КОНЦЕПЦІЇ ПАРАЛЕЛЬНОЇ ОБРОБКИ

Співпрограми

Використовуючи співпрограми (coroutine, рис. Б.1), говорять про концепцію обмеженої паралельної обробки інформації. В основу цієї обробки покладено однопроцесорну модель, де є тільки один потік інструкцій з послідовно виконуваним керуючим потоком, що передбачає організоване заняття та звільнення операційного ресурсу "процесор" співпрограмою.

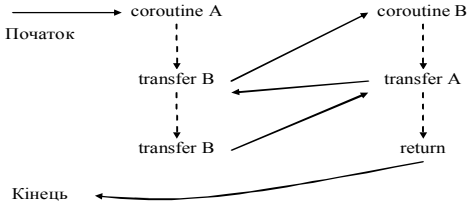

Рис.Б.1. Обмежена паралельна обробка (coroutine)

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 виклику до наступного. Обробка починається за викликом одної 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 у явній формі. Він повинен також піклуватися про те, щоб кожна співпрограма була доступною і щоб керуючий потік правильно визначав точки співпрограм. з яких продовжується обчислення.

У зв'язку з тим, що ця концепція будується на одному єдиному процесорі, вона не потребує відповідних управлінських витрат на мультизадачність. Однак тут не відбувається справжньої паралельної обробки!

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

Конструкції *fork* і j*oin (*які в операційній системі Unix відомі як fork і wait) належать до найперших паралельних мовних конструкцій (рис.Б.2).

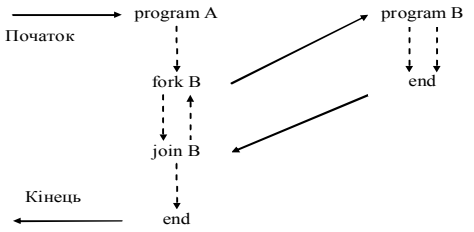

Рис.Б.2. Паралельні конструкції Fork та Join

В операційній системі Unix є можливість запускати паралельні процеси за допомогою операції fork, а закінчення їх чекати операцією wait. На відміну від співпрограм 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: декларування паралельних процесів і 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.

У системі Unix виклик операції fork функціонує не так наочно, як показано на рис. Б.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 –процесори - 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 таким чином:

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

Виклик операції fork видає для батьківського процесу номер процесу-дитини як зворотній зв'язок (тут fork() не є нулем), у той час як для процесу-дитини видається величина нуль; тим самим гарантується, що процес-дитина

звичайно 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.

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 паралельно (рис. Б.3).

Часто застосовуються також слова cobegin, coend. Ця концепція імплементована, наприклад, в мові програмування роботів AL, де можуть однією паралельною програмою керуватись одночасно декілька роботів, а їхній рух координується за допомогою семафорів. Системні семафори дають змогу реалізувати контрольовані зайнятість та звільнення обчислювальних засобів, причому запити, що не можуть бути виконані, спричиняють блокування з подальшим звільненням запитувача.

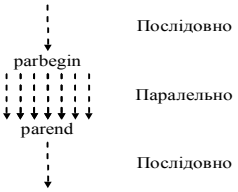

Рис.Б.3. Паралельний блок інструкцій

Синхронізація за допомогою семафорів є 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.

Процеси

Процеси - це паралельна концепція для MIMD-систем. Вони декларуються подібно до процедур і запускаються в явній формі за допомогою деякої інструкції. Якщо процес існує в різноманітних варіантах, то його треба запускати відповідно кілька разів, по можливості з різними параметрами. Синхронізація паралельно виконуваних процесів регулюється за концепцією семафора або монітора із застосуванням умовних змінних величин. Монітори в порівнянні з семафорами забезпечують надійнішу синхронізацію на вищому рівні абстракцій.

Монітор (рис.Б.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.

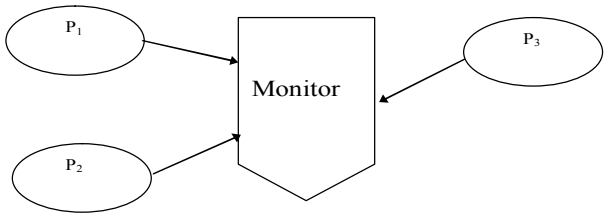

Рис.Б.4. Синхронізація процесів монітором

Однак явна синхронізація паралельних процесів дає не тільки додатковий обсяг роботи щодо керування, а й постійний острах до помилок. Цей спосіб організації паралельних обчислень потребує виняткової 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, яка наведена нижче.

Дистанційний виклик

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

Програмна система поділяється на декілька паралельних процесів, причому кожний процес бере на себе роль сервера чи клієнта. Кожний сервер має в основному один нескінченний цикл, в якому він чекає на черговий запит, виконує обчислення і видає результат в обумовленому вигляді. Кожний сервер при цьому може стати

також клієнтом, і тоді він приймає до уваги дії іншого сервера. Кожний клієнт передає блоки задач одному або декільком відповідно конфігурованим сервер-процесам.

Цей спосіб паралельного розподілення робіт імплементується механізмом дистанційного виклику ("remote procedure call", RPC, рис.Б.5).

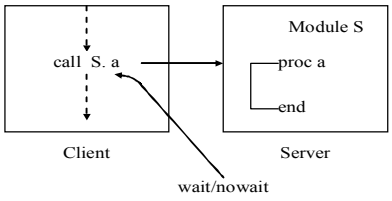

Рис.Б.5. RPC – Механізм розпаралелення

Здатність до обробки інформації 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 з цієї вимоги кращого завантаження апаратури та її більшої ефективності виникають і нові проблеми: параметри зворотного зв'язку недоступні негайно після виконання сервер-операції, оскільки 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 виходу сервера з ладу.

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. 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 або APL . В APL немає жодної контрольної структури високого рівня, яка безумовно потрібна в будь-якій (послідовній чи паралельній) мові програмування.

Як показано на рис.Б.6, математичний запис додавання матриць містить у собі неявну паралельність, яка в цьому випадку досить просто може бути перетворена на деяку паралельну обчислювальну структуру методом автоматичного розпаралелювання (наприклад, паралельність на рівні виразів).

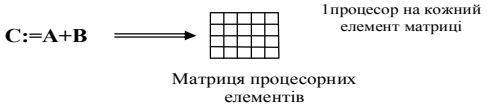

Рис.Б.6. Неявна паралельність при виконанні матричних операцій

Неявна паралельність розвантажує програміста, бо він не повинен 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.

Додаток В

ПРОБЛЕМИ АСИНХРОННОЇ ПАРАЛЕЛЬНОСТІ

Асинхронне паралельне програмування є досить складним і таким, що приводить до помилок. Найчастіше 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 як несумісні дані та взаємоблокування (DeadLock/LiveLock.)

Несумісні дані

Після виконання паралельної операції числове значення або співвідношення між даними стає несумісним 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. Тут необхідно розрізняти 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 інженера становить 1000 гривень. Процес 1 має (поряд з іншим) збільшити заробіток на 50 гривень. Процес 2, що виконується паралельно з першим процесом, має збільшити заробіток на 10%. Залежно від порядку виконання або об'єднання обох процесів при їхньому виконанні, дістанемо різні результати.

Взагалі всі результати можуть бути знайдені шляхом систематичного випробування всіх можливих сумісних послідовностей виконання процесів.

Несумісний аналіз

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

Наприклад, 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).

Не підтверджена залежність

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

У системах банків даних транзакції імплементуються як "атомарні операції". Транзакція може бути або успішною, або зазнає невдачі. Успішна транзакція оброблюється повністю *(commiti),* 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.

Блокування

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

Визначення DeadLocks (взаємне блокування)

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

Взаємні блокування (DeadLocks) з'являються, наприклад, тоді, коли всі процеси блоковані в черзі очікування семафора або умови. Можливе 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 (DeadLocks). 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 (DeadLocks). На рис.В.1 показано небезпеку взаємоблокування у вигляді програми.

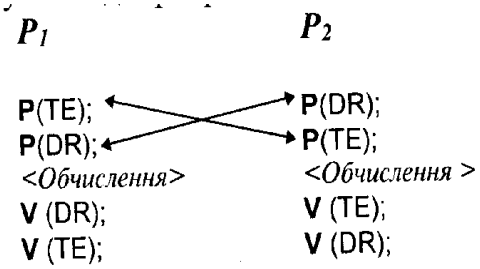

Рис. В.1 Небезпека взаємоблокування

Передумови, за яких може з'явитись взаємоблокування:

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.

А). 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.

Б). Кожний процес має один раз замовити потрібні йому операційні ресурси. Цей метод передбачає ведення протоколу замовлень, якого мають дотримуватися всі процеси. Запит на декілька операційних ресурсів має реалізуватись як атомарна операція, 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 всі без винятку операційні засоби, що були ним зайняті, а потім запитати одночасно всю сукупність операційних засобів, які тепер потребує.

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

Балансування завантаження

Проблемою асинхронного програмування є балансування завантаження процесорів. Неправильний розподіл процесів між процесорами може призвести до значних втрат ефективності.

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. 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 системи падає до можливостей одного процесора, тоді як у даній ситуації можна було б продовжити паралельне виконання процесів.

Щоб уникнути подібних втрат продуктивності, розроблено інші моделі планування процесів, які 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 "міграції 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 системі.

Переваги та недоліки методів балансування завантаження.

"+": Досягається більше завантаження процесорів і не втрачається можлива паралельність.

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

"-": Виникає високий обсяг функцій керування завантаженням процесорів.

"-": 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 дану

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. Загальний обсяг ресурсів для балансування завантаження в даному випадку спричинить необгрунтовані витрати машинного часу.

Додаток Г

ПРОБЛЕМИ СИНХРОННОЇ ПАРАЛЕЛЬНОСТІ

Ніяка з проблем асинхронної паралельності (див. Додаток В) не має відношення до синхроної паралельності. Тут не існує ні несумісних даних, ні блокувань, 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 основному 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.

Індексовані векторні операції

Терміни *Gather* (збирання, операція зчитування) і *Scatter* (розкладання, запис в пам'ять) характеризують дві основні векторні операції, синхронна паралельна реалізація яких пов'язана з труднощами. В принципі виникають проблеми під час векторизації індексованих доступів до даних.

Дані кожного вектора розподіляються покомпонентно 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 Gather і Scatter можуть виконуватися через повільніші, але універсальні структури зв'язку - маршрутизатори, якщо

Приклад:

```
SCALAR s   INTEGER,
VECTOR a   ARRAY [1  10] OF INTEGER,
           u,v   INTEGER,

u  = a[s], (* скалярне індексування *)
u  = a[v], (* векторне індексування *)
```

вони є в системі. В інших випадках - лише послідовне виконання. Деякі SIMD- системи мають проблеми навіть з векторною індексацією локальних масивів в процесорних елементах.

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.

Відображення віртуальних процесорів на фізичні процесори

Застосування цього рівня абстракції дає в розпорядження програмісту 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 такі проблеми:.

- 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, проблема автоматичної трансляції інструкцій щодо обміну даними або декларувань зв'язків (якщо вони наявні в SIMD- системі) в тип мережі зв'язку є набагато важчою;.

- якщо кількість потрібних віртуальних процесорів перевищує кількість наявних фізичних ПЕ, то виконання операцій обміну даними можливе тільки через велику буферну область пам'яті, що приводить до суттєвих втрат 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.

Зменшення пропускної спроможності під час підключення периферійної апаратури

У разі підключення до SIMD- системи периферійної апаратури, наприклад для доступу до великих масивів даних, дуже легко може зменшитись продуктивність з-за швидкості роботи каналів зв’язку.

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 дисплеї, 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. Вузьке місце під час підключення периферійної апаратури виникає тільки і 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. Запобігти цій проблемі можна за допомогою паралельного підключення периферії.

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 масиви паралельного обміну даними значно менші, ніж в MIMD – системі. Швидкість виконання локального обміну даними між фізичними сусідніми ПЕ приблизно така сама, як 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 відповідними двома різними величинами ширини смуги частот комутаційних мереж, вплив яких на продуктивність системи 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 (навіть аж після трансляції в послідовність обміну даними, що виконуватиметься за кілька 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.

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.

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. Однак для цього ще треба вручну вставити в систему відповідну плату маршрутизатора, якою доповнюється комутаційна решітка зменшеного тора.