем ясно доведено, що Speedup (прискорення) не є мірою всіх характеристик, що в багатьох випадках інші міри, такі як Scaleup (масштабний здобуток), дають змогу одержати набагато реальніші (та цікавіші) відомості про потенційну продуктивність паралельних обчислювальних систем. Порівняльний аналіз взаємовідносин між різними мірами продуктивності має бути неодмінно рекомендований тому, хто критично розглядає часто неадекватне застосування показників Speedup.

Головному призначенню книги – бути вступом до принципів та проблем паралельного програмування – відповідає цілий ряд ретельно розроблених завдань для програмування, які для всіх зацікавлених представлені також в електронній формі. Я б пропонував усім, хто читає цю книгу самостійно без слухання лекцій та інших засобів навчання, принаймні спробувати розв'язати ці завдання. Це значно полегшить та підтримає розуміння принципів, що подані у тексті книги. Одночасно книга є дуже хорошою основою для постановки курсу лекцій "Паралельне програмування", а також для організації практикуму з цієї тематики.

Я вже наголошував, що швидке та успішне використання потенційної продуктивності параллельних обчислювальних систем залежить не в останню чергу від того, наскільки швидко та грунтовно будуть залучені до цієї нової технології користувачі-програмісти з різних галузей науки і техніки. Це потребує багато зусиль у зовсім різних галузях – від надання відповідних обчислювальних систем шляхом інсталяції мережних інфраструктур, що забезпечать доступ до цих систем з робочих місць науковців, до імплементації відповідних програмних середовищ з візуальними засобами тестової допомоги. Проте в центрі цих зусиль мають бути курси з лекціями, практикою та іншими формами навчання, які б наблизили до програмістів потрібну для використання паралельності модель мислення і сприяли б тому, щоб вони змогли перетворити цю модель у відповідну техніку програмування. У вирішення цього завдання книга робить виключно цінний внесок. Вона охоплює у відносно стислому обсязі всі важливі аспекти проблеми, утримується від зайвих формалізацій там, де без них можна обійтися, та поглиблює зміст матеріалу завдяки прикладам з беспосередніх практичних використань паралельності. Треба розглядати як щасливий випадок, що ця праця виходить не тільки

німецькою, а й англійською, українською та російською мовами. Я бажаю книзі широкого кола читачів, а читачам – творчих успіхів у спробах використати здобуті знання в паралельних програмах.

Андреас Ройтер

Передмова

Ця книга дає потрібні початкові знання, вводить у галузь паралельного програмування і орієнтована передусім на студентів-інформатиків старших курсів. Тематика книги розподілена на чотири великих частини. Після основ викладаються два головних розділи паралельного програмування – "звичайне" асинхронне програмування та синхронне "масивно паралельне", або паралельне за даними програмування з тисячею та більше процесорів. Заключна частина присвячена іншим паралельним моделям, які не можуть бути віднесені до двох зазначених видів програмування, а також питанням автоматичного розпаралелювання та векторизації, аналізу ефективності паралельних систем.

Книга грунтується на курсі лекцій того самого найменування, який уперше прочитаний мною в зимовому семестрі 1990/91 навчального року в Штутгартському університеті. Практичні заняття та семінари доповнили матеріал курсу вправами, що орієнтовані на його розділи. Англійський переклад цього тексту був зроблений Б.Блайвінсом у співдружності з автором, переклад на українську та російську мову виконано професором В.А.Святним та його співробітниками.

Я висловлюю особливу подяку професору доктору Андреасу Ройтеру за його підтримку та ініціативу в підготовці книги, Астрід Бек, Брайану Блайвінсу, Стефану Енгельгардту та передусім Клаусу Бреннеру за коректурні читання рукопису та численні пропозиції щодо поліпшення його змісту, а також Христині Драбек та Хартмуту Келлеру за внесення першої версії рукопису в комп'ютерну текстову систему. Я дякую також професору доктору Юргену Немеру, професору доктору Евальду фон Путткамеру та професору доктору Каі Хвангу, лекції яких дали мені наснагу для роботи над цією книгою.

Щира подяка також співробітникам і студентам, які

з великим ентузіазмом та працездатністю перетворили використані в книзі базові концепції програмування на мовах Modula-P та Parallaxis в програмні середовища, що в цей час перебувають в стадії запровадження як Public-Domain-Software в ряді країн світу. Інго Барт, Франк Сембах і Стефан Енгельгардт розробили повне програмне середовище з компілятором, симулятором та редактором помилок для мови Parallaxis, а Роланд Норц розробив компілятор для мови Modula-P.

Всі, хто хотів би випробувати самостійно наведені в книзі приклади паралельних алгоритмів, можуть безкоштовно зняти копії компіляторів і систем моделювання обох паралельних мов – Modula-P і Parallaxis– через мережу Internet за допомогою "анонімного ftp". Internet-адреса обчислювальної системи має вигляд: *ftp.informatik.uni – stuttgart.de*, назви матеріалу для копіювання: *pub/modula-p* та *pub/parallaxis*.

Томас Бройнль

Передмова до українського видання

Одним з основних напрямів збільшення ефективності засобів цифрової обчислювальної техніки є пошук архітектурних рішень, які б сприяли реальному переходу від послідовної машини фон Ноймана до паралельного виконання основних та допоміжних процесів в обчислювальних системах. В останні роки ці пошуки привели до появи на комп'ютерному ринку ряду паралельних обчислювальних систем MIMD- та SIMD-структур. За оцінками експертів, в найближчий час ці паралельні системи стануть основною силою на верхніх сходинах ієрархії обчислювальних мереж. Цей факт викликав неабиякий інтерес до засобів паралельного програмування – як системних, так і прикладних. Виявилося, що потужні паралельні системи не можуть використовуватися так широко, як їхні попередники, через брак "дружніх до споживачів" засобів програмного забезпечення. Паралельними системами здебільшого користуються тільки професіонали-програмісти, які розв'язують системні задачі інформатики. Практично немає підручників та навчальних посібників, які б уводили в коло споживачів паралельних систем студентів та науково-інженерний персонал як фаху "інформатика", так і інших напрямів діяльності. Ця ситуація, що склалася за кордоном і передусім у країнах Західної Європи, для України виглядає набагато гостріше: ми не маємо діючих паралельних систем, побудованих за новітньою технологією, у нас немає можливості займатися практичним паралельним програмуванням навіть для тих паралельних алгоритмів і структур систем, які були розроблені нашими вченими та інженерами в процесі досліджень у цій галузі.

Запропонована українським читачам книга Томаса Бройнля "Паралельне програмування" (початковий курс) певною мірою зможе допомогти у вирішенні названих вище актуальних проблем. По-перше, вона є перевіреним

практикою навчання студентів підручником з повнозмістовним набором тем і актуальним відображенням світового стану теоретичних досліджень та практичних результатів у галузі структур і програмного забезпечення паралельних обчислювальних систем. Ця особливість книги дасть змогу студентам та тим фахівцям, що тільки-но звернули увагу на цю перспективну проблематику, швидко увійти в коло задач та досягнень теорії і практики паралельного програмування, знайти сферу можливої особистої діяльності в цій галузі. По-друге, матеріал книги стимулює фахівців, що мають певний досвід у розробленні паралельних алгоритмів, програм, структурних та системних рішень паралельної обчислювальної техніки, до постановки нових задач, що можуть безумовно виникнути після ознайомлення з наведеним тут аналізом публікацій з даної проблематики, із сферами застосування паралельних систем, з альтернативними вирішеннями вже відомих проблем, які потребують паралельних ресурсів обчислювальної техніки. По-третє, автор книги доктор інформатики Томас Бройнль має особистий солідний досвід у розробленні системного параллельного програмного забезпечення: головним результатом його наукової діяльності є паралельні середовища програмування Modula-P та Parallaxis, які визнані фірмами і фахівцями світу. Погляд висококваліфікованого програміста-"паралельника" на цю проблематику відчувається в усіх розділах книги.

Українське видання книги Т.Бройнля допоможе викладачам та студентам нашої держави в поступовому переведенні навчального процесу на українську мову, сприятиме розвитку творчих дружніх зв'язків з німецькими колегами. Робота над перекладом та підготовкою рукопису книги до друку стала однією з частин наукового співробітництва між Донецьким технічним та Штутгартським університетами в напрямах розроблення та запровадження масивно паралельних моделюючих середовищ для складних динамічних систем з концентрованими та розподіленими параметрами, у здійсненні технічних та програмно–системних передумов теледоступу в обчислювальну мережу факультету інформатики Штутгартського університету, у використанні паралельних систем MasPar, Intel Paragon та ін. в наукових дослідженнях та навчальному процесі ДонДТУ. Від імені творчого колективу висловлюю щиру подяку професорові Андреасу Ройтеру, засновникові інституту паралельних та розподілених надпотужних обчислювальних систем (IPVR), почесному докторові ДонДТУ за його вирішальний вклад в організацію цього співробітництва, підтримку наших ініціатив щодо видання книги, за товариську допомогу в проведенні наукової роботи співробітниками ДонДТУ в IPVR.

Авторові книги Т. Бройнлю я дякую за його персональну активну участь у нашій спільній роботі над паралельними моделюючими середовищами, його дружню допомогу під час моєї наукової роботи в IPVR.

Підготовку рукопису до друку ретельно виконали А.В.Молдованов та О.М.Розанов, за що їм вдячні автор та перекладач.

В.А. Святний

I

ОСНОВИ

Паралельність може розглядатися з різних боків. Паралельні ЕОМ можуть класифікуватись відповідно до їхніх машинних структур, у той час як паралельні операції відрізняються відповідними їм рівнями абстракцій або типами аргументів. Це дає принципово різні погляди на тему "паралельність". Виходячи з цих поглядів, можна кваліфікувати послідовне (звичайне) програмування як спеціальний вид паралельного програмування. Мережі Петрі є ефективним допоміжним інструментом для визначення та наочного зображення асинхронно паралельних процесів. Вони допомагають з'ясувати взаємодії та усунути можливі проблеми ще до того, як розробка деякої паралельної системи реалізується у вигляді паралельної програми. Мовні концепції роботи з паралельністю, що найчастіше вживаються, а також найважливіші структури комутацій, що дістали застосування в паралельних ЕОМ, мають фундаментальне значення для розуміння проблем паралельного оброблення даних.

1. ПАРАЛЕЛЬНІСТЬ : ЗАГАЛЬНІ ПОЛОЖЕННЯ

Всесвіт паралельний! Цю принципову точку зору можна підтвердити багатьма прикладами. Чи йде мова про явища в природі, про складні технічні процеси чи навіть про зміни в суспільстві, завжди вони надзвичайно паралельні. Ріст рослини залежить від великої кількості факторів, що діють одночасно, запуск двигуна потребує паралельної взаємодії багатьох його компонентів, біржеві курси встановлюються одночасними відносинами тисяч продавців та покупців. Паралельність має цікаві прояви не тільки в конкретному фізичному світі, а й в абстрактних процесах.

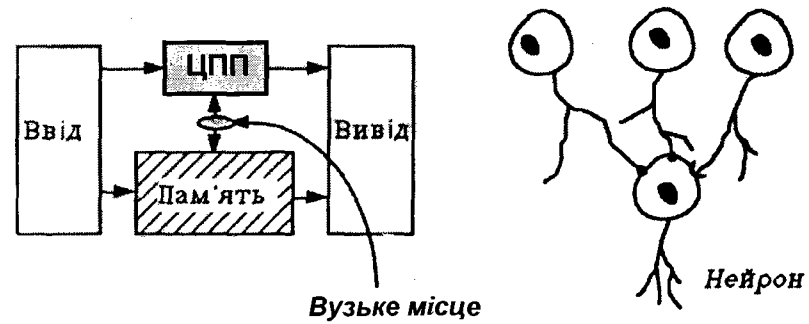


Рис. 1.1. Машина фон Ноймана та нейрон

Паралельні суперкомп'ютери належать до найновітніших розробок в інформатиці. Проте не можна не бачити, що власне в простому персональному комп'ютері паралельно виконується чимало операцій: до них належать операції вводу/виводу через канали безпосереднього доступу в пам'ять, паралельного керування різними функціональними блоками з боку центрального процесорного пристрою за допомогою мікрокоманд. Якщо розглядати арифметичний пристрій як сукупність самостійних однорозрядних елементів, то можна зробити

висновок, що 16-розрядна арифметика працює паралельно на бітовому рівні.

Людський мозок працює також паралельно. На рис.1.1 зображені для порівняння послідовна ЕОМ фон Ноймана та нейрон мозку людини.

Рис.1.2 показує в числах незбалансованість взаємовідносин між єдиним активним центральним процесорним пристроєм машини фон Ноймана з багатьма пасивними елементами пам'яті. На противагу цьому в гомогенній структурі мозку всі елементи його постійно активні.

Машина фон Ноймана	**Людський мозок**
Кількість елементів: *≈ 10^9 транзисторів для:* *1 CPU (10^6 транз.), пост. акт.* *10^9 елементів ОП, часто неакт.*	*≈ 10^{10} нейронів ("ЦПП+ОП") майже постійно активні*
Тривалість перемикання: *10^{-9} с = 1нс*	*10^{-3} с = 1мс*
Кількість перемикань за 1с: *10^{18} (теоретичне число)* *10^{10} (практ. число через неактивні елементи ОП)*	*10^{13}*

Рис. 1.2. Порівняння продуктивності

Перевага мозку порівняно з ЕОМ фон Ноймана полягає в більшій кількості перекомутацій за секунду (в 1000 разів), причому вища, ніж у транзистора, складність функції комутації нейрона навіть не бралася до уваги. Причиною більш високої продуктивності мозку є паралельне оброблення інформації.

З наведеного вище випливає теза про те, що паралельне оброблення є натуральною формою роботи з інформацією. Послідовність, що домінує сьогодні в

програмуванні, є просто історично зумовленим наслідком застосування моделі ЕОМ фон Ноймана. Ця модель була надзвичайно успішною, але передбачена в ній послідовність як принцип функціонування зостається її штучним обмеженням. Часто постановка задачі, що годиться для паралельного виконання, описується та вирішується засобами мови паралельного програмування простіше, ніж мовою послідовного програмування. Паралельне формулювання будь-якої проблеми має вищий інформаційний зміст, якщо порівнювати з аналогічним послідовним варіантом.

Таким чином, мета полягає в тому, щоб паралельні постановки задач описувати паралельною мовою програмування, що дасть більш зрозумілі програми і суттєво прискорить виконання їх у паралельній ЕОМ (рис. 1.2).

Спробі застосування паралельних ресурсів передують, як правило, два запитання: "Навіщо застосовується паралельне вирішення задачі?" та "Як воно застосовується?"

Відповідь на перше запитання знаходимо відразу: крім природної мети швидше одержати результати ми можемо мати справу з проблемами, для розв'язання яких не вистачає ресурсів звичайних послідовних ЕОМ. Два приклади із сучасної практики наочно це ілюструють.

Приклад 1. Моделювання потоків струменевої механіки потребує декількох діб роботи однопроцесорних ЕОМ, тому залучення паралельних методів та обчислювальних систем є ефективним засобом побудови працездатних моделей.

Приклад 2. Керування автономними рухомими об'єктами (наприклад, роботами) на основі інформації, що надходить з телекамери, потребує паралельного її оброблення з двох причин. По-перше, потрібне дуже швидке (паралельне, оперативне) розпізнавання образів для

того, щоб можна було тримати тривалість циклу на рівні 40 мс (що відповідає частоті повтору кадрів 25 Гц). По-друге, система керування має асинхронно реагувати на несподівані події, що неможливо для послідовно працюючої програми.

Розпаралелювання проблеми полягає в її розподілі на менші, по можливості мало залежні між собою частини (це щодо питання "ЯК?"). Ці частини можуть мати величину процедур (*великоблокова паралельність*) або складатись лише з елементів деякого вектора (*малоблокова, масована паралельність*). Для роботи з кожною частиною проблеми передбачається окремий процесор (рис. 1.3).

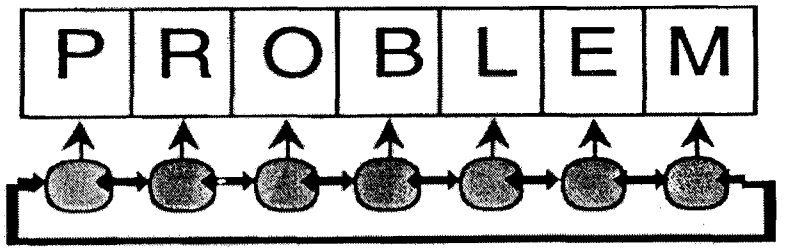


Рис. 1. 3. Декомпозиція проблеми

Проте розпаралелювання проблеми полягає не тільки в її розподілі між наявними процесорами. Визначені частини певною мірою залежні між собою, а результати роботи процесорів треба якось звести до загального результату. Тому виникає потреба в цілеспрямованому погодженні (синхронізації) роботи процесорів та обміні даними між ними. Ці функції виконує спеціальна комунікаційна мережа (на рис. 1.3, наприклад, процесори з'єднані в кільце). Від завантаженості та пропускної спроможності цієї мережі великою мірою залежить загальна продуктивність паралельної обчислювальної системи. Щоб досягти максимального ефекту від розпаралелювання, треба вже на етапі розподілу проблеми брати до уваги властивості наявної комунікаційної мережі. На жаль, розподіл проблеми та паралельне вирішення її частин ще не гарантують більш швидкого одержання загального результату. Ефект залежить від співвідношення між добутком за рахунок паралельної роботи процесорів та втратами, які неодмінно супроводжують синхронізацію і обмін даними. Тому треба підкреслити, що не існує ні універсальних по відношенню до будь-яких проблем паралельних ЕОМ, ні ідеальних паралельних алгоритмів. Кожний клас паралельних систем має з огляду на вирішувану проблему свої переваги та недоліки. Паралельні алгоритми мусять бути узгоджені (принаймні на процедурному рівні) як з вирішуваною проблемою, так і з моделлю паралельної системи, яка буде застосовуватись для реалізації алгоритмів. Правильному виборові паралельних структур залежно від вирішуваних проблем та пошукові відповідних паралельних алгоритмів ми приділимо належну увагу в наступних главах книги.

2. КЛАСИФІКАЦІЇ

У цьому розділі розглядаються три різні класифікації паралельних обчислювальних систем: за принципами їх побудови, за рівнями абстракції паралельних обчислювальних процесів, за типами аргументів паралельних операцій.

2.1. Класифікація обчислювальних машин

Класифікація ЕОМ, запропонована Флінном [Flynn 66], розподіляє світ комп'ютерів на чотири групи: SISD, SIMD, MIMD та MISD (рис. 2.1.). Для нас передусім цікаві два класи: SIMD (синхронна паралельність) та MIMD (асинхронна паралельність). В групі SISD йдеться про однопроцесорні машини фон Ноймана, а машини типу MISD можуть розглядатись як конвеєрні ЕОМ.

За умови синхронної паралельності маємо тільки один керуючий потік, бо саме один високопродуктивний

процесор опрацьовує програму, а всі інші, простіші за побудовою процесори, виконують одночасно, синхронно команди цього процесора.

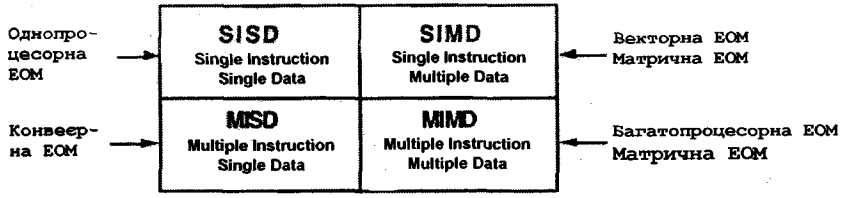


Рис. 2. 1. Класифікація засобів обчислювальної техніки за Флінном

Асинхронна паралельність має декілька керуючих потоків, зумовлених тим, що кожний процесор виконує власну програму. Для обміну даними між будь-якими двома процесорами, що працюють незалежно й асинхронно, має бути забезпечена синхронізація.

MIMD- та SIMD-класи паралельних ЕОМ можуть бути розділені кожний на два підкласи відповідно до типу зв'язку між процесорами, а саме:

MIMD:

- тісний зв'язок через загальну пам'ять, багатопроцесорна ЕОМ (обчислювальна система);
- розподілена обчислювальна система з нетісним зв'язком через мережу комутації та обміном повідомленнями;

SIMD:

- векторна ЕОМ (обчислювальна система) без безпосереднього зв'язку між процесорними елементами;
- матрична (масивна) ЕОМ зі зв'язком між процесорними елементами через комутаційну мережу.

Розгляньмо детальніше паралельні конвеєрні ЕОМ, MIMD- та SIMD-системи.

Конвеєрна ЕОМ

Принцип побудови простої конвеєрної ЕОМ показано на рис.2.2.

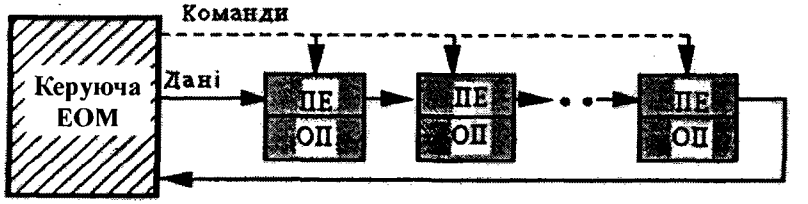


Рис. 2.2. Конвеєрна ЕОМ

Часові діаграми на рис.2.3 пояснюють дискретне опрацювання програмних інструкцій в ЕОМ з одним конвеєром, причому тут дається порівняння послідовного і конвеєрно-паралельного виконання програм.

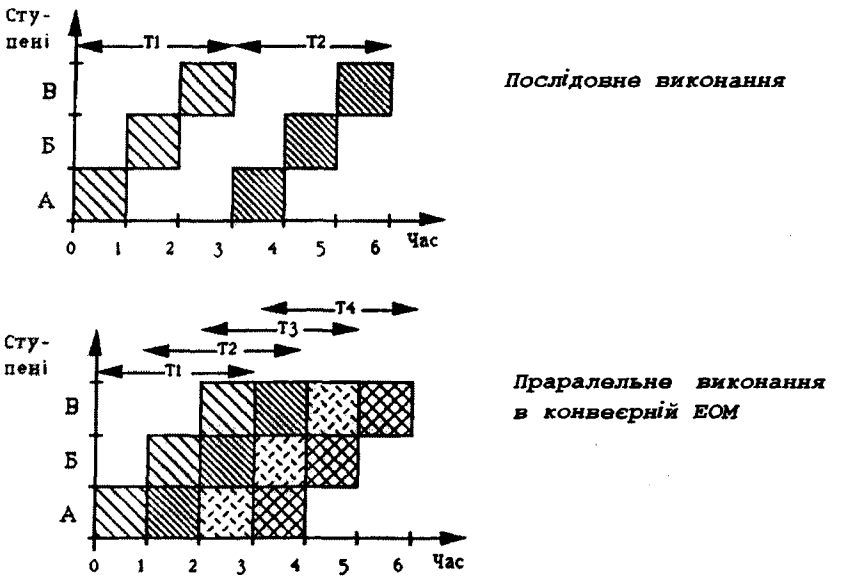


Рис 2.3. Виконання програми в конвеєрній ЕОМ

Конвеєр має три ступеня (*А*, *Б* і *В*), які, наприклад,

можуть відповідати операціям "ввести дані x та y", "помножити x на y", "додати результат множення до числа S". Якщо ця послідовність операцій буде виконуватися в ітеративному циклі, то при послідовному виконанні загальна тривалість розв'язання задачі дорівнює сумі інтервалів $T1+T2$ (рис.2.3., верхня діаграма), тоді як при конвеєрно-паралельному виконанні після початкової фази ініціалізації (завантаження конвеєра) в кожному часовому інтервалі закінчується вся послідовність операцій. Завдяки цьому n-ступеневий конвеєр після фази завантаження (n–1 крок) на кожному кроці досягає результату, який відповідає показнику паралельності n. Конвейєри – це спеціальні машинні структури і тому можуть застосовуватися для розв'язання задач певного класу. Ці задачі завжди відрізняються послідовностями операторів, що можуть трапитися в програмних циклах; навряд чи можна тут уникнути залежностей від порядку операторів, а довжина послідовностей операторів має бути узгоджена з довжиною конвеєра. Вичерпну інформацію про побудову конвеєрних ЕОМ подано в книзі [Hockney, Jesshope 88].

Простий конвеєрний комп'ютер з одним конвеєром має також один керуючий потік, отже, він відповідає однопроцесорному комп'ютеру з допоміжним векторно-конвеєрним процесорним пристроєм (див. рис.2.2). При наявності багатьох функціонально незалежних конвеєрів можна говорити про MIMD-систему з конвеєрами (мульти-конвеєрна система).

MIMD (multiple instruction, multiple data)

Обчислювальні машини цього класу мають порівняно з SIMD більш загальну структуру і працюють завжди асинхронно. Кожний процесор виконує свою самостійну програму і відповідно має свій власний потік керуючих команд. Це означає, що в MIMD-системі мають місце різні керуючі потоки. Ці системи поділяються на MIMD-ЕОМ із загальною пам'яттю (рис.2.4) та MIMD-ЕОМ без загальної пам'яті (рис.2.5).

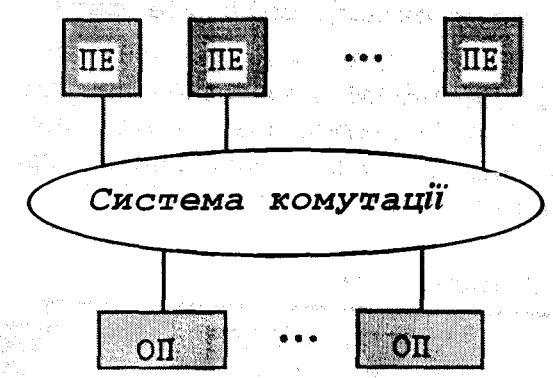


Рис. 2.4. MIMD-система із загальною пам'яттю

Системи із загальною пам'яттю називають "тісно зв'язаними". Синхронізація та обмін даними виконуються через області пам'яті, до яких можуть звертатися різні процесори відповідно до координації їх роботи. MIMD-ЕОМ без загальної пам'яті називають "слабко зв'язаними".

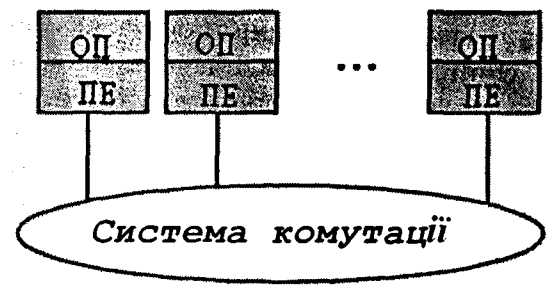


Рис.2.5. MIMD-система без загальної пам'яті

Вони мають локальну пам'ять у кожному процесорному елементі і тому відповідають звичайному уявленню про тісний зв'язок незалежних ЕОМ.

Синхронізація та обмін інформацією в системах без загальної пам'яті потребують значно більших витрат, бо всі види інформації мають проходити через комутаційну мережу.

SIMD (single instruction, multiple data)

Матрична ЕОМ, для якої як синонім часто вживається назва SIMD-ЕОМ, побудована простіше, ніж MIMD-ЕОМ (рис. 2.6). Апаратура циклу команд (читання і декодування команд, ведення лічильника програм) зосереджена тільки в центральній керуючій ЕОМ.

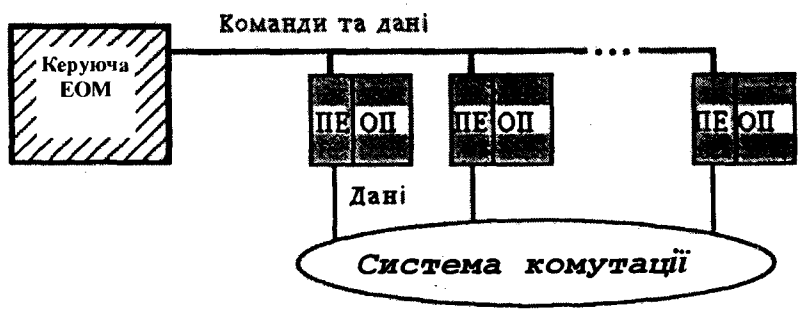


Рис. 2.6. Матрична ЕОМ

Цю ЕОМ часто називають пристроєм матричного керування "array control unit" (ACU) або програмним автоматом, що задає порядок операцій ("sequencer"). Процесорні елементи (ПЕ) мають тільки арифметико-логічний блок (ALU), локальну пам'ять і комунікаційне обладнання для мережі зв'язків між ПЕ. У зв'язку з тим, що для всіх ПЕ наявний єдиний задавач команд, виконання будь-якої SIMD-програми завжди відбувається синхронно. Це означає, що на противагу MIMD-системам тут може бути тільки один керуючий потік: або кожний ПЕ виконує ту саму команду (наприклад, складання), що і всі інші ПЕ, над своїми локальними даними, або він перебуває в пасивному стані.

Векторна ЕОМ (рис. 2.7) має ще простішу побудову, ніж розглянута вище матрична машина, бо в ній немає глобальної комутаційної мережі між процесорними елементами. "Локальні дані" кожного ПЕ тут представлені у вигляді векторного регістра і над ним покомпонентно

виконуються арифметико-логічні операції. Прості види обміну даними між ПЕ, такі як переміщення та кругообмін, виконуються за допомогою спеціальних ліній зв'язку.

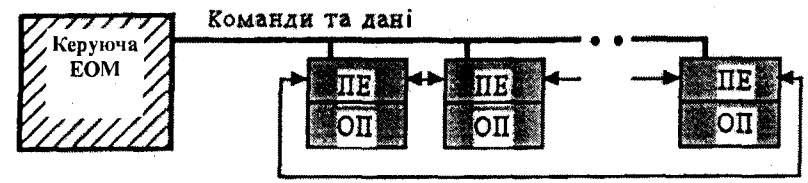


Рис 2.7. Векторна ЕОМ

Гібридні паралельні ЕОМ

З розглянутих класів ЕОМ типу конвеєрних, MIMD- та SIMD-структур може бути утворений цілий ряд систем змішаної форми, деякі з яких тут розглянуті.

Мультиконвеєр

Як уже зазначалося, конвеєрна ЕОМ може мати декілька незалежних між собою конвеєрів і вони працюють паралельно і незалежно. Такі ЕОМ використовують ідею сумісного функціонування MIMD-структури і конвеєрної організації.

Мульти-SIMD

У цій змішаній структурі (MSIMD) передбачено декілька керуючих машин (ACU), кожна з яких несе відповідальність за деяку кількість процесорних елементів (ПЕ). Така побудова системи відповідає ідеї MIMD-подібної структурної організації паралельної роботи багатьох незалежних SIMD-ЕОМ.

Систоличні матричні ЕОМ

Поєднання SIMD-, MIMD- та конвеєрної систем дістало назву "систоличних масивів (матриць)" [Kung,

Leiserson 79]. Тут кожний із матричних елементів є самостійним MIMD-процесором, але він керується центральним задавачем тактів. Уздовж усіх координат матричного масиву відбувається конвеєроподібне оброблення інформації. Це означає, що дані подаються весь час іззовні в паралельний масив процесорів, де вони трансформуються, переміщуються між процесорами далі і видаються з системи (рис. 2.8).

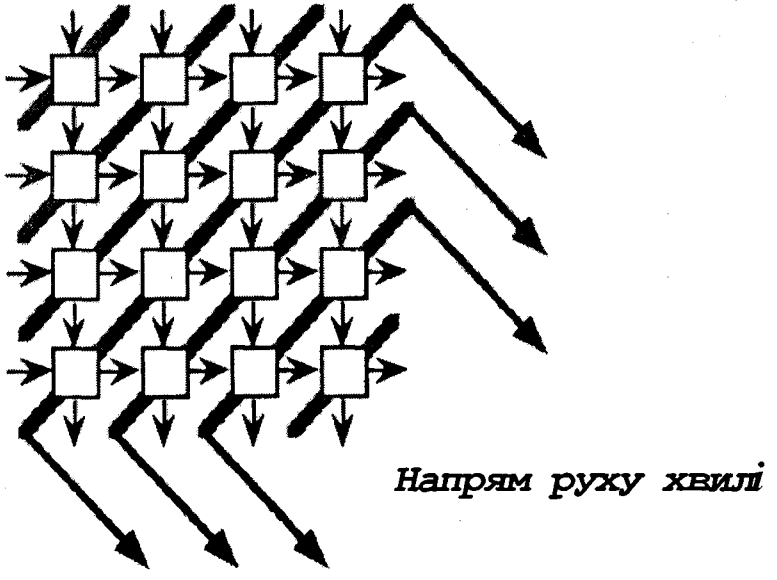


Рис. 2.8. Систоличний масив

Матричні ЕОМ з керуванням потоками даних (Wavefront Array)

Ці системи з'явились як розширення поняття про систоличні паралельні системи [Kung, Lo, Jean, Hwang 87]. Центральний такт, реалізація якого у великих систоличних системах пов'язана з різними труднощами, тут замінено завдяки концепції потоку даних.

Системи з розширеним словом команди (Very Long Instruction Word, VLIW)

Наступною комбінацією MIMD- і конвеєрної ЕОМ є "Very long instruction word", VLIW-EOM та "Multiflow Trace" паралельна ЕОМ [Fisher 84], [Hwang, DeGroot 89]. В цих машинах паралельність досягається, як у горизонтальному мікрокодові завдяки незвичайно широкому формату команд, який забезпечує одночасне незалежне виконання багатьох арифметико-логічних операцій. Йдеться про спеціальне, з перекриттям, виконання скалярних операцій без векторизації.

Цей метод оброблення інформації пов'язаний з рядом проблем, серед яких, наприклад, формування згаданих широких команд (інструкцій) та попереднє розміщення програм у пам'яті ще до (!) умовного розгалуження. Формування формату VLIW з окремих інструкцій відіграє вирішальну роль у розподілі завантаження процесорних ресурсів та досягненні показників розпаралелювання й ефективності паралельних ЕОМ цього типу. Перетворення звичайного послідовного програмного коду в код VLIW може бути здійснене тільки дорогим за розробкою, "інтелігентним" компілятором: справа в тому, що за час, відведений на компіляцію, може бути не знайдено оптимального пакування інструкцій, а це змушує застосовувати евристичні методи. Паралельна ЕОМ цього типу не мала комерційного успіху і більше не з'являється на ринку. Система потерпіла явне фіаско в претензіях на ефективне автоматичне розпаралелювання послідовних програм, на задачу, яка в цій формі, напевно, не може бути розв'язаною.

Same Program Multiple Data (SPMD)

Ця комбінація SIMD і MIMD, яка дістала назву SPMD-моделі [Lewis 91], подає надзвичайні надії на успіх. Як показує назва системи, паралельною ЕОМ керує деяка програма (або точніше – керуючий потік команд), що дає можливість комбінувати простоту SIMD-програмування з

гнучкістю MIMD-EOM. Поряд з тим, що SIMD-модель, попри її природні обмеженості, задовольняє потреби задач багатьох класів, в ній мають місце за рахунок тільки паралельних розгалужень програм типу IF-THEN-ELSE відчутні втрати ефективності, яких можна уникнути при виконанні програм в SPMD-машині. Справа в тому, що в SIMD-EOM процесорні елементи виконують інструкції THEN-групи і ELSE-групи розгалуженої програми послідовно, а в SPMD-EOM – одночасно паралельно, бо тут кожному процесору надається тільки один з можливих шляхів розгалуження (рис. 2.9). Те саме має місце для скільки завгодно виконуваних програмних циклів на різних процесорних елементах. Таким чином, кожний процесор SPMD-машини виконує одну і ту саму SIMD-програму над своїми локальними даними і з своїм власним потоком керуючих команд. При цьому згідно з принципом побудови SPMD-EOM процес оброблення інформації може в будь-який момент перейти від синхронного SIMD-тактування до асинхронної MIMD-організації. На противагу SIMD, в системі SPMD має забезпечуватися синхронізація одного процесора з іншими процесорами тільки тоді, коли за програмою виконується операція обміну даними. До класу SPMD-EOM належить машина CM-5 (Connection Mashine).

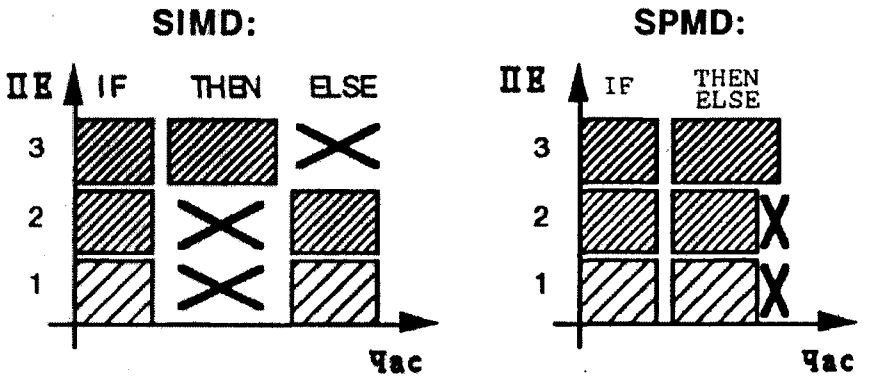


Рис. 2.9. Менша тривалість виконання програм в SPMD-системах

2.2. Рівні розпаралелювання

У багатьох публікаціях і передусім в [Kober, 88] наведено класифікацію паралельності систем за її рівнями (рис.2.10.), що відрізняються показниками абстрактності розпаралелювання задач. Чим "глибше" рівень, в якому наступає паралельність, тим детальнішим, малоелементнішим буде розпаралелювання, що торкається елементів програми (інструкція, елементи інструкції тощо). Чим вище розміщено рівень абстракції, тим більші блоки має паралельність.

Великоблоковий ↕ Дрібноблоковий

Рівні	Об'єкт обробки	Система-приклад
Програмний	Робота, Задача	Мультизадачна ОС
Процедурний	ПРОЦЕС	MIMD-система
Рівень формул	ІНСТРУКЦІЯ	SIMD-система
Біт-рівень	В межах інструкції	Машина фон Ноймана, напр., 16-б. ALU

Рис. 2.10. Рівні паралельності

Кожний рівень має повністю різні аспекти паралельного оброблення. Методи і конструктиви даного рівня обмежуються тільки цим рівнем і не можуть бути поширені на інші рівні. Розгляньмо коротко деякі з рівнів, маючи на увазі, що найбільший інтерес викликають рівень процедур (великоблокова, асинхронна паралельність) та рівень арифметичних виразів (малоелементна, детальна або масивна синхронна паралельність).

Програмний рівень

На цьому найвищому рівні одночасно (або щонайменше розподілено за часом) виконуються комплектні програми (рис.2.11). Машина, що виконує ці програми, не повинна бути паралельною ЕОМ, досить

того, що в ній наявна багатозадачна операційна система (наприклад, реалізована як система *розподілу часу*). В цій системі кожному користувачеві відповідно до його пріоритету планувальник (*Scheduler*) виділяє відрізок часу різної тривалості. Користувач одержує ресурси центрального процесорного блоку тільки впродовж короткого часу, а потім стає в чергу на обслуговування.

У тому випадку, коли в ЕОМ недостатня кількість процесорів для всіх користувачів (або процесів), що, як правило, найбільш імовірно, в системі моделюється паралельне обслуговування користувачів за допомогою "квазіпаралельних" процесів.

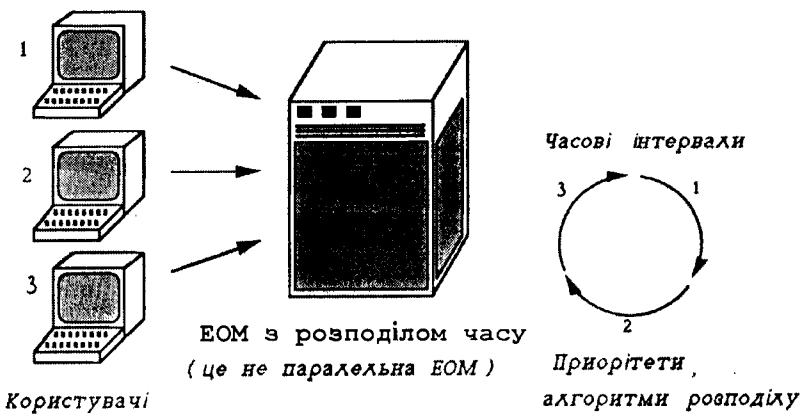


Рис. 2.11. Паралельність на програмному рівні

Рівень процедур

На цьому рівні різні розділи однієї й тієї самої програми мають виконуватися паралельно. Ці розділи називаються "процесами" і відповідають приблизно послідовним процедурам. Проблеми поділяються на суттєво незалежні частини так, щоб по можливості рідше виконувати операції обміну даними між процесами, які потребують відносно великих витрат часу. В різних галузях застосування стає ясно, що цей рівень паралельності ні в

якому разі не обмежується розпаралелюванням послідовних програм. Існує великий ряд проблем, які потребують паралельних структур цього типу навіть тоді, коли так само, як і на програмному рівні, у користувача є тільки один процесор.

Галузі застосування:

- програмування в реальному темпі часу; використовується для керування критичними за часом технічними процесами, наприклад на електростанції;

- керуючі ЕОМ, для яких характерним є одночасний виклик багатьох апаратних компонентів, наприклад у керуванні робототехнікою (рис. 2.12);

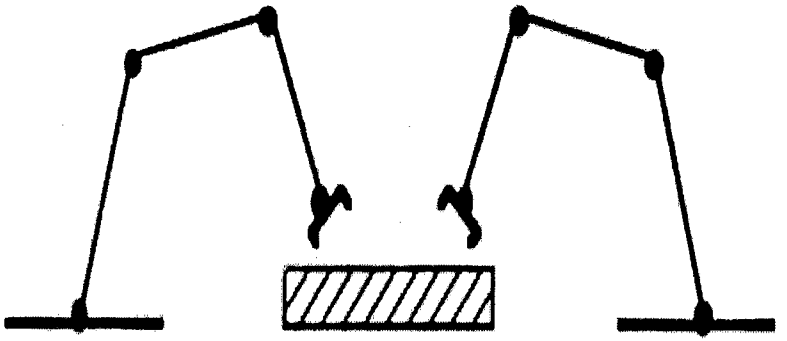


Рис. 2.12. Зкооперовані хвати роботів

- загальне паралельне оброблення інформації. В цій галузі застосовується поділ вирішуваної проблеми на паралельні задачі – частини, які вирішуються багатьма процесорами з метою підвищення обчислювальної продуктивності (приклад наведено на рис.2.13).

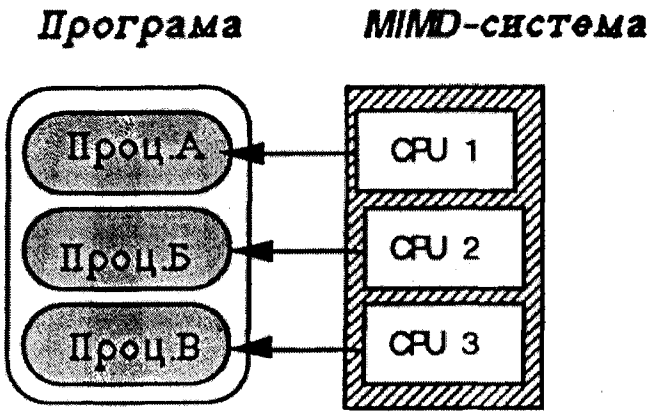


Рис. 2.13. Паралельність на рівні процедур

Рівень арифметичних виразів

Арифметичні вирази виконуються паралельно покомпонентно, причому в суттєво простіших синхронних методах. Якщо, наприклад, йдеться про арифметичний вираз складання матриць (рис.2.14), то він синхронно розпаралелюється дуже просто тому, що кожному процесорові підпорядковується один елемент матриці.

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

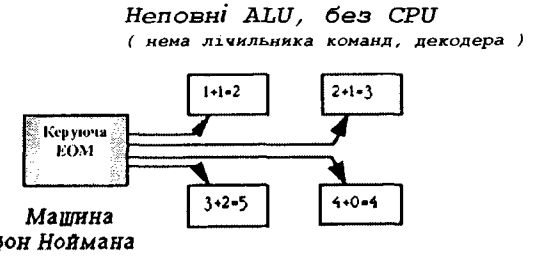


Рис.2.14.
Паралельність на рівні операцій

При застосуванні n*n процесорних елементів можна одержати суму двох матриць порядку n*n за час виконання

одної операції складання (за винятком часу, потрібного на читування та запис даних). Цьому рівню притаманні засоби векторизації та так званої паралельності даних. Останнє поняття пов'язане з детальністю розпаралелювання, а саме – з його поширенням на оброблювані дані. Майже кожному елементу даних тут підпорядковується свій процесор, завдяки чому ті дані, що в машині фон Ноймана були пасивними, перетворюються на "активні обчислювальні пристрої". До цього питання ми повернемося детальніше в наступних розділах.

Рівень двоїчних розрядів

На цьому рівні відбувається паралельне виконання однових операцій в межах одного слова (рис.2.15). Паралельність на рівні бітів можна знайти в будь-якому працюючому мікропроцесорі. Наприклад, у 8-розрядному арифметико-логічному пристрої побітова обробка виконується паралельними апаратними засобами. Цей вид паралельності реалізується відносно просто і тому в книзі не буде розглядатись.

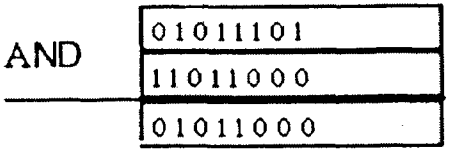


Рис. 2.15. Паралельність на рівні бітів

2.3. Паралельні операції

Зовсім інший образ паралельності виникає з аналізу математичних операцій над окремими елементами даних або над групами даних. Розрізняють скалярні дані, операції над якими виконуються послідовно, і векторні дані, над якими можна виконати потрібні математичні операції

паралельно. Операції, які нижче розглядаються, є основними функціями, що реалізуються у векторних та матричних ЕОМ.

Прості операції над векторами, наприклад складання двох векторів, можуть бути виконані безпосередньо синхронно і паралельно. У цьому випадку можна кожному елементу вектора підпорядкувати один процесор. При складніших операціях, таких як формування всіх часткових сум, побудова ефективного паралельного алгоритму є не зовсім очевидною справою. Далі розрізняються одномісцеві (монадні) та двомісцеві (діадні) операції і до кожного типу операцій наведено характерні приклади.

Одномісцеві операції

а) Скаляр → скаляр — Послідовне виконання
Приклад: 9 → 3 — "Корінь"

б) Скаляр → вектор — Размноження числової величини
Приклад: 9 → (9,9,9,9) — "Broadcast"

в) Вектор → скаляр — Редукція вектора в скаляр
Приклад: (1,2,3,4) → 10 — "Складання"

г) Вектор → вектор — (із спрощувальним припущенням, що довжина вектора не змінюється)

г-1) Локальна векторна покомпонентна операція
Приклад: (1,4,9,16) → (1,2,3,4) — "Корінь"

г-2) Глобальна векторна операція з перестановками
Приклад: (1,2,3,4) → (2,4,3,1) — "Зміна місць компонент вектора"

г-3) Глобальна векторна операція (часто складається з простих операцій)
Приклад: (1,2,3,4) → (1,3,6,10) — "Часткові суми"

Двомісцеві операції

а) (скаляр, скаляр) → скаляр — Послідовне виконання
Приклад: (1,2) → 3 — "Скалярне складання"

б) (скаляр, вектор) → вектор — Покомпонентне застосування операції над скаляром і вектором
Приклад: (3,(1,2,3,4))→(4,5,6,7) — "Складання скаляра з вектором";
операція виконується як послідовність операцій б) та є) складання векторів:

$$3 \mapsto \begin{array}{r} (3,\ 3,\ 3,\ 3) \\ +\ (1,\ 2,\ 3,\ 4) \\ \hline (4,\ 5,\ 6,\ 7) \end{array}$$

в) (вектор, вектор) → вектор — Покомпонентне застосування операції над двома векторами
Приклад: ((1,2,3,4),(0,1,3,2)) →(1,3,6,6) — "Складання векторів"

Покажемо застосування цих операцій на простому прикладі: йдеться про обчислення скалярного добутку двох векторів. Цей результат можна одержати простим послідовним застосуванням базових операцій є) (тут – покомпонентне множення двох векторів) і в) – (редукція одного вектора в один скаляр (тут – за допомогою складання).

Приклад: Скалярний добуток

$$((1,2,3),(4,2,1)) \xrightarrow{\in} (4,4,3) \xrightarrow{+} 11$$

3. МЕРЕЖІ ПЕТРІ

Мережі Петрі були розроблені ще в 1962 р. і призначалися для того, щоб представити координацію асинхронних подій [Petri 62,Baumgarten 90]. Мережі Петрі дуже часто застосовуються для описування взаємовідносин між паралельними процесами та їх синхронізації.

За визначенням мережа Петрі – це орієнтований, дводольний граф з мітками (марками). Це визначення треба розуміти так (рис.3.1):

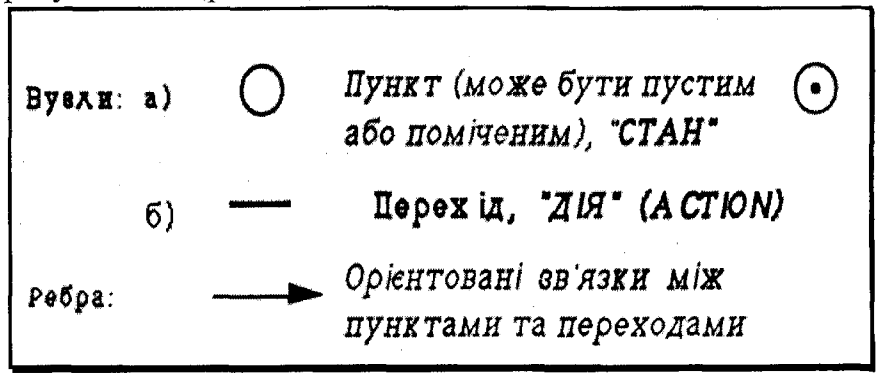


Рис. 3.1. Елементи мережі Петрі

Кожна мережа Петрі є графом, який має у своєму розпорядженні дві різні групи вершин: вузли та переходи. Між вузлами та переходами можуть міститися орієнтовані ребра (дуги), але два вузли або два переходи не можуть з'єднуватися ребрами. Між кожною парою вузол/перехід може існувати максимально одне ребро від вузла до переходу (ребро входу) і максимально одне ребро від переходу до вузла (ребро виходу). Вузли можуть бути вільними або зайнятими міткою (маркованими); переходи не можуть бути маркованими. Вузли, що є стартовими пунктами одного ребра до одного переходу t, називаються далі вхідними вузлами переходу t. Вузли, що є кінцевими пунктами ребра від переходу t, називаються відповідно виходними вузлами переходу t.

На рис.3.2 показано просту мережу Петрі, що має один перехід, три ребра і три вузли, два з яких марковані і один не маркований. Кожен вузол зв'язаний з переходом за допомогою одного ребра. Спосіб функціонування цієї мережі Петрі стає зрозумілим, якщо взяти до уваги наступні визначення.

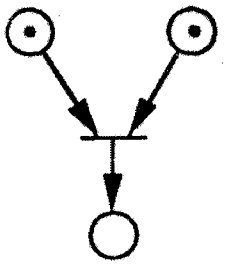


Рис. 3.2. Мережа Петрі

3.1. Прості мережі Петрі

Визначення:

Активізація: перехід t активізований, якщо всі вхід-	
(*Стан*)	ні вузли p_i цього переходу марковані.

Таким чином, активізація – це залежна від часу властивість переходу і описує деякий стан. Перехід мережі Петрі на рис.3.2 активізований, бо обидва вузли ребер, що входять у перехід, марковані.

Ввімкнення:активізований перехід t може вмикатися.	
(*Подія*)	Тоді зникають марки з усіх вхідних вузлів p_i переходу t і маркуються всі вихідні вузли p_j цього переходу.

Процес увімкнення переходу має передумовою його активізацію.

Як видно з рис.3.3, в процесі ввімкнення відбувається зміна маркування вузлів мережі Петрі: перехід активізований, бо обидва вузли вхідних ребер марковані. Після ввімкнення переходу маркування обох верхніх (вхідних) вузлів зникає, в той час як на нижньому (вихідному) вузлі з'являється нова марка. Загальна кількість маркувань у будь-якій мережі Петрі не залишається постійною. Якби на вихідному вузлі вже була марка, то вона б переписалася, тобто вузол як і раніше був би зайнятий маркою.

Рис. 3.3. Перемикання переходу

Невизначеність: якщо одночасно активізовані декілька переходів, то не зовсім зрозуміло, який з них перемикається першим.

Зроблені вище визначення не дають уявлення про порядок перемикання декількох активізованих переходів. Одночасне перемикання їх неможливе! Як показано на рис.3.4, у цьому випадку можуть відбуватися два процеси і який з них фактично має місце, в мережі Петрі визначити не можна. У зв'язку з тим, що ввімкнення одного або іншого переходу не може бути здійснене за допомогою зовнішніх параметрів, мережа Петрі має в цьому випадку невизначеність.

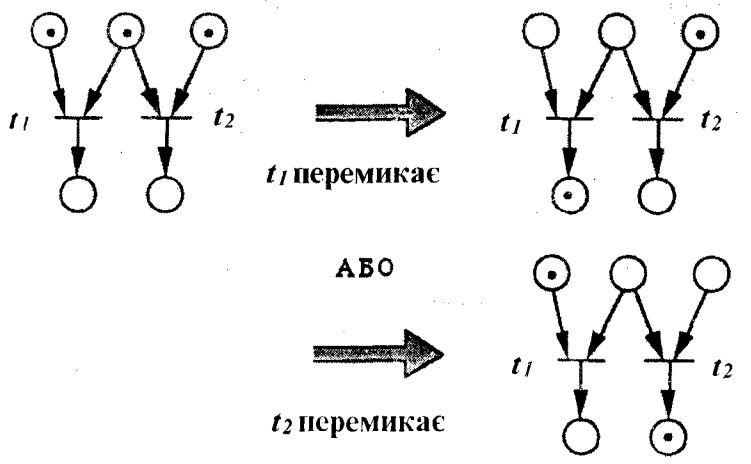
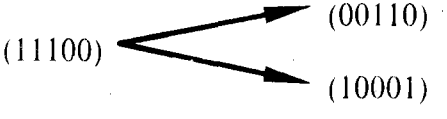


Рис. 3.4. Недетермінованість перемикань мережі Петрі

Стан: стан маркування (або, коротко, стан) мережі Петрі до деякого моменту часу т визначено як сукупність маркувань кожного окремого вузла мережі.

У простих мережах Петрі стан маркування можна відобразити за допомогою деякої послідовності бінарних цифр (двоїчний рядок). Можливе перемикання переходу задається за допомогою переходу в послідовність стану (якщо за аналогією з показаним на рис.3.4 можуть перемикатися декілька переходів, то існують різноманітні можливі послідовності станів):



У тому випадку, коли перемикання кожного переходу визначається цим правилом, всі послідовності станів одного заданого початкового стану можуть бути

"обчислені" комп'ютером і можуть бути розпізнані вірогідні блокування (див. нижче).

Генерування марок: перехід, що не має жодного вхідного ребра, завжди активізований і може постійно видавати нові марки на вихідні вузли, що з'єднані з ним.

Знищення марок: перехід, що не має жодного вихідного ребра і має тільки одне вхідне ребро, завжди активізований тоді, коли це місце марковано і він може постійно знищувати марку.

"Мертвий" стан: Мережа Петрі перебуває в "мертвому" стані (блокована), якщо жоден з її переходів не активізований.

Рис. 3.5. Генерування та гасіння марок

Блокована мережа Петрі є статичною, тобто в ній немає нових послідовностей станів. Наприклад, обидві мережі Петрі, що показані на рис.3.4, блоковані.

"Живий" стан: мережа Петрі перебуває в "живому" стані (не блокована в жодному моменті часу), якщо хоча б один з її переходів активізований і це справедливо також для кожного наступного стану.

Треба мати на увазі, що "живий" стан не є протилежністю "мертвого" стану. Мережа Петрі в "живому" стані не блокована і не перебуває в жодному із можливих послідовностей станів, в той час як не блокована мережа Петрі не обов'язково має бути в "живому" стані. Наприклад, блокування може відбутися тільки після багатьох послідовних кроків. На рис.3.6 показано два приклади мереж.

Ліва мережа Петрі на рис.3.6 є "живою" тому, що її маркування змінюється від одного вузла до іншого, але не циклає; перехід завжди тут активізований. У правої мережі Петрі (рис.3.6) інша ситуація. По-перше, тут можуть перемикатись або перехід ***u*** (і тоді мережа негайно переходить у "мертвий" стан), або перехід ***s***. Нарешті можуть перемикатись один раз переходи ***t*** і U, що веде також до "мертвого" стану мережі Петрі.

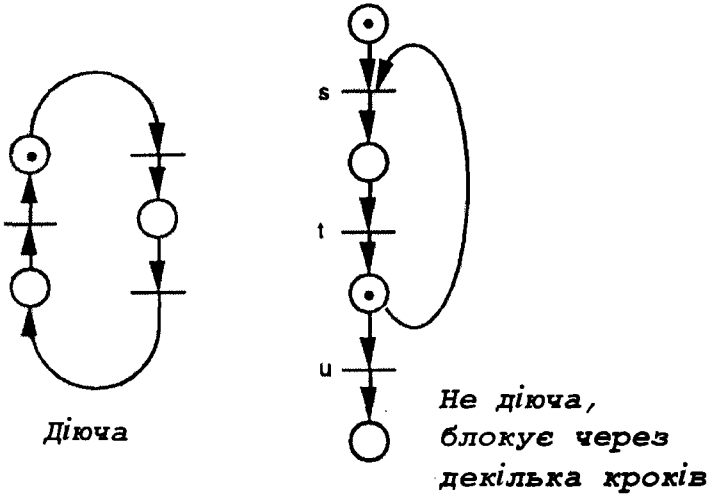


Рис. 3.6. Діюча та блокуюча мережі Петрі

Як уже згадувалося, за допомогою мереж Петрі може описуватися синхронізація асинхронних паралельно виконуваних процесів. Це може бути потрібним, щоб уникнути взаємоблокування процесів або виникнення

суперечних даних у тих випадках, коли два процеси мають звернутися до однієї й тієї області даних (рис.3.7).

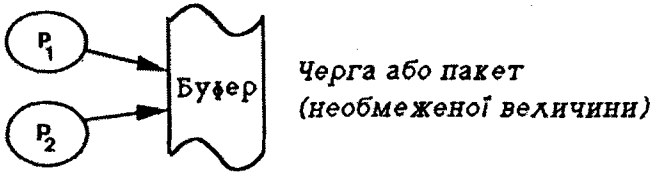


Рис. 3.7. Доступ двох процесів P_1, P_2 до загальних даних

Процес P_1 (*виробник*) генерує тут незалежно від процесу P_2 дані, записує їх у буферну область пам'яті і хоче без затримки працювати далі. Процес P_2 (*споживач*) читає дані із цього буфера і використовує їх паралельно з процесом P_1. Щоб уникнути суперечливих даних, треба виконати таку умову:

одночасний доступ процесів в одну й ту саму область пам'яті має бути виключений.

Це викликає потребу в синхронізації процесів, тобто один з них має деякий час почекати.

На рис.3.8 показано простий приклад для випадку коли процеси P_1 та P_2 мають бути синхронізовані, тому що

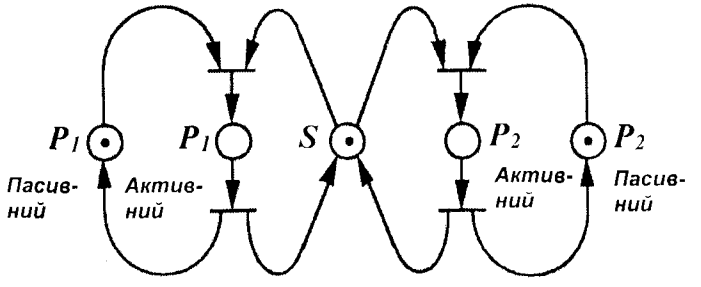


Рис. 3.8. Мережа Петрі для синхронізації процесів

вони, наприклад, хочуть користуватися одними й тими самими даними. Кожний процес має тут два стани – активний і пасивний, які символізуються двома вузлами. Процес перебуває в тому стані, який позначається відповідною маркою (це означає, що на рис.3.8 обидва процеси пасивні). Кожен з двох процесів може змінювати свій стан з пасивного на активний і навпаки, перемикаючи відповідний перехід. Обидва цикли стану для P_1 і P_2 аналогічні простому контуру, що зображений зліва на рис.3.6, але ці цикли пов'язані між собою за допомогою так званого вузла-семафора S. Процес, що хотів би змінити свій стан з пасивного на активний (щоб звернутися до загальних даних), потребує для цього переходу маркування в S. Це означає, що він тільки тоді може змінити свій стан на активний, коли марка семафора не використовується іншим процесом (який в цей час перебуває в активному стані). При переході в активний стан семафор S звільняється від маркування; у тому випадку, коли інший процес запросить перехід з пасивного стану в активний (доступ до загальних даних), він має чекати, поки перший процес не перейде знову зі стану активного в пасивний і знову з'явиться семафор-марка S.

Синхронізація за допомогою цієї мережі Петрі є високонадійною, тому що в кожний момент часу тільки один процес перебуває в активному стані. Таким чином, в разі одночасного запросу процеси з паралельних перетворюються в послідовні і блокування їх виникнути не може.

3.2. Розширені мережі Петрі

Прості мережі Петрі, що були тут представлені, повністю забезпечують цілий ряд різноманітних застосувань. Проте за допомогою трьох простих розширень вони стають суттєво продуктивнішими. Як показано Хопкрофтом і Ульманом, "розширені мережі

Петрі" (навіть без додатково уведених нижче ваг дуг) за своєю ефективністю дорівнюють машині Тьюрінга [Hopcroft, Ullmann 69], тобто вони можуть застосовуватись як загальна модель обчислюваності.

Розширення:

1. Багаторазове маркування

 відповідає (4)

Кожен вузол може мати будь-яку кількість маркувань (при зображенні мережі Петрі допускається маркування коротко позначати числом). Правила активізації та перемикань змінюються відповідно:

- перехід активізований тільки тоді, коли число, що відповідає кількості маркувань кожного його вхідного вузла, є більшим за одиницю або дорівнює їй;
- якщо активізований перехід перемикається, то числа-маркування усіх вхідних вузлів цього переходу зменшуються на одиницю, а усіх вихідних вузлів – збільшуються на одиницю.

За цим способом кількість маркувань одного вузла може бути як завгодно великою, але відповідне число не може бути меншим від нуля.

2. Дуги-заперечення

Дуги-заперечення в мережах Петрі зображаються кружечком на кінці замість стрілки (рис.3.9) і не мають дугової ваги; дуги, що були визначені раніше, розглядаються як позитивні дуги.

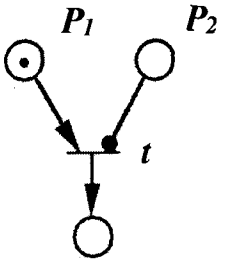


Рис. 3.9. Дуга-заперечення

Вони завжди можуть бути направлені тільки від деякого вузла до деякого переходу, зворотного напрямку не дозволяється. Введення дуг-заперечень зумовлює оновлені пристосування правил активізації та перемикань:

- перехід активізований тільки тоді, коли кількість маркувань кожного вхідного вузла з позитивною дугою більша або дорівнює одиниці і коли кількість маркувань кожного вхідного вузла з дугою-запереченням дорівнює нулю (на рис.3.9 перехід t активізований, якщо P_1 маркований і P_2 не маркований);
- якщо активізований перехід перемикається, то кількість маркувань усіх вхідних вузлів з позитивними дугами цього переходу зменшується на одиницю, в той час як кількість маркувань вхідних вузлів з дугами-запереченнями залишається незмінною. Числа, що відповідають кількості маркувань усіх вихідних вузлів, як і раніше, збільшуються на одиницю.

3. Вага дуг

Кожна дуга, що не є дугою-запереченням, може мати постійну цілочислову вагу, що більша або дорівнює одиниці (число, що використовується за умовчанням,

рис. 3.10). Для активізації і перемикання переходу діє правило:

- перехід активізований тільки тоді, коли кількість маркувань кожного його вхідного вузла більше або дорівнює відповідній вазі дуги, а для дуги-заперечення дорівнює нулю;

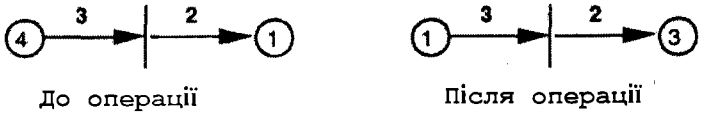


Рис. 3.10. Вага дуг

- при перемиканні переходу кількість маркувань кожного вхідного вузла зменшується на вагу відповідної вхідної дуги (вона залишається незмінною для дуг-заперечень); кількість маркувань кожного вихідного вузла збільшується на вагу відповідної вихідної дуги.

3.3. Приклади мереж Петрі

Наведемо ряд прикладів розширених мереж Петрі, які частково базуються на прикладах з роботи [Krishnamurthy 89]. В принципі можна кожну програму, написану на будь-якій мові програмування, представити у формі розширеної мережі Петрі. При цьому треба мати на увазі, що "пам'ять маркувань" деякого вузла може містити тільки додатне число або нуль. Від'ємні числа можуть, наприклад, представлятися додадковим вузлом, що визначає знаки чисел. Кожна наведена тут мережа має спеціальний "стартовий" вузол, з якого починається обчислення, і вузол "готовності", який маркується тоді, коли обчислення повністю завершено і з'явився результат. Спеціальні старт- і готовність-вузли особливо важливі тоді, коли треба скласти розширену мережу Петрі із наявних елементів.

Суматор

Суматор (рис. 3.11) має полічити маркування від пункту **Y** до маркувань в **Z**, тобто створити суму ***Z+Y***. Тут маємо спеціальний стартовий вузол, який на початку операції зайнятий маркуванням "1". Після перемикання переходу ***s*** ця марка зникне з місця старту, а марка у вузлі готовності з'явиться тільки тоді, коли закінчиться процес складання і сума буде у вузлі **Z**.

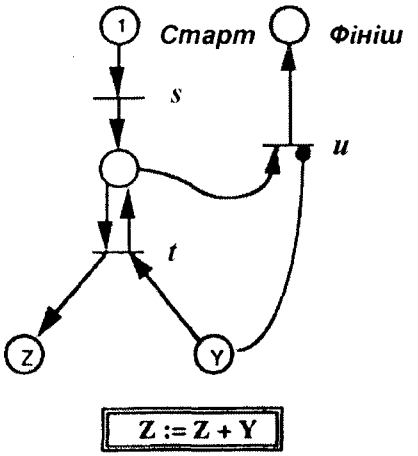


Рис. 3.11. Мережа Петрі як суматор

Після перемикання стартового переходу ***s*** маркується середній вузол мережі Петрі. Крок за кроком, перемикаючи перехід **t**, одне маркування береться з вузла **Y** (вхід) і одночасно вводиться у вузол **Z** (вихід). Середній вузол залишається при цьому маркованим, бо він містить у собі як вихідну, так і вхідну дуги (1-1+1=1). Одначе цей вузол не є зайвим, бо він дбає про те, щоб операція складання відбувалась як результуюча після розблокування стартового вузла. В кінці операції, коли **Y=0**, сума з'являється у вузлі **Z** і перехід ***t*** більше не активізується. Для завершення операції призначений перехід ***u***, що тепер активізований; якщо він перемикається, то марка із

середнього вузла виводиться і генерується марка у вузлі готовності, що сигналізує про закінчення обчислення.

У цьому та наступних прикладах треба мати на увазі, що початкове числове значення **Y** знищується (у нашому випадку стає нулем) і не може використовуватися для можливих подальших операцій. Якщо змінна **Y** потрібна для інших операцій, що відбуваються після одержання суми, то мережу Петрі треба будувати так, щоб в ній відновлювалося значення величини **Y**.

Пристрій для віднімання

Простий пристрій для віднімання (рис.3.12 зліва) можна побудувати із суматора (рис.3.11), в якому змінено напрямок дуги від **Z** (штрихова стрілка). На кожному кроці тепер маркування не йде на місце **Z**, а навпаки, виводиться з **Z**. Ця методика функціонує, правда, за тих умов, що **Z≥Y**; в інших випадках віднімання зупиняється без маркування кінця операції у вузлі готовності! Це, звичайно, не має сенсу, тому на рис.3.12 праворуч показано розширений пристрій, який обчислює симетричну різницю.

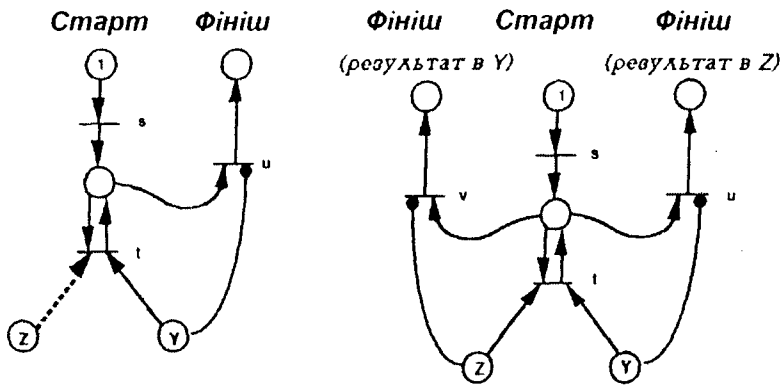


Рис. 3.12. Мережа Петрі для операції віднімання

При симетричному відніманні різниця **Z–Y** заноситься в **Z**, якщо **Z ≥Y**, а різниця **Y–Z** заноситься в **Y**, якщо **Y≥Z**. У зв'язку з цим мережа Петрі доповнюється симетрично переходом *v*.

Пристрій для множення

Пристрій для множення (рис. 3.13) має складнішу будову, але в переходах *s*, *t* і *v* можна впізнати основні елементи суматора. Це змінює одночасно принцип роботи перемножувача: під час кожного проходу **X** зменшується

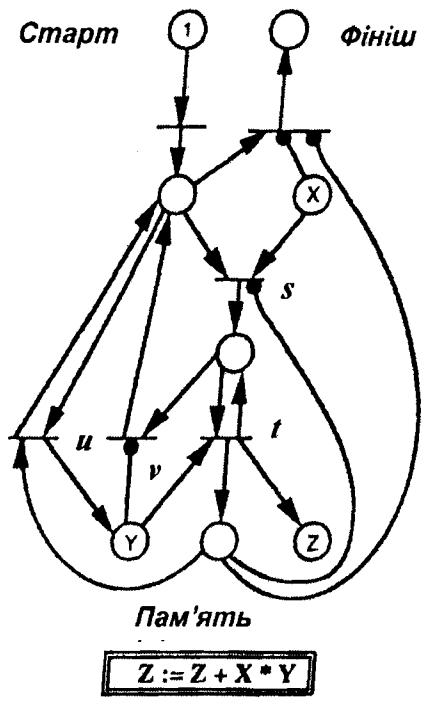


Рис. 3.13. Мережа Петрі для операції множення

на одиницю і **Y** покроково складається з **Z**. У вузлі "пам'ять" копіюється початкова величина **Y**, і в кінці операції через перехід *u* на **Y** робиться нова копія, щоб **Y** можна було використати в наступних розрахунках. Новий

обчислювальний цикл починається, якщо "пам'ять" дорівнює нулю; потім перехід знову активізується через дугу заперечення. Цикли, в яких величина **Y** складається з **Z**, виконуються доти, поки **X** стане дорівнювати нулю, тобто добуток **X*Y** буде складено з **Z**.

Багато з виконуваних обчислювальних операцій можуть бути легко розміщені у вигляді ланцюга активізацій, в якому їхні стартові і фінішні вузли зв'язуються між собою послідовно відповідно до порядку обчислень або також паралельно. При багаторазовому доступі зчитування з одного і того самого елемента пам'яті обчислення мають вестися послідовно. Крім цього, усі операції мають відновлювати вихідну величину, що була в елементі пам'яті. Ця умова виконана в наведених прикладах тільки для змінної величини **Y** в пристрої для множення і в разі потреби має доповнюватись.

На рис.3.14 показано можливість розмноження значень змінної величини. На рис.3.15, 3.16 наведено приклади складних мереж Петрі, які скомпоновані з більш простих модулів за методом чорного ящика.

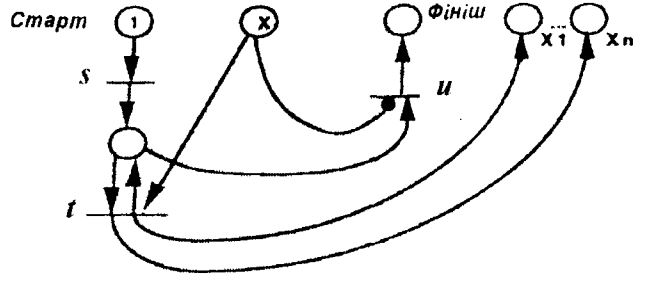


Рис. 3.14. Розмноження значень змінної **X**

Послідовні обчислення

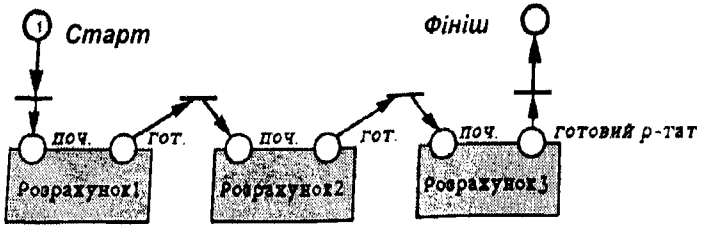


Рис. 3.15. Послідовне використання мережі Петрі

Паралельні обчислення

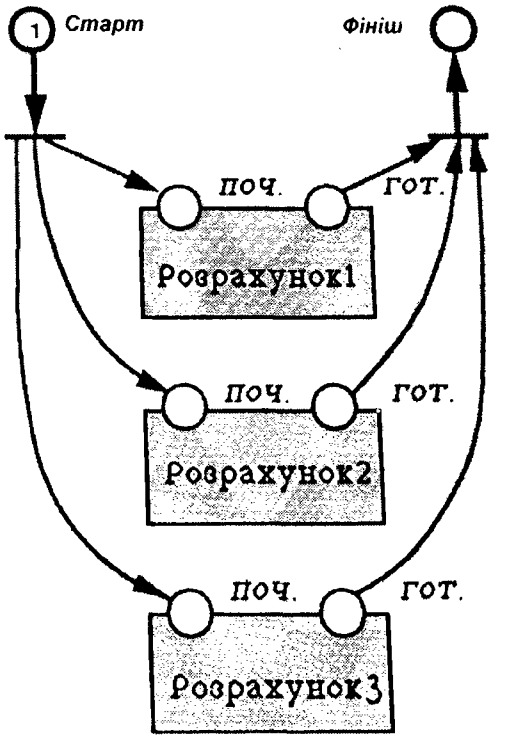


Рис. 3.16. Паралельне використання мережі Петрі

4. КОНЦЕПЦІЇ ПАРАЛЕЛЬНОЇ ОБРОБКИ

Тут наведено основні концепції паралельної обробки інформації. Всі моделі в цьому розділі будуть розглянуті тільки коротко, а найважливіші концепції, а саме асинхронна та синхронна паралельність, будуть деталізовані в наступних розділах книги.

4.1. Співпрограми

Використовуючи співпрограми (coroutine, рис. 4.1), говорять про концепцію обмеженої паралельної обробки інформації, яка поряд з іншим має місце у мові Модула-2 Вірта [Wirth 83]. В основу цієї обробки покладено однопроцесорну модель. Тут маємо тільки один потік інструкцій з послідовно виконуваним керуючим потоком, що передбачає організоване заняття та звільнення операційного ресурсу "процесор" співпрограмою.

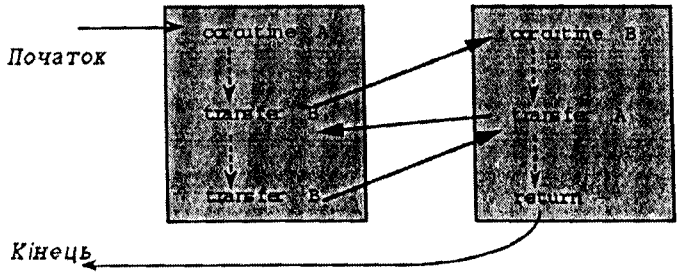


Рис. 4.1. Обмежена паралельна обробка (COROUTINE)

Цей "квазі-паралельний" процес відбувається між двома або багатьма співпрограмами і може розглядатись як один з видів процедур, локальні дані яких залишаються незмінними від одного виклика до наступного. Обробка починається за викликом однієї співпрограми. Кожна співпрограма може мати в багатьох місцях інструкцію для перемикання керуючого потоку на іншу співпрограму. Це не є процедурним викликом, тобто викликана співпрограма має віддавати керування на співпрограму, що її викликала, а може також переключатися на інші співпрограми. Якщо співпрограма, що була активною до цього, одержує дозвіл на ресурс, то обробка продовжується з тієї інструкції, перед якою було виконано перехід до іншої співпрограми. Якщо активна співпрограма термінує ("зависає"), то закінчується процес виконання всіх інших співпрограм. Переходи між співпрограмами мають описуватися програмістом-користувачем у явній формі. Він повинен також дбати про те, щоб кожна співпрограма була доступною і щоб керуючий потік правильно визначав пункти співпрограм, з яких продовжується обчислення після перерви.

У зв'язку з тим, що ця концепція будується на одному єдиному процесорі (і не може бути розширена на декілька процесорів), вона не потребує відповідних управлінських витрат на мультизадачність. Однак тут не відбувається справжньої паралельної обробки!

Для переходів між співпрограмами в мові Модула-2 передбачено процедуру "Transfer":

```
PROCEDURE TRANSFER (VAR Source, Destination:
                                ADRESS);
```

Під час кожного подальшого переходу потрібно задавати в явному вигляді ім'я актуальної та наступної співпрограм.

4.2. Fork (розгалуження, виникнення паралельних процесів) і Join (об'єднання)

Конструкції fork і join (які в операційній системі Unix відомі як fork і wait) були введені авторами робіт [Conway 63, Dennis, Van Horn 66] і належать взагалі до найперших паралельних мовних конструкцій (рис.4.2).

В операційній системі Unix [Kernighen, Pike 84] є можливість запускати паралельні процеси за допомогою операції fork, а закінчення їх чекати операцією wait. На відміну від співпрограм тут мова йде про приклад істинно паралельних процесів, що можуть за браком паралельних

апаратних ресурсів оброблятись на одному процесорі в мультизадачному режимі з дискретним розподілом часу. У цьому способі паралельного програмування об'єднано дві принципово різні концепції: по-перше, декларування паралельних процесів і по-друге – синхронізація процесів. Оскільки йдеться про концептуально різні задачі, було б доцільніше, з огляду на основи програмної інженерії, щоб вони були чітко розділені в мові програмування введенням різних мовних конструктивів.

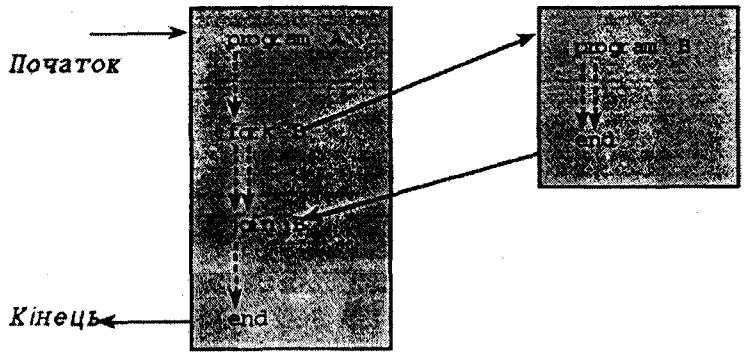


Рис. 4.2. Паралельні конструктиви Fork та Join

У системі Unix виклик операції fork функціонує не так наочно, як показано на рис. 4.2. Замість цього виготовляється ідентична копія процесу, що викликається, і вона виконується паралельно йому. Змінні величини та стартова величина програмного лічильника нового процесу такі самі, як у вихідному процесі. Єдина можливість відрізнити ці процеси (які з них – процеси-батьки, а які – діти), полягає в тому, що fork-операція видає ідентифікаційний номер, який треба оцінити. Його величина для процесів-дітей дорівнює нулю, а у процесів-батьків він збігається з ідентифікаційним номером процесів в ОС Unix (число, що завжди відрізняється від нуля). Такий спосіб організації процесів не відповідає проблематиці програмування, що ставить за мету побудову зрозумілих, зручних для читання та перегляду програм.

Щоб все ж таки запустити іншу програму і чекати на її закінчення, можна обидві системні Unix-процедури виклику сконструювати мовою програмування С в такий спосіб:

```
int status;
if (fork( ) == 0) execlp ("program_B",...);   /* процес-дитя  */
...
wait ( &status);                              /* процес-батько*/
```

Виклик операції fork видає для батьківського процесу номер процесу-дитини як зворотний зв'язок (тут fork() не є нулем), у той час як для процесу-дитини видається величина нуль; тим самим гарантується, що процес-дитина негайно виконує нову програму операцією execlp і це не є текстом батьківського коду. Одночасно батьківський процес виконує наступні інструкції паралельно процесу-дитині. Батьківський процес може чекати закінчення процесу-дитини операцією wait; параметр зворотного зв'язку містить статус закінчення процесу-дитини.

4.3. ParBegin та ParEnd

За аналогією з відомими ключовими словами begin та end, що відокремлюють послідовні блоки інструкцій, словами parbegin та parend визначаються паралельні блоки, в яких інструкції мають виконуватись одночасно (паралельно) (рис. 4.3).

Часто застосовуються також слова cobegin, coend. Ця концепція імплементована, наприклад, в мові програмування роботів AL [Mujtaba, Goldman 81], де можуть однією паралельною програмою керуватись одночасно декілька роботів, а їхній рух координується за допомогою семафорів (див. гл. 7). Подібну конструкцію має Алгол-68 у вигляді оператора par. Він дозволяє

паралельний старт процесів, що також синхронізуються семафорами. Подібно до семафорних сигналів на залізниці,

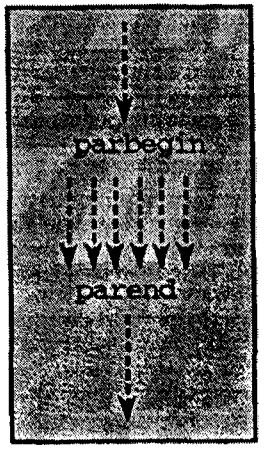


Рис. 4.3. Паралельний блок інструкцій

системні семафори дають змогу реалізувати контрольовані зайнятість та звільнення обчислювальних засобів, причому запити, що не можуть бути виконані, спричиняють блокування з подальшим звільненням запитувача. Однак синхронізація за допомогою семафорів є дещо примітивною і не наочною. Тому у концепції parbegin/parend немає засобів високого рівня синхронізації та передачі інформації, які підтримують паралельне програмування. У зв'язку з названими обмеженнями концепція паралельних блоків інструкцій (команд) в сучасних мовах паралельного програмування не використовується.

4.4. Процеси

Процеси – це паралельна концепція для MIMD систем. Вони декларуються подібно до процедур і запускаються в явній формі за допомогою деякої інструкції

Якщо процес має існувати в різноманітних варіантах, то його треба запускати відповідно кілька разів, по можливості з різними параметрами (так, як з параметрами процедур). Синхронізація паралельно виконуваних процесів регулюється за концепцією семафора або монітора (гл. 7) із застосуванням умовних змінних величин [Hoare 74]. Монітори в порівнянні з семафорами забезпечують надійнішу синхронізацію на вищому рівні абстракцій.

Монітор (рис.4.4) – єдине ціле із загальних даних, операцій доступу і списку черг чекання. Тільки один процес з багатьох може в деякий момент часу ввійти в монітор, чим з самого початку виключається багато проблем синхронізації. Блокування та розблокування процесів всередині монітора проводиться за умовними списками черг. Однак явна синхронізація паралельних процесів дає

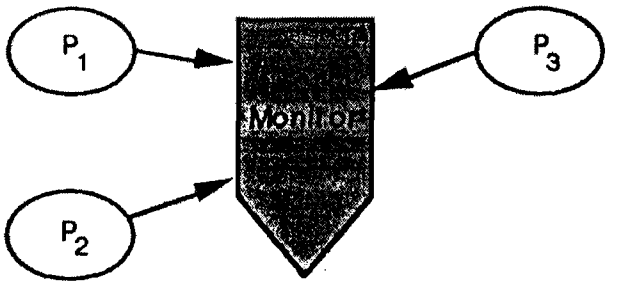


Рис. 4.4. Синхронізація процесів монітором

не тільки додатковий обсяг роботи щодо керування, а й постійний острах до помилок. Цей спосіб організації паралельних обчислень потребує воістину винятковой уваги під час програмування. Якщо різні процеси мають звертатися до одних і тих самих даних, то ці так звані "критичні розділи" треба захистити за допомогою синхронізаційних конструкцій. Це означає, що в кожний момент часу до цього розділу має право звернутися лише

один процес і працювати з загальними даними. Найпоширенішими помилками є неконтрольований доступ до критичного розділу або вихід з нього (програміст забув про операції синхронізації) та помилкове керування процесами, що чекають, тобто блокованими. Ці помилки призводять передусім до виникнення неправильних даних, а крім цього, можуть спричинити блокування окремих процесів або навіть всієї системи процесів.

Обмін інформацією та синхронізація виконуються в системах з загальною пам'яттю ("тісно зв'язані" системи) через монітори з умовами. В системах без загальної пам'яті ("слабко зв'язані" системи) потрібна концепція передачі та прийому повідомлень, про яку йдеться в наступному параграфі.

4.5. Дистанційний виклик

Щоб поширити концепцію процесів на паралельні ЕОМ без загальної пам'яті, потрібно реалізувати спілкування процесів на різних процесорах через обмін повідомленнями. Концепція повідомлень може бути імплементована і на MIMD-ЕОМ із загальною пам'яттю (комфортабельніше, але з вищими витратами на керування).

Програмна система поділяється на декілька паралельних процесів, причому кожний процес бере на себе роль або сервера, або клієнта. Кожний сервер має в основному один нескінченний цикл, в якому він чекає на черговий запит, виконує бажані послуги (обчислення) і видає результат в обумовленому вигляді. Кожний сервер при цьому може стати також клієнтом, і тоді він бере до уваги дії іншого сервера. Кожний клієнт передає блоки задач одному або декільком відповідно конфігурованим сервер-процесам (рис.4.5).

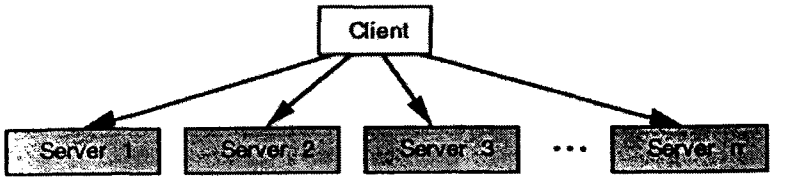


Рис. 4.5. Клієнт та декілька серверів

Цей спосіб паралельного розподілення робіт імплементується механізмом дистанційного виклику ("remote procedure call", RPC, рис.4.6).

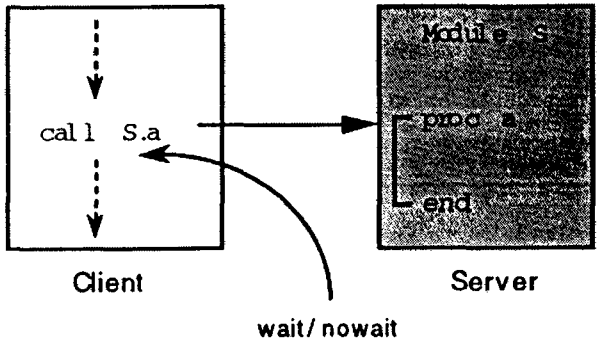


Рис. 4.6. RPC–механізм розпаралелювання (Remote Procedure Call)

Спроможність до обробки інформації, природно, суттєво зростає, якщо клієнт не чекає на результати сервера під час кожного звертання до нього, а паралельно йому може вести подальші розрахунки на своєму процесорі. Однак із цієї вимоги на краще завантаження апаратури та її більшу ефективність виникають і нові проблеми: параметри зворотного зв'язку недоступні негайно після виконання сервер-операції, бо вони більше відповідають операції типу "віддача замовлення". Результати, на які чекає клієнт, в

цьому випадку після обчислень у сервері мають бути переслані у зворотному напрямку від сервера до клієнта у вигляді явного обміну даними. Труднощі в RPC-методі викликає також виготовлення безпомилкових протоколів вводу в дію і повторного пропуску задач після виходу сервера з ладу.

4.6. Неявна паралельність

Усі представлені концепції паралельності застосовують спеціальні, явні мовні конструктиви для керування паралельною обробкою інформації. Суттєво елегантнішими є мови програмування, які виходять з того, щоб робити можливою паралельну обробку, не застосовуючи мовних конструктивів для обслуговування паралельності. Подібні мови програмування називають мовами з неявною паралельністю. Проте під час роботи на цих мовах програміст має мало засобів впливу на застосування паралельних процесорів для вирішення його проблеми. Тут має бути впевненість, що у програміста наявна достатня кількість процедуральної інформації ("знань"), щоб зробити можливим ефективне розпаралелювання. Подібне завдання може вирішувати, наприклад, "інтелектуальний компілятор" без спілкування з програмістом-користувачем. Ця проблема стає зрозумілою передусім в декларативних мовах програмування, таких як Lisp (функціонально) або Prolog (логічно). Через декларативне представлення знань, принаймні постановки задачі (наприклад, складна математична формула), розв'язок її визначається в основному однозначно. Однак суттєві труднощі виникають при спробах перевести ці знання в командний паралельний програмний процес, тобто скласти програму для обчислень за формулою і проблему розчленити на задачі-частки, які б могли розв'язуватися паралельно.

Неявна паралельність може бути безпосередньо виділена із векторних конструкцій мов програмування, наприклад FP (Functional Programming [Backus, 78], (див. п. 1.7.2) або APL (A Programming Language [Iverson 62]). В APL немає жодної контрольної структури високого рівня, яка безумовно потрібна в будь-якій (послідовній чи паралельній) мові програмування.

Як показано на рис.4.7, математичний запис складання матриць містить у собі неявну паралельність, яка в цьому випадку досить просто може бути перетворена на деяку паралельну обчислювальну структуру методом автоматичного розпаралелювання (див. паралельність на рівні виразів в п.2.2).

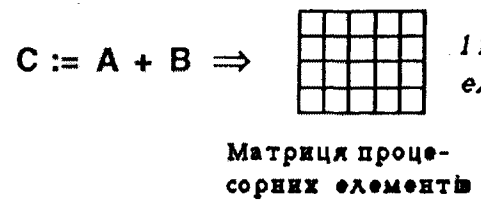


Рис. 4.7. Неявна паралельність при виконанні матричних операцій

4.7. Порівняння явної та неявної паралельностей

На закінчення цієї глави проведімо зіставлення переваг та недоліків явної та неявної паралельностей (рис.4.8).

Неявна паралельність розвантажує програміста, бо він не повинен тут займатися задачами керування та контролю. Програмування відбувається на високому рівні абстракцій, тому неявна паралельність часто має місце в непроцедурних мовах програмування високого рівня. На протилежність цьому явна паралельність дає програмісту суттєво більшу свободу дій, щоб досягти вищої

обчислювальної продуктивності від правильного завантаження процесорів. Ця перевага пов'язана із складнішим програмуванням, якому притаманна більша вірогідність помилок.

Зіставлення явної та неявної паралельностей

Явна паралельність	Неявна паралельність
Програміст має повний контроль над паралельними операціями	Програміста звільнено від задач керування паралельними операціями
Ефективне виконання програм (залежить від програміста і часто потребує спеціальних знань)	Здебільшого менш ефективне виконання програм
Складне, пов'язане з помилками програмування	Просте програмування, менша вірогідність помилок
Переважно процедурні мови програмування (виняток – * LISP)	Переважно не процедурні мови програмування або розпаралелені (векторизовані) компілятори для процедурних мов (наприклад, Фортран)

Рис. 4.8. Порівняння явної та неявної паралельностей

8. СТРУКТУРИ ЗВ'ЯЗКУ МІЖ ПРОЦЕСОРАМИ

Кожна паралельна ЕОМ має у своєму складі ряд процесорів та один або декілька модулів пам'яті. Ці функціональні компоненти мають бути зв'язані через відповідні комутаційні структури, щоб утворити єдину цільну систему. Всі високорівневі концепції комунікацій, такі як "schared memory" (загальна пам'ять, реальна або віртуальна) або обмін повідомленнями, реалізуються в паралельних системах деякими наявними структурами. Враховуючи задачі обчислювальної системи, її структура зв'язку має відповідати різним критеріям. По-перше, забезпечувати по можливості високу коннективність, тобто гарантувати зв'язок між двома будь-якими процесорами (або модулями пам'яті) без потреби переходу через проміжні пункти комутацій. Крім цього, має забезпечуватись найбільша кількість одночасних зв'язків, щоб комутаційна мережа не обмежувала паралельну обробку інформації. Проте об'єктивно існують різноманітні обмеження на реалізацію цих критеріїв: кількість ліній зв'язку на один процесор не може збільшуватися безмежно, має також фізичні обмеження і ширина частотної смуги пропускання (швидкість передачі) комутаційної мережі.

Для паралельної системи з n процесорними елементами (ПЕ) можна визначити такі види витрат:

а) *кількість зв'язків на один ПЕ (витрати на виготовлення системи);*

б) *дистанція між ПЕ (витрати під час експлуатації).*

Як видно з наведених вище міркувань, кількість зв'язків на один процесор і дистанція, тобто найкоротша лінія зв'язку між двома заданими ПЕ, мають бути якомога меншими. Структура зв'язку має бути розширюваною за масштабом, тобто малі мережі зв'язку мають збільшуватися шляхи доповненням.

Структури зв'язку поділяються на три великих класи, що розглядаються в наступних параграфах:

- Шинні сітки.
- Сітки з комутаторами.
- Структури, що забезпечують зв'зок типу "пункт-пункт".

5.1. Шинні сітки

Шина як структура зв'язку відома ще з часів побудови машини фон Ноймана. В той час як шина в послідовних машинах з'єднує їхні функціональні блоки, в паралельних ЕОМ через неї можна з'єднати між собою процесори або модулі пам'яті (рис.5.1).

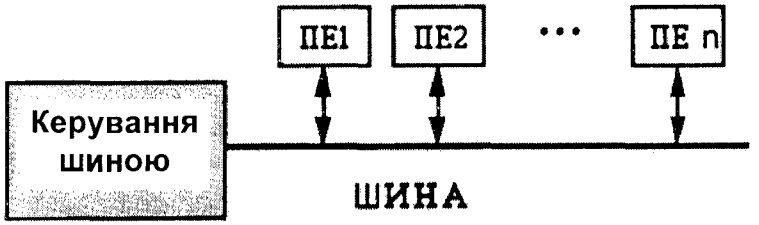


Рис. 5.1. Шинна система

Тут кількість зв'язків на один ПЕ завжди дорівнює 1, тобто оптимальна. Відстань між двома ПЕ також зостається незмінною і дорівнює 2 (не 1, тому що кожний зв'язок має йти від ПЕ до шини і назад: $ПЕ_i \rightarrow$шина$\rightarrow ПЕ_i$). Це також практично оптимальне число. Недоліки шини зумовлені тим, що в кожний момент може відбуватись одне з'єднання. Паралельне читання за однією й тією адресою модуля пам'яті можливе, однак запис – ні. Ні в якому разі процесори не можуть незалежно обмінюватися даними, паралельна обробка при обміні даними між процесорами неможлива.

Система керування шиною має гарантувати послідовне упорядкування одночасних запитів на послуги шини.

Якщо система розширюється на багато процесорів, то ширина частотної смуги пропускання шини лишається незмінною. Цей важливий недолік визначає масштабні спроможності шинних структур.

У зв'язку із сказаним шинні системи як засоби зв'язку в паралельних ЕОМ, в яких кількість процесорів перевищує десять, застосовуватись не можуть.

5.2. Сітки з комутаторами

Сітки з комутаторами – це динамічні структури зв'язку. Тут за допомогою ліній керування можуть бути реалізовані різноманітні варіанти спілкування процесорів перед початком виконання паралельної програми. Розглянемо динамічні сітки типу розподілювача перехресних шин, дельта-сітки, мережі зв'язку Клоса та сітки типу Fat-Tree.

Розподілювач перехресних шин (рис.5.2)

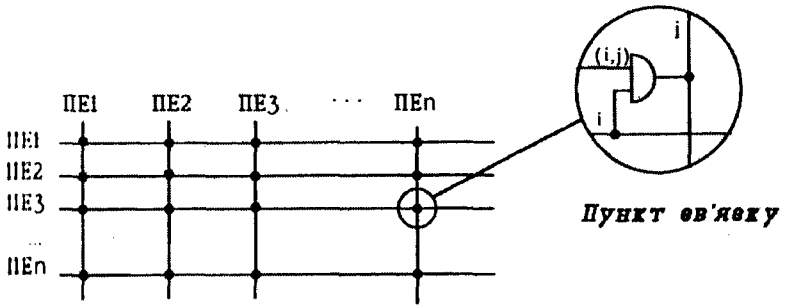


Рис. 5.2. Шинний розподілювач

Кожний ПЕ має n-1 пункт комутації, бо діагональні пункти не потребують контакту. Загалом ця сітка має n*(n-1) пунктів комутації і дає можливість встановити будь-яку кількість з'єднань між всіма ПЕ (кожна "перестановка ПЕ-

з'єднань", п.2.3.), тобто без всяких колізій може відбуватис повністю паралельний обмін даними. Суттєвим недолік цього комутатора є витрати на його побудову, щ становлять n^2-n пунктів з'єднань для n ПЕ. Практично так кількість пунктів може бути реалізована лише дл невеликої кількості процесорів. Вже для 100 ПЕ потрібн 9900 перемикань, що забезпечить роботу всіх пункті з'єднань.

Дельта-сітки

Щоб зменшити витрати порядку n^2, що виникают під час побудови розподілювачів перехресних шин, бул розроблені дельта-сітки. В найпростішому випадку (рис.5.3

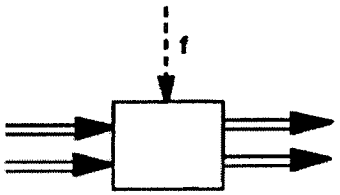


Рис. 5.3. Дельта-мережі комутації

дві лінії даних можуть перемикатись нахрест за допомогок єдиної лінії керування або на пряму передачу з входу н вихід.

На рис.5.4 показано, як проходять сигнали всередин "чорного ящика" деякої дельта-сітки. Якщо керуючи сигнал дорівнює нулю, то обидві лінії даних (або пучки

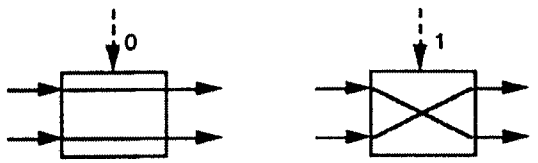


Рис. 5.4. Перемикач дельта-мережі

лінії) приєднуються прямо до вихідних ліній. У випадку, коли керуючий сигнал дорівнює одиниці, лінії даних перемикаються на виході нахрест.

З таких простих базових елементів можна будувати більш об'ємні сітки.

На рис.5.5 наведено триступеневу дельта-сітку, яка містить 8 входів з 8 виходами. Кожний елемент сітки відповідає базисному елементу, що показаний на рис.5.3.

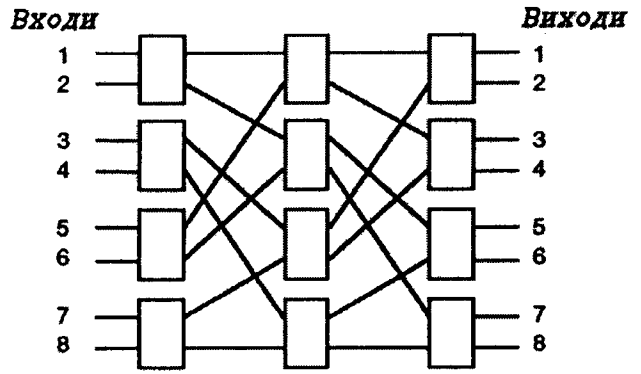


Рис. 5.5. Дельта-сітка розміром 8x8

Перевагою дельта-сіток над розподілювачами перехресних шин є менша потреба у дельта-перемикачах, що оцінюється як $\frac{n}{2} * \log n$. Вирішальним же недоліком є те, що не всі можливі комбінації зв'язків між процесорами можуть бути реалізовані. Тут можуть виникати блокування, для запобігання яким треба використовувати спеціальні концепції програмування. Блокування суттєво затримує виконання паралельної програми, що зумовлено необхідністю переконфігурації комутуючої сітки та побудови нової схеми сполучень між процесорами. Ця ситуація аналогічна тій, що має місце в телефонних мережах.

Комутуючі мережі Клоса

Спробу об'єднати кращі властивості розподілювача перехресних шин та дельта-сіток зроблено в комутуючих мережах Клоса [Clos 53], [Gonauser, Miva 89].

Вимогою до цієї мережної структури є те, що має бути забезпечена яка завгодно комбінація зв'язків між процесорами, тобто не допускається поява блокувань. З другого боку, витрати на реалізацію (кількість пунктів комутації) мають бути мінімальними. Це досягається побудовою мінімізованої за витратами багатоступеневої мережі, причому елементи одного ступеню будуються на основі простіших малих розподілювачів перехресних шин. На рис.5.6 показано триступеневу мережу Клоса з N=12 входами, розподіленими на a=4 групи, кожна з яких має m=3 входів.

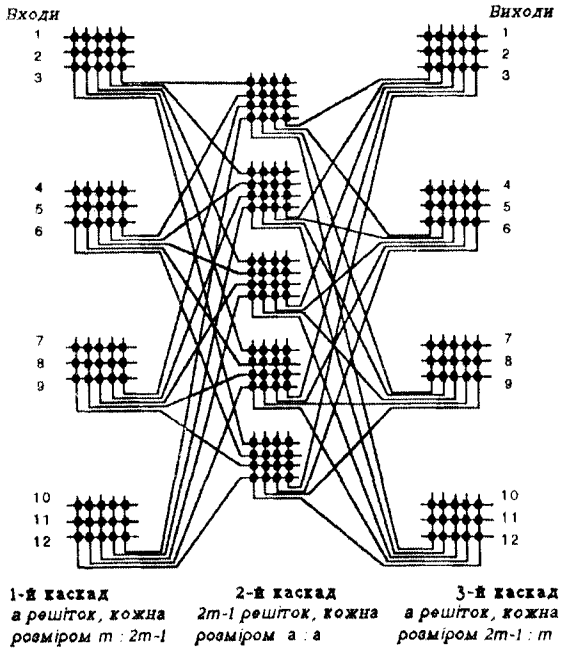


Рис. 5.6. Трикаскадна мережа Клоса для N=12 (при a=4, m=3)

У триступеневій мережі Клоса загальна кількість N шин, що перемикаються наскрізь, реалізується в першому ступені **a** малими розподілювачами перехресних шин, кожний з яких має **m** входів і 2*(**m**-1) виходів (N=**a***m). Другий ступінь побудований на 2*(**m**-1) розподілювачах перехресних шин, кожний з яких має **a** входів і **a** виходів, а третій – на розподілювачах з 2*(**m**-1) входами і **m** виходами. Параметром, який може вибиратися під час конфігурування мережі Клоса, є, таким чином, **m**. Як легко довідатись, загальна кількість пунктів комутації триступеневої мережі Клоса буде приблизно мінімальною, якщо **m** визначити за формулою

$$m \approx \sqrt{N/2}$$

для N≥24, коли мережа Клоса ефективніша, ніж простий розподілювач перехресних шин).

Загальна кількість пунктів комутації в триступеневій мережі Клоса обчислюється за формулою

$$K = m*(2*m-1)*a + a^2*(2*m-1) + (2*m-1)*m*a =$$

$$= m*(2*m-1)*2*a + a^2*(2*m-1)$$

Вона містить (2***a**) розподілювача перехресних шин розміром "**m**:(2***m**-1)" на ступенях 1, 3 і (2***m**-1) розподілювача розміром "**a**:**a**" на ступені 2. З урахуванням того, що N=**a***m, маємо

$$\Rightarrow K = (2*m-1)*\left(\frac{N^2}{m^2} + 2*N\right).$$

Якщо кількість входів **m** оптимальна, то витрати на триступеневу мережу Клоса становлять приблизно $\approx \sqrt{32}*N^{3/2}$.

На противагу цьому витрати на повний розподілювач перехресних шин відповідного розміру становлять приблизно N^2.

Приклад.

Для триступеневої мережі Клоса з 1000 входами та виходами справедливо:

$$m \approx \sqrt{1000/2} \approx 22{,}4$$

Беремо найближче **m**=20, що дає **a**=N/**m**=50.

Загальна кількість пунктів комутації буде

$$K = 39 * (\frac{1000000}{400} + 2000) = 175500$$

Ця кількість значно менша, ніж у повному розподілювачі перехресних шин цього розміру:

$$K_{KS} = N^2 - N = 999000$$

Таким чином, мережа Клоса заощаджує понад 82% пунктів комутації порівняно з розподілювачем перехресних шин. Її можна назвати "багатоступеневим розподілювачем перехресних шин", в якому зменшена кількість пунктів комутації коштує більшої затримки комунікацій між процесорами (дані мають пройти через ступені мережі Клоса). Розміри розподілювачів перехресних шин, що застосовуються як елементи мережі Клоса, вибирають відповідно до кількості комутованих процесорів і це виключає можливість блокувань зв'язків.

Мережі типу Fat-Tree

Fat-Tree – це воістину нова розробка в галузі деревоподібних структур зв'язку [Leiserson 85]. Ця решітчаста структура може масштабуватись як за кількістю процесорів, так і за кількістю можливих з'єднань, що відбуваються одночасно. Процесори – це листя повного двоїчного дерева, в той час як внутрішні вузли – це ключові елементи (рис.5.7).

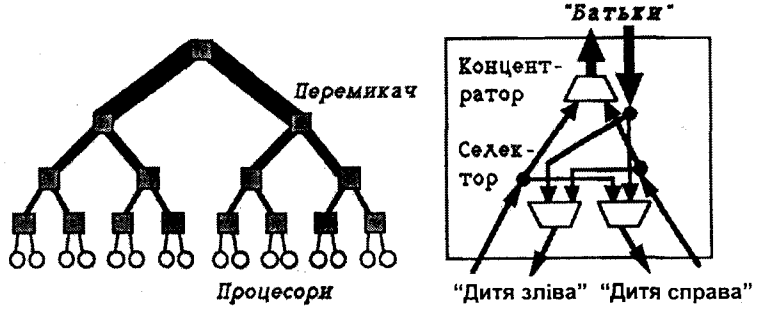


Рис. 5.7. Мережа типу "Батьківське дерево" (Fat-Tree) та перемикач

Кількість процесорів становить двоїчний порядок N=2^m , а кількість ключів перемикання дорівнює N-1. Чим вище міститься ключовий елемент у дереві, тим більше ліній зв'язку перемикається в ньому (звичайно, немає потреби у подвоєнні кількості ліній на кожному рівні дерева).

У публікації [Leiserson 85] показано, що для будь-якої довільної комунікаційної мережі може бути побудована модель у вигляді мережі Fat-Tree таким самим об'ємом апаратури і її швидкість комутації менша за швидкість довільної мережі на величину, що характеризується полілогарифмічним фактором. Це означає, що комутаційна структура Fat-Tree близька до оптимальної. Ця властивість робить її дуже цікавою для застосування в паралельних ЕОМ. Система CM-5 (Connection Machine) фірми Thinking Machines Corporation побудована на основі Fat-Tree.

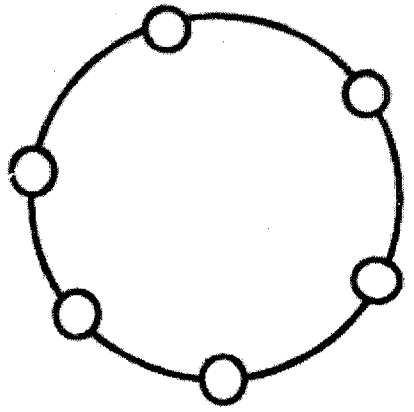
5.3. Структури, що забезпечують зв'язок типу "пункт-пункт"

Розгляньмо статичні структури зв'язу "пункт-пункт". При цьому застосуємо такі параметри:

n – кількість процесорних елементів (ПЕ) в мережі.
V– кількість ліній зв'язку у кожного ПЕ.
A– максимальна відстань між двома ПЕ.

Кільце

Кільцева структура (рис.5.8) має тільки дві лінії зв'язку на кожний ПЕ (дуже позитивна властивість) і потребує в найгіршому випадку n/2 кроки для обміну даними між двома ПЕ, що розміщуються в кільці на найбільшій відстані один від одного (дуже негативна властивість).



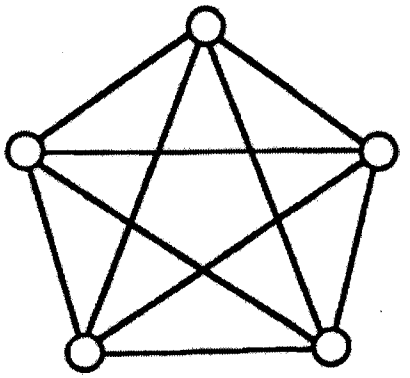
$V = 2$

$A = \frac{n}{2}$

Рис. 5.8. Кільце

Повний граф

Протилежністю кільця є повний граф (рис.5.9). Його оптимальна коннективність (кожний ПЕ досягається безпосередньо з будь-якого іншого ПЕ) забезпечується n-1 лінією зв'язку на кожний ПЕ.



$V = n-1$

$A = 1$

Рис. 5.9. Повний граф

Решітки і тори

Дуже часто застосовуються решітчасті структури зв'язку і їхні замкнуті варіанти – тори. На рис.5.10 показано різницю між чотири- і восьмизв'язковими структурами.

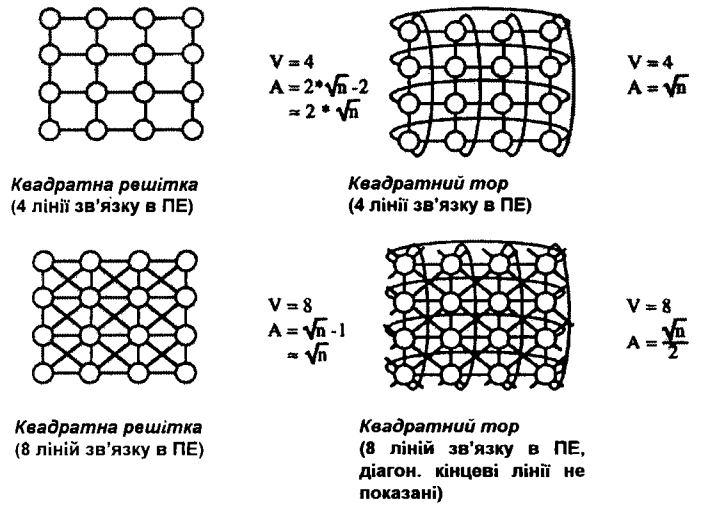


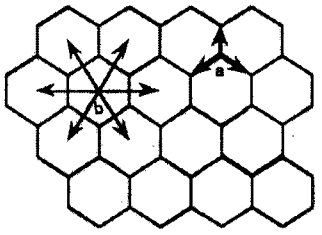
Рис. 5.10. Квадратні решітки та тори

Усі квадратні решітки мають максимальну відстань між ПЕ, що дорівнює квадратному кореню від кількісті ПЕ. Збільшення кількості ліній вдвічі у восьмизв'язковій решітці робить відстань між ПЕ наполовину меншою. Такий самий ефект досягається переходом до замкнутого тора, в якому кількість ПЕ залишається без зміни.

Гексагональна решітка

Гексагональна решітка, що також має дві координати, може розглядатись як видозміна квадратної решітки. Залежно від того, де розміщені процесорні елементи (на

перехресті чи в центрі стільника, рис.5.11 а,б), вони потребують 3 або 6 ліній зв'язку. Максимальна відстань між процесорами зумовлена двовимірністю решітки і в обох випадках пропорційна $\sqrt{n}$.



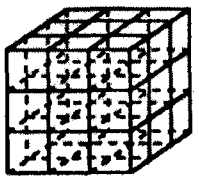
a) *ПЕ в кутку*
$V = 3$
$A = 2 * \sqrt{n}$

b) *ПЕ в центрі*
$V = 6$
$A = \frac{3}{2} * \sqrt{n}$

Рис. 5.11. Гексагональна решітка

Кубічна решітка

У кубічній решітці зроблено перехід від двовимірної до тривимірної решітки. Процесорні елементи розміщені в просторі куба, потребують 6 ліній зв'язку з ПЕ-сусідами. Максимальна відстань між ПЕ тут зменшується, вона пропорційна $\sqrt[3]{n}$ (рис.5.12).



$V = 6$
$A = 3 * \sqrt[3]{n} - 3$
$\approx 3 * \sqrt[3]{n}$

Рис. 5.12. Кубічна решітка

Гіперкуб

Гіперкуб нульової вимірності – це єдиний елемент (рис.5.13, ліворуч). Гіперкуб вимірності $i+1$ виникає з двох гіперкубів вимірності i, в яких елементи, що взаємодіють, з'єднані між собою. Наприклад, на рис.5.13 показано, що з двох квадратів (гіперкубів вимірності 2) можна побудувати куб (гіперкуб вимірності 3), з'єднавши кожний елемент "переднього" з кожним елементом "заднього" квадрата.

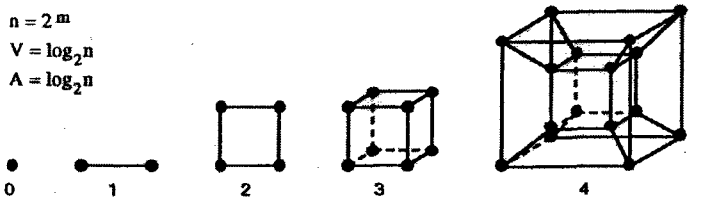
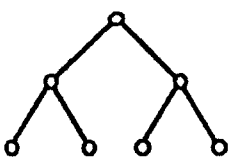


Рис.5.13. Гіперкуб з розмірностями від нуля до чотирьох

Кожний ПЕ потребує $\log_2 n$ з'єднанн з сусіднім ПЕ. Кількість зв'язків тут на відміну від попередніх решітчастих структур не є постійною. До того ж максимальна відстань між ПЕ змінюється на таку саму логарифмічну величину. Тому гіперкуб є універсальною мережею зв'язку з невеликою логарифмічною відстанню і не дуже великою логарифмічною кількістю ліній зв'язку у кожного ПЕ.

Двоїчне дерево

Наступна структура зв'язку, що має логарифмічну відстань між ПЕ, це двоїчне дерево (рис.5.14). У двоїчних дерев, що тут розглядаються, логарифмічна відстань забезпечується лише трьома лініями зв'язку кожного ПЕ. Недоліком цієї деревоподібної структури є "вузьке місце кореня", яке обмежує обмін даними між парами ПЕ з різних піддерев подібно до того, як це робить шинна система. Деякою мірою цей недолік зменшує застосування Fat-Tree (п.5.2).

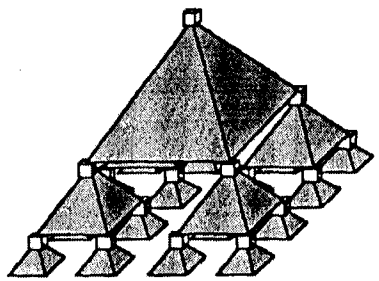


$n = 2^m - 1$
$V = 3$
$A = 2 * \log_2 (n+1) - 2$
$\approx 2 * \log_2 n$

Рис. 5.14. Двоїчне дерево

Пірамідальне дерево

У цьому дереві (quadtree, рис.5.15) кожна вершина має 4 послідовники в дереві, що, між іншим, може використовуватися для побудови алгоритмів розподілу образів у квадрантах. Максимальна відстань між ПЕ в деревах завжди дорівнює подвоєній висоті дерева, тому що в найнесприятливішому випадку зв'язок між двома листями має відбуватися через корінь. Ця відстань обчислюється за формулою, що наведена на рис.5.15.



$n = \frac{1}{3}(4^m - 1)$

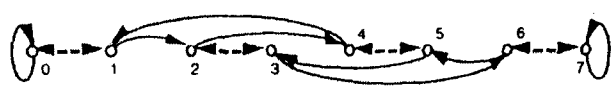
$V = 5$

$A = 2 * \log_4 (3*n+1) - 2$
$\approx 2 * \log_4 n$

Рис. 5.15. Пірамідальне дерево

Ротація-заміна (Shuffle-Exchange)

До структур з логарифмічною вимірністю належить також мережа Shuffle-Exchange (ротація, або циклічний зсув та заміна). Вона складається з двох розділених видів зв'язку – одностороннього "Shuffle" та двостороннього "Exchange" (рис.5.16).



$n = 2^m$

$V = 2$ (1 *двонаправлена і 2 однонаправлені лінії*)
$A = 2 * \log n$

Рис. 5.16. Мережа типу "Shuffle-Exchange"

Обидві структури зв'язку можуть бути дуже просто представлені за допомогою операції відображення номерів ПЕ бінарним способом запису (p_i–біт бінарного номера ПЕ, i=1, 2, ..., m):

Shuffle $(p_m, p_{m-1}, ..., p_1) = (p_{m-1}, ..., p_1, p_m)$ – одна ротація вліво бінарного номера ПЕ (**m** разів);

Exchange $(p_m, ..., p_1) = (p_m, ..., p_2, \overline{p_1})$ – заміна молодшого біта бінарного номера ПЕ операцією заперечення.

Shuffle-частина з'єднує кожний ПЕ з тим ПЕ, бінарний номер якого обчислюється циклічною операцією ротації вліво. Перший та останній елементи, що мають однакові двоїчні розряди в номерах (на рис.5.16 це ПЕ N_0 – 000, ПЕ N_7 – 111) відображаються самі на себе, тоді як інші ПЕ з'єднані в цикли.

Exchange-зв'язок у двоїчній формі запису заперечує наймолодший біт ПЕ, тобто Exchange з'єднує двонаправлено кожний парний ПЕ з своїм правим сусідом.

Приклад застосування операцій Shuffle та Exchange:

1) $001 \xrightarrow{sh.} 010 \xrightarrow{sh.} 100 \xrightarrow{sh} 001$

2) $011 \xrightarrow{ex.} 010 \xrightarrow{ex} 011$ двосторонній зв'язок ПЕ3 ↔ПЕ2

Плюс-мінус 2^i-мережа (PM2I)

Остання з розглянутих тут мереж дещо складніша. При наявності $n = 2^m$ вузлів (вершин) мережі (процесорних елементів ПЕ) вона складається з $2*m-1$ окремих структур зв'язку, які позначаються як

$$PM_{+0}, PM_{-0}, PM_{+1}, PM_{-1}, PM_{+2}, PM_{-2}, ..., PM_{m-1}.$$

Для *PMs* справедливі такі визначення:

$$PM_{+i}(j) = (j + 2^i) \bmod n;$$

$$PM_{-i}(j) = (j - 2^i) \bmod n.$$

Індекс кожної односторонньої структури показує, якою має бути відстань до сусідньої вершини (у двоїчному порядку). Для PM_{+0} відстань буде $+2^0 = +1$, для

PM_{-0} відповідно $-2^0 = -1$. Відстань між вершинами для PM_{+2} при цьому є $+2^2 = +4$, для PM_{-2} буде – 4. Для вищого показника *m-1* з урахуванням замкнутості структури має місце співвідношення:

$$PM_{+(m-1)} \equiv PM_{-(m-1)}.$$

Обидві структури ідентичні, тому існує тільки *2*m-1*, а не *2*m* структур.

На рис.5.17 показано мережу *PM2I* з n=8 ПЕ. PM_{+0} і PM_{-0} зв'язують кожний ПЕ з своїм правим або лівим сусідом і створюють таким чином двостороннє кільце зв'язку. PM_{+1} і PM_{-1} включають дві розділені односторонні структури по чотири ПЕ. Кожний ПЕ має сусіда з номером через один, причому Modulo-операція утворює структуру зв'язку типу "кільце". Разом вони утворюють два окремих двосторонніх кільця, як представлено на рис.5.17. Остання структура PM_{+2} ідентична з PM_{-2}. Кожний ПЕ зв'язаний тут з іншим ПЕ на відстані 4 (Modulo 8).

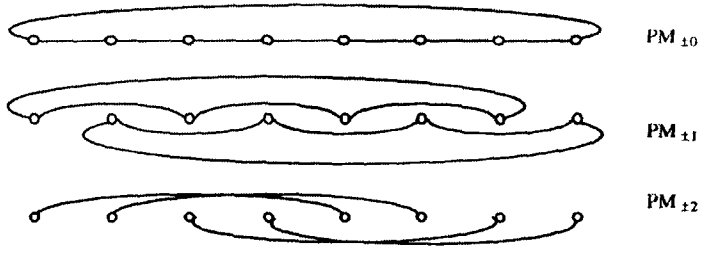


Рис. 5.17. Мережа типу *PM2I*

5.4. Порівняння мереж

Як уже було зауважено, порівняння мереж можливе тільки на основі параметрів V (кількість ліній зв'язку кожного ПЕ) і A (максимальна відстань між двома ПЕ). Це порівняння або аналіз того, які переваги має певна мережа, можуть дати корисні результати тільки з урахуванням особливостей вирішуваної паралельної задачі. Залежно від умов застосування одна структура зв'язку може бути ефективнішою за іншу структуру. Якщо алгоритм розв'язування задач потребує певної мережі, то, як правило, фізична реалізація якраз цієї, можливо "гіршої" за параметрами V і A мережі, дасть кращі результати, ніж пристосування для цієї задачі іншої мережі, що має "кращі" параметри.

Оскільки для вирішення кожної проблеми неможливо будувати спеціальну ЕОМ, треба йти іншими шляхами. Кожна паралельна ЕОМ має у своєму розпорядженні тільки одну або дві (як в деяких випадках) комутаційних структури, які, звичайно, можуть бути як мінімально багатосторонніми і, можливо, здатними до динамічної конфігурації. Усі алгоритми, які підлягають імплементації в цій паралельній ЕОМ, мають задовольнятися наявною мережею (або її частиною). Цінність мережі залежить, таким чином, від того, наскільки якісно вона може використовуватись в середньому для вирішення проблем, що постають найчастіше. Паралельна ЕОМ MasPar MP-1 (див.п.11.1) функціонує, наприклад, із застосуванням двох різних мереж: решітчастої структури та приступеневої мережі Клоса, яка пропонує хорошу загальну конективність. Алгоритми, які потребують решітчастої структури, можуть використовувати безпосередньо цю швидку локальну структуру, тоді як алгоритми, яким потрібні інші мережні структури, можуть бути реалізовані з використанням маршрутизації на глобальній мережі Клоса з деякими втратами ефективності. Інші паралельні ЕОМ, такі як "Distributed Array Processor" (DAP, див. п.11.1), мають лише квадратну решітку, що обмежує можливості використання їх, порівняно з наведеними вище, на ті прикладні області, які можуть ефективно функціонувати з решітчастою структурою (наприклад, чисельні алгоритми або розпізнавання образів).

Зігель [Siegel 79] моделював одні мережі застосуванням інших мереж. При цьому виявляєтьс передусім перевага Shuffle-Exchange та гіперкуба я "універсальних мереж", тобто як засобів ефективно емуляції декількох різноманітних мережних структур Решітка тільки тоді виправдана, коли решітчаста структур потрібна для реалізації алгоритмів; для моделюванн інших мереж вона непридатна. Насамперед емуляція мереж може бути виконана автоматично компілятором лише тод коли крім мережної структури цільової ЕОМ відома тако структура, яка потрібна для реалізації прикладно програми. Однак, на жаль, це трапляється лише небагатьох випадках. Найчастіше ті реляції пр міжпроцесорні зв'язки, що мають забезпечити обмі даними, задані у вигляді складних арифметичних виразів мало відрізняються від "стандарту" або взагал обчислюються і стають відомими лише перед запуском програм. З цих відомостей неможливо автоматичн визначити , про яку структуру комутаційної мережі йд мова. Обмін даними в цих випадках може проводитис тільки за допомогою загального і найменш ефективног алгоритму маршрутизації, який не бере до уваги жодно інформації з прикладних програм.

Таблиця на рис. 5.18 дає кількість потрібних цикл моделювання залежно від кількості n процесорни елементів.

Мережа 1 моделює ↓ / мережу 2 →	Двовимірна решітка	*PM2I*	Shuffle-Exchange	Гіперкуб
Двовимірна решітка	–	$\approx \sqrt{n}/2$	$\approx \sqrt{n}$	$\sqrt{n}$
PM2I	1	–	$\approx \log_2 n$	2
Shuffle-Exchange	$\approx 2\log_2 n$	$\approx 2\log_2 n$	–	$\approx \log_2 n+1$
Гіперкуб	$\log_2 n$	$\log_2 n$	$\log_2 n$	–

Рис. 5.18. Порівняння комутаційних мереж /Siegel 79/

Вправи до розділу I

1. Векторний (перехресний) добуток від множення двох векторів треба представити за допомогою базисних операцій. Окрім ряду проміжних кроків тут слід також записувати в пам'ять вектор проміжних результатів у формі допоміжної змінної:

$((a_x,a_y,a_z),(b_x,b_y,b_z)) \mapsto ... \mapsto (a_yb_z - a_zb_y, a_zb_x - a_xb_z, a_xb_y - a_yb_x)$.

2. Запропонуйте просту мережу Петрі для синхронізації трьох паралельних процесів.

3. Побудуйте розширену мережу Петрі, котра виконує такі обчислення:

$$\mathbf{Z}:= \begin{cases} \mathbf{X}+2, \text{ якщо } \mathbf{Y}>0; \\ 2*\mathbf{X}, \text{ якщо } \mathbf{Y}=0. \end{cases}$$

4. Побудуйте розширену мережу Петрі, котра виконує операції "розділити":

$$\mathbf{Z}:= \mathbf{X} \text{ div } \mathbf{Y}.$$

Вершини для **X**, **Y** та **Z** містять невід'ємні числа і можуть з'являтися лише один раз. У розширеній мережі Петрі мають бути помічені вершини "старт" , "готово", "помилка". В разі спроби "розділити на нуль" повинні маркуватись вершини "готово" та "помилка".

5. Побудуйте розширену мережу Петрі, котра виконує такі обчислення:

$$\mathbf{Z}: = \mathbf{X} \bmod \mathbf{Y}$$

Вершини для **X**, **Y**, **Z** містять невід'ємні числа і можуть з'явитися лише один раз. У мережі мають бути помічені вершини "старт", "готово" та "помилка". У випадку, якщо **Y**=0, мають маркуватися вершини "готово", "помилка".

6. Побудуйте розширену мережу Петрі для віднімання цілих чисел. Знак числа має встановлюватися за допомогою додаткової вершини.

7. а) доведіть, що в комутаційній мережі типу Shuffle-Exchange треба виконати в найнесприятливішому

випадку 2*log(n)-1 кроків, які забезпечать обмін даними між двома процесорними вузлами;

б) у чому полягає взаємозв'язок між дельта-мережею та мережею Shuffle-Exchange?

в) у чому полягає взаємозв'язок між гіперкубом та мережею типу PM2I?

8. Найдіть (приблизно) оптимальні значення для **m** та **a** в триступеневій мережі Клоса з N=16.384

9. Загальна кількість ліній зв'язку триступеневої мережі Клоса є майже оптимальною, якщо для групування m вибрати число $\mathbf{m} \approx \sqrt{N/2}$.

Доведіть це правило оптимізації.

10. Мережа Клоса лише за умови $N \geq 24$ економічно вигідніша, ніж перехресний шинний розподілювач.

Доведіть справедливість цього твердження за допомогою правила із завдання 9.

II

АСИНХРОННА ПАРАЛЕЛЬНІСТЬ

Відповідно до двох великих класів паралельної обробки інформації розрізняють синхронну та асинхронну паралельності. За "класичної" асинхронної паралельності задача, що підлягає розв'язанню, поділяється на задачі-частини, які у формі процесів підпорядковуються групі самостійних, незалежних процесорів, тобто асинхронна паралельна програма складається з багатьох керуючих потоків. Задачі-частини не можуть бути повністю незалежними одна від одної, тому відповідні цим задачам процеси мають обмінюватися даними і для цього обміну себе взаємно синхронізувати. Процеси виконують, як правило, великі задачі-частини, тому що поділ на менші задачі, такі як арифметичні вирази, вимагає в порівнянні з ефектом розпаралелювання дуже високих витрат на синхронізацію. З цієї причини асинхронну паралельність часто називають "великоблоковою паралельністю".

6. ПОБУДОВА MIMD-ЕОМ

Узагальнену модель MIMD-EOM (multiple instrucktion, multiple data) показано на рис.6.1.

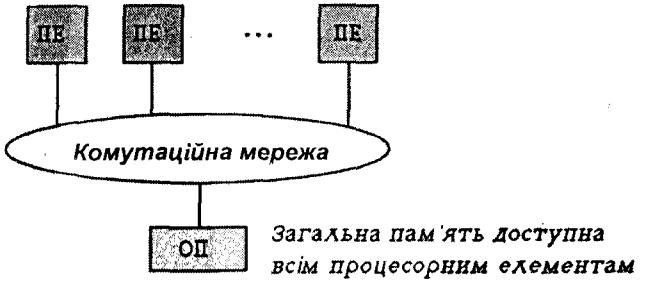


Рис.6.1. MIMD-модель обчислювальної системи з загальною пам'яттю

Процесори (ПЕ) – це самостійні машини, які можуть виконувати програми незалежно один від одного. Залежно від конфігурації вони можуть мати у своєму розпорядженні локальну пам'ять або звертатися до загальних глобальних блоків пам'яті. Зв'язок між процесорами і глобальними блоками пам'яті здійснюється через комутаційну мережу. Про різницю між тісно зв'язаними MIMD-EOM з загальною глобальною пам'яттю і слабко зв'язаними MIMD-EOM з виключно локальною пам'яттю вже говорилось в гл.2. У той час як у MIMD-EOM із загальною пам'яттю в основному застосовується шинна система зв'язку між процесорами, в MIMD-EOM без загальної пам'яті часто впроваджуються складніші мережні структури. Враховуючи, що шинна структура годиться тільки для обмеженої кількості процесорів, більшість MIMD-EOM з великою кількістю процесорів не мають фізичної загальної пам'яті. У цих системах може моделюватися загальна область пам'яті, що потрібна для протоколів обміну

даними, на основі концепції "віртуальної сумісно використовуваної пам'яті" ("shared virtual memori").

Процесори працюють незалежно один від одного. Для обміну даними вони мають синхронізуватися. Структурно MIMD-програма відображає апаратну структуру: вона поділяється на декілька самостійних процесів, які виконуються асинхронно паралельно на закріплених за ними процесорах і для обміну даними мають синхронізуватися за допомогою спеціальних механізмів. Ідеальним для цих систем було б співвідношення 1 процес/ 1 процесор, але на практиці у зв'язку з обмеженою кількістю процесорів, що мають у своєму розпорядженні MIMD-системи, це неможливо. Тому загальне співвідношення виглядає як n процесів/1 процесор. Це означає, що один процесор має обслуговувати декілька процесів за методом розподілу часу, який потребує програми-диспетчера для кожного процесора та додаткових витрат ресурсів на керування.

Як типові представники MIMD-класу паралельних систем із загальною пам'яттю та без неї тут розглянуто системи Sequent Symmetry, а також Intel IPSC Hypercube та Paragon. Повнішу інформацію про побудову і, зокрема, апаратну сторону паралельних систем дано в [Hwang, Briggs 84], [Almasi, Gottlieb 89].

6.1. Обчислювальні системи типу MIMD

У цьому параграфі коротко представлені обчислювальні MIMD-системи Sequent Symmetry (шинна структура зв'язку), Intel Hupercube (мережа гіперкуба) і Intel Paragon (двовимірна решітка).

Sequent Symmetry

Ця система є прикладом тісного зв'язку між процесорами та загальною глобальною пам'яттю через

центральну шину (рис. 6.2).

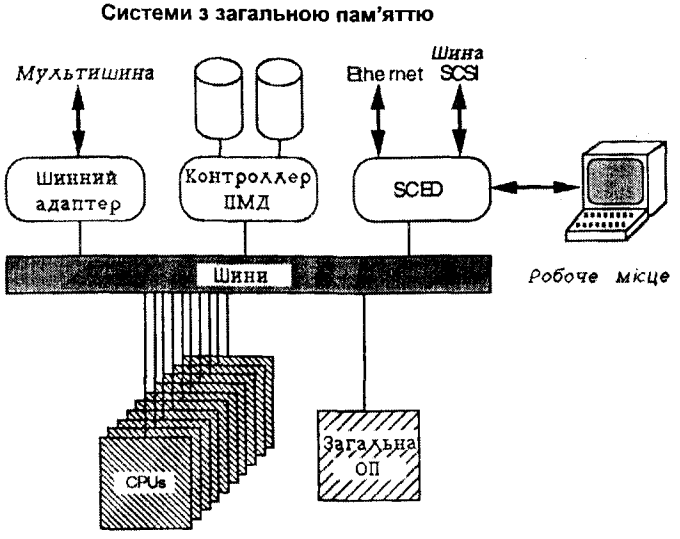


Рис. 6.2. Блочна структура системи Sequent Symmetry

Усі десять центральних процесорних пристроїв (CPU), загальна пам'ять та засоби прив'язки периферійних приладів взаємодіють через системну шину.

Шина не допускає паралельності під час обміну даними, бо тут завжди можуть взаємодіяти лише два партнери. Це обмежує можливість розширення цієї паралельної системи. Процесори, що застосовуються в Sequent Symmetry (80386), вже не належать до найпродуктивніших. Сьогодні на ринку є однопроцесорні робочі місця, що мають більшу потужність, ніж ця паралельна ЕОМ з десятьма процесорами, і не потребують розпаралелювання програми.

Intel iPSC Hypercube та Paragon

Ряд Intel Scientific Computer (iSC) складається з трьох поколінь iPC/1 (Intel Personal Supercomputer), iPSC/2 та iPC/860 [Trew, Wilson 91], базою яких є процесорні блоки

CPU 80286, 80386 та 80860. Попередником архітектури, що застосовується фірмою Intel , був комп'ютер CosmicCube, побудований на CalTech, Pasadena.

В системі iPSC/860 можуть об'єднуватися до 128 високопродуктивних процесорів. Ця структура зв'язку забезпечує набагато вищу паралельність обміну даними, ніж шина або кільце.

Системна архітектура Intel Hypercube не має загальної пам'яті. Процесори "слабко" з'єднані тільки через комутаційну мережу і для обміну даними між собою мають використовувати відносно повільні засоби передачі повідомлень. Механізми синхронізації процесів, що базуються на загальній пам'яті, наприклад семафори та монітори, можуть використовуватися лише локально на окремих процесорах, але не між різними процесорними вузлами.

Paragon XP/S – це найновіша на цей час модель паралельної ЕОМ фірми Intel. Вона з'явилась за проектом "Touchstone-Delta" і будується на процесорах i860XP. Ця обчислювальна система може містити до 512 вузлів, причому кожний вузол складається з двох процесорів типу i860 XP, один з яких призначений для виконання арифметико-логічних задач, а другий – тільки для обміну даними. Структура зв'язку між вузлами системи Paragon набагато простіша, ніж в системі IPSC Hypercube: вузли з'єднані між собою через двовимірну решітку (4-way nearest neighbor). Вузли можуть динамічно конфігуруватися за стовпцями на потребу користувача (compute node), для системних цілей (service node) або з метою підключення периферійної апаратури (I/O node).

6.2. Стани процесів

Процесс – це самостійна частина програми, що виконується асинхронно паралельно відносно інших

процесів. У зв'язку з тим що в загальному випадку на одному процесорі виконується декілька процесів, розподіл процесорного часу між процесами здійснюється у формі часових сегментів. Після закінчення належного йому часу активний процес, що виконується, переходить зі стану "розрахунок" в стан "готовий" і чекає в черзі на свою нову активізацію. При цьому в системному блоці керування процесом (Process-Control-Block PCB) мусять бути запам'ятовані дані для зміни станів процесу (програмний лічильник, регістр, адреси даних тощо), щоб потім знову бути введеними для продовження виконання процесу. Нові процеси, що з'являються на вході системи, вводяться в чергу чекання "готовий", а ті, що використали свій час, звільняють процесор для наступного процеса.

Процеси, що змушені порівняно довго чекати на появу умови, за якої може бути продовжене виконання їх (звільнення певного операційного ресурса, синхронізація з іншим процесом тощо), не обов'язково мають завантажувати процесор переходами зі стану "готовий" в стан "розрахунок". Вони переходять в стан "блокований" і чекають в специфічній черзі на появу умови для розблокування. Тільки з появою цієї умови блокований процес знову переводиться в чергу "готовий" і здійснює запит на обслуговування процесором. Частина операційної системи, яка виконує операції зміни стану процесів, називається планувальником (*scheduler*). Для кожного процесора існує такий планувальник, який керує послідовністю обробки процесів в цьому процесорі. Складніші методи, про які мова йтиме далі, виконують планування відносно декількох процесів одночасно (наприклад, перезавантаження деякого процесу з одного процесора в інший, менш завантажений процесор). Метою такого планування є кращий розподіл завантаження системи. При цьому питання, який процес яким процесором обслуговується, вирішується незалежно від програміста.

СИНХРОНІЗАЦІЯ ТА КОМУНІКАЦІЯ В MIMD-СИСТЕМАХ

У цій та наступних главах береться за основу концепція паралельності процесів та їхньої взаємодії із застосуванням Remote-Procedure-Call (рис.7.1), що представлена в п.п. 4.4, 4.5. За умови паралельного виконання процесів виникають дві проблеми MIMD-програмування.

Рис. 7.1. Обмін повідомленнями

1. Два процеси бажають обмінятися між собою даними.
2. Якщо є загальна пам'ять, то має бути виключена можливість одночасного доступу декількох процесів до однієї й тієї області даних (синхронізація для запобігання виникненню помилкових даних або блокувань – див. гл.8).

Два партнери одного обміну даними діють незалежно, асинхронно і, можливо, на різних процесорах. Для обміну даними вони мають спочатку синхронізуватись: обмін може бути здійснений тільки після того, як обидва партнери будуть для цього готові. Друга проблема виникає під час вирішення першої: якщо обмін даними проводиться через загальну частину пам'яті, то треба бути певним, що доступ (на читання або запис) є послідовним. Якщо це не так, то можуть виникнути помилкові дані або блокування (наприклад, запит помилкових даних перед операціями синхронізації).

Почнемо розгляд з найпростішого випадку, коли обидва процеси, що мають здійснити взаємодію, виконуються на одному й тому процесорі. Для цього існує

цілий ряд рішень на різних рівнях абстракції. Більш складний випадок взаємодії між процесорами розглядається наприкінці глави. Загальний хід операцій синхронізації може бути промодельований і проаналізований за допомогою мереж Петрі (див. гл. 3).

7.1. Програмне рішення

Петерсон і Зільбершатц описують рішення проблеми синхронізації для систем із загальною пам'яттю, яка обходиться без спеціальної апаратури [Peterson, Silberschatz 85]. Це чисто програмне рішення розглядається тут поступово.

Програма складається з двох паралельних процесів, що хотіли б звернутися до загальної області даних. Операції над даними (читати або записувати) не мають принципового значення і тому називаються далі як <*критичний розділ*>. Інструкції кожного процесу, що не пов'язані з синхронізацією, позначені як <*інші інструкції*>.

Обидва процеси P_1 і P_2 запускаються в головній програмі один за одним і виконуються з цього моменту далі паралельно:

```
....
start(P1);
start(P2);
....
```

Перший дослід

Досліджувана тут можливість для синхронізації – це застосування змінних синхронізації. Обидва процеси можуть звернутися до загальної змінної величини turn. При цьому читання або запис значення цієї змінної розуміються як атомарні (елементарні, неділимі) операції.


```
var turn:1..2;
                Ініціалізація: turn:=1;
```

P_1	P_2
loop	**loop**
while turn≠1 **do** (**пусто**) **end**;	**while** turn≠2 **do** (**пусто**) **end**;
<*критичний розділ*>	<*критичний розділ*>
turn:=2;	turn:=1;
<*інші інструкції*>	<*інші інструкції*>
end	**end**

Аналіз:

- це рішення гарантує, що в <*критичний розділ*> може звернутися лише один процес;
- але тут існує суттєвий недолік: примусовий альтернативний доступ обох процесів. Цей недолік обмежує можливості програмного розв'язання задачі синхронізації.

Кожен процес (P_1 і P_2) чекає на доступ в <*критичний розділ*> в циклі типу "busy-wait" (зайняти чергу) до того моменту, коли змінна синхронізації turn не набуде значення, що дорівнює номеру даного процесу (так, для P_1 turn=1). Після цього виконується програма <*критичний розділ*> і змінна turn набуває значення номера іншого процесу. Ця синхронізація функціонує, але так, що після закінчення процесом P_1 роботи в <*критичному розділі*> він може знову туди звернутися лише після того, як процес P_2 виконає програму <*критичний розділ*>. Така побудова синхронізації означає нічим не виправдане обмеження взаємодії процесів: на практиці часто може бути така ситуація, що процес P_1 має багаторазово виконати програму <*критичний розділ*> і записати там дані, які процес P_2 має лише один раз прочитати.

Другий дослід

Змінна синхронізації замінюється тут полем з двох булівських значень (одне для кожного процесу). Перед тим, як одержати доступ в <*критичний розділ*>, кожний процес

надає своєму елементу поля значення true і після закінчення програми <*критичний розділ*> повертає йому значення false. Цим процес позначає для будь-якого іншого процесу одержання дозволу на доступ в <*критичний розділ*>. Всі інші процеси чекають в циклі типу "busy-wait" (while...do (**пусто**) end) моменту, коли звільниться <*критичний розділ*>. Це рішення програмується так:

```
var flag: array [1..2] of BOOLEAN;
Ініціалізація: flag[1]:= false; flag[2]:= false;
```

P_1	P_2
loop	**loop**
while flag[2] **do**	**while** flag[1] **do**
(**пусто**)	(**пусто**)
end;	**end**;
flag[1]:=true;	g[1]:=true;
<*критичний розділ*>	<*критичний розділ*>
flag[1]:=false;	flag[2]:=false;
<*інші інструкції*>	<*інші інструкції*>
end	**end**

Аналіз:

- незважаючи на попередню перевірку значення змінної flag в циклах чекання, існує можливість одночасного доступу обох процесів до <*критичного розділу*>. Це зумовлює небезпеку помилок!

Аналіз цього рішення показує, що синхронізація дається не так просто. В тому випадку, коли обидва процеси одночасно залишають свої while-цикли, вони одержують доступ в <*критичний розділ*>. Але якраз ця ситуація має бути виключена, якщо синхронізація правильна!

Третій дослід

Як показав другий дослід, попереднє опитування значення змінної синхронізації не задовольняє організацію взаємодії процесів P_1 і P_2. Тому напрошується ідея такої модифікації програми, яка б забезпечувала присвоєння значення true логічній змінній flag циклом чекання while ще до "busy-wait". Тим самим опитування і чекання, так би мовити, відбуваються разом з <*критичним розділом*>.

Програма з цією модифікацією:

```
var flag: array [1,2] of BOOLEAN;
Ініціалізація: flag[1]:= false; flag[2]:= false;
```

P_1	P_2
loop	**loop**
flag[1]:= true;	flag[2]:= true;
while flag[2] **do** (**пусто**) **end**;	**while** flag[1] **do**(**пусто**) **end**;
<*критичний розділ*>	<*критичний розділ*>
flag[1]:= false;	flag[2]:= false;
<*інші інструкції*>	<*інші інструкції*>
end	**end**

Аналіз:

- тут гарантується, що в деякий момент тільки один процес може ввійти в <*критичний розділ*>;
- у випадку, коли обидва процеси одночасно "виставлять" flag:=true і потім будуть чекати, виникне блокування ("Livelock").

Отже, цей варіант може призвести до помилок і "зависання" обчислювального процесу. Хоч умова взаємного виключення одночасного доступу досягнута, все ж може виникнути ситуація, коли жоден процес не може вийти з циклу чекання, тобто настане блокування паралельної системи програмування.

Четвертий дослід

Цей дослід дає найскладніше, але (нарешті) коректне рішення проблеми синхронізації за Петерсоном [Peterson 81]. Тут застосовуються як проста змінна синхронізації turn, так і поле логічних змінних flag. Спочатку за допомогою змінної flag процес показує, що він має потребу звернутися до <*критичного розділу*>. Потім він дає змінній turn номер

іншого процесу і чекає (якщо інший процес також уже виставив flag і дав змінній turn номер цього процесу) до моменту, коли настане зміна. Після доступу в *<критичний розділ>* кожен процес гасить свій елемент flag.

Програма має вигляд:

```
var turn: 1..2
flag: array [1..2] of BOOLEAN;
Ініціалізація:  turn:= 1; (*будь-яка*)
flag[1]:=false; flag[2]:=false;
```

P_1	P_2
loop	**loop**
flag[1]:=true;	flag[2]:=true;
turn:=2;	turn:=1;
while flag[2] **and** (turn=2)	**while** flag[1] **and** (turn=1)
do	**do**
(*пусто*)	(*пусто*)
end;	**end;**
<критичний розділ>	*<критичний розділ>*
flag[1]:=false;	flag[2]:=false;
<інші інструкції>	*<інші інструкції>*
end	**end**

Аналіз:

- тут гарантується, що в деякий момент в *<критичний розділ>* може ввійти тільки один процес;
- у цьому варіанті не може виникнути блокування.

Таким чином, одержано коректне рішення.

Розширене опитування умови чекання в while-циклі допомагає уникнути блокувань. Процес чекає тільки тоді, коли інший процес виставив змінну flag і він фактично стоїть у черзі на доступ в *<критичний розділ>*. Як тільки процес, що першим одержав доступ, залишає *<критичний розділ>*, він гасить змінну flag й інший процес може вийти з циклу чекання, щоб одержати доступ в *<критичний розділ>*.

Існують також опубліковані алгоритми, наприклад в [Ben-Ari 82], [Eisenberg, McGuire 72], які вирішують проблему синхронізації для довільної кількості процесів. Але цим алгоритмам притаманні такі недоліки: має місце значна втрата обчислювальної потужності, бо, як правило, немає можливості кожному процесу виділити свій процесор. Складність перешкоджає практичному застосуванню алгоритмів (див. апаратне рішення в наступному параграфі).

7.2. Апаратне рішення

Хоч програмне рішення проблеми синхронізації паралельних процесів, як показано в попередньому параграфі, можливе, але воно досить громіздке. З цієї причини в більшості обчислювальних систем застосовується апаратне рішення. В простій одноопроцесорній системі, де процеси обробляються за методом мультиплексування, це рішення може здійснюватися блокуванням реакцій на всі переривання (disable). Це означає, що переривання програм за часом неможливі і не може трапитися зміна процесів. Тільки виконуваний процес може без перешкод увійти в *<критичний розділ>* для виконання там відповідних операцій. Суттєво краще підходить для цієї проблеми операція "test_and_set" (перевірити і встановити). Це не якась-будь там "екзотична" операція, вона дуже корисна для побудови мультизадачних операційних систем, а також введена в систему команд мікропроцесорів 68020 (TAS test and set, CAS compare and swap) та 80286 (XCHG exchange).

Операція test_and_set – це дуже просте апаратне рішення і складається з двох субоперацій:

1. Читання булівського значення змінної.
2. Переписування цього значення змінної на true.

```
procedure test_and_set
(var lock: BOOLEAN): BOOLEAN;
var mem:boolean;
begin (*неділима операція*)
  mem:= lock;
  lock:= true;
  return(mem)
end; (*неділима операція*)
```

Ці обидві субопераційї мають виконуватися безпосередньо одна за одною як неділима операція, чим виключається можливість того, що між ними може "вклинитись" інший процес зі спробою доступу в *<критичний розділ>*. Тому операція test_and_set визначається як атомарна операція і часто реалізується у вигляді спеціальної команди процесора, що виконується в одному циклі інструкцій. Наприклад, в командах TAS і CAS процесора 68020 це досягається активізацією сигналу RMC (read-modify-write cycle), впродовж якої ніякий пристрій не може одержати доступу до шини або вимагати переривання виконуваної програми.

Із застосуванням апаратного рішення test_and_set значно простіше вирішується проблема синхронізації будь-якої кількості процесів. Зворотне установлення значення змінної в false може виконуватись як проста операція присвоєння і не потребує спеціальної апаратної підтримки.

Програма для процесу P_i:

```
var  lock: BOOLEAN;
Ініціалізація :  lock:= false;
                    Pi
loop
  while test_and_set (lock) do (*пусто*) end;
      <критичний розділ>
  lock:=false;
      <інші інструкції>
end.
```


Кожний процес чекає в операції "busy-wait" на момент, коли змінна синхронізації lock одержить значення false. Якщо запит командою test_and_set успішний, то змінна lock одержує значення true і процес P_i може ввійти в *<критичний розділ>*. Після виходу з цього розділу процес знову надає змінній lock значення false і тепер процес, що чекує, може за допомогою команди test_and_set резервувати собі цю змінну.

Рішення проблеми синхронізації стало тепер простим і наочним, але воно залишається дещо неефективним через необхідність виконувати цикли чекання "busy-wait". В цих циклах втрачається багато машинного часу. Якраз для заміни циклів чекання більш продуктивними чергами чекання призначено синхронізаційні конструктиви високого рівня, про які йдеться в наступних параграфах.

7.3. Семафори

Семафори введені в 1965 р. Дійкстра за аналогією з сигналами, що застосовуються для регулювання руху поїздів [Dijkstra 65]. Подібні конструктиви вже були раніше імплементовані в декількох операційних системах, але без даної назви. В найпростішому випадку семафор може бути тільки в двох станах: вільно або зайнято. Це відповідає дозволу на доступ в *<критичний розділ>* або свідчить про потребу чекати цього дозволу. Операції вмикання або вимикання сигналу семафора (щоб, наприклад, можна було проїхати одноколійну частину магістралі) називаються **P** та **V** від голландських слів, що означають "зайняти" та "залишити" *<критичний розділ>*. Семафори можуть застосовуватись не тільки для захисту *<критичних розділів>*, а й для сигналізації про зайнятість та звільненість операційних засобів (прінтер, термінал тощо). На противагу циклам чекання "busy-wait" семафори є дуже ефективним методом синхронізації.

Застосування семафору до процесу P_i:

```
Pi
....
P(sema);
<критичний розділ>
V(sema);
....
```

Як видно з цього фрагменту програми, застосування семафора мислиться просто: кожний процес P_i "бере в дужки" свої *<критичні розділи>* семафорними операціями **P**(sema) та **V**(sema). При цьому для всіх процесів, що бажають мати доступ в один і той самий *<критичний розділ>* для одержання там загальних для них даних, має бути застосована ідентична семафорна змінна! Таким чином, семафор певною мірою належить до загальних даних, а не до окремого процесу.

Реалізація семафора

Структура даних для семафора – це поєднання семафорної величини та списка черги процесів, що чекають на сигнал "вільно" цього семафора. Семафорна величина (змінна) в простих випадках може бути булівським значенням істинності й тоді йдеться про "булівський семафор". Ця величина може бути також числом типу integer (як описується тут). В цьому випадку буде так званий "загальний семафор":

```
type Semaphore = record
                   value: INTEGER;
                   L    : List_of_ProcID;
end;
var S: Semaphore;
```

Операції **P** і **V** за логікою семафора мають імплементуватись як атомарні (неділимі) операції, бо різні процеси бажають мати доступ до загальних семафорних даних.

Ініцілізація семафора:

S.L ← *пустий список*

S.value ← *кількість дозволених* **P**-*операцій без появи* **V**-*операцій.*

Простий семафор, наприклад для захисту одного *<критичного розділу>*, завжди ініціалізується числом 1. Є також випадки, в яких доцільно використовувати більше (або менше) число ініціалізації. Число, більше за 1, відповідає кількості послідовно виконуваних **P**-операцій (крім виконуваної поміж ними **V**-операції) без блокування процесу в семафорі:

```
P(S): S.value := S.value - 1;
      if S.value < 0 then
      begin
         append (S.L, actproc); (*цей процес завести в S.L*)
         block (actproc); (*і перевести в стан "блокований"*)
      end;
```

P-операція зменшує семафорне число на 1 і перевіряє, чи не менше воно від нуля, тобто чи вже зайнятий семафор. Якщо так, то виконуваний процес входиться в чергу семафора і блокується:

```
V(S): S.value := S.value + 1;
      if S.value ≤ 0 then
      begin
         getfirst (S.L, P); (*процес P вивести із S.L*)
         ready(P);          (*і перевести в стан  "готовий"*)
      end;
```

V-операція збільшує семафорне число на 1 і перевіряє, чи не залишається воно меншим або таким, що дорівнює 0, тобто процеси ще перебувають в черзі семафора. Якщо так, то наступний процес, що очікує,

виводиться із черги чекання і вводиться в список готових процесів.

Як же досягнути того, щоб **P** і **V** самі були атомарними операціями?

- Програмне рішення (див. п. 7.1):
досягається за допомогою короткого "busy-wait"-циклу з короткою **P**- або **V**-операцією як *<критичним розділом>*. Ці цикли залишаються, природно, неефективною операцією, але тепер *<критичний розділ>* містить всього дві-чотири елементарних інструкції **P**- або **V**-операцій семафора. Це означає, що витрати часу на чекання суттєво зменшуються.

- Апаратне рішення (див. п. 7.2):
досягається за допомогою короткого "busy-wait"-циклу для команди "test_and_set" перед початком **P**- або **V**-операції. В апаратному рішенні також неможливо уникнути короткого циклу чекання, але тут справедливе те, що сказано вище: завдяки дуже короткому *<критичному розділу>*, тобто самим **P**- і **V**-операціям, не виникає помітних втрат ефективності.

Однак виникає проблема, якщо процес втрачає свій часовий інтервал (сегмент) під час виконання цих "чотирьох елементарних операцій". В цьому випадку всі інші процеси, які хотіли точно так виконати семафорну операцію, використовують свій загальний часовий інтервал для "busy-wait"-циклу до того моменту, коли названий процес повернеться в чергу і може завершити семафорну операцію. Як показано далі, ця проблема стає помітною, якщо вона виникає на рівні черги чекання.

Конвой-феномен

Досвід побудови операційних систем перших поколінь показав, що внаслідок невдалої імплементації стратегії розподілу часу мав місце конвой-феномен [Blasgen та інші 79]. Подібно до автомобільних пробок на

магістралях процеси P_1-P_n можуть потрапити в чергу очікування, якщо вони часто і регулярно використовують семафори (high traffic lock). Цей феномен має місце тоді, коли кількість процесів значно перевищує кількість процесорів і якщо проводиться пріоритетний ("pre-emptive") розподіл часу за FIFO-стратегією (first in first out). Пріоритетність полягає в тому, що програмний планувальник (*scheduler*) може перевести процеси, що зайняли семафор, із стану "розрахунок" в стан "готовий". FIFO- стратегія означає, що процеси, які чекають на семафор, звільняються в порядку розташування їх у черзі.

У тому випадку, коли процес P_1 втрачає свій інтервал часу якраз під час виконання своїх критичних операцій, тобто в момент "зайняття" семафора **S** (high traffic), жоден з решти процесів P_2 -P_n не зможе зайняти семафор. Щоправда вірогідність такого випадку невелика, але все ж таки не дорівнює нулю, а це означає, що така ситуація може виникнути будь-коли, непередбачено і без можливості її репродукувати. Втрата інтервалу часу за умови зайнятого "high traffic"-семафора означає, що дуже швидко після цього процеси P_1 - P_n (велика кількість процесів), які бажають виконати тільки коротку операцію в захищеному семафором *<критичному розділі>*, будуть вишикувані в чергу семафора і продуктивність обчислювальної системи драматично зменшиться. Тільки після того як P_1 активізується на новому інтервалі часу і звільнить семафор, наступний процес P_2 зможе продовжити обчислення. Враховуючи те, що зміна процесу, порівняно з арифметичними командами або семафорними операціями є дуже дорогою операцією і P_1 має швидко знову зайняти семафор для виконання критичної операції, процес P_1 буде вертатись у чергу на обслуговування семафором **S** якраз на цьому інтервалі часу після останньої активізації. Виникає "пробка" (lock thrashing), яка самостійно може ліквідуватися з великими труднощами. В цьому випадку більшість процесорного часу використовується на зміну процесів, яка не дає ефекту.

У сучасних операційних системах вдалося уникнути конвой-феномена завдяки таким засобам:

- зміна диспетчера, що базується на повідомленнях про зайнятий семафор з найвищою завантаженістю ("high traffic"-семафор);
- зміна стратегії розподілу ресурсів; замість FIFO використовуються цикли чекання "busy-wait" або при V-операції звільняються всі процеси, що очікують і мають знову виконати P-операцію (включити P в цикл while);
- генеральне уникнення ситуації "high traffic lock";
- зменшення кількості операцій, які потрібні для зміни процесу.

На закінчення розгляньмо деякі типові приклади застосування способів синхронізації. Для запису програм в цьому та наступному параграфах використовується мова Modula-P [Bräunl, Hinkel, von Puttkamer 86], яка є розширеним за рахунок концепції процесів варіантом мови Modula-2 і детально описується в п. 9.2. Тут мова прислуговує лише для ілюстрації концепцій синхронізації.

Проблема "виробник-користувач"

Треба синхронізувати два процеси, які обмінюються даними через загальну буферну область пам'яті. Перший процес генерує дані і записує їх у буфер, другий процес бере ці дані з буфера і обробляє їх далі. За допомогою семафорів виключаються такі можливі ситуації:

- якщо процес-виробник (генератор) більш швидкий, ніж користувач, то на місце ще не використаних даних можуть потрапити нові дані, які знищать частину інформації;
- якщо процес-користувач більш швидкий, ніж процес-генератор, то можливе багаторазове зчитування одних і тих самих даних.

Для керування буферною пам'яттю в цьому прикладі використовується два булівських семафори. Один з них показує, чи буфер пустий, а другий – чи буфер вже заповнений даними. Текст програми має такий вигляд:

```
Декларування та ініціалізація:
var leer : semaphore [1]; (*пустий*)
    voll : semaphore [0]; (*повний*)
```

генератор	*користувач*
process Erzeuger;	**process**Verbraucher;
begin	**begin**
loop	**loop**
<генеруй дані>	**P** (voll);
P (leer);	*<звільни буфер>*
<заповни буфер>	**V** (leer);
V (voll);	*<обробляй дані>*
end;	**end**;
end process Erzeuger.	**end process** Verbraucher.

Спочатку ініціалізацією задаються значення 1 семафору leer (пустий) та 0 семафору voll (повний), тобто буфер переводиться в стан leer (пустий). Процес-виробник генерує нові дані, потім страхує себе P-операцією на семафорі leer, тобто впевнюється, що буфер пустий (або в процесі роботи чекає, поки буфер стане пустим), заповнює буфер своїми даними (це операція *<критичного розділу>*) і наприкінці сигналізує V-операцією на семафорі voll, що користувач, що очікує, вже може читати дані з буфера. Користувач зі свого боку перевіряє, чи буфер вже заповнений або чекає заповнення (P-операція на семафорі voll), потім читає дані з буфера (*<критичний розділ>*), виконує наприкінці V-операцію на семафорі leer (тим самим сигналізує процесу-виробнику, що він може записувати

нові дані в буфер) і тільки після цього використовує одержані дані.

На рис.7.2 проблема синхронізації пояснюється за допомогою простої мережі Петрі. Кожний з двох процесів являє собою один контур, що відповідає нескінченному циклу loop у програмі. **P**-операція на одному з двох семафорів leer або voll зображається вхідною дугою переходу, **V**-операція – вихідною дугою. Якщо, наприклад, перемикається верхній перехід в процесі-генераторі, то зникає марка в семафорній вершині leer і блокує цей процес в наступному робочому циклі (можливо інший, тут не показаний процес-генератор) до моменту, поки користувач виконує **V**-операцію. Це відповідає на рис.7.2 перемиканню верхнього переходу користувача, який займає вихідну дугу на вершині leer і тим самим видає знову марку.

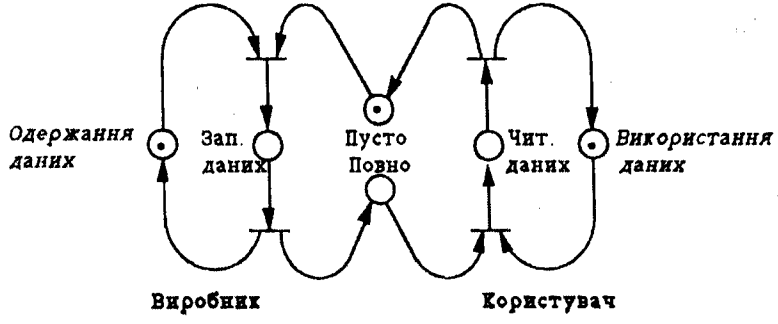


Рис.7.2. Мережа Петрі для проблеми "виробник-користувач"

Проблема обмеженого буфера

Хоча в розглянутому вище прикладі процеси "виробник" і "користувач" могли працювати досить незалежно і паралельно, все ж буферна область, що має одне місце для даних (воно або вільне або зайняте), суттєво обмежує можливу паралельність: наприклад, цілком імовірно, що як під час генерування, так і обробки даних будуть потрібні різні інтервали часу на обчислювальні операції. Якщо обробка йде з одним буфером, то ці інтервали складаються в невигідний спосіб. Зарадити цій проблемі можна, використовуючи розширену (або, як і раніше, обмежену) буферну область. У прикладі, що подається далі, організовано керування буфером, що має n буферних місць пам'яті. Тут використовується один булівський семафор для захисту *<критичного розділу>* і два лічильних семафори для керування показником завантаження буфера.

Декларування та ініціалізація:

```
var kritisch : semaphore [1]; (*критичний*)
    frei     : semaphore [n]; (*вільний, n буферних місць*)
    belegt   : semaphore [0]; (*зайнятий*)
```

виробник	*користувач*
process Erzeuger;	**process** Verbraucher;
begin	**begin**
loop	**loop**
<генерування даних>	**P** (belegt); (*зайнятий*)
P(frei);(*вільний*)	**P** (kritisch); (*критичний*)
P(kritisch);(*критичний*)	*<читати із буфера>*
<записувати в буфер>	**V** (kritisch);
V (kritisch);	**V** (frei);
V (belegt);	*<обробка даних>*
end;	**end**;
end process Erzeuger;	**end process** Verbraucher;

Під час ініціалізації семафор frei (вільний) одержує стартове значення n, а семафор belegt (зайнятий) - стартове значення 0, тобто буфер має n вільних місць для даних. Семафор kritisch (критичний) ініціалізується значенням 1; це гарантує, що тільки один процес може ввійти в *<критичний розділ>*. В програмному розділі обох процесів операції запису в буфер та читання з буфера замкнуті **P**- та **V**-дужками з семафором для *<критичного розділу>*. Це необхідно, бо на противагу раніше розглянутому прикладу буферна область може мати *одночасно як вільні, так і* зайняті місця. Як і раніше, процес-виробник виконує спочатку **P**-операцію на семафорі frei, записує дані в буфер і

виконує наприкінці V-операцію на семафорі belegt. У такий самий спосіб процес-користувач починає P-операцією на семафорі belegt (тільки після того, як у буфері є дані) і виконує наприкінці V-операцію на семафорі frei (завдяки зчитуванню елемента даних із буфера з'являється нове вільне місце в пам'яті). Значення при цьому легко змінилося, бо тепер n місць в пам'яті керуються двома загальними семафорами замість керування одним місцем у пам'яті двома булівськими семафорами. В станах frei (вільний) та belegt (зайнятий) вільні та зайняті місця в пам'яті рахуються відповідно в прямому та зворотному порядку, причому кожне з них враховується після того, як над ними виконана P- або V-операція.

Цей механізм синхронізації пояснюється на рис. 7.3 за допомогою розширеної мережі Петрі. Вона виникає з мережі, що показана на рис.7.2, якщо замінити обидві булівські семафорні вершини загальними семафорами, які в розширеній мережі Петрі зображаються вершиною з будь-якою кількістю маркувань.

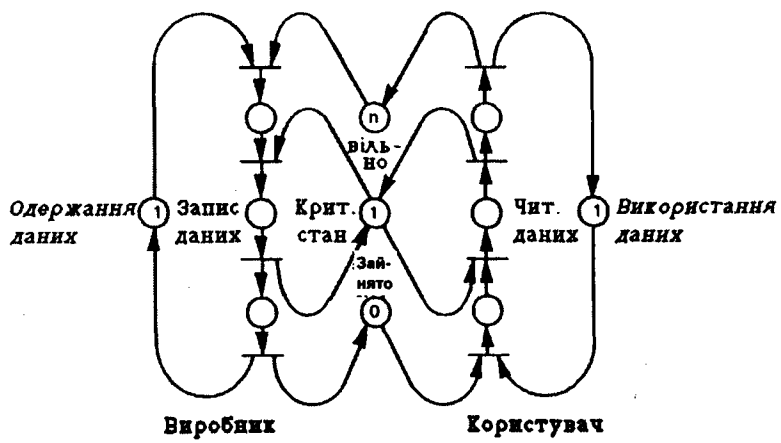


Рис.7.3. Розширена мережа Петрі для проблеми обмеженого буфера

Крім цього, вводиться вершина-семафор kritisch (критичний). Цей булівський семафор використовується для синхронізації багатьох процесів-виробників та користувачів. Вершина belegt (зайнято) дає кількість зайнятих в даний момент буферних місць, а вершина frei (звільнено) – кількість вільних буферних місць.

Проблема запису-зчитування

Ця проблема виникає, як правило, тоді, коли одна група процесів потребує деякого операційного ресурсу монопольно, а іншій групі процесів надається можливість одночасного паралельного його використання. Така організація роботи процесів має місце, наприклад, тоді, коли декілька процесів хотіли б мати доступ до загальної області пам'яті, причому, одні з них – в режимі запису, а інші – в режимі зчитування. В той час як в режимі зчитування одних і тих самих даних можна допустити одночасну роботу багатьох процесів, режим запису може бути гарантованим лише одному процесу. Під час запису забороняється доступ всім читачам, бо можуть виникнути некоректні дані (див. рис. 8.1). За тривіального рішення з одним семафором активним може бути тільки один процес-читач, тобто втрачається можливе розпаралелювання. Наведене тут рішення так званої проблеми запису-зчитування [Courtois, Heymans, Parnas 71] допускає максимально можливу паралельність.

Кожен процес-читач має перед доступом до даних перевірити, чи не є він першим з процесів, які б хотіли мати доступ у загальну для всіх процесів область даних. У такому випадку змінна readcount (лічильник читачів) дорівнює нулю і семафор r_w задіяний для читача. Наступні процеси-читачі звільняють лічильник readcount, але не виконують семафорних операцій. Якщо тепер один процес-записувач захотів би мати доступ до загальної області даних, то він блокується під час виклику P(r_w). У міру того, як процеси-читачі залишають критичну область, вони зменшують число в readcount-лічильнику. Коли останній процес-читач виконує V-операцію на семафорі r_w, процес-

записувач, що очікує, активізується, а всі інші процеси-читачі і записувачі мають чекати до моменту, коли закінчиться монопольне використання ресурсу процесом-записувачем. Програма має такий вигляд:

Декларування та ініціалізація:

```
var count :semaphore [1] ;
    r_w   :semaphore [1];   (*один записувач або багато читачів*)
    readcount :INTEGER;     (*лічильник читачів*)
      Ініціалізація:
 readcount := 0;
```

```
process Leser; (*читач*)
begin
  loop
    P (count);
    if readcount=0 then P(r_w)
    end;
    readcount := readcount+1;
    V (count);
    <читати з буфера>
    P (count);
    readcount := readcount-1;
    if readcount=0 then V(r_w)
    end;
    V (count);
    <використання даних>
  end; (*loop*)
end process Leser;
```

```
process Schreiber; (*записувач*)
  begin
    loop
      <генерування даних>
      P (r_w);
      <записи в буфер>
      V (r_w);
    end; (*loop*)
end process Schreiber;
```

Тут операції-лічильники та умовні операції-семафори утворюють в процесах-читачах два додаткових критичних розділи, які мають захищатись окремим семафором (count). При такому рішенні цілком можлива ситуація, коли виникає "голод" на записування при наявності значної кількості процесів-читачів: якщо постійно зростає кількість нових процесів-читачів, що

отримують доступ в *<критичний розділ>*, то процеси-записувачі можуть не потрапити в чергу. Така можливість виключається при більш складному рішенні тим, що при появі в черзі першого процесу-записувача ні один з процесів-читачів не одержує доступу в *<критичний розділ>*.

На рис. 7.4 це рішення представлене у вигляді розширеної мережі Петрі, причому тут показано тільки один процес-читач і один процес-записувач. Для наочності зв'язки з центральною семафорною вершиною r_w виконано товстими лініями, а сторонні зв'язки між вершинами та переходами – подвійними стрілками (або стрілкою з інверсією, запереченням). Якщо є декілька процесів-читачів, то вони звертаються до вершин count, read_count та r_w; відповідно всі процеси-читачі спільно мають одну і ту саму вершину r_w!

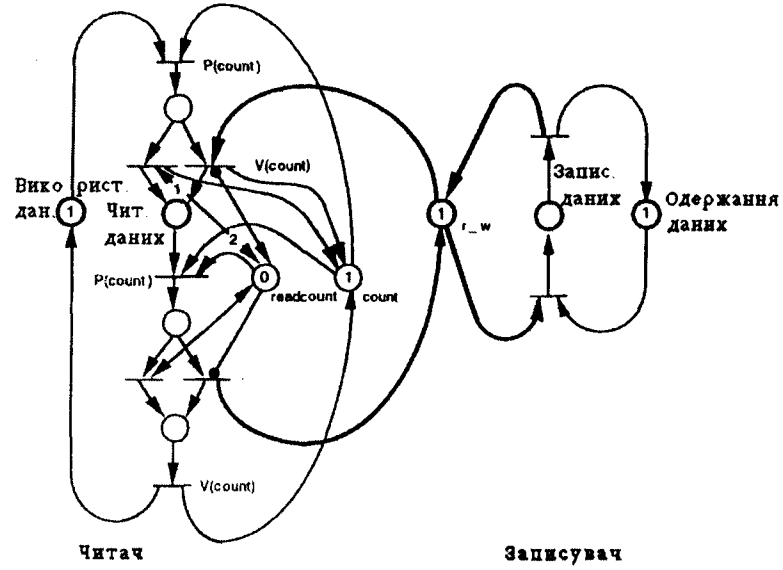


Рис. 7.4. Розширення мережі Петрі для проблеми "Читання–записування"

Після того як перший процес-читач зайняв семафор count (виведення марки), проводиться аналіз на визначення

випадку, який має місце, за допомогою дуги-заперечення, тобто з'ясовується, чи змінна read_count дорівнює нулю (процес-читач є першим), чи більша за нуль (на даний момент вже декілька процесів-читачів користується загальною областю даних). Тільки якщо readcount=0, під час перемикання відповідного переходу виконується також P-операція на центральному семафорі r_w (виведення марки). В обох випадках під час перемикання переходу змінна read_count збільшується на 1 (якщо read_count>0, то спочатку віднімається вага дуги 1, а потім додається 2, тобто в результаті додається приріст на 1). Після цього процес-читач може виконувати свої операції над загальною областю даних. Якщо йому треба залишити цю область, то він має знову зайняти семафор count для критичного розділу і зменшити на 1 число в лічильнику read_count. Тільки тоді, коли нарешті read_count=0, тобто з загальною областю даних відпрацював останній активний процес-читач, проводиться V-операція на семафорі r_w (генерується марка в r_w), а семафор count звільняється (генерується марка в count) тільки після виконання цих двох умов.

Поведінка процеса-записувача порівняно з описаним є дуже простою. Перед кожним доступом гаситься марка семафорної вершини r_w за допомогою вхідної дуги (P-операція), а після кожного доступу вона знову генерується вихідною дугою (V-операція).

У системах банків даних поряд з розглянутим тут семафором виключного типу ("exclusive" semaphore) застосовується також семафор загального типу ("shared" semaphore) [Gray, Reuter 92]. Цей семафор може бути або зайнятим одночасно багатьма процесами в режимі "загальний" (наприклад, багатьма процесами-читачами), або лише одним процесом в режимі "виключний" (наприклад, одним записувачем). Це означає, що проблема запису-читання може вирішуватися загальним семафором тривіально. Отже, це універсальніша модель семафора, яку можна імплементувати за допомогою виключного семафора подібно розглянутому вище.

Закінчуючи цей розділ, треба підкреслити, що семафори, будучи простим засобом синхронізації процесів, є і основним джерелом помилок. Воно охоплює всі помилки– від часто "забуваних" P- та V-операцій до реакції на виняткові ситуації (exeption handling). Якщо в одному з процесів після зайняття семафора (P-операція) виникає помилка (exception) і він з цієї причини припиняє своє виконання (термінується), то за цих умов може блокуватися вся система взаємодії процесів: не може відбутися звільнення семафора. Лише в операційних системах, що мають дорогі засоби "функціонального відновлення" ("functional recovery"), може бути досягнута певна толерантність до помилок за рахунок спроб звільнити семафори, що зайняті "термінованими" процесами. Семафори належать до інструментарію системного програмування і мають бути замінені в прикладних програмах по можливості засобами більш високого рівня, такими, наприклад, як описувані нижче монітори.

7.4. Монітори

Монітор – це керуюча програма, високорівневий механізм взаємодії та синхронізації процесів, що забезпечує організацію доступу до нероздільних ресурсів. Монітор складається з процедур доступу до ресурсів, кожна з яких може бути викликана тільки з одного процесу одночасно. Процес, який робить спробу звернутися до процедури монітору, коли той обслуговує інший процес, ставиться в чергу і переходить в стан очікування.

Монітори запропоновані в 1974–1975 роках як нова концепція синхронізації процесів [Hoare 74], [Brinch Hansen 75]. На відміну від розглянутих семафорів, монітори є типами даних вищого рівня абстракції. Кожний монітор охоплює як захищувані дані, так і підпорядковані їм механізми доступу та синхронізації, які називаються

"входами" ("Entrys") та "умовами" ("Conditions"). Застосування монітора можна проілюструвати так:

Процес P_1	Процес P_2
....	....
Puffer:Daten Schreiben(x)	Puffer:Daten Lesen(x)
(*запис даних*)	(*читання даних*)
....	....

Виклики взаємовиключають один одного, вони синхронізуються.

Практичне використання моніторів значно простіше, ніж семафорів. Процеси, що мають синхронізуватися, викликають моніторні входи з відповідними параметрами за допомогою спеціальних процедур виклику. Тут не можуть виникнути проблеми типу "забутих P- або V-операцій". Як показує наступний приклад, розробка моніторних входів (викликів) разом з відповідними умовами – це не проста справа. Треба розпорядитися буферним пакетом (стеком), показаним на рис.7.5.

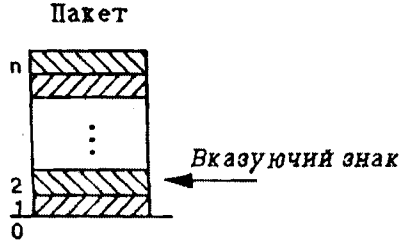


Рис. 7.5. Пакет буфера

У моніторі передусім визначаються моніторні дані, які мають такі властивості:

- статичність, тобто моніторні дані зберігають свої значення між двома викликами входів монітора (на противагу даним процедур);
- локальність, тобто доступ до моніторних даних забезпечується виключно через входи монітора (аналогічно локальним даним процедур).

Далі йдуть декларування входів ("монітор-процедури"). Це єдині конструктиви, що видимі ззовні і можуть викликатися процесами за допомогою числових параметрів або параметрів результатів обчислень (VAR). У зв'язку з тим, що моніторні входи ведуть до загальних даних, вони виключають один одного. Потрібна для цього синхронізація виконується автоматично операційною системою. Це означає, що під час виконання одним процесом моніторного входу всі інші процеси, що бажають повернутися туди ж, мусять чекати.

У кінці монітора розташована ініціалізуюча частина, яка виконується тільки один раз під час старту всієї програмної системи ще до старту процесу, тобто до ініціалізації головної програми.

Умови (Conditions) – це черги очікування, аналогічні чергам семафорів, але вони не мають лічильників. На умовах визначено три операції, які детальніше пояснюються в реалізуючій частині:

- **wait** (Cond)

Виконуваний процес блокує себе сам і чекає, поки деякий інший процес виконує signal-операцію над цією умовою.

- **signal** (Cond)

В умові Cond усі процеси, що очікують, реактивуються і заявляють про свою потребу доступу до монітора (інший варіант звільняє лише один процес, а саме наступний в послідовності процесів).

- **status** (Cond)

Ця функція видає кількість процесів, які очікують на виконання умови.

Розгляньмо програму-приклад на мові Modula-P, в якій реалізується керування буферним стеком.

```
monitor Puffer;
var Stapel: array [1..n] of Datensatz; (*стек даних*)
    Zeiger: 0..n  (*покажчик*)
    frei, belegt: condition; (*вільно, зайнято*)
```

```
entry DatenSchreiben (a: Datensatz); (*вхід запису даних*)
begin
  while Zeiger=n do (*стек повний*)
     wait (frei)
  end;
  inc(Zeiger)
  Stapel[Zeiger] := a;
  if Zeiger=1 then signal(belegt) end;
end DatenSchreiben;

entry DatenLesen (var a:Datensatz);
begin
  while Zeiger=0 do (*стек пустий*)
    wait (belegt)
  end;
  a := Stapel[Zeiger];
  dec(Zeiger);
  if Zeiger=n-1 then signal(frei) end;
end DatenLesen;

begin
  Zeiger := 0;  (*ініціалізація монітора*)
end monitor Puffer;
```

У цьому прикладі декларуються дві умови, які називаються "вільно" (frei) і "зайнято" (belegt). У чергу чекання стають процеси, які чекають на появу відповідної умови. Для входу запису даних (entry DatenSchreiben) забезпечено такий порядок: якщо умова zeiger=n виконана, то стек повний і процес повинен чекати, поки звільниться місце в буфері. Це чекання має відбуватися в циклі while, інакше могло б трапитись так, що при наявності багатьох процесів, що очікують в черзі, міг би з'явитись інший процес і використати знову звільнене місце (операція signal реактивує всі процеси, що чекають в Condition).

Наприкінці йдуть операції над даними монітора (запис масиву даних) і останнім подається сигнал про умову belegt (зайнято), яка активізує процеси, що чекають на вхід до критичного розділу DatenLesen (читання даних).

Вхід (entry) DatenLesen побудований аналогічно. Тут треба чекати на настання умови belegt (зайнято), коли можуть читатися дані із стека, а останнім подається сигнал про умову frei (вільно), яка активізує процеси-записувачі, що очікують. Операції signal потребують досить багато часу, тому вони включені в if-запити і виконуються лише тоді, коли є реальна можливість того, що на цю умову чекають процеси (наприклад, умова Zeiger=1 для сигналу на стан belegt в entry DatenSchreiben означає, що перед цим покажчик був Zeiger=0 і буфер був повністю пустим).

Реалізація

Монітори імплементуються в загальному випадку з використанням семафорів за шість етапів [Nehmer 85]:

1. Декларування булівського семафора для кожного потрібного монітора

```
var Msema: semaphore[1].
```

2. Трансформація входів (entrys) у процедури, причому кожна з процедур закривається дужками із **P**- і **V**-операцій над моніторним семафором. Це гарантує, що тільки один процес може викликати вхід монітора:

```
procedure xyz(...)
begin
  P(MSema);
  <інструкції>
  V(MSema);
end xyz.
```

3. Оформлення ініціалізації монітора у вигляді процедури і розташування відповідного виклику на початку головної програми.

4. Імплементація операції wait. Виконуваний процес заводиться в умовну чергу очікування (Condition-черга) і блокується. Після цього знову звільняється моніторний семафор (це для того, щоб якийсь інший процес міг дістати

доступ до монітора) і наступний процес, що має стан "готовий", вводиться в процесор для виконання. Паралельна константа actproc вказує номер процеса, який є активним у даний момент:

```
procedure wait (Cond: condition; Msema: semaphore);
begin
  append (Cond, actproc); (*ProcID ввести в Condition-чергу*)
  block (actproc);        (*ProcID ввести в список блокуванньо*)
  V (Msema);              (*моніторний семафор знову звільнити*)
  assign;                 (*ввести наступний готовий процес*)
end wait;
```

5. Імплементація сигнальної операції:

а) варіант, який виводить з черги тільки один процес, що очікує (рис. 7.6): wait-операція тут завжди має бути в середині if-циклу, щоб не чекати на умову, що вже з'являється. Наступний процес, що чекає в черзі Condition, виводиться з неї, і для цього процесу виконується **P**-операція над моніторним семафором (але не для процесу, який виконує сигнальну операцію!)

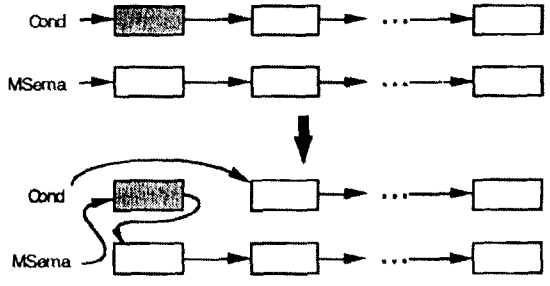


Рис. 7.6. Операція "Сигнал" (wait), варіант (а)

```
procedure signal (Cond: condition; Msema: semaphore);
var NewProc: Proc_ID;
begin
  if status(Cond)>0 then
    getfirst (Cond, NewProc);
    putfirst (MSema.L, NewProc);
    dec (MSema.Value)
  end
end signal;
```

б) варіант, який виводить з черги всі процеси, що очікують (наприклад, в Modula-P) (рис. 7.7)

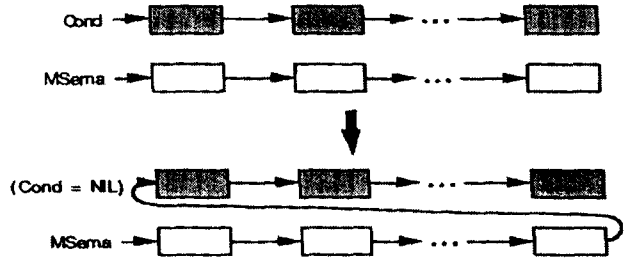


Рис. 7.7. Операція "Сигнал" (wait), варіант (б)

Wait-операція тут завжди має бути всередині while-циклу, бо інакше інший процес, який щойно дістав дозвіл на виконання, може знову зробити несприятливою умову звільнення з черги:

```
procedure signal (Cond: condition; Msema: semaphore);
var NewProc: Proc_ID;
  begin (*тут не треба робити запит про статус*)
    MSema.Value := MSema.Value - status (Cond);
    append (MSema.L, Cond);
    (*навішування списків один на одного*)
    Cond := nil;
  end signal;
```

6. Імплементація status-операції.

Ця функція дає інформацію про довжину черги очікування Condition:

```
procedure status (Cond: condition): CARDINAL;
begin
  return length (Cond);
end status;
```

7.5. Повідомлення та дистанційний виклик (Remote-Procedure-Call)

Усі розглянуті механізми синхронізації можуть бути використані лише тоді, коли всі процесори, на яких виконуються процеси, взаємодіють через загальну пам'ять. В розподілених системах єдина можливість синхронізації процесів або обміну даними – це обмін посланнями чи повідомленнями. З другого боку, концепції обміну повідомленнями та побудований на цій основі механізм дистанційного виклику (Remote_Procedure_Call) годяться для застосування і в системах із загальною пам'яттю. Цей метод, будучи дуже комфортабельним, все ж таки потребує більших витрат обчислювальних ресурсів на розв'язування задач керування. Процеси, які виконують процедуру дистанційного виклику, називаються далі "клієнтами" (client; вони виступають, так би мовити, користувачами ресурсів інших процесів), а процеси, що викликані, називаються "серверами" (ці процеси пропонують ресурси для інших процесів).

На рис.7.8 показано просту модель виконання заявок за допомогою обміну повідомленнями між процесами [Nehmer 85]. У зв'язку з тим, що кожен процес може бути як клієнтом, так і сервером, кожному процесу підпорядковуються два різних буфери для інформації: "A" для заявок і "R" для зворотних відповідей.

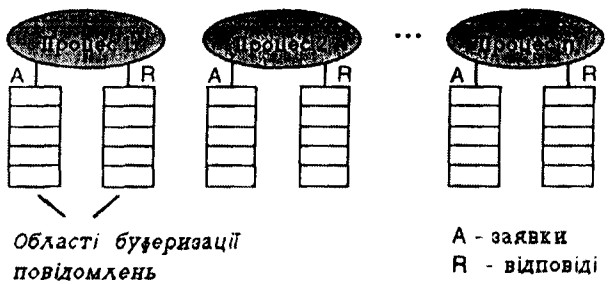


Рис. 7.8. Процеси з буферизацією повідомлень

Для обміну повідомленнями визначено такі операції:

- Відправка замовлення

Send_A (приймач, повідомлення);

- Приймання деякого замовлення

Receive_A (var відправник, var повідомлення);

- Відправлення деякої відповіді

Send_R (приймач, повідомлення);

- Приймання деякої відповіді

Reseive_R (var відправник, var повідомлення);

Природно, що можна було б обійтися всього двома операціями для відправлення та приймання повідомлень. Однак вибрані тут варіанти дають змогу відрізняти одержані замовлення і відповіді, та вибирати більш гнучку послідовність обробки, щоб не міг виникнути випадок, коли процес, що очікує, наприклад на відповідь, блокується новим замовленням, що надійшло у цей час.

Застосування

Клієнт-процес P_c відправляє своє замовлення сервер-процесу P_s. Зараз же після цього він може виконувати наступні завдання впродовж часу очікування на появу відповіді сервера. Сервер обробляє запити різних клієнтів-процесів у безкінечному циклі. Запит читається, виконується і результат відправляється замовникові:

P_c	P_s
... Send_A (*запит на_сервер*); ... Receive_R (*від сервера, результат*)	**loop** Receive_A (*клієнт,запит*) <*запит виконати*> Send_R (*клієнт,результат*); **end**

Реалізація

У паралельних ЕОМ із загальною пам'яттю концепція повідомлень може бути реалізована за допомогою монітора. У паралельних ЕОМ, що не мають загальної пам'яті, потрібно мати допоміжний децентралізований пристрій з протоколами повідомлень і методами маршрутизації, про які тут не будемо вести мову.

Розгляньмо короткий ескіз імплементації керування повідомленнями в межах обчислювального вузла (для розподілених систем) або ж для систем з як завгодно великою кількістю процесорів і загальною пам'яттю.

Як показано на рис.7.9, усі повідомлення мають керуватись в деякій глобальній області, що розподіляється динамічно. Паралельна константа actproc локально постачає для кожного процеса його номер.

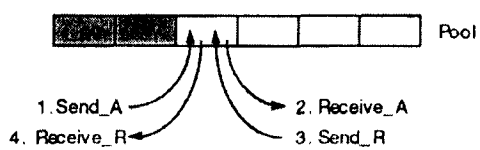


Рис. 7.9. Pool повідомлень

За Немером [Nehmer 85] ескіз імплементації має вигляд:

```
type PoolElem = record
        free: BOOLEAN;
      from: 1..anzahlProcs; (кількість процесів)
        info: Nachricht; (повідомлення)
     end;
Schlange = record
              contents: 1..max;
               next     : pointer to Schlange (черга)
            end;

monitor  Nachrichten; (повідомлення)
var Pool : array  [1..max] of PoolElem;
    (*глобальна область повідомлень*)
```



```
    pfree : CONDITION;
    (*черга чекання, якщо Pool постійно зайнято*)
    Afull, Rfull : array  [1..anzahlProcs] of  CONDITION;
    (*черга для кожного процесу відповідно до повідомлення*)
    queueA,queueR: array  0[1.. anzahlProcs] of  Schlange;
    (*локальний Pool повідомлень для кожного процесу *)
entry  Send_A (nach: 1.. anzahlProcs; a: Nachricht);
  var  id: 1..max;
  begin
    while not  GetFreeElem(id)  do wait  (pfree) end;
    with  pool [id]  do
       free :=  false;
       from := actproc;
       info := a;
    end;
     append (queueA[nach],id);
    (*внести номер позиції в чергу замовлень*)
    signal  (Afull [nach]);
  end  Send_A;

entry Receive_A (var von:1..anzahlProcs; var a: Nachricht);
 var  id: 1..max;
 begin
   while  empty(queueA[actproc])  do wait  (Afull[actproc])  end ;
        id    := head(queueA[actproc]);
        von := pool[id].from;
        a     := pool[id].info; (* pool [id] ще не звільнено*)
 end  Receive_A;

entry  Send_R (nach: 1.. anzahlProcs;ergebnis: Nachricht );
                    (* ergebnis - результат *)
var  id: 1..max;
begin
        id := head(queueA[actproc]);
        tail (queueA[actproc]);
        (* вилучається перший елемент (head) черги *)
        pool[id].from := actproc;
        pool[id].info := ergebnis;
        append(queueR[nach],id);
        (* внести номер позиції в чергу відповідей *)
        signal (Rfull[nach])
 end  Send_R;
```

```
entry Receive_R ( var von: 1.. anzahlProcs; var erg: Nachricht);
var id: 1..max;
begin
   while empty(queueR[actproc]) do wait (Rfull[actproc]) end ;
    id := head(queueR[actproc]);
    tail(queueR[actproc]);(*вилучається перший елемент (head) черги*)
    with pool[id] do
       von := from;
       erg := info;
       free := true ; (* звільнення Pool- елемента *)
   end ;
   signal (pfree); (* є вільний Pool- елемент *)
end Receive_R;
```

Ми тут розглянули MIMD-системи без загальної пам'яті дуже коротко. Більш детальну інформацію щодо розподілених обчислювальних систем можна знайти в працях [Mullender 89], [Coulouris, Dollimore 88].

8. ПРОБЛЕМИ АСИНХРОННОЇ ПАРАЛЕЛЬНОСТІ

Як було показано в попередніх главах, асинхронне паралельне програмування є досить складним і таким, що дуже сприйнятливе до помилок. Найчастійше помилки пов'язані із "забуванням" P- та V-операцій або застосуванням неправильного семафора. Вони призводять до серйозних наслідків на стадії виконання програм. Із застосуванням моніторів існує небезпека зробити помилку вживання умовних змінних та операцій wait і signal. Аналіз показує, що помилки програмування синхронізації процесів можуть спричинити такі проблеми:

а) несумісні дані;
б) взаємоблокування (DeadLock/LiveLock)

У наступних двох параграфах розглядаються причини виникнення цих проблем, іхні наслідки, можливості передбачення їх та запобігання їм. У третьому, заключному, параграфі дискутується проблема балансування завантаження, вирішивши яку, можна досягти ефективного використання процесорів.

8.1. Несумісні дані

Після виконання паралельної операції числове значення або співвідношення між даними стає несумісним якраз тоді, коли воно не дорівнює числу, яке могли б дістати в результаті послідовної роботи ресурсів. Без достатніх паралельних механізмів контролю при виконанні паралельних процесів досить легко виникають дані, що містять у собі помилки. Як показано в [Date 86], слід відрізняти тут такі проблеми:

- втрачена модифікація даних (*lost update problem*);
- несумісний аналіз (*inconsistent analysis problem*);
- не підтверджена залежність (*uncommitted dependency problem*).

Втрачена модифікація даних

Цю проблему пояснимо на такому прикладі: хай заробіток пана Мюллера становить 1000 DM. Процес 1 має (поряд з іншим) збільшити заробіток пана Мюллера на 50 DM. Процес 2, що виконується паралельно з першим процесом, має збільшити заробіток пана Мюллера на 10%. Залежно від порядку виконання або об'єднання обох процесів при їхньому виконанні, дістанемо різні результати. Вони, природно, не можуть бути всі однаково коректними.

Приклад на рис.8.1 показує всі чотири можливих результати. Взагалі всі результати можуть бути знайдені шляхом систематичного випробування всіх можливих сумісних послідовностей виконання процесів.

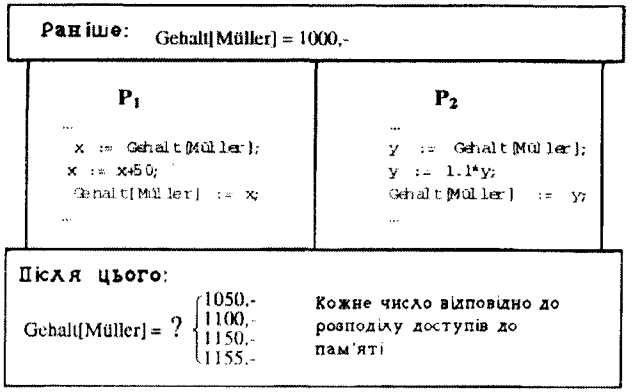


Рис.8.1. Втрачена модифікація даних

Число 1050, наприклад, дістали в такий спосіб: Обидва процеси виконуються спочатку одночасно. В результаті перших операцій присвоєння змінні **X** і **Y** набувають початкових значень 1000. Далі процес P_2 виконує свої дві команди, що залишились, і встановлює панові Мюллеру заробіток 1.1*1000 = 1100 DM. Однак далі процес P_1 виконує свої дві команди,що залишились і встановлює остаточно заробіток 1000+50 = 1050 DM. Число, одержане раніше процесом P_2, не прийняте до уваги і не переписане.

Відповідно до змістовної мети обох транзакцій (які тут не описуються) маємо точно визначену послідовність їх виконання, наприклад P_1 виконується повністю перед P_2. Таке послідовне виконання дає коректний результат (а саме 1155 DM). Всі інші числа неправильні, вони з'являються внаслідок помилкової послідовності виконання процесів. Ці помилки розглядаються як "залежні від часу помилки". У зв'язку з тим, що ці помилки залежать від деякої кількості

раметрів, що впливає на процеси не безпосередньо ількість наявних процесорів, завантаження системи тощо) они особливо підступні. Помилки, залежні від часу, як равило, не можуть репродуктуватись і, таким чином, не ожуть бути знайдені за допомогою систематичних тестів!

есумісний аналіз

Типовий приклад щодо цього – банківський переказ рошей. Сума обох рахунків, що беруть участь в операції, о і після транзакції (операції переказу грошей з одного рахунку на інший) ідентична. Але деякий інший процес, який читає (аналізує) актуальний стан рахунків під час проведення операції переказу, може прийти до неправильних результатів.

Як показано на рис. 8.2, у той час, коли процес P_1 виконує транзакцію переказу, деякий час співвідношення між двома даними (стан розрахунків) не відповідає дійсності. Процес P_2, що аналізує, одержує помилкові дані, якщо він читає якраз у цей момент актуальні значення одного або обох елементів даних (стан рахунків).

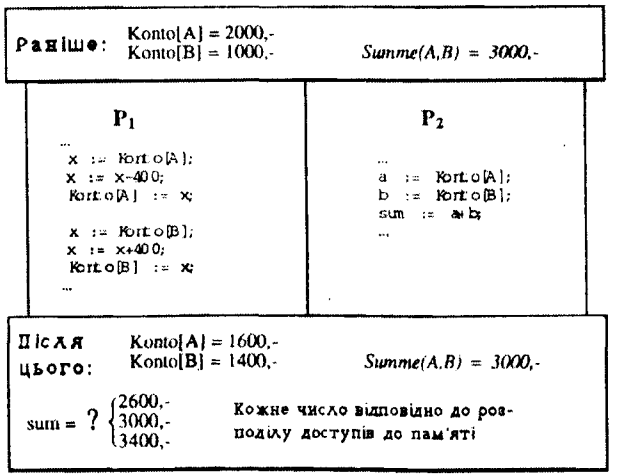


Рис.8.2. Несумісний аналіз

Не підтверджена залежність

Можлива залежність від непідтверджених (*uncommitted*) транзакцій – це типова проблема в банках даних. У зв'язку з тим, що транзакція може завершуватись успішно або ні, зміни глобальних даних, виконувані транзакційно перед її успішним закінченням, завжди дійсні лише з застереженням. Якщо транзакція A вносить зміни в глобальні дані, транзакція B ці дані читає, а транзакція A наприкінці не завершується успішно (фіаско транзакції), то транзакція B використовує некоректні дані.

У системах банків даних транзакції імплементуються як "атомарні операції" (це зовсім не проста задача, як показують розглянуті тут випадки [Date 86]). Транзакція може бути або успішною, або зазнає невдачі. Успішна транзакція оброблюється повністю (*commit*), а транзакція, що супроводжується помилками, має бути повністю повернута у вихідне положення (*rollback*) для перезапуску. У випадку, коли до появи помилки вже виконані деякі операції в банку даних, їх треба зворотним методом повернути у стан, що відповідає стану справ до виконання цієї транзакції. Дотримання концепції транзакцій запобігає появі несумісних даних.

8.2. Блокування

Є два типи блокувань: DeadLocks і LiveLocks. DeadLock описує стан, в якому всі процеси блоковані і не можуть самі вивести себе із стану блокування. У випадку LiveLock йдеться про блокування активних процесів, тобто тих процесів, які хоч і не є в стані "блоковано", але виконують при цьому, наприклад, операції очікування і все ж таки не можуть вийти із блокування.

Визначення DeadLock (взаємне блокування)

> Група процесів чекає на появу умови, яка може бути створена тільки процесами цієї групи (взаємозалежність)

Взаємні блокування (DeadLock) з'являються, наприклад, тоді, коли всі процеси блоковані в черзі очікування семафора або умови. Наступний приклад показує виникнення взаємоблокувань у разі застосування семафорів. Кожний з двох процесів використовує системні ресурси, наприклад термінал (TE) і друкарську машинку (DR), але запити виконуються у формі p-операцій в різній послідовності. У той час, як P_1 запитує спочатку термінал, а потім друкарську машинку, P_2 намагається використати ці системні ресурси в зворотному порядку. Незважаючи на те що кожний процес веде себе коректно з точки зору його особистих функцій, взаємодія двох процесів може привести до взаємоблокування (DeadLock). Це трапляється не завжди, а тільки тоді, коли P_1 зайняв термінал, а P_2 – друкарську машинку. Тоді P_1 чекає в своїй P-операції на друкарську машинку, яку зайняв P_2 і звільнить її лише тоді, коли він зможе зайняти термінал, який в свою чергу зайнятий процесом P_1. Тим самим обидва процеси опинились у взаємозалежності, із якої не зможуть звільнитись. Це означає, що настало взаємоблокування (DeadLock). На рис.8.3 показано небезпеку взаємоблокування у вигляді програми.

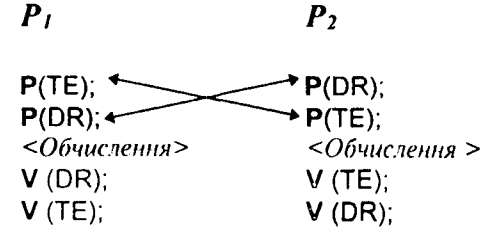


Рис. 8.3. Небезпека взаємоблокування

Передумови, за яких може з'явитись взаємоблокування [Coffman, Elphick, Soshani 71]:

1. Операційні засоби можуть використовуватися тільки на основі виключного права (ексклюзивно).

2. Процеси перебувають в режимі використання операційних засобів в той час, коли вони вже потребують інших, нових ресурсів.

3. Операційні засоби не можуть забиратися у процесів в примусовому порядку.

4. Існує циркулярний ланцюг процесів, так що кожний процес використовує операційний ресурс, який замовляється наступним процесом цього ланцюга.

На практиці існує ряд алгоритмів для розпізнавання взаємоблокувань та ліквідації їх. Вони будуються на основі вказаних чотирьох пунктів і спрямовані на те, щоб не виникало передумов появи взаємоблокувань.

Розгляньмо приклади того, як не допустити передумов блокувань.

• Уникнення передумови 3:

Якщо виникає взаємоблокування, то процеси треба звільнити від роботи з ресурсами, які були між ними розподілені. Для цього потрібні надійні методи розпізнавання фактів взаємоблокування і вводу в дію процесів, що повернуті в початковий стан при розблокуванні.

• Уникнення передумови 2:

Кожний процес має один раз замовити операційні ресурси, які йому потрібні. Цей метод передбачає ведення протоколу замовлень, якого мають дотримуватися всі процеси. Запит на декілька операційних ресурсів має реалізуватись як атомарна операція, тобто, наприклад, для критичного сегмента вона має замикатися через семафор. Якщо процес на момент виконання виявляє, що він потребує нових операційних засобів, то він має спершу звільнити якраз до цього моменту всі без винятку операційні засоби, що були зайняті, поки він заново зможе запросити одночасно всю сукупність операційних засобів, що тепер потребує.

Якщо всі процеси дотримуються цього протоколу замовлень, то в жоден момент не може виникнути блокування. Це потребує збільшеного об'єму управлінських операцій для зайняття та звільнення операційних ресурсів. Крім того, контроль правильності дотримання протоколу всіма процесами важко реалізувати, а операційні ресурси в цих умовах будуть зайняті помітно довше, ніж цього потребують процеси.

Інші методи уникнення взаємоблокувань наведено в книзі [Haberman 76].

8.3. Балансування завантаження

Ще одним великим колом проблем асинхронного програмування, щоправда не з такими різкими впливами на працездатність програм, як у вище розглянутих проблемах, є балансування завантаження процесорів. Невмілий розподіл процесів між процесорами може призвести до помітних втрат ефективності, яких саме при застосуванні паралельної обчислювальної системи хотілось би, природно, уникнути.

У випадку простої моделі планування (*scheduling*-модель) застосовується статичний розподіл процесів між процесорами, тобто на момент виконання не відбувається пересилки процесів, що вже розподілені, на інші процесори, що в даний момент менше завантажені. Як показано на

рис.8.4, цей метод за таких умов може призвести до великих втрат ефективності паралельних обчислювальних систем. Наприклад, на початок обчислювальних процедур дев'ять процесів рівномірно розподілені між трьома наявними процесорами. Під час виконання всі процеси другого і третього процесорів блокуються в черзі очікування, а всі процеси першого процесора залишаються активними. Потужність паралельної ЕОМ падає до можливостей одного процесора, тоді як у даній ситуації можна було б продовжити паралельне виконання процесів.

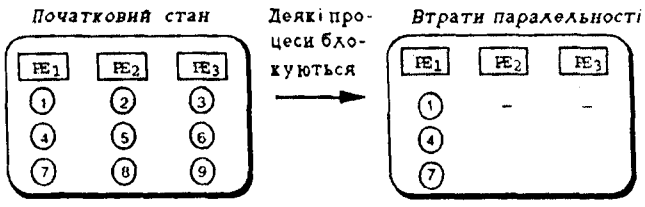


Рис.8.4. Статичне балансування завантаження процесорів

Щоб уникнути подібних втрат потужності, розроблено інші моделі планування процесів [Hwang, Deboot 89]. Вони забезпечують динамічний розподіл процесів між процесорами на момент їх виконання (*dynamic load balancing*) за допомогою перегрупування процесів, що були закріплені за певними процесорами (*process migration*) залежно від їхнього завантаження відносно деякого порогового числа (*thres hold*). Існують три принципово різних методи керування центральною операцією "міграція процесів".

1. Ініціатива адресата: процесори з малим завантаженням дають запит на обробку інших процесів. Цей метод ефективний у разі високого завантаження системи.

2. Ініціатива відправника: процесори з високим завантаженням роблять спробу віддати частину процесів, що їм підпорядковані. Цей метод ефективний у випадку невисокого завантаження системи.

3. Гібридний метод: перехід від ініціативи адресата до ініціативи відправника і навпаки залежно від глобального завантаження системи.

Переваги та недоліки методів балансування завантаження.

\+ Досягається більше завантаження процесорів і не втрачається можлива паралельність.

0 Запобігання циркуляційній "міграції процесу", тобто можливому тривалому пересиланню деякого процесу між процесорами, має бути реалізоване за рахунок відповідних паралельних алгоритмів і порогових чисел.

– Виникає помітно високий обсяг функцій керування завантаженням процесорів (а саме порівняно з установленням глобального системного завантаження).

– Перенесення деякого процесу з одного процесора на інший, менше завантажений процесор ("міграція процесів"). Це дорога операція і має застосовуватися тільки для процесів, що виконуються довгий час; але цю властивість процесу не можна встановити на момент початку його виконання без спеціальної допоміжної інформації.

– Усі методи балансування вступають в роботу надто пізно, а саме тоді, коли вже суттєво розладилося рівномірне завантаження процесорів. Деяке "передбачене балансування" неможливо реалізувати без допоміжної інформації про реальні витрати часу на виконання процесів.

– У разі повного паралельного системного завантаження будь-яке його балансування втрачає сенс: у цьому випадку можна керувати лише нестачею завантажень і загальний обсяг ресурсів для

балансування завантаження спричинить необгрунтовані витрати машинного часу.

9. МОВИ ПРОГРАМУВАННЯ ДЛЯ MIMD-СИСТЕМ

Розгляньмо передусім процедурні паралельні мови програмування для MIMD-систем. Непроцедурні паралельні мови (функціональні та логічні) наведено в гл. 17. Деякі мови подано коротко; найважливіші концепції обговорюються та пояснюються за допомогою прикладів програм. Порівняння публикацій про MIMD-мови програмування можна знайти в праці [Gehani, McGettrick 88].

9.1. Паралельний Паскаль

Автор: Per Brinch Hansen, 1975.

Паралельний Паскаль [Brinch Hansen 75,77] – це перша паралельна мова програмування, виконана як паралельне розширення послідовної мови програмування Паскаль, розробленої Віртом. Найважливійшими концепціями паралельності у цій мові є:

- введення концепції процесу (декларування process-типів і змінних, що їм належать);
- синхронізація паралельних процесів через монітори з умовними змінними (тут вони названі queue);
- зведення програмних елементів у класи (абстрактні типи даних).

Ми тут не будемо далі деталізувати паралельний Паскаль, це буде продовжено в параграфі 9.7 під час описування мови Modula-P, що є подальшим розвитком паралельного Паскаля на базі мови Modula-2.

9.2. Мова CSP (Communicating Sequential Process)

Автор: C.A.R. Hoare, 1978.

Через декілька років після розробки концепції монітора C.Hoare разом з B.Hansen запропонував свою паралельну мову CSP [Hoare 78,85]. Хоч і існує для цієї мови компілятор, все ж багато хто розглядає CSP не як мову програмування, а тільки як ескізний проект, тим більше на противагу мові OCCAM, що базується на CSP.

CSP – це досить криптична мова (неявна паралельність). Вона складається з деякої кількості процесів, кожний з яких виконується власне послідовно і при потребі обмінюється даними з іншими процесами. Основними принципами CSP є примітивні команди вводу-виводу для обміну повідомленнями, команди для паралельного виконання інструкцій і застосування захищених команд ("*guarded commands*", введені Dijkstra). У зв'язку з тим, що загальна синхронізація та взаємообмін між процесами виконуються в CSP за допомогою концепції повідомлень, тут нема потреби в семафорах або в моніторах.

Паралельні мовні конструкції:

[P1 P2]	Старт паралельних процесів
terminal? zahl	прийняти дані (zahl-число) від процесу terminal
drucker! zeile	послати дані (zeil-рядок) процесу drucker
[x=1 → m:=a x=2 → m:=b]	"*guarded command*" *{IF x=1 THEN m:=a ELSE* *IF x=2 THEN m:=b ELSE error}.*

Тут ідеться про інструкцію селекції, причому перед кожним оператором вибору ("case") обов'язково вводиться умова (guard), за допомогою якої програміст повинен впевнитись, що наступна інструкція виконується коректно:

```
*[x=1 → m:=a
  x=2 → m:=b]
```

"*repetitive command*" (повторювана команда, що позначається символом "*"); послідовність команд виконується ітеративно в умовному циклі за параметром guard.

Розгляньмо приклад CSP-програми, взятий із [Hoare 78]. В програмі реалізовано проблему обмеженого буфера (Bounded-Buffer)(рис. 9.1.) з десятьма буферними місцями пам'яті (тут не відбувається, правда, аналізу на переповнення змінних in та out) .

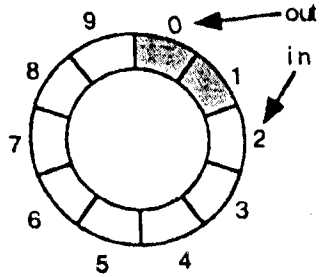


Рис. 9.1. Схема обмеженого буфера.

Програма для Bounded-Buffer на мові CSP:

Виклик з боку виконавця: BB!p — запис нових даних

Виклик з боку користувача: BB!more ( ); BB?p — читання нових даних

```
BB::
buffer : (0..9) datensatz;  (* запис даних*)
in,out : integer; in:=0; out:=0;
    *[in < out+10; erzeuger?buffer(in mod 10)  (*виробник*)
          →      in :=in+1
     [ out < in;     verbraucher?more()
          → verbraucher!buffer(out mod1 0); (*користувач*)
       out := out+1
     ]
```

Після декларування та ініціалізації локальних даних використовується одна єдина команда (guarded), яка повторюється в безкінечному циклі. При цьому кожна умова (guard) складається з двох частин, які мають виконуватися разом. Повідомлення з процессу-виробника (генератора) будуть взяті тільки у випадку, коли є ще вільні місця в буфері. В інших випадках генератор має чекати (guard: in<out+10). При наявності достатньої кількості місця в буфері дані зчитуються і in-індекс збільшується на одиницю. Керування буфером виконується за принципом циклічної черги.

Користувач має замовити потрібні дані за допомогою повідомлення more(). У випадку, коли щонайменше одне місце в буфері зайняте (out<in), процес Bounded-Buffer надсилає зміст наступного місця в пам'яті користувачу і збільшує out-індекс на одиницю.

Процес обмеженого буфера зупиняється, якщо потік заявок пустий (in=out); зупиняється і генератор, тому що більше не задовольняється ніяка умова.

9.3. OCCAM

Автор: Inmos Limited, 1984.

Безпосереднім комерційним послідовником мови CSP є мова OCCAM, розроблена фірмою Inmos [Inmos 84]. Вона побудована спеціально для мережі трансп'ютерів, яка містить (в старому поколінні) висикопотужні мікропроцесори, кожний з яких має чотири канали зв'язку для обміну даними між трансп'ютерами. У трансп'ютерних процесорів нового покоління, що мають значно більшу обчислювальну потужність, забезпечується проста побудова мереж більшої розмірності за допомогою спеціальної комунікаційної мікросхеми.

Паралельні мовні конструкції:

SEQ – послідовне виконання блоків інструкцій;
PAR – паралельне виконання блоків інструкцій;

PRI – вибіркове закріплення процесу за деяким фізичним процесором з метою оптимізації;

ALT – вибір процесу з багатьох можливих процесів;

CHAN – декларування комунікаційних каналів між процесорами (channels);

PROTOCOL – визначення типу даних, що обмінюються в деякому каналі;

TIMER – визначення часових інтервалів, наприклад для періодичних процесів або тайм-аутів;

! – переслати дані (аналогічно CSP, п. 9.2)

? – прийняти дані (аналогічно CSP, п. 9.2)

У мові OCCAM вживаються ті самі концепції, про які йшлося в п. 9.2 відносно мови CSP. Програмні конструктиви високого рівня в мові OCCAM наявні лише частково. Окрім визначення типу даних передбачені ще й динамічні типи даних, але тут неможливе рекурсивне програмування. Найбільшою проблемою у великих програмах і системах з багатьма процесорами в OCCAM залишається пошук відповідного відображення процесів на процесори (тобто побудова конфігурації). Цей пошук недостатньо підтримується мовою програмування і часто не вирішується задовільно самою операційною системою.

Програма регулювання гучномовця (за публікацією [Pountain, May 87]):

```
VAL max IS 100 :
VAL min IS 0 :
BOOL aktiv :
INT lautstärke,wert : (*гучність, значення *)
SEQ
    aktiv := TRUE
    lautstärke:= min       (*гучність *)
    verstärker!lautstärke (*підсилювач ! гучність*)
    WHILE aktiv
        ALT
          (lautstärke<max) & lauter?wert
            SEQ
                lautstärke := lautstärke + wert
                IF
                  lautstärke > max
                    lautstärke := max
                verstärker!lautstärke

          (lautstärke>min) & leiser?wert
            SEQ
                lautstärke := lautstärke - wert
                IF
                  lautstärke < min
                      lautstärke := min
                verstärker!lautstärke
          aus?wert        (*вимкнути?значення*)
            PAR
                verstärker!min
                aktiv :=FALSE
```

Ця програма-приклад використовує найважливіші мовні конструкції OCCAM. SEQ і PAR вводять відповідно послідовний і паралельний порядок виконання конструкцій, а ALT дає змогу вибрати потрібне повідомлення з тих, що надходять. При цьому такі повідомлення, як lauter?wert (гучніше?значення) можуть наділятись умовою: так, умова (lautstärke<max (гучність<max)) має бути виконана, щоб повідомлення сприйнялося. Наведена вище програма бере від інших процесів повідомлення, що мають імена lauter, leiser і aus (гучніше, тихіше, вимкнути), а для визначення сили гучності використовує передавані числові величини, які в результаті відсилаються процесу verstärker (підсилювач).

9.4.ADA

Автор: Міністерство оборони США, 1979.

Мова програмування ADA [Sommerville, Morrison 87] з'явилась як результат розробок для американського міністерства оборони. Мета розробки – спробувати звести до однієї мови ті численні мови програмування, що

застосовувалися в системі міністерства. Кілька дослідницьких груп світового рівня дали свої розробки в цей проект, а комісія вибрала з цих розробок кращу і назвала ADA. Щоб відповідати вимогам універсальності, ADA мала використати декілька мовних концепцій, а це, у свою чергу, вело до певного перевантаження мови.

В ADA для обміну даними між паралельними процесами (tasks) вибрана концепція повідомлень (тут вона названа "концепція рандеву"). Наведімо далі короткий огляд найважливіших мовних конструкцій, які пояснюються програмою-прикладом.

Паралельні мовні конструкції:

task – паралельний процес;
entry – "вхідні пункти" процесу; декларування імен входів-повідомлень, які можуть викликатись іншими процесами;
accept – чекати на виклик entry іншим процесом. Виклик "вхідного пункта" з параметрами: ім'я процесу, крапка, ім'я entry, можливий параметр
Приклад: regles.lauter (5)

select – інструкція вибору, наприклад для очікування різних вхідних пунктів entrys (спочатку розшифровуються повідомлення, що надходять);

when – обмеження можливостей вибору при виконанні інструкції select; тільки у випадку, коли додержується булівська умова, може виконуватись цей напрям у програмі (порівняємо з конструкцією "guarded commands", автором якої є Dijkstra, п. 9.2)

У наступному прикладі (змінений приклад із п. 9.3) застосовуються вказані мовні конструкції. В ADA задача (task, процес) охоплює як специфікаційну частину, в якій декларуються всі пункти початку виконання процедур (entrys), так і тіло задачі, в якому міститься визначення локальних змінних і процедур та описування дії у вигляді

інструкцій. Процес (task), що має ім'я regler (регулятор), містить у безкінечному циклі селекцію як зовнішню інструкцію. Тут можна розрізнити три класи повідомлень (entries), що можуть надходити, а саме lauter (гучніше), leiser (тихіше) і aus (вимкнути). Кожна з трьох підпрограм (рутин) може викликатись іншими процесами за допомогою відповідного повідомлення, причому процес, що викликає, блокується під час виконання accept-рутини (підпрограми). Під час обміну даними вхідні (in) та вихідні (out) дані (тут не використовуються) можуть передаватись як параметри. Перші дві accept-інструкції захищені за допомогою when-умови. Тільки тоді, коли змінна величина lautstärke (гучність) знаходиться у потрібних межах, відповідна accept-інструкція може також бути обраною для даного повідомлення; вона завжди може бути виконаною для класу повідомлень aus, бо тут нема умови.

Програма для регулювання гучності:

```
task regler is
        entry lauter(wert:in integer);
        entry leiser(wert:in integer);
        entry aus( );
end;

task body regler is
max: CONSTANT integer := 100;
min: CONSTANT integer := 0;
lautstärke: integer;
begin
        loop
            select
                when lautstärke<max =>
                accept lauter(wert: in integer) do
                  lautstärke := lautstärke + wert;
                  if lautstärke>max then lautstärke:=max
                  end if;
                  verstärker.eigang(lautstärke);
                end lauter;
            or
                when lautstärke>min =>
```

```
                accept leiser(wert: in integer) do
                    lautstärke := lautstärke – wert;
                    if lautstärke<min then lautstärke:=min
                    end if;
                    verstärker.eigang(lautstärke);
                end leiser;
            or
                accept aus() do
                    verstärker.eingang(min);
                end aus;
            end select
        end loop
    end regler;
```

Наведена вище програма визначає "окрему задачу (task, процес)", тобто в паралельній програмі існує тільки один єдиний процес з ім'ям regler (controller). Він автоматично запускається під час фази ініціалізації ADA-програми. Тут також можливо задавати так звані "типи задач (tasks, процесів)" і декларувати декілька задач одного й того самого типу (див. наступний приклад). Навіть масиви (arrays) або записи (records) можуть декларуватись одним типом задач. Кожна задача одного типу автоматично запускається під час ініціалізації її декларування.

Приклад програми:

```
task type regler is
        entry lauter(wert: in integer);
        entry leiser(wert: in integer);
        entry aus( );
end regler;

c : regler;
multy: array (1..10) of regler;
```

9.5. Sequent-C

Автор: Sequent Computer Systems Incorporation, 1987.

Для різних мов програмування параллельної обчислювальної системи Sequent Symmetry і серед них мови

C розроблена "бібліотека паралельного програмування" [Sequent 87], яка розширює ці мови засобами паралельного виклику бібліотеки. Ці бібліотечні підпрограми (рутини) можуть викликатись як процедури або функції з параметрами зворотного зв'язку. Завдяки паралельній бібліотеці стають можливими як зовсім просте керування процесами (його можна порівняти з операторами fork і wait в UNIX, порівняйте з п. 4.2), так і рудіментарна синхронізація процесів. Ці рішення погоджені із системною структурою паралельної ЕОМ Sequent Symmetry (MIMD-система з зв'язаними через шини процесорами, має загальну пам'ять, див. п. 6.1).

Паралельні бібліотечні функції:

cpus_anline () – повідомлення про кількість наявних фізичних процесорів

m_set_procs (число) – визначення кількості потрібних процесорів

m_fork (func,arg1,...argn) – дублювання процедури і старт на багатьох процесорах (з ідентичними значеннями параметрів)

m_get_myid() - повідомляє власний номер "процеса-дитини"

m_get_numprocs() – повідомляє загальну кількість всіх "процесів-дітей" (або "процесів-побратимів")

m_kill_procs() – операція гасіння "процесів-дітей" ("процеси-діти" зупиняються в стані чекання і мають в ньому припинити існування).

Імплементація семафора:

S_init_lock (sema) – ініціалізація семафора
S_lock (sema) – **P**-операція
S_unlock (sema) – **V**-операція.

Розгляньмо приклад програми, який показує деякі з цих паралельних викликів бібліотеки у взаємозв'язку.

Фрагмент програми:

```
m_set_procs(3);          /*запит трьох наступних процесорів*/
m_fork(parproc,a,b);     /*старт паралельних процесів-дітей*/
m_kill_procs();          /*ліквідація процесів-дітей після їх термінування*/
void parproc(a,b)        /* паралельна процедура процесів-дітей */
{...
n=m_get_numprocs(); /* опитування і визначення кількості всіх процесів-
                        дітей */
m=m_get_myId();          /* опитування власного номера процесу */
...
}
```

Рис. 9.2 демонструє виконання чотирьох процесів. Після старту трьох процесів-дітей (за допомогою m_fork) головний процес (P_1) чекає на закінчення всіх процесів-дітей. Ці процеси виконують іхні процедури з різноманітними даними і завершуються в різні моменти, причому ті процеси-діти, що закінчилися раніше, чекають на закінчення всіх інших "побратимів" у циклі busy-wait.

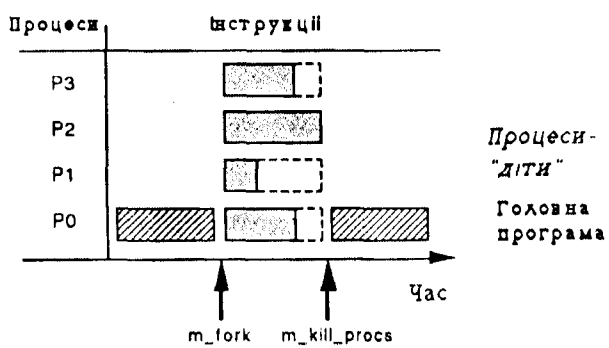


Рис. 9.2. Хід виконання програми-прикладу

Процеси-діти можуть мати доступ до глобальних даних щодо запису і читання, одначе вони самі несуть відповідальність за їхню синхронізацію (наприклад, за допомогою семафорів). Процеси-діти гасяться одночасно

(оператор m_kill_procs, в момент завершення процеса P_3 на рис.9.2), і головний процес продовжує роботу з своєю програмою.

Якщо в апаратній частині системи бракує достатньої кількості процесорів або операційна система виділяє їх менше, ніж потрібно, то розпаралелювана задача мусить вирішуватись ітеративно.

Щоб уникнути витрат ресурсів на стартування і термінування процесів, цикли мають бути розташовані всередині процедур обробки даних (процесів-дітей), що стартують паралельно.

Приклад програми для ітерації всередині процесів-дітей:

(N проходів циклу)

```
void parproc(a,b)                /* процес-дитина */
float a, b;
{ int zaehl,id,pe;
   pe=m_get_numprocs();          /* кількість всіх процесів-дітей */
   id=m_get_myId();              /* власний номер процесу */
   for (zaehl=id; zaehl<=N; zaehl+=pe)
  { /* безпосередні обчислення */ }
}
```

Припустімо, що треба виконати N=20 проходів циклу, але в наявності лише p=6 процесорів. У цьому випадку процесори дістають ідентифікаційні номери (id) від 1 до 6. Програма-приклад виконується тоді в такому порядку:

Процесор 1 виконує циклічні ітерації 1, 7, 13, 19
Процесор 2 виконує циклічні ітерації 2, 8, 14, 20
Процесор 3 виконує циклічні ітерації 3, 9, 15
Процесор 4 виконує циклічні ітерації 4, 10, 16
Процесор 5 виконує циклічні ітерації 5, 11, 17
Процесор 6 виконує циклічні ітерації 6, 12, 18

Кожний процесор виконує послідовно одну за одною виділені йому ітерації циклу. Процесори 1 і 2 мають виконувати більшу кількість ітерацій, ніж інші процесори, тому що число N=20 не ділиться без остачі на число p=6. Цей "виняток", одначе, не потребує спеціальної обробки в наведеній вище програмі. Кожний процесор виконує ітерації за умовою zaehl≤N, яка в процесорах 1, 2 дає на одну ітерацію більше. Це означає, що за умовою приблизно однакової швидкості обчислень процесори 3-6 вже закінчили роботу, а процесори 1-2 продовжують паралельно виконувати два останніх проходи циклу (№№ 19, 20).

9.6. Linda

Автори: Nicholas Carriero, David Gelernter, 1986.

Концепції паралельного програмування можуть бути повністю незалежними від наявної мови програмування. В Linda [Ahuja, Carriero, Gelernter 86], [Carriero, Gelernter 89] не стільки йдеться про мову програмування, як про деяку компактну множину мовнонезалежних паралельних концепцій для MIMD-систем. Вони можуть бути реалізовані на різних мовах – С, Фортран, Модула-2 та ін. Можливість застосування паралельних принципів доведена не тільки стосовно системи Linda, а й для інших мов паралельного програмування.

Концепція розпаралелювання системи Linda складається із загальної динамічної області даних (*datapool*), яка називається Tuple Space (кортеж множин даних) і до якої можуть одночасно мати доступ всі процеси (активні кортежі), щоб читати або записувати дані (пасивні кортежі). Кортежі (*tupel*) можуть містити в собі багато елементів даних будь-якого типу. Linda має шість основних

операцій, що забезпечують доступ до області даних Tuple Space.

Паралельні операції:

OUT – записи даних в кортеж множини даних. *Генерація пасивного кортежу.*

RD – читання даних з кортежу (не видаляючи їх). При цьому частини кортежу можуть бути зайняті раніше (див. рис.9.2); у цьому випадку мова може йти лише про відповідні кортежі (matchende Tupel). *Читання пасивного кортежу.*

RDR – предікат читання (булівська тестова операція для даних в кортежі). Перевірка без читання, чи є там відповідний кортеж даних.

IN – читання та вилучення елемента даних із кортежу. Відповідає операції RD із заключним гасінням даних кортежа. *Вилучення пасивного кортежу.*

INP – предікат читання. Аналогічний RDR без читання елемента даних, але із заключним гасінням.

EVAL – старт нового процесу. *Генерування активного кортежу.*

Термінування роботи програмної системи:

якщо нема жодного активного кортежу в динамічній області або якщо всі активні кортежі блоковані в операціях читання, то система термінується. В цьому випадку замовлених даних нема, ці дані також не можуть генеруватись іншими процесами. Отже, тут ситуація блокування виступає як умова термінування.

Як видно із розглянутого, на виконання може запускатися будь-яка кількість процесів, які обмінюються даними через один центральний Pool (пул, динамічно розподілювана область). Їхні паралельні доступи синхронізуються базисною операційною системою. Обрана тут MIMD-модель комунікацій зумовлює, одначе, деякі обмеження, через які імплементація мови Linda на MIMD-системі без загальної пам'яті можлива лише із суттєвими додатковими витратами ресурсів на обмін інформацією, а

це зменшує загальну ефективність паралельного розв'язання задач. На рис. 9.3 показано взаємодію операцій в TupelSpace.

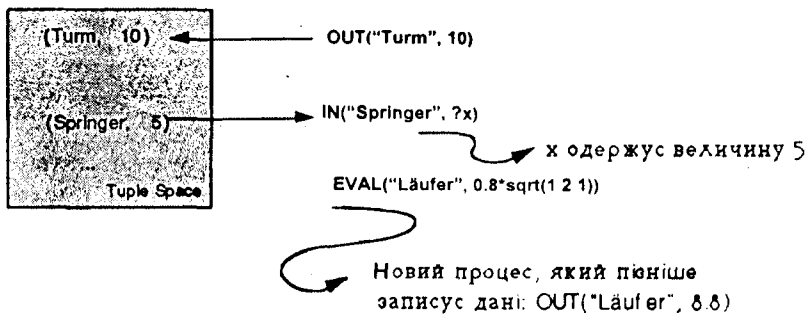


Рис. 9.3. Концепція паралельності мови Linda

Для ілюстрації функціональних особливостей цієї мови наведімо програму визначення простих чисел в C-Linda [Carriero, Gelernter 89]:

```
lmain( )
{ int i,ok;
        for (i=2; i<Limit; ++i) {
                EVAL("prim", i, is_prime(i));
}
        for (i=2; i<=Limit; ++i) {
                RD("prim", i, ?ok);
                if(ok) printf("%d\n", i);
}
```

Linda-програма стартує з оператора lmain і визначає прості числа від 2 до Limit. Спочатку запускається для кожного числа паралельний процес (операція EVAL), який за допомогою функції is_prime (її тут детально не розглядаємо) аналізує властивості простих чисел. Наприкінці операцією RD послідовно зчитуються результати із Tuple Space. Якщо якийсь із результатів ще не готовий, то операція зчитування чекає на нього, поки відповідний кортеж (Tupel) не буде оброблений підпорядкованим йому процесом. Усі кортежі,

про які тут йдеться, мають перший елемент даних – послідовність знаків prim, другий елемент – ціле число, третій – результат 0 або 1. На рис.9.4 проілюстровано частину кортежу даних (Datentupel) із програми.

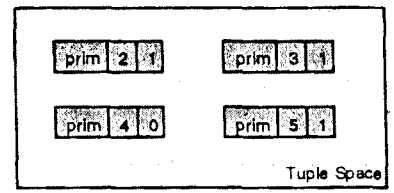


Рис.9.4. Побудова даних в програмі простих чисел

Переваги та недоліки паралельної мовної концепції Linda

+ Високий рівень описування проблеми.

Кожний процес працює незалежно від іншого і синхронізується з ним самостійно, обмінюючись даними через Tuple Space. Синхронізація відбувається цілком непомітно для користувача.

+ Апаратна незалежність Linda передбачає MIMD-систему із загальною пам'яттю, тому не існує жодного обмеження на імплементацію мови Linda в системах цього класу.

+ Мовна незалежність

Linda не залежить від будь-якої мови програмування. Імплементовані на сьогодні версії беруть до уваги обмеження процедурних мов програмування.

– Linda-модель меншою мірою придатна для MIMD-систем без загальної пам'яті. В цих розподілених системах виникає відносно високий і залежний від вирішуваних проблем рівень витрат на керування процесами, яке здійснюється через обчислювальну мережу за рахунок відповідних розширень в змінній частині кортежів (Tuple-Updates).

– Зрозумілість мовної концепції. Незважаючи на те що функції окремих мовних конструкцій легко зрозуміти,

побудова повнокомплексної Linda-програми далеко не проста. Особливо це стосується часових взаємодій процесів, що неявно виникають і їх часто дуже важко простежити під час написання програм.

9.7. Modula-P.

Автор: Томас Бройнль, 1986.

Мова Modula-P – це розширення мови Modula-2, що забезпечує асинхронну паралельність [Bräunl, Hinkel, von Puttkamer 86] і вже застосовувалася нами в попередніх розділах. Для демонстрації прикладів програм в Modula-P застосовані деякі концепції із Concurrent-Pascal (п. 9.1). Ця мова описує також різноманітні рівні синхронізації процесів за допомогою модулів рівня. Компілятор для Modula-P наданий в розпорядження користувачів як безкоштовне програмне забезпечення [Bräunl, Norz 92].

В Modula-P є модулі трьох видів:

• Модуль-процесор.

У центральному (головному) модулі здійснюється, якщо треба, синхронізація між процесорами або ЕОМ через повідомлення (за допомогою операції дистанційного виклику Remote-Procedure-Call). В цьому модулі починається ініціалізація всієї системи процесів.

• Модуль високого рівня (Highlevel-Modul).

З цього високого рівня процеси можуть декларуватись і запускатись на виконання. Існують також мовні конструкції для реакції на позаштатні ситуації з пунктами перезапуску процесів. Синхронізація між процесами одних і тих самих обчислювальних вузлів виконується через монітори та умови. Глобальні дані не існують!

• Модуль нижнього рівня (Lowlevel-Modul).

Цей модуль є найнижчим рівнем виконання програми. Він застосовується лише для запрограмованих задач реального часу або машинного керування (наприклад, роботом) з використанням певної області пам'яті. Переривання програм можуть декларуватись і виконуватись підпрограмами сервісу переривань. Синхронізація процесів одного й того самого вузла виконується за допомогою семафорів.

Реалізована тут триступенева ієрархія паралельної програми в загальних рисах відповідає розподілу концепцій синхронизації та обміну даними за їхніми рівнями абстракцій (повідомлення, монітори і семафори).

Паралельні мовні конструктиви

Переривання програм можуть декларуватися на початку модуля нижнього рівня як константи. Якщо в процесі виконання програми з'явиться сигнал переривання, то процес буде зупинено і будуть викликані відповідні сервісні підпрограми Event:

```
INTERRUPT Ctrl_C = 2;
...
EVENT Ctrl_C;
BEGIN
WriteString("Control-C взято на виконання ");
END Ctrl_C
```

Для мікропроцесорних систем в Modula-P існує конструктив для контрольованої відміни всіх переривань і спричинених ними змін процесів на період дії деякої послідовності команд і поновлення їх наприкінці цієї послідовності. Цей конструктив аналогічний блокові BEGIN...END. В UNIX-системах і багатопроцесорних ЕОМ вживання цього конструктиву неможливе. Конструктив має вигляд:

```
DISABLE
... (* критичні інструкції *)
ENABLE
```

Синхронізація через загальні семафори – це конструктив, який може бути вставлений всередині модулів низького рівня. Семафори декларуються як змінні типу SEMAPHORE, при цьому в прямокутних дужках задається число ініціалізації. На семафорах визначені операції **P** і **V** (див. приклади програм в гл.10):

```
VAR schutz: SEMAPHORE[1]; (* VAR захист *)
...
P(schutz);
...(*критичні інструкції, наприклад, доступ до глобальних даних*)
V(schutz)
```

У модулях высокого рівня паралельні процеси можуть декларуватись і запускатись на виконання в явній формі (подібно до процедур):

```
PROCESS abc(i: INTEGER);
BEGIN

... (* інструкції процесу *)
END PROCESS abc;
...
START(abc(1)); (* двократний старт процесу "abc"*)
START(abc(7)); (* з різними стартовими числами *)
```

Синхронізація процесів у модулях високого рівня виконується за допомогою моніторів з умовними чергами очікування. Монітори визначаються як блок, що трансформує операції над даними і операції доступу. Останні відповідають спеціальним процедурам і мають назву ENTRY. Вони створюють тільки одну можливість доступу до локальних даних відповідного монітора. Операції ENTRY здобувають під час виклику із деякого процесу ім'я монітора з двома крапками як префіксом, щоб

показати, що виклики моніторів взаємно виключаються. В той час, коли один процес за викликом операції ENTRY займає монітор, усі інші процеси, які б хотіли викликати той самий монітор, мають чекати. Щоб уникнути взаємного блокування, монітор-ENTRY не може викликати інший ENTRY. Приклад програми:

```
MONITOR sync;
VAR a:  ARRAY[1..10] OF INTEGER;(* дані монітора *)
    j:  INTEGER;

ENTRY lesen(i: INTEGER; VAR wert: INTEGER); (*читати*)

BEGIN
    wert:=a[i];
END lesen;

ENTRY schreiben (i,wert: INTEGER); (* записувати *)
BEGIN
    a[i]:=wert;
END schreiben;

BEGIN                          (* ініціалізація монітора *)
   FOR j:=1 TO 10 DO a[j]:=0 END
END MONITOR sync;
```

Виклик операції Монітор-ENTRY із деякого процесу:

```
sinc: lesen(10,w); (*читати*)
```

Логічні умови можуть декларуватись як змінні типу CONDITION в межах монітора. За цими умовами визначаються операції WAIT, SIGNAL та STATUS (кількість процесів, що перебувають в стані очікування). У зв'язку з тим що при імплементації SIGNAL-операції відповідно всі процеси, що очікують, деблокуються, операція WAIT має завжди бути в циклі WHILE, щоб уникнути взаємоблокувань:

```
VAR voll: CONDITION;
...
WHILE inhalt=0 DO WAIT(voll);
```

```
...
SIGNAL(voll);
```

Для кожного процесу може декларуватися програма реакції на виняткову ситуацію (ПРВС), яка замінює стандартну ПРВС (*exception-handler*) при появі деякої помилки (наприклад, ділення на нуль). В межах ПРВС може аналізуватися помилка за допомогою числового значення EXEPTNO і застосовуватися відповідна протидія. Затриманий процес може продовжуватись за допомогою операцій RESUME або заново запуститись операцією RESTART. Процедурою RAISE (<*номер помилки*>) також можуть генеруватися ці ситуації в явній формі:

```
PROCESS abc(z : INTEGER);
VAR i:INTEGER;

EXCEPTION
        IF EXEPTNO = 8 THEN (*Floating Point Exception*)
                i:=0;
        RESUME;
        END;
END EXCEPTION;

BEGIN
... (* інструкції процесу *)
END PROCESS abc;
```

Обмін інформацією між двома процесами, які локалізовані в різних обчислювальних вузлах і не мають загальної пам'яті для спілкування, здійснюється за рахунок обміну повідомленнями в формі Remote-Procedure-Call. Мовним конструктивом для цього обміну є COMMUNICATION. Зв'язки між процесами будуються процедурою INITCOM в ініціалізуючій частині процесор-модуля і можуть викликатися деяким процесом за допомогою конструктива CALL так само, як і процедура:

```
COMMUNICATION pythagoras (x,y: REAL; VAR ergebnis: REAL);
VAR r: REAL;
BEGIN
        r := x*x + y*y;
        ergebnis := SQRT(r);
END COMMUNICATION pythagoras;
```

Ініціалізація мовного конструктива COMMUNICATION:

```
CONST vince = Computer("sparc", "vincent"); (* клас та ім'я ЕОМ *)

INITCOM(pythagoras, vince);
```

Виклик конструктива COMMUNICATION:

```
CALL vince: pythagoras (a,b,c);
```

Процесами та комунікаційно-серверними процесами можуть запускатися на виконання декілька ідентичних копій. Для цього треба задати кількість копій процесів як допоміжний параметр. Керування запитами на сервер-процеси, що можуть використовуватися багаторазово, бере на себе базова операційна система.

10. ВЕЛИКОБЛОКОВІ ПАРАЛЕЛЬНІ АЛГОРИТМИ

У разі застосування MIMD-систем (на відміну від SIMD-систем) витрати на обмін даними між процесами набагато більші, ніж на арифметико-логічні операції. У зв'язку з цим доцільно, маючи на увазі раціональне використання ресурсів, виконувати локально в процесах якомога більше обчислювальних операцій між будь-якими двома необхідними операціями обміну даними. На основі такого підходу виникають так звані великоблочні ("крупнозернисті") паралельні алгоритми на противагу дрібноблоковим ("дрібнозернистим") паралельним

алгоритмам у SIMD-систем. Процесори, на яких побудовано MIMD-системи, здебільшого набагато перевершують за обчислювальною потужністю процесори SIMD-систем, а тому вони потребують значно більше інтегральних схем. Це призводить до того, що MIMD-системи мають набагато менше процесорів, ніж SIMD-системи. Алгоритми для MIMD-ЕОМ розробляються з огляду на невелику кількість високопотужних процесорів, в той час як в основі алгоритмів для SIMD-ЕОМ лежить велика кількість простих малопотужних процесорних елементів (масивна паралельність).

Доповнімо розглянуті в попередніх главах поняття процесу, принципи побудови механізмів синхронізації на базі семафорів та моніторів рядом повнозмістовних програм, написаних на мові паралельного програмування Modula-P (див. п.9.7).

10.1. Обмежувальний буфер з семафорами

Показана тут імплементація проблеми обмежувального буфера із п.7.3 застосовує для синхронізації процесів семафори. Процес Erzeuger (виробник, генератор) постійно генерує нові дані (від 0 до 9 в безкінечному циклі), а процес Verbraucher (користувач) постійно зчитує ці дані та обчислює суму всіх цифр, що зчитані на деякий момент.

Програма складається з двох модулів. Головним модулем є PROCESSOR MODULE, в якому визначаються і запускаються на виконання обидва процеси. Модуль LOWLEVEL MODULE містить семафори як засоби синхронізації:

```
1 PROCESSOR MODULE bounded_buffer;
2 IMPLEMENTATION
3 IMPORT io,synch;
4
5 PROCESS Erzeuger;
6 VAR i: INTEGER;
7 BEGIN
8       i:=0;
9       LOOP
10      i:=(i+1) MOD 10;
11      erzeuge(i);
12 END
13 END PROCESS Erzeuger;
14
15 PROCESS Verbraucher;
16 VAR i,quer: INTEGER;
17 BEGIN
18      quer:=0;
19      LOOP
20      verbrauche(i);
21      quer:=(quer+i) MOD 10;  (*використання даних*)
22 END
23 END PROCESS Verbraucher;
24
25 BEGIN
26      WriteString("Init Processor Module");WriteLn;
27      START(Erzeuger);
28      START(Verbraucher);
29 END PROCESSOR MODULE bounded_buffer.
```

```
1 LOWLEVEL MODULE synch;
2 EXPORT
3 PROCEDURE erzeuge (i: INTEGER); (*процедура генерування *)
4 PROCEDURE verbrauche(var i:INTEGER); (*проц.  використання*)
5
6 IMPLEMENTATION
7 IMPORT io;
8
9 CONST n:=5;
10 VAR buf     : ARRAY [1..n] OF INTEGER;
11      pos,z  : INTEGER;
12      Kritisch: SEMAPHORE[1];
```

```
13        Frei    : SEMAPHORE[n];
14        Belegt  : SEMAPHORE[0];
15
16 PROCEDURE erzeuge(i: INTEGER);
17 BEGIN
18        P(Frei);
19        P(Kritisch);
20        IF pos>=n THEN WriteString("помилка в    генераторі");
21                  WriteLn; HALT;
22           END;
23        pos:=pos+1;
24        buf[pos]:=i;
25        (* *) WriteString("schreiben Pos:");WriteInt(pos,5);
26        (* *) WriteInt(i,5);WriteLn;
27       V(Kritisch);
28        V(Belegt);
29 END erzeuge;
30
31 PROCEDURE verbrauche(VAR i: INTEGER);
32 BEGIN
33        P(Belegt);
34        P(Kritisch);
35 IF pos<=0 THEN WriteString("помилка використання");
36           WriteLn; HALT;
37      END;
38        i:=buf[pos];
39        (* *) WriteString("читати Pos:");WriteInt(pos,5);
40        (* *) WriteInt(i,5);WriteLn;
41        pos:=pos-1;
42    V(Kritisch);
43    V(Frei);
44 END verbrauche;
45
46 BEGIN
47        WriteString("Init Synch");WriteLn;
48        pos:=0;
49        FOR z:=1 TO n DO buf[z]:=0 END;
50 END LOWLEVEL MODULE synch.
```

Приклад виконання програми семафорів (реалізація в MIMD-системі Sequent Symmetry):


```
Init Synch
Init Processor Module
schreiben Pos:  1     1
schreiben Pos:  2     2
lesen     Pos:  2     2
schreiben Pos:  2     3
lesen     Pos:  2     3
schreiben Pos:  2     4
lesen     Pos:  2     4
schreiben Pos:  2     5
lesen     Pos:  2     5
schreiben Pos:  2     5
lesen     Pos:  2     5
schreiben Pos:  2     6
lesen     Pos:  2     6
.....
```

В основному операції запису і читання виконуються навперемінно, тому що на генерацію та використання даних система потребує приблизно однаковий час. Одначе, як показують перша і друга виконувані операції в цьому прикладі (помічено стрілкою), вони не повинні взаємозмінюватися (в серії виконуються дві операції запису, write), паралельність обмежується тільки величиною буфера.

10.2. Обмежувальний буфер з монітором

Розгляньмо вирішення проблеми обмежувального буфера (*Bounded-Buffer-Problem*) за допомогою монітора (замість семафора). Як і в попередньому прикладі, процес Erzeuger генерує постійно нові дані, в той час як процес Verbrauchher ці дані постійно використовує й обчислює суму зчитаних чисел. Програма, що наводиться нижче, складається з двох модулів. Головним, як і раніше, є модуль

PROCESSOR MODULE, в якому визначаються і запускаються на виконання процеси. Другий модуль високого рівня (*Highlevel Module*) містить у собі засоби синхронізації за допомогою монітора з двома входами (Entries) і двома умовами (Conditions):

```
1 PROCESSOR MODULE highlevel_buffer;
2 IMPLEMENTATION
3 IMPORT io,msynch;
4
5 PROCESS Erzeuger;
6 VAR i: INTEGER;
7 BEGIN
8       i:=0;
9       LOOP
10              i:=(i+1) MOD 1000; (* генерування даних *)
11              Buffer:schreiben(i);
12      END
13 END PROCESS Erzeuger;
14
15 PROCESS Verbraucher;
16 VAR i,quer: INTEGER;
17 BEGIN
18      quer:=0;
19      LOOP
20              Buffer:lesen(i);
21              quer:=(quer+i) MOD 10; (* використання даних *)
22      END
23 END PROCESS Verbraucher;
24
25 BEGIN
26      WriteString("Init Processor Module");WriteLn;
27      START(Erzeuger);
28      START(Verbraucher);
29 END PROCESSOR MODULE highlevel_buffer.
```

```
1 MODULE monitor_buffer;
2
3 EXPORT MONITOR Puffer;
4       ENTRY schreiben(a: INTEGER);
5       ENTRY lesen (VAR a: INTEGER);
6
7 IMPLEMENTATION
8 IMPORT io;
9
10 MONITOR Puffer;
11 CONST max = 5;
12
13 VAR Stapel: ARRAY [1..max] OF INTEGER;
14      Zeiger: INTEGER;
15      Frei, Belegt: CONDITION;
16
17 ENTRY schreiben(a: INTEGER);
18 BEGIN
19 WHILE Zeiger=max (*буфер повний*) DO WAIT(Frei)     END;
20      inc(Zeiger);
21      Stapel[Zeiger] :=a;
22      IF Zeiger=1 THEN SIGNAL(Belegt) END;
23      (* *) WriteString("запис"); WriteInt(Zeiger,3);
24      (* *) WriteInt(a,3); WriteLn;
25 END schreiben;
26
27 ENTRY lesen(VAR a: INTEGER);
28 BEGIN
29      WHILE Zeiger=0 (* буфер пустий *) DO WAIT(Belegt) END;
30      a:=Stapel[Zeiger];
31      (* *) WriteString("читати");WriteInt(Zeiger,3);
32      (* *) WriteInt(a,3);WriteLn;
33      dec(Zeiger);
34      IF Zeiger = max-1 THEN SIGNAL(Frei) END;
35 END lesen;
36
37 BEGIN (* Monitor – Initialisierung *)
38      WriteString("Init Monitor");WriteLn;
39      Zeiger:=0;
40 END MONITOR Puffer;
41
42 BEGIN
43      WriteString("Init Highlevel Module");WriteLn;
44 END MODULE monitor_buffer.
```

Приклад виконання програми "Монітор" (реалізація на системі Sequent Symmetry) :

```
Init Monitor
 Init Highlevel Module
 Init Processor Module
schreiben :   1     1
lesen     :   1     1
schreiben :   1     2
lesen     :   1     2
schreiben :   1     3
schreiben :   2     4
lesen     :   2     4
schreiben :   2     5
lesen     :   2     5
schreiben :   2     6
lesen     :   2     6
schreiben :   2     7
lesen     :   2     7
schreiben :   2     8
lesen     :   2     8
.....
```

Як видно з наведеного нижче результату, операції читання (lesen) та запису (schreiben) в цьому випадку також трапляються однаково часто, але в рядках 5, 6 виконуються підряд дві операції запису (помічено стрілкою). Таким чином, обидва процеси можуть виконуватися незалежно, паралельно до тих пір, поки вони не перевищують величини буфера.

10.3. Розподіл заявок через монітор

На рис.10.1 показано наближений розрахунок числа π за алгоритмом [Babb 89], в якому застосовується правило прямокутників:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \approx \sum_{i=1}^{Interv} \frac{4}{1+i^2} * (\text{ширина})$$

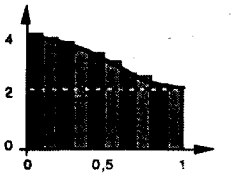


Рис.10.1. Наближене обчислення інтеграла

Цей алгоритм меншою мірою може застосовуватися для порівняння паралельних мов програмування, бо тут за винятком складання площ прямокутників не відбувається жодного обміну між процесорами.

Вісь x розбивається на таку кількість інтервалів, яка потрібна для досягнення точності обчислень. Завдання кожного робочого процесу полягає в обчисленні площ багатьох інтервалів. Площі-частки складаються у моніторі в загальну суму, яка є наближеним значенням числа π. За правилом прямокутників кожне значення функції обчислюється в центрі інтервалу та перемножується з шириною інтервалу. Аналогічну постановку задачі реалізовано в п.15.1 як масивно паралельний алгоритм для SIMD-системи:

```
PROCESSOR MODULE pi_calc;
IMPLEMENTATION
IMPORT io;
CONST intervals = 100; (* кількість інтервалів *)
   width = 1.0 / FLOAT(intervals);(* ширина інтервалу *)
   num_work  = 5; (* кількість робочих процесів *)

PROCEDURE f (x: REAL): REAL;
(*функція, що інтегрується *)
 BEGIN
      RETURN(4.0 / (1.0 + x*x))
 END f;

 PROCEDURE start_procs;
 VAR i: INTEGER;
 BEGIN
```

```
17      FOR i:=1 TO num_work DO START(worker(i)) END
18  END start_procs;
19
20  MONITOR assignment;
21  VAR sum : REAL;
22      pos,answers: INTEGER;
23
24  ENTRY get_interval(VAR int: INTEGER);
25  BEGIN
26      pos := pos +1;
27      IF pos<=intervals THEN int := pos
28                        ELSE int := -1 (* готово *)
29  END;
30  END get_interval;
31
32  ENTRY put_result(res: REAL);
33  BEGIN
34      sum := sum + res;
35      answers := answers + 1;
36      IF answers = intervals THEN (* видати результат *)
37         WriteString("Pi = "); WriteReal(sum,10); WriteLn;
38  END;
39  END put_result;
40
41  BEGIN (* monitor-init *)
42       pos := 0; answer := 0;
43       sum := 0.0;
44  END MONITOR assignment;
45
46  PROCESS worker(id: INTEGER);
47  VAR iv : INTEGER;
48      res: REAL;
49  BEGIN
50  assignment:get_interval(iv); (*читати перше завдання з монітора*)
51    WHILE iv > 0 DO
52      res := width * f((FLOAT(iv))-0.5) * width);
53      assignment:put_result(res);(*результат відіслати в монітор*)
54      assignment:get_interval(iv); (* читати завдання з монітора *)
55    END
56  END PROCESS worker;
57
58  BEGIN
59      start_procs;
60  END PROCESSOR MODULE pi_calc.
```


Кожний робочий процес за допомогою операцій get_interval в циклі викликає будь-яку наступну заявку із монітора. Після проведеного розрахунку величина площі-частки знову повертається в монітор за допомогою операції put_result і там складається в загальну суму. У зв'язку з простотою цієї постановки задачі часткові заявки завжди однакові, тому можна було б також зекономити на дорогих entry-викликах для операції get_interval і підпорядковувати інтервали за допомогою параметра процесу id, що тут не використовується (зверніть увагу на приклад для Sequent-C в п.9.5).

Вправи до розділу II

1. Покажіть, який результат одержано кожною послідовністю виконання операцій, що показані на рис.8.1.

2. Декілька ідентичних процесів у великоблоковій паралельній MIMD-системі мають звертатися до загальних даних і синхронізуватись за допомогою семафора. Отже, тільки одному процесу в деякий момент часу надано можливість звертатися до загальних даних.

2.1. Яким значенням має ініціалізуватися семафор?

2.2. Доповніть цю програму тільки тими операціями синхронізації, яких бракує:

```
PROCESS aufgabe;
VAR s: SEMAPHORE[...];
BEGIN
        LOOP
        .....
              (* виконай завдання над загальними даними *)
        .....
        END;
END PROCESS aufgabe;
```

2.3. Якщо процеси, що визначені в завданні 2.2, звертаються до загальних даних, щоб їх читати, то це можуть робити одночасно багато процесів. Хай із умов ефективності максимальна кількість процесів, що читають, обмежена до 5. Як має змінитися фрагмент програми в завданні 2.2 відповідно до цієї нової умови синхронізації?

3. Декілька ідентичних процесів у паралельній MIMD-системі мають звертатися до загальних даних і синхронізуватися за допомогою монітора. Отже, тільки одному процесу в деякий момент часу надано можливість звертатись до загальних даних.

Доповніть цю програму тільки тими операціями синхронізації, яких бракує:

```
MONITOR aufgabe;
VAR Platz_Frei,
        Element_da: CONDITION;
        d : Gemeinsame_Daten; (*загальні дані *)
BEGIN
    ENTRY Daten_ablegen (e: Daten_Element); (* дані внести *)
    WHILE "Puffer voll" DO .....
        (* додайте елемент даних до загальних даних*)
..................................      END

 Daten_ablegen;  ENTRY Daten_wegnehmen(VAR e: Daten_Element);
(*дані вибрати*)
WHILE "Puffer leer" DO .....
.....
END Daten_wegnehmen;

BEGIN
(* ініціалізація монітора *)
END MONITOR aufgabe;
```

4. У паралельній MIMD-системі запускаються на виконання копії деякого "робочого процесу", що мають ім'я zahl-arbeiter. Між цими копіями треба рівномірно розподілити виклики виконуваних процедур від f(1) до

zahl_aufgaben), де параметр zahl_aufgaben визначає кількість завдач.

Доповніть з використанням символів констант для розв'язання вказаної задачі процес Arbeiter в цій Modula-P-програмі (припустімо, що процедуру f задано):

```
CONST zahl_arbeiter  = 10;
      zahl_aufgaben = 256;

PROCESS Arbeiter (nr: INTEGER);
VAR i: INTEGER;
BEGIN
    .....
END PROCESS Arbeiter;

PROCEDURE Init;
VAR z: INTEGER;
BEGIN (* ініціалізація*)
    FOR z:=1 TO zahl_arbeiter DO START(Arbeiter(z)) END;
END Init;
```

5.

а) Опишіть на мові Modula-P як систему процесів мережу Петрі, показану на рис.3.8, причому мусить застосовуватись тільки один семафор.

б) Розробіть варіант мережі Петрі для вирішення проблеми "виробник-користувач", що показана на рис.7.2. Варіант повинен обходитись лише одним семафор-вузлом. Це дасть змогу синхронізувати тільки одного виробника (генератора) та тільки одного користувача.

6. Найдіть таке вирішення проблеми читання-запису (Readers-Writers-проблема), в якому після першого процеса-записувача, що очікує, не допускається жодного процеса-читача. Після того, як всі процеси-читачі покинули критичну область, процес-записувач може працювати виключно з загальними даними.

7. Хай змінна A має значення 500. Яку величину зможе прийняти змінна A після несинхронізованого виконання наступних трьох процесів?

P_1	P_2	P_3
x:=a;	y:=a;	z:=a;
x:=10*x;	y:=y+1;	z:=z-3;
a:=x;	a:=y;	a:=z;
x:=a;	y:=y+5;	z:=a;
x:=2*x;	a:=y;	z:=z-7;
a:=x;		a:=z;

Запропонуйте загальний метод для визначення всіх можливих величин змінної (а також їх максимальної кількості), які можуть з'явитись завдяки взаємозв'язаній за часом обробці несинхронізованих процесів.

8. Реалізуйте на мовах OCCAM і ADA розподіл циклічних ітерацій, поданий в п. 9.5 для Sequent-C.

9. У різних мовах програмування, як, наприклад в ADA, передбачено комунікаційні конструктиви для обміну повідомленнями. Семафорні операції **P** і **V** мають імплементуватись виключно за допомогою обміну повідомленнями. Наступна програма реалізує деякий булівський семафор, в якому спочатку дозволяється виконання **P**-операцій. Наступні комунікації з "**P**-повідомленнями" запам'ятовуються в буфері повідомлень до того моменту, коли з'явиться "**V**-повідомлення". Можливість появи багатьох **V**-операцій безпосередньо одна за одною в програмі не передбачена!

```
TASK BODY bool_semaphor IS
BEGIN
   LOOP
        ACCEPT P
        ACCEPT V
   END LOOP;
END bool_semaphor;
```

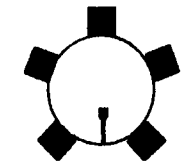

Доповніть дану нижче у псевдо-записі програму так, щоб реалізувався деякий загальний семафор. На початку семафор має дозволити 10 **P**-операцій без появи **V**-операцій.

```
TASK BODY int_semaphor IS
zaehler : INTEGER;
BEGIN
        zaehler := ..... ;
        LOOP
             SELECT
                WHEN ..... =>
                 ACCEPT P DO
                 .....
                 END P;

                 ACCEPT V DO
                 .....
                 END V;
             END SELECT;
        END LOOP;
END int_semaphor;
```

10. Напишіть програму на мові Sequent-C для проблеми "виробник-користувач".

11. Напишіть програму на мові MODULA-P для вирішення проблеми філософа Дінінга (гляньте на цей рисунок):



За круглим столом сидять п'ять філософів, а поміж кожним з них лежить відповідно одна виделка. Кожний філософ перебуває в таких станах: думає → зголоднів → приймає

їжу → думає. На початку всі філософи думають; якщо один з них відчуває голод, то він мусить взяти обидві виделки, що лежать зліва і справа біля нього, для того щоб можно було поїсти. Може легко виникнути блокування, якщо, наприклад, всі філософи візьмуть одночасно виделку зліва від себе і будуть чекати (назавжди) виделку справа. Програмна реалізація має так координувати "режим доступу" філософів, щоб не могло з'явитися блокування. Одне з можливих рішень полягає в тому, що спочатку треба впевнитися, що виделки зліва і справа вільні. Якщо це так, то філософ бере обидві виделки одночасно і їсть. Якщо ні, то він мусить, незважаючи на голод, чекати, поки звільняться обидві виделки. Одначе при такому простому вирішенні може завжди настати "виснаження" деяких філософів.

12. Вирішіть проблему філософа Дінінга із завдання 11 за допомогою мережі Петрі.

13. Напишіть програму на мові Modula-P для паралельного керування роботою складу матеріалів. У зв'язку з тим що багато процесів можуть одночасно звернутись до фондів складу, треба синхронізувати ці звернення за допомогою монітора.

Речення типу "наявність" мають форму:
<номер артикула><ціна><кількість>
Речення типу "бюджет" мають форму:
<стаття витрат><бюджет>

Декілька ідентичних процесів керування мають обробляти звернення (запити) таких форм:

- вилучення з наявності та списання суми з вказанної статті витрат;
- надходження товарів;

- видавання інформації щодо актуальної наявності товарів.

14. Напишіть програму на мові Modula-P для паралельного обчислення фрактальних зображень за такими комплексними ітераційними формулами:

а) $z_0 := 0$

$z_{i+1} := z_i^2 + c$

б) $z_0 := 0$

$z_{i+1} := z_i^3 + (c-1)z_i - c$

Прорахуйте поле з 100x100 пунктами зображення і виберіть, наприклад, для частини а) область зображення [(-0.76+0.01j), (-0.74+0.03j)]. Ітерація має перерватись, якщо $|z|>2$ або максимальна кількість ітераційних кроків дорівнює 200. Кожний пункт зображення фарбується відповідно до свого ітераційного числа (в найпростішому випадку: 200=чорний, інакше-білий). Запустіть на виконання 5 робочих процесів, які резервують собі через монітор запитів квадратні області розміром 10x10. Зверніть увагу на таке: якщо межа деякого часткового квадрата повністю конвертована (при максимальному ітераційному числі кожного крайнього елемента), загальна площа також конвертується і тому далі не обчислюється.

16. Наступна програма показує два процеси, які керують ресурсами, а саме терміналом TE та прінтером DR:

P1	***P2***
loop	loop
<інструкції>	*<інструкції>*
P(TE);	P(DR);
P(DR);	P(TE);
<обчислення>	*<обчислення>*
V(DR);	V(TE);
V(TE);	V(DR);

end; end;

Побудуйте відповідну мережу Петрі за умови, що до початку виконання процеси знаходяться в стані *<інструкції>* . Перевірте живучість побудованої мережі.

III

СИНХРОННА ПАРАЛЕЛЬНІСТЬ

У разі синхронної паралельності процесори, що використовуються для розв'язання задачі, працюють за командами центральної програми, що надходять в однакові такти часу. Процесори не можуть функціонувати незалежно один від одного, бо синхронна паралельна програма має лише один керуючий потік команд. Хоч ця спрощена модель обчислень і має обмеження, проте вона дає змогу більшою мірою інтегрувати процесори за рахунок простішої побудови їх. Тому з'являється можливість компонувати обчислювальні системи із значно більшою кількістю процесорів, ніж це можливо у разі асинхронної паралельності. Ці обставини дають в результаті поняття "масивної паралельності". Синхронізація між процесорами реалізується неявно на кожному кроці і програмісту не треба займатись її організацією. Процесори працюють над меншими об'єктами обробки, причому основна частина їх зосереджується на векторних математичних виразах. Це так зване "паралельне за даними" програмування відкриває

багато нових можливостей, але й потребує нових алгоритмічних підходів.

11. ПОБУДОВА ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ SIMD-СТРУКТУРИ

Синхронній моделі паралельності відповідає ЕОМ SIMD-структури (single instruction, multiple data), як показано на рис.11.1. Центральна керуюча ЕОМ – це звичайна послідовна машина (SISD: single instruction, single data), якій, як правило, підпорядковані й периферійні прилади. Паралельні процесорні елементи (ПЕ) не виконують своєї програми, а одержують команди тільки від керуючої ЕОМ. За обставин, коли ПЕ не мають у своїй структурі блоку команд, вони не є "повнофункціональними" процесорами, а є несамостійними арифметико-логічними пристроями (ALU, arithmetic logic unit) з локальною пам'яттю та засобами для комунікацій. Із цього

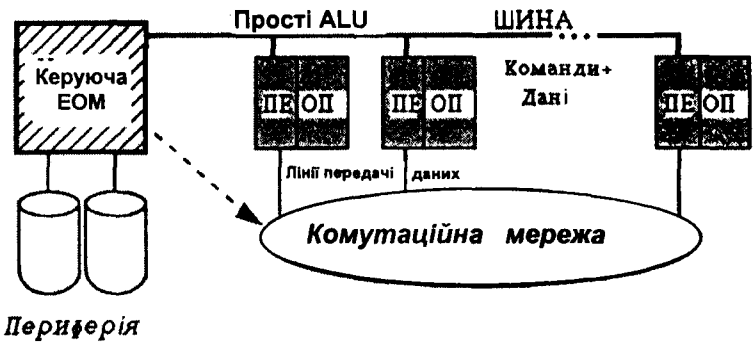


Рис. 11.1. Обчислювальна система SIMD-структури

спрощення моделі процесорних елементів, природно, виникають обмеження SIMD-моделі обчислювальних систем. ПЕ не можуть виконувати різні команди в один і той самий момент: вони або виконують одержану від

керуючої ЕОМ команду над їхніми локальними даними, або ж перебувають в пасивному стані. Кожне паралельне розгалуження (if-команда) має розкладатися на два этапи:

1) спочатку виконується then-частина на всіх тих ПЕ, де виконано умову розгалуження (селекції, вибору напрямку обчислень); на цьому етапі інші ПЕ, де ця умова не виконана, залишаються в пасивному стані;

2) виконується else-частина if-команди на тій групі ПЕ, що були пасивними, а ті ПЕ, що працювали на першому етапі, переводяться в неактивний стан.

Очевидно, що таке двохетапне виконання if-команд дуже неефективне. Спрощені ПЕ дають можливість інтегруватись значно більшою мірою, а це зумовлює те, що SIMD-системи мають незрівнянно більшу кількість процесорів, ніж MIMD-системи. Масивна паралельність (що означає застосування тисячі і більше ПЕ) повністю компенсує цю неефективність за відповідного використання SIMD-систем.

Процесорні елементи взаємоз'язані за допомогою комунікаційної мережі, яка може бути побудована як мережа з постійною конфігурацією або як мережа з можливостями реконфігурації. Вона забезпечує швидкий паралельний обмін даними між групами процесорних елементів (ПЕ). Відбувається також обмін даними між керуючою ЕОМ та окремими ПЕ (селективно) або всіма ПЕ (broadcast, передача інформації від ЕОМ усім ПЕ).

11.1. Обчислювальні SIMD-системи

Нижче наводяться найважливіші характеристики трьох типових паралельних обчислювальних SIMD-систем. Дані щодо максимальної теоретичної продуктивності самі по собі лише частково характеризують системи. У зв'язку з тим, що ці дані належать до дуже простих операцій (наприклад, множення скалярних величин), а прикладні SIMD-програми виконують у загальному випадку значно

складніші обчислення, на практиці ці теоретичні дані не досягаються. Проте вони досить корисні для порівняння різних SIMD-систем, що виконують одні й ті самі операції. Тут використовуються одиниці продуктивності ЕОМ MIPS (million instruction per secomd) та MFLOPS (million floating point operation per second), причому мають також братися до уваги довжина слова (наприклад, 32 розряди) та вид взятої для вимірювання операції (наприклад, тільки складання, пошук середньої величини або скалярний добуток).

Connection Machine

Виробник: Thinking Machines Corporation Cambridge, Massachusetts
Модель: CM-2 ;
Процесори: 65536 ПЕ (однорозрядні ПЕ);
Пам'ять кожного ПЕ: 128 КБ (максимально);
Продуктивність: 2500 MIPS (32 розряди);
10000 MFLOPS (32 розряди);
5000 MFLOPS (64 розряди);

Комутаційна мережа: глобальний гіперкуб та чотиривимірна реконфігурована решітка, що з'єднує сусідні процесори (реалізується через гіперкуб).

Мови програмування:

1.CMLISP (первісний варіант Lisp);
2.*Lisp (розширення Common Lisp)
3. C* (розширення мови C);
4.CMFortran (на основі Fortran 90);
5.C/Paris (мова C з викликом асемблерних бібліотечних підпрограм).

Паралельна обчислювальна система CM-2 – це найпотужніша і найвідоміша SIMD-система. Вона представлена процесорними елементами, які мають лише однорозрядні арифметико-логічні пристрої (ALU). Кожним 32 процесорним елементам відповідає один співпроцесор з плаваючою комою, що дає значне збільшення

продуктивності в межах гіга-FLOPS (GFLOPS) порівняно з системами, в яких не застосовуються арифметичні співпроцесори. Система CM-2 має дві різні комутаційні структури: глобальну гіперкубічну мережу для загальних комунікацій і швидку локальну решітку ПЕ-сусідів, яка реалізується за допомогою гіперкуба і може гнучко реконфігуруватись. Кожні 16К процесорних елементів одержують команди від незалежної керуючої ЕОМ ("Sequencer"), тому CM-2 повної конфігурації еквівалентна чотирьом об'єднаним SIMD-системам, які можуть працювати і роздільно. Ще однією особливістю CM-2 є те, що вона пропонує користувачу віртуальні процесори з спеціальною апаратною підтримкою. Це дає змогу під час програмування задач використовувати більше процесорних елементів, ніж їх фізично є (див. п.11.3). Дуже важливим є те, що CM-2 пропонує декілька мов програмування – від Асемблера та розширеного Фортрану до функціональних мов програмування.

Новітня модель паралельної ЕОМ фірми Thinking Machines – це CM-5, в якій реалізовано поєднання MIMD- і SIMD-структур (SPMD-модель, п.2.1).

MasPar

Виробник: MasPar Computer Corporation, Sunnyvale, Каліфорнія
Модель: MP-1216;
Процесори: 16384ПЕ (чотирирозрядні ПЕ);
Пам'ять кожного ПЕ: 64 КБ (максимально);
Продуктивність: 30000 MIPS (32-розрядні оп.);
1500 MFLOPS (32 розряди);
600 MFLOPS (64 розряди);

Комутаційна мережа: триступеневий глобальний розподілювач перехресних шин (Router) і восьмивимірна решітка, що з'єднує сусідні ПЕ (незалежно від розподілювача).

Мови програмування:
1.MPL (розширення C);
2.MPFortran (на основі мови Fortran 90).

Серія MP-1 фірми MasPar порівнянно з CM-2 наполовину дешевша, але менш продуктивна. Немає арифметичних співпроцесорів, що змушує виконувати операції з плаваючою комою програмними засобами. Це зумовлює те, що MasPar не може досягти високої продуктивності, характерної для цілих чисел, в області операцій з плаваючою комою. В MasPar ще чіткіше, ніж в CM-2, можна визначити дві окремі комунікаційні структури: швидкодіюча локальна решітка для сусідніх ПЕ (хоч і має вісім напрямків комутації, проте без реконфігурацій), а також глобальний триступеневий маршрутизатор (Router) для будь-яких з'єднань між ПЕ. В MP-1 не існує апаратної підтримки для віртуальних процесорів: цю задачу має брати на себе "інтелігентний компілятор". Послідовний керуючий комп'ютер для MP-1 на відміну від звичайної ЕОМ фон Ноймана має роздільну пам'ять для даних і для програм (так звана архітектура "Harvard-Style"). Фірма MasPar як мови програмування пропонує розширення мов C і Фортран, причому тільки Фортран забезпечує деяке керування віртуальними процесорами.

Розподілений матричний процесор (Distributed Array Prozessor, DAP)

Виробник:	Cambridge Parallel Processing Limited, Reading, Англія; A Division of Cambridge Management Co.,Irvine CA (раніше: Active Memory Technology AMT).
Модель:	DAP 610 ;
Процесори:	4096 ПЕ (1-розрядні процесори + 8-розрядні співпроцесори);
Пам'ять:	32 КБ ;
Продуктивність:	40000 MIPS (1-розрядна оп.); 20000 MIPS (8-розрядна оп.); 560 MFLOPS;

Комутаційна мережа: чотиривимірна решітка для сусідніх ПЕ (без глобальної мережі).

Мова програмування: Fortran-Plus (на основі Фортран 90).

Система DAP з оглядом на її архітектуру може застосовуватися тільки для деяких видів задач; аналіз масивно паралельних розв'язань задач за допомогою DAP наведено в праці [Parkinson, Litt 90]. DAP має лише 4K процесорних елементи, що замало порівняно з іншими системами. ПЕ, як і в CM-2, – це прості однорозрядні обчислювальні елементи, але вони доповнюються восьмирозрядними співпроцесорами. Це дає змогу DAP, незважаючи на відносно малу кількість ПЕ, досягати помітно високої продуктивності в області MFLOPS. В DAP застосовується лише локальна решітка зв'язку між сусідніми процесорними елементами, тут немає глобальної комутаційної мережі. У зв'язку з цим DAP має обмеження в побудові обчислювальних структур: хоч для цілого ряду задач локальна решітка підходить, все ж довільні зв'язки між ПЕ вона може забезпечити лише покроково з великими втратами часу, моделюючи передачі інформації (при наявності n ПЕ треба зробити $\sqrt{n}$ кроків обміну даними для одного з'єднання між довільними ПЕ; ці дані наведено в п.5.4). Такий обмін даними досить важко програмувати. Таким чином, задачі, що потребують складних структур зв'язку між ПЕ, можуть реалізуватися на DAP лише з великими втратами ефективності.

У цій системі користувачам пропонується лише одна мова програмування, а саме паралельна версія Фортрану на основі Фортран-90. Варіант мови C*, аналогічний

застосованому в Connection Machine CM-2, розробляється, але, природно, залишається відкритим питання, скільки функціональних можливостей C* можна буде реалізувати на примітивній структурі зв'язку системи DAP.

На завершення можна сказати, що поряд з різноманітністю структур розглянутих тут SIMD-систем помітні і їхні деякі загальні властивості. Всі фірми-виробники пропонують версії паралельного Фортрану, що базується на новому стандарті мови Фортран-90, але ці версії з різних причин несумісні. Використання Фортрану може бути пояснено бажанням авторів SIMD-систем залучити до кола користувачів фахівців із традиційних Фортран-областей інженерних та природничих наук. Однак проблема полягає в тому, що програмні пакети, написані на мові Фортран-77, не можуть бути автоматично розпаралелені, вони потребують перепрограмування заново в мові Фортран-90 (див. гл. 16).

Усі розглянуті SIMD-системи мають щонайменше двовимірну решітку як засіб швидкого зв'язку між процесорними елементами. Ці решітки забезпечують застосування SIMD-систем лише в деяких областях, як, наприклад, розпізнавання образів та вирішення проблем обчислювальної математики. Якраз впровадження більшої частини SIMD-систем в цих областях зумовлює наявність цих структур зв'язку.

11.2. Паралельність даних

На відміну від MIMD-систем в SIMD-системах завжди функціонує одна програма в центральній керуючій ЕОМ, інструкції якої виконуються послідовно, але одночасно багатьма процесорними елементами, тобто паралельно за даними (векторно на процесорних елементах). Програмування внаслідок цього дуже полегшується завдяки тому факту, що існує тільки один

керуючий потік і немає асинхронно виконуваних незалежних процесів.

Та обставина, що всі процесорні елементи обчислюють свої дані за єдиною програмою в "одному такті" (тобто певною мірою синхронізуються на кожному кроці дискретизації), робить непотрібними такі дорогі і пов'язані з можливими помилками механізми синхронізації, як семафори або монітори. SIMD-програми обходяться без цих концепцій синхронізації. Безумовно, процесорні елементи обмінюються даними, але, як детальніше пояснюється далі, тут йдеться не про синхронізацію двох ПЕ для одного обміну даними: тут відбувається колективний обмін даними між всіма ПЕ або в межах групи ПЕ. У той час як у багатьох MIMD-системах обмін даними є вузьким місцем, в SIMD-системах він виконується майже абсолютно паралельно і потребує в системах з решітками зв'язку сусідніх ПЕ такого самого часу, як і на виконання однієї арифметико-логічної команди. Хоч ця швидкість і не забезпечується в SIMD-системах, що мають глобальні структури зв'язку з вільним вибором процесорних конфігурацій, все ж таки операції обміну даними в цілому виконуються значно швидше, ніж в MIMD-системах і в них беруть участь одночасно тисячі процесорних елементів.

Порівняно з класичною ЕОМ фон Ноймана, де тільки один активний пристрій (CPU) виконує операції для великої кількості пасивних елементів (чарунки пам'яті), в SIMD-системах співвідношення між активними і пасивними елементами більш збалансоване. Кожний елемент даних деякого великого блоку даних, який розподілено між процесорними елементами, розміщується тут в локальній області пам'яті деякого ПЕ, тобто всі елементи даних можуть виступати як пристрої, що активно працюють. Це робить можливим (а також вимагає) зовсім новий стиль програмування, в якому операції над елементами даних виконуються безпосередньо паралельно без послідовного завантаження, обробки в CPU (ЦПП) компонентів числового масиву та запису в пам'ять.

Паралельність даних, притаманна SIMD-системам, потребує нового мислення, відходу від моделей ЕОМ фон Ноймана, до якої звик програміст останніх десятиліть. В принципі паралельність даних простіша, бо вона дає можливість вирішити проблеми, що мають природну паралельність, без огляду на штучні обмеження моделі фон Ноймана.

11.3. Віртуальні процесори

Презентовані в п.11.1 SIMD-системи характерні наявністю величезної кількості процесорних елементів, але все ж можуть траплятися задачі, для вирішення яких фізично наявних 64К (65536) процесорів замало (наприклад, якщо деяке зображення має оброблятися в 500*500=250000 пунктах і в ідеалі кожному пункту має належати свій ПЕ). В цьому випадку програміст мусить розмістити на наявних фізичних ПЕ всі віртуальні ПЕ, що потрібні для розв'язання задачі. Дуже бажано, щоб цю операцію, що досить часто зустрічається на практиці, виконувало програмне середовище або сам паралельний комп'ютер.

Таким чином, якщо кількість ПЕ, що потребує деяка програма, перевершує кількість наявних ПЕ, то в системі мають бути надані програмісту віртуальні ПЕ в необхідній кількості на деякому абстрактному рівні. SIMD-система за допомогою апаратури або програм має відображати віртуальні ПЕ на фізичні. Концепція віртуальних процесорів аналогічна концепції віртуальної пам'яті. Якщо деяка програма потребує менше віртуальних ПЕ, ніж їх фізично є, то залишки ПЕ треба просто виключити з роботи. Вони будуть в неактивному стані і за концепцією SIMD-моделі не зможуть в даній задачі використовуватися за якимось іншим призначенням. Якщо, навпаки, програма потребує більше віртуальних процесорів, ніж має SIMD-

система, то віртуальні ПЕ мусять бути відображені на фізичні ПЕ за декілька ітераційних кроків, як це показано нижче на прикладі.

Приклад для відображення віртуальних ПЕ

Припустімо, що: 2500 віртуальних ПЕ потребує програма;
1000 фізичних ПЕ в наявності.

Розв'язання за допомогою ітераційних кроків показано на рис.11.2

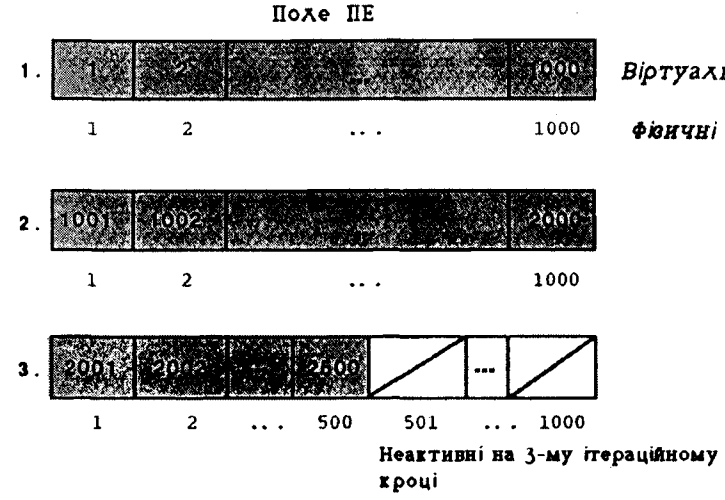


Рис. 11.2. Ітерації груп фізичних ПЕ

Ця ітерація здійснюється для кожної елементарної операції (наприклад, складання) або для послідовностей команд, в яких не відбувається обміну даними. Тут виникають втрати паралельності за рахунок неактивних ПЕ, якщо кількість віртуальних ПЕ не ділиться без остачі на кількість фізичних ПЕ. Так, у наведеному прикладі в кожній третій ітерації неактивними будуть процесорні елементи з номерами від 501 до 1000 (відповідно 50%), тобто за рахунок цих "відходів" система буде мати

заздалегідь оцінені втрати обчислювальної потужності близько 17% від можливої. Максимальне теоретичне завантаження фізичних ПЕ не може бути більше ніж 83%. Якщо V–кількість віртуальних ПЕ, а P–кількість фізичних ПЕ, то коефіцієнт віртуалізації визначається як R=V/P.

Рис.11.3 показує характеристики часу виконання і завантаження ПЕ залежно від коефіцієнта віртуалізації R для деякої гіпотетичної прикладної програми, яка містить у собі виключно векторні операції (без скалярних команд або неактивних ПЕ).

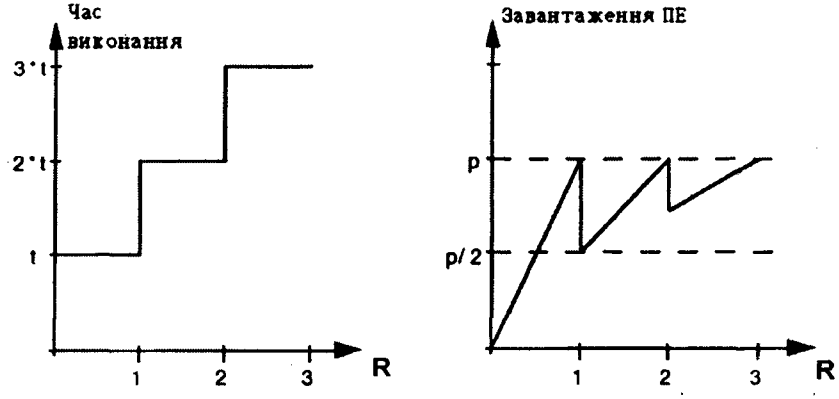


Рис. 11.3. Тривалість виконання програми та завантаження процесорів при збільшенні кількості віртуальних ПЕ

Ітеративне відображення віртуальних ПЕ на фізичні суттєво ускладнює обмін даними. Якщо уявити, що з простої за структурою, але великої за розмірами віртуальної решітки треба забезпечити зв'язок з малою решіткою, що є часткою великої, то комплексна структура зв'язку повинна мати на межах малої решітки засоби зв'язку з багатьма фізичними процесорами, а це, природно, уповільнює операції обміну даними. Дані, що обчислюються фізичними процесорами відповідно до ітераційного відображення віртуальних ПЕ, мають запам'ятовуватися в буферній пам'яті, ємкість якої

відповідає кількості віртуальних ПЕ, тобто в кожному фізичному ПЕ пам'ять має бути віддана в розпорядження декількох віртуальних ПЕ.

Відображення віртуальних ПЕ на фізичні має відбуватися непомітно для програміста. Це означає, що воно може реалізовуватися за допомогою спеціальної апаратури або тільки програмним шляхом.

Апаратна реалізація віртуальних процесорів

Система CM-2 (Connection Machine) має апаратну підтримку віртуальних процесорних елементів. Співвідношення між віртуальними і фізичними ПЕ може задаватися безпосередньо. Головна пам'ять кожного ПЕ розподіляється автоматично між декількома віртуальними ПЕ, і кожна команда виконується для всіх віртуальних ПЕ, тобто багаторазово (ітеративно) на фізичних ПЕ. Це апаратне рішення має цілий ряд переваг. Гарантується ефективна обробка, а самі системні програми (наприклад, компілятор) можуть бути виконані значно легше, ніж без спеціальної апаратури. Недолік полягає у збільшенні системних втрат у випадках, коли віртуалізація процесорів не використовується. Якщо 64K фізичних ПЕ (ця кількість завжди переконує) забезпечують розв'язання задачі, то за рахунок невикористовуваних ресурсів керування загальна продуктивність падає майже на 25%.

Програмна реалізація віртуальних процесорів

Система MasPar MP-1 не має апаратної реалізації віртуальних процесорів. Тут використовується програмне рішення за допомогою "інтелігентного компілятора". Залежно від того, потребує прикладна програма віртуалізації процесорів чи ні, компілятор має генерувати спеціальні коди для відображення віртуальних ПЕ на

фізичні або видавати програмні коди без віртуалізації. В останньому випадку процесорний час не буде витрачатися на керування процесами віртуалізації. Хоч прикладний програміст не повинен помічати різниці між апаратним та програмним розв'язанням задачі віртуалізації, системне програмування (наприклад, розробка нового компілятора) для програмного способу віртуалізації стає помітно важчим. Потрібні для віртуалізації витрати на керування і одержувані при цьому заплановані показники обчислювальної продуктивності значною мірою залежать від компілятора.

На сьогодні віртуалізація як програмне рішення в MasPar MP-1 реалізована у вигляді додаткових MPFortran-програм. Для MPL ще не існує "інтелігентного компілятора" і це означає, що задачу віртуалізації повинен вирішувати програміст-користувач.

12. СПІЛКУВАННЯ В SIMD-СИСТЕМАХ

У зв'язку з тим, що в SIMD-системах усі процеси протікають синхронно, тут немає потреби в тій синхронізації процесорів, яка була описана для MIMD-систем. Вище вже було показано, що обмін даними в SIMD-системах є "колективним" явищем. Це означає, що тут не відбувається обміну між якимись двома процесорними елементами (ПЕ), тут всі активні ПЕ беруть участь в обміні даними. Це можуть бути або всі ПЕ загальної системи, або тільки деяка частина їх. В будь-якому з цих випадків обмін даними в SIMD-системах є простішою і дешевшою операцією порівняно з MIMD-системами, бо побудова синхронного зв'язку може бути швидше реалізована,

комутаційна мережа має в більшості випадків також вищу контактоспроможність і ширшу частотну смугу пропускання. На рис.12.1 показано один з прикладів обміну даними. Процесорні елементи розміщені у вигляді прямокутника 3x4 або ПЕ-матриці, причому кожний ПЕ має координати [i,j], i=1,2,3 (номер рядка знизу вгору), j=1,2,3,4 (номер стовпця зліва направо). Кожний ПЕ передає значення своєї локальної змінної ***x*** направо до сусіднього ПЕ (зрушення числа вправо на один крок решітки). У зв'язку з тим що більшість SIMD-систем працюють з дуже швидкою, постійно заданою структурою зв'язку, можна передусім відобразити прості регулярні структури на реальну фізичну структуру зв'язку між ПЕ. Спроби виконати інші види зв'язку, що відрізняються від заданої структури, коштують помітних втрат продуктивності системи, бо в цих випадках кожний обмін даними мусить бути виконаний за декілька кроків або через повільну глобальну структуру зв'язку (якщо вона є в системі).

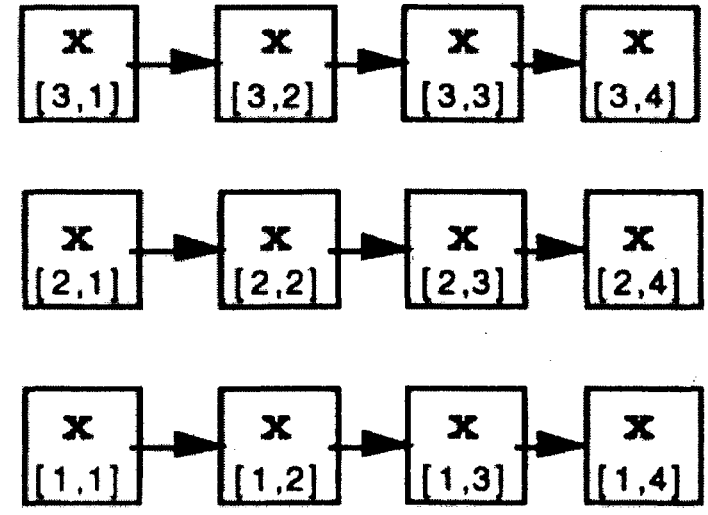


Рис.12.1. Обмін даними в SIMD-системах

12.1. SIMD-обмін даними

Обмін даними в SIMD-системах виконується в три этапи.

1. Виділення (селекція) деякої групи процесорних елементів (активізація).
2. Вибір заздалегідь визначеного напрямку зв'язку між ПЕ (або ж динамічне визначення структури зв'язку) .
3. Проведення обміну даними попарно між всіма активними ПЕ відповідно до вибраної структури зв'язку.

В SIMD-мові програмування PARALLAXIS [Bräunl 90] подібний обмін даними описується такими мовними засобами:

0. Підготовчий етап: визначення (описування) структури зв'язку

Приклад:

```
CONFIGURATION ring [0..11];
CONNECTION rechts: ring[i]↔ ring[(i+1) mod 12].links;
```

Структура зв'язку на ім'я "ring" (кільце) має 12 елементів, яким надані номери від нуля до одинадцяти. Система має повний двосторонній (бі-дирекціональний) зв'язок із символічними іменами напрямків зв'язку "направо" і "наліво" (rechts, links). Кожний ПЕ-і реалізує зв'язок направо із сусіднім ПЕ, що має наступний за даним номер ідентифікації, тобто ПЕ-(i+1); елемент ПЕ-11 за допомогою функції MODULO (mod 12) з'єднується з ПЕ-0. Так само виконується зв'язок наліво. Таким чином, структура зв'язку дає замкнене кільце.

1. Селекція групи ПЕ

Приклад:

```
PARALLEL ring[3..8]
   ...
ENDPARALLEL
```

За допомогою цього паралельного блоку команд (PARALLELEL, ENDPARALLELEL) з усіх ПЕ, що включені в кільце, виділяються процесорні елементи з номерами від 3 до 8 (ring[3..8]), вони стають активними, а всі інші залишаються пасивними.

2.+3. Виконання паралельного обміну даними (всередині паралельного блоку)

Приклад:

```
PROPAGATE.rechts(x)
```

Кожний активний процесорний елемент, тобто ПЕ-3, ПЕ-4, ПЕ-5, ПЕ-6, ПЕ-7, ПЕ-8, проводить обмін даними з сусіднім ПЕ, розміщеним за ходом стрілки годинника. Спочатку кожний ПЕ посилає значення своєї локальної величини x в лінію зв'язку (передача, "send", рис.12.2 зліва), а після закінчення читає значення своєї локальної змінної x від свого ПЕ-попередника (прийом, "receive", рис.12.2 справа). Внаслідок активізації тільки частини процесорних елементів крайні ПЕ мають особливості функціонування під час обміну даними: ПЕ-3 не має ПЕ-попередника для приймання значення x, а ПЕ-8 не має ПЕ-послідовника для передачі йому значення x. Така сама ситуація має місце у відкритих топологіях, наприклад у кінцевих пунктах лінійного списку.Показані на рис.12.2 процесорні елементи після обміну даними мають такі результати: ПЕ-3 зберігає старе значення x, бо він не отримує від іншого ПЕ даних, а ПЕ-8 втрачає своє старе значення x, бо воно не сприймається і не запам'ятовується ПЕ-послідовником.

Наведена на рис.12.2 кільцева структура зв'язку між ПЕ може бути дуже просто перетворена в найчастіше

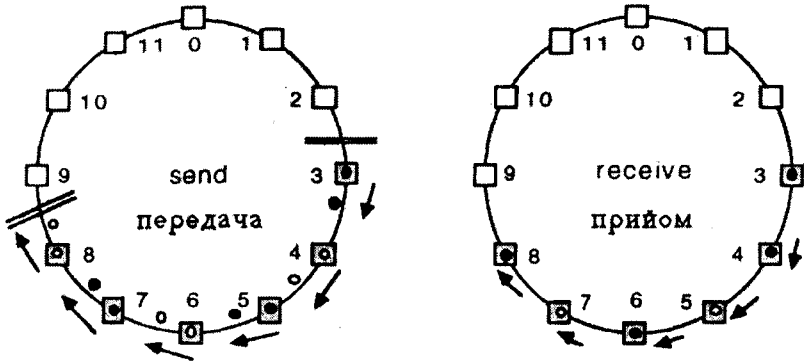


Рис. 12.2. Прийом та передача даних

застосовувану фізичну структуру зв'язку, а саме – двовимірну решітку (рис.12.3). При цьому для деяких ПЕ фізичної решітки мають виконуватися спеціальні правила. В той час як внутрішні елементи сітки можуть за один крок з'єднатися в напрямку другої координати, праві кінцеві елементи решітки мають з'єднуватися наліво вгору, а останній елемент кільця N11 має бути з'єднаним з першим елементом кільця N0. На рис.12.3 використовуються тільки ті ПЕ решітки, які позначені сірим кольором.

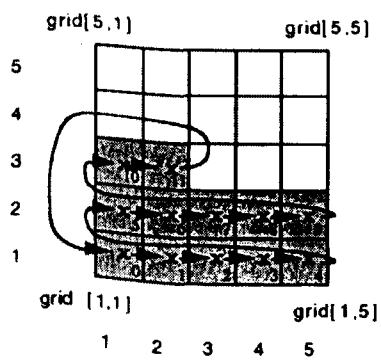


Рис. 12.3. Відображення кільця в решітці

Для простого на перший погляд обміну даними вздовж віртуальної кільцевої структури треба провести дорогі за витратами часу операції обміну у фізичній решітчатій структурі. Програмна конструкція

```
PARALLEL ring [0..11]
        PROPAGATE.rechts(x)
ENDPARALLEL,
```

що відповідає обміну в структурі рис. 12.2, перетворюється на таку послідовність операцій:

Операція A: Один крок направо

```
PARALLEL grid [1..2], [1..4]
            grid [3],    [1]
        "grid[i, j] → grid[i, j+1]"
ENDPARALLEL;
```

Операція B: Початок рядка, що розміщений на один крок вище

```
PARALLEL grid [1..2], [5]
        "grid[i, j] → grid[i+1, 1]"
ENDPARALLEL;
```

Операція C: Назад до початку кільця, тобто ПЕ [1,1]

```
PARALLEL grid[3, 2]
"grid[3, 2] → grid[1, 1]"
 ENDPARALLEL;
```

У зв'язку з тим, що розпізнавання операцій A, B, C в SIMD-системі має реалізуватися послідовно, у цьому прикладі потрібно здійснити три фізичних операції обміну даними на один віртуальний обмін. Крім цього, операції призначення мають виконуватися через буферну область, бо інакше локальні дані в ПЕ вже були б переписані до того, як вони пересилаються в сусідні ПЕ.

Автоматичне відображення віртуальних комутаційних структур на постійну фізичну структуру (як було

показано вище, на решітчасту структуру) вкрай важко реалізувати, у чому можна переконатися з наведеного вище прикладу. Чим складніша бажана віртуальна структура, тим важчою стає ця задача. У зв'язку з цим автоматичне відображення може проводитися тільки для простих структур. Для складних структур воно не може визнатися доцільним з огляду на зростаючу кількість варіантів і тривалість їх виконання. Тут застосовується глобальна динамічна структура зв'язку SIMD-системи ("Router"), якщо така є в системі. Визначаючи для кожного фізичного ПЕ цільову адресу деякого ПЕ, можна розглядати загальний випадок неструктурованого зв'язку як перестановку векторів (рис.12.4).

Застосування глобальних структур зв'язку набагато легше програмувати і воно може без проблем автоматизуватись. Одначе глобальні структури значно повільніші, ніж локальні (решітчасті). Правильне застосування локальних або глобальних структур зв'язку – це важлива проблема SIMD-програмування.

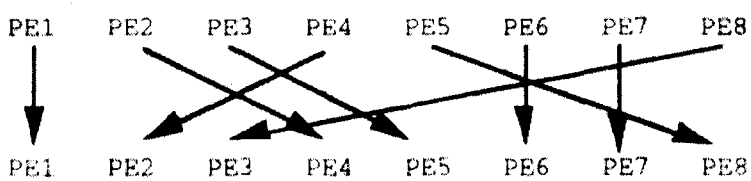


Рис. 12.4. Обмін даними як пермутація вектора

Ще значно важчою стає проблема спілкування процесорів у разі застосування віртуальних процесорних елементів (див. п.11.3, 13.2). Внаслідок фрагментного відображення віртуальних ПЕ на фізичні ПЕ навіть із простого структурованого зв'язку між ПЕ може виникнути деякий складний і неструктурований зв'язок. Поряд з цим ускладнюється автоматичне відображення віртуальних структур зв'язку на фізичні структури з ефективним

обміном даними, бо воно потребує області буферної пам'яті для всіх віртуальних ПЕ.

12.2. Структури зв'язку SIMD-систем

У цьому параграфі розглядаються в деталях структури зв'язку паралельних SIMD-систем CM-2 (Connection Machine) і MasPar MP-1 (система DAP не має в цьому розпорядженні глобальної комутаційної мережі). Як уже згадувалось, обидві названі системи мають окрім локальної решітчастої структури також глобальну комутаційну мережу, що дає можливість довільно вибирати варіанти зв'язку між процесорними елементами (ПЕ). Обидві комутаційні решітки – це значно швидші структури зв'язку (наприклад, придатні для вирішення проблем розпізнавання зображень), в той час як повільна глобальна комутаційна мережа може реалізувати довільні динамічні зв'язки між ПЕ.

Connection Machine CM-2 з 65336 процесорами (64K ПЕ)

1. Локальна решітчаста структура системи CM-2 – це динамічно керована решітка з чотирикратним зв'язком із сусідніми ПЕ (решітка називається "NEWS" за початковими літерами географічних напрямків north (північ), east (південь), west (захід), south (схід), англ. 4-way nearest neighbor). Решітка може з деякими обмеженнями мати багатовимірну структуру і велику кількість вузлів (ПЕ, що з'єднуються). Ця швидка локальна структура зв'язку реалізується як частина глобального гіперкубічного зв'язку (див. п.2) із спеціальною апаратною

підтримкою. На рис.12.5 подано ескіз двовимірного зв'язку між ПЕ (з 256x256 ПЕ).

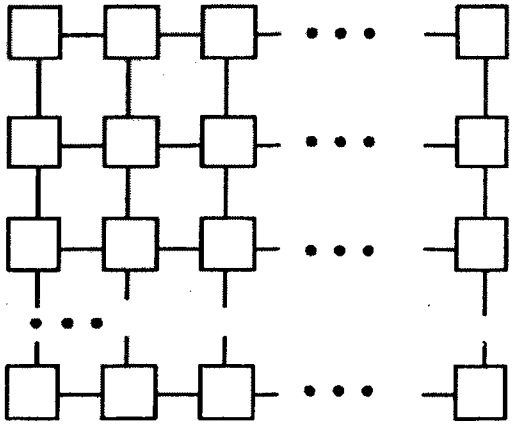


Рис. 12.5. Решітчаста комутаційна мережа системи Connection Machine CM-2

2. Як глобальна структура зв'язку в CM-2 використовується 12-вимірний гіперкуб. Він містить у собі всього 2^{12} = 4096 елементів, так що загальна кількість 65536 процесорних елементів має розподілитися в 4096 кластерах (групах ПЕ), а в кожному кластері має бути 16 ПЕ. На рис. 12.6 показано гіперкубічну комутаційну мережу SIMD-системи CM-2 у спрощеному вигляді: кластер-структуру з огляду на недостатню кількість місця показано з чотиривимірним гіперкубом. Від кожного кластера йдуть 12 ліній зв'язку в гіперкуб, а 16 процесорних елементів всередині кластера зв'язані безпосередньо через маршрутизатор (Router), що реалізований на мікросхемі. Якщо всі ПЕ в кластері потребують передачі або прийому даних за межами кластера, то загальний обмін реалізується за 16 кроків (кластер є вузьким місцем системи).

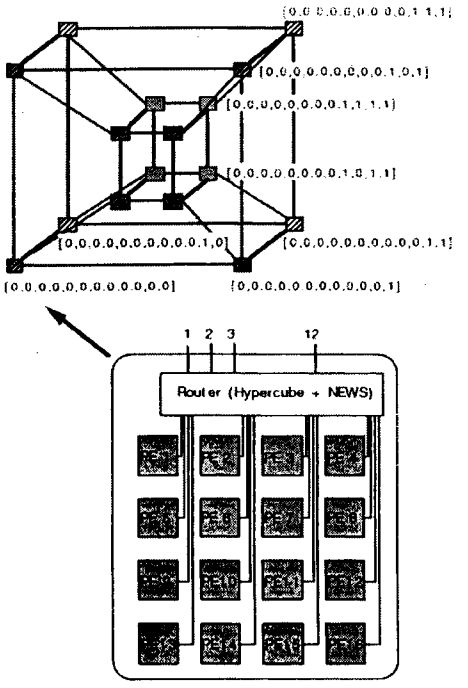


Рис. 12.6. Комутаційна мережа ГІПЕРКУБ системи CM-2

Конструктивне кластер-рішення дещо обмежує глобальну структуру зв'язку системи CM-2, але воно дає змогу побудувати паралельну обчислювальну систему з 64К процесорами із прийнятними витратами на лінії зв'язку.

Реалізована в CM-2 структура потребує в цілому:

12*4096*12 = 24576 ліній зв'язку для гіперкуба;
65536*1 = 65536 ліній зв'язку для ПЕ як кластерних елементів;

Сума = 90112 ліній

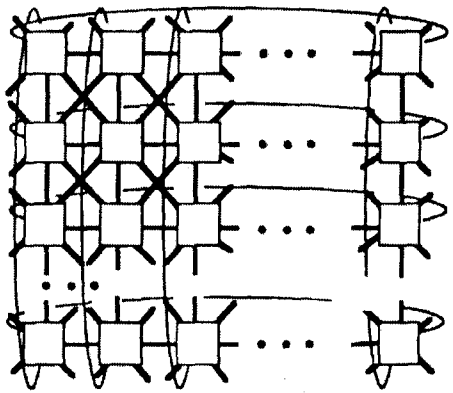
Порівняно з цим рішенням для реалізації повного 16-вимірного гіперкуба потрібно було б значно більше ліній зв'язку:

1/2*65536*16 = 524 288 ліній

Це було б майже в шість разів більше, ніж у розглянутому кластерному рішенні, і значно дорожче!

MasPar MP-1, модель MP-1216 з 16384 процесорами

1. Локальна решітчаста структура системи MP-1 реалізована у вигляді постійного двовимірного тора, в якому розміщено 128x128=16384 процесорних елемента (рис.12.7).



(Без діагональних зв'язків з краю)

Рис. 12.7. Решітчаста комутаційна мережа MasPar MP-1

Кожний ПЕ має зв'язок з вісьмома сусідніми ПЕ. Ця решітка має назву восьмивимірної ***x***-мережі за ***x***-подібністю напрямків зв'язку з діагональними ПЕ-сусідами (англ. 8-way nearest neighbor, "***x***-net").

2. Глобальна комутаційна структура системи MP-1 називається "глобальним маршрутизатором" і складається з триступеневої комутаційної мережі Клоса з 1024 входами і виходами (рис.12.8). Це означає, що кожні 16 ПЕ, об'єднані в кластер, можуть користуватись одним входом і одним виходом маршрутизатора. За цих обставин із 16384 ПЕ тільки одна шістнадцята частина їх, тобто 1024 ПЕ, можуть одночасно побудувати схему зв'язку між собою до деякого загального моменту. Якщо всі процесорні елементи потребують одночасного обміну даними, то на це знадобиться щонайменше 16 тактів, якщо не виникне нових колізій.

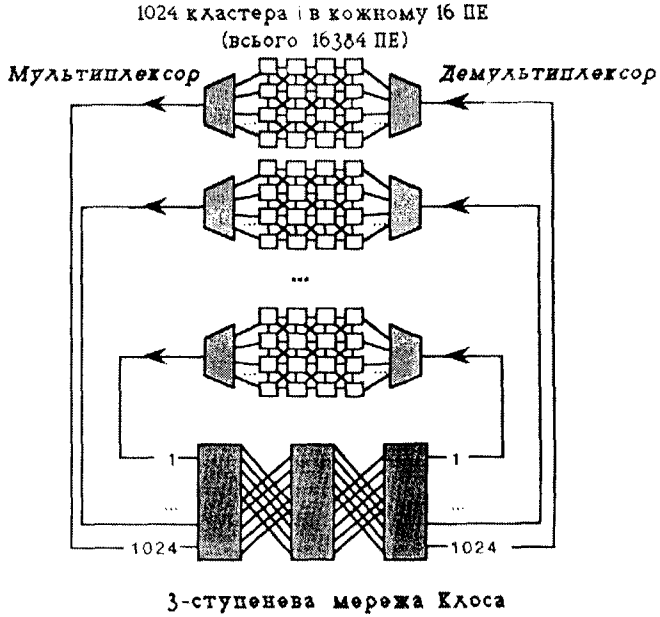


Рис. 12.8. Роутер системи MasPar MP-1

Таким чином, для побудови глобальної мережі зв'язку в MP-1 вибрано такий самий компроміс, як і в СМ-2. Витрати на зв'язок для n входів/виходів триступеневої мережі Клоса оцінюються приблизно величиною $\sqrt{32} * n^{3/2}$ (див.п.5.2). Повна комутаційна мережа Клоса для всіх ПЕ мала б у 16 разів більший розмір і в 64 рази більші витрати. Перехресний розподілювач шин

(витрати n*n) з розміром 1024 був би дорожчим в 5,7 раза, а розподілювач для 16384 ПЕ – дорожчий у 1448 разів.

12.3. Редукція вектора

Однією з основних операцій векторних ЕОМ і SIMD-систем є операція редукції вектора. В більшості систем вона представлена як апаратне рішення або як базова операція. Операція переводить деякий вектор (тобто компоненти вектора, розміщені в окремих ПЕ) у визначену скалярну величину (див. п.2.3). Це може бути зроблено за допомогою будь-якої діадної операції – складання, множення, вибір максимуму або мінімуму, логічне "і", логічне "або" тощо. Одначе треба мати на увазі, що операція редукції є асоціативною та комутативною (тобто забороняється віднімання і ділення). Нехтування цим зауваженням може призвести до результатів, що залежать від порядку виконання операції редукції.

Як показано на рис.12.9, вектор складається покомпонентно, а отримана сума – це скалярна величина.

Рис. 12.9. Редукція вектора операцією суми

Порядок виконання операцій, що визначається розміщенням дужок, в асоціативних операціях може бути довільним, але слід зазначити, що завдяки деревоподібній послідовності операцій паралельне виконання може бути застосоване ефективніше, ніж послідовне (рис.12.10, рис. 12.11). Так, під час послідовного складання n чисел треба виконати n–1 крок. У разі деревоподібної послідовності операцій є можливість розподілити ту саму кількість складань між процесорами, розташованими на відповідних рівнях дерева, і вся операція потребує лише $\log_2 n$ тактів.

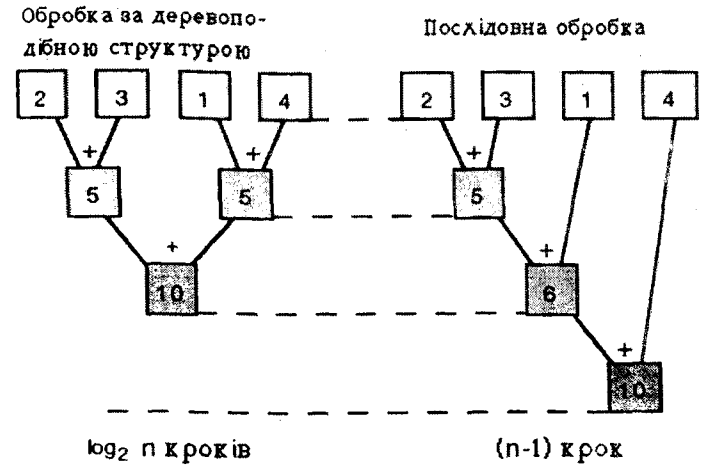


Рис. 12.10. Редукція вектора

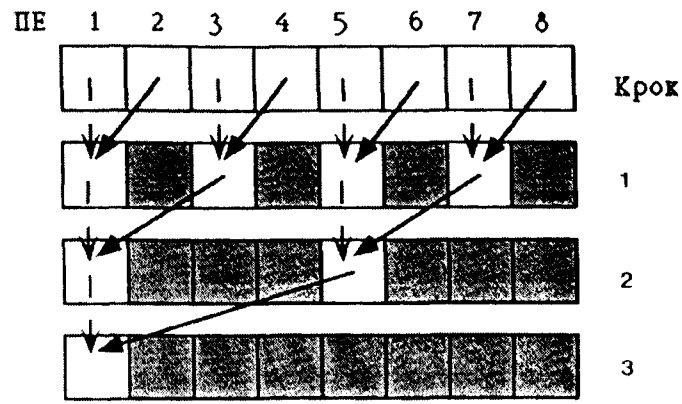


Рис. 12.11. Виконання редукції вектора

На мові PARALLAXIS редукція вектора V в скаляр S за допомогою операції SUM записується так:

```
S := REDUCE.sum(V)
```

Імплементація редукції вектора виконується за допомогою програмних засобів методом явного обміну даними з арифметичними операціями, що виконуються

після редукції, або за допомогою спеціальної апаратури (як, наприклад, в системі СМ-2). Апаратне рішення забезпечує виконання операції редукції паралельно з передачами даних.

13. ПРОБЛЕМИ СИНХРОНОЇ ПАРАЛЕЛЬНОСТІ

Ніяка з проблем асинхронної паралельності, що розглянуті в гл.8, не має відношення до синхронної паралельності. Тут не існує ні несумісних даних, ні блокувань, також (за системними умовами) не може застосовуватися балансування завантаження процесорів. Це знову підкреслює принципову різницю між синхронним та асинхронним паралельним програмуванням. Обидві ці моделі паралельності базуються на різних основних положеннях і мають свої різні області застосування. Тому асинхронні паралельні рішення різних проблем не можуть бути перетворені в синхронно-паралельні рішення і навпаки.

Проблеми, що виникають під час реалізації синхронної паралельності, пов'язані в основній своїй частині з обмеженнями, що притаманні SIMD-моделі. У зв'язку з тим що всі процесорні елементи мають виконувати одну й ту саму операцію або бути в неактивному стані, деякі векторні операції можуть розпаралелюватися недостатньо. Застосування рівня абстракції для керування процесорами незалежно від фактично наявної кількості фізичних процесорів також може призвести до проблеми ефективності.

Ще однією проблемою є використання периферійної апаратури, яка дуже часто стає вузьким місцем під час передачі даних. Детальніше ці проблеми буде розглянуто нижче.

13.1. Індексовані векторні операції

Терміни Gather (збирання, операція зчитування) і Scatter (розкладання, запис в пам'ять) характеризують дві основні векторні операції [Quinn 87], синхронна паралельна реалізація яких пов'язана з труднощами. В принципі виникають проблеми під час векторизації індексованих доступів до даних.

Дані кожного вектора розподіляються по-компонентно між процесорними елементами. Операція Gather звертається до вектора через деякий індексний вектор з метою зчитування, тоді як операція Scatter звертається до вектора через деякий індексний вектор з метою запису. Функціональність цих операцій відображена в наступному фрагменті псевдопрограми:

Gather:
```
for i:=1 to n do
   a[i] := b[index[i]]
end;
```

Scatter:
```
for i:=1 to n do
   a[index[i]] := b[i]
end;
```

У кожному з цих двох випадків йдеться про залежний від даних доступ до компонентів другого вектора, тобто здійснюється неструктурована перестановка векторів або, інакше кажучи, відбувається довільний доступ до даних інших процесорів. Цей вид доступу, природно, не може бути розпаралелений за допомогою типової структури зв'язку між ПЕ-сусідами. Послідовне виконання цієї важливої операції також небажане.

Цілий ряд векторних ЕОМ вирішують цю проблему за допомогою спеціальної апаратури, а всі інші потребують дорогих за витратами часу програмних рішень. В масивно паралельних системах індексовані доступи Gathen і Scatter можуть виконуватися через повільніші, але універсальні структури зв'язку – маршрутизатори, якщо вони є в системі. В інших випадках – лише послідовне виконання. Деякі

SIMD-системи мають проблеми навіть з векторною індексацією локальних масивів в процесорних елементах.

Приклад:

```
SCALAR s : INTEGER;
VECTOR a : ARRAY [1..10] OF INTEGER;
       u,v : INTEGER;
...
u := a[s]; (* скалярне індексування *)
u := a[v]; (* векторне індексування *)
```

У кожній із двох інструкцій векторній змінній u присвоюється елемент векторного масиву a. Всі SIMD-системи допускають скалярну індексацію локального масиву, в якій всі ПЕ використовують однаковий індекс, бо інакше векторні масиви були б неможливими. Векторна адресація, в якій індекси можуть бути різними в різних ПЕ, в системі СМ-2 з апаратних причин неможлива, тоді як в MasPar MP-1 вона реалізується без труднощів.

13.2. Відображення віртуальних процесорів на фізичні процесори

Роль віртуальних процесорів для SIMD-систем була описана в п.11.3. Застосування цього рівня абстракції дає в розпорядження програміста-користувача як завгодно багато процесорів з довільною структурою зв'язку. Тільки таке абстрагування від фізичної апаратури робить можливою розробку машинно незалежних паралельних за даними програм.

Відображення віртуальних процесорів на фізичні з їхніми зв'язками відбувається непомітно для користувача за допомогою спеціальних апаратних засобів або "інтелігентного компілятора". При цьому треба вирішити ряд проблем.

- Віртуальні процесори мають бути рівномірно розподілені між наявними фізичними процесорами. Ця задача вирішується методом ітерацій спеціальною апаратурою або допоміжними кодами компілятора. Правильне розподілення полегшує побудову потрібних віртуальних зв'язків між процесорами.
- По можливості в разі наявності декількох комутаційних структур треба використовувати найшвидшу структуру зв'язку (здебільшого решітчасту структуру).
- Автоматичний пошук оптимального топологічного відображення віртуальних структур ПЕ на фізичні залежно від застосовуваної мови програмування і наявних мовних конструкцій може бути важким і навіть принципово неможливим. Незважаючи на те що для деяких типів зв'язку існують алгоритми відображення однієї структури зв'язку на іншу (п.5.4, [Siegel 79]), проблема автоматичної трансляції інструкцій щодо обміну даними або декларувань зв'язків (якщо вони наявні в SIMD-системі) в тип мережі зв'язку набагато важча.
- Якщо кількість потрібних віртуальних процесорів перевищує кількість наявних фізичних ПЕ, то виконання операцій обміну даними можливе тільки через велику буферну область пам'яті з помітними втратами часу.

Обмін даними між процесорами – це критична проблема під час віртуалізації, яка може бути вирішена з прийнятними витратами часу за умови, що існує загальна глобальна структура зв'язку між ПЕ (маршрутизатор). У зв'язку з тим, що кожному фізичному ПЕ можна поставити у відповідність декілька віртуальних ПЕ і всі вони беруть участь у виконанні однієї (віртуальної) команди на обмін даними, кожна операція спілкування процесорних елементів має виконуватися за декілька кроків.

13.3. Зменшення пропускної спроможності під час підключення периферійної апаратури

Найважливійшою якістю SIMD-системи є те, що в ній ліквідовано "горло пляшки фон Ноймана", тобто вузьке місце традиційної послідовної однопроцесорної ЕОМ. Завдяки усуненню нерівноваги між одним процесором, що активно працює (центральний процесорний пристрій), і великою кількістю пасивних елементів (елементи пам'яті), а також введенню високопродуктивних комунікаційних мереж для паралельних зв'язків між процесорами дані в SIMD-системах можуть передаватися дуже швидко; тут більше не існує "горла пляшки фон Ноймана".

На жаль, у разі підключення до SIMD-системи периферійної апаратури, наприклад для доступу до великих масивів даних, дуже легко може виникнути нове "горло пляшки" (рис.13.1).

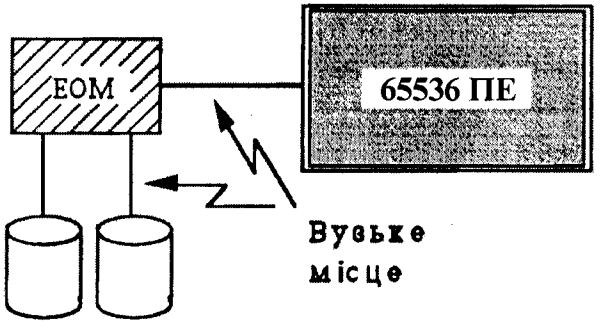


Рис. 13.1. Работа периферійних пристроїв в масивно паралельних системах

Тут виникають дві основні проблеми. По-перше, це зв'язок між паралельними процесорними елементами і центральним керуючим комп'ютером (HOST), а по-друге, це підключення периферійної апаратури, такої як дискова пам'ять або графічні дісплеї, які в найпростішому випадку вмикаються безпосередньо в HOST. Вузьке місце під час обміну даними з HOST не може бути ліквідовано SIMD-структурою. Тому прикладні паралельні програми мають будуватися так, щоб вони по можливості не переривалися на послідовний обмін даними з центральним HOST, особливо на запис та читання скалярних масивів даних. Ці операції треба проводити до початку і наприкінці виконання паралельних програм. Як видно з рис.13.1, вузьке місце під час підключення периферійної апаратури (на рис.13.1 дискової пам'яті) виникає тільки тому, що між цією апаратурою і паралельними ПЕ стоїть послідовний керуючий комп'ютер. Через нього дані мають проходити покроково, щоб досягти конкретного ПЕ за його адресою. Тут можна запобігти цій проблемі за допомогою паралельного підключення периферії. Це технічне рішення в системі CM-2 називається *Data Vault*, а в системі MasPar MP-1 відоме як *Parallel Disk Array*. В МП-1 паралельне підключення дискової пам'яті виконується через швидкодіючу шину, яка у свою чергу має зв'язок через маршрутизатор безпосередньо з процесорними елементами. Це означає, що всі ПЕ SIMD-системи можуть паралельно записувати (або зчитувати) дані в периферійну пам'ять. У зв'язку з тим, що пристрій дискової пам'яті (вінчестер) є послідовним елементом, система потребує розпаралелювання, що досягається підключенням багатьох механізмів дискової пам'яті на паралельну роботу (рис.13.2). Ця техніка забезпечує швидкий потік даних, обминаючи вузьке місце, яким є керуючий комп'ютер HOST.

Аналогічний масив дискової пам'яті – це так звана RAID-система (redundant array of inexpensive disks), яка пропонується для послідовних ЕОМ і забезпечує приріст швидкості обміну даними і збільшеня надійності роботи апаратури.

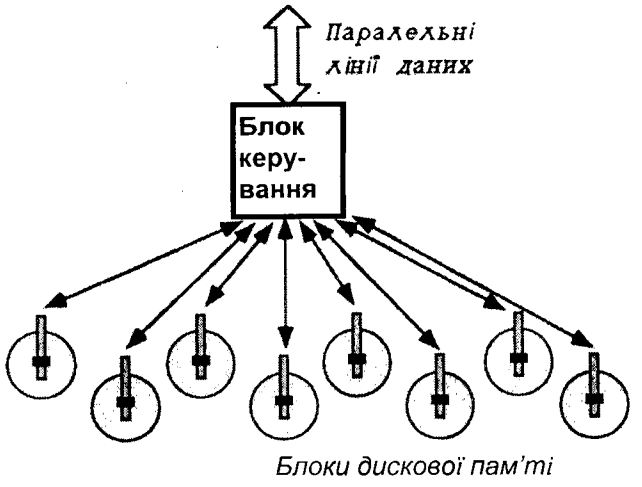


Рис. 13.2. Паралельна робота блоків ДП

13.4. Ширина частотної смуги комутаційних мереж

Пропускна спроможність комутаційної мережі SIMD-структури значною мірою визначає продуктивність всієї системи. З одного боку, витрати часу на побудову схеми зв'язку і проведення комунікації, а також масивно паралельного обміну даними значно менші, ніж в MIMD-системі. Швидкість виконання локального обміну даними між фізичними сусідніми ПЕ приблизно така сама, як і швидкість виконання однієї паралельної операції. З іншого боку, неструктуровані, глобальні операції обміну даними потребують для їхнього проведення значно більше часу. SIMD-системи, що мають локальну та глобальну комутаційні структури, характеризуються також відповідно двома різними величинами ширини смуги частот комутаційних мереж, вплив яких на продуктивність систем залежить від прикладної програми. В деяких програмах із складними структурами зв'язку вплив смуги пропускання комутаційної мережі дуже посилюється і може стати в

певних умовах критичним фактором. Тому ширина смуги пропускання має бути якомога більшою.

З огляду на обмежену смугу пропускання глобальної комутаційної мережі прикладні програми мають відповідати таким вимогам:

- треба уникати тих глобальних операцій обміну даними, без яких можна обійтись, бо вони значно дорожчі за витратами часу, ніж арифметичні команди;
- застосування структурованих топологій зменшує витрати на спілкування ПЕ, якщо для їхньої реалізації можливе застосування швидких локальних зв'язків між сусідніми ПЕ (навіть аж після трансляції в послідовність обміну даними, що виконуватиметься за декілька кроків).

Арифметико-логічні пристрої паралельних процесорів під час виконання більшості прикладних програм мають задовільну швидкість порівняно з глобальними комутаційними мережами. Помітно вищу продуктивність всієї паралельної обчислювальної системи, тобто вищу швидкість обчислень, можна забезпечити тільки за рахунок збільшення пропускної спроможності (смуги пропускання) комутаційної мережі.

13.5. Робота в режимі багатьох користувачів і толерантність до помилок

SIMD-системи ніяк не підходять для організації одночасної роботи багатьох користувачів. У нормальних умовах в SIMD-системі працює лише один керуючий комп'ютер (HOST), тому в кожний момент може виконуватися тільки одна програма. Паралельне виконання незалежних програм неможливе. Єдиною можливістю залишається квазі-паралельне виконання програм мультиплексуванням часу в HOST, але тут виникає ще

більша проблема великого об'єму пам'яті даних, яка розподілена локально між ПЕ (наприклад, в сумі 1 GB для MasPar MP-1, модель 1216). Ця величезна кількість даних не може, природно, в ті секундні частки часу, що виділяються на перемикання задач в режимі розподілу часу, записуватися на вінчестер і потім зчитуватися відтіля. Описана вище проблема підключення периферійної апаратури зумовлює те, що кожна операція читання або запису загальної паралельної пам'яті даних має при наявності дискової пам'яті потрібний час на виконання десь близько 10 с! Тому для паралельної пам'яті SIMD-системи концепція віртуальної пам'яті на сьогодні не може бути застосована.

Власне, якщо часові інтервали великі (як правило, приблизно 10 с), то завантаження та очищення паралельної пам'яті даних неможливі. Рішення полягає в тому, щоб наявну пам'ять кожного ПЕ розподілити між активними користувачами. Через недостатню пропускну спроможність обмежується кількість одночасно активних програм. Кожна програма замовляє до початку виконання кількість потрібної паралельної пам'яті (наприклад, 4 K в кожному ПЕ в системі MasPar MP-1). Якщо це місце в пам'яті системи може бути надано, то воно виділяється програмі, яка може виконуватися за способом мультиплексування часу. Але у випадку, коли бракує потрібної кількості місця в пам'яті, програма має чекати на завершення інших програм, що в даний час активні.

Ще одна проблема – це толерантність SIMD-систем до помилок. У разі виходу з ладу навіть одного єдиного ПЕ здебільшого немає можливості вирішити цю проблему програмним способом. В цьому випадку може зарадити (крім заміни дефектної плати процесорів) тільки зменшення конфігурації процесорів наполовину. Наприклад, MasPar MP-1, що має 16384 ПЕ, у разі виходу з ладу одного ПЕ може конфігуруватися лише як система з 8192 ПЕ. Одначе для цього ще треба вручну вставити в систему відповідну плату маршрутизатора, якою доповнюється комутаційна

решітка зменшеного тора.

14. SIMD-МОВИ ПРОГРАМУВАННЯ

Як і в асинхронному паралельному програмуванні, у синхронному програмуванні розглядаються тільки процедурні мови (до непроцедурних мов повернемося в гл.17). На рівні абстракцій процедурного програмування є сенс чітко відрізняти в паралельному програмуванні специфіку MIMD- та SIMD-систем. У зв'язку з тим, що SIMD- і MIMD-програми відрізняються алгоритмічно, розробка деякої "універсальної" паралельної мови програмування процедурного типу приречена на невдачу. Кожна програма могла б бути ефективно виконана тільки на одній із систем обох класів навіть у тому випадку, якби концепції їх побудови застосовувалися сумісно. Можливо, що на рівні абстракції, вищому за рівень процедурного програмування, робити таку чітку різницю між SIMD та MIMD не буде потреби. Наприклад, за допомогою непроцедурних мов можна реалізувати паралельність так, щоб користувач не помічав, на якій системі це зроблено.

Розгляньмо найважливіші концепції деяких мов і пояснімо їх прикладами.

14.1. Фортран-90

Автор: ANSI-комітет X3J3, 1978-1991

Відносно молода синхронна паралельність, як і інші, не може існувати без традиційного Фортрану. Кожний виробник масивно паралельних обчислювальних систем пропонує поряд з іншими мовами також паралельний діалект Фортрану, бо, на жаль, ці мови несумісні між собою.

Так, існують CMFortran для системи Connection Machine, MPFortran для MasPar, а також Fortran-Plus, що є єдиним засобом програмування в системі AMTDAP. Всі паралельні діалекти Фортрану належать до Fortran-90 стандарту цієї мови дев'яностих років [Metcalf, Reid 90]. У зв'язку з цим є надія, що застосовувані сьогодні паралельні варіанти Фортрану мають сумісність з Fortran-90.

Мова Fortran-90 – це подальший розвиток Fortran 77, в якій інтегровані також паралельні за даними розширення концепції масивів. Одначе з цієї безперервності розвитку Фортрану ні в якому разі не можна робити висновок, що старі програми на Фортрані (*"dusty decks"*) можуть бути дієздатними після простої компіляції на SIMD-системі! Це можливо лише в дуже обмеженій мірі і на сьогодні на не дуже задовільному рівні (див. автоматичну векторизацію в гл.16). Всі алгоритми мають бути заново перероблені і реімплементуватися з огляду на паралельне за даними виконання.

Паралельні операції над даними у Фортрані-90 виконуються за векторними командами, що називаються *"array expressions"*. У зв'язку з тим що Фортран-90 має працювати як у звичайних послідовних, так і в паралельних обчислювальних архітектурах, у самому Фортрані-90 не існує мовних ознак того, чи масив має оброблятися скалярно на керуючій ЕОМ чи векторно шляхом розподілу між процесорними елементами. З цієї причини синтаксис наявних паралельних Фортран-варіантів відрізняється від пропонованого Фортрана-90. Так, у Фортрані-плюс декларування паралельного вектора позначається зірочкою (*). На противагу цьому у MP-Фортрані визначають операції доступу до масива через скалярні або векторно розподілені ресурси. Декларація масива тут не має ніякого значення; тут можуть задаватись і відповідні директиви компілятора.

Паралельні мовні конструктиви

Наступна декларація задає паралельний вектор з 50

компонентами типу integer. Цей вектор може розподілятися покомпонентно між процесорними елементами (ПЕ) паралельної системи:

```
INTEGER, DIMENSION(50) : : V
```

За бажанням може також задаватися нижня величина індекса (наприклад, вектор з 50 елементами від 41 до 90):

```
INTEGER, DIMENSION (41:90) : : W
```

Три двовимірних вектори з кількістю елементів 100x50 декларуються так:

```
INTEGER, DIMENSION (100,50) : : A, B, C
```

Складання матриць може описуватись як проста команда, що виконується процесорними елементами паралельно покомпонентно:

```
A = B + C
```

В одній інструкції не завжди мають брати участь всі компоненти вектора. Часткові зони або тільки деякі компоненти можуть вибиратися за різними способами, а ПЕ інших компонент залишаються неактивними. Наприклад, наведена нижче інструкція активізує селективно часткову область з 2-ї до 10-ї компоненти вектора і присвоює кожному значення 1 (скалярні значення даних переносяться на всі вектори):

```
V(2:10) = 1
```

За бажанням можна задати величину кроку як третій параметр в межах масиву. Так, присвоєння деякої величини кожному другому елементу, починаючи з V(1), має вигляд:

```
V(1:21:2) = 1
```

Селекція процесорних елементів може також виконуватися відкиданням індексів. Наприклад, операція присвоєння вектору V рядка № 77 матриці A записується так:

```
V = A(77,:)
```

За допомогою інструкції WHERE операція селекції векторних елементів для деякої команди може здійснюватись простим способом. Наприклад, команда

```
WHERE(V. LT. 0)V = 7
```

присвоює нове значення лише тим компонентам вектора, які менші від нуля.

Для редукції вектора в скаляр, наприклад складнанням усіх компонент, передбачено ряд стандартних функцій. Одначе визначення редукційних функцій з рівня користувача неможливе. Наприклад, інструкція редукції S=SUM(V) дає суму всіх компонент вектора V і присвоює цю суму скалярній змінній S.

У розпорядженні програміста є такі оператори редукції:

ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCKT, SUM

Оператори ALL і ANY відповідають булівським "і" (and), "або" (or), а оператор COUNT паралельно визначає кількість елементів поля, що мають значення TRUE. Крім цього, існують стандартні функції для скалярного добутку та перемноження матриць (не плутати з поелементним множенням двох матриць, якому відповідає простий оператор множення):

```
DOTPRODUCT (Vektor_A,Vektor_B)
MATMUL (Matrix_A,Matrix_B)
```

Програмування масивів на Фортрані-90 (а також в діалектах) дає можливість лише з великими обмеженнями використовувати паралельність. Математичні формули при цьому мають відносно легко трансформуватися, тоді як складні алгоритми можуть спричиняти труднощі у зв'язку з недостатньою гнучкістю мовних конструктивів.

На закінчення розділу про Фортран-90 наводимо дві програми як приклади. Перша з них – для обчислення скалярного добутку, а друга – для обчислення оператора Лапласа щодо визначення контурів у задачах розпізнавання графічних образів. Обидва ці приклади використовуються також для інших представлених тут SIMD-мов програмування.

Обчислення скалярного добутку в Фортрані-90:

```
      INTEGER S_PROD
      INTEGER, DIMENSION(100) : : A, B, C
      .....
      C = A * B
      S_PROD = SUM(C)
```


У зв'язку з тим, що скалярний добуток є стандартною функцією Фортрана-90, його можна визначити за допомогою оператора:

```
S_PROD = DOTPRODUCT (A, B),
```

де спочатку здійснюється покомпонентне множення, а потім редукція складанням. Тут не треба реалізувати явний обмін даними між процесорними елементами.

Оператор Лапласа належить до множини операторів, які в рисунках сірого кольору виділяють межі (*edge detektion*). Цей оператор виконує просте локальне диференціювання (див. рис. 14.1) і дуже добре підходить до паралельного виконання. Оператор Лапласа застосовується паралельно для кожного пункту рисунка (фотографії) разом з його чотирма сусідніми пунктами.

Рис. 14.1. Розпізнавання країв об'єкта оператором Лапласа

Обчислення оператора Лапласа в Фортрані-90:

```
INTEGER, DIMENSION (0:101,0:101) : : BILD
...
BILD(1:100,1:100) = 4*BILD(1:100,1:100)
                  - BILD(0:99,1:100) - BILD(2:101,1:100)
                  - BILD(1:100,0:99) - BILD(1:100,2:101)
```

Навкруги поля з 100x100 елементами передбачено невикористовувану межу, яка допускає простіше визначення за допомогою описування масива. Тут і в наступних програмах-прикладах не проаналізовано, чи результуюче число належить допустимій області (наприклад, 0..255). Цим аналізом має доповнюватися закінчена програма.

Фортран-90 з його неявно паралельними конструктивами є без сумніву хорошою базою як новий Фортран-стандарт для послідовних обчислювальних архітектур. Одначе під час застосування цієї мови в паралельних обчислювальних архітектурах пропадають можливості програміста впливати на зростання продуктивності паралельної програми за допомогою явних конструктивів. З цієї причини рядом виробників паралельних та супер-ЕОМ, що об'єднані в "High Performance Fortran Forum" (HPFF), розробляється новий діалект Фортрану. Цей "High Performance Fortran" (HPF) має використовувати Фортран-90 як основу і мати в собі паралельні мовні концепції Фортрану-D [Fox, Hiranandani та ін. 91].

Фортран D

Автори : Fox, Hiranandani, Kennedy, Koelbel, Kremer, Tseng, Wu, 1991.

Фортран-D (D – це "date decomposition") орієнтований на застосування як в SIMD-, так і в MIMD-системах. Основна ідея паралельних мовних концепцій тут полягає в розподілі ("*decomposition*") або в упорядкуванні даних, що мають оброблятися паралельно. У Фортрані-D не існує явних операцій обміну даними, замість них використовуються індексовані оператори присвоєння.

Паралельні мовні конструктиви

Визначення паралельного масиву даних виконується

в три етапи (рис.14.2, порівняйте з п. 14.4, CONFIGURATIO та CONNECTION в мові PARALLAXIS):

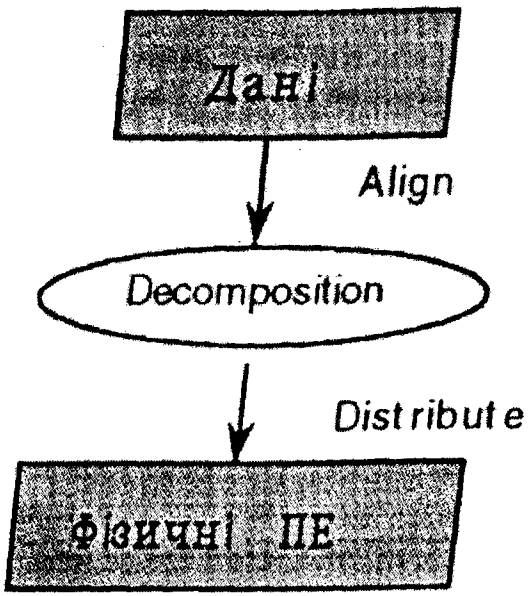


Рис. 14.2. *Розподіл масиву даних між ПЕ*

Визначення логічного розміщення (оператор DECOMPOSITION)

У разі логічного розміщення елементів масиву визначається структура масиву з його ім'ям, кількість та величина мір (*Dimension*); пам'ять при цьому не розподіляється. Так, конструктив

DECOMPOSITION S(100,100)

визначає двовимірну структуру S по 100 елементів в кожній мірі.

Визначення логічного підпорядкування (оператор ALIGN)

Масиви, що визначені раніше, можуть бути підпорядковані конструктивом ALIGN деякій декомпозиції без конкретного зв'язку з фізичним процесором. Наприклад, масив A може відображатися на структуру S:

```
REAL A(100,100)
DECOMPOSITION S(100,100)
ALIGN A(I,J) WITH S(I,J).
```

Це відображення не завжди може йти безпосередньо: в ALIGN-конструктиві права частина може містити операції над індексами, а також індексні вектори для будь-яких перестановок:

```
ALIGN A(I,J) WITH S(I+1,2*J-1)
```

Можуть також цілі рядки або колонки бути підпорядковані одному єдиному структурному елементу ("*collapse*"). В цьому прикладі

```
DECOMPOSITION T(100)
ALIGN A(I,J) WITH T(I)
```

кожному елементу структури T підпорядковується рядок масиву A.

Вибірково в процесі підпорядкування оператором ALIGN за допомогою ключового слова RANGE можна визначити також тільки бажану частину індексної області:

```
ALIGN A(I,J) WITH S(I,J) RANGE (1:100,50:60)
```

Для крайніх елементів масиву, які у разі застосування ALIGN-конструктива можуть опинитися за його межами, треба використовувати один з трьох засобів застереження:

- **ERROR** (default) – до елемента, що виявився за межами масиву, не можна звертатись, інакше виникає помилка під час виконання програми;
- **TRUNC** – всі елементи, що виявлені за межами масиву, підпорядковуються одному й тому ж крайньому елементу;
- **WRAP** – всі елементи, що виявлені за межами масиву, підпорядковуються так, як в торовій структурі, відновлюючись від початку рядка або колонки.

Визначення фізичного підпорядкування (оператор DISTRIBUTE)

На цьому третьому і заключному етапі віртуальні елементи деякої структури (включаючи елементи масиву, підпорядковані цій структурі) розподіляються між наявними фізичними процесорами. Для кожної міри (координати) структури тут може вибиратися один з трьох методів розподілу (рис.14.3):

- BLOCK – розподіл декомпозиції на взаємозв'язані блоки однакової величини для кожного процесора;
- CYCLIC – кожний елемент декомпозиції підпорядковується циклічно наступному процесору;
- BLOCK_CYCLIC(X) – розподіл здійснюється так, як в CYCLIC, але блоками величини x;
- * – всі без винятку елементи декомпозиції даної міри (координати) підпорядковуються одному й тому самому процесору.

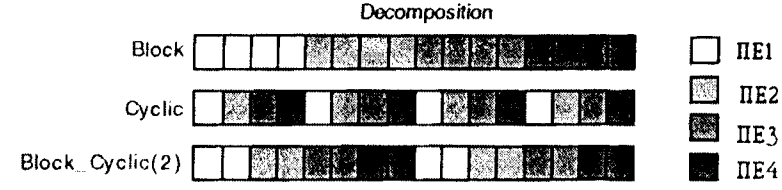


Рис. 14.3. Визначення фізичного упорядкування (DISTRIBUTE)

Наведені далі приклади показують розміщення структури S із 100 рядками і 100 колонками на 50 фізичних процесорах:

- DISTRIBUTE S(BLOCK, *) – кожний процесор одержує два сусідніх рядки;
- DISTRIBUTE S(CYCLIC, *) – кожний процесор одержує два рядки, які розміщені один від одного на відстані 50 кроків;

• DISTRIBUTE S(BLOCK, CYCLIC) – рядки підпорядковуються за способом блоку, а колонки – циклічно.

Щоб останнє підпорядкування було однозначним, ключове слово BLOCK має задавати точну кількість рядків фізичних процесорів (наприклад, BLOCK(5) – для процесорів поля 5x10), бо у Фортрані-D не підтримуються віртуальні процесори. Принаймі ця обставина робить розглянуту методику підпорядкування процесорів елементам даних дещо не зовсім наочною!

Для паралельного виконання інструкцій у Фортрані-D передбачено FORALL-конструктив. Всі ітераційні цикли виконуються за його допомогою паралельно на різних процесорах відповідно до узгодженого підпорядкування масиву:

```
FORALL I = 1, 100
   FORALL J = 1, 100
         A(I, J) = 5 * A(I, J) - 3
   ENDDO
ENDDO.
```

За допомогою ON-речення, яке в наведеному прикладі не використовується, можна досягти безпосереднього підпорядкування ітераційних циклів фізичним процесорам.

У Фортрані-D є також операція редукції, що має характер стандартної операції. На противагу Фортрану-90 ця операція має бути включена в послідовний або паралельний DO-цикл. Завчасно визначеними є оператори:

SUM, PROD, MIN, MAX, AND, OR.

Можуть застосовуватися також довільні функції редукцій, що визначаються самим користувачем. Наступний приклад демонструє редукцію одновимірного масиву Y в скалярне число X, що є сумою елементів Y:


```
REAL  X, Y(N)
FORALL I = 1, N
   REDUCE (SUM, X, Y(I))
ENDDO
```

Паралельні цикли, що містять операцію редукції, перетворюються компілятором у відповідну бінарну деревоподібну структуру.

14.2.Мова програмування C*

Автори:John Rose і Guy Steele, Thinking Machines, 1987-1990.

Мова C* ("C-star") була розроблена Rose і Steele спеціально для машин серії Connection-Machine [Rose, Steele 87]. Запропонована тим часом фірмою Thinking Machines версія 6 мови C* як її подальший розвиток являє собою досить апаратно-незалежну розробку і тому може розглядатись як спроба побудувати стандартну SIMD-мову програмування.

Паралельна мова C* – це розширення послідовної мови C паралельними мовними конструктивами. На мові C* можливе елегантне програмування на рівні віртуальних процесорів. Перехід з віртуальних процесорів на фізичні виконується в системі CM-2 апаратним способом. Мова C* повідомляє програмісту-користувачу картину гомогенного адресного простору, тобто кожний ПЕ може через індексні вирази мати доступ до локальної пам'яті кожного довільно взятого другого ПЕ. Під час переходу від C*-версії 5 до C*-версії 6 було повністю змінено мовне описання. В той час як C*-версія 5 готувалася на мові C**, основою C*-версії 6 є ANSI-C. В новому визначенні мови немає об'єктно-орієнтованих концепцій, змінені паралельні конструктиви і немає ніякої переносимості. Програми, що написані на C*-версії 5, мають заново імплементуватись! Взагалі треба підкреслити, що C*-версія 5 і C*-версія 6 – це різні паралельні мови програмування.

У С*-версії 5 скалярні змінні декларуються з ключовими словами mono, а векторні – з словами poly. Групи віртуальних процесорів декларуються за допомогою domain-конструктиву. Обмін даними між ПЕ виконується операторами присвоєння, що мають вирази для визначення вказівників (Pointer) в області даних відповідних ПЕ-передавачів і ПЕ-приймачів.

Все, що розглядається тут, далі має відношення лише до С*-версії 6.

Паралельні мовні конструктиви

У разі декларування змінних величин відповідно до SIMD-моделі чітко розрізняють дані, які у вигляді скалярних величин вводяться в керуючу ЕОМ (HOST) і обробляються тільки там, і ті дані, які мають бути розподілені покомпонентно між віртуальними ПЕ. Скалярні дані декларуються так, як у звичайній мові С; векторні змінні декларуються через shape-описуваня, яке визначає структуру вектора аналогічно декларуванню масива (порівняйте CONFIGURATION-описування в мові Parallaxis п.14.4). Наприклад, одновимірний вектор змінних V, що має 50 компонент, декларується таким конструктивом:

```
shape [50] ein_dim;
int : ein_dimV
```

Для кожного вектора змінних має бути зроблено одне визначення структури. Так, визначення трьох векторних двовимірних полів A, B і C з кількістю компонент 100x50 буде мати вигляд:

```
shape [100] [50] zwei_dim;
int : zwei_dim A,B,C.
```


Як типи даних можуть використовуватися всі наявні в С типи даних, включаючи структури і масиви (Strucktures, Arrays), що потім підпорядковуються локально кожному віртуальному ПЕ. Наприклад, декларування

```
int : ein_dim Feld [100]
```

підпорядковує вектор з 50 компонентами відповідно до 50 процесорних елементів, причому кожна компонента вектора – це локальний масив, що має, у свою чергу, 100 елементів.

Щоб можна було виконувати паралельні операції над векторами, в інструктивній частині програми спочатку мають бути вибрані відповідні оператори shape з операцією with. Якщо кількість наявних ПЕ більша, ніж визначено в shape, то решта ПЕ залишається в межах цього блоку програми неактивною. Наприклад, у фрагменті програми

```
with (zwei_dim)
        { A = B + C; }
```

спочатку вибирається двовимірна структура zwei_dim, а потім виконується складання матриць.

Як і у Фортрані-90, в С* є також інструкція where, що виконує тільки ті наступні інструкції програми над векторними компонентами (віртуальними процесорами), для яких виконується ця булівська умова. Щоб можна було знову звернутися до всіх векторних компонент (віртуальних ПЕ) всередині WHERE-інструкції, що, можливо, має складну побудову, передбачено додатковий мовний конструктив EVERYWHERE. Наприклад, у фрагменті програми

```
with (ein_dim)
    where (V < 0) { V=7; }
```

тільки ті компоненти вектора V, що менші від нуля, мають отримати нове числове значення.

У принципі в одному виразі або інструкції допустимо разом бути тільки тим векторам, які належать

до одного й того оператора shape, тобто мають однакову структуру і передусім ту саму кількість компонент. Операція присвоєння типу A=V тут неможлива. Винятком є операції із скалярними величинами (як наведена вище операція з V<0 і V=7). Вони трансформуються спочатку у вектор з ідентичними компонентами і потім використовуються далі. Навпаки, присвоїти деякому скаляру вектор (або векторний вираз) можна тільки через "type-casting". Відповідь на запитання, яка компонента вектора все ж таки присвоюється скалярній змінній, залежить від імплементації! Це дуже чутлива до помилок і до того ж непотрібна операція, якої треба уникати:

```
int S;
.....
S = (int) V;
```

Увага:
присвоєння деякої
невизначеної компоненти!

Крім операції присвоєння над вектором можуть виконуватися паралельно також всі інші С-операції. При цьому складні операції присвоєння із мови С (а також деякі інші) застосовуються як оператори редукції векторів у скаляри. Наступне, складене з двох операцій присвоєння обчислює суму всіх компонент вектора V і надає їх скалярній змінній S:

```
S = 0 ;
S + = V;
```

У зв'язку з тим, що редукційне число складається з початковим значенням S, змінна S тут ініціалізується з нуля.

У мові С* існують такі оператори редукції:

```
+=( сума );  *=( добуток );  &=(і);  I=( або);  ^=(xor);
<?=( мінімум );  >?=( максимум )
```

Застосування операторів редукції за визначенням користувача не дозволяється.

На жаль, в С* операції редукції можуть бути проблематичними. У той час як в ANSI_C обидві інструкції

```
S+=V;
S = S + V;
```

є ідентичними, в С* це не дійсно у випадку, коли S є скаляром, а V – вектором. Якщо з обох боків оператора += стоять вектори, то операція типу V+=W дає не редукцію, а просте складання двох векторів.

Приклад:

```
int S;
int: ein_dim V, W;
```

S+=V → $S := S + \sum_i V_i$ – редукція вектора;

S=S+V → **Помилка** – спроба надати скаляру значення вектора, дивіться вище "type-casting";

V+=S → V := V+S – додавання скаляра до вектора;

V+=W → V := V+W – складання вектора з вектором.

Той факт, що ці інструкції мають різний зміст залежно від типу операторів, є дуже неприємним. Віртуальна структура зв'язку між процесорними елементами не декларується на етапі підготовки програми, її реалізує автоматична програма-рутина під час кожного доступа до даних сусідніх ПЕ. Тут існують два різних типи комунікацій між процесорними елементами – "загальна комунікація" і "комунікація-решітка". Як уже було показано в п. 11.1, загальна комунікація виконується через глобальний гіперкуб СМ-2, а комунікація за структурою решітки реалізується швидкодіючою локальною NEWS-решіткою. "Цільові адреси" задаються так званими індексними змінними, причому вектор-передавач і вектор-приймач можуть належати до різних структур (shape). В прикладі, що наведений нижче, вектор V одержує компоненти від вектора W, але в іншому порядку

(перестановка), ніж задано вектором Index. Тут всі компоненти мають бути переставлені вправо на дві позиції (рис. 14.4):

```
Shape [50] ein_dim;
int: ein_dim V, W, Index;
...
/*index містить перестановку значень: 0, 1, 2, ..., 49*/
with (ein_dim) {
    [Index]V = W;
              }
```

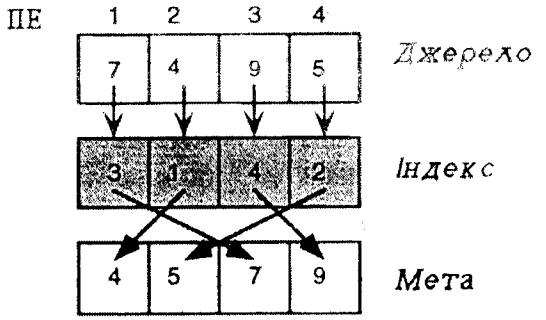


Рис. 14.4. Загальний обмін даними

Векторні компоненти V[0] та V[1] набувають при цьому невизначених значень і тому мають бути деактивовані where-операцією, щоб не виникли програмні помилки.

У загальному випадку справедливе також твердження: чим складнішою є застосована топологія, тим незручнішим і менш осяжним стає індексний вираз у мові C*. Тут було б доцільніше використовувати структуроване задання структури зв'язку подібно тому, як це зроблено в мові Parallaxis. Індекси можуть бути розміщені як у лівій, так і в правій стороні операції присвоєння. У випадку, коли оператор shape є багатовимірним, для кожної координати треба задавати власний індекс. За допомогою індексів можна також виділяти окремі векторні компоненти і надавати їхні значення скаляру:

S = [11]V.

Для використання швидкої решітчастої комунікації може застосовуватися стандартна функція pcoord. Вона видає позицію векторної компоненти (тобто координати віртуального ПЕ) відносно раніше визначеного оператора shape в координатах, що задані у вигляді параметра.

Приклад:

```
shape [100] [50] zwei_dim;
int : zwei_dim A,B;
....
with (zwei_dim) {
        A = pcoord (0);
        B = pcoord (1);
              }
```

Тут матриця A здобуває позицію кожної компоненти відносно рядків, а B – позицію відносно колонок:

$$A=\begin{pmatrix} 0 & 0 & \dots & 0 \\ 1 & 1 & \dots & 1 \\ \dots & & & \\ 99 & 99 & \dots & 99 \end{pmatrix}; \qquad B=\begin{pmatrix} 0 & 1 & \dots & 49 \\ 0 & 1 & \dots & 49 \\ \dots & & & \\ 0 & 1 & \dots & 49 \end{pmatrix}.$$

Застосування функції pcoord в адитивному індексному виразі розпізнається компілятором і приводить до генерації більш ефективного програмного коду. Комунікація для переміщення елементів вектора на дві позиції може бути записана так:

```
shape [50] ein_dim;
int : ein_dim V, W, Index;
...
        with (ein_dim) {
          where (pcoord (0) < 50)
            [pcoord(0) + 2]V = W;
        }
```

Для полегшення паралельного за даними програмування в мові C* існує цілий ряд операцій, які керуються параметрами, що тут докладно не розглядаються. Ось деякі з цих операцій:

• **scan**: обчислення часткових результатів операції над вектором.

Приклад: обчислення всіх часткових сум вектора wert

```
part_sum= scan(wert, 0 , CMC_combiner_add, CMC_upward,
                    CMC_none, CMC_no_field, CMC_inclusive);
```

• **spread**: розподіл результатів паралельної операції на компоненти іншої векторної змінної, можливо з різною структурою (за допомогою розмноження відповідних числових даних);

• **enumerate**: обчислення позиції кожної активної компоненти вектора, керування за допомогою параметра (загальна версія операції pcoord);

• **rank**: обчислення позиції кожної компоненти вектора відносно сортування компонент за їхніми числовими номерами.

Під кінець розглянемо два паралельних за даними алгоритми-приклади для скалярного добутку та оператора Лапласа.

Скалярний добуток на мові C* :

```
shape [max] liste;
float : liste x, y;
float s_prod = 0 . 0;
...
with (liste) {
  s_prod += x*y;
}
```

Процесорні елементи упорядковані у вигляді списку. Після операції виділення (селекції) процесорні елементи виконують покомпонентно операції множення і за допомогою редукції додають результати до скаляра.

Операція Лапласа на мові C*:

```
shape [100] [100] grid;
int : grid pixel, dim1, dim2;
...
with (grid) {
dim1 = pcoord ( 0 );
dim2 = pcoord ( 1 );
pixel = 4*pixel - [dim1 - 1] [ dim2 ] pixel
                - [dim1 +1] [ dim2 ] pixel
                - [dim1     ] [dim2 - 1]pixel
                - [dim1     ] [dim2 +1]pixel;
}
```

Для застосування функції pcoord, що визначає позицію в решітці, в C* існують скорочені конструкції, які дещо спрощують наведені вище записи:

```
with (grid) {
pixel = 4*pixel - [.-1] [ . ]pixel - [. + 1]  [ . ] pixel
                - [ . ] [ . -1]pixel - [ . ] [. + 1]pixel;
}
```

14.3. Мова програмування системи MasPar – MPL

Автори : MasPar Computer Corporation, 1990.

MPL (MasPar Programming Language) подібно до мови C* є розширенням послідовної мови програмування C паралельною концепцією, що була розроблена фірмою MasPar спеціально для SIMD-системи MP-1[MasPar 91]. У зв'язку з тим, що MP-1 функціонує з двома різними комунікаційними мережами, тут для ефективного використання ресурсів комутації передбачено ряд машинно-залежних інструкцій, що забезпечують дію швидкої решітчастої структури ("X-net") поряд із загальними інструкціями щодо комунікації через глобальний маршрутизатор (Router). Ці мовні конструктиви зменшують апаратну незалежність мови MPL.

Наявні паралельні концепції в MPL дещо примітивніші, ніж у більш розвиненій мові програмування C*

Паралельні мовні конструктиви

Декларування змінних величин, як і практично в усіх SIMD-мовах, чітко розрізняє скалярні дані, які декларуються так, як у послідовній мові C, і векторні дані, які виділяються ключовим словом plurar. Так, декларування

```
plurar int i, j;
```

підпорядковує змінні ***i, j*** типу integer всім фізично наявним процесорним елементам.

Віртуальні процесори не підтримуються, немає також можливості компонувати різні групи ПЕ (конфігурації або топології). В неявній формі завжди передбачається фізична структура MP-1 з двовимірною решіткою.

Для зв'язку між послідовною головною програмою, що виконується у керуючій ЕОМ (HOST, front end), і паралельною програмою, що виконується у MasPar (back end), можна позначити деякі змінні величини ключовим словом visible. До цих змінних можна звертатися з обох названих обчислювальних ресурсів.

Обмін даними між процесорними елементами може виконуватися двома різними способами відповідно до двох структур зв'язку MP-1: швидкий локальний обмін даними в решітці з восьмикратним зв'язком між сусідніми ПЕ за командою xnet і обмін у довільній топології зв'язку між ПЕ через повільний глобальний маршрутизатор за допомогою команди router. Для xnet-зв'язку використовується вісім різних команд, що відповідають географічним напрямкам:

xnetN, xnetNE, xnetE, xnetSE, xnetS, xnetSW, xnetW, xnetNW

Так, в інструкції щодо комунікації через X-мережу

```
j = xnetW[2] . i;
```

значення векторної змінної ***i*** переміщується на дві позиції вліво (захід, W), а потім надається вектору ***j***. Аналогічний ефект досягається комунікацією через маршрутизатор

(роутер), якщо вектор index з відповідним ідентифікаційним номером був розміщений у цільових процесорних елементах:

```
j = router [index]. i;
```

Для застосування координат ПЕ в паралельних інструкціях мова MPL має такі скалярні та векторні константи:

Скалярні

- **nproc** – загальна кількість усіх процесорних елементів в MasPar-системі;
- **nxproc** – кількість колонок у масиві ПЕ MasPar-системи;
- **nyproc** – кількість рядків у масиві ПЕ MasPar-системи.

Векторні

- **iproc** – ідентифікаційний номер ПЕ (0 .. nproc -1);
- **ixproc** – позиція ПЕ в рядку (0 .. nxproc -1);
- **iyproc** – позиція ПЕ в колонці (0 .. nyproc -1).

Паралельні інструкції виконуються або на всіх ПЕ, або можуть бути виділені групи ПЕ неявним способом за векторіальною умовою простими if- або while-інструкціями. В останньому випадку then- і else-напрями розгалуженої програми можуть виконуватись один за одним, а саме тоді, коли для однієї частини ПЕ векторна умова приводить до логічної величини true, а для іншої – до false. Командою all знову можуть бути активізовані всі процесори в межах зони селекції або циклу. Явна операція селекції (вибору) тут не існує.

За допомогою команди proc можна здійснити доступ до одного заданого ПЕ; наприклад, керуюча ЕОМ може

виконати обмін даними (читання/запис) з указаними ПЕ. Координати процесорного елемента можуть бути задані у вигляді одного індекса iproc (номер ПЕ) або двох індексних чисел відповідно до позицій iyproc та ixproc (рядок, колонка):

```
int s;
plurar int v;
...
s = proc [1023] . v;   компонента з номером 1023 вектора v;
s = proc [ 5 ] [ 7 ] . v; компонента 5-го рядка і 7-ї колонки вектора v;
```

Редукція вектора у скаляр може виконуватися такими уже визначеними операціями:

reduceADD, reduceMUL, reduceAND, reduceOR, reduceMAX, reduceMIN.

Кожному оператору як суффікс має задаватись тип операнда: наприклад, оператор reduceADDf визначає редукцію вектора типу "floating-point" (числа з плаваючою комою) операцією складання елементів. Застосування операцій редукції, визначених користувачем, в MPL неможливе.

Одним із недоліків MPL є вибрана з міркувань підвищення продуктивності машинно-залежна імплементація операцій обміну даними. Крім того, було б бажано мати в MPL мовну концепцію більш високого рівня для програмування віртуальних процесорів та структур їхнього зв'язку.

Як і для мов, розглянутих вище, наводимо два приклади на мові MPL.

Скалярний добуток на мові MPL:

```
float s_prod (a, b)
plural float a, b;
{ plural float prod;
  prod = a*b;
  return reduceADDf (prod);
}
```


Обидва вектори покомпонентно перемножуються і операцією складання трансформуються в деякий скаляр.

Оператор Лапласа на мові MPL:

```
plural int pixel;
...
pixel = 4*pixel - xnetN[1] . pixel - xnetS[1] . pixel
                - xnetW[1] . pixel - xnetE[1] . pixel;
```

Обмін даними у двовимірній решітці може виконуватися машинно-залежною командою "xnet", що реалізує всього вісім напрямків зв'язку (північ, південь, захід, схід та 4 напрямки між ними). Параметр у прямокутних дужках визначає кількість тактів (кроків) переміщення змінних. Для всіх інших структур зв'язку має застосовуватися повільніший глобальний маршрутизатор (Router).

14.4. Мова Parallaxis

Автор: Томас Бройнль, 1989.

Мова Parallaxis [Bräunl 89], [Bräunl 90], [Bräunl 91b] розроблена на базі послідовної мови Модула-2 [Wirth 83]. Розробка полягає в розширенні Модули-2 паралельними концепціями обробки даних. Ця мова повністю машинно-незалежна, що дає змогу переносити Parallaxis-програми на різні SIMD-системи. Існує також система моделювання для Parallaxis із засобами редагування, відпрацювання та візуалізації програм, яка реалізована на великій кількості однопроцесорних рабочих місць та персональних ЕОМ. Ця система дає змогу розробляти малі за об'ємами даних паралельні програми, тестувати та коректувати їх. Крім цього, існують компілятори Parallaxis-програм на системах MasPar MP-1 і Connection Machine CM-2, що забезпечують паралельне виконання програм. Моделююче середовище

дає можливість навчитися паралельному за даними програмуванню на простих комп'ютерних системах, а також забезпечує ефективну розробку паралельних програм, які згодом запускаються в роботу на дорогих паралельних SIMD-системах. Parallaxis-середовище програмування доступне користувачам у вигляді Public-Domain [Bart, Bräunl, Engelhardt, Sembach 92].

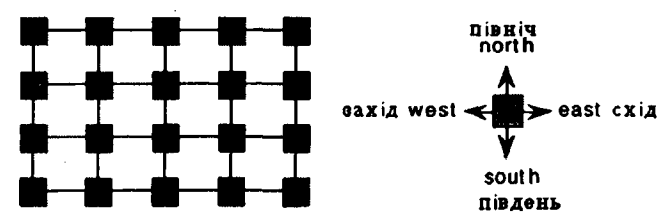
Основна особливість Parallaxis – це програмування на абстрактному рівні з віртуальними процесорними елементами (ВПЕ) і застосування віртуальних зв'язків між ВПЕ. Кожна програма містить у собі окрім описування алгоритму також напівдинамічне декларування зв'язків між ПЕ у функціональній формі. Це означає, що бажана (необхідна) топологія визначається заздалегідь для кожної програми (або процедури) і може під символічним ім'ям (замість складних арифметичних індексних або вказівних виразів) розміщуватись в алгоритмічній частині програми.

Паралельні мовні конструктиви

На відміну від розглянутих мов паралельного за даними програмування в мові Parallaxis для кожної програми (або сепаратно для кожної процедури) визначається віртуальна машина, що складається з процесорів та структури зв'язку між ними. Це виконується за два простих етапи. Спочатку задається кількість процесорів та розміщення їх у системі координат ключовим словом CONFIGURATION аналогічно тому, як виконується декларування масиву. Тут ще немає визначення, як з'єднуються між собою процесори. Це визначення здійснюється у функціональній формі, що вводиться з ключовим словом CONNECTION. Кожне з'єднання одержує символічне ім'я і визначає зв'язок будь-якого процесорного елемента (ПЕ) з належним йому сусіднім ПЕ. Задання відносного ПЕ-сусіда виконується за допомогою арифметичного конструктиву з індексів ПЕ-приймача, а

також імені вхідного каналу. В паралельній програмі обмін даними буде виконуватися за символічними іменами.

На рис. 14.5 наведено простий приклад визначення на мові Parallaxis розміщення ПЕ у двовимірній решітчастій структурі. Декларація CONFIGURATION виділяє користувачу 4x5 віртуальних процесорів, які за допомогою декларації CONNECTION зв'язуються між собою у віртуальну решітку. У зв'язку з тим, що в SIMD-системах апаратно підтримуються передусім симетричні структури або топології, для решітки довільних розмірів досить чотирьох декларацій зв'язку. Кожна одиночна лінія зв'язку визначається в одному географічному напрямку (північ-південь, захід-схід). Наприклад, зв'язок у північному напрямку задається збільшенням на 1 першого індекса (i+1) і вказівкою на те, що південний канал сусіднього ПЕ північного напрямку є його входом (конструктив north : grid[i, j] → grid[i+1, j] . south;). Коли номер i+1 досягає останнього в цьому напрямку ПЕ, то його вихід на північ не знаходить сусіда, тобто для ПЕ, що розміщений на межі процесорного поля, не існує зв'язку в деякому напрямку і він не бере участі в обміні даними в цьому напрямку.




```
CONFIGURATION grid    [1..4], [1..5];
CONNECTION    north : grid[i,j] → grid [i+1,  j ]. south;
              south : grid[i,j] → grid [i-1,  j ]. north ;
              east  : grid[i,j] → grid [i,   j+1]. west;
              west  : grid[i,j] → grid [i,   j-1]. east;
```

Рис. 14.5. Двовимірна решітчаста структура, окремий ПЕ та їх декларування мовою PARALLAXIS

Цей простий спосіб визначення віртуальної структури процесорів можна поширити на цілий ряд складніших топологій.

На рис.14.6 показано Parallaxis-конструктиви для топологій типу двоїчного дерева та гіперкуба. Так, в гіперкубі зв'язки параметризовані, що дає можливість проводити обмін даними в обчислюваному напрямку. Для визначення двоїчного дерева використовуються стрілки, що вказують зв'язок у прямому і зворотному напрямках (бідирекціональний, двосторонній зв'язок). Використовуючи однонаправлені зв'язки, визначення дерева має розрізняти парні та непарні ПЕ відповідно до заданої структури дерева:

```
CONFIGURATION tree [1..15];
CONNECTION lchild:  tree[i]   → tree[2*i] . parent;
           rchild:  tree[i]   → tree[2*i+1] . parent;
           parent:  tree[i] →{even i}tree[i DIV 2 ] . lchild,
                             {odd  i}tree[i DIV 2 ] . rchild;
```

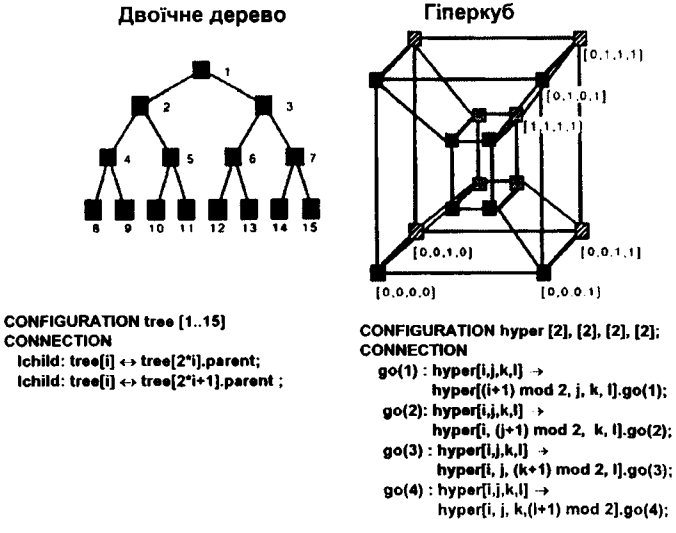


Рис. 14.6. Структури ПЕ та їх декларування мовою PARALLAXIS

У parent-зв'язку розрізняються два можливих напрямки: якщо номер ПЕ – парне число (even), то виконується зв'язок з лівим "дитям", а для непарного номера(odd) – зв'язок з правим "дитям".

За допомогою наведених зв'язків можуть бути побудовані будь-які структури ПЕ, навіть такі, що мають нерегулярності.

Є можливість визначити в одній програмі декілька топологій. Вони можуть бути самостійними і розміщуватися на різних ПЕ; у цьому випадку топології будуть працювати з різними векторними структурами даних. Різні топології можуть визначатись як види "різних поглядів" на рішення проблеми і розміщуватися на одному й тому ж полі ПЕ (з ідентичною структурою пам'яті). В процедурах, що не перекривають одна одну, можуть задаватися локальні топології, тобто в Parallaxis можлива напівдинамічна побудова структур зв'язку.

Як і в інших SIMD-мовах програмування, в Parallaxis також проводиться чітка грань між скалярами та векторами як при декларуванні даних, так і при визначенні параметрів процедур та їхніх результатів. Скалярні дані розміщуються тільки в керуючій ЕОМ, а вектори покомпонентно і паралельно розподіляються між продекларованими віртуальними ПЕ. Замість ключового слова VAR, що використовується в мові Modula-2, в Parallaxis введено відповідно слова SCALAR та VECTOR.

Обмін даними між процесорами може виконуватися завдяки описаним деклараціям відносно просто відповідно до символічних імен напрямків зв'язку. За допомогою стандартної процедури PROPAGATE може відбуватися паралельний обмін даними однієї локальної векторної змінної між усіма ПЕ або в межах виділеної групи ПЕ. На рис.14.7 показано як приклад обмін даними у визначеній вище решітчастій структурі. Векторна змінна **X** переміщується тут на всіх ПЕ на один крок у напрямку "схід".

Під час обміну даними можуть задаватися також два параметри: перший містить у собі арифметичний вираз, результат якого має бути відправлений, а другий задає

змінну, що приймає числове значення цього виразу.

Приклад: PROPAGATE.east (4*y, x);

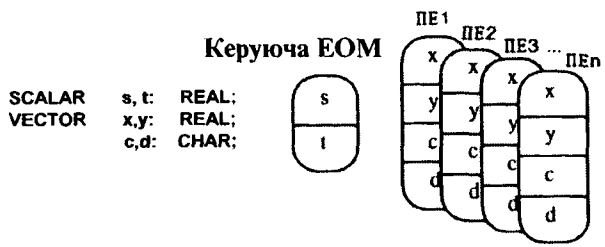


Рис. 14.7. Розміщення скалярних та векторних даних

У разі виконання операції PROPAGATE як передавач, так і приймач обміну даними мають бути активованими. На відміну від цього в операції SEND активним має бути лише передавач, а в операції RECEIVE – тільки приймач. Ці операції використовуються під час обміну даними між різними топологіями, бо згідно з SIMD-моделлю в кожний момент часу активна тільки одна визначена процесорна структура.

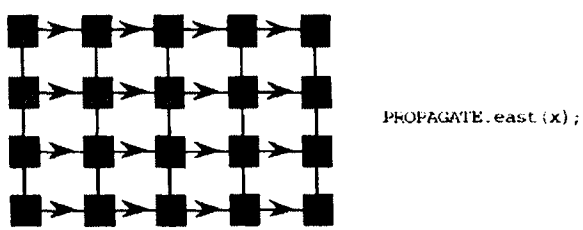


Рис. 14.8. Синхронний обмін даними

Приклад: SEND grid.east(4*y) TO grid.west(x);
RECEIVE tree.parent(t) FROM grid.east(x);

Паралельні інструкції задаються на відміну від інших SIMD-мов явним способом, що полягає в їхньому розміщенні в блоці "PARALLEL...ENDPARALLEL". При цьому можна виконати декілька видів операції селекції потрібної структури процесорних елементів: для паралельного блоку вона задається в явній формі як частина області, визначеної в CONFIGURATION, як кількість ПЕ або як булівський вираз. В неявній формі селекція може реалізуватись всередені деякого паралельного блоку відповідною паралельною командою селекції або ітераційного циклу (IF, WHILE, REPEAT, CASE).

Рис.14.9 показує паралельне за даними виконання операцій x := a+b на явно виділеній групі ПЕ.

```
VECTOR  x,a,b:  REAL;
...     (*тільки частина ПЕ активна*)
PARALLEL  grid[2..3],[1..4]
  x := a+b
ENDPARALLEL;
```

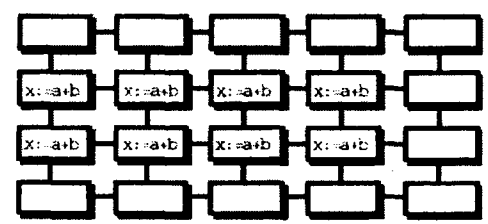


Рис. 14.9. Інструкція паралельного складання даних

Поряд з локальними даними у векторних виразах можуть також бути точні номери процесорних елементів (id_no) або позиція будь-якого ПЕ в межах розмірів декларованої ПЕ-конфігурації (DIM1, DIM2, ...). Ці параметри мають декларуватися заздалегідь як векторні константи.

Обмін даними між керуючою ЕОМ і процесорними елементами потребує спеціальних мовних конструктивів з відповідною семантикою. Покомпонентна передача скалярного поля даних і перетворення його в паралельний вектор реалізується процедурою LOAD, а зворотна операція "вектор-скалярне поле" – операцією STORE. Кожний ПЕ в цих операціях записує або читає, як правило, інше значення компоненти вектора (поля даних).

На рис.14.10 показано передачу скалярного елемента даних (константи або змінної) керуючою ЕОМ на всі процесорні елементи або на їхні групи. Кожна компонента вектора набуває при цьому значення елемента скалярного поля даних (рис.14.10, ліва частина, операції LOAD (v, s), STORE (v, s)) або значення однієї й тієї ж скалярної величини (рис.14.10, права частина, операція v := t). Остання операція реалізується за допомогою неявного оператора Broadcast (одне число – всім ПЕ).

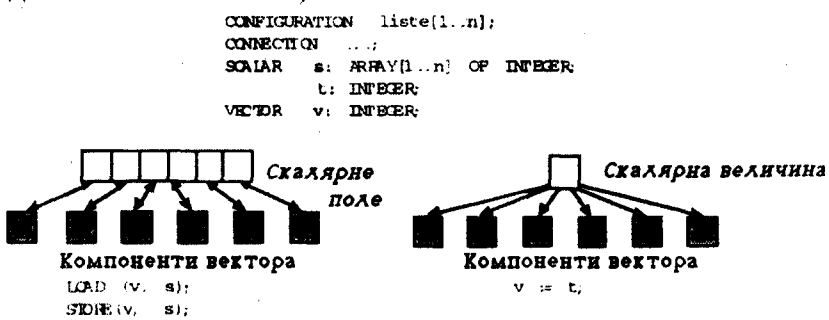


Рис. 14.10. Обмін даними між ПЕ та керуючою ЕОМ

Остання важлива операція – це редукція вектора в скаляр. Для цього в Parallaxis передбачено операцію REDUCE, що використовується разом із заздалегідь визначеною або вільно програмованою операцією редукції (рис14.11).

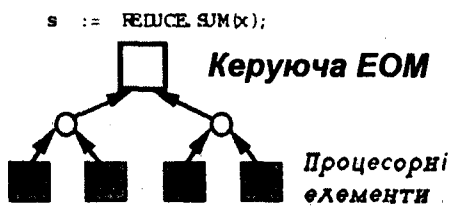


Рис. 14.11. Редукція вектора в мові Parallaxis

Заздалегідь визначеними операціями в Parallaxis є такі:

SUM, PRODUCT, MAX, MIN, AND, OR, FIRST, LAST.

Оператори FIRST і LAST повертають в керуючу ЕОМ значення змінної відповідно першого й останнього в даний момент активного ПЕ в процесорному масиві з ідентифікуючими номерами id_no. Зміст інших операторів редукції можна зрозуміти за їхніми назвами.

Структура зв'язку між ПЕ, що задана декларацією CONNECTION, не обов'язково має бути зв'язком типу 1:1. Зв'язок типу 1:n здійснюється наявною операцією Broadcast. Одначе під час реалізації зв'язків типу n:1 (а також взагалі для зв'язків типу m:n) треба мати на увазі, що завжди на вхідний порт кожного ПЕ надходить тільки один елемент даних. З цієї причини кожній операції обміну даними (PROPAGATE, SEND або RECEIVE) можна придати одну операцію редукції вектора. Розгляньмо приклад:

```
CONFIGURATION tree [1..max];
CONNECTION parent : tree[i]<->tree[2*i ] . children,
                    tree[i]<->tree[2*i+1].children;
...
SEND tree.children(y)  TO  tree.parent(x) REDUCE . SUM;
```

Тут побудовано дерево з n:1 зв'язками (по два ПЕ-дитя на одного ПЕ-батька). Операція SEND передає суму двох ПЕ-дітей в батьківський ПЕ.

Як і для інших мов, наводимо реалізовані на мові Parallaxis алгоритми скалярного добутку і оператори Лапласа.

Скалярний добуток на мові Parallaxis:

```
CONFIGURATION  liste [max];
CONNECTION;  (*пусто*)
SCALAR s_prod:    REAL;
VECTOR x, y, prod:  REAL;
...
PARALLEL
        prod := x*y
ENDPARALLEL;
s_prod := REDUCE.SUM(prod);
```

У паралельному блоці виконується покомпонентне множення. Наприкінці операцією редукції визначається сума всіх добутків.

Оператор Лапласа на мові Parallaxis:

```
CONFIGURATION grid [1..100], [1..100];
CONNECTION  north : grid[i, j] → grid[i+1, j].south;
            south : grid[i, j] → grid[i-1, j].north;
            east  : grid[i, j] → grid[i, j+1].west;
            west  : grid[i, j] → grid[i, j-1].east:

VECTOR pixel, n, s, w, e : INTEGER;
...
PARALLEL
        PROPAGATE . north (pixel, s);
        PROPAGATE . south (pixel, n);
        PROPAGATE . west  (pixel, e);
        PROPAGATE . east  (pixel, w);
        pixel := 4*pixel - n - s - w - e;
ENDPARALLEL;
```

Числові дані чотирьох сусідніх ПЕ обчислюються інструкціями PROPAGATE і запам'ятовуються як проміжні результати у вигляді векторних допоміжних змінних n, s, e, w (для чотирьох напрямків світу). Ці величини використовуються в арифметичному виразі для обчислення нового значення змінної pixel.

15. МАСИВНО ПАРАЛЕЛЬНІ АЛГОРИТМИ

Масивна паралельність за сьогоднішнього стану інтеграції обчислювальних систем пов'язана виключно з SIMD-системами, що мають паралельність потоків даних. При цьому на відміну від "звичайних" великоблокових паралельних алгоритмів виникла потреба в зовсім іншій, новій техніці програмування. Не тільки обмін даними, а й

селекція процесорів, циклічні програми мають в SIMD-системах іншу семантику, яка впливає на ефективність програм. Завантаження процесорів в SIMD-системах не є головною метою програмування. Менше завантаження процесорів в SIMD-системах порівняно з MIMD-системами зумовлене їхньою системною організацією, але за суттєво більшої кількості ПЕ цей недолік стає практично непомітним.

У центрі уваги розробки масивно паралельних алгоритмів стоїть їхнє формулювання на основі притаманної природним явищам паралельності.
Такий підхід дає можливість простіше розробляти паралельні за потоками даних програми; вони стають зрозумілішими, бо відпадає потреба робити всі операції послідовно, не існує обмежень моделі ЕОМ фон Ноймана.

Розглянуті в цій главі програми-приклади імплементовані на мові Parallaxis (п.14.4). Більш повне зібрання паралельних алгоритмів (в основному для SIMD-систем) опубліковане в працях [Akl 89], [Jaja 92], [Gybbons, Rytter 88].

15.1. Чисельне інтегрування

Наступний алгоритм із публікації [Babb 89] застосовує правило прямокутників для наближеного обчислення інтеграла (рис.15.1). Така сама постановка реалізована в п.10.3 як великоблоковий паралельний алгоритм для MIMD-системи.

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

$$\approx \sum_{i=1}^{Interv} \frac{4}{1+((i-0.5)*Breite)^2} * Breite$$

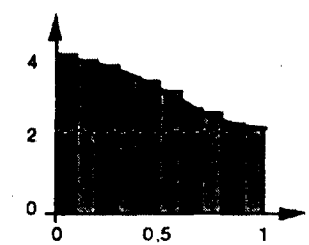


Рис. 15.1. *Наближене обчислення інтеграла*

Область [0, 1] по осі x розбивається на таку кількість інтервалів, яка потрібна для забезпечення заданої точності обчислень, причому для кожного інтервалу передбачається один віртуальний ПЕ. Кожний ПЕ обчислює значення функції f всередині свого інтервалу і перемножує його на ширину інтервалу.

Складання всіх часткових площ тільки за $\log_2 n$ кроків (проти n-1 кроків в MIMD-алгоритмі із п.10.3) дає наближену величину числа π:

```
1   SYSTEM compute_pi;
2   (* паралельний алгоритм, R. Babb *)
3   CONST intervals = 1000;
4         width     = 1.0 / FLOAT(intervals);
5   CONFIGURATION list [1..intervals];
6   CONNECTION (*пусто*);
7
8   VECTOR val : REAL;
9
10  PROCEDURE f (VECTOR x: REAL) : VECTOR REAL;
11  (* функція, що інтегрується *)
12  BEGIN
13        RETURN(4.0 / (1.0 + x*x))
14  END f;
15
16  BEGIN
17        PARALLEL (* обчисл. інт. за прав.прямокутника *)
18              val := width * f((FLOAT(id_no) - 0.5) * width);
19        ENDPARALLEL;
20        WriteReal (REDUCE.SUM(val), 15);
21  END compute_pi.
```

Функція, від якої береться інтеграл, обчислюється функціональною програмою f. Блок, що введений з ключовим словом PARALLEL, активізує всі ПЕ, бо немає ніяких вказівок на виділення (селекцію) деякої групи активізованих ПЕ. Векторна константа id_no дає номер кожного ПЕ від 1 до числа, що дорівнює кількості ПЕ (тут означено як intervals).

15.2. Клітчасті автомати

Клітчасті автомати – це широке поле застосування SIMD-систем. Кожній клітці може придаватися свій процесор і кожна з них виконує однакове для всіх завдання обробки інформації. Найбільше відомий клітчастий автомат "Гра життя" (Conway). Це двовимірна структура, яка з часом змінюється. Показаний нижче у вигляді Parallaxis-програми автомат є одновимірним, але він генерує в функції часу двовимірну картину (один рядок за кожний ітераційний крок). Завдання з переробки інформації відносно просте: кожна клітка має лише два стани і виконує операцію виключного АБО (OR) над двоїчним станом лівої та правої сусідніх кліток. Середня клітка ініціалізується змінною TRUE (видається як "X"), всі інші – змінними FALSE (видаються пусті символи).

```
1   SYSTEM cellular_automation;
2   CONST n=79;   (* кількість елементів *)
3         m=32;   (* кількість проходів циклу *)
4   CONFIGURATION list[1..n];
5   CONNECTION left  : list[i] -> list[i- 1] . right;
6                right: list[i] -> list[i+1] . left;
7
8   SCALAR i          : INTEGER;
9   VECTOR val, l, r : BOOLEAN;
10
11  PROCEDURE out;
12  VECTOR c: CHAR;
13  BEGIN
14     IF val THEN c := "X" ELSE c := " " END;
15     Write (c); WriteLn
16  END out;
17
18  BEGIN
19     PARALLEL (*ініціалізація *)
20       val := id_no = (n+1 DIV 2);(*середн. елем.= TRUE*)
21     ENDPARALLEL;
22
23     FOR i:=1 TO m DO
```

```
24        PARALLEL
25              out;
26              PROPAGATE . left  (val, l);
27              PROPAGATE . right (val,r);
28              val  := l<>r;
29        ENDPARALLEL;
30    END;
31  END cellular_automation.
```

Декларації CONFIGURATION і CONNECTION визначають лінійний список зв'язаних між собою процесорних елементів. Процедура out видає на екран дісплея актуальний булівський стан рядка ПЕ, причому булівські змінні перетворюються в знаки типу CHAR. Компонентий вектор знаків видається на екран послідовно векторною операцією Write. Під час ініціалізації в головній програмі логічна змінна TRUE присвоюється тільки середньому ПЕ, що має номер n+1 DIV2, а всі інші ПЕ одержують змінну FALSE. Наприкінці виконується скалярний цикл, в якому здійснюється видача та обчислення нового стану клітки за допомогою обміну даними (PROPAGATE) з будь-яким лівим і правим ПЕ-сусідом. На рис.15.2 показано стани цього клітчастого автомата, причому координата часу має початок вгорі.

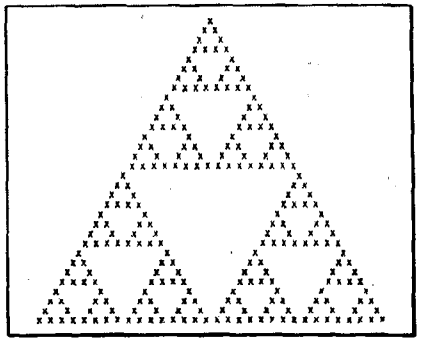


Рис. 15.2. Результати на виході клітчастого автомата

Останнім часом виникла велика кількість нових областей застосування клітчастих автоматів. Сюди

належать і так звані решітчасті газові автомати ("Lattice-Gas-Automat"), що використовуються для моделювання течій газів та рідин [Doolen 90], [Chen, Doolen, Matthaeus 91]. Тут замість звичайного розв'язання диференціальних рівнянь для моделювання двовимірних течій використовується гексогональна решітка з дискретних клітин. Кожна клітина визначає свій дискретний у функції часу стан клітини, і дані локального ПЕ-сусіда в решітці використовуються як параметри індекса таблиці. На рис. 15.3 показано змодельовані на решітчастому газовому автоматі потоки газу на несучому профілі (автори Putzold та Brenner, Штуттгартський університет).

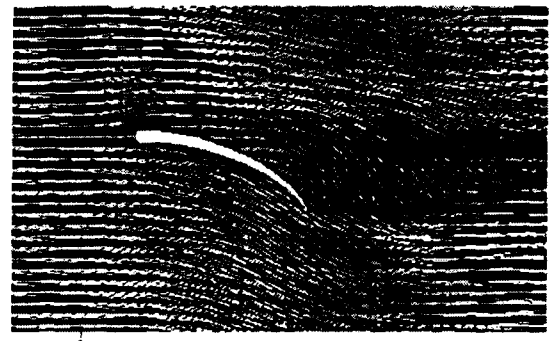


Рис. 15.3. Моделювання струмкових течій на автоматі типу "Lattice-Gas"

15.3. Генерування простих чисел

Метод генерування простих чисел, що використано тут, є паралельною версією метода "Сито Ератосфена". За допомогою n процесорів визначаються всі прості числа від 2 до n. Кожний ПЕ представляє число n, що відповідає його ідентифікаційному номеру (рис.15.4). Кожний ПЕ залишається активним упродовж часу, потрібного для ідентифікації його номера як потенційного простого числа. На кожному кроці друкується найменше ще активне число як просте, і всі останні ПЕ перевіряють, чи не є їхній номер

числом, кратним визначеному простому. Якщо це так, то в наступному циклі ці процесори уже будуть неактивними. Таким чином, на кожному кроці паралельно вилучаються з пошуку числа, кратні знайденому простому числу:

```
1   SYSTEM sieve;
2   CONFIGURATION list [2..200];
3   CONNECTION (* пусто *);
4
5   SCALAR prime     : INTEGER;
6   VECTOR removed   : BOOLEAN;
7
8   BEGIN
9     PARALLEL
10      REPEAT
11        prime := REDUCE . FIRST(DIM1);
12        WriteInt(prime,10); WriteLn; (*видати просте число*)
13          removed := DIM1  MOD  prime = 0
14          (* багаторазове вилучення *)
15        UNTIL  removed
16    ENDPARALLEL
17  END sieve.
```

Рис. 15.4. Паралельне генерування простих чисел

15.4. Сортування

Для проблеми сортування існує ряд різних SIMD-алгоритмів. Розгляньмо тут один з цих алгоритмів – "Odd-Even Transposition Sorting" (OETS), алгоритм, який можна розуміти як паралельний варіант сортування за ознакою "непарний-парний". Алгоритм OETS сортує n чисел за допомогою n ПЕ за n кроків. На рис.15.5 показано дію паралельного алгоритма. Кожний ПЕ цілеспрямовано одержує одне число, що підлягає сортуванню. Під час сортування розрізняються непарні і парні кроки. На непарних кроках всі ПЕ, що мають непарні ідентифікаційні номери, порівнюють їхні числа з правим сусідом (1-2, 3-4, 5-6 і т. д.) і обмінюються числами, якщо своє число має бути більшим за число сусіда. Аналогічно на парних кроках всі ПЕ, що мають парні номери, виконують порівняння та обмін з їхніми сусідами справа (2-3, 4-5, 6-7, і т. д.). Після n кроків послідовність чисел буде розсортована (на рис.15.5 – по порядку збільшення чисел, тобто 1, 2, 3, 4, 5).

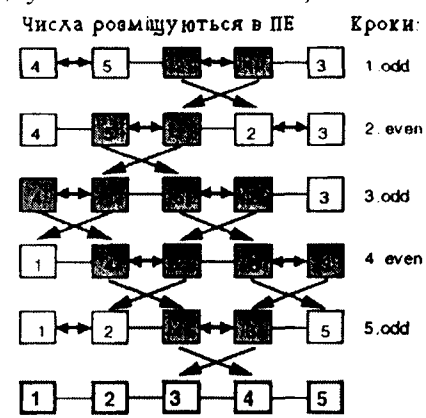


Рис. 15.5.Приклад сортування за методом перестановки непарних-парних чисел (Odd-Even-Transposition)

У Parallaxis-програмі за допомогою змінної lhs встановлюється, яку роль виконує ПЕ у разі порівняння –

лівого чи правого партнера. Ця роль змінюється під час кожного нового циклу. В кожному проході циклу всі ПЕ одержують спочатку числові дані лівого та правого сусіда (l та r). Партнер зліва бере число правого сусіда як число для порівняння, а сусід справа використовує число лівого сусіда. Комплексне порівняння як логічна операція "lhs=(comp<val)" є виконаним для лівого ПЕ-партнера, якщо число правого сусіда більше, ніж своє число ПЕ, а для правого ПЕ-партнера – якщо число зліва виявляється меншим, ніж своє число. Таким чином, за одне порівняння всі ПЕ можуть паралельно застосувати потрібний обмін порівнюваними числами. За рахунок застосування відповідно скомпонованої топології можна оптимізувати програму так, щоб на кожному кроці можна було обійтись однією операцією обміну даними. Parallaxis-програма має вигляд:

```
1    SYSTEM sort;
2    (* Odd-Even Transposition Sort. (Паралельна реалізація)*)
3    CONST n = 10;
4    CONFIGURATION list [1.. n];
5    CONNECTION left   : list[i] -> list [i- 1].right;
6                        right : list[i] -> list [i+1].left;
7
8    SCALAR step : INTEGER;
9                a    : ARRAY[1.. n]  OF  INTEGER;
10
11  VECTOR val, r , l , comp :  INTEGER;
12                          lhs    :  BOOLEAN;
13
14  BEGIN
15   WriteString( 'Введіть, будь ласка, дані: ');
16   FOR step := 1  TO  n  DO  ReadInt(a[step])  END;
17   LOAD(val,a);
18
19   PARALLEL
20      lhs := ODD (id_no); (*ПЕ-це ліва частина порівняння*)
21      FOR step := 1 TO  n  DO
22        PROPAGATE . right (val, l);
23        PROPAGATE . left (val, r);
24        IF lhs THEN comp := r ELSE comp := l END;
25        IF lhs = (comp<val) THEN val := comp END; (*lhs&(comp<val)*)
26        lhs := NOT lhs; (*або (or) rhs & (comp≥val) *)
27      END;
28   ENDPARALLEL;
29
30   STORE(val,a);
31   FOR step:=1 TO n DO WriteInt(a[step],10); WriteLn END;
32 END sort.
```


15.5. Систолічне множення матриць

Множення двох матриць – це одна з найважливіших і найчастіше використовуваних паралельних операцій. Поряд з іншими вона є базовою операцією в комп'ютерній графіці і в робототехніці. Так зване "Систолічне множення" двох матриць – це ефективна версія алгоритму множення для SIMD-систем. На рис.15.6 пояснюється процес виконання цієї операції: вхідні матриці спочатку деформуються навкіс і потім покроково трансформуються в матрицю-результат. На кожному кроці для кожного елемента результуючої матриці-добутку перемножуються елементи матриць A і B і результат множення додається до елемента матриці-результата. За цим алгоритмом досить *(3*n-2)* кроків на n^2 ПЕ, щоб перемножити дві матриці розміром nxn кожна.

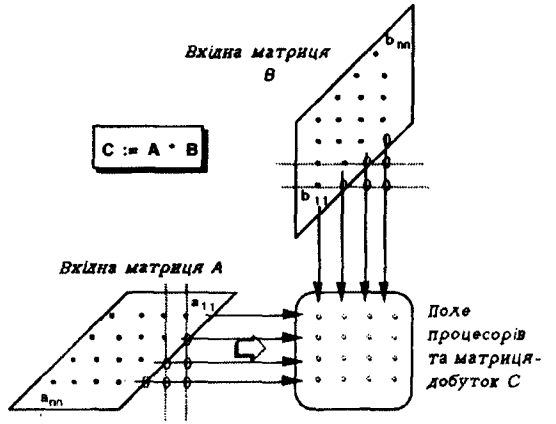


Рис. 15.6. Систолічне множення матриць

У Parallaxis-програмі цей паралельний алгоритм оптимізовано додатково і це привело до того, що результуюча матриця формується за n кроків. У процедурі matrix_mult обидві вхідні матриці (A і B) спочатку без зміни записуються в поле ПЕ оператором LOAD, а потім робиться підготовчий крок, в якому матриця A деформується навкіс вліво, а B – вгору. В наступних ітераціях A переміщується рядками вліво, а B – стовпцями вгору. Результат C=A*B видається в скалярне поле керуючої ЕОМ оператором STORE. Parallaxis-програма має такий текст:

```
1   SYSTEM systolic_array;
2   (* обчислення матричного добутку "c := a * b" *)
3    CONST max = 10;
4    TYPE matrix = ARRAY [1..max], [1..max] OF REAL;
5
6    CONFIGURATION grid [max], [max];
7    CONNECTION left: grid[i, j] -> grid[i, (j-1) MOD max].left;
8                 up : grid[i, j] -> grid[(i-1) MOD max, j].up;
9                 verA: grid[i, j] -> grid[i, (j-i) MOD max].verA;
10                verB: grid[i, j] -> grid[(i-j) MOD max, j].verB;
11
12   SCALAR i, j     :   INTEGER;
13          a, b, c :   matrix;
14
15
16   PROCEDURE matrix_mult(SCALAR VAR a,b,c : matrix);
17   (*множення матриць c := a * b *)
18   SCALAR k: INTEGER;
19   VECTOR ra, rb, rc : REAL;
20   BEGIN
21      LOAD (ra, a);
22      LOAD (rb, b);
23      PARALLEL
24         PROPAGATE.verA(ra);
25        PROPAGATE.verB(rb);
26         rc := ra * rb;
27         FOR k := 2 TO max DO
28            PROPAGATE.left (ra);
29            PROPAGATE.up (rb);
30            rc := rc + ra * rb;
31         END;
32      ENDPARALLEL;
33      STORE (rc, c);
34   END matrix_mult;
35
36   BEGIN
37      ...(* зчитування матриць a і b *)
38      matrix_mult(a,b,c);
39      ...(* видача матриці c *)
40   END systolic_array.
```

15.6. Генерування фракталей

Представлений тут алгоритм вирішує проблему, яка може бути синхронно розпаралелена за методом "Divide-and-Conquer" (розділяй і властвуй). Розпаралелювання ні в якому разі не може бути застосоване для всіх подібних алгоритмів, бо різні гілки деревоподібних алгоритмів найчастіше виконують різні частини програми. В алгоритмі, про який іде мова, генерується одновимірна фрактальна крива за допомогою переміщення центра відрізка ламаної лінії [Peitgen, Saupe 88]. Вихідне положення представлено відрізком прямої лінії, який в центрі зміщується вгору або вниз на виважену випадкову величину. Внаслідок цього утворюється два окремих відрізки прямої лінії з різними кутами нахилу. Обидва відрізки можуть бути далі оброблені паралельно, як і перший відрізок (рис. 15.7).

Рис. 15.7. Реалізація методу "Розділяй і властвуй" за допомогою деревоподібної структури

На кожному кроці подвоюється кількість відрізків прямої, що підлягають обробленню. Так продовжується доти, поки не буде досягнуто бажаного розкладу того відрізка, з якого і почали. Як процесорна топологія тут напрошується структура двоїчного дерева. Починаючи з кореня, на кожному кроці активізується наступний рівень дерева аж до самого листя. У зв'язку з тим, що для комунікації завжди активний тільки один рівень дерева, тут можна було б обійтися половиною процесорних елементів і складнішою структурою зв'язку (наприклад, гіперкубом).

У Parallaxis-програмі декларується проста деревовидна структура, в якій передаються початкові та кінцеві пункти відрізків прямих ліній (символом "**" визначається операція потенціювання). Функція Gauss генерує вектор реальних гауссових випадкових чисел, яку далі пояснювати не будемо. Процедура inorder також призначена для задач керування : вона видає в правильному порядку результуючі дані фрактальної кривої, що запам'ятовуються у вузлах-листях дерева процесорів.

Власне обробка відбувається в процедурі MidPointRec. В першому паралельному блоці активізується актуальний рівень дерева і переміщується центр відрізка прямої лінії. В другому паралельному блоці за умови, що рівень листя дерева ще не досягнуто, вводяться числові дані low і high вузлів-дітей деревоподібної структури. При цьому ліве дитя-вузол одержує числа low і x як його стартові дані, а праве дитя-вузол – числа x і high. В головній програмі після ініціалізації ітеративно викликається програма переміщення центра відрізків прямих. Кожен рівень самостійно веде паралельну обробку. Під час видачі даних, що зумовлюють вузли фрактальної кривої, треба дбати про правильну послідовність їх (виклик процедури inorder), бо елементи дерева запам'ятовуються в масиві не лінійно, а відповідно до рівня.

Parallaxis-програма має такий текст:

```
1   SYSTEM fractal;
2    CONST     maxlevel  = 7;
3              low_val   = 0.0;
4              high_val  = 1.0;
5              maxnode = 2**maxlevel - 1;
6   (* декларація структури дерева *)
7   CONFIGURATION tree  [1.. maxnode];
8   CONNECTION child_l: tree[i] <-> tree[2*i].parent;
9                      child_r: tree[i] <-> tree[2*i+1].parent;
10  SCALAR i        : INTEGER;
11         delta    : REAL;
12         field    : ARRAY [1..maxnode] OF REAL;
13  VECTOR x, low, high  :  REAL;
14
15  PROCEDURE Gauss(): VECTOR REAL;
16  (*генер. вектора випадк. чисел з розподілом Гаусса*)
..  ...
26  END Gauss;
27
28  PROCEDURE inorder(SCALAR node: INTEGER);
29  (* видача елементів дерева в лінійній послідовності *)
..  ...
36  END inorder;
37
38 PROCEDURE MidPointRec(SCALAR delta:REAL;
                          SCALAR level : INT EGER);
39  BEGIN (*вибір рівнів дерева*)
40   PARALLEL [2**(level - 1) .. 2**level - 1]
41      x := 0.5 * (low + high) + delta*Gauss() ;
42      IF level < maxlevel THEN (*значення для "дітей"*)
43        SEND tree.child_l (low)   TO tree.parent(low);
44        SEND tree.child_l  (x)    TO tree.parent(high);
45        SEND tree.child_r  (x)    TO tree.parent(low);
46        SEND tree.child_r  (high) TO tree.parent(high);
47      END;
48   ENDPARALLEL;
49  END MidPoint;
50
51  BEGIN (*головна програма*)
52   PARALLEL (*ініціалізація з стартовими даними*)
53        low    := low_val;
54        high   := high_val;
55        x      := 0.0
56   ENDPARALLEL;
57   FOR i := 1 TO maxlevel  DO
```

```
58        delta := 0.5 ** (FLOAT(i) / 2.0);
59        MidPoint (delta, i);
60      END;
61      STORE(x, field);
62    WriteFixPt (low_val,  10, 3); WriteLn;
63     inorder (1); (*друкування чисел послідовно*)
64     WriteFixPt (high_val,  10, 3); WriteLn;
65  END fractal.
```

Витрати часу на виконання цієї програми досить малі, бо алгоритм потребує лише $\log_2 n$ кроків, де n - кількість вузлів дерева на рівні листя. Це дорівнює якраз висоті дерева, тобто кількості рівнів дерева. На рис.15.8 показано фрактальну криву, що побудована за допомогою наведеної програми з 127 процесорними елементами, а на рис.15.9 подано інтерпретацію послідовності звуків, що відповідає початку іншої фрактальної послідовності чисел.

Рис. 15.8. Програмна генерація фрактальної кривої

Рис. 15.9. Фрактально згенерована послідовність звуків

15.7. Аналіз стереозображень

Просторовий зір – це винятково чарівний феномен. Особливо переконуємось у цьому, якщо беремо до уваги, що це природне для людей явище може бути реалізоване на комп'ютері тільки з величезними витратами процесорного часу. Алгоритми, що лежать в основі обробки

стереозображень, блискуче реалізуються в синхронно паралельних програмах.

У людей виникає тривимірне уявлення зору тому, що ліве і праве око сприймають об'єкти зору дещо по-різному завдяки невеликій різниці в кутах зору обох очей (рис.15.10: у лівого ока відстань A'B' більша, ніж відстань A"B" у правого ока). Зміщення відповідних пунктів у лівому і правому образах об'єкта зору називається "диспаритетом" (невідповідністю) і є мірою різниці висот між пунктами образів. На жаль, визначити коректне співвідношення між пунктами образів лівого та правого ока зовсім не просто.

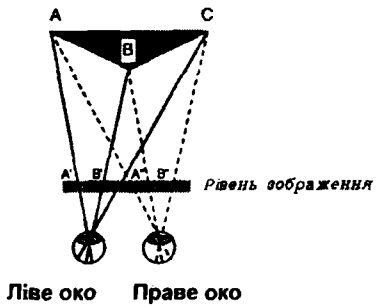
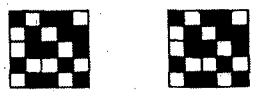


Рис. 15.10. Стереозір

Той факт, що людський мозок здатний інтерпретувати стереограми, не аналізуючи змісту об'єкта спостереження, опубліковано в працях [Julesz 60], [Julesz 78] (так звані "Random-Dot"-стереограми, тобто стереограми випадкових пунктів об'єкта зору). Лівий і правий образи генеровані випадково, якщо розглядати їх самі по собі, і не містять у собі змісту образу спостереження. Тривимірний об'ємний ефект виникає передусім через взаємодію обох образів на мозок, де аналізуються зміщення випадкових образів і перетворюються в деяке просторове сприйняття. Random-Dot-стереограми дуже легко реалізуються на комп'ютері, вони не мають недоліків фотознімків (неточність або проблема наведення). Вони також не потребують розпізнавання країв, бо Random-Dot-знімки складаються майже тільки з самих ліній країв зображень.

Порядок генерування Random-Dot-стереограм

1. Наповнення лівого і правого образів однаковими випадковими числами.



2. Переміщення вгору або вниз областей образу.

Об'єкт, що має вилучитися з поверхні зображення, зміщується відповідно до бажаної висоти на декілька пікселів у правому знімку вліво (або вправо, якщо він має сприйматися за рівнем образу). Пусті місця, що з'являються після зміщень, заповнюються заново випадковим зображенням. Лівий знімок залишається без змін. Цей крок повторюється для кожної області, що представляється об'ємно.

Показ стереограм

Для розгляду стереограм існує ряд різних технічних засобів, які передусім дбають про те, щоб лівий знімок розглядався тільки лівим оком, а правий – відповідно тільки правим оком (наприклад, призми, LCD-Shutter, поляризаційний фільтр, червоно-зелений фільтр, двохекранний дісплей, засоби простого фіксування кожного знімка одним оком). Розгляньмо коротко метод, що застосовує червоно-зелені окуляри і може бути реалізований найпростішими засобами (метод Anaglyphen). Спочатку лівий знімок фарбується в зелено-білий колір, а правий – у червоно-білий. Потім обидва знімки накладаються один на одний і настільки легко

взаємозамінюються, що без стереоокулярів не можна помітити їхнього зміщення. Якщо дві кольорові крапки знімка друкуються послідовно одна на одну, то на папері або на екрані дісплея вони здаються однією чорною крапкою. Далі крапки зображень будемо називати пікселями (Pixel – абревіатура терміну Picture Element, широко вживаний термін в літературі з комп'ютерної графіки).

Червоний окулярний фільтр, що розміщений перед лівим оком, пропускає тільки червоний колір і тому зелений і чорний пікселі знімка (зображення) сприймаються оком як темні, а червоні пікселі навпаки, як світлі. Це точно відповідає початковому стану лівого знімка(зображення).

Зелений фільтр, що розміщений перед правим оком, пропускає тільки зелений колір, тому тут червоні та чорні пікселі сприймаються як темні. Через те правий знімок вифільтровується з накладених одна на одну стереограм (рис.15.11). Досвід показує, що синтезуюче обчислення стереограм, тобто одержання інформації про висоти пікселів з лівого та правого знімків, значно важче і потребує більше часу, ніж генерування стереограм. Крім того, ці обчислення не можуть бути на всі 100% коректними; як правило, трапляються окремі пункти зображення, яким приписуються неправильні значення висот.

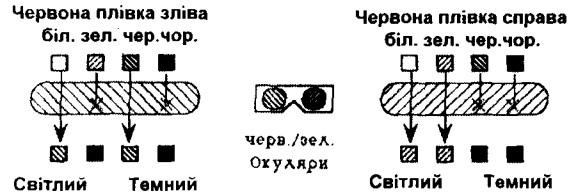
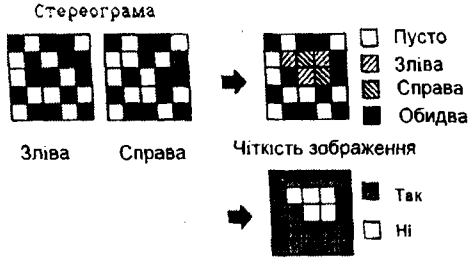


Рис. 15.11. Дія стереоокулярів

Аналіз Random-Dot-стереограм (обчислення інформації про висоти із стереограм)

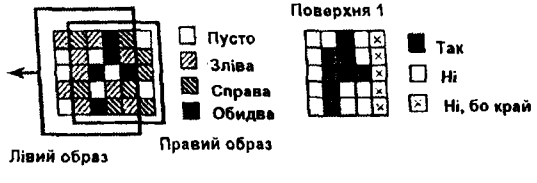
1. Суміщення лівого та правого зображень і пошук збігів.

Збіг пари відповідних пікселів зображення (відповідно по одному пікселю з лівого і правого зображень, що мають однакове розміщення) має місце тільки тоді, коли обидва пікселі є чорними або білими. Ця операція може виконуватися паралельно над усіма пікселями зображення (див. рисунки, де пікселям відповідають квадратики).



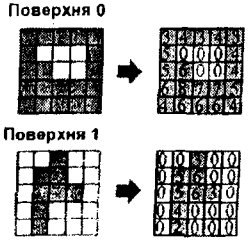
2. Зміщення лівого зображення (один піксель вліво) та порівняння з правим зображенням.

Цей етап проводиться ітеративно для кожного рівня висот. Внаслідок зміщення і порівняння одержують для кожної поверхні висот пікселі, що збігаються в обох зображеннях (див. рисунок).



3. Визначення оточення пікселів для кожного рівня висот.

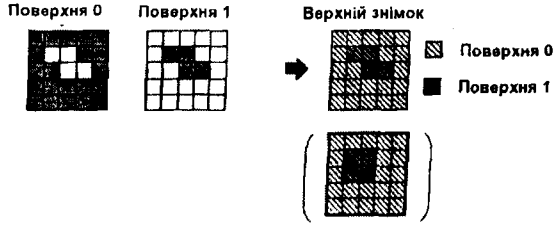
Цей крок виконується ітеративно для кожного рівня висот і паралельно для всіх пікслелів. Дані з локального поля сусідніх елементів (квадратики на рисунку, пікселі) розміром 5x5 (на рисунку береться поле 3x3 всередені квадрата 5x5) застосовуються для аналізу оточення пікселів і при цьому використовується восьмивимірний решітчастий зв'язок між пікселями. Пікселі, що не збігаються, набувають значення 0, пікселі із збігом набувають число, що дорівнює кількості сусідніх пікселів, в яких наявний збіг, плюс одиницю для самих себе (на рисунках у квадратиках показано результуючі числа; так, 5=1+1+0+1+1+1 – перший піксель у другому порядку).



4. Вибір найбільш підхожого рівня для кожного пікселя.

Вибирається рівень (висота), який має найбільше числове значення (за однакових чисел – нижній рівень). Якщо числа оточення всіх рівнів дорівнюють нулю, то вибирається рівень сусіднього пікселя. Ця обробка може вестись паралельно для всіх пікселів.

Дані всіх рівнів можна тепер звести в загальну картину висот ("знімок з фальшивим кольором"), як тут показано. Колір кожного пікселя дає при цьому його висоту. Рішення, що знайдено в цьому прикладі (див. рисунок "картина висот"), не зовсім коректне, бо не вдалось уникнути різниці в один піксель. Коректна картина висот показана на рисунку в дужках.



5. Фільтрування (за вибором).

Щоб виключити помилкові пікселі на кожному рівні локального сусіднього поля розміром 3x3 складаються ті пікселі, що були впорядковані. Наприкінці гасяться ті пікселі, в яких дуже мало пікселів-сусідів (кількість менше, ніж порогове число). Погашені пікселі розміщуються потім на новому рівні. Ця операція фільтрації може також виконуватися паралельно.

Продемонстрований тут алгоритм аналізу Random-Dot-стереограм може легко трансформуватися в паралельну за потоком даних програму. На рис. 5.12 показано стереограму і відповідно до неї розраховану картину висот (з деякими неусуненими дефектами) розміром 128x128 пікселів. Під час застосування пар фотографічних знімків або відеозображень, які можуть бути зчитані в ЕОМ через сканер або зовнішній буфер, проблема дуже ускладнюється. Фотографії не є монохромними, вони містять сірі ступені, орієнтація обох знімків відносно один одного з точністю до пікселя неможлива, з'являється цілий ряд неточностей та хиб зображення, які не можуть мати місця при комп'ютерному генеруванні Random-Dot-стереограм. Для фотознімків треба виконати два попередніх кроки перед обробкою. По-перше, розрахунки "відносної орієнтації" обох знімків один до одного, що пов'язано з відповідним пристосуванням одного з двох знімків. По-друге, визначення країв обох знімків, бо вони означають більше, ніж дані сірих знімків.

На цій правильно орієнтованій обмеженій стереограмі можна застосувати алгоритм стерео-Matching.

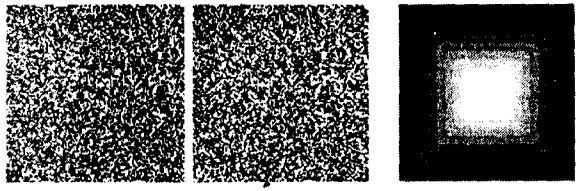


Рис. 15.12. Стереограма типу "Random-Dot" з обчисленням висот

Вправи до розділу III

1. Структуру зв'язків між процесорами на мові Parallaxis задано так:

```
CONFIGURATION  seltsam [0..Max-1];
CONNECTION     a:seltsam[i]<->seltsam[3*i+1].d;
               b:seltsam[i]<->seltsam[3*i+2].d;
               c:seltsam[i]<->seltsam[3*i+3].d,
```

a) зробіть ескіз схеми комутації seltsam, що має параметр Max=10;

b) знайдіть величину V – кількість ліній зв'язку кожного ПЕ для схеми seltsam;

c) яку величину має параметр A, максимальна відстань між двома параметрами в seltsam ?(залежно від n, кількості ПЕ).

2. Для наведеного далі фрагмента Parallaxis-програми дайте відповідь на такі запитання:

а) який результат буде, якщо вектору відповідає матриця:

$$V = \begin{pmatrix} 3 & 2 & 5 \\ 4 & 7 & 8 \\ 9 & 1 & 6 \end{pmatrix};$$

б) яку операцію виконує ця програма над деякою матрицею загального виду ?

```
CONFIGURATION feld[1..max],[1..max];
CONNECTION rechts : feld[i,j]<->feld[i,j+1].links;

SCALAR i, ergebnis:integer;
VECTOR  puffer, v :integer;

BEGIN
   PARALLEL
      ...(*завантажити дані у вектор V*)
      FOR i:=1 TO max-1 DO
```

```
        puffer:=v;
        PROPAGATE.links(v);
        IF puffer>v THEN v:=puffer END;
      END;
      ergebnis:=REDUCE.min [*],[1] (v);
    ENDPARALLEL
```

....

3. Перекладіть наступну програму, написану на Фортрані-90, в Parallaxis-програму (".LT." застосовується як "less than" або "<")

Фортран-90:

```
INTEGER, DIMENSION(100,500) :: Matrix
...
Matrix(1:50,1:500) = Matrix(2:51,1:500);
WHERE (Matrix .LT. 7) Matrix = 10;
```

Parallaxis:

```
SYSTEM FtoP;
CONFIGURATION feld[1..100],[1..500];
CONNECTION rechts:feld[i,j]<->feld[i,j+1].links;
           oben :feld[i,j]<->feld[i+1,j].unten;

VECTOR Matrix:INTEGER;

BEGIN
.....
END FtoP.
```

4. Для наведеного нижче фрагмента Parallaxis-програми:

а) який результат одержимо при такому програмному вводі програми:

3 2 5 3 9 9 1 2 3 7 2 4 8 5 6 2

Визначте вид вектора f на початку роботи програми і після кожного проходу циклу;

б) яку операцію виконує ця програма над заданою послідовністю чисел ?

в) скільки операцій обміну даними (PROPAGATE) виконується за один проход програми ?


```
SYSTEM was_bin_ich;
CONST ebenen = 5;
      anzahl = (2**ebenen)-1;
      anfang = 2**(ebenen-1);
      ende = anzahl;
CONFIGURAATION baum [1..anzahl];
CONNECTION kind_l:baum[i]<->baum[2*i ].eltern_l;
           kind_r:baum[i]<->baum[2*i+1].eltern_r;
VECTOR f,links,rechts : integer;
SCALAR    feld:ARRAY [1..anzahl] OF integer;
      z,x,ergebnis :integer;

BEGIN
(*завантажити дані у вектор f*)
FOR z:=1 TO anfang-1 DO feld[z]:=0; END;
FOR z:=anfang TO ende DO ReadInt (feld[z]) END;
LOAD(f, feld);
FOR x:=1 TO ebenen-1 DO
    PARALLEL
      PROPAGATE.eltern_l(f,links);
      PROPAGATE.eltern_r(f,rechts);
      IF links<rechts THEN f:=links ELSE f:=rechts; END;
    ENDPARALLEL;
END;
STORE[1](f,ergebnis);
WriteInt(ergebnis, 4);
END was_bin_ich.
```

5. Напишіть програму систоличного множення матриць на мові Фортран-90.

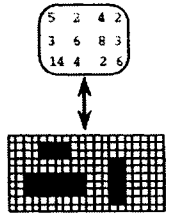
6. Напишіть програму генерування простих чисел з ситом Ерастофена на мові C*.

7. Напишіть програму паралельного розв'язання системи лінійних рівнянь на мові Parallaxis.

8. Напишіть Parallaxis-програму, яка читає числа прямокутників з даних, що їх описують, і розміщує їх у паралельному полі як числа зображення (див. рис.). Дані можуть бути відкритими або закритими для читання-запису такими стандартними операціями:

```
OpenInput ("filename") CloseInput
OpenOutput("filename") CloseOutput
```

Кожний рядок даних містить запис такої форми:
<Стартова позиція X><Стартова позиція У><Ширина><Висота>



9. Напишіть Parallaxis-програму, яка використовує дані зображення, що генеруються в завданні 8. Прямокутники, що містяться в паралельному полі, мають розпізнаватись і заново відображатись у вигляді даних, що їх описують.

10. Реалізуйте на мові Parallaxis клітчастий Conway автомат "Game of life". Застосуйте для кожної клітини один віртуальний ПЕ.

11. В алгоритмі сортування OETS (п.15.4) проводяться дві операції обміну даними на кожному кроці, незважаючи на те що кожний ПЕ потребує тільки число свого ПЕ-сусіда. Перепишіть програму за допомогою складних (умовних) структур зв'язку так, щоб вона могла обійтися лише однією операцією обміну даними на кожному кроці сортування.

12. Напишіть Parallaxis-програму для генерування Random-Dot-стереограм. Для генерування булівського випадкового вектора можна застосувати безпараметрову стандартну функцію VBRandom().

13. Напишіть Parallaxis-програму для аналізу Random-Dot-стереограм (одержання інформації про висоти).

IV

ІНШІ МОДЕЛІ ПАРАЛЕЛЬНОСТІ

У процедурних мовах паралельного програмування, що були розглянуті в попередніх розділах, має місце безпосередній контроль паралельності за допомогою мовних конструктивів спеціального призначення. Однак в ідеальному випадку програміст не має потреби засереджуватися на деталях паралельного виконання своєї програми, бо цю задачу вирішує комп'ютер або операційна система непомітно для користувача. Лише тимчасовим рішенням на шляху до цієї мети є автоматичне розпаралелювання або векторизація послідовних процедурних програм, бо за його допомогою часто не можна досягти задовільного підвіщення продуктивності. У непроцедурних мовах має місце як неявне розпаралелювання, так і явні паралельні концепції, аналогічні розглянутим в процедурних мовах. Нейронні мережі – це також одна з форм описування паралельних процесів, одначе на більш високому рівні абстракцій. Важливе значення для всіх паралельних моделей і застосування паралельних ЕОМ взагалі має оцінювання продуктивності обчислювальних систем цього класу.

16. АВТОМАТИЧНЕ РОЗПАРАЛЕЛЮВАННЯ ТА ВЕКТОРИЗАЦІЯ

Як було вже показано для MIMD-систем, написання паралельних програм пов'язане порівняно з звичайними послідовними програмами з більшою складністю, більшими витратами, більш високим ризиком допуститися помилок. До того ж у багатьох користувачів експлуатуються великі бібліотеки послідовних програм, які мали б використовуватися в подальших розробках, але з огляду на їхні неосяжні розміри (а також не в останню чергу через брак потрібної документації) ці програмні доробки не можуть бути переписані у відповідні паралельні варіанти без докорінного переосмислення та опрацювання. Висока вартість розробки нових програмних засобів суттєво обмежує потенційно можливе коло користувачів (і покупців) паралельних систем, тому порівняно із звичайними ЕОМ паралельні системи як спеціальні засоби вирішення проблем з інтенсивними розрахунками мають незначну частину комп'ютерного ринку. Одна з надій на ширший вихід паралельних систем на ринок пов'язана з можливістю автоматичного розпаралелювання (векторизації) послідовних програм. Реалізація цієї ідеї дала б можливість за допомогою нового транслювання використати всі наявні послідовні програми, не розробляючи спеціально нових паралельних програм. На жаль, на сьогодні автоматичне розпаралелювання і векторизація відповідної якості поки що неможливі і цілком резонно виникає питання, чи взагалі в якомусь майбутньому це може стати реальністю.

Нема жодного сенсу писати програми на звичайній послідовній мові, довіряючи автоматичному розпаралелюванню чи векторизації. Наприклад, скалярний добуток, який запрограмований великими зусиллями користувача в послідовному циклі (й до того ж кожним

користувачем по-різному!), сприймається автоматичним розпаралелювачем/векторизатором як послідовна програмна реалізація і має транслюватись у відповідні паралельні або векторні операції. Набагато простіше написати цей алгоритм на мові паралельного програмування. Доповненням до неї міг би бути значно простіший в імплементації "автоматичний розпослідовлювач", тобто транслятор паралельних програм у послідовні для тих випадків, коли паралельна програма має виконуватися на послідовній ЕОМ.

На ринок постачається векторизатор-перетворювач "VAST-2" програм, написаних на Фортрані-77, в програми MPFortran (діалект Фортрана-90), розроблений фірмою MasPar Computer Co. [MasPar 92]. VAST-2 перетворює послідовні фортранівські DO-цикли та IF-вибори у відповідні ARRAY-вирази; цикли із складними структурами перетворюються в операції над багатовимірними масивами. При цьому враховується взаємозалежність даних (розглядається далі) різних операторів і робиться спроба мінімізувати кількість і величину таких фрагментів програм. Продуктивність цього векторизатора значною мірою залежить від інтерактивної взаємодії з програмістом. У процесі векторизації компілятор чекає перекладених директив від програміста; крім цього, відповідно до діагностичних повідомлень векторизатора програміст повинен реструктурувати деякі частини в програмному коді. Таким чином, генерування паралельної програми тут відбувається у вигляді ітеративного процесу. Цілком ймовірно, що недосвідченому в паралельному програмуванні користувачу буде не під силу генерувати за допомогою векторизатора VAST-2 ефективні паралельні програми.

У цій главі розглядається як розпаралелювання послідовних програм для MIMD-систем (асинхронна паралельність), так і векторизація послідовних програм для SIMD-систем (синхронна паралельність).

Поряд з псевдотекстами програм застосовуємо тут і тексти на Фортрані-90 [Metcalt, Reid 90], які зручні для векторного програмування на SIMD- або MIMD-системах. Фортран-90 описано в п.14.1. Розгляньмо приклад векторизації for-циклу:

```
for i:= 1 to n do
        A[i] := b[i]+C[i];
        D[i] := A[i]*5;
end ;
```

За правилами мови Фортран-90 маємо:

```
A(1 : n) = B(1 : n) + C(1 : n)
D(1 : n) = A(1 : n)*5
```

У зв'язку з тим, що в кожному проходженні циклу виконуються однакові операції над елементами масиву і ці елементи незалежні між собою, є можливість перетворити кожну інструкцію всередині for-циклу у відповідну векторну інструкцію. Запис виду A(1 : n) означає в Фортрані-90 деякий вектор з n компонентами.

Увага:

```
for i := 1 to n do
  A[i] := B[i] + C[i];
  D[i] := A[i+1]*5;
end;
```

Використовується старе значення A

```
A(1 : n) = B(1 : n) + C(1 : n)
D(1: n) = A(2 : n+1)*5
```

Використовується нове значення A

Правильною є змінена послідовність операцій:

```
D(1: n) = A(2 : n+1)*5
A(1: n) = B(1: n) + C(1: n)
```


Як видно з цього прикладу, заміна навіть одного індекса (A[i+1] замість A[i]) помітно ускладнює векторизацію. Тут маємо типову взаємозалежність даних: у кожному проході циклу скалярний оператор №2 (D[i] :=...) звертається до старих, ще не змінених числових даних масиву A. Це має бути обов'язково враховано під час векторизації. Тільки детальним переглядом програми можна виявити, що бажаний результат досягається простою зміною послідовності векторних операторів D(1:n), A(1:n). Ці взаємозалежності і правила врахування їх детальніше розглянуто в наступних розділах.

16.1. Взаємозалежність даних

Перед початком розпаралелювання або векторизації послідовної програми треба визначити взаємозалежності даних, які обробляються різними операторами (інструкціями) і обумовлюють відповідні залежності між останніми. Наступні визначення й способи базуються на результатах, опублікованих в [Kuck, Kuhn, Leasure, Wolfe 80] і [Hwang, DeGroot 89].

Допоміжні визначення

Вхідна множина деякої інструкції (оператора) *S*:

IN(S) = "Множина всіх елементів даних, значення яких читає оператор *S*".

Вихідна (результуюча) множина деякої інструкції (оператора) *S*:

OUT(S) = "Множина всіх елементів даних, значення яких змінює оператор *S*".

Пояснімо ці визначення на прикладі простого for-циклу:

```
        for i :=1 to 5 do
S:        X[i] := A[i+1]*B
        end;
```

"IN"-множина містить усі вхідні елементи даних оператора S для всієї області циклу, а "OUT"-множина – відповідні елементи даних, що з'являються як результати оператора S в циклі, а саме:

IN(S) = {A[2], A[3], A[4], A[5], A[6], B}
OUT(S) = {X[1], X[2], X[3], X[4], X[5]}.

Визначення послідовності виконання:

Якщо оператор S має в циклі індекс i, то символом $S^{i'}$ позначається операція, яку виконує S під час проходження циклу i=i'.

$S_1 \Theta S_2 :\Leftrightarrow$ операція S_1 може виконуватися перед операцією оператора S_2 в процесі реалізації програми;

$S_1^{i'} \Theta S_2^{i''} :\Leftrightarrow$ оператори S_1 і S_2 включені в один і той же цикл; $S_1^{i'}$ виконується перед $S_2^{i''}$.

Визначення порядку, в якому виконуються два оператори, дуже важливе для виявлення їхніх взаїмозалежностей від даних. Тут і в наступних спрощуючих припущеннях будемо вважати, що :

а) змінна циклу (інкремент) завжди дорівнює 1;

б) розглядаються тільки операції присвоєння.

Визначення реляцій залежності даних

Залежність даних визначається передусім для інструкцій (операторів), яких немає в деякому циклі, а також для операцій (інстанцій), що виконуються інструкціями (операторами) (тобто для операцій в межах відомого проходу циклу; індекси циклу для наочності записів опущені).

(1) $\exists x: x \in OUT(S_1) \wedge x \in IN(S_2) \wedge S_1 \Theta S_2 \wedge \nexists k: (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT(S_k))$

$\Leftrightarrow$: S_2 є поточно залежним від S_1: $S_1 \delta S_2$

S_2 користується значенням X, яке обчислене в S_1

(2) $\exists x: x \in IN(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \wedge \nexists k: (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT((S_k))$

$\Leftrightarrow$: S_2 є зворотно залежним від S_1: $S_1 \delta S_2$

S_1 використовує значення X до того, як воно змінюється в S_2

(3) $\exists x: x \in OUT(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \wedge \nexists k: (S_1 \Theta S_k \Theta S_2 \wedge x \in OUT((S_k))$

$\Leftrightarrow$: S_2 є залежним за виходами результата від S_1: $S_1 \delta S_2$

S_2 переписує значення X , яке до того обчислене в S_1

Якщо ні *(1)*, ні *(2)*, ні *(3)* не діють, то S_1 і S_2 є незалежними за даними.

Два оператори в одному циклі з індексом i визначаються як такі, що мають залежність даних, якщо існують дві операції цих операторів, які залежні одна від одної. Те саме справедливо для складних циклів:

$S_1 \ \delta \ S_2 : \Leftrightarrow \exists \ i', i'': S_1^{i'} \ \delta \ S_2^{i''}$

$S_1 \ \bar{\delta} \ S_2 : \Leftrightarrow \exists \ i', i'': S_1^{i'} \ \bar{\delta} \ S_2^{i''}$

$S_1 \ \delta^\circ \ S_2 : \Leftrightarrow \exists \ i', i'': S_1^{i'} \ \delta^\circ \ S_2^{i''}$

Таким чином, два оператори (інструкції) програми можуть бути в трьох видах взаємозалежності за даними: поточна залежність (Flow-Dependence), зворотна залежність (Anti-Dependence), залежність від вихідних результатів (Output-Dependence). Пояснімо їх:

Flow-Dependence.

Оператор S_2 читає значення даних, яке перед тим було записане оператором S_1: через цю залежність має зберігатися порядок виконання S_1 перед S_2 і в паралелізованому (векторизованому) програмному коді, бо інакше оператор S_2 читав би неправильне числове значення.

Anti-Dependence.

Оператор S_2 переписує число, яке щойно було зчитано оператором S_1. У цьому випадку автоматичне розпаралелювання (векторизація) має зберегти ту саму послідовність виконання S_1 перед S_2, інакше в паралельному коді читалося б не те число.

Output-Dependence.

Спочатку оператор S_1 переписує число, яке наприкінці заново переписується оператором S_2. Як і в перших двох випадках, під час автоматичного розпаралелювання (векторизації) має зберегтися послідовність виконання S_1 перед S_2, інакше описана змінна набула б неправильного значення (число від S_1 замість числа від S_2) після виконання обох операцій.

Спрощені правила.

Наведене вище визначення реляцій дуже складно реалізувати через правила, що заперечують існування ("немає жодного k з..."). На противагу цим достатнім і необхідним правилам ("точно тоді, якщо") часто можна обійтися спрощеними правилами, які наведено нижче, і хоча вони і є необхідними, але вони недостатні. Перевірка операторів, розміщених між S_1 та S_2, може тут не проводитись. Умови для нових лівих сторін (див. нижче) є необхідними для незалежності правих сторін; зворотний порядок, як правило, недійсний.

$$(1')\ \exists\ x:\ x \in OUT(S_1) \wedge x \in IN(S_2) \wedge S_1 \Theta S_2 \Leftarrow S_1\ \delta\ S_2$$
$$(2')\ \exists\ x:\ x \in IN(S_2) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \Leftarrow S_1\ \bar{\delta}\ S_2$$
$$(3')\ \exists\ x:\ x \in OUT(S_1) \wedge x \in OUT(S_2) \wedge S_1 \Theta S_2 \Leftarrow S_1\ \delta^{\circ}\ S_2$$

В обмежених реалізаціях можна було б застосовувати ці спрощені правила, але це означає, що було б знайдено в цьому випадку "забагато" залежностей операторів від даних (тобто і такі пари реляцій, що не є залежними). За таких умов неможливо провести оптимальне розпаралелювання. З іншого боку, за цими простими правилами не пропускається жодна залежність від даних, тобто автоматично розпаралелені програми були б коректними.

Різницю в правилах (1), (2), (3), і (1'),(2'),(3') можна пояснити на такому прикладі:

S_1: A := B + D;

S_2: C := A * 3;

S_3: A := A + C;

S_4: E := A / 2;

Як легко бачити, ця послідовність інструкцій містить такі залежності даних:

$S_1\ \delta\ S_2$ (через A)

$S_1\ \delta\ S_3$ (через A)

$S_2\ \delta\ S_3$ (через C)

$S_3\ \delta\ S_4$ (через A)

$S_2\ \bar{\delta}\ S_3$ (через A)

$S_1\ \delta^{\circ}\ S_3$ (через A)

Одначе недійсне твердження

$S_1 \ \delta \ S_4$, незважаючи на те, що формально
$S_1 \Theta S_4$ та $\mathrm{OUT}(S_1) \cap \mathrm{IN}(S_4) = \{A\}$.

У даному випадку "спрощене правило" (1') знаходить залежність від даних там, де її не існує за правилом (1): реально числове значення змінної A хоч і було записане оператором S_1, одначе "по дорозі" від S_1 до S_4 було переписане оператором S_3. Якраз цей проміжний оператор і не враховується в спрощеному правилі (1').

Визначення непрямої залежності даних

Це визначення об'єднує три види залежностей даних і розширює їх до деякого ланцюга залежностей :

S_2 залежить від даних $S_1 : S_1 \ \delta^* S_2$

$: \Leftrightarrow S_1 \ \delta \ S_2 \ \vee \ S_1 \ \overline{\delta} \ S_2 \ \vee \ S_1 \ \delta^\circ \ S_2$

S_2 непрямо залежить від даних S_1: $S_1 \ \Delta \ S_2$

$: \Leftrightarrow \exists \ S_{k1}, S_{k2}, ..., S_{kn} (n \geq 0): S_1 \ \delta^* \ S_{k1} \ \delta^* \ S_{k2} \ \delta^* ... \delta^* \ S_{kn} \ \delta^* \ S_2$

Визначення спрямованої залежності даних

Напрямок залежності даних вказує на те, чи є залежні операції в одному й тому ж проході циклу або може одна з них уже виконана в щойно закінченому проході циклу (з відповідно іншим значенням індексу). У зв'язку з тим, що інструкції (оператори) можуть бути в багатьох структурованих циклах, для кожного циклу має задаватися свій напрям.

Припустімо, що S_1 і S_2 входять в d циклів з індексами $i_1, ..., i_d$. Якщо існують дві певні циклові інстанції $I' = (i_1', ..., i_d')$ та $I'' = (i_1'', ..., i_d'')$ для циклових індексів $i_1, ..., i_d$, таких, що для відповідних інстанцій S_1 та S_2 справедливо

$S_1^{i_1' ... i_d'} \ \delta^* \ S_2^{i_1'' ... i_d''}$

і якщо для цих обох індексних векторів діє реляція $I' \ \Psi \ I''$ (це означає : $\Psi = (\Psi_1, ..., \Psi_d)$, причому $\Psi_i \in \{<, =, \leq, >, \geq, \neq, ?\}$, (знак "?" є деякою невідомою реляцією), що дає

$i_1' \ \Psi_1 \ i_1''$
$i_2' \ \Psi_2 \ i_2''$
...
$i_d' \ \Psi_d \ i_d'' \ )$,

то можна дати визначення

$\Leftrightarrow$: *S_2 залежить за даними з напрямком* Ψ від S_1 :

$S_1 \ \delta_\Psi^* \ S_2$

Спрямовані залежності даних $\delta_\Psi, \overline{\delta}_\Psi, \delta_\Psi^\circ$ визначаються так само.

Приклад напрямку залежності за даними.

```
for i:=1 to n do
      for j:=2 to m do
S1:     A[i, j] := B[i, j];
S2:     C[i, j] := A[i, j-1];
      end;
   end;
```

Тут має місце $S_1 \ \delta_\Psi \ S_2$, бо операція із S_1 записує значення A[i,j], яке читається деякою операцією із S_2. Напрямок залежності даних можна встановити порівнянням індексів. Наприклад, в проході циклу з i=2, j=2

записується елемент A[2,2], а в проході з i=2, j=3 той самий елемент A[2,2] читається. Порівняння індексів дає:

2=2 (для зовнішнього циклу з індексом i),

2<3 (для внутрішнього циклу з індексом j).

Це означає, що між S_1 і S_2 існує така спрямована залежність даних:

$S_1 \ \delta_{(=,<)} \ S_2$.

Можна в спрощеному вигляді сформулювати таке правило:

> Якщо є залежність даних і індекс деякої змінної під час доступу запису дорівнює i, а під час доступу читання дорівнює:
>
> **i+1** – то читається старе значення змінної;
>
> **i–1** – то читається нове значення змінної.

У наступних параграфах розглядається векторизація (для синхронної паралельності, SIMD-комп'ютер) і розпаралелювання (асинхронна паралельність, MIMD-комп'ютер) паралельних циклів. У складних структурованих циклах з міркувань ефективності (витрати ресурсів на старт і термінування процесів), як правило, для розпаралелювання застосовують зовнішні цикли, а для векторизації – внутрішні.

16.2. Векторизація циклу

Після введених формальних визначень розгляньмо тепер методи автоматичного перетворення послідовних програм у паралельні.

Правила векторизації:

> • якщо існує залежність даних операторів $S_x \ \delta^* \ S_y$ в межах циклу, що підлягає векторизації, то у векторизованому коді оператор (інструкція) S_x має виконуватись перед оператором (інструкцією) S_y.
>
> • залежності даних з напрямками "<" або ">" в наявних вміщаючих циклах можуть не братись до уваги.
>
> • якщо за наявності багатьох залежностей даних неможливо встановити однозначну чітку послідовність операторів (інструкцій), то такий цикл не може бути векторизований безпосередньо.

Приклад векторизації (за Вольфе [Hwang, DeGoot 89])

```
        for i:=1 to n do
S1:       A[i] := B[i] + C[i];
S2:       D[i] := A[i+1] + 1;
S3:       C[i] := D[i];
        end;
```

Спочатку мають бути виявлені залежності даних всіх трьох інструкцій циклу. Із кожної окремої залежності формулюється умова для послідовності паралельного виконання інструкцій.

У векторизованому коді має бути:

$S_1 \ \overline{\delta_{(=)}} \ S_3$ (через C) $\Rightarrow$ S_1 перед S_3

$S_2 \ \overline{\delta_{(<)}} \ S_1$ (через A) $\Rightarrow$ S_2 перед S_1

$S_2 \ \overline{\delta_{(=)}} \ S_3$ (через D) $\Rightarrow$ S_2 перед S_3

У наведеному прикладі є три залежності даних, які треба взяти до уваги під час векторизації. Як зворотні залежності $S_1\ \overline{\delta_{(=)}}\ S_3$, $S_2\ \overline{\delta_{(<)}}\ S_1$, так і поточна залежність $S_2\ \overline{\delta_{(=)}}\ S_3$ визначають порядок виконання інструкцій у векторизованому коді, тобто S_1 перед S_2, S_2 перед S_1 і S_2 перед S_3.

Можна зробити висновок, що з цих трьох умов виникає така послідовність операторів: S_2, S_1, S_3.

Після виявлення цієї послідовності можна записати векторизовану версію цього циклу-прикладу в термінах Фортрана-90. Кожний оператор перетворений у векторне поле відповідно до меж циклу (від 1 до n):

```
S2: D(1 : n) = A(2 : n+1) + 1
S1: A(1 : n) = B(1 : n) + C(1: n)
S3: C(1 : n) = D(1 : n).
```

Розгляньмо приклад, що ілюструє правило векторизації внутрішніх циклів:

```
for i := 1 to n do
  for j := 1 to m do
    S1:  A[ i, j] := B[ i, j];
    S2:  C[ i, j] := A[ i-1, j];
  end;
end;
```

Тут маємо одну залежність даних:

$$S_1\ \delta_{(=,<)}\ S_2 \text{ (через A)}$$

Згідно з наведеними вище правилами можна цю залежність не брати до уваги, тобто цикл може бути векторизованим безпосередньо. Це правило діє для всіх залежностей даних типу $\delta^*_{(<,=)}, \delta^*_{(<,<)}, \delta^*_{(<,>)}$. Залежність даних має місце в подібних випадках між двома різними проходами зовнішніх циклів; залишаючись незмінною, вона не впливає на векторизацію. Завдяки цьому векторизована програма має вигляд:


```
do i = 1, n
   S1: A(i, 1 : m) = B(i, 1 : m)
   S2: C(i, 1 : m) = A(i-1, 1 : m)
end do
```

16.3. Розпаралелювання циклу

Під час розпаралелювання частин програм йдеться про застосування їх в асинхронному MIMD-комп'ютері. Принциповим тут є надання окремих циклових проходів різним процесорам, яке називається "do-across". Часто в наявності є менше процесорів, ніж кількість проходів циклу, що мають бути виконані, тому потрібен ще один рівень абстракції, на якому замість процесорів можна було б розглядати процеси. В цьому параграфі виходимо з того, що кількість фізичних процесорів має дорівнювати кількості процесів.

На відміну від векторизації циклу, де головна мета полягає в тому, щоб всі циклічні виконання деякої інструкції (оператора) зосередити в одній векторній команді, під час розпаралелювання робиться спроба кожному процесу (фізичному процесорові) доручити розрахунки щодо загальної послідовності інструкцій (команд, операторів) одного проходу циклу. На рис.16.1 векторизація і розпаралелювання представлені як горизонтальний і вертикальний способи розподілу обчислювальних робіт.

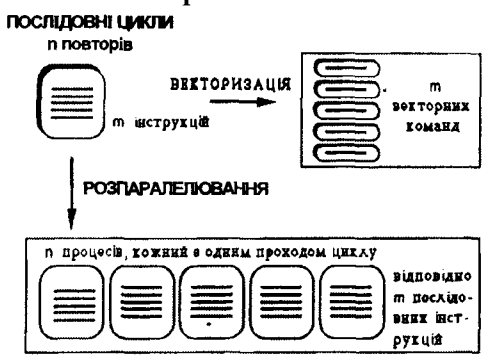


Рис. 16.1. Розпаралелювання та векторизація

При наявності залежності даних для розпаралелювання можуть знадобитися допоміжні оператори синхронизації процесів. Якщо, наприклад, існує залежність даних між двома проходами циклу (а під час розпаралелювання – між двома процесами), то кожний процес має чекати на появу залежних від іншого процесу даних, що обчислюються у відповідному проході циклу. Ці оператори синхронізації зумовлюють часові інтервали очікування процесів і зменшують загальну продуктивність паралельної програми.

Подальший аналіз показує, що не всі залежності даних потребують синхронізації під час розпаралелювання. У зв'язку з тим що на відміну від векторизації послідовність всіх інструкцій в межах циклу залишається незмінною, можна не брати до уваги залежності даних в межах проходу циклу.

Правила розпаралелювання:

а) залежності даних напрямку "=" розпаралелюваного циклу не треба синхронізувати;

б) залежності даних "<" або ">" в можливо наявних зовнішніх циклах при розпаралелюванні циклів можуть не братися до уваги;

в) можливо наявні внутрішні цикли можуть під час розпаралелювання не братись до уваги: вони переносяться в паралельний код комплектно без змін;

г) будь-яка інша залежність даних має синхронізуватися за допомогою свого власного масиву семафорів, що визначають послідовність виконання процесів;

д) для підвищення ефективності паралельної програми може бути змінена послідовність виконання інструкцій в межах наявних залежностей даних; отже, якщо має місце залежність $S_x \ \delta^*_{(=)} \ S_y$

всередені циклу, то в паралелізованому коді інструкція S_x має виконуватися перед інструкцією S_y.

Приклад до правила а):

```
        for i := 1 to n do
S1:       A[i] := C[i];
S2:       B[i] := A[i];
        end;
```

Тут має місце залежність даних типу $S_1 \ \delta_{(=)} \ S_2$ (через A[i]) і проявляє себе в одному й тому ж проході циклу, який під час розпаралелювання має виконуватись тим самим процесом. У цьому випадку синхронізація не потрібна.

За допомогою паралельного мовного конструктиву doacross виконується розподіл: кожному проходові циклу надається процес (в ідеальному випадку – свій процесор). Розпаралелений цикл має такий вигляд:

```
        doacross i := 1 to n do
S1:       A[i] := C[i];
S2:       B[i] := A[i];
        enddoacross;
```

Приклад до правила б):

```
        for i := 1 to n do
          for j := 1 to m do
S1:         A[i, j] := C[i, j];
S2:         B[i, j] := A[i-1, j-1];
          end;
        end;
```

Тут існує залежність даних $S_1 \ \delta_{(<,<)} \ S_2$ (через A[i,j]) між двома проходами зовнішнього циклу (напрямок

залежності "<"). Якщо в цьому прикладі має розпаралелюватися *тільки внутрішній цикл*, то потреба в синхронізації відпадає.

Розпаралелений цикл:

```
                for i := 1 to n do
                  doacross j := 1 to m do
S1:                 A[i, j] := C[i, j];
S2:                 B[i, j] := A[i-1, j-1];
                  enddoacross;
                end;
```

Приклад до правила в)

Має бути розпаралелений зовнішній цикл :

```
          for i := 1 to n do
            for j := 1 to n do
S1:           A[i, j] := B[i, j];
S2:           B[i, j] := A[i, j-1];
            end;
          end;
```

Знайдімо всі залежності даних:

$S_1\ \delta_{(=,<)}\ S_2$ (через A)

$S_2\ \overline{\delta_{(=,=)}}\ S_1$ (через B)

$\Rightarrow$немає ніяких обов'язкових умов синхронізації

Якщо має розпаралелюватися зовнішній цикл, то обидві залежності даних не потребують синхронізації відповідно до наведених вище правил (частина а) : напрямок у першому індексі кожної залежності є "=", тобто залежність наявна в тому самому проході циклу. Це дає можливість виконати безпосереднє розпаралелювання за допомогою оператора *doacross*. В цьому випадку зовнішній цикл розподіляється між окремими процесами, а внутрішній цикл обробляється одним процесом послідовно.

Розпаралелювання внутрішнього циклу через наявність залежності даних напрямку "<" (другий індекс залежності від A) неможливе. Розпаралелювання зовнішнього циклу має такий вигляд:

```
 doacross i := 1 to n do
    for j := 1 to n do
S1:     A[i, j] := B[i, j];
S2:     B[i, j] := A[i, j-1];
    end;
enddoacross;
```

Приклад до правил г) і д) із [Hwang, DeGroot 89] :

```
for i := 1 to n do
S1:     A[i] := B[i]+C[i];
S2:     D[i] := A[i]+E[i-1];
S3:     E[i] := E[i]+2*B[i];
S4:     F[i] := E[i]+1;
end;
```

Аналіз залежностей від даних показує такі результати:

$S_1\ \delta_{(=)}\ S_2$ (через A[i])

$S_3\ \delta_{(=)}\ S_2$ (через E[i])

$S_3\ \delta_{(<)}\ S_2$ (через E[i])

Перші дві залежності мають індекси "=" і тому не синхронізуються. Вони тільки визначають, що в розпаралеленому програмному коді інструкція S_1 має виконуватися перед S_2, а S_3 – перед S_4. А це не так важливо, бо в самій послідовній програмі передбачено саме цей порядок виконання інструкцій.

Тільки остання залежність від даних має бути синхронізована, бо вона пов'язує між собою два різних

проходи циклу, тобто два процеси в розпаралеленій програмі. Це ясно із напрямку "<" залежності. Справедливо, наприклад, таке співвідношення:

$S_3^4 \ \delta \ S_2^5$ (через E[4], для індекса циклу маємо 4<5)

У проході циклу i=4 оператор S_3 записує число в E[4] і в проході i=5 оператор S_2 читає число із E[4]. З цієї причини оператор S_2 може виконуватися в кожному проході циклу тільки тоді, коли закінчився прохід циклу (процес) з виконанням оператора S_3. Це приводить до такої паралельної програми розпаралелювання циклу з синхронізацією:

```
var sync: array[1.. n] of semaphore [0];

doacross i := 1 to n do
S1:   A[i] := B[i] + C[i];
      if i>1 then P(sync[i-1]) end;
S2:   D[i] := A[i] + E[i-1];
S3:   E[i] := E[i] + 2*B[i];
        V(sync[i]);
S4::  F[i] := E[i] + 1;
enddoacross;
```

Процеси (або процесори, якщо вони є в достатній кількості) синхронізуються попарно за допомогою семафора, який ініціалізується нулем і має бути звільнений процесом-попередником операцією **V** (за винятком першого процесу). У цей час процес з наступним проходом циклу може виконати свою **P**-операцію. Наприклад, процес №2 може виконувати свою інструкцію S_1 паралельно з усіма

іншими процесами, але наприкінці він має у своїй **P**-операції чекати на відповідну **V**-операцію другого процесу над семафором sync[1]. Цю **V**-операцію над sync[1] виконує процес №1, але тільки після того, як він сам обробить S_1, S_2 і S_3. Таким чином, виникає більша затримка часу, яка з цих причин поширюється на всі процеси. Як видно з рис.16.2, виникає велика втрата ефективності розпаралелювання.

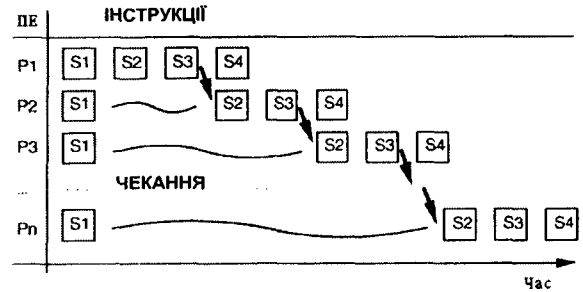


Рис. 16.2. Розпаралелювання без оптимізації

Ефективне розпаралелювання циклу (за правилом д)

Суттєво ефективніше розпаралелювання досягається за допомогою перестановки інструкцій всередині проходу циклу в межах тих циклів, які допускають залежності від даних всередині проходу циклу (напрямок залежності "="). При цьому належить спочатку спробувати виконати **V**-операцію перед **P**-операцією проходу циклу, а потім спробувати досягти максимальної відстані між **V**- та **P**-операціями. Наступне рішення досягає максимальної відстані, причому заздалегідь задані відповідно до залежності даних крайові умови зберігають порядок виконання операторів S_1 перед S_2, а S_3 перед S_4.

```
var sync : array [1..n] of semaphore[0];
```

```
doacross i:=1 to n do
S3:  E[i] := E[i] + 2*B[i];
        V(sync [i]);

S1:  A[i] := B[i] + C[i];
S4:  F[i] := E[i]+1; S1:

        if i>1 then P(sync [i-1]) end;
S2:  D[i] := A[i] + E[i-1];
enddoacross;
```

У цьому рішенні після першої інструкції виконується **V**-операція. Далі кожний процес незалежно від інших процесів може виконати ще дві інструкції, а потім перед останньою інструкцією він має очікувати на результат процеса-попередника. Проте в зв'язку з тим, що **V**-операція виконується в кожному проході циклу перед **P**-операцією, то, як правило, процес-попередник уже виконав необхідну **V**-операцію, якщо процес наступного проходу циклу виконує свою **P**-операцію. Це дає змогу майже повністю уникнути інтервалів очікування (за винятком витрат часу на операції синхронізації **P** і **V**). На рис.16.3 показано значно кращі співвідношення між витратами часу на виконання S_1, S_2, S_3 та S_4, що досягаються в оптимізованому паралельному рішенні.

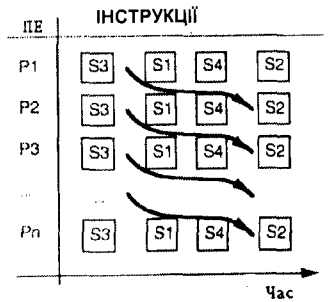


Рис. 16.3. Оптимізоване розпаралелювання

16.4. Розв'язування складних задач залежності даних.

Представлені вище методи векторизації і розпаралелювання не дають ефекту у випадках складних залежностей даних. Розгляньмо три простих методи, які підтримують векторизацію або розпаралелювання циклу.

Циркуляційні залежності

Якщо в деякому циклі має місце циркуляційна залежність між інструкціями типу S_1 перед S_2, S_2 перед S_3, S_3 перед S_1, то розглянутий в п.16.3 метод векторизації не може бути застосований, бо інструкції не допускають відповідної перестановки їх. Для роз'єднання цього ланцюга залежностей застосовують допоміжні змінні. Наприклад:

```
for i:=1 to n do
S1:    A[i] := B[i];
S2:    B[i] := A[i+1];
end;
```

Тут мають місце такі залежності даних:

$S_2 \overline{\delta_{(<)}} S_1$ (через A) $\Rightarrow$ S_2 перед S_1

$S_1 \overline{\delta_{(=)}} S_2$ (через B) $\Rightarrow$ S_1 перед S_2

Таким чином, наявна циркуляційна залежність. Вводимо допоміжну змінну Hilf, яка забезпечує запис значень векторної змінної B в пам'ять проміжного результату, а потім цей результат переписується в рамках виявленої залежності даних:

```
for i := 1 to n do
SH:    Hilf[i] := B[i];
S1:    A[i] := Hilf[i];
```

```
S2:   B[i] := A[i+1];
end;
```

Залежності даних змінюються завдяки такому прийому:

$S_H \ \delta_{(=)} \ S_1$ (через Hilf) $\Rightarrow$ S_H перед S_1

$S_H \ \overline{\delta_{(=)}} \ S_2$ (через B) $\Rightarrow$ S_H перед S_2

$S_2 \ \overline{\delta_{(<)}} \ S_1$ (через A) $\Rightarrow$ S_2 перед S_1

Тепер векторизація можлива:

```
SH:   Hilf(1:N) = B(1:N);
S2:   B(1:N)    = A(2:N+1);
S1:   A(1:N)    = Hilf(1:N);
```

Заміна циклів.

У випадках складних циклів під час векторизації, як правило, видозмінюється внутрішній цикл, а під час розпаралелювання – зовнішній. Внаслідок цього векторна (масивно паралельна) програма SIMD-системи містить у вихідному зовнішньому циклі деяку кількість векторних команд, в той час як MIMD-паралельна програма містить doacross-виклик замість зовнішнього циклу, а внутрішні цикли в кожному процесі залишаються ідентичними.

У випадку, коли в зовнішньому циклі має місце розв'язна залежність даних, а у внутрішньому – нерозв'язна, то безпосередня векторизація неможлива (відповідно навпаки під час розпаралелювання). Ця проблема може бути вирішена методом заміни циклів (зовнішнього на внутрішній і навпаки), якщо немає залежності даних в напрямках (<,>), тобто в напрямку "<" в зовнішньому і в напрямку ">" у внутрішньому циклах.

Заміна циклів має застосовуватися також тоді, коли це дасть ефективніший програмний код. Так, якщо зовнішній цикл виконується для індексу n=1000, а

внутрішній – для m=10, то під час векторизації для SIMD-системи з 1000 ПЕ перетворення зовнішнього циклу з 1000 операцій (повне завантаження всіх ПЕ) суттєво ефективніше, ніж перетворення внутрішнього циклу з 10 операціями (у цьому випадку 990 ПЕ були б неактивними).

Приклад заміни циклів (автор Wolfe в [Hwang, DeGoot 89]):

```
for i := 1 to n do
  for j := 1 to n do
S1:  A[i, j] := A[i, j-1] + A[i, j+1];
  end; (* j *)
end; (* i *)
```

Тут мають місце такі залежності даних:

$S_1 \ \delta_{(=,<)} \ S_1$ (через A[i, j-1]) $\Rightarrow$ S_1 перед S_1

$S_1 \ \overline{\delta_{(=,<)}} \ S_1$ (через A[i, j+1]) $\Rightarrow$ S_1 перед S_1

Наявність їх не дає змоги використовувати внутрішній цикл, але у зв'язку з тим, що немає напрямку залежності (<,>), ці цикли можна поміняти місцями, тобто:

```
for j := 1 to n do
  for i := 1 to n do
S1:  A[i, j] := A[i, j-1]+A[i, j+1];
  end; (* i *)
end; (* j *)
```

В цій зміненій програмі існують такі залежності даних:

$S_1 \ \delta_{(<,=)} \ S_1$ (через A[i, j-1])

$S_1 \ \overline{\delta_{(<,=)}} \ S_1$ (через A[i, j+1])

$\Rightarrow$ ніяких обмежень, бо зовнішній цикл має напрям залежності "<".

Внутрішній цикл може векторизуватися без синхронізації, бо існує залежність даних тільки через індекс зовнішнього циклу (який не змінюється). Векторизована програма має такий вигляд на мові Фортран- 90:

```
do j=1, n
        A(1: n, j) = A(1: n, j-1) + A(1: n, j+1)
end do
```

Якщо існує залежність між межами внутрішнього і зовнішнього циклів, то при заміні циклів межі їхніх індексів мають бути узгоджені. Якщо залежність індексу внутрішнього циклу від індексу зовнішнього циклу нелінійна, то заміна у циклі може бути неможливою [Wolfe 86].

Приклад заміни циклів із взаємозв'язаними межами:

```
for i := 1 to n do
  for j := 1 to i do
S1:  A[i, j] := A[j, i];
  end; (* j *)
end; (* i *)
```

Як видно з програми, кількість повторів внутрішнього циклу j залежить від значення зовнішнього індексу i. Межі циклів при заміні їх мають бути узгоджені (обидві мають дорівнювати n):

```
for j := 1 to n do
  for i := j to n do
S1:      A[i, j] := [j, i];
   end; (* j *)
end; (*i *)
```

Новий зовнішній цикл (індекс j) виконується в повному діапазоні від 1 до n, а новий внутрішній цикл (індекс i) операцією i := j узгоджується на його повторі в межах від j до n. На рис.16.4 пояснюється це узгодження. Векторизована програма має такий вигляд:

```
do j = 1, n
  A(j: n, j) = A(j, j: n);
end do
```

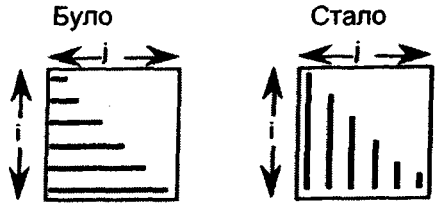


Рис.16.4. Узгодження індексів відповідно до зміни циклів

Комплексні залежності

У практиці програмування трапляються випадки, коли прості методи векторизації (розпаралелювання) не можуть бути застосовані. Розгляньмо цикл із двічі індексованим доступом:

```
   for i := 1 to n do
S1:   A[C[i]] := B[i];
   end;
```

Тут існує залежність типу $S_1 \ \delta^{\circ}_{(<)} \ S_1$ (через A[C[i]]), бо без визначення величин C[i] виникає ситуація, коли в один і той самий елемент пам'яті A[x] будуть багаторазово записуватись дані. Розв'язання цього конфлікту неможливе.

Ще одна проблема, яка, правда, має вирішуватися простіше, це залежність даних в межах однієї інструкції:

```
   for i := 1 to n do
S1:  A[i] := A[i+1];
   end;
```

У цій програмі виникає залежність типу

$S_1 \quad \delta_{(<)} \quad S_1$ (через A)

Під час розпаралелювання потрібна синхронізація між процесами за допомогою семафора. У разі векторизації не виникає залежності даних, бо паралельний код генерується так, що спочатку оцінюється права частина інструкції, а потім виконується операція присвоєння:

S_1: A(1: n) = A(2 : n+1)

Треба також відмітити, що цей вираз на мові Фортран-90 містить неявний обмін даними (в даних умовах дорожча операція). Векторні дані в SIMD-системах розміщені покомпонентно в різних процесорних елементах (ПЕ), тому векторне присвоєння з різними межами індексів– це не що інше, як обмін даними від процесора i+1 до процесора i. Всі ПЕ передають далі числові значення їхньої локальної змінної A лівому сусіду.

17. НЕПРОЦЕДУРНІ ПАРАЛЕЛЬНІ МОВИ ПРОГРАМУВАННЯ

Поряд з процедурними, імперативними мовами програмування останнім часом набувають все більшого значення непроцедурні, декларативні мови програмування. В цій главі охарактеризовані деякі представники паралельних функціональних і логічних мов програмування. Той факт, що непроцедурні мови програмування – це далеко не новий напрям розробок, підтверджується датою появи мови Lisp, яка запропонована в 1962 р. McCarthy і взагалі належить до найстаріших мов програмування.

Непроцедурні мови порівняно з процедурними належать до більш високих рівнів абстракції. У зв'язку з цим поділ цих мов на послідовні та паралельні практично нічого не дає. Такі мови, як FP або APL не потребують

ніяких явних паралельних мовних конструктивів, щоб відображати паралельні явища або ефективно їх розпаралелювати. Тут вся паралельність тримається непомітною для користувача; завдяки декларативному характеру мови задачі керування розпаралелюванням можуть розв'язуватися самою системою транспарентно. Мова банків даних SQL може також розглядатись як така ж паралельна непроцедурна мова програмування.

Автоматичне розпаралелювання і векторизація в функціональних програмах можуть виконуватися значно простіше, ніж в послідовних процедурних програмах (гл.16). Незважаючи на це, існує цілий ряд паралельних функціональних мов з явними паралельними мовними конструктивами. В мову *Lisp перенесено безпосередньо з C* імперативні паралельні конструктиви для паралельності даних. По-іншому складаються обставини в нових паралельних функціональних мовах програмування. Так, в "пара-функціональній" мові Haskell [Hudak, Wadler 90], [Szymanski 91] вводяться коди-анотації для упорядкування частин програм, що виконуються паралельно.

Як представник паралельних логічних мов програмування розглядається мова Concurrent Prolog. Ця мова майже не відрізняється від мов Parlog, Strand і GHC. Такі відомі непроцедурні мови, як мова ID EOM, що керується потоком даних ([Nikhil 88], [Szymanski 91]), а також логічна мова Unity, що не базується на мові Prolog ([Chandy, Misra 88], [Kurfeβ 91]), тут детально не розглядаються.

17.1. *Lisp (зірка-ліспп).

Автор: Thinking Machines Corporation, 1986.

Мову "Star-Lisp" розроблено названою корпорацією як другий паралельний варіант мови Lisp для SIMD-системи CM-2 [Thinking Machines 86]. Ця мова майже не схожа на мову CMLisp, що розроблена Хіллісом як основна

мова програмування для ряду Connection Machine [Hillis 85; Steele, Hillis 86]. Мова CMLisp застосовує велику кількість допоміжних конструктивів, типів та абстрактних образів, що нагадують мову APL і часто є важкими для розуміння. Ще одним, менш успішним варіантом, є мова Paralation Lisp [Sabot 88]. В ній можуть декларуватись і паралельно поелементно виконуватися групи віртуальних процесорних елементів (Paralations); обмін даними реалізується за допомогою різноманітних відображень (mappings). Не будемо тут розглядати ці варіанти Lisp. В мові *Lisp на відміну від названих CMLisp і Paralation Lisp містяться такі ж прості паралельні конструктиви, як і в C*, але вони вбудовані в Lisp-середовище. Базовою мовою *Lisp є мова Common Lisp.

Паралельні мовні конструктиви:

• Паралельні системні функції мають префікс "*". Багатосистемні функції існують в скалярній і векторній версіях.

Приклад:

(*defvar a) – декларування паралельних векторів.

• Пралалельні оператори мають суфікс "!!". Більшість операторів існує в скалярній і векторній версіях.

Приклад:

(+!! a b) – сумма двох векторів
(*!! a b) – добуток двох векторів
(!! 2) – векторна константа з ідентичними компонентами
(+!! a (!!2)) – складання вектора з векторною константою.

Завдяки простому поширенню відомих скалярних системних функцій та операторів на векторну область стає

можливим зрозуміле й елегантне паралельне програмування.

• Можливе визначення і виділення із загальних процесорних структур груп процесорів за допомогою функції def-vp-set ("define virtual processor set") сумісно з функцією greate-geometry, чим підтримуються віртуальні процесори. Відповідно до локальної решітки ("NEWS-Grid") системи CM-2 за допомогою цієї машинно-залежної функції можуть визначатись n-вимірні блоки процесорних елементів. Це така сама функція, як CONFIGURATION в мові Parallaxis (див. п.14.4).

Так, для декларування і фізичної реалізації тривимірної решітки ПЕ, що має ім'я wuerfel і розміри 50x50x50 ПЕ, використовується запис:

(def-vp-set wuerfel'(50 50 50)).

• Власний номер кожного ПЕ може бути одержаний за допомогою таких функцій:

self!! – видає адресний об'єкт, який охоплює всі розмірності декларованої решітки.

(self_address_grid!!(!!p)) – видає адресний об'єкт тільки для p-ої розмірності решітки.

self_address!! – видає "адресу відправника" для кожного ПЕ. Вона в загальному випадку не збігається з адресою ПЕ в решітці (у тому числі в одновимірній решітці).

• Для обміну даними між ПЕ відповідно до структури CM-2 існують два конструктиви:

1. Для локальної передачі даних через швидку решітчасту структуру:

*news – виконує локальний обмін даними вздовж решітки між виразом-джерелом і цільовою змінною величиною;

news!! – повертає назад "локально-переміщені" (або ротовані) дані виразу-джерела ;

news-border!! – дозволяє обмін, як news!!, одначе можуть бути розкриті доступи до позицій за межами решітки із заданим як параметр "прикордонним вектором";

*news-direction – обмін як і *news, але тільки вздовж одного напрямку розміру решітки;

news_direction!! – обмін як і news!!, але тільки вздовж одного напрямку розміру решітки.

Приклад: (news!!source 1 2)

Цей оператор постачає дані вектора source, які на запит переміщені в решітці на одну позицію вліво і на дві позиції вгору.

2. Для глобальної передачі даних через повільний роутер (маршрутизатор):

*pset – виконує глобальний обмін даними між відправником і цільовою змінною величиною;

pref!! – повертає назад "глобально обмінені" дані відправника.

Приклад: (pref!! source address)

Реалізує перестановку даних вектора source відповідно до векторної адреси address.

• Редукція вектора в скаляр за допомогою стандартних функцій:

*and *or *xor *logand *logior *logxor
*sum *min *max *integer-length

Мова йде про відомі арифметичні та булівські оператори ("log"-оператори виконують логічні операції над даними типу integer). Операція *integer-length визначає

щонайменшу довжину двоїчного слова, яка потрібна, щоб можна було представити кожну величину заданого вектора. В мові *Lisp можлива також редукція вектора за допомогою операцій редукції, заданих користувачем.

• Індікація булівського числового значення всіх фізичних процесорних елементів на панелі системи СМ-2 за допомогою функції *light.

Паралельні мовні конструктиви *Lisp дуже подібні до таких же конструктивів мови С*. Особливою перевагою *Lisp є наявність імітатора, який побудований на базі Common-Lisp і робить можливим виконання паралельних програм на послідовних ЕОМ. Недоліком *Lisp є занадто велика, майже неозора кількість паралельних операцій та їхніх додаткових параметрів. В цьому параграфі представлені тільки найважливіші операції.

Скалярний добуток мовою *Lisp:

а) із загальною функцією редукції

```
(*defun s_prod (ab)
        (reduce!! #'+!!(*!!ab)))
```

б) скорочена операція з стандартною функцією визначення суми

```
(*defun s_prod (ab)
        (*sum (*!!ab)))
```

Оператор Лапласа мовою *Lisp:

```
(def-vp-set grid ' (100 100))
(*defvar pixel)
...
(*with vp-set grid
        (*set pixel (-!! (*!! pixel (!! 4))
                         (news!! pixel  0  1 )
                         (news!! pixel  0 -1 )
                         (news!! pixel  1  0 )
                         (news!! pixel -1  0 ))
))
```

Обмін інформацією виконується тут більш швидкими, але машинно-залежними підпрограмами обміну в решітчастій структурі. Команди "news!!" читають дані вектора з відносним зміщенням (тут –1, 0 або +1) і тому операції з абсолютними адресами не потрібні.

17.2. Мова FP

Автор: John Backus, 1978

Мова FP розроблена Бекусом [Backus 78] як чисто функціональна мова програмування і містить цілий ряд елементів, подібних до мови APL [Iverson 62]. Основні складові частини загальної "FP-системи":

а. Деяка область об'єктних даних (наприклад, числа типу integer, real, дані character, string тощо). Кожний об'єкт може бути або атомом (елементарним об'єктом), або списком об'єктів, що замикаються гострокутними дужками і розділяються комами.

б.Деяка множина примітивних функцій (наприклад, арифметичні оператори +,-,*,/ тощо). Застосування функції позначається знаком ":" між функцією та аргументом.

в.Деяка множина операцій побудови програм "program forming operations" PFO (наприклад, поелементне застосування функції відносно списку, послідовне виконання, редукція тощо).

Після застосування конкретних величин відносно цих трьох пунктів "виникає", як показано в [Eisenbach 87], мова FP.

а. Області даних для об'єктів включають:

цілі числа (integer), дійсні числа (real), символи (character) і послідовності символів (string), а також символ ± для "невизначеної" операції (наприклад, помилкової операції ділення на нуль). Атоми T та F інтерпретуються як булівські величини "true" та "false".

Приклади атомів : 10, -5.25, c, T, hallo, ⊥

Приклади списків: <1,2,3>; <<1,2>,<a,b>,<c,1>>; <>
(простий список; складний список; пустий список).

б. Наявні такі примітивні функції.

б.1. Арифметичні операції:+,-,*,/ (визначаються тільки для чисел).

Приклади:

+ : <1,2>=3

* : <2,hallo>= ⊥

б.2. Оператори порівняння: eq (equal, дорівнює), ne (not equal, не дорівнює), gt (greater than, більше ніж), ge (greater or equal, більше або дорівнює), lt (less than, менше ніж), le (less or equal, менше або дорівнює). Ці оператори визначаються тільки відносно чисел.

Приклади :

eq:<1,1> = T
gt:<1,2> = F

б.3. Булівські операції: and, or, not.

Приклади :

and:<T,F> = F

not: F = T

б.4. Перевірка, чи список є пустим: null

Визначення: null:x= $\begin{cases} T, \text{ якщо } x = <> \\ F, \text{ якщо } <x^1,...,x^n> \text{ при } n \geq 1 \\ \perp, \text{ в інших випадках} \end{cases}$

Приклад : null:<2, 3> = F

б.5.Визначення довжини списку: len

Визначення: len:x = $\begin{cases} 0, \text{ якщо } x = <> \\ n, \text{ якщо } <x^1,...,x^n> \text{ при } n \geq 1 \\ \perp, \text{ в інших випадках} \end{cases}$

б.6. Доповнення списку зліва або зправа: al, ar (append left, append right).

Визначення: al:<y, <>>=<y>
al:<y, <z_1,...,z_m>>=<y, z_1,...,z_m>
ar:<<>, y>=<y>
ar:<<z_1,z_2,...,z_n>,y>=<z_1,...,z_m,y>

Приклад : al: <5,<7,4,7>>=<5,7,4,7>

б.7.Вибір елементів із списку: 1,2,3,...,1r,2r,3r,... (вибирається і-й елемент зліва або зправа (r))

Приклади : 1:<10,11,12>=10 (перший зліва)

1r:<10,11,12>=12 (перший зправа)

4: <10,11,12>=$\perp$ (операція невизначена, бо в списку лише 3 елементи)

б.8. Ідентичність: id

Визначення: id:x = x

Приклад: id:<2,3>=<2,3>

б.9. Транспонування матриці: trans

Визначення:

trans: <<>,...,<>>=<>
trans:
<<x_{11},...,x_{1n}>,<x_{21},...,x_{2n}>,...,<x_{m1},...,x_{mn}>>=
<<x_{11},...,x_{m1}>,<x_{12},...,x_{m2}>,...,<x_{1n},...,x_{mn}>>

Приклад:

trans: <<a,b>,<c,d>,<e,f>>=<<a,c,e>,<b,d,f>>

б.10. Розподіл об'єкта між елементами вектора: distl, distr (distribute left, distribute right)

Визначення :

distl: <x,<>>=<>
distl: <x,<z_1,...,z_m>>=<<x,z_1>,...,<x,z_m>>
distr: <<>,x>=<>
distr: <<z_1,...,z_m>,x>=<<z_1,x>,...,<z_m,x>>

Приклад : distl:<1,<1,2,3>>=<<1,1>,<1,2>,<1,3>>>

б.11. Генерування вектора чисел: iota

Приклади : iota: 0 = <>
iota: 5 = <1,2,3,4,5>

в. Наявні такі операції побудови програм (PFO):

в.1.Послідовне виконання, композиція:

($f \circ g$):x=f:(g:x)

Приклад: (not ° and):<F,T>=not:(and:<F,T>)=not:F=T

в.2.Умова, вибір:

$$(p\rightarrow f;\ g):\ =\ \begin{cases} f:x, \text{ якщо } p:x = T \\ g:x, \text{ якщо } p:x = T \\ \perp, \text{ в інших випадках} \end{cases}$$

Приклад : (eq → +,-):<5,3>=-:<5,3>=2

в.3.Асинхронне паралельне виконання, конструкція:

$[f_1,f_2,...,f_m]:x=<f_1:x,f_2:x,...,f_m:x>$

Приклад : [+,*]:<1,2>=<+:<1,2>,*:<1,2>>=<3,2>

в.4.Синхронне паралельне виконання (apply to all)

$\alpha f:<x1,...,xm>=<f:x1,...,f:xm>$

Приклад :
α+:<<1,2>,<3,4>,<5,6>>=<+:<1,2>,+:<3,4>,+:<5,6>>=<3,7,11>

в.5.Редукція (вліво і вправо)

$$/f:x\ =\ \begin{cases} Z, \text{ якщо } x=<Z> \\ f:<z_1, /f:<z_2,...,z_m>>, \text{ якщо } x=<z_1,...,z_m>,\ m\geq 2 \\ \perp, \text{ в інших випадках} \end{cases}$$

$$\backslash f:x\ =\ \begin{cases} Z, \text{ якщо } x=<Z> \\ f:<\backslash f:<z_1,...,z_{m-1}>,z_m>, \text{ якщо } x=<z_1,...,z_m>,\ m\geq 2 \\ \perp, \text{ в інших випадках} \end{cases}$$

Приклад: /or:<F,T,F>=or:<F,/or:<T,F>>
=or:<F,or:<T,/or:<F>>>=or:<F,or:<T,F>>
=or:<F,T>=T

в.6. Оператор констант (напр., **1**, **2**, **3**,...)

$$\mathbf{f}:x\ =\ \begin{cases} \perp, \text{ якщо } x=\perp \\ k, \text{ якщо в інших випадках (} k \text{ – константа, що визначається } \mathbf{f}, \text{ аргумент } x \text{ не застосовується)} \end{cases}$$

Приклад: **5**:<1,2,3>=5

Програмні операції в.3) та в.4) точно відповідають моделям асинхронної (MIMD) та синхронної (SIMD) паралельностей.В зв'язку с цим FP-програми можуть бути легко перекладені відповідним компілятором в паралельні програми для обчислювальних систем заданого типу. При цьому можуть використовуватись різні технічні прийоми: "eager evaluation", тобто негайна оцінка частин програми, коли можливо (data driven); "lazy evaluation", тобто відкладення оцінки частин програми до того моменту,коли вони фактично стають потрібними (demand driven). Замість розглянутих тут операцій редукції вліво та вправо можна побудувати синхронно-паралельну версію, яка проводить редукцію списочного об'єкта (вектора) в атом (скаляр) всього за $\log_2 n$ кроки (n-кількість елементів). В мові FP можливо конструювати прикладні програми за допомогою заново визначених функцій і тих функцій та операцій, які існують в цій мові. Для цього передбачено конструктив визначень def. Розгляньмо як приклад визначення скалярного добутку [Eisenach 87]. Цю операцію ми наводили раніше стосовно інших мов програмування.

def s_prod=(/+) ° (a*) ° trans
s_prod:<<1,2,3>,<4,5,6> >

=(/+) ° (a*) ° trans:<<1,2,3>,<4,5,6>>	Визначення s_prod
=/+:(a*:(trans:<<1,2,3>,<4,5,6>>))	Розкриття послідовності операцій
=/+:(a*:<<1,4>,<2,5>,<3,6>>)	Розкриття операції trans-транспозиція
=/+:<*:<1,4>,*:<2,5>,*:<3,6>>	Запис паралельного синхронного виконання операції множення
=/+:<4,10,18>"	Результат паралельного множення "*"
=+:<4,/+:<10,18>>	Редукція,1-й крок
=+:<4,+:<10</+:<18>>>	Редукція,2-й крок
=+:<4,+:<10,18>>	Редукція,3-й крок для одного елемента
=+:<4,28>	Редукція,4-й крок "+"
=32	Редукція 5-й крок "+" , кінцевий результат.

Як видно з прикладу, функція s_prod поступово розширюється і виконує дії над аргументом, що складається з двох векторів (<1,2,3>,<4,5,6>). Відповідно до визначення "справа наліво" виконуються такі операції:

(1) – Транспозиція обох векторів,тобто творення пар елементів,взятих відповідно з першого і другого векторів (результат : <1,4>,<2,5>,<3,6>).

(2) – Синхронно паралельне множення всіх пар елементів.

(3) – Складання всіх одержаних пар (редукція).

17.3. Паралельний Пролог

Автор: Ehud Shapiro, 1983

Мова Concurrent Prolog розроблена як розширення логічної мови Prolog і призначена для паралельного програмування [Shapiro 87]. Були розроблені також варіанти Parlog, Strand, GHC (Guarded Horn Clouses), Flat Concurrent Prolog та інші [Shapiro 87]. Зупинімося на мові Concurrent Prolog, бо інші варіанти мало відрізняються один від одного.

У зв'зку з тим що логічне програмування на високому, декларативному рівні виконується як процедурне, імперативне програмування, то доцільно паралельно виконувати логічний базис правил неявно. Це означає, що можна обійтися без явних паралельних мовних конструктивів, використовуючи тільки прості (неявні) механізми синхронізації. На рис. 17.1 показано можливі відправні пункти для паралельної обробки, що витікають з моделі послідовної мови Prolog.

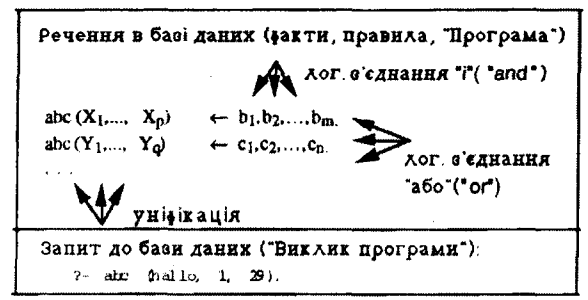


Рис.17.1. Вузлові моменти паралельності в мові Prolog

Відповідно до трьох основних операцій – уніфікація, "і"-об'єднання (and), "або"-об'єднання (or) – з'являються три можливих відправних пункти для паралельної обробки.

1. Уніфікована паралельність:

паралельна пара параметрів речення може суттєво скоротити час, потрібний на уніфікацію. При цьому можуть паралельно оброблятись як параметри (top-level), так і комплексні параметри з баготьма процесорами (рекурсивно).

2. АБО (OR)-паралельність:

якщо для одного речення (логічного виразу) одночасно існують декілька правил або фактів, то всі варіанти можуть проглядатися паралельно. Такий паралельний аналіз конфліктує з семантикою послідовної

мови Prolog, бо в ній застосовується перше парне правило. Таким чином, в послідовній версії мови Prolog порядок проходження правил і фактів має принципове значення. Якщо ведеться пошук тільки одного, а не всіх можливих висновків (рішень) деякого звертання до бази даних (як в семантиці послідовного прологу), то при паралельній обробці АБО-варіантів здебільшого виникає потреба в недоцільному збільшенні витрат на обчислювальну потужність. Це залежить від відповідної програми і семантики паралельного варіанту мови Prolog, яка має визначатися.

3. "І"(AND)-паралельність:

Речення (логічні вирази), які бувають в правилах (або запитах), об'єднуються логічною операцією "І" і можуть оцінюватися паралельно, якщо вони не залежать одне від одного, тобто якщо вони не містять сумісних вільних змінних. Частіше трапляються випадки, коли частини речень взаємозалежні. Тому окремі логічні речення можуть лише частково оброблятися паралельно, а при появі вільних змінних вони повинні синхронізуватись, тобто чекати, поки інше паралельне логічне речення не буде реалізоване і не видасть очікуваного значення змінної.

Із розглянутих трьох відправних пунктів очевидно, що визначення паралельної логічної мови програмування не таке просте, як можна було б очікувати з огляду на рівень абстракції. Ці відправні пункти паралельної логічної обробки поширюються виключно на MIMD-модель, бо при цьому виникають асинхронні частини задач з різними потоками керуючих команд. Показані тут проблеми, що виникають під час паралельної обробки в мові Concurrent Prolog, не дають ніяких конкретних тверджень про ефективність паралельних прикладних програм.

Під час визначення мови Concurrent Prolog існує ряд синтаксичних і семантичних відмін відносно до послідовної мови Prolog. Речення одного правила поділяються на дві групи – Guard і Body, які розділяються Commit-оператором

"|". Застосування АБО-паралельності допускається в межах групи Guard. Перша успішно оброблена група Guard виконує Commit-операцію і зупиняє нею всі інші АБО-паралельні процеси. Група Body деякого правила обробляється І-паралельно в межах того, що допускає синхронізація взаємозалежних логічних речень. Явні мовні конструктиви для уніфікованої паралельності не передбачені, але вони для можливого розпаралелювання не потрібні.

Структура логічного речення:

$$\underset{goal}{A} \leftarrow \underset{guards}{G_1 \ldots G_m} \quad | \quad \underset{commit}{} \quad \underset{body}{B_1 \ldots B_n}. \ (m,n>0)$$

Порядок обробки речення:

- Всі варіанти ($G_1,\ldots,G_m$) обробляються паралельно, поки в першому варіанті всі G_j будуть виконані. Тільки цей варіант іде далі ("commit"); всі інші відкидаються. АБО-паралельність обмежена на Guard-реченнях, при цьому обчислювальна потужність не виконуваних АБО-процесів не використовується.
- Всі речення, зв'язані логічною операцією І, виконуються паралельно. Ті речення, що взаємозалежні, мають синхронізуватися (див. наступний пункт) І-паралельність, обмежена на Body-речення, виконується асинхронно паралельно.
- Синхронізація І-паралельно виконуваних Body-речень забезпечується за допомогою так званих "read-only"-змінних, які позначаються знаком запитання. Наприклад: x?. Процес, який звернувся з метою читання до "read-only"-змінної, що в даний момент ще не визначена (в мові Prolog var(x)), блокується до того моменту,поки ця змінна синхронізації дістане визначення від другого речення-процесу.

Дуже цікаво самому написати інтерпретатор для паралельного розширення мови Prolog з метою

моделювання і тестування в послідовній мові Prolog. Це показує наступне визначення Шапіро простого "трирядкового метаінтерпретатора" (виклик, наприклад, має вигляд: ?_solve(my_task(x).). Імплементація на деякому реальному паралельному комп'ютері має неодмінно виконуватися на більш низькому рівні.

```
1) solve (true)            :-!
2) solve ((Goal,Rest))  :-solve (Goal), solve (Rest)
3) solve (Head)            :-clause(Head,Body), solve (Body)
```

Дамо такі коментарії до цього метаінтерпретатора.

(1) Умова Goal true виконана негайно.

(2) Якщо декілька Goal зв'язані логічною операцією І (коми в мові Prolog),то вони мусять бути оброблені послідовно (часткова рекурсія через змінну Rest).

(3) Для розв'язання умови Goal шукаються методом перебору з поверненням (Backtracking) послідовно в базі даних різні речення (правила або факти), які підходять до цієї "голови". Відносно "тіла" знайденого правила застосовується рекурсивно метаінтерпретатор solve.

Цей метаінтерпретатор може поступово розширюватись і пристосовуватись до нових синтаксису та семантики відповідного варіанта мови Prolog. Як фрагмент із метаінтерпретатора для мови Concurrent Prolog [Shapiro 83] наведено розширені правила уніфікації із змінними типу "read-only" в яких допускається тільки їх читання:

```
1) unify (X,Y):-(var(x); var(Y) ), ! , X=Y.
```

Це правило перевіряє,чи є одна з двох змінних (X, Y) такою, що не потребує негайної обробки і, якщо ця умова виконується, проводить з "X=Y" "нормальну" уніфікацію Prolog-системи після "cut"-оператора, який запобігає зворотному ходу;

```
2) unify (X?,Y):-!,nonvar (X),unify (X,Y)
3) unify (X,Y?):-!,nonvar (Y),unify (X,Y)
```


Ці два правила обробляють симетрично "read-only" - суффікс "?". Якщо відповідна змінна вже набула числового значення (nonvar (x)), то рекурсивно викликається уніфікація без знака запитання; в іншому випадку уніфікація не відбувається ("cut"-оператор). Разом з метапредикаторами, які тут не розглядаються,це веде до тимчасового блокування виклику.

```
4) unify ([X|Xs],[Y|Ys]):-!,unify (X,Y),unify (Xs,Ys).
5) unify ([ ] ,[] ):-!.
```

Ці два правила обробляють уніфікацію списків, причому уніфікація обох заголовків та обох тіл логічних виразів виконується так, як в звичайній мові Prolog.

```
6) unify (X,Y):-X=..[F|Xs],Y=..[F|Ys],unify (Xs,Ys).
```

Це правило відкриває можливість розділити структуровані параметри на функтори і параметри. Функтор повинен бути ідентичним, в той час як відносно параметрів застосовується рекурсивний виклик рутин уніфікації.

Наприкінці розглянемо два приклади на мові Concurrent Prolog. Відмовимося від обчислень оператора Лапласа, які є типовою проблемою для SIMD-систем, бо мова Concurrent Prolog найбільше підходить для вирішення MIMD-проблем.

Складання елементів дерева на мові Concurrent Prolog:

Визначення деревоподібної структури:

```
baum (element,teilbaum_links,teilbaum_rechts)
(*дерево(елемент, ліва_частина_дерева, права_частина_дерева) *)
```

Пусті дерева позначаються словом leer.

На рис.17.2 показано приклад дерева:

```
baum(7,
        baum(3,baum(5,leer,leer),baum(2,leer,leer)),
        baum(9,leer,baum(6,leer,leer)))
```

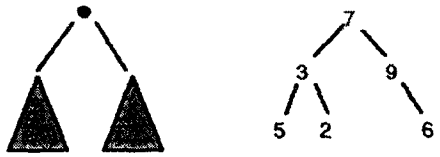


Рис. 17.2. Деревоподібна структура мовою PROLOG та приклад

Програма складання:

```
add(leer,0) .
add(baum(E,L,R),Summe):-add (L?,SL),
                        add (R?,SR),
                        Summe:=E?+SL?+SR?.
```

Виклик програми:

```
?-add(baum(3,baum(5,leer,leer),baum(2,leer,leer)),S).
      ⇒ S=10
```

Оцінювання лівого та правого піддерев, а також складання проміжних результатів і вузлових елементів запускаються як три паралельних процеси і за допомогою "read-only"-анотацій ("?") синхронізуються за спільною змінною. Піддерева L та R мають в рекурсивних викликах знак "?", щоб уникнути безкінечного циклу під час виклику (add(x,y)), що не потребує негайної обробки. При одержанні суми всі складові змінні повинні мати числове значення, тобто цей процес повинен чекати, поки обидва процеси,що складають піддерева, будуть закінчені.

Скалярний добуток, запрограмований мовою Concurrent Prolog.

```
s_prod ([ ],[ ],0):- |.
s_prod ([H1|T1],[H2 | T2 ],X):-|
        X:=H1*H2+Rest?,
        s_prod(T1?,T2?,Rest).
```

Виклик програми:

```
?-s_prod([1 2 3],[4 5 6],SP).
        ⇒ SP=32
```

Commit-оператор запобігає непотрібному в цьому випадку пошуку альтернативного речення (обмеження АБО-паралельності), бо тут немає деякої множини речень з однаковим заголовком. Обидва речення правил (присвоєння і рекурсивний виклик) виконуються паралельно, причому операція присвоєння повинна чекати, поки її змінна синхронізації "Rest?" не набуде числового значення від рекурсивного виклику процедури s_prod. З огляду на рекурсивність активізації і зворотної видачі результатів від цього прикладу застосування мови Concurrent Prolog не слід чекати високої ефективності розпаралелювання.

17.4. Мова SQL

Автор: Chamberlin та Boyce (IBM), 1974.

На перший погляд може здатись дещо незвичайним зарахування мови реляційних банків даних SQL до паралельних мов програмування (SQL-"struktured query language", раніше-SEQUEL) [Chamberlin, Boyce 74]. Одначе в дійсності SQL є не що інше, як мова з неявною паралельністю, яка може використовуватись відповідним компілятором в розподіленій системі банків даних, не потребуючи для цього спеціальних мовних конструктивів.

Крім цього, незаперечна більшість транзакцій SQL-програм має оброблятись в банку даних різними користувачами паралельно або з рознесенням часових інтервалів виконання і взаємозалежністю даних. За допомогою відповідних семафорних блокувань ("locks") компілятор має гарантувати інтеграцію даних всіх транзакцій. SQL-інструкції можуть використовуватися безпосередньо (інтерактивно) в банку даних або інтегруватися в одну із стандартних мов програмування ("host programming language", напр. Kobol або PL/I) як "вбудований SQL ("embedded SQL"), [Date 86].

Реляційний банк даних – це сукупність таблиць, причому кожна з них містить або Entities (діла, речі, об'єкти), або Relationships (взаємовідносини між Entities). Для доступу до окремих рядків таблиць існують ключі доступу, серед яких відрізняють наявні прима-ключі та можливі другорядні ключі. Знайомство зі всіма елементами мови SQL виходить за рамки даної книги, тому лише коротко розглядаємо можливості SQL.

Поряд з генеруванням таблиць та індексів в SQL існують чотири основні операції:

SELECT – вибір одного речення або поля речень із таблиці;

UPDATE –внесення змін у речення або поля речень, що містяться в таблиці;

DELETE – вилучення речення із таблиці (гасіння):

INSERT – внесення речення в таблицю.

Типовий запит на мові SQL в банку даних службовців міг би виглядати так:

```
SELECT  ім'я (прізвище)
FROM    службовець
WHERE   зарплата >10000
```

Пошук у банку даних за цим запитом може вестись неявно паралельно. Як результат беруть із таблиці службовців прізвища (імена) всіх службовців, що мають

заробітну плату понад число 10000. Список,що видається, може за допомогою команд GROUP і ORDER відповідно впорядковуватись або сортуватись.

Зміна змісту таблиць, що пов'язана, наприклад, з можливим підвищенням заробітків, виконується так:

```
UPDATE  службовці
SET     зарплата=зарплата+100
WHERE   зарплата<2000
```

За цією SQL-конструкцією всі групи службовців з нижньою межею заробітку (менше 2000) могли б одержати збільшення зарплати на число 100. Ці операції зміни заробітку, що проводяться відносно можливо великої кількості службовців (елементів даних), виконуються неявно паралельно.

У взаємодії з основними операціями можуть використовуватися такі стандартні функції SQL:

COUNT – кількість елементів;
SUM – сума відносно всіх елементів;
AVR – середній показник всіх елементів;
MAX – максимум всіх елементів;
MIN – мінімум всіх елементів.

Мабуть найсильнішими операціями в SQL є так звані Join Queries, тобто запити, для відповіді на які мають зв'язуватися між собою різні таблиці.

Незважаючи на те, що ці запити можуть бути трудоємкими за величиною використовуваних таблиць, за видами доступів та за наявними (або відсутніми) індексами, саме тут може окупити себе застосування паралельного оброблення. Наступний приклад показує операцію Join над таблицею службовців деякої фірми і таблицею членів спортивного клубу. Вибираються всі особи, які є службовцями фірми і членами клубу (в передбаченні, що

ім'я (NAME) – це однозначно ідентифікаційна ознака, тобто основний ключ):

```
SELECT службовець.прізвище
FROM службовець,член_клубу
WHERE службовець.прізвище=член_клубу.прізвище;
```

SQL – це найпоширеніша сьогодні мова банків даних, а саме у сфері банків даних очікується широке застосування паралельних обчислювальних систем. Для прикладного програміста цієї сфери перехід на застосування SQL в паралельній або розподіленій системі банків даних означає лише збільшення обчислювальної потужності: він не повинен клопотатися про деталі паралельної імплементації.

18. НЕЙРОННІ МЕРЕЖІ

Нейронні мережі – це така справді велика і самостійна область паралельного програмування, що в цій книзі можна навести лише її огляд. Так зване "програмування" за допомогою нейронних мереж з заздалегідь заданими параметрами (функції активізації, способи навчання тощо) показує відносно класів проблем простоту і обмеженість цього методу порівняно до логічного програмування з незмінною стратегією пошуку (наприклад, метод зворотного руху в мові Prolog).

Основи штучних нейронних мереж було розроблено ще в 1943 р. вченими McCulloch і Pitts, які описали природний нейрон як абстрактний логічний пороговий елемент. Із досліджень останнього часу заслуговують на особливу увагу результати дослідницької групи Parallel Distributed Processing (Rumelhart, McClelland 86]. Вступ до цієї тематики є в працях [Hecht-Nielsen 90] та [Arbib 87].

В основу моделі штучної нейронної мережі покладено спрощену модель біологічного оброблення інформації (рис. 18.1).

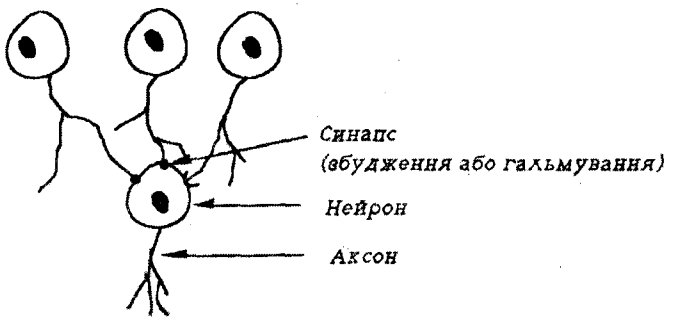


Рис. 18.1. Природна нейронна мережа

Тут "нейрони" абстрагуються як пристрої обробки інформації, що є простим обчислювачем з локальною пам'яттю. Штучна нейронна мережа складається з багатьох дуже простих, але завжди активних елементів (нейронів), які паралельно виконують ідентичні рутинні операції оброблення інформації. Хоч всередені нейронної мережі оброблення йде асинхронно, ця мережа дає змогу найкраще вподобитись SIMD-моделі. Дійсно, якщо всі нейрони виконують однакові операції над своїми локальними даними і асинхронність не є умовою функціонування, то SIMD-системи якнайкраще годяться для моделювання (імітації) нейронних мереж. В подальшому тексті під терміном "нейронні мережі" завжди будемо розуміти штучні нейронні мережі.

18.1. Властивості нейронних мереж

Побудова нейронної мережі характеризується такими їхніми властивостями:

• Велика кількість нейронів із вхідними та вихідними лініями зв'язку.

• Постійна, проблемноорієнтована структура зв'язку (перемикання) нейронів, що здебільшого ієрархічна, але необов'язково регулярна.

• Зв'язки мають вагу і можуть бути або стимулюючими (*excitatory*, з позитивною вагою), або стримуючими (inhibitory, з негативною вагою).

• Кожний нейрон постійно читає (в синхронному або асинхронному циклі) свою вхідну величину, сприймає її відповідно до величин ваги ліній зв'язку і застосовує свою функцію активізації (збудження нейрона). В найпростішому випадку функція активізації – це ідентичність, тотожність.

• Відносно активізації (збудження) нейрона використовується вихідна функція (*output function*). В найпростішому випадку це є порівняння активізації з деяким локальним пороговим числом (*threshold*, рис.18.2, зліва). Стосовно цих бінарних станів нейрона говорять також про його "вогонь".

• Вихідна величина нейрона передається для вводу в інші нейрони через лінії зв'язку, що мають вагу.

• Нейрони часто об'єднують в прошарки (*layers* (рівні)), причому зв'язки існують тільки між сусідніми прошарками (рівнями).

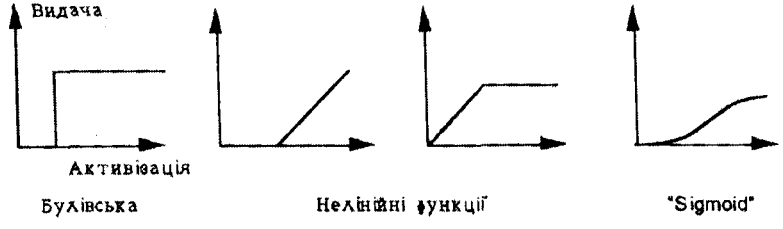


Рис. 18.2. Вихідні функції

Вихідна функція нейрона залежить від його активізації, яка в найпростішому випадку відповідає сумі величин ваг усіх вхідних величин. На рис. 18.2 представлено ряд можливих форм вихідних функцій.

Залежно від вихідних функцій нейронній мережі притаманні різноманітні конвергентні властивості, які використовуються в різних методах навчання мереж.

Переваги та недоліки нейронних мереж:

+ Здатність до навчання.

Нейронні мережі можуть засвоїти характерні властивості тренувальних образів.

+ Узагальнення.

Інформація, що добута із тренувальних образів та результатів тренувань, може бути узагальнена і поширена на невідомі мережі вхідні дані.

+ Толерантність до помилок.

Вихід з ладу або поява помилкових даних одного нейрона не призводить до непридатності всієї мережі, результат її роботи лише поступово погіршується (*graceful degradation*).

+ Нечутливість до шумів.

Якщо суттєві характеристики вхідного образу не дуже змінені шумами або іншими перешкодами, то нейрон продовжує вірно розрізняти вхідну інформацію.

0 Методи навчання не гарантують 100-відсотковий показник успіху.

У реальних умовах застосування нейронних мереж після інтенсивної фази навчання показник розпізнавання не завжди досягає 100%. Залишається певний фактор ненадійності, який в деяких прикладних галузях може стати пропускним бар'єром у застосуванні нейронних мереж.

– Неможливий аналіз вивчених критеріїв класифікації.

Верифікація інформації, засвоєної нейронною мережею, тобто аналіз того, які характеристики використовуються для класифікації, неможлива. Це пов'язано з тим, що інформація, яка запам'ятовується в числових значеннях зв'язків, не може бути пояснена

семантично (перший вклад у цьому напрямку – це "методи інверсії").

– Навчання можливе тільки способом тренажу. З цього виходить, що потрібно готувати "відповідні" тренувальні дані; інші, безпосередні інформаційні завдання (наприклад, процедурні знання) ввести неможливо.

– Методи навчання потребують великих витрат часу. Більшість методів навчання потребують багаторазового повторення навчальних циклів, щоб адаптувати мережу на розпізнавання бажаних вхідних даних.

– Мережі можуть застосовуватись тільки в спеціальних класах проблем.

Поряд з тим, що загальне паралельне програмування і особливо синхронне, доцільно застосовувати для певних класів проблем, у нейронних мережах ці класи знову обмежуються. Більшість нейронних мереж використовується для задач класифікації.

– Немає універсальної нейронної мережі.

Основна ідея нейронних мереж полягала в спробі ліквідувати явне паралельне програмування, але цього не досягнуто. Не існує "універсальної" нейронної мережі для вирішення загальних проблем, тому для кожної проблеми потрібно розробляти свою мережу. Розробка мережі на замовлення (як правило, дуже складна), вибір структури рівнів, ліній зв'язку між нейронами, вибір початкових величин ваги зв'язків, вихідних функцій та відповідних їм порогових констант, а також вибір придатних методів навчання з відповідними тренувальними даними – все це має вирішуватися самим програмістом. Отже, така розробка проводиться як деяке "метапрограмування".

18.2. Мережі типу "feed-forward"

Одним з найпоширеніших класів нейронних мереж є

мережі типу "feed-forward". На відміну від так званих "рекурентних" мереж вони не мають зворотних зв'язків. Fedd-forward-мережі – це розширення "перцептрону" Розенблата [Rosenblatt 62], які містили тільки шари (рівні) вводу-виводу і тому були значно обмеженими у своїх можливостях [Minsky, Papert 69]. Цих обмежень не маємо в багаторівневих мережах.

Головне бажання під час запровадження нейронних мереж полягає в тому, щоб замість трудомісткого (паралельного) програмування адаптувати нейронну мережу за допомогою її навчання на основі ряду введених прикладів, а також, з огляду на методи навчання, за допомогою одержуваних вихідних даних.

Розробляючи нейронну мережу, встановлюють кількість і розміщення нейронів та ліній зв'язку між ними. При цьому з міркувань наочності нейрони розміщують здебільшого в прошарках (рівнях). Знизу розміщується рівень вводу, де існують "рецептори" нейронної мережі. Мережа одержує тут свої вхідні дані; стани активізації цих нейронів пов'язані з відповідними значеннями даних. Далі йдуть один або декілька закритих проміжних рівнів, дія яких не може сприйматися зовнішніми об'єктами. Мережа закінчується рівнем видачі інформації. Те, що видають нейрони цього останнього рівня – це вихідні дані нейронної мережі.

Кількість інформації, або "рівень знань" нейронної мережі, в постійних мережних структурах визначається виключно вагою ліній зв'язку (дуг) між нейронами. Ваги дуг спочатку задаються як випадкові числа і мають уточнюватися повторним заданням тренувальних даних. У циклі навчання нейронній мережі презентуються ітеративно всі тренувальні образи по черзі. Будь-який вихідний результат мережі порівнюється з відомим рішенням відповідного тренувального завдання. Різниця між результатами, що видає мережа, і коректним рішенням подається в мережу як інформація зворотного зв'язку і використовується для коректування вагових характеристик

ліній зв'язку (дуг графа мережі), наприклад для підсилення при коректному результаті і ослаблення при некоректному. У зв'язку з тим що вагові характеристики зв'язків ініціалізуються випадковими величинами, мережа на початку тренажу дуже рідко може "радити" правильну відповідь для деякого образу. Одначе, як правило, у міру збільшення терміну тренінгу досягаються все кращі результати за вихідними даними навчання аж до ідеального стану, коли мережа добре засвоює задане. Після цього вивчені вагові характеристики зв'язків "заморожуються", тобто більше не змінюються, і нейронна мережа може застосовуватися для розв'язання своєї задачі розпізнавання.

Існує цілий ряд різноманітних методів навчання, які залежно від класів задач мають свої переваги і недоліки. Не будемо тут детальніше на них зупинятися. Найбільш відомим методом є дельта-правило із зворотним ходом Backpropagation. В цьому методі різниця між вихідними даними нейронної мережі і результуючими даними тренінгу відстежується на кожному рівні і при цьому коректуються значення ваг відповідних зв'язків на деяку невелику позитивну або негативну величину.

Побудова мережі, вибір даних для тренінгу та його тривалість – все це вирішальні фактори для нейронної мережі, яка може бути визначена тільки емпірично, стосовно даних тренінгу. Мережа має давати результати від дуже добрих до безпомилкових; у такому разі є надія, що за допомогою вивчених даних тренінгу мережа зможе поширити свій набутий досвід на "реальні" і можливо "зашумлені" дані під час наступного застосування. З іншого боку, нейронна мережа не може "вчити напам'ять" дані тренінгу, бо тоді вона б не змогла застосовувати вивчену інформацію на аналогічні вхідні дані.

На рис.18.3 наведено приклад нейронної мережі, яка побудована за допомогою високорозвиненої моделюючої системи SNNS (Stuttgarter Neuronale Netze Simulator). Система SNNS розроблена під керівництвом Целля. Її можна одержати як безкоштовний програмний продукт

[Zell, Mache та інші, 92]. Вона дає можливість розробляти нейронні мережі в інтерактивному режимі, тренувати їх, а також наочно демонструвати результати в графічній формі.

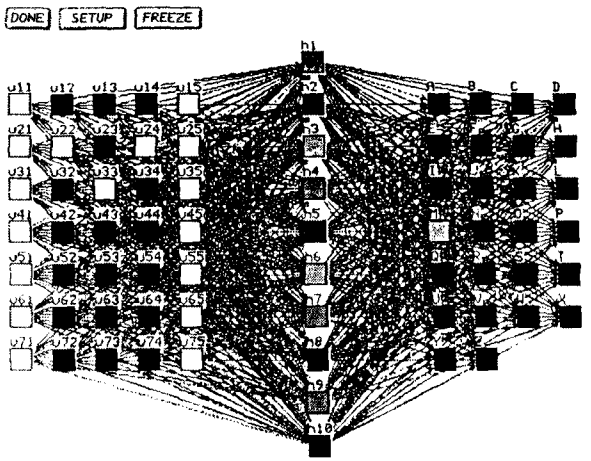


Рис. 18.3. Штуттгартська система моделювання нейронних мереж SNNS

Показана на рис.18.3 мережа повинна служити для розпізнавання великих літер. Вона складається з трьох шарів (рівнів):

• Шар вводу даних

35 нейронів розміщені у вигляді матриці розміром 5x7 відповідно до вхідних даних однієї літери (на рисунку показана активізація для літери "M")

• Схований проміжний шар (рівень)

Він містить 10 нейронів і використовується тільки для зв'язку: як видно з рис.18.3, існують зв'язки між нейронами проміжного шару і нейронами вхідного і вихідного шарів. В цьому прикладі кожний нейрон вводу інформації з'єднаний з кожним нейроном проміжного шару, а той – з кожним нейроном видачі даних. Таким чином, маємо (рис.18.3) всього 35*10+10*26=610 ліній зв'язку.

• Шар видачі даних

Можливі дані для видачі – це великі літери від "A" до "Z". Відповідно до цього передбачено 26 нейронів видачі, кожний для своєї літери (на рис.18.3 активований нейрон для літери "M": нейронна мережа вивчила, таким чином, асоціацію з цією літерою).

18.3. Мережі, що самоорганізуються

Зовсім по-іншому, ніж розглянуті "feed-forward"-мережі, побудовані карти Кохонена, що самоорганізуються ("Kohonen feeture maps", [Kohonen 82]). Вони мають лише один шар нейронів, що з'єднані між собою в мережу, причому між сусідніми нейронами визначено двовимірні або тривимірні взаємовідносини. Вхідні дані і міри ваг зв'язків між нейронами інтерпретуються як просторові вектори. Під час вивчення відомих ознак заданого образу нейрони карти Кохонена змінюють свою вагу відповідно до "відстані" від об'єкта збудження. Нейрон, який розміщений поряд з "центром збудження" (подразнення) відсувається найдалі, в той час як переміщення сусідніх нейронів структури Кохонена, наприклад, складають за функцією Гауса. Це відображає хід розвитку людського організму, в якому сусідні області, такі, наприклад, як чутливі до дотику сенсори шкіри, відображаються на відповідному прошарку нейронів мозку з еквівалентними сусідніми взаємовідносинами. Регіони шкіри з важливими сенсорами, такі наприклад, як пальці, відображаються на більших областях нейронів, ніж регіони з менш важливими сенсорами, як, наприклад, спина.

19. ПРОДУКТИВНІСТЬ ПАРАЛЕЛЬНИХ СИСТЕМ

Паралельні системи будуються насамперед через одну їхню властивість: високу обчислювальну потужність.

У главах 8 та 13 вже було показано при вивченні проблем, що виникають в системах різного класу, що ефективність паралельних систем не завжди узгоджується з їхніми теоретичними величинами потужності. Крім цього, від конкретної постановки проблеми залежить, якого класу і якого типу паралельну систему треба застосовувати і чи взагалі доцільне це застосування. Від кожного застосовування залежить, чи буде досягнуто очікуваний ефект від розпаралелювання.

Оцінюючи продуктивність , безпосередньо на паралельних системах, відзначають позитивний ефект від розпаралелювання (Speedup-прискорення) і вигоди від збільшення масштабності розв'язаних задач (Scaleup), які можна одержати від працюючої паралельної програми [Reuter 92]. Показник Speedup визначає, у скільки разів швидше може бути вирішена одна й та ж задача на N процесорах порівняно з її вирішенням на одному процесорі. Показник Skaleup визначає, у скільки разів більшу за розмірами проблему можна вирішити за той же час N процесорами порівняно з проблемою, що вирішується одним процесором.

19.1.Показник прискорення (Speedup)

Ще в 1967 р. Амдал проаналізував продуктивність паралельних обчислювальних систем, які за цей час стали відомими як "закон Амдала" [Amdahl 1967]. Тут єдиним параметром є поділ програми на послідовну і розпаралелювану частини; масштаб вирішуваної задачі залишається постійним. Розгляньмо дещо спрощену форму цього закону.

Визначення для заданої програми A

P_c–максимальний ступінь розпаралелювання (досягається програмою A з моделлю паралельності C): це максимальна кількість процесорів, які можуть бути

включені в роботу паралельно в будь-який момент часу в період виконання програми A. Тут вирішальне значення має також застосовувана модель паралельності (наприклад, MIMD або SIMD).

T_k – тривалість виконання програм A з максимальним показником розпаралелювання $P_c \geq k$ на системі з k процесорами.

N – кількість процесорів у паралельній обчислювальній системі.

f – послідовна частина програми (у програмі A з моделлю паралельності C): процентна частка операцій, які не можуть бути виконані паралельно на N процесорах, а виконуються тільки послідовно.

У нашому спрощеному розгляді будемо використовувати тільки показники розпаралелювання 1 або N, без проміжних значень.

Тривалість виконання програми на паралельній системі з N процесорами оцінюється формулою:

$$T_N = f * T_1 + (1-f) * \frac{T_1}{N},$$

звідки дістанемо показник прискорення (speedup) в системі з N процесорами

$$S_N = \frac{T_1}{T_N} = \frac{N}{1 + f * (1-N)}.$$

У зв'язку з тим що $0 \leq f \leq 1$, справедливе таке співвідношення:

$$1 \leq S_N \leq N,$$

тобто показник прискорення не може бути більшим ніж кількість процесорів N.

Як міра досягнутого прискорення відносно максимального визначається ефективність системи з N процесорами

$$E_N = S_N / N$$

Область межі ефективності є : $1/N \leq E_N \leq 1$

На практиці використовують значення E_N у відсотках. При E_N = 0,9, наприклад, могла б бути досягнута ефективність, що дорівнює 90% максимально можливої.

Приклади застосування закону Амдала

1. Цільова система має N=1000 процесорів

- Програма має максимальний показник розпаралелювання 1000
- 0,1% програми має виконуватися послідовно (наприклад, операції вводу-виводу), тобто f=0,001

Обчислення показника прискорення (speedup) дають:

$$S_{1000} = \frac{1000}{1 + \frac{1000-1}{1000}} \approx 500$$

Таким чином, незважаючи на суттєво малу послідовну частину програми, у цьому випадку досягається тільки половина максимально можливого прискорення,тобто S_{max} =1000. Ефективність буде E_{1000} =50%.

2. Цільова система має N=1000 процесорів

- Програма має максимальний показник розпаралелювання 1000.
- 1% програми має виконуватися послідовно, тобто f=0,01

Показник прискорення (speedup)

$$S_{1000} = \frac{1000}{1 + \frac{1000-1}{100}} \approx 91$$

Ефективність E_{1000}=9,1%, тобто використовується тільки 9,1% загальної потужності процесорів і це при малій, на перший погляд, послідовній частині програми!

Чим більша послідовна частина програми, тим швидше падає завантаження процесорів, а з ним і показник прискорення порівняно з послідовною обчислювальною системою. Для кожної послідовної частини програми може бути обчислений максимально можливий показник прискорення незалежно від кількості застосовуваних процесорів:

$$\lim_{N\to\infty} S_N(f) = \frac{N}{1+f*(N-1)} = \frac{1}{f}$$

Це означає, що програма із скалярною частиною, яка становить 1%, ніколи не зможе досягти показника прискорення (speedup), що був би більшим 100 – незалежно від того, застосовується 100, 1000 або 1000000 процесорів.

Рис.19.1 показує залежність показника прискорення від кількості процесорів N для різних величин послідовної частини програми f. Головна діагональ належить до лінійного прискорення і може бути побудована при f=0.

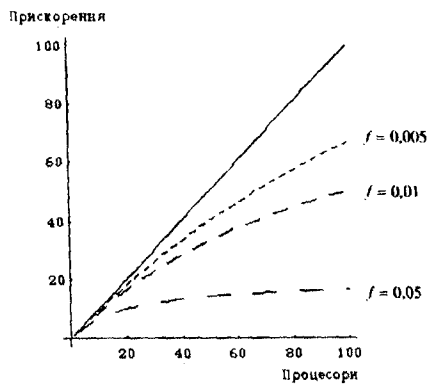


Рис. 19.1. Залежність прискорення від кількості процесорів

Рис 19. 2 показує залежність ефективності від N і f.

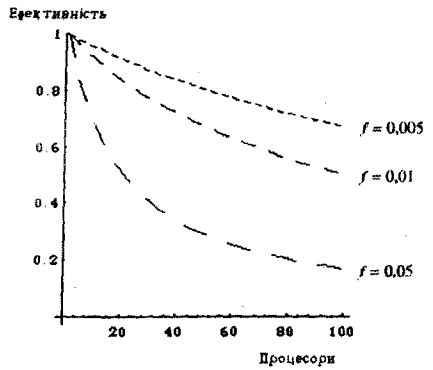


Рис. 19.2. Залежність ефективності від кількості процесорів

19.2. Показник масштабності (Scaleup)

Оскільки для вимірювання ефективності застосовується та сама програма, то показник f в законі Амдала залишається незмінним. Одначе не треба забувати, що кожна паралельна програма, незалежно від її величини, має послідовно обробляти деяку постійну мінімальну кількість f своїх інструкцій. Це означає, що графіки, показані на рис.19.1, 19.2, із зміною розмірів проблеми стають недійсними! Практичні заміри на паралельних системах з дуже великою кількістю процесорів [Gustafson 88] показали: чим більшими є вирішувана проблема і кількість застосовуваних процесорів, тим меншою стає відсоткова послідовна частина обчислень. Це дає підставу зробити висновок, що на паралельній системі з дуже великою кількістю процесорів можна досягти високої ефективності. Напрошується ідея провести дослідження залежностей, що мають місце між різними за величиною, масштабними версіями програм, що реалізують один і той

самий алгоритм (наприклад, з більшою областю даних або з вищою точністю).

Визначення для варіантів програми, що мають різну величину

$T_K(m)$ – тривалість виконання програми з величиною проблеми m і максимальним показником розпаралелювання $P_i \geq k$ на k процесорах.

Показник масштабності (skaleup) деякої проблеми, величина якої n на k процесорах порівняно з меншою проблемою m ($m<n$) на одному процесорі визначається так: якщо $T_1(m) = T_K(n)$, тобто тривалість виконання "малої" програми на одному процесорі дорівнює тривалості виконання "великої" програми на k процесорах, то показник масштабності (skaleup) можна виразити формулою:

$$SC_k = \frac{n}{m}$$

Треба мати на увазі, що тривалість виконання залежить від такого параметра, як "величина проблеми", яка точно не визначена. Її в даному аспекті можна трактувати як кількість даних, які обробляються варіантами програм різної величини за одним і тим же алгоритмом.

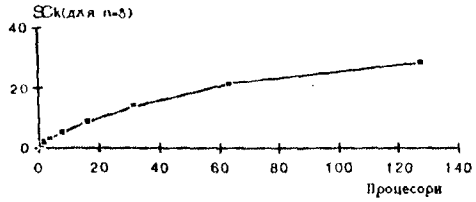


Рис. 19.3. Залежність показника збільшення складності вирішуваних задач SCk від кількості процесорів

Як видно з практики роботи з паралельними програмами, із більшенням кількості процесорів

найчастіше вирішуються більші проблеми і в більшості практичних застосувань не ставиться задача вирішувати ті ж проблеми швидше за рахунок збільшення кількості процесорів. У таких практичних областях показник Skaleup має більше значення, ніж Speedup.

19.3.MIMD проти SIMD

Під час теоретичного аналізу продуктивності паралельних обчислювальних систем, проведеному вище, ми не розрізняли MIMD- та SIMD-системи, але кожна програма містить у собі як мінімум два різних максимальних показника паралельності: один для асинхронної паралельної обробки, P_{MIMD}, а другий – для синхронної, P_{SIMD}. У зв'язку з універсальністю асинхронної паралельної обробки має місце співвідношення $P_{SIMD} \leq P_{MIMD}$.

На практиці має велике значення ще один момент: якщо в MIMD-системі є вільні процесори, що не використовуються в даній задачі, то вони можуть бути застосовані іншими користувачами для своїх задач, що розв'язуються одночасно. А ось в SIMD-системах неактивізовані процесори не використовуються. Тому, ставлячи задачі на SIMD-системах, треба виходити з того, що для проблеми, яка потребує 100 ПЕ, недоцільно використовувати SIMD-систему з 16.384 ПЕ! Спосіб обробки інформації в SIMD-програмах дає суттєво менші показники завантаження ПЕ порівняно з MIMD-програмами. Так, тільки для одного єдиного програмного розгалуження IF-THEN-ELSE, що дає програми двох напрямків з однаковою тривалістю виконання (у загальному випадку – різні тривалості), середня завантаженість падає для цієї інструкції на 50%. Треба підкреслити також, що процесорні елементи SIMD-систем, як правило, менш потужні, ніж процесори, що застосовуються в MIMD-системах, але цей недолік

компенсується за рахунок значно більшої кількості процесорів SIMD-систем порівняно з MIMD-системами.

У MIMD-системах часто можна помітити феномен, що полягає в програмуванні задач в "SIMD-режимі", тобто не використовується можлива незалежність окремих процесорів. Особливо це помічається, коли задача, що роз'язується, розподіляється між великою кількістю процесорів, тобто коли мова йде фактично про масивну паралельність. Більшість проблем, що допускають розподіл їх на 100, 1000 або більше процесорів, є об'єктом обробки інформації, для якого було б достатньо простої SIMD-системи. Ця обставина є одним з найвагоміших аргументів на користь SPMD-моделі паралельності (same program, multiple data), яка виникає внаслідок змішаного використання SIMD і MIMD в рамках однієї системи (див. п.2.1).

У зв'зку з тим що за законом Амдала, як уже було показано, ефективність сильно залежить від кількості процесорів, а з іншого боку, не має можливості уникнути деякої послідовної частини програми (як мінімум, залишаються програми вводу-виводу даних), постає питання: чи доцільно взагалі використовувати паралельну SIMD-систему з кількістю процесорів понад 65000? Як буде показано нижче, це питання цілком резонне.

Якщо помилково застосувати закон Амдала, то можна одержати неправильні результати, до яких насамперед призводить спосіб переліку інструкцій (рис.19.4). У той час як в програмі A (наприклад, в MIMD-

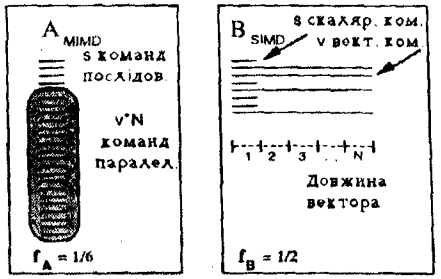


Рис. 19.4. Оцінка кількості паралельних інструкцій

програмі) відповідно до формули Амдала враховується кожна елементарна операція (це дає фактор f_A), в програмі B для SIMD-системи як скалярні, так і векторні операції враховуються як одна інструкція (це дає фактор f_B). Таке визначення є природним для SIMD-системи, бо кожна операція, що так враховується, потребує приблизно однакового часу на виконання. Отже, маємо:

f_A–дає послідовну частину програми відносно *кількості елементарних операцій.*

f_B–дає послідовну частину програми відносно *тривалості виконання операцій.*

У прикладі, показаному на рис.19.4, SIMD-програма може працювати паралельно половину свого часу (f_B=0,5). Водночас кількість послідовно виконуваних елементарних операцій помітно менша, а саме f_A=1/6. Можна виразити залежність між цими двома факторами формулою

$$f_A = \frac{f_B}{N*(1-f_B)+f_B}.$$

Для того щоб одержати точні дані продуктивності, можна застосувати для кожної SIMD-інструкції виражену у відсотках кількість активних в цій інструкції процесорів відносно їхньої загальної кількості: це буде число між нулем (у випадку виконання скалярної операції в керуючій ЕОМ) та одиницею (випадок, коли всі ПЕ активні). Таке зважене за фактором часу складання вказаних процентних величин дасть точнішу міру завантаження паралельних ПЕ прикладною програмою, ніж чисте перерахування всіх інструкцій.

Для визначення показника прискорення (speedup) SIMD-системи можна перерахувати T_N на T_1 відповідно до запитання: "Скільки часу було б потрібно послідовній ЕОМ для виконання цієї паралельної програми?"[Bräunl 91a].

Визначення:

f_B–виражена у відсотках частина скалярних інструкцій SIMD-програми відносно загальної кількості (скалярних і векторних) інструкцій.

Можна вважати ідентичним і таке визначення f_B: *це виражена у відсотках частина часу виконання послідовних операцій відносно загальної тривалості виконання паралельної програми.*

Перерахунок для послідовного виконання паралельної програми здійснюється за формулою

$$T_1 = f_B * T_N + (1 - f_B) * N * T_N$$

Показник прискорення (speedup) за незмінної кількості процесорів N дістанемо у вигляді

$$S_N = \frac{T_1}{T_N} = f_B + (1 - f_B) * N$$

Приклади застосування цього зміненого закону

1. • цільова система має 1000 процесорів.
   • Програма має максимальний показник розпаралелювання 1000.
   • 0,1 % програми (в SIMD-обчисленні) має виконуватися послідовно, тобто f_B=1/1000.

   Розрахунок показника прискорення (speedup) дає

$$S_{1000} = 0.001 + (1 - 0.001) * 1000 \approx 999$$

У цьому випадку досягається прискорення, близьке до теоретично максимально можливого (E_{1000}=100%).

Такий же результат одержуємо і за формулою Амдала з відповідним f_A:

$$f_A = \frac{f_B}{N * (1 - f_B) + f_B} = \frac{0.001}{1000 * (1 - 0.001) + 0.001} = 10^{-6}.$$

Показник прискорення за формулою Амдела:

$$S_{1000} = \frac{1000}{1 + 10 * (1000 - 1)} = 999.$$

2. • Цільова система має 1000 процесорів
   • Програма має максимальний показник розпаралелювання 1000
   • 10% програми (в SIMD-обчисленні) мають виконуватися послідовно, тобто f_B=0,1.

   Розрахунок показника прискорення (speedup) дає S_{1000} = 0,1+0,9*1000 = 900.

Незважаючи на помітно велику послідовну частину SIMD-програми досягається 90% максимального прискорення!

Наведені тут міркування спрямовані на деяку чітко окреслену проблему (з відповідно великим максимальним показником розпаралелювання) і на задану незмінну кількість процесорних елементів: ні розміри задачі, ні кількість процесорів не змінюються, проводиться тільки порівняння з послідовною системою. У випадку, коли змінюється кількість ПЕ в процесі виконання програми, оцінити, як змінюється при цьому показник прискорення (speedup), за наведеною формулою для S_N неможливо, бо вона базується на параметрі T_N! Збільшення кількості ПЕ не привело б до швидшого виконання тієї самої SIMD-програми в області $N > P_{SIMD}$ (кількість ПЕ більша або дорівнює максимальному показнику SIMD-

розпаралелювання), бо ці додаткові ПЕ були б зайвими і залишались би неактивними. Обчисливши всі T_i, де 1<i<N, можна знайти відповідні показники прискорення S_i:

$$T_i = f_B T_N + (1 - f_B) * \frac{N * T_N}{i};$$

$$S_i = \frac{T_1}{T_i}.$$

Цим величинам відповідають графіки на рис. 19.1.

Наведена вище формула для S_N використана в праці [Gustafson 88] для екстраполяції тривалостей виконання масштабованих варіантів проблеми з лінійним масштабним коеффіцієнтом (Skaleup) і визначена як "scaled speedup" (показник прискорення, пов'язаний з масштабами задач), що легко вводить в оману: тут були досліджені часові показники різних за масштабами варіантів програми (див. п.19.2) і обраховані показники прискорення відносно цих варіантів. Було також прийнято, що послідовна частина (за формулою Амдала) деякої "реалістичної" програми при її масштабуванні залишається постійною, а збільшується лише паралельна частина. Проте за визначенням це є точнісенько клас програм з лінійним показником масштабності (scaleup), тобто для нього справедлива формула

$$T_N(A) = T_K * n(K * A)$$

де k – число, яке показує, що в k разів більший варіант програми виконується в k разів більшою кількістю процесорів за той же час.

Крім того, завжди виконується умова:

$T_1(k * A) = k * T_1(A)$ – програма, більша в k разів, потребує при виконанні на одному процесорі в k разів більше часу.

З цієї передумови, тобто з цього обмеженого класу задач, випливає, природно, лінійний приріст показника "scaled speedup":

$$\frac{S_{k*N}(k*A)}{S_N(A)} = \frac{\frac{T_1(k*A)}{T_{k*N}*k*A)}}{\frac{T_1(A)}{T_N(A)}} = \frac{\frac{k*T_1(A)}{T_{k*N}(k*A)}}{\frac{T_1(A)}{T_N(A)}} = \frac{k*T_N(A)}{T_{k*N}(k*A)} = k.$$

19.4.Оцінка величин продуктивності

Усі без винятку дані, що характеризують продуктивність паралельних обчислювальних систем, мають в принципі розглядатися критично. Для цього є цілий ряд причин.

- Такі характеристики продуктивності, як показник прискорення (speedup) і показник масштабності (scaleup), завжди пов'язані з конкретним застосуванням паралельних систем,тобто ці показники справедливі тільки для даного спеціального застосування і дуже умовно можуть бути поширені на інші постановки задач.

- Показник прискорення (speedup) деякої програми стосується одного окремого процесора паралельної системи, але в SIMD-системах процесорні елементи, як правило, мають значно меншу потужність, ніж порівнювані MIMD-процесори, що може дати на порядок, а то й більше, різницю в оцінках швидкості.

- Завантаження паралельних процесорів – це головна мета в MIMD-системах. У SIMD-системах ця мета не є такою ж важливою, бо неактивні процесори не можуть бути використані іншими задачами. Незважаючи на це, показник прискорення обчислюється залежно від характеристики завантаження. Якщо SIMD-програма якимсь чином спробує "включити в роботу" непотрібні ПЕ (зекономити непотрібну операцію явного

виключання ПЕ), то дані завантаження, а з ними і показники прискорення, стануть недійсними. Паралельна програма в цьому випадку буде менш ефективною, ніж за результатами тестування.

• Дані продуктивності в MIPS (million instructions per second) або MFLOPS (million flacting point operations per second) мало про що говорять, бо для того, щоб їх одержати, виконуються найшвидші операції над цілими числами або числами з плаваючою комою. Наприклад, яка проблема потребує *виключно* векторної операції додавання, що є основою для даних продуктивності, які наводить фірма-виробник обчислювальної системи? Між іншим, в подібних даних немає жодного натяку на швидкість обміну даними. Пакети для порівнянного аналізу продуктивності таких послідовних ЕОМ, як Linpack, Whetstone, Dhrystone або SPEC Benchmarks, для паралельних систем часто не можуть бути застосовані. Еталонні тести спеціально для паралельних систем перебувають в стадії розробки.

• Наявні мови програмування високого рівня найчастіше не можуть використовувати всі потенційні можливості апаратури паралельних систем. Для досягнення декларованих фірмами-виробниками даних продуктивності часто виникає потреба в програмуванні на нижчому рівні (асемблер) або у використанні машинно-специфічних бібліотечних підпрограм.

• Порівнюючи системи різних типів, наприклад паралельні системи з системами паралельного обслуговування рабочих місць (Workstation), оцінюють заразом відповідні компіляції з мов програмування і систем вводу-виводу даних. В

принципі тут можуть порівнюватися тільки власне тривалості виконання програм (час центрального процесорного блока, CPU time, або також астрономічний час одного користувача від початку обробки програми до його закінчення, elapsed time).

• Порівнюючи паралельну систему (наприклад, векторну) з деякою послідовною (скалярною), не обов'язково доцільно застосовувати два рази один і той же алгоритм (в паралельній і в послідовній версіях). Наприклад, OETS-алгоритм – це типовий алгоритм сортування для SIMD-систем (див. п.15.4), але для послідовних ЕОМ найефективнішим є алгоритм Quicksort – в усякому разі Quicksort не може бути так просто переведений в ефективну паралельну програму для SIMD-систем. Порівняння "OETS-паралельного" і "OETS-послідовного" алгоритмів дає хороші величини для паралельної системи, але воно не має практичного глузду! Залишається під питанням доцільність порівняння "OETS-паралельний" "Quicksort-послідовний" (або взагалі порівняння паралельного алгоритма з найшвидшим відомим сьогодні послідовним алгоритмом для тієї ж проблеми). Це визначення хоч і є суттєвим для практичного вживання (наприклад, з точки зору потенційного власника паралельної системи), одначе результати порівняння були б потім при певних умовах піддані часовим змінам.

Вправи до розділу IV

1. Знайдіть усі залежності даних в послідовності інструкцій:

S_1: A:=B+C;
S_2: B:=A+C;

S_3: D:=B*C-2;
S_4: A:=B/C

2. Виконайте такі завдання для фрагмента програми, що наведена нижче:

а) знайдіть усі залежності даних з напрямками;

б) виведіть із залежностей даних відповідні правила;

в) векторизуйте цикли в записах Фортран-90 для деякої векторної ЕОМ.

```
    for i:=2 to n-1 do
S1:     A[i]:=B[i]+C[i];
S2:     B[i-1]:=2*D[i]+1;
S3:     C[i]:=A[i]+B[i];
S4:     E[i]:=A[i+1]/7;
    end;
```

3. Виконайте такі завдання для фрагмента програми, що наведений нижче:

а)знайдіть усі залежності даних з напрямками;

б)виділіть ті залежності, які мають бути синхронізованими;

в) розпаралельте цикли для деякої MIMD-системи (у псевдозаписах "doacross"). Спробуйте при цьому досягти максимальної паралельності!

```
    for i:=5 to n do
S1:     A[i-1]:= 2*B[i]+3;
S2:     B[i]  := 2*D[i]+1;
S3:     E[i]  := E[i]+5;
S4:     C[i]  := A[i]+D[i];
    end;
```

4. Виконайте такі завдання для фрагмента програми, що наведений нижче:

а) знайдіть усі залежності даних з напрямками ;

б) виведіть із кожної залежності відповідне правило;

в) векторизуйте цикли на мові Фортран-90 (якщо можливо).

```
    for i:=5 to n do
S1:     A[i] := B[i]+D[i+1];
S2:     B[i] := D[i-1]+1;
S3:     C[i] := A[i-1]+B[i+1];
S4:     D[i] := 15;
    end;
```

5. Виконайте такі завдання для фрагмента програми, що наведений нижче:

а) знайдіть всі залежності даних з напрямками;

б) виведіть з кожної залежності відповідне правило;

в) векторизуйте цикли на мові Фортран-90 (якщо можливо).

```
    for i:=5 to n do
S1:     A[i] := B[i]+D[i+1];
S2:     B[i] := D[i-1]+1;
S3:     C[i] := A[i-1]+A[i+1];
S4:     D[i] := 15;
    end;
```

6. Виконайте такі завдання для фрагмента програми, що наведений нижче:

а)знайдіть усі залежності даних;

б)розпаралельте зовнішній цикл (у псевдозаписах "doacross") для MIMD-системи. Спробуйте при цьому досягти максимальної паралельності! (Внутрішній цикл залишається незмінним);

в)векторизуйте внутрішній цикл (на мові Фортран-90) для SIMD-системи. (Зовнішній цикл залишається незмінним).

7. Побудуйте синхронно паралельний оператор редукції для непроцедурної мови програмування FP. Розділіть для цього список аргументів на дві частини, для

яких рекурсивно паралельно викликається операція редукції. Редукція n елементів має виконуватися впрододовж часу $\log_2 n$ (за достатньої кількості процесорів).

8. Паралельна програма має виконуватися на MIMD-системі з 100 процесорами, 3% всіх команд при проході програми мають виконуватися послідовно (наприклад, внаслідок синхронізації, вводу-виводу тощо), а решта може виконуватися паралельно на всіх процесорах. Яке значення має показник прискорення (speedup) цієї програми на обраній обчислювальній системі?

9. Деяка паралельна програма, що має 10-відсоткову послідовну частину, має виконуватися на MIMD-системі. Чи існує деякий максимально можливий показник прискорення (speedup), незалежний від кількості процесорів системи?

10. Паралельна програма має виконуватися на MIMD-системі з 100 процесорами, одначе:

2% всіх команд при проході програми мають виконуватись послідовно;

20% всіх команд можуть виконуватись тільки на 50 процесорах.

Яке значення має показник прискорення (speedup) цієї програми для даної паралельної системи?

11. Паралельна програма має виконуватися в SIMD-системі, що має 10000 процесорних елементів, одначе вона містить при виконанні 20% скалярних команд (наприклад, через передачу даних до центральної ЕОМ). Решта – це векторні команди, які виконуються на всіх ПЕ. Яке значення має показник прискорення (speedup) цієї програми на даній паралельній системі?

12. Паралельна програма виконується на SIMD-системі, що має 10000 ПЕ. Заміри показують, що в середньому всі ПЕ були активними впродовж 30% загальної тривалості виконання програми, а решту часу були неактивними (наприклад, через інструкції типу IF-THEN-ELSE).

Яке значення має показник прискорення цієї програми на даній паралельній системі?

13. Паралельна програма має виконуватися на SIMD-системі, що має 100000 ПЕ, одначе:

20% всіх виконуваних інструкцій є скалярними командами;

10% всіх інструкцій можуть виконуватися векторно тільки на 100 процесорах;

40% всіх інструкцій можуть виконуватись векторно тільки на 50000 процесорів;

решта інструкцій може виконуватися векторно на всіх ПЕ.

Яке значення має показник прискорення цієї програми на даній паралельній системі?

Список використаної літератури

[Ahuja, Carriero, Gelernter 86] S.Ahuja, N.Carriero, D.Gelernter, *Linda and Friends*, IEEE Computer,vol.19,no 8, Aug.1986,pp.26-34 (9)

[Akl 89] S.Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, International Editions, Englewood Cliffs NJ, 1989

[Almasi, Cottlieb 89] G.Almasi, A.Cottlieb, *Highly Parallel Computing*, Redwood City CA, 1989

[Amdahl 67] G.Amdahl, *Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*, AFIPS Conference Proceedings, vol. 30, Atlantic City NJ, Apr. 1967, pp. 483-485(3)

[Arbib 87] M.Abrib, Brains, *Machines, and Mathematics, Second Edition*, Springer-Verlad, Berlin Heidelberg New York,1987

[Babb 89] R.Babb (Ed.), *Programming Parallel Processors*, Addison-Wesley, Reading MA,1989

[Backus 78] J.Backus, *Can programming be liberated from the won Neumann style? A functional style and its algebra of programs*, Communications of the ACM, vol.21, no.8, Aug.1978, pp.613-641 (29)

[Barth, Bräunl, Engelhardt, Sembach 92] I.Barth, T.Bräunl, S.Engelhard, F.Sembach, *Parallaxis Version 2 User Manual*, Second Edition, Computer Science Report, no.2/92, Universität Stuttgart, Feb.1992, pp. (110)

[Baumgarten 90] B.Baumgarten, *Petri-Netze Grundlagen und Anwendungen*, BI Wissenschaftsverlag, Mannheim Wien Zürich,1990

[Breizer 90] B. Breizer, *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, New York NY, 1990

[Ben-Ari 82] M.Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall International, Englewood Cliffs NJ, 1982

[Black 87] U.Black, *Data Communications and Distributed Networks*, Second Edition, Prentice-Hall, Englewood Cliffs NJ, 1987

[Blasgen, Gray, Mitoma, Price 79] M.Blasgen, J.Gray, M.Mitoma, T.Price, *The Convoy Phenomenon*, ACM Operating Systems Review, vol.13, no.2, April 1979 pp. 20-25 (6)

[Bräunl, Hinkel, von Puttkamer 86] T.Bräunl, R.Hinkel, E.von Puttkamer, *Konzepte der Programmiersprache Modula-P*, Interner Bericht Nr.158/86,Universität Kaiserslautern,1986

[Bräunl 89] T.Bräunl, *Structured SIMD Programming in Parallaxis*, Structured Programming, vol.10, no.3, Juli, 1989, pp.121-132 (12)

[Bräunl 90] T.Bräunl, *Massiv parallele Programmierung mit dem Parallaxis-Modell*, Springer-Verlag, Berlin Heidelberg New York, Informatik-Fachberichte Nr.246, 1990

[Bräunl 91a] T.Bräunl, *Braunl's Law*, IEEE Computer, The Open Channel, vol.24, no.8, Aug.1991, pp.120 (1)

[Bräunl 91b] T.Bräunl, *Designing Massively Parallel Algorithms with Parallaxis*, Proceedings of the 15th Annual International Computer Softwar &Applications Conference, compas91, Sep.1991, pp.612-617 (6)

[Bräunl, Norz 92] T.Bräunl, R.Norz, *Modula-P User Manual*, Computer Science Report, no.5/92, Universität Stuttgart, August 1992

[Brinch Hansen 75] P.Brinch Hansen, *The Programming Language Concurrent Pascal*, IEEE Transcations on Software Engineering, vol.1, no.2, Juni 1975, pp.199-207 (9)

[Brinch Hansen 77] P.Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs NJ, 1977, (auf Deutsch: *Konstruction von Mehrprozeßprogrammen*, Oldenbourg Verlag, 1981)

[Carriero, Gelenter 89] N.Carriero, D.Gelenter, *How to Write Parallel Programs: A Guide to the Perplexed*, ACM Computing Surveys, vol.21, no.3, Sep, 1989, pp.323-357 (35)

[Chamberlin, Boyce 74] D.Chamberlin, r.Boyce, *SEQUEL: A Structured English Query Language*, Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control, Mai 1974

[Chandy, Misra 88] K.M.Chandy, J.Misra, Parallel Pragram Design, Addison-Wesley, Reading MA, 1988

[Chen,Doolen,Matthaeus 91] S.Chen, G.Doolen, W.Matthaeus, *Lattice Gas Automata for Simple and Complex*

Fluids, Journal of Statistical Physics, vol.64, no.5/6, 1991, pp.1133-1162 (30)

[Clos 53] C.Clos,*A Study of Nonblocking Switching Networks*, Bell System Technical Journal, vol.32, no.2, 1953, pp.406-424 (19)

[Coffman, Elphick, Soshani 71] E.Coffman, M.Elphick, A.Soshani, *System Deadlocks*, ACM Computing Surveys, vol.3, 1971, pp.67-68 (12)

[Conway 63] M.Conway, *A Multipracessor System Design*, Proceedings of the AFIPS Fall Joint Conference, 1963, pp.139-146 (8)

[Coulouris, Dollimore 88] G.Coulouris, J.Dollimore, *Distributed Systems Concepts and Design*, International Computer Science Series, Addision-Wesley, Reading MA 1988

[Courtois, Heymans, Parnas 71] P.Courtois, F.Heymans, D.Parnas, *Concurrent Control with 'Readers' and 'Writers'*, Communications of the ACM, vol.14, no.10, Okt.1971, pp.667-668 (2)

[Date 86] C.Date, *An Introduction to Database Systems* (2vols.) Addison-Wesley, Reading MA, 1986

[Dennis, Van Horn 66] J.Dennis, E.Van Horn, Programming Semantics for Multiprogrammed Computations, Communications of the ACM, vol.9, no.3, Marz 1966, pp.143-155 (13)

[Dijkstra 65] E.Dijkstra, Cooperating Sequential Processes, Technical Report EWD-123, Technisch Universität Eindhoven, 1965, (auch enthalten in [Genuys 68])

[Doolen 90] G.Doolen (Ed.), *Lattice Gas Methods for Partial Differential Equations*, Addison-Wesley, Reading MA, 1990

[Eisenbach 87] S.Eisenbach (Ed.), *Functional Programming languages, tools and architectures*, Ellis Horwood, Chichester England, 1987

[Eisenberg, McGuire 72] M.Eisenberg, M.McGuire, *Furher Comments on Dijkstra's Concurrent Programming Control Problem*, Communications of the ACM, vol.15, no.11, Nov.1972, pp.999 (1)

[Fisher 84] J.Fisher, *The VLIW Machine:A Multiprocessor for Compiling Scientific Code*, IEEE Computer, vol.17, no.7, Juli 1984, pp.37-47 (11)

[Flynn 66] M.Flynn, *Very High Speed Computing Systems*, Proceedings of the IEEE, vol.54, 1966, pp.1901-1990 (9)

[Fox, Hiranandani, Kennedy, Koelbel, Kremer, Tseng, Wu 91] G.Kennedy, S.Haranandani, K.Kennedy, C.Koelbel, U.Kremer, C.Tseng, M.Wu, *Fortran D Language Specification*, Technical Report, Rice University, HoustonTX, April 1991, pp.(37)

[Gehani, McGettrick 88] N.Gehani, A.McGettrick (Eds.), *Concurrent Programming*, Addison-Wesley, International Computer Science Series, Reading MA, 1988

[Genuys 68] F.Genuys (Ed.), *Programming Languages*, Academic Press, London, 1968

[Gibbons, Rytter 88] A.Gibbons, W.Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge England, 1988

[Gonauser, Mrva 89] M.Gonauser, M.Mrva (Eds.), *Multiprozessor-Systeme*, Springer-Verlag, Berlin Heidelberg New York, 1989

[Goscinski 91] A. Goscinski, *Distributed Operation Systems The Logical Design*, Addision-Wesley, Reading MA, 1991

[Gray, Reuter 92] J.Gray, A.Rauter, *Transaction Processing: Concepts and Tehniques*, Morgan Kaufmann, Los Altos CA, 1992

[Gustafson 88] J.Gustafson, *Reevaluating Amdahl's Law*, Communications of the ACM, Technical Note, vol.31, no.5, Mai 1988, pp.532-533 (2)

[Habermann 76] A.N.Habermann, *Introduction to Operating System Design*, Science Research Associates Inc./ IBM, SRA Computer Science Series, Chicago, 1976

[Hecht-Nielsen 90] R.Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, Reading MA, 1990

[Hillis 85] W.D.Hillis, *The Connection Machine*, Ph.D.Thesis, MIT-Press, Cambridge MA,1985

[Hoare 74] C.A.R.Hoare, Monitors: An Operating System Structuring Concept, Communications of the ACM, vol.17, no.10, Okt. 1974, pp.549-557 (9)

[Hoare 78] C.A.R.Hoare, Communicating Sequential Processes, Communications of the ACM, vol.21, no.8, Aug. 1978, pp.666-667

[Hoare 85] C.A.R.Hoare, *Communicating Sequential Processes*, Printice Hall, International Series in Computer Science, Englewood Cliffs NJ, 1985

[Hockney, Jesshope 88] R.Hockney, C.Jesshope, *Parallel Computer 2*, Second Edition, Adam Hilger IOP Publishing Ltd., Bristol, 1988

[Hopcroft, Ullman 69] J.Hopcroft, J.Ullman, *Formal Languages and their Relation to Automata*, Addison-Wesley, Reading MA, 1969

[Hudak, Wadler 90] P.Hudak, P. Wadler (Eds.), *Report on the Programming Language Haskell, a Non-strict Purely Functional Language (Version 1.0)*, Technical Report no. YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990

[Hwang, Briggs 84] K.Hwang, F.Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984

[Hwang, DeGroot 89] K.Hwang, D.DeGroot (Eds.), *Parallel Processing for Supercomputers & Artificial Intelligence*, McGraw-Hill, New York, 1989

[Inmos 84] Inmos Limited, *occam Programming Manual*, Prentice Hall International, Englewood Cliffs NJ, 1984

[Iverson 62] K.E.Iverson, *A Programming Language*, Wiley, New York, 1962

[JáJá 92] J.JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley,Reading MA,1992

[Julesz 60] B.Julesz, *Binocular Depth Perception of Computer Generated Patterns*, Bell Systems Technical Journal, no.38, 1960, pp.1001-1020 (20)

[Julesz 78] B.Julesz, *Cooperative Phenomena in Binocular Depth Perception*, Sensory Physiology, no.8, 1978, pp.215-252 (38)

[Kernighan, Pike 84] B.Kernighan, R.Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs NJ, 1984

[Kober 88] R.Kober, *Parallelrechner-Architekturen*, Springer-Verlag, Berlin Heidelberg New York, 1988

[Kohonen 82] T.Kohonen, *Self-Organized Formation of Topologically Correct Feature Maps*, Biological Cybernetics, no.43, 1982, pp.59-69 (11)

[Krishnamurthy 89] E.V.Krishnamurthy, *Parallel Processing and Practice*, Addison-Wesley, Reading MA,1989

[Kuck, Kuhn, Leasure, Wolfe 80] D.Kuck, R.Kuhn, B.Leasure, M.Wolfe, *The Structure of an Advanced Vektorizer for Pipelined Processors*, Proceedings of the 4th International Computer Softwar and Applications Conference, compsac80, Chicago IL, Okt.1980, pp.709-715 (7)

[Kung, Leiserson 79] H.T.Kung, C.E.Leiserson, *Systolic Arrays (for VLSI)*, Sparse Matrix Proceedings 78, Academic Press, Orlando FL, 1979, pp.256-

282 (27), (ebenfalls enthalten in [Mead, Conway 80])

[Kung, Lo, Jean, Hwang 87] S.Kung, S.Lo, S.Jean, J.Hwang, *Wavefront Array Processors – Concept to Implementation*, IEEE Computer, vol.20, no.7, Juli 1987, pp.18-33 (16)

[Kurfeß 91] F.Kurfeß, *Parallelism in Logic*, Vieweg Verlagsgesellschaft, Artificial Intelligence Series, Braunschweig, 1991

[Leiserson 85] C.Leiserson, *Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing*, IEEE Transactions on Computers, vol.C-34, no.10, Okt.1985, pp.892-901 (10)

[Lewis 91] T.Lewis, *Data parallel computing: An alternative for the 1990s*, IEEE Computer, Viewpoints, vol.24, no.9, Sep.1991, pp.110-111 (2)

[MasPar 91] MasPar Computer Corporation, *MasPar Programming Language (ANSI C compatible MPL) User Guide*, Software Version 2.2 , MasPar System Documentation, DPN 9302-0101, Dec.1991

[McCulloch, Pitts 43] W.McCulloch, W.Pitts, *A logical Calculus of the Ideas immanent in Nervous Activity*, Bulletin of Mathematical Biopysiology, no.5, 1943, pp.115-133 (19)

[Mead, Conway 80] C.Mead, L.Conway (Eds.), *Introduction to VLSI Systems*, Addison-Wesley, Reading MA, 1980

[Metcalf, Reid 90] M.Metcalf, J.Reid, *Fortran 90 Explained*, Oxford New York Tokyo, 1990

[Minsky, Papert 69] M.Minsky, S.Papert, *Perceptrons*, MIT Press, Cambridge MA,1969

[Mujtaba, Goldman 81] S.Mujtaba, R.Goldman, *AL Users'Manual*, Third Edition, Stanford Artificial Intelligence Laboratory Report, Stanford University, Dez. 1981

[Nehmer 85] J.Nehmer, *Softwaretechnik für verteilte Systeme*, Springer-Verlag, Berlin Heidelberg New York, 1985

[Parkinson, Litt 90] D.Parkinson, J.Litt (Eds.), *Massively Parallel Computing with the DAP*, Pitman Publishing, London, and the MIT Press, Cambridge MA, 1990

[Peitgen, Saupe 88] H.O.Peitgen, D.Saupe (Eds.), *The Science of Fractal Images*, Springer-Verlag, Berlin Heidelberg New York, 1988

[Perrot 87] R.Perrot, *Parallel Programming*, Addison-Wesley, Reading MA, 1987

[Peterson J. 81] J.Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs NJ, 1981

[Peterson G. 81] G.Peterson, *Myths About the Mutual Exclusion Problem*, Information Processing Letters, vol.12, no.3, Juni 1981, pp.115-116 (2)

[Peterson, Silberschatz 85] J.Peterson, A.Silberschatz, *Operating System Concepts*, Second Edition, Addison-Wesley, Reading MA, 1985

[Petri 62] C.A.Petri, *Kommunikation mit Automaten*, Dissertation, Universität Bonn, 1962

[Pountain, May 87] D.Pountain, D.May, *A Tutorial Introduction to occam Programming*, Blockwell Scientific Publications Ltd.,INMOS,1987

[Quinn 87] M.Quinn, *Desugning Efficient Algorithms for Parallel Computing*, McGraw-Hill, New York, 1987, (auf Deutsch: *Algorithmenbau und Parallelcimputer*, McGraw-Hill, New York, 1988)

[Reuter 92] A.Reuter, *Grenzen der Parallelitat*, Informationstechnick it, vol.34, no.1, 1992, pp.62-74 (13)

[Rose, Steele 87] J.Rose, G.SteeleC*: *An Extended C Language for Data Parallel Programming*, Thinking Machines Corporation, Technical Report, PL87-5, 1987; auch: Second International Conference on Supercomputing, Mai 1987, pp.(36)

[Rosenblatt 62] F.Rosenblatt, *Principles of Neurodynamics*, Spartan, New York NY, 1962

[Rumelhart, McClelland 86] D.Rumelhart, J.McClelland(Eds.), *Parallel Distributed Programming* (2 vols.), MIT-Press, Cambridge MA, 1986

[Sabot 88] G.Sabot, *The Paralation Model*, MIT Press, Cambridge MA, 1988

[Sequent 87] Sequent Coputer Systems Inc., *Sequent Guide to Parallel Programming*, Sequent Computer Systems, Report 1003-44459, 1987

[Shapiro 83] E.Shapiro, *A Subset of Concurrent Prolog and ItsInterpreter*, Institute for New Generation Computer Technology, Tokyo, ICOT Technical Report no.TR-003, 1983, (auch enthalten in [Shapiro 87])

[Shapiro 87] E.Shapiro (Ed.), *Concurrent Prolog* (2 vols.), MIT-Press, Cambridge MA, 1987

[Siegel 79] H.J.Siegel, *Interconnection Networks for SIMD Machines*, IEEE Computer, vol.12, no.6, Juni 1979, pp. 57-65 (9)

[Sommerville, Morrison 87] I.Sommerville, R.Morrison, *Software Development with Ada*, Addison-Wesley, Reading MA, 1987

[Steele, Hillis 86] G.Steele, W.D.Hillis, *Connection Machine Lisp: Fine Grained Parallel Symbolic Processing*, Thinking Machines Co., Technical Report Series PL86-2, 1986; auch: ACM Symposium on Lisp and Functional Programming, Aug.1986, pp.(42)

[Szymanski 91] B. Szymanski (Ed.), *Parallel Functional Languages and Compilers*, ACM Press New York NY, Addison-Wesley, Reading MA, 1991

[Tanenbaum 89] A.Tanenbaum, *Computer Networks*, Second Edition, Prentice-Hall, Englewood Cliffs NJ, 1989

[Trew,Wilson 91] A.Trew, G.Wilson (Eds.), *Past, Present,*

Parallel, Springer-Verlag, Berlin Heidelberg New York, 1991

[Thinking Machines 86] Thinking Machines Corporation, *Introduction to Data Level Parallelism*, Thinking Machines Co., Technical Report Series TR86-14, 1986, pp.(60)

[Thinking Machines 90] Thinking Machines Corporation, *C* Programming Guide Version 6.0*, Thinking Machines System Documentation, Nov.1990

[United States Department of Defense 81] United Sates Department of Defense, *The Programming Language Ada Reference Manual*, Lecture Notes in Computer Science, no. 106, Springer-Verlag, Berlin Heidelberg New York, 1981

[Weicker 90] R.Weicker, *An Overview of Common Benchmarks*, IEEE Computer, vol.23, no.12, Dez.1990, pp.65-75 (11)

[Wirth 83] N.Wirth, *Programming in Modula-2*, Springer-Verlag, Berlin Heidelberg New York, 1983

[Wolfe 86] M.Wolfe, *Advanced Loop Interchanging*, Processing of the 1986 International Conference on Parallel Processing, St.Charles, IEEE CS Press, Washington, D.C., Aug.1986, pp.536-543 (8)

[Zell, Mache, Hübner, Schmalzl, Sommer, Korb 92] A.Zell, N.Mache, R.Hübner, M.Schmalzl, T.Korb, *SNNS Stuttgart Neuronal Network Simulator User Manual Version 3.0*, Computer Science Report, No.3/92, Universitat Stuttgart, 1992, pp.(157)

ЗМІСТ