Міністерство освіти і науки України
Дніпропетровський національний університет
ім. Олеся Гончара

М.М. Ясько

НАВЧАЛЬНИЙ ПОСІБНИК ДО ВИВЧЕННЯ КУРСУ “ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ”

Дніпропетровськ
РВВ ДНУ
2011

УДК 004.4
Я86

Рецензенти: проф., д-р.фіз.-мат.наук., О.І. Михальов
проф., д-р.фіз.-мат.наук., О.А. Приходько

Я86 Ясько, М.М. Навчальний посібник до вивчення курсів “Паралельна обробка даних” та “Мови обчислень та кластерні системи” [Текст] /М.М.Ясько. – Д.: РВВ ДНУ, 2010. – 76с.

Уміщені основні теоретичні відомості щодо архітектури багатопроцесорних обчислювальних систем та поданий детальний опис середовища паралельного програмування MPI та основи програмування з використанням OpenMP. Наведені приклади програм.

Для студентів ДНУ спеціальності “Інформатика”.

Темплан 2010, поз.33.

Навчальне видання
Сергій Вікторович Чернишенко
Микола Миколайович Ясько

Навчальний посібник
до вивчення курсів
“Паралельна обробка даних”
“Мови обчислень та кластерні системи”

Редактор І.І.Стадник
Техредактор Л.П.Замятіна
Коректор А.А.Гриженко

Підписано до друку 15.02.10 Формат 60x84/16.

Папір друкарський. Друк плоский.

Ум. друк. арк. 4,4. Ум. фарбовідб. 4,4 Обл.-вид. арк. 4,6

Тираж 100 пр. Зам. №

РВВ ДНУ, просп. Гагаріна, 72, м. Дніпропетровськ, 49010.

Друкарня ДНУ, вул. Наукова, 5, м. Дніпропетровськ, 49050

© Чернишенко С.В., Ясько М.М., 2010

ЗМІСТ

Вступ

З моменту появи перших електронних обчислювальних машин – комп'ютерів пройшло більше 60 років. За цей час сфера їх використання охопила практично всі галузі людської діяльності. Сьогодні неможливо уявити собі ефективну організацію праці без застосування комп'ютерів у таких галузях, як планування й управління виробництвом, проектування і розробка складних технічних пристроїв, видавнича діяльність, освіта - словом, у всіх галузях, де виникає необхідність в обробці великих обсягів інформації. Проте найбільш важливим, як і раніше, залишається використання їх у тому напрямі, для якого вони власне і створювалися, а саме для виконання великих завдань, що вимагають здійснення величезних обсягів обчислень. Такі завдання виникли в середині минулого століття у зв'язку з розвитком атомної енергетики, авіабудування, ракетно-космічних технологій і ряду інших галузей науки і техніки.

У наш час коло завдань, що вимагають для свого виконання застосування могутніх обчислювальних ресурсів, ще більш розширилося. Це пов'язано з тим, що відбулися фундаментальні зміни в самій організації наукових досліджень. Унаслідок широкого впровадження обчислювальної техніки значно посилився напрям числового моделювання і числових експериментів. Числове моделювання, заповнюючи прогалину між фізичними експериментами й аналітичними підходами, дозволило вивчати явища, які є або дуже складними для дослідження аналітичними методами, або дуже дорогими, або небезпечними для експериментального вивчення. При цьому числовий експеримент дозволив значно прискорити процес наукового і технологічного пошуку. Стало можливим моделювати в реальному часі процеси інтенсивних фізико-хімічних і ядерних реакцій, глобальні атмосферні процеси, процеси економічного і промислового розвитку регіонів і т. д. Очевидно, що виконання таких масштабних завдань вимагає значних обчислювальних ресурсів [1,2].

Ідея паралельної обробки даних як могутнього резерву збільшення продуктивності обчислювальних апаратів була висловлена ще Чарльзом Беббіджем приблизно за сто років до появи першої ЕОМ. Проте рівень розвитку технологій середини XIX століття не дозволив йому реалізувати цю ідею. З появою перших ЕОМ ці ідеї неодноразово ставали відправною точкою у процесі розробки передових і продуктивних обчислювальних систем. Без перебільшення можна сказати, що вся історія розвитку високопродуктивних обчислювальних систем - це історія реалізації ідей паралельної обробки на тому або іншому етапі розвитку комп'ютерних технологій, природно, у поєднанні зі збільшенням частоти роботи електронних схем.

Принципово важливими рішеннями в підвищенні продуктивності обчислювальних систем були:

- введення конвеєрної організації виконання команд;
- включення в систему команд векторних операцій, що дозволяють однією командою обробляти цілі масиви даних;
- розподіл обчислень на безліч процесорів.

Поєднання цих трьох механізмів в архітектурі суперкомп'ютера Earth Simulator, що складається з 5120 векторно-конвеєрних процесорів, і дозволило йому досягти рекордної продуктивності, яка в 20000 разів перевищує продуктивність сучасних персональних комп'ютерів.

Очевидно, що такі системи надзвичайно дорогі й виготовляються в одиничних екземплярах. А що ж виготовляється сьогодні в промислових масштабах? Широку різноманітність вироблюваних у світі комп'ютерів з великим ступенем умовності можна розділити на чотири класи:

- персональні комп'ютери (Personal Computer - PC);
- робочі станції (WorkStation - WS);
- суперкомп'ютери (Supercomputer - SC);
- кластерні системи.

Умовність розділення пов'язана в першу чергу зі швидким прогресом у розвитку мікроелектронних технологій. Продуктивність комп'ютерів у кожному з класів подвоюється останніми роками приблизно за 18 місяців (закон Мура). У зв'язку з цим суперкомп'ютери початку 90-х років часто поступаються в продуктивності сучасним робочим станціям, а персональні комп'ютери починають успішно конкурувати за продуктивністю з робочими станціями. Проте спробуємо якимось чином класифікувати їх.

Персональні комп'ютери. Як правило, в цьому випадку маються на увазі однопроцесорні системи на платформі Intel або AMD, що працюють під керуванням одного користувача операційних систем (Microsoft Windows і ін.). Використовуються в основному як персональні робочі місця.

Робочі станції. Це найчастіше комп'ютери з RISC процесорами з багатокористувацькими операційними системами, що належать до сім'ї ОС UNIX. Містять від одного до чотирьох процесорів. Підтримують видалений доступ. Можуть обслуговувати обчислювальні потреби невеликої групи користувачів.

Суперкомп'ютери. Відмітною особливістю суперкомп'ютерів є те, що це, як правило, великі і надзвичайно дорогі багатопроцесорні системи. У більшості випадків у суперкомп'ютерах використовуються ті ж процесори, що серійно випускаються та використовуються у робочих станціях. Тому часто відмінність між ними не стільки якісна, скільки кількісна. Як приклад, можна назвати 4-процесорну робочу станцію фірми SUN і 64-процесорний суперкомп'ютер фірми SUN. В обох випадках використовуються одні й ті ж мікропроцесори.

Кластерні системи. Останніми роками широко використовуються у всьому світі як дешева альтернатива суперкомп'ютерам. Система необхідної продуктивності збирається з готових комп'ютерів, що серійно випускаються, об'єднаних знову ж таки за допомогою деякого комунікаційного устаткування, що серійно випускається.

Таким чином, багатопроцесорні системи, які раніше асоціювалися в основному із суперкомп'ютерами, в даний час міцно утвердились у всьому діапазоні вироблюваних обчислювальних систем, починаючи від персональних комп'ютерів і закінчуючи суперкомп'ютерами на базі векторно-конвеєрних

процесорів. Ця обставина, з одного боку, збільшує доступність суперкомп'ютерних технологій, а з іншого – підвищує актуальність їх освоєння, оскільки для всіх типів багатопроцесорних систем потрібне застосування спеціальних технологій програмування для того, щоб програма могла повною мірою використовувати ресурси високопродуктивної обчислювальної системи. Зазвичай це досягається розділенням програми за допомогою того або іншого механізму на паралельні гілки, кожна з яких виконується на окремому процесорі.

Суперкомп'ютери розробляються в першу чергу для того, щоб з їх допомогою виконувати складні завдання, що вимагають величезних обсягів обчислень. При цьому мається на увазі, що може бути створена єдина програма, для виконання якої будуть задіяні всі ресурси суперкомп'ютера. Проте не завжди така єдина програма може бути створена або її створення доцільне. Насправді, під час розробки паралельної програми для багатопроцесорної системи мало розбити програму на паралельні гілки. Для ефективного використання ресурсів необхідно забезпечити рівномірне завантаження всіх процесорів, що, у свою чергу, означає, що всі гілки програми повинні виконати приблизно однаковий обсяг обчислювальної роботи. Проте не завжди цього можна досягти. Наприклад, у разі розв’язування деякої параметричної задачі для різних значень параметрів час пошуку розв’язку може значно відрізнятися. У таких випадках, мабуть, доцільніше незалежно виконувати розрахунки для кожного параметра за допомогою звичайної однопроцесорної програми. Але навіть у такому простому випадку можуть знадобитися значні суперкомп'ютерні ресурси, оскільки виконання повного розрахунку на однопроцесорній системі може потребувати дуже тривалого часу. Паралельне виконання безлічі програм для різних значень параметрів дозволяє істотно прискорити процес розв’язування задачі. Нарешті, слід зазначити, що використання суперкомп'ютерів завжди ефективніше для обслуговування обчислювальних потреб великої групи користувачів, ніж використання еквівалентної кількості однопроцесорних робочих станцій, оскільки в цьому випадку за допомогою деякої системи управління завданнями легко забезпечити рівномірне й ефективніше завантаження обчислювальних ресурсів.

На відміну від звичайних багатокористувацьких систем для досягнення максимальної швидкості виконання програм операційні системи суперкомп'ютерів, як правило, не дозволяють розділяти ресурси одного процесора різними програмами, що одночасно виконуються. Тому, як два крайні варіанти, можливі такі режими використання n-процесорної системи:

- всі ресурси використовуються для виконання однієї програми, і тоді ми маємо право чекати n-кратного прискорення роботи програми порівняно з однопроцесорною системою;
- одночасно виконується n звичайних однопроцесорних програм, при цьому користувач має право розраховувати, що на швидкість виконання його програми не впливатимуть інші програми.

Питання про те, наскільки ефективно можуть використовуватися ресурси багатопроцесорної системи в цих двох режимах роботи детальніше може бути вирішене під час вивчення архітектури конкретних суперкомп'ютерів.

Через низку обставин з перерахованих вище 4 класів обчислювальних систем у нашій країні достатнього поширення набули тільки персональні комп'ютери, і відповідно технологія їх використання більш-менш освоєна. Комп'ютерний парк систем, що належать до класу мультипроцесорних робочих станцій і особливо до класу суперкомп'ютерів, надзвичайно малий. Внаслідок цього намітилося істотне відставання в підготовці фахівців у галузі програмування для таких систем, без яких, у свою чергу, неможливе ефективне використання навіть наявного комп'ютерного парку. У цьому плані досить показовий досвід експлуатації в Дніпропетровському національному університеті багатопроцесорної системи CLUSTER40. Система була встановлена в Дніпропетровському національному університеті в 2006 році, проте знадобилося близько 4 місяців, перш ніж на ній запрацювали перші реальні паралельні прикладні програми, призначені для розв’язання задач математичного моделювання. Більше того, значною мірою це виявилося можливим тільки після включення в програмне забезпечення CLUSTER40 мобільного середовища паралельного програмування MPI і заснованих на ній бібліотек паралельних підпрограм. Враховуючи, що моральне старіння комп'ютерів у даний час відбувається дуже швидко, такий тривалий період освоєння, звичайно, недопустимий. На щастя, виявилось, що розроблене програмне забезпечення легко переноситься на інші багатопроцесорні системи, а освоєні технології програмування мають досить універсальний характер.

Даний навчальний посібник адресований тим, хто бажає ознайомитися з технологіями програмування для багатопроцесорних систем. У ньому не обговорюються складні теоретичні питання паралельного програмування. Мова в основному піде про системи, подібні CLUSTER40, - системи з розподіленою пам'яттю. До таких систем належать і дуже поширені в даний час кластерні системи.

Посібник складається з двох частин. У першій частині наводиться огляд архітектури багатопроцесорних обчислювальних систем і засобів їх програмування. Розглядаються питання, пов'язані з підвищенням продуктивності обчислень. У завершальних розділах першої частини розглядаються архітектура й засоби програмування обчислювальних систем кластерного центру кафедри комп’ютерних технологій Дніпропетровського національного університету: CLUSTER40. Друга частина повністю присвячена розгляду найбільш поширеного середовища паралельного програмування - комунікаційної бібліотеки MPI, що стала в даний час фактичним стандартом для розробки мобільних паралельних програм.

1. Огляд архітектури багатопроцесорних обчислювальних систем

У процесі розвитку суперкомп'ютерних технологій ідею підвищення продуктивності обчислювальної системи за рахунок збільшення числа процесорів реалізовували неодноразово. Якщо не вдаватися до історичного екскурсу й обговорення всіх таких спроб, то можна таким чином коротко описати розвиток подій.

Експериментальні розробки зі створення багатопроцесорних обчислювальних систем почалися в 70-х роках XX століття. Однією з перших таких систем стала розроблена в університеті Іллінойсу ЕОМ ILLIAC IV, яка включала 64 (у проекті до 256) процесорні елементи (ПЕ), такі, що працюють за єдиною програмою, яка обробляє вміст власної оперативної пам'яті кожного ПЕ. Обмін даними між процесорами здійснювався через спеціальну матрицю комунікаційних каналів. Вказана особливість комунікаційної системи дала назву "матричні суперкомп'ютери" відповідному класу МОС. Відзначимо, що ширший клас МОС з розподіленою пам'яттю і довільною комунікаційною системою отримав згодом назву "багатопроцесорні системи з масовим паралелізмом", або МОС з MPP-архітектурою (MPP - Massively Parallel Processing). При цьому, як правило, кожен з ПЕ MPP-системи є універсальним процесором, що діє за своєю власною програмою (на відміну від спільної програми для всіх ПЕ матричної МОС).

Перші матричні МОС випускалися буквально поштучно, тому їх вартість була фантастично високою. Серійні ж зразки подібних систем, такі як ICL DAP, що включали до 8192 ПЕ, з'явилися значно пізніше, проте не набули значного поширення через складність програмування МОС з одним потоком управління (з однією програмою, загальною для всіх ПЕ).

Перші промислові зразки мультипроцесорних систем з'явилися на базі векторно-конвеєрних комп'ютерів у середині 80-х років. Найбільш поширеними МОС такого типу були суперкомп'ютери фірми Cray. Проте такі системи були надзвичайно дорогими і вироблялися невеликими серіями. Як правило, в подібних комп'ютерах об'єднувалися від 2 до 16 процесорів, які мали рівноправний (симетричний) доступ до загальної оперативної пам'яті. У зв'язку з цим вони отримали назву симетричні мультипроцесорні системи (Symmetric Multi-Processing - SMP).

Як альтернатива таким дорогим мультипроцесорним системам на базі векторно-конвеєрних процесорів була запропонована ідея будувати еквівалентні за потужністю багатопроцесорні системи з великого числа дешевих мікропроцесорів, що серійно випускалися. Проте дуже скоро виявилося, що SMP-архітектура має дуже обмежені можливості з нарощування числа процесорів у системі через різке збільшення числа конфліктів у разі звернення до загальної шини пам'яті. У зв'язку з цим виправданою здавалася ідея забезпечити кожен процесор власною оперативною пам'яттю, перетворюючи комп'ютер на об'єднання незалежних обчислювальних вузлів. Такий підхід значно збільшив

ступінь масштабованості багатопроцесорних систем, але, у свою чергу, потребував розробки спеціального способу обміну даними між обчислювальними вузлами, що реалізовується зазвичай у вигляді механізму передачі повідомлень (Message Passing). Комп'ютери з такою архітектурою є найбільш яскравими представниками MPP-систем. У даний час ці два напрями (або якісь їх комбінації) є домінуючими в розвитку суперкомп'ютерних технологій.

Щось середнє між SMP і MPP є NUMA-архітектура (Non Uniform Memory Access), в якій пам'ять фізично розділена, але логічно загальнодоступна. При цьому час доступу до різних блоків пам'яті стає неоднаковим. В одній з перших систем цього типу Cray T3D час доступу до пам'яті іншого процесора був у 6 разів більший, ніж до своєї власної.

У даний час розвиток суперкомп'ютерних технологій відбувається за чотирма основними напрямами: векторно-конвеєрні суперкомп'ютери, SMP-системи, MPP-системи і кластерні системи. Розглянемо основні особливості перерахованих архітектур.

1.1. Векторно-конвеєрні суперкомп'ютери

Перший векторно-конвеєрний комп'ютер “Cray-1” з'явився в 1976 році. Архітектура його виявилася настільки вдалою, що він поклав початок цілій сім’ї комп'ютерів, назву якій дали два принципи, закладені в архітектурі процесорів:

- конвеєрна організація обробки потоку команд;
- введення в систему команд набору векторних операцій, які дозволяють оперувати з цілими масивами даних [2].

Довжина одночасно оброблюваних векторів у сучасних векторних комп'ютерах складає, як правило, 128 або 256 елементів. Очевидно, що векторні процесори повинні мати набагато складнішу структуру і по суті справи містити безліч арифметичних пристроїв. Основне призначення векторних операцій полягає в розпаралелюванні виконання операторів циклу, в яких в основному і зосереджена основна частина обчислювальної роботи. Для цього цикли піддаються процедурі векторизації для того, щоб вони могли бути реалізовані з використанням векторних команд. Як правило, це виконується автоматично компіляторами у процесі компіляції програми. Тому векторно-конвеєрні комп'ютери не вимагали якоїсь спеціальної технології програмування, що стало вирішальним чинником в їх успіху на комп'ютерному ринку. Проте було потрібно дотримуватись деяких правил під час написання циклів для того, щоб компілятор міг їх ефективно векторизовувати.

Історично це були перші комп'ютери, до яких повною мірою було застосовне поняття суперкомп'ютер. Як правило, декілька векторно-конвеєрних процесорів (2-16) працюють у режимі із загальною пам'яттю (SMP), утворюючи обчислювальний вузол, а декілька таких вузлів об'єднуються за допомогою комутаторів, утворюючи або NUMA, або MPP-систему. Типовими представниками такої архітектури є комп'ютери “CRAY J90/T90”, “CRAY SV1”,

“NEC SX-4/SX-5”. Рівень розвитку мікроелектронних технологій не дозволяє в даний час проводити однокристальні векторні процесори, тому ці системи досить громіздкі й надзвичайно дорогі. У зв'язку з цим, починаючи з середини 90-х років, коли з'явилися достатньо могутні суперскалярні мікропроцесори, інтерес до цього напряму був значною мірою ослаблений. Суперкомп'ютери з векторно-конвеєрною архітектурою почали програвати системам з масовим паралелізмом. Проте в березні 2002 р. корпорація NEC представила систему “Earth Simulator” з 5120 векторно-конвеєрних процесорів, яка в 5 разів перевищила продуктивність попереднього рекордсмена – MPP-системи “ASCI White” з 8192 суперскалярних мікропроцесорів. Це, звичайно ж, змусило переглянути перспективи розвитку векторно-конвеєрних систем.

1.2. Симетричні мультипроцесорні системи SMP

Характерною межею багатопроцесорних систем SMP-архітектури є те, що всі процесори мають прямий і рівноправний доступ до будь-якої точки загальної пам'яті. Перші системи SMP складалися з декількох однорідних процесорів і масиву загальної пам'яті, до якої процесори підключалися через загальну системну шину. Проте дуже скоро виявилося, що така архітектура непридатна для створення масштабних систем. Перша виникла проблема - велике число конфліктів під час звернення до загальної шини. Гостроту цієї проблеми вдалося частково зняти розділенням пам'яті на блоки, підключення до яких за допомогою комутаторів дозволило розпаралелювати звернення від різних процесорів. Проте і в такому підході неприйнятно великими здавалися накладні витрати для систем більше ніж з 32 процесорами.

Сучасні системи SMP-архітектури складаються, як правило, з декількох однорідних мікропроцесорів, що серійно випускаються, і масиву загальної пам'яті, підключення до якої проводиться або за допомогою загальної шини, або за допомогою комутатора (рис. 1.1).

Наявність загальної пам'яті значно спрощує організацію взаємодії процесорів між собою, а також спрощує програмування, оскільки паралельна програма працює в єдиному адресному просторі. Проте за цією позірною простотою приховані великі проблеми, властиві системам цього типу. Всі вони так або інакше пов'язані з оперативною пам'яттю. Річ у тому, що в даний час навіть в однопроцесорних системах найуразливішим місцем є оперативна пам'ять, швидкість роботи якої значно відстає від швидкості роботи процесора.

Для того щоб згладити цей розрив, сучасні процесори забезпечуються швидкісною буферною пам'яттю (кеш-пам'яттю), швидкість роботи якої значно вища, ніж швидкість роботи основної пам'яті. Як приклад наведемо дані вимірювання пропускної здатності кеш-пам'яті й основної пам'яті для персонального комп'ютера на базі процесора Pentium III 1000 Мгц. У даному процесорі кеш-пам'ять має два рівні:

- L1 (буферна пам'ять команд) - ємність 32 Кб, швидкість обміну 9976 Мб/с;
- L2 (буферна пам'ять даних) - ємність 256 Кб, швидкість обміну 4446 Мб/с.

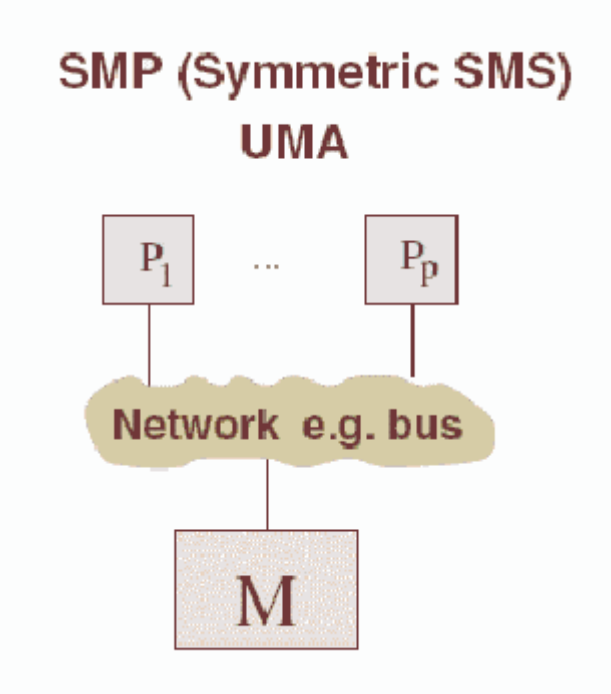

Рис. 1.1. Архітектура симетричних мультипроцесорних систем

Водночас швидкість обміну з основною пам'яттю складає всього 255 Мб/с. Це означає, що для 100% узгодженості зі швидкістю роботи процесора (1000 Мгц) швидкість роботи основної пам'яті повинна бути в 40 разів вища!

Очевидно, що під час проектування багатопроцесорних систем ці проблеми ще більше загострюються. Крім добре відомої проблеми конфліктів у разі звернення до загальної шини пам'яті виникла й нова проблема, пов'язана з ієрархічною структурою організації пам'яті сучасних комп'ютерів. У багатопроцесорних системах, побудованих на базі мікропроцесорів з вбудованою кеш-пам'яттю, порушується принцип рівноправного доступу до будь-якої точки пам'яті. Дані, що знаходяться в кеш-пам'яті деякого процесора, недоступні для інших процесорів. Це означає, що після кожної модифікації копії деякої змінної, що знаходиться в кеш-пам'яті якого-небудь процесора, необхідно проводити синхронну модифікацію самої цієї змінної, що знаходиться в основній пам'яті.

Більш-менш успішно ці проблеми вирішуються в рамках загальноприйнятої в даний час архітектури ccNUMA (cache coherent Non Uniform Memory Access). У цій архітектурі пам'ять фізично розподілена, але логічно загальнодоступна. Це, з одного боку, дозволяє працювати з єдиним адресним простором, а з іншого - збільшує масштабованість систем. Когерентність кеш-пам'яті підтримується на апаратному рівні, що не позбавляє, проте, від накладних витрат на її підтримку. На відміну від класичних систем SMP-пам'ять стає трирівневою:

- кеш-пам'ять процесора;
- локальна оперативна пам'ять;
- видалена оперативна пам'ять.

Час звернення до різних рівнів може відрізнятися на порядок, що дуже ускладнює написання ефективних паралельних програм для таких систем.

Перераховані обставини значно обмежують можливості з нарощування продуктивності ccNUMA систем шляхом простого збільшення числа процесорів. Проте ця технологія дозволяє в даний час створювати системи, що містять до 256 процесорів із загальною продуктивністю порядку 200 млрд операцій за секунду. Системи цього типу серійно проводяться багатьма комп'ютерними фірмами як багатопроцесорні сервери з числом процесорів від 2 до 128 і міцно утримують лідерство в класі малих суперкомп'ютерів. Типовими представниками даного класу суперкомп'ютерів є комп'ютери SUN “StarFire 15K”, “SGI Origin 3000”, “HP Superdome”. Хороший опис однієї з найбільш вдалих систем цього типу - комп'ютера “Superdome” фірми Hewlett-Packard - можна знайти у праці М. Гері і Д. Джонсона [4]. Неприємною властивістю SMP-систем є те, що їх вартість зростає швидше, ніж продуктивність у разі збільшення числа процесорів у системі. Крім того, через затримки під час звернення до загальної пам'яті неминуче взаємне гальмування при паралельному виконанні навіть незалежних програм.

1.3. Системи з масовим паралелізмом (MPP)

Проблеми, властиві багатопроцесорним системам із загальною пам'яттю, просто і природно усуваються в системах із масовим паралелізмом. Комп'ютерами цього типу є багатопроцесорні системи з розподіленою пам'яттю, в яких за допомогою деякого комунікаційного середовища об'єднуються однорідні обчислювальні вузли (рис. 1.2).

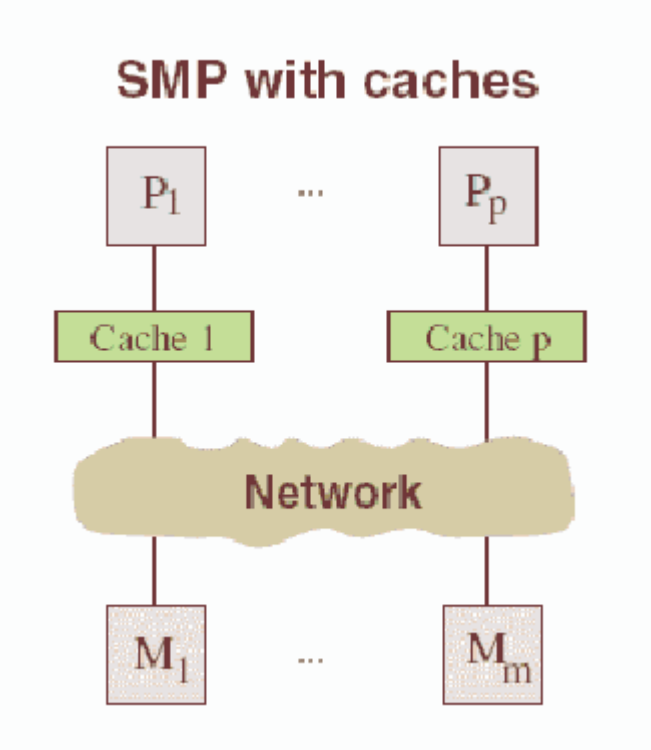

Рис. 1.2. Архітектура систем з розподіленою пам'яттю

Кожен з вузлів складається з одного або декількох процесорів, власної оперативної пам'яті, комунікаційного устаткування, підсистеми введення/виведення, тобто володіє всім необхідним для незалежного функціонування. При цьому на кожному вузлі може функціонувати або повноцінна операційна система (як у системі “RS/6000 SP2”), або урізаний варіант, що підтримує тільки базові функції ядра, а повноцінна ОС працює на спеціальному керуючому комп'ютері (наприклад, як у системі “Cray T3E”).

Процесори в таких системах мають прямий доступ тільки до своєї локальної пам'яті. Доступ до пам'яті інших вузлів реалізується зазвичай за допомогою механізму передачі повідомлень. Така архітектура обчислювальної системи усуває одночасно як проблему конфліктів під час звернення до пам'яті, так і проблему когерентності кеш-пам'яті. Це дає можливість практично необмеженого нарощування числа процесорів у системі, збільшуючи тим самим її продуктивність. Успішно функціонують MPP-системи з сотнями і тисячами процесорів (“ASCI White” - 8192, “Blue Mountain” - 6144). Продуктивність найбільш могутніх систем досягає 10 трильйонів оп/с (10 Tflops). Важливою властивістю MPP-систем є їх високий ступінь масштабованості. Залежно від обчислювальних потреб для досягнення необхідної продуктивності слід просто зібрати систему з потрібним числом вузлів.

На практиці все, звичайно, набагато складніше. Усунення одних проблем, як це зазвичай буває, породжує інші. Для MPP-систем на перший план виходить проблема ефективності комунікаційного середовища. Наприклад, завдання зібрати систему з 100 вузлів може видатися на перший погляд легким. Проте відразу ж виникає питання, яким чином з’єднати в єдине ціле таку кількість вузлів? Найпростішим і найбільш ефективним було б з'єднання кожного процесора з кожним. Але тоді на кожному вузлі було б потрібно 999 комунікаційних каналів, бажано двонаправлених. Очевидно, що це нереально. Різні виробники MPP-систем використовували різні топології. У комп'ютерах “Intel Paragon” процесори утворювали прямокутну двовимірну сітку. Для цього в кожному вузлі достатньо мати чотири комунікаційні канали. У комп'ютерах “Cray T3D/T3E” використовувалася топологія тривимірного тора. Відповідно, у вузлах цього комп'ютера було шість комунікаційних каналів. Фірма nCUBE використовувала у своїх комп'ютерах топологію n-вимірного гіперкуба. Кожна з розглянутих топологій має свої переваги і недоліки. Відзначимо, що в разі обміну даними між процесорами, що не мають відповідного каналу, відбувається трансляція даних через проміжні вузли. Очевидно, що у вузлах повинні бути передбачені якісь апаратні засоби, які звільняли б центральний процесор від участі у трансляції даних. Останнім часом для з'єднання обчислювальних вузлів частіше застосовується ієрархічна система високошвидкісних комутаторів, як це вперше було реалізовано в комп'ютерах “IBM SP2”. Така топологія дає можливість прямого обміну даними між будь-якими вузлами, без участі в цьому проміжних вузлів.

Системи з розподіленою пам'яттю ідеально підходять для паралельного виконання незалежних програм, оскільки при цьому кожна програма виконується на своєму вузлі і ніяким чином не впливає на виконання інших програм. Проте під

час розробки паралельних програм доводиться враховувати складнішу, ніж у SMP-системах, організацію пам'яті. Оперативна пам'ять в MPP-системах звичайно має 3-рівневу структуру:

- кеш-пам'ять процесора;
- локальна оперативна пам'ять вузла;
- оперативна пам'ять інших вузлів.

При цьому відсутня можливість прямого доступу до даних, розподілених в інших вузлах. Для їх використання ці дані повинні бути заздалегідь передані в той вузол, який у даний момент їх потребує. Це значно ускладнює програмування. Крім того, обміни даними між вузлами виконуються значно повільніше, ніж обробка даних у локальній оперативній пам'яті вузлів. Тому написання ефективних паралельних програм для таких комп'ютерів є складнішим завданням, ніж для SMP-систем.

1.4. Класифікація обчислювальних систем

Велику різноманітність обчислювальних систем породило природне бажання ввести для них певну класифікацію. Ця класифікація повинна однозначно відносити ту або іншу обчислювальну систему до деякого класу, який, у свою чергу, повинен достатньо повно її характеризувати. Таких спроб було багато. Одна з перших класифікацій, посилання на яку найчастіше зустрічаються в літературі, була запропонована М. Флінном у кінці 60-х років минулого століття. Вона базується на поняттях двох потоків: команд і даних, які обробляє процесор. На основі числа цих потоків Флінн виділив чотири класи архітектури:

1. SISD (Single Instruction Single Data) - єдиний потік команд і єдиний потік даних. По суті справи, це класична машина фон Неймана. До цього класу належать всі однопроцесорні системи. У таких машинах всі команди виконуються послідовно одна за одною. Для збільшення швидкості виконання команд може використовуватися конвеєрна обробка.
2. SIMD (Single Instruction Multiple Data) - єдиний потік команд і багато потоків даних. Паралелізм у таких архітектурах полягає в можливості одночасного виконання однієї й тієї ж операції над декількаами елементами даних. Типовими представниками є матричні комп’ютери, в яких всі процесорні елементи виконують одну й ту ж програму, яка обробляє свої (різні для кожного процесу) локальні дані. Деякі автори до цього класу відносять і векторно-конвеєрні комп’ютери, якщо кожний елемент вектора розглядати як окремий елемент потоку даних. Спеціалізовані векторні процесори часто вбудовуються в комп’ютери загального призначення. Зокрема, в багатьох сучасних мікропроцесорах вбудовані обмежені можливості векторних обчислень.

3. MISD (Multiple Instruction Single Date) - множинний потік команд і єдиний потік даних. М. Флінн не зміг навести жодного прикладу реально існуючої системи, що працює за цим принципом. Деякі автори як представники такої архітектури називають векторно-конвеєрні комп'ютери, проте такий погляд не отримав широкої підтримки.
4. MIMD (Multiple Instruction Multiple Date) - множинний потік команд і множинний потік даних. Найпоширеніший на сьогоднішній день клас паралельних архітектур, в яких кожний процесор здатний працювати незалежно від інших над своїм завданням. До цього класу належать практично всі сучасні багатопроцесорні системи.

Запропонована класифікація використовується і в наш час для початкової характеристики того чи іншого комп’ютера. Якщо сказано, що комп’ютер належить до класу SIMD або MIMD, то зразу стають зрозумілими базові принципи його роботи і в багатьох випадках цього буває достатньо. Але ця схема має і певні вади. Так, деякі сучасні архітектури явно не вписуються в дану класифікацію. Інший недолік полягає в тому, що всі сучасні багатопроцесорні системи належать до одного класу MIMD, тобто дана класифікація не враховує різницю за числом процесорів, топологію зв’язку між ними, способу організації пам’яті , тощо.

Протягом останніх сорока років здійснювалося багато спроб зробити доповнення до класифікації Флінна, але вони не є загальноприйнятими і тому в даному посібнику розглядатися не будуть.

2. Засоби програмування багатопроцесорних систем

Ефективність використання комп'ютерів залежить від складу і якості програмного забезпечення, встановленого на них. У першу чергу це стосується програмного забезпечення, призначеного для розробки прикладних програм. Так, недостатня розвиненість таких засобів для систем MPP-типу довгий час була стримувальним чинником для їх широкого використання. У даний час ситуація змінилася, і завдяки кластерним технологіям MPP-системи стали найпоширенішим і найдоступнішим різновидом високопродуктивних обчислювальних систем. У даному розділі ми стисло розглянемо засоби паралельного програмування для багатопроцесорних систем.

Як наголошувалося раніше, основною характеристикою при класифікації багатопроцесорних систем є наявність загальної (SMP-системи) або розподіленої (MPP-системи) пам'яті. Ця відмінність - найважливіший чинник, що визначає способи паралельного програмування і відповідно структуру програмного забезпечення.

2.1. Системи із загальною пам'яттю

До систем цього типу належать комп'ютери з SMP-архітектурою, різні різновиди NUMA-систем і мультипроцесорні векторно-конвеєрні комп'ютери. Основну характеристику цих комп'ютерів можна передати словом "єдиний": єдина оперативна пам'ять, єдина операційна система, єдина підсистема введення-виведення. Тільки процесори утворюють множину. Єдина UNIX-подібна операційна система, що керує роботою всього комп'ютера, функціонує у вигляді безлічі процесів. Кожна призначена для користувача програма також запускається як окремий процес. Операційна система сама якимось чином розподіляє процеси по процесорах. У принципі, для розпаралелювання програм можна застосовувати механізм породження процесів. Проте цей механізм не дуже зручний, оскільки кожен процес функціонує у своєму адресному просторі, і основне достоїнство цих систем - загальна пам'ять - не може бути використане простим і природним чином. Для розпаралелювання програм застосовується механізм породження потоків (threads) - легковагих процесів, для яких не створюється окремий адресний простір, але які на багатопроцесорних системах також розподіляються по процесорах. У мові програмування С можливе пряме застосування цього механізму для розпаралелювання програм за допомогою виклику відповідних системних функцій, а в компіляторах з мови FORTRAN цей механізм застосовується або для автоматичного розпаралелювання, або в режимі задання розпаралелюючих директив компілятору (такий підхід підтримують і компілятори з мови С).

Всі виробники симетричних мультипроцесорних систем тією чи іншою мірою підтримують стандарт POSIX Pthread і включають у програмне забезпечення компілятори для популярних мов програмування, що

розпаралелюють код або надають набір директив компілятору для розпаралелювання програм. Зокрема, багато постачальників комп'ютерів SMP-архітектури (Sun, HP, SGI) у своїх компіляторах надають спеціальні директиви для розпаралелювання циклів. Проте ці набори директив, по-перше, досить обмежені і, по-друге, несумісні між собою. У результаті цього розробникам доводиться розпаралелювати прикладні програми окремо для кожної платформи.

Останніми роками все більш популярною стає система програмування OPENMP [7;10], що є багато в чому узагальненням і розширенням цих наборів директив. Інтерфейс OPENMP задуманий як стандарт для програмування в моделі загальної пам'яті. У OPENMP входять специфікації набору директив компілятору, процедур і змінних середовища. По суті справи, він реалізує ідею "інкрементального розпаралелювання", запозичену з мови HPF (High Performance Fortran - Fortran для високопродуктивних обчислень). Розробник не створює нову паралельну програму, а просто додає в текст послідовної програми OpenMP-директиви. При цьому система програмування OPENMP надає розробникові великі можливості з контролю над поведінкою паралельного застосування. Вся програма розбивається на послідовні й паралельні області. Всі послідовні області виконує головний потік, що породжується під час запуску програми, а при вході в паралельну область головний потік породжує додаткові потоки. Передбачається, що OpenMP-програма без будь-якої модифікації повинна працювати як на багатопроцесорних системах, так і на однопроцесорних. В останньому випадку директиви OpenMP просто ігноруються. Слід зазначити, що наявність загальної пам'яті не перешкоджає застосуванню технологій програмування, розроблених для систем з розподіленою пам'яттю. Багато виробників SMP-систем надають також такі технології програмування, як MPI і PVM. У цьому випадку як комунікаційне середовище виступає пам'ять, що розподіляється між процесами.

2.2. Системи з розподіленою пам'яттю

У системах цього типу на кожному обчислювальному вузлі функціонують власні копії операційної системи, під керуванням яких виконуються незалежні програми. Це можуть бути як дійсно незалежні програми, так і паралельні гілки однієї програми. У цьому випадку єдино можливим механізмом взаємодії між ними є механізм передачі повідомлень.

Прагнення добитися максимальної продуктивності примушує розробників у процесі реалізації механізму передачі повідомлень враховувати особливості архітектури багатопроцесорної системи. Це сприяє написанню ефективніших, але орієнтованих на конкретний комп'ютер програм. Водночас незалежними розробниками програмного забезпечення було запропоновано безліч реалізацій механізму передачі повідомлень, незалежних від конкретної платформи. Найбільш відомими з них є EXPRESS компанії Parasoft і комунікаційна бібліотека PVM (Parallel Virtual Machine), розроблена в Oak Ridge National Laboratory.

У 1994 р. був прийнятий стандарт механізму передачі повідомлень MPI (Message Passing Interface)[9]. Він готувався з 1992 по 1994 рр. групою Message

Passing Interface Forum, до якої увійшли представники більше ніж 40 організацій з Америки і Європи. Основна мета, яку ставили перед собою розробники MPI, - це забезпечення повної незалежності застосувань, написаних з використанням MPI, від архітектури багатопроцесорної системи без істотної втрати продуктивності. За задумом авторів, це повинно було стати могутнім стимулом для розробки прикладного програмного забезпечення і стандартизованих бібліотек підпрограм для багатопроцесорних систем з розподіленою пам'яттю. Підтвердженням досягнення цієї мети служить той факт, що в даний час цей стандарт підтримується практично всіма виробниками багатопроцесорних систем. Реалізації MPI успішно працюють не тільки на класичних MPP-системах, але також на SMP-системах і на мережах робочих станцій (у тому числі і неоднорідних).

MPI - це бібліотека функцій, що забезпечує взаємодію паралельних процесів за допомогою механізму передачі повідомлень. Підтримуються інтерфейси для мов C і FORTRAN. Останнім часом додана підтримка мови C++. Бібліотека включає безліч функцій передачі повідомлень типу точка-точка, розвинений набір функцій для виконання колективних операцій і управління процесами паралельного застосування. Основна відмінність MPI від попередників у тому, що явно вводяться поняття груп процесів, з якими можна оперувати як з кінцевими множинами, а також областей зв'язку і комунікаторів, що описують ці області зв'язку. Це надає програмісту дуже гнучкі засоби для написання ефективних паралельних програм. У даний час MPI є основною технологією програмування для багатопроцесорних систем з розподіленою пам'яттю.

Альтернативний підхід надає парадигма паралельної обробки даних, реалізована в мові високого рівня HPF [8]. Від програміста потрібно тільки задати розподіл даних по процесорах, а компілятор автоматично генерує виклики функцій синхронізації та передачі повідомлень (невдале розташування даних може викликати істотне збільшення накладних витрат). Для розпаралелювання циклів використовуються або спеціальні конструкції мови (оператор FORALL), або директиви компілятору, що задаються у вигляді псевдокоментарів ($HPF INDEPENDENT). Мова HPF реалізує ідею інкрементального розпаралелювання і модель загальної пам'яті на системах з розподіленою пам'яттю. Ці дві обставини і визначають простоту програмування і відповідно привабливість цієї технології. Одна й та ж програма, без будь-якої модифікації, повинна ефективно працювати як на однопроцесорних системах, так і на багатопроцесорних обчислювальних системах.

Програми на мові HPF істотно коротші від функціонально ідентичних програм, що використовують прямі виклики функцій обміну повідомленнями. Мабуть, мови цього типу активно розвиватимуться й поступово витіснять з широкого обігу пакети передачі повідомлень. Останнім відводитиметься роль, яка в сучасному програмуванні відводиться асемблеру: до них вдаватимуться лише для розробки мов високого рівня і під час написання бібліотечних підпрограм, від яких потрібна максимальна ефективність.

У середині 90-х років, коли з'явився HPF, з ним пов'язувалися великі надії, проте труднощі з його реалізацією поки що не дозволили створити досить

ефективних компіляторів. Мабуть, найбільш вдалими були проекти, в яких компіляція HPF-програм виконується двома етапами. На першому етапі HPF-програма перетвориться на стандартну Фортран-програму, доповнену викликами комунікаційних підпрограм. А на другому етапі відбувається її компіляція стандартними компіляторами. Тут слід відзначити систему компіляції Adaptor, розроблену німецьким Інститутом алгоритмів і наукових обчислень, і систему розробки паралельних програм DVM, створену в Інституті прикладної математики ім. М.В. Келдиша РАН. Система DVM підтримує не тільки мову Фортран, але і С [4]. Наш досвід роботи з системою Adaptor показав, що в більшості випадків ефективність паралельних HPF-програм значно нижча, ніж MPI-програм. Проте в деяких простих випадках Adaptor дозволяє розпаралелювати звичайну програму додаванням всього декількох рядків.

Слід зазначити область, де механізму передачі повідомлень немає альтернативи, - це обслуговування функціонального паралелізму. Якщо кожен вузол виконує свій власний алгоритм, що істотно відрізняється від того, що робить сусідній процесор, а взаємодія між ними має нерегулярний характер, то нічого іншого, крім механізму передачі повідомлень, запропонувати неможливо.

Описані в даному розділі технології, звичайно ж, не вичерпують весь список – наведені тільки найбільш універсальні і широко використовувані в даний час. Докладніший огляд сучасних технологій паралельного програмування можна знайти в праці [5].

2.3. Паралельне програмування на MPP-системах

Розв’язування на комп'ютері обчислювальної задачі згідно з обраним алгоритмом полягає у виконанні деякого фіксованого обсягу арифметичних операцій. Прискорити виконання задачі можна одним з трьох способів:

1) використовувати продуктивнішу обчислювальну систему із швидшим процесором і більш продуктивною системною шиною;
2) оптимізувати програму, наприклад, з метою ефективнішого використання швидкісної кеш-пам'яті;
3) розподілити обчислювальну роботу між декількома процесорами, тобто перейти на паралельні технології.

Розробка паралельних програм є досить трудомістким процесом, особливо для систем MPP-типу, тому, перш ніж приступати до цієї роботи, важливо правильно оцінити як очікуваний ефект від розпаралелювання, так і трудомісткість виконання цієї роботи. Очевидно, що без розпаралелювання не обійтися під час програмування алгоритмів розв’язування таких задач, які у принципі не можуть бути розв’язані на однопроцесорних системах. Це може виявитися у двох випадках: або коли для виконання завдання потрібно дуже багато часу, або коли для програми недостатньо оперативної пам'яті на однопроцесорній системі. Для невеликих завдань часто виявляється, що

паралельна версія працює повільніше, ніж однопроцесорна. Така ситуація спостерігається, наприклад, під час розв'язування на обчислювальному кластері систем лінійних алгебричних рівнянь, що містять менше 100 невідомих. Помітний ефект від розпаралелювання починає спостерігатися під час розв'язання систем з 1000 і більше невідомими. На кластерних системах ситуація ще гірша. Розробники вже згадуваного раніше пакета ScaLAPACK для багатопроцесорних систем з прийнятним співвідношенням між продуктивністю вузла і швидкістю обміну (сформульованого в розділі 1.4) надають таку формулу для кількості процесорів, яку рекомендується використовувати під час розв'язування задач лінійної алгебри: $P=M*N/10^6$, де MxN - розмірність матриці.

Інакше кажучи, кількість процесорів повинна бути такою, щоб на процесор доводився блок матриці розміром приблизно 1000x1000. Ця формула, звичайно, має рекомендаційний характер, проте наочно ілюструє, для задач якого масштабу розроблявся пакет ScaLAPACK. Зростання ефективності розпаралелювання у разі збільшення розміру розв'язуваної системи рівнянь пояснюється дуже просто: у разі збільшення розмірності системи рівнянь обсяг обчислювальної роботи зростає пропорційно n^3, а обсяг обмінів між процесорами пропорційно n^2. Це знижує відносну частку комунікаційних витрат у разі збільшення розмірності системи рівнянь. Проте, як ми побачимо далі, не тільки комунікаційні витрати впливають на ефективність паралельного застосування.

Паралельні технології на MPP-системах допускають дві моделі програмування (схожі на класифікацію М. Флінна):

1) SPMD (Single Program Multiple Date) - на всіх процесорах виконуються копії однієї програми, яка обробляє різні блоки даних;
2) MPMD (Multiple Program Multiple Date) - на процесорах виконуються різні програми, які обробляють різні дані.

Другий варіант іноді називають функціональним розпаралелюванням. Такий підхід, зокрема, застосовується в системах обробки відеоінформації, коли безліч квантів даних повинні проходити декілька етапів обробки. У цьому випадку цілком виправданою буде конвеєрна організація обчислень, за якої кожен етап обробки виконується на окремому процесорі. Проте такий підхід має дуже обмежене застосування, оскільки організувати достатньо довгий конвеєр та ще з рівномірним завантаженням всіх процесорів, досить складно.

Найбільш поширеним режимом роботи на системах з розподіленою пам'яттю є завантаження в деяке число процесорів однієї й тієї ж копії програми. Розглянемо питання, яким чином при цьому можна досягти більшої швидкості розв'язання задачі порівняно з однопроцесорною системою.

Розробка паралельної програми передбачає розбиття завдання на P підзадач, кожна з яких розв'язується на окремому процесорі. Таким чином, спрощену схему паралельної програми, що використовує механізм передачі повідомлень, можна подати таким чином:

```
IF (proc_id.EQ.0) THEN
  Ierr = task1
```

```
END IF
IF (proc_id.EQ.1) THEN
  Ierr = task2
END IF
...
result = reduce(result_task1, result_task2, ...)
END
```

Тут proc_id - ідентифікатор процесора, а функція reduce формує певний глобальний результат на основі локальних результатів, одержаних на кожному процесорі. У цьому випадку одна й та ж копія програми виконуватиметься на P процесорах, але кожен процесор розв'язуватиме тільки свою підзадачу. Якщо розбиття на підзадачі достатньо рівномірне, а накладні витрати на обміни не дуже великі, то можна чекати близького до P коефіцієнта прискорення розв'язування задачі.

Окремого обговорення вимагає поняття ***підзадача***. У паралельному програмуванні це поняття має досить широкий сенс. У MPMD-моделі під підзадачею розуміється деякий функціонально виділений фрагмент програми. У SPMD-моделі під підзадачею частіше розуміється обробка деякого блока даних. На практиці процедура розпаралелювання найчастіше застосовується до циклів. Тоді як окремі підзадачі можуть виступати екземпляри тіла циклу, що виконуються для різних значень змінної циклу. Розглянемо простий приклад:

```
DO I = 1,1000
  C(I) = C(I)+ A(I+1)
END DO
```

У даному прикладі можна виділити 1000 незалежних підзадач обчислення компонентів вектора C, кожна з яких, у принципі, може бути виконана на окремому процесорі. Припустимо, що в розпорядженні програміста є 10-процесорна система, тоді як незалежну підзадачу можна оформити обчислення 100 елементів вектора C. При цьому до виконання обчислень необхідно ухвалити рішення про спосіб розміщення цих масивів у пам'яті процесорів. Тут можливі два варіанти:

1. Всі масиви цілком зберігаються в кожному процесорі, тоді процедура розпаралелювання зводиться до обчислення стартового і кінцевого значень змінної циклу для кожного процесора. У кожному процесорі зберігатиметься своя копія всього масиву, в якій буде модифікована тільки частина елементів. У кінці обчислень, можливо, буде потрібно збирання модифікованих частин з усіх процесорів.
2. Всі або частина масивів розподілені по процесорах, тобто в кожному процесорі зберігається 1/P частини масиву. Тоді може знадобитися алгоритм встановлення зв'язку індексів локального масиву в деякому процесорі з глобальними індексами всього масиву, наприклад, якщо значення елемента масиву є деякою функцією індексу. Якщо в процесі обчислень виявиться, що якісь потрібні компоненти масиву відсутні в даному процесорі, то буде потрібне їх пересилання з інших процесорів.

Відзначимо, що питання розподілу даних по процесорах і зв'язок цього розподілу з ефективністю паралельної програми є основним питанням паралельного програмування. Зберігання копій усіх масивів у всіх процесорах у багатьох випадках зменшує накладні витрати на пересилання даних, проте не дає виграшу в плані обсягу розв'язуваної задачі і створює труднощі синхронізації копій масиву у разі незалежної зміни елементів цього масиву різними процесорами. Розподіл масивів по процесорах дозволяє розв'язувати значно об'ємніші задачі (їх якраз і є сенс розпаралелювати), але тоді на перший план виступає проблема мінімізації пересилань даних.

Розглянутий вище приклад достатньо добре укладається в схему методологічного підходу до розв'язання задачі на багатопроцесорній системі, який формулюється Фостером [9]. Автор виділяє 4 етапи розробки паралельного алгоритму:

1) розбиття задачі на мінімальні незалежні підзадачі (partitioning);
2) встановлення зв'язків між підзадачами (communication);
3) об'єднання підзадач з метою мінімізації комунікацій (agglomeration);
4) розподіл збільшених підзадач по процесорах так, щоб забезпечити рівномірне завантаження процесорів (mapping).

Ця схема, звичайно, не більше ніж опис філософії паралельного програмування, яка лише підкреслює відсутність будь-якого формалізованого підходу в паралельному програмуванні для MPP-систем. Якщо 1-й і 2-й пункти мають більш-менш однозначне розв'язання, то розв'язання задач 3-го і 4-го пунктів ґрунтується головним чином на інтуїції програміста. Щоб проілюструвати цю обставину, розглянемо таке завдання. Припустимо, потрібно досліджувати поведінку визначника матриці залежно від деякого параметра. Один з підходів полягає в тому, щоб написати паралельну версію підпрограми обчислення визначника й обчислити його значення для досліджуваного інтервалу значень параметра. Проте, якщо розмір матриці відносно невеликий, може виявитися, що значні зусилля на розробку паралельної підпрограми обчислення визначника не дадуть істотного виграшу в швидкості роботи програми. В цьому випадку, найімовірніше, продуктивнішим підходом буде використати для знаходження визначника стандартної оптимізованої однопроцесорної підпрограми, а по процесорах розкласти досліджуваний діапазон змін параметра.

На закінчення розглянемо, які властивості повинна мати багатопроцесорна система MPP-типу для виконання на ній паралельних програм. Мінімально необхідний набір потрібних засобів надзвичайно малий:

1. Процесори в системі повинні мати унікальні ідентифікатори (номери).
2. Має існувати функція ідентифікації процесором самого себе.
3. Повинні існувати функції обміну між двома процесорами: посилання повідомлення одним процесором і прийом повідомлення іншим процесором.

Парадигма передачі повідомлень передбачає асиметрію функцій передачі і прийому повідомлень. Ініціатива ініціалізації обміну належить стороні, що передає. Приймальний процесор може прийняти тільки те, що йому було послано. Різні реалізації механізму передачі повідомлень для полегшення розробки паралельних програм вводять ті або інші розширення мінімально необхідного набору функцій.

2.4. Ефективність паралельних програм

В ідеалі розв'язування задачі на P процесорах повинне виконуватися в P раз швидше, ніж на одному процесорі, або/і повинне дозволити розв'язати завдання з обсягами даних, в P раз більшими. Насправді таке прискорення практично ніколи не досягається. Причина цього добре ілюструється законом Амдала [6]

$$S \le (f+(1+f)/P)^{-1} , \qquad (2.1)$$

де S - прискорення роботи програми на P процесорах, а f - частка непаралельного коду в програмі.

Закон Амдала показує, що приріст ефективності обчислень залежить від алгоритму задачі й обмежений зверху для будь-якої задачі де $f \neq 0$. Більше того, не для всякої задачі має сенс нарощувати кількість процесів в обчислювальній системі. Так, якщо враховувати час, необхідний для передачі даних між вузлами обчислювальної системи, то залежність часу обчислень від числа вузлів буде мати максимум. Це накладає обмеження на масштабування системи, тобто з деякого моменту додавання нових вузлів буде збільшувати час розв'язування задачі.

Закон Амдала справджується і в разі програмування в моделі загальної пам'яті, і в моделі передачі повідомлень. Дещо інше значення вкладається в поняття "частка непаралельного коду". Для SMP-систем цю частку утворюють ті оператори, які виконує тільки головна нитка програми. Для MPP-систем непаралельна частина коду утворюється за рахунок операторів, виконання яких дублюється всіма процесорами. Оцінити цю величину з аналізу тексту програми практично неможливо. Таку оцінку можуть дати тільки реальні прорахунки на різному числі процесорів. З формули (2.1) випливає, що P-кратне прискорення може бути досягнуте, тільки коли частка непаралельного коду дорівнює 0. Очевидно, що досягти цього практично неможливо. Дуже наочно дію закону Амдала демонструє табл. 1.

Таблиця 1. Прискорення роботи програми залежно від частки непаралельного коду

Число	Частка послідовних обчислень %

процесорів	50	25	10	5	2
	Прискорення програми				
2	1.33	1.60	1.82	1.90	1.96
4	1.60	2.28	3.07	3.48	3.77
8	1.78	2.91	4.71	5.93	7.02
16	1.88	3.36	6.40	9.14	12.31
32	1.94	3.66	7.80	12.55	19.75
512	1.99	3.97	9.83	19.28	45.63

Відповідно до табл. 1, якщо, наприклад, частка послідовного коду складає 2%, то більше ніж 50-кратне прискорення у принципі одержати неможливо. З іншого боку, мабуть, недоцільно запускати таку програму на 2048 процесорах для того, щоб одержати 49-кратне прискорення. Проте таке завдання достатньо ефективно виконуватиметься на 16 процесорах, а в деяких випадках втрата 37% продуктивності у процесі виконання завдання на 32 процесорах може бути цілком прийнятною. В деякому розумінні, закон Амдала встановлює граничне число процесорів, на якому програма виконуватиметься з прийнятною ефективністю залежно від частки непаралельного коду. Зазначимо, що ця формула не враховує накладні витрати на обміни між процесорами, тому в реальному житті ситуація може бути ще гіршою.

Не слід забувати, що розпаралелювання програми - це лише один із засобів прискорення її роботи. Не менший ефект, а іноді й більший, може дати оптимізація однопроцесорної програми. Надзвичайної актуальності ця проблема набула останнім часом через великий розрив у швидкості роботи кеш-пам'яті й основної пам'яті. На жаль, часто цій проблемі не приділяється належної уваги. Це призводить до того, що витрачаються значні зусилля на розпаралелювання явно неефективних програм. Цю ситуацію достатньо наочно проілюструє наступний розділ.

3. Середовище паралельного програмування MPI

Комунікаційна бібліотека MPI стала загальновизнаним стандартом у паралельному програмуванні із застосуванням механізму передачі повідомлень. Повний і строгий опис середовища програмування MPI можна знайти в авторському описі розробників [8;9]. На жаль, дотепер немає перекладу цього документа українською мовою. Пропонований увазі читача опис MPI не є повним, проте містить достатньо матеріалу для написання досить складних програм. Мета даного посібника полягає в тому, щоб, по-перше, ознайомити читача з функціональними можливостями цієї комунікаційної бібліотеки і, по-друге, розглянути набір підпрограм, достатній для програмування будь-яких алгоритмів. Приклади паралельних програм з використанням комунікаційної бібліотеки MPI, наведені в кінці даної частини, перевірені на різних багатопроцесорних системах (Linux-кластері, 2-процесорній системі Xeon).

3.1. Загальна організація MPI

MPI-програма є набір незалежних процесів, кожний з яких виконує свою власну програму (не обов'язково одну й ту ж), написану на мові C або FORTRAN. З'явилися реалізації MPI для C++, проте розробники стандарту MPI за них відповідальності не несуть. Процеси MPI-програми взаємодіють один з одним за допомогою виклику комунікаційних процедур. Як правило, кожен процес виконується у своєму власному адресному просторі, проте допускається і розділення пам'яті. MPI не специфікує модель виконання процесу — це може бути як послідовний процес, так і багатопотоковий. MPI не надає ніяких засобів для розподілу процесів по обчислювальних вузлах і для запуску їх на виконання. Ці функції покладаються або на операційну систему, або на програміста. Зокрема, на кластерах - спеціальна команда mpiexec, яка припускає, що скомпільовані програми вже якимось чином розподілені по комп'ютерах кластера. Описаний у даному навчальному посібнику стандарт MPI 1.1 не містить механізмів динамічного створення і знищення процесів під час виконання програми. MPI не накладає будь-яких обмежень на те, як процеси будуть розподілені по процесорах, зокрема, можливий запуск MPI-програми з декількома процесами на звичайній однопроцесорній системі.

Для ідентифікації наборів процесів вводиться поняття ***групи***, що об'єднує всі або якусь частину процесів. Кожна група утворює ***область зв'язку***, з яким зв'язується спеціальний об'єкт - ***комунікатор*** області зв'язку. Процеси всередині групи нумеруються цілим числом в діапазоні 0..groupsize-1. Всі комунікаційні операції з деяким комунікатором виконуватимуться тільки всередині області зв'язку, що описується цим комунікатором. У процесі ініціалізації MPI створюється область зв'язку, що містить всі процеси MPI-програми, з якою зв'язується комунікатор MPI_COMM_WORLD. У більшості випадків на кожному процесорі запускається один окремий процес, і тоді терміни “процес” і “процесор” стають синонімами, а величина groupsize стає рівною NPROCS - числу

процесорів, виділених завданню. У подальшому обговоренні будемо мати на увазі саме таку ситуацію і не будемо строго дотримуватися термінології.

Отже, якщо сформулювати коротко, MPI - це бібліотека функцій, що забезпечує взаємодію паралельних процесів за допомогою механізму передачі повідомлень. Це достатньо об'ємна і складна бібліотека, що складається приблизно з 130 функцій, до числа яких входять:

- функції ініціалізації і закриття MPI-процесів;
- функції реалізації комунікаційних операцій типу точка-точка;
- функції реалізації колективних операцій;
- функції для роботи з групами процесів і комунікаторами;
- функції для роботи зі структурами даних;
- функції формування топології процесів.

Набір функцій бібліотеки MPI виходить далеко за межі набору функцій, мінімально необхідного для підтримки механізму передачі повідомлень, описаного в першій частині. Проте складність цієї бібліотеки не повинна лякати користувачів, оскільки вся ця безліч функцій призначена для полегшення розробки ефективних паралельних програм. Врешті-решт, користувачу належить право самому вирішувати, які засоби з арсеналу, що надається, використовувати, а які ні. У принципі, будь-яка паралельна програма може бути написана з використанням всього 6-MPI функцій, а достатньо повне і зручне середовище програмування створює набір з 24 функцій.

Кожна з MPI-функцій характеризується способом виконання:

1. Локальна функція – виконується усередині конкретного процесу. Її завершення не вимагає комунікацій.
2. Нелокальна функція – для її завершення потрібне виконання MPI-процедури іншим процесом.
3. Глобальна функція - процедуру повинні виконувати всі процеси групи. Недотримання цієї умови може призводити до зависання завдання.
4. Блокуюча функція - повернення керування з процедури гарантує можливість повторного використання параметрів, що беруть участь у виклику. Ніяких змін у стані процесу, що викликав блокуючий запит, до виходу з процедури не буде.
5. Неблокувальна функція - повернення з процедури відбувається негайно, без очікування закінчення операції і до того, як буде дозволене повторне використання параметрів, що беруть участь у запиті. Завершення неблокувальних операцій здійснюється спеціальними функціями.

Використання бібліотеки MPI має деякі відмінності в мовах C і FORTRAN.

У мові C всі процедури є функціями, і більшість з них повертає код помилки. Використовуючи імена підпрограм і іменованих констант, необхідно строго дотримуватися регістра символів. Масиви індексуються з 0. Логічні змінні подаються типом int (true відповідає 1, a false - 0). Визначення всіх іменованих

констант, прототипів функцій і визначення типів виконується підключенням файлу mpi.h. Введення власних типів у MPI було продиктовано тією обставиною, що стандартні типи мови С на різних платформах можуть мати різний розмір. MPI допускає можливість запуску процесів паралельної програми на комп'ютерах різних платформ, забезпечуючи при цьому автоматичне перетворення даних під час пересилань. У табл. 2 наведена відповідність зумовлених у MPI типів стандартним типам мови С та Фортран.

Таблиця 2. Відповідність між MPI-типами і типами мови С та Фортран

Тип MPI	Тип мови С	Тип мови Фортран
MPI_CHAR	signed char	integer(1)
MPI_SHORT	signed short int	integer(2)
MPI_INT	signed int	integer(4)
MPI_LONG	signed long int	integer(8)
MPI_UNSIGNED_CHAR	unsigned char	
MPI_UNSIGNED_SHORT	unsigned short int	
MPI_UNSIGNED	unsigned int	
MPI_UNSIGNED_LONG	unsigned long int	
MPI_FLOAT	float	real(4)
MPI_DOUBLE	double	real(8)
MPI_LONG_DOUBLE	long double	real(12)
MPI_BYTE		integer(1)
MPI_PACKED		

У старій реалізації MPI для мови FORTRAN-77 більшість MPI-процедур є підпрограмами типу SUBROTINE (викликаються за допомогою оператора CALL), а код помилки повертають через додатковий останній параметр процедури. Декілька процедур, оформлених у вигляді функцій, код помилки не повертають. Не вимагається строгого дотримання регістра символів в іменах підпрограм і іменованих констант. Масиви за замовчуванням індексуються з 1. Об'єкти MPI, які в мові С є структурами, в мові FORTRAN подаються масивами цілого типу. Визначення всіх іменованих констант і визначення типів виконується підключенням файлу **mpif.h.**

В даний час на кафедрі комп'ютерних технологій Дніпропетровського національного університету реалізовано новий інтерфейс MPI для стандарту FORTRAN-2003. Цей стандарт значно поліпшує взаємодію Фортран-програми з підпрограмами, написаними на мові С. При цьому зникає необхідність мати

додаткові бібліотеки, як це було в старій реалізації. Цей інтерфейс перевірено на компіляторі Фортана G95. В даній реалізації всі інтерфейси процедур MPI знаходяться в модулі MPI (файл **MPI.f03**). Всі процедури MPI реалізовані у вигляді функцій і мають точно такий же вигляд, як і в мові С. Далі всі прототипи функцій будуть описуватись на мові С, а приклади програм будуть наведені на мові Фортран.

У табл. 2 наведена відповідність зумовлених у MPI типів стандартним типам мови FORTRAN. Слід відзначити, що в табл. 2 перерахований обов'язковий мінімум підтримуваних стандартних типів, проте, якщо в базовій системі представлені й інші типи, то їх підтримку здійснюватиме і MPI. Типи MPI_BYTE і MPI_PACKED застосовуються для передачі двійкової інформації без якого-небудь перетворення. Крім того, програмісту надаються засоби створення власних типів на базі стандартних (розділ 5.1).

Вивчення MPI почнемо з розгляду базового набору з 6 функцій, які створюють мінімально повний набір, достатній для написання простих програм. Під час опису параметрів процедур символами **IN** указуватимемо вхідні параметри процедур, символами **OUT** вихідні, а **INOUT** - вхідні параметри, що модифікуються процедурою.

3.2. Базові функції MPI

Будь-яка прикладна MPI-програма (застосування) повинна починатися з виклику функції ініціалізації MPI: `MPI_Init`. У результаті виконання цієї функції створюється група процесів, в яку поміщаються всі процеси, і створюється область зв'язку, що описується комунікатором `MPI_COMM_WORLD`. Ця область зв'язку об'єднує всі процеси-застосування. Процеси в групі впорядковані і пронумеровані від 0 до groupsize-1, де groupsize дорівнює числу процесів у групі. Крім того, створюється зумовлений комунікатор MPI_COMM_SELF, що описує свою область зв'язку для кожного окремого процесу.

Синтаксис функції ініціалізації `MPI_Init` значно відрізняється в мовах С і FORTRAN:

Прототип функції MPI_Init на мові С:

```
int MPI_Init(int *argc, char ***argv)
```

На мові FORTRAN це можна використати таким чином:

```
if( MPI_Init(0,0) /= MPI_SUCCESS) stop 'MPI error'
```

Функція завершення MPI-програм MPI_Finalize

С:

```
int MPI_Finalize(void)
```

FORTRAN:

```
Ierr = MPI_FINALIZE()
```

Функція закриває всі MPI-процеси і ліквідує всі області зв'язку.

Далі всі прототипи функцій MPI будуть наводитись тільки мовою С.

Функція визначення числа процесів у області зв'язку `MPI_Comm_size`

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

IN comm - комунікатор;

OUT size - число процесів у області зв'язку комунікатора comm.

Функція повертає кількість процесів у області зв'язку комунікатора comm.

До створення явним чином груп і пов'язаних з ними комунікаторів (розділ 6) єдино можливими значеннями параметра COMM є MPI_COMM_WORLD і MPI_COMM_SELF, які створюються автоматично у процесі ініціалізації MPI. Підпрограма є локальною.

Функція визначення номера процесу MPI_Comm_rank

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

IN comm - комунікатор;

OUT Rank - номер процесу, який викликав функцію.

Функція повертає номер процесу, що викликав цю функцію. Номери процесів лежать у діапазоні **`0..size-1`** (значення **`size`** може бути визначене за допомогою попередньої функції). Підпрограма є локальною.

У мінімальний набір слід включити також дві функції передачі й прийому повідомлень.

Функція передачі повідомлення MPI_Send

```
int MPI_Send(void* buf, int count MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

IN buf - адреса почала розташування даних, що пересилаються;

IN count - число елементів, що пересилаються;

Ndatatype - тип надісланих елементів;

IN dest - номер процесу-одержувача в групі, пов'язаній з комунікатором comm;

IN tag - ідентифікатор повідомлення;

IN comm - Комунікатор області зв'язку.

Функція виконує посилку count елементів типу datatype повідомлення з ідентифікатором tag процесу dest у області зв'язку комунікатора comm. Змінна buf - це, як правило, масив або скаляр. В останньому випадку значення count = 1.

Функція прийому повідомлення MPI_Recv

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status *status)
```

OUT buf - адреса почала розташування повідомлення, що приймається;

IN count - максимальне число елементів, що приймаються;

IN datatype - тип елементів повідомлення, що приймається;

IN source - номер процесу-відправника;

IN tag - ідентифікатор повідомлення;

IN comm - комунікатор області зв'язку;

OUT status - атрибути прийнятого повідомлення.

Функція виконує прийом count елементів типу datatype повідомлення з ідентифікатором tag від процесу source у області зв'язку комунікатора comm.

Детальніше операції обміну повідомленнями будуть описані в наступному розділі, а на закінчення цього розділу розглянемо функцію, яка не входить в окреслений нами мінімум, але важлива для розробки ефективних програм. Йдеться про функцію отримання відліку часу - таймер. З одного боку, такі функції є у складі всіх операційних систем, а з іншого – існує цілковите свавілля в їх реалізації. Досвід роботи з різними операційними системами показує, що в разі перенесення програм з однієї платформи на іншу перше (а іноді і єдине), що доводиться переробляти, - це звернення до функцій обліку часу. Тому розробники MPI, добиваючись повної незалежності застосувань від операційного середовища, визначили і свої функції відліку часу.

Функція відліку часу (таймер) MPI_Wtime

```
double MPI_Wtime(void)
```

Функція повертає астрономічний час (у секундах), що пройшов з деякого моменту у минулому (точки відліку). Гарантується, що ця точка відліку не буде змінена протягом життя процесу. Для хронометражу ділянки програми виклик функції робиться на початку й кінці ділянки й визначається різниця між показаннями таймера. Це проілюстровано в наступному фрагменті програми на мові С:

```
{
   double start, end;
   start = MPI_Wtime();
  ... ділянка ..., що хронометрується
   end = MPI_Wtime();
  printf("Виконання зайняло %f секунд\n", end-start);
 }
```

Функція MPI_Wtick, що має такий самий синтаксис, повертає розрізнення таймера (мінімальне значення кванта часу).

3.3. Комунікаційні операції типу точка-точка

До операцій цього типу належать дві представлені в попередньому розділі комунікаційні процедури. У комунікаційних операціях типу точка-точка завжди беруть участь не більше двох процесів: той, що передає, і той, що приймає. У MPI є безліч функцій, що реалізовують такий тип обмінів. Різноманіття пояснюється

можливістю організації таких обмінів безліччю способів. Описані в попередньому розділі функції реалізують стандартний режим з блокуванням.

Блокувальні функції дають можливість виходу з них тільки після повного закінчення операції, тобто процес блокується, поки операція не буде завершена. Для функції посилки повідомлення це означає, що всі дані, що пересилаються, поміщені в буфер (для різних реалізацій MPI це може бути або якийсь проміжний системний буфер, або безпосередньо буфер одержувача). Для функції прийому повідомлення блокується виконання інших операцій, поки всі дані з буфера не будуть поміщені в адресний простір приймального процесу.

Неблокувальні функції передбачають поєднання операцій обміну з іншими операціями, тому неблокувальні функції передачі і прийому по суті є функціями ініціалізації відповідних операцій. Для опитування завершеності операції (і завершення) вводяться додаткові функції.

Як для блокувальних, так і неблокувальних операцій MPI підтримує чотири режими виконання. Ці режими стосуються тільки функцій передачі даних, тому для блокувальних і неблокувальних операцій є по чотири функції посилки повідомлення. У табл. 3 перераховані імена базових комунікаційних функцій типу точка-точка, наявних у бібліотеці MPI.

Таблиця 3. Список комунікаційних функцій типу точка-точка

Режими виконання	**З блокуванням**	**Без блокування**
Стандартна посилка	`MPI_Send`	`MPI_Isend`
Синхронна посилка	`MPI_Ssend`	`MPI_Issend`
Посилка, що буферизує	`MPI_Bsend`	`MPI_Ibsend`
Узгоджена посилка	`MPI_Rsend`	`MPI_Irsend`
Прийом інформації	`MPI_Recv`	`MPI_Irecv`

Табл. 3 демонструє принцип формування імен функцій. До імен базових функцій Send/Recv додаються різні префікси.

Префікс S (synchronous) - означає синхронний режим передачі даних. Операція передачі даних закінчується тільки тоді, коли закінчується прийом даних. Функція нелокальна.

Префікс B (buffered) - означає режим передачі даних, що буферизує. В адресному просторі процесу, що передає дані, за допомогою спеціальної функції створюється буфер обміну, який використовується в операціях обміну. Операція посилки закінчується, коли дані поміщені в цей буфер. Функція має локальний характер.

Префікс R (ready) - узгоджений або підготовлений режим передачі даних. Операція передачі даних починається тільки тоді, коли приймальний процесор

виставив ознаку готовності до прийому даних, тобто ініціював операцію прийому. Функція нелокальна.

Префікс I (immediate) - означає неблокувальні операції.

Всі функції передачі і прийому повідомлень можуть використовуватися в будь-якій комбінації один з одним. Функції передачі, що знаходяться в одному стовпці, мають абсолютно однаковий синтаксис і відрізняються тільки внутрішньою реалізацією. Тому надалі розглядатимемо тільки стандартний режим, який в обов'язковому порядку підтримують всі реалізації MPI.

3.4. Блокувальні комунікаційні операції

Синтаксис базових комунікаційних функцій MPI_Send і MPI_Recv був наведений у розділі 3.2, тому тут ми розглянемо тільки семантику цих операцій.

У стандартному режимі виконання операція обміну включає три етапи:

1. Сторона, що передає, формує пакет повідомлення, в який крім інформації, що передається, упаковуються адреса відправника (source), адреса одержувача (dest), ідентифікатор повідомлення (tag) і комунікатор (comm). Цей пакет передається відправником у системний буфер, і на цьому функція посилки повідомлення закінчується.
2. Повідомлення системними засобами передається адресатові.
3. Приймальний процесор витягує повідомлення з системного буфера, коли в нього з'явиться потреба в цих даних. Змістова частина повідомлення поміщається в адресний простір приймального процесу (параметр buf), а службова - в параметр status.

Оскільки операція виконується в асинхронному режимі, адресна частина прийнятого повідомлення складається з трьох полів:

- комунікатора (comm), оскільки кожен процес може одночасно входити в декілька областей зв'язку;
- номера відправника в цій області зв'язку (source);
- ідентифікатора повідомлення (tag), який застосовується для взаємної прив'язки конкретної пари операцій посилки і прийому повідомлень.

Параметр count (кількість елементів повідомлення, що приймаються) в процедурі прийому повідомлення повинен бути не менший, ніж довжина повідомлення, що приймається. При цьому реально прийматиметься стільки елементів, скільки знаходиться в буфері. Така реалізація операції читання пов'язана з тим, що MPI допускає використання розширених запитів:

- для ідентифікаторів повідомлень (MPI_ANY_TAG - читати повідомлення з будь-яким ідентифікатором);
- для адрес відправника (MPI_ANY_SOURCE - читати повідомлення від будь-якого відправника).

Не допускаються розширені запити для комунікаторів. Розширені запити можливі тільки в операціях читання. У цьому відбивається фундаментальна властивість механізму передачі повідомлень: асиметрія операцій передачі і прийому повідомлень, пов'язана з тим, що ініціатива в організації обміну належить стороні, яка передає.

Таким чином, після читання повідомлення деякі параметри можуть виявитися невідомими, а саме: число надісланих елементів, ідентифікатор повідомлення і адреса відправника. Цю інформацію можна отримати за допомогою параметра status. Змінні status повинні бути явно оголошені в MPI-програмі. У мові C status - це структура типу MPI_Status з трьома полями MPI_SOURCE, MPI_TAG, MPI_ERROR. У мові FORTRAN status - масив типу INTEGER розміру MPI_STATUS_SIZE. Константи MPI_SOURCE, MPI_TAG і MPI_ERROR визначають індекси елементів. Призначення полів змінною status подане в табл. 4.

Таблиця 4. Призначення полів змінною status

Поля status	**C**	**FORTRAN**
Процес-відправник	`status.MPI_SOURCE`	`status(MPI_SOURCE)`
Ідентифікатор повідомлення	`status.MPI_TAG`	`status(MPI_TAG)`
Код помилки	`status.MPI_ERROR`	`status(MPI_ERROR)`

Як видно із табл. 4, кількість елементів у змінну status не заноситься. Для визначення числа фактично отриманих елементів повідомлення необхідно використовувати спеціальну функцію MPI_Get_count:

```
int MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int
*count)
```

IN status - атрибути прийнятого повідомлення;
IN datatype - тип елементів прийнятого повідомлення;
OUT count - число отриманих елементів.

Підпрограма MPI_Get_count може бути викликана або після читання повідомлення (функціями MPI_Recv, MPI_Irecv), або після опитування факту надходження повідомлення (функціями MPI_Probe, MPI_Iprobe). Операція читання безповоротно знищує інформацію в буфері прийому. При цьому спроба прочитати повідомлення з параметром count меншим, ніж число елементів в буфері, призводить до втрати повідомлення.

Визначити параметри отриманого повідомлення без його читання можна за допомогою функції MPI_Probe.

```
int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status
*status)
```

IN source - номер процесу-відправника;
IN tag - ідентифікатор повідомлення;
IN comm - комунікатор;

OUT status - атрибути опитаного повідомлення.

Підпрограма MPI_Probe виконується з блокуванням, тому завершиться вона лише тоді, коли повідомлення з відповідним ідентифікатором і номером процесу-відправника буде доступне для отримання. Атрибути цього повідомлення повертаються в змінній status. Наступний за MPI_Probe виклик MPI_Recv з тими ж атрибутами повідомлення (номером процесу-відправника, ідентифікатором повідомлення і комунікатором) помістить у буфер прийому саме те повідомлення, наявність якого була опитана підпрограмою MPI_Probe.

У разі використання блокувального режиму передачі повідомлень існує потенційна небезпека виникнення тупикових ситуацій, в яких операції обміну даними блокують один одного. Наведемо приклад некоректної програми на мові Фортран, яка зависатиме за будь-яких умов.

```
Ierr = MPI_COMM_RANK(comm, rank)
IF (rank.EQ.0) THEN
   Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,1,tag,comm, status)
   Ierr=MPI_SEND(loc(sendbuf), count, MPI_REAL, 1, tag, comm)
ELSE IF (rank.EQ.1) THEN
   Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,0,tag,comm, status)
   Ierr=MPI_SEND(loc(sendbuf), count, MPI_REAL, 0, tag, comm)
END IF
```

У даному прикладі обидва процеси (0-й і 1-й) входять у режим взаємного очікування повідомлення один від одного. Такі тупикові ситуації виникатимуть завжди під час утворення циклічних ланцюжків блокувальних операцій читання.

Приведемо варіант правильної програми.

```
Ierr = MPI_COMM_RANK(comm, rank)
IF (rank.EQ.0) THEN
   Ierr=MPI_SEND(loc(sendbuf),count, MPI_REAL, 1, tag, comm)
   Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,1,tag,comm, status)
ELSE IF (rank.EQ.1) THEN
   Ierr=MPI_RECV(loc(recvbuf),count,MPI_REAL,0,tag,comm,status)
   Ierr=MPI_SEND(loc(sendbuf),count,MPI_REAL,0,tag,comm)
END IF
```

Інші комбінації операцій SEND/RECV можуть працювати або не працювати залежно від реалізації MPI (обмін з буферизацією, чи ні).

У ситуаціях, коли потрібно виконати взаємний обмін даними між процесами, безпечніше використовувати суміщену операцію MPI_Sendrecv.

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype
sendtype, int dest, int sendtag, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int source, MPI_Datatypea recvtag,
MPI_Comm comm, MPI_Status *status)
```

IN sendbuf - адреса початку надісланого повідомлення;
IN sendcount - число надісланих елементів;
IN sendtype - тип надісланих елементів;
IN dest - номер процесу, який отримує повідомлення;
IN sendtag - ідентифікатор надісланого повідомлення;

OUT recvbuf - адреса початку повідомлення, що приймається;
IN recvcount - максимальне число елементів, що приймаються;
IN recvtype - тип елементів повідомлення, що приймаються;
IN source - номер процесу-відправника;
IN recvtag - ідентифікатор повідомлення, що приймається;
IN comm - комунікатор області зв'язку;
OUT status - атрибути прийнятого повідомлення.

Функція MPI_Sendrecv суміщає виконання операцій передачі і прийому. Обидві операції використовують один і той же комунікатор, але ідентифікатори повідомлень можуть відрізнятися. Розміщення в адресному просторі процесу даних, що приймаються і передаються, не повинно перетинатися. Дані, що пересилаються, можуть бути різного типу й мати різну довжину.

У тих випадках, коли необхідний обмін даними одного типу із заміщенням надісланих даних тими, що приймаються, зручніше користуватися функцією MPI_Sendrecv_replace.

```
MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype
datatype, int dest, int sendtag, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

INOUT buf - адреса початку повідомлення, що приймається;
IN count - число елементів, що передаються;
IN datatype - тип елементів, що передаються;
IN dest - номер процесу-одержувача;
IN sendtag - ідентифікатор надісланого повідомлення;
IN source - номер процесу-відправника;
IN recvtag - ідентифікатор повідомлення, що приймається;
IN comm - комунікатор області зв'язку;
OUT status - атрибути прийнятого повідомлення.

У даній операції надіслані дані з масиву buf заміщаються даними, що приймаються. Як адресати source і dest в операціях пересилання даних можна використовувати спеціальну адресу MPI_PROC_NULL. Комунікаційні операції з такою адресою нічого не роблять. Застосування цієї адреси буває зручним замість використання логічних конструкцій для аналізу умов посилати/читати повідомлення чи ні. Цей прийом буде використаний нами далі в одному з прикладів, а саме в програмі розв'язування рівняння Лапласа методом Якобі.

3.5. Неблокувальні комунікаційні операції

Використання неблокувальних комунікаційних операцій підвищує безпеку з огляду на виникнення тупикових ситуацій, а також може збільшити швидкість роботи програми за рахунок поєднання виконання обчислювальних і комунікаційних операцій. Для цього комунікаційні операції розділяються на дві стадії: ініціацію операції і перевірку завершення операції.

Неблокувальні операції використовують спеціальний прихований (opaque) об'єкт "запит обміну" (request) для зв'язку між функціями обміну і функціями опитування їх завершення. Для прикладних програм доступ до цього об'єкта

можливий тільки через виклики MPI-функцій. Якщо операція обміну завершена, підпрограма перевірки знімає "запит обміну", встановлюючи його в значення MPI_REQUEST_NULL. Зняти запит без очікування завершення операції можна підпрограмою MPI_Request_free.

Функція передачі повідомлення без блокування MPI_Isend

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int
dest, int tag, MPI_Comm comm, MPI_Request *request)
```

IN buf - адреса даних, що передаються;
IN count - число надісланих елементів;
IN datatype - тип надісланих елементів;
IN dest - номер процесу-одержувача;
IN tag - ідентифікатор повідомлення;
IN comm - комунікатор;
OUT request - "запит обміну".

Повернення з підпрограми відбувається негайно (immediate), без очікування закінчення передачі даних. Цим пояснюється префікс I в іменах функцій. Тому змінну buf повторно не можна використовувати доти, поки не буде погашений "запит обміну". Це можна зробити за допомогою підпрограм MPI_Wait або MPI_Test, передавши їм параметр request.

Функція прийому повідомлення без блокування MPI_Irecv

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Request *request)
```

OUT buf - адреса для даних, що приймаються;
IN count - максимальне число елементів, що приймаються;
IN datatype - тип елементів повідомлення, що приймається;
IN source - номер процесу-відправника;
IN tag - ідентифікатор повідомлення;
IN comm - комунікатор;
OUT request - "запит обміну".

Повернення з підпрограми відбувається негайно, без очікування закінчення прийому даних. Визначити момент закінчення прийому можна за допомогою підпрограм MPI_Wait або MPI_Test з відповідним параметром request.

Як і в блокувальних операціях часто виникає необхідність опитування параметрів отриманого повідомлення без його фактичного читання. Це робиться за допомогою функції MPI_Iprobe.

Неблокувальна функція читання параметрів отриманого повідомлення MPI_Iprobe

```
int MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag,
MPI_Status *status)
```

IN source - номер процесу-відправника;

IN tag - ідентифікатор повідомлення;
IN comm - комунікатор;
OUT flag - ознака завершеності операції;
OUT status - атрибути опитуваного повідомлення.

Якщо flag=true, то операція завершилася і в змінній status знаходяться атрибути цього повідомлення.

Скористатися результатом неблокувальної комунікаційної операції або повторно використовувати її параметри можна тільки після її повного завершення. Є два типи функцій завершення неблокувальних операцій:

1. Операції очікування завершення сім'ї WAIT блокують виконання процесу до повного завершення операції.
2. Операції перевірки завершення сім'ї TEST повертають значення TRUE або FALSE залежно від того, завершилася операція чи ні. Вони не блокують роботу процесу й корисні для попереднього визначення факту завершення операції.

Функція очікування завершення неблокувальної операції MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

INOUT request - "запит обміну";
OUT status - атрибути повідомлення.

Це нелокальна блокувальна операція. Повернення відбувається після завершення операції, пов'язаної із запитом request. У параметрі status повертається інформація про закінчену операцію.

Функція перевірки завершення неблокувальної операції MPI_Test

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status
*status)
```

INOUT request - "запит обміну";
OUT flag - ознака завершеності операції, що перевіряється;
OUT status - атрибути повідомлення, якщо операція завершилася.

Це локальна неблокувальна операція. Якщо пов'язана із запитом request операція завершена, повертається flag = true, а status містить інформацію про завершену операцію. Якщо операція, що перевіряється, не завершена, повертається flag = false, а значення status у цьому випадку не визначене.

Розглянемо приклад використання неблокувальних операцій і функції MPI_Wait.

```
Ierr = MPI_COMM_RANK(comm, rank)
IF (rank.EQ.0) THEN
    Ierr = MPI_ISEND(loc(a(1)),10,MPI_REAL,1,tag,comm,request)
!**** Виконання обчислень під час передачі повідомлення ****
   Ierr = MPI_WAIT(request, status)
ELSE
   Ierr = MPI_IRECV(loc(a(1)),15,MPI_REAL,0,tag,comm,request)
```

```
!**** Виконання обчислень під час прийому повідомлення  ****
   Ierr = MPI_WAIT(request, status)
END IF
```

Функція зняття запиту без очікування завершення неблокувальної операції MPI_Request_free

```
int MPI_Request_free(MPI_Request *request)
```

INOUT request - запит зв'язку.

Параметр request встановлюється в значення MPI_REQUEST_NULL. Пов'язана з цим запитом операція не обривається, проте перевірити її завершення за допомогою MPI_Wait або MPI_Test вже не можна. Для переривання комунікаційної операції слід застосувати функцію

`MPI_Cancel(MPI_Request *request).`

MPI має набір підпрограм для одночасної перевірки на завершення декількох операцій. Без докладного обговорення приведемо їх перелік (табл. 5).

Таблиця 5. Функції колективного завершення неблокувальних операцій

Виконувана перевірка	**Функції очікування (блокувальні)**	**Функції перевірки(неблокувальні)**
Завершилися всі операції	`MPI_Waitall`	`MPI_Testall`
Завершилася принаймні одна операція	`MPI_Waitany`	`MPI_Testany`
Завершилася одна із списку тих, що перевіряються	`MPI_Waitsome`	`MPI_Testsome`

Крім того, MPI дозволяє для неблокувальних операцій формувати цілі пакети запитів на комунікаційні операції MPI_Send_init і MPI_Recv_init, які запускаються функціями MPI_Start або MPI_Startall. Перевірка завершення виконання проводиться звичайними засобами за допомогою функцій сім'ї WAIT і TEST.

4. Колективні операції

Набір операцій типу точка-точка достатній для програмування будь-яких алгоритмів, проте MPI навряд чи б завоював таку популярність, якби обмежувався тільки цим набором комунікаційних операцій. Однією з найпривабливіших сторін MPI є наявність широкого набору колективних операцій, які беруть на себе виконання найбільш поширених дій, що часто зустрічаються під час програмування. Наприклад, часто виникає потреба розіслати деяку змінну або масив з одного процесора всім іншим. Кожен програміст може написати таку процедуру з використанням операцій Send/Recv, проте набагато зручніше скористатися колективною операцією MPI_Bcast. Причому гарантовано, що ця операція виконуватиметься набагато ефективніше, оскільки MPI-функція реалізована з використанням внутрішніх можливостей комунікаційного середовища.

4.1. Огляд колективних операцій

Головна відмінність колективних операцій від операцій типу точка-точка полягає в тому, що в них завжди беруть участь всі процеси, пов'язані з деяким комунікатором. Недотримання цього правила призводить або до аварійного завершення завдання, або до ще неприємнішого зависання завдання.

Набір колективних операцій включає:

1. Синхронізацію всіх процесів за допомогою бар'єрів (MPI_Barrier).
2. Колективні комунікаційні операції, до числа яких входять:

- розсилка інформації від одного процесу решті членів деякої області зв'язку (MPI_Bcast);
- збирання (gather) розподіленого по процесах масиву в один масив із збереженням його в адресному просторі виділеного (root) процесу (MPI_Gather, MPI_Gatherv);
- збирання (gather) розподіленого масиву в один масив з розсилкою його всім процесам деякої області зв'язку (MPI_Allgather, MPI_Allgatherv);
- розбиття масиву й розсилка його фрагментів (scatter) всім процесам області зв'язку (MPI_Scatter, MPI_Scatterv);
- суміщена операція Scatter/Gather (All-to-All), кожен процес ділить дані зі свого буфера передачі й розкидає фрагменти решті процесів, одночасно збираючи фрагменти, послані іншими процесами у свій буфер прийому (MPI_Alltoall, MPI_Alltoallv).

3. Глобальні обчислювальні операції (sum, min, max і ін.) над даними, розподіленими в адресних просторах різних процесів:

- із збереженням результату в адресному просторі одного процесу (MPI_Reduce);
- з розсилкою результату всім процесам (MPI_Allreduce);
- суміщена операція Reduce/Scatter (MPI_Reduce_scatter);
- префіксна редукція (MPI_Scan).

Всі комунікаційні підпрограми, за винятком MPI_Bcast, представлені двома варіантами:

- простий варіант, коли всі частини повідомлення, що передається, мають однакову довжину і займають суміжні області в адресному просторі процесів;
- "векторний" варіант, який надає ширші можливості з організації колективних комунікацій, знімаючи обмеження, властиві простому варіанту, як у частині довжин блоків, так і в частині розміщення даних в адресному просторі процесів. Векторні варіанти відрізняються додатковим символом "v" у кінці імені функції.

Відмінні особливості колективних операцій:

- Колективні комунікації не взаємодіють з комунікаціями типу точка-точка.
- Колективні комунікації виконуються в режимі з блокуванням. Повернення з підпрограми в кожному процесі відбувається тоді, коли його участь у колективній операції завершилася, проте це не означає, що інші процеси завершили операцію.
- Кількість отриманих даних повинна дорівнювати кількості надісланих даних.
- Типи елементів надісланих і отримуваних повідомлень повинні збігатись.
- Повідомлення не мають ідентифікаторів.

Примітка. У даному розділі часто використовуватимуться поняття *буфер обміну*, *буфер передачі*, *буфер прийому*. Не слід розуміти ці поняття в буквальному розумінні як якусь спеціальну область пам'яті, куди поміщаються дані перед викликом комунікаційної функції. Насправді це, як правило, звичайні масиви, які безпосередньо можуть брати участь у комунікаційних операціях. У викликах підпрограм передається адреса початку безперервної області пам'яті, яка братиме участь в операції обміну.

Вивчення колективних операцій почнемо з розгляду двох функцій, що стоять окремо: MPI_Barrier і MPI_Bcast.

Функція синхронізації процесів MPI_Barrier блокує роботу процесу, що викликав її, доти, поки всі інші процеси групи також не викличуть цю функцію. Завершення роботи цієї функції можливе тільки всіма процесами одночасно (всі процеси "долають бар'єр" одночасно).

```
int MPI_Barrier(MPI_Comm comm )
```

IN comm - комунікатор.

Синхронізація за допомогою бар'єрів застосовується, наприклад, для завершення всіма процесами деякого етапу розв’язання задачі, результати якого використовуватимуться на наступному етапі. Використання бар'єра гарантує, що жоден з процесів не приступить завчасно до виконання наступного етапу, поки результат роботи попереднього не буде остаточно сформований. Неявну синхронізацію процесів виконує будь-яка колективна функція.

Широкомовна розсилка даних виконується за допомогою функції MPI_Bcast. Процес з номером root розсилає повідомлення зі свого буфера передачі всім процесам області зв'язку комунікатора comm.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
int root, MPI_Comm comm )
```

INOUT buffer - адреса даних, що розсилаються;
IN count - число надісланих елементів;
IN datatype - тип надісланих елементів;
IN root - номер процесу-відправника;
IN comm - комунікатор.

Після завершення підпрограми кожен процес в області зв'язку комунікатора comm, включаючи й самого відправника, отримає копію повідомлення від процесу-відправника root. На рис. 4.1 подана графічна інтерпретація операції Bcast.

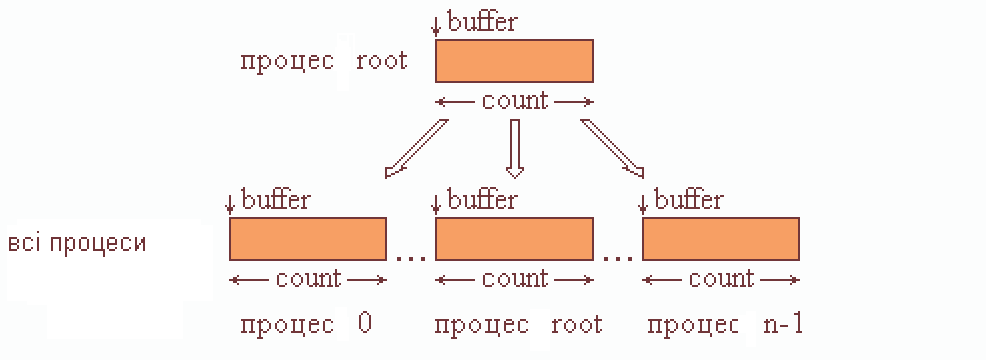

Рис 4.1. Графічна інтерпретація операції Bcast

Приклад використання функції MPI_Bcast.

```
IF ( MYID .EQ. 0 ) THEN
   PRINT *, 'ВВЕДИТЕ ПАРАМЕТР N : '
   READ (*,*) N
END IF
Ierr = MPI_BCAST(loc(N), 1, MPI_INT, 0, MPI_COMM_WORLD)
```

4.2. Функції збирання блоків даних від всіх процесів групи

Сім’я функцій збирання блоків даних від всіх процесів групи складається з чотирьох підпрограм: MPI_Gather, MPI_Allgather, MPI_Gatherv, MPI_Allgatherv. Кожна з вказаних підпрограм розширює функціональні можливості попередніх.

Функція MPI_Gather проводить збирання блоків даних, що надсилаються всіма процесами групи, в один масив процесу з номером root. Довжина блоків передбачається однаковою. Об'єднання відбувається в порядку збільшення номерів процесів-відправників. Тобто дані, надіслані процесом і зі свого буфера sendbuf, поміщаються в i-ту порцію буфера recvbuf процесу root. Довжина масиву, в який збираються дані, повинна бути достатньою для їх розміщення.

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)
```

IN sendbuf - адреса початку розміщення надісланих даних;
IN sendcount - число надісланих елементів;
IN sendtype - тип надісланих елементів;
OUT recvbuf - адреса буфера прийому (застосовується тільки в процесі-одержувачі root);
IN recvcount - число елементів, що отримуються від кожного процесу (застосовується тільки в процесі-одержувачі root);
IN recvtype - тип отримуваних елементів;
IN root - номер процесу, який отримає дані;
IN comm - комунікатор.

Тип надісланих елементів sendtype повинен збігатися з типом recvtype отримуваних елементів, а число sendcount повинне дорівнювати числу recvcount. Тобто recvcount у виклику з процесу root - це число елементів, зібраних від кожного процесу, а не загальна кількість зібраних елементів. Графічна інтерпретація операції Gather подана на рис. 4.2.

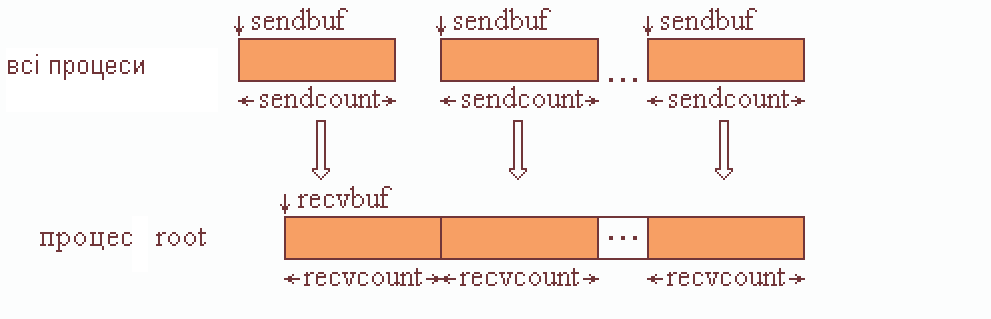

Рис. 4.2. Графічна інтерпретація операції Gather

Приклад програми з використанням функції MPI_Gather.

```
MPI_Comm comm;
int array[100];
int root, *rbuf;
. . .
MPI_Comm_size(comm, &gsize);
rbuf = (int *) malloc( gsize * 100 * sizeof(int));
MPI_Gather(array, 100, MPI_INT, rbuf,100, MPI_INT, root, comm);
```

Функція MPI_Allgather виконується так само, як MPI_Gather, але одержувачами є всі процеси групи. Дані, надіслані процесом і зі свого буфера

sendbuf, поміщаються в i-ту порцію буфера recvbuf кожного процесу. Після завершення операції вміст буферів прийому recvbuf у всіх процесів однаковий.

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)
```

IN sendbuf - адреса буфера посилки;
IN sendcount - число надісланих елементів;
IN sendtype - тип надісланих елементів;
OUT recvbuf - адреса буфера прийому;
IN recvcount - число елементів, що отримуються від кожного процесу;
IN recvtype - тип отримуваних елементів;
IN comm - комунікатор.

Графічна інтерпретація операції Allgather подана на рис 4.3. На цій схемі по вертикалі розташовані процеси групи, а по горизонталі - блоки даних.

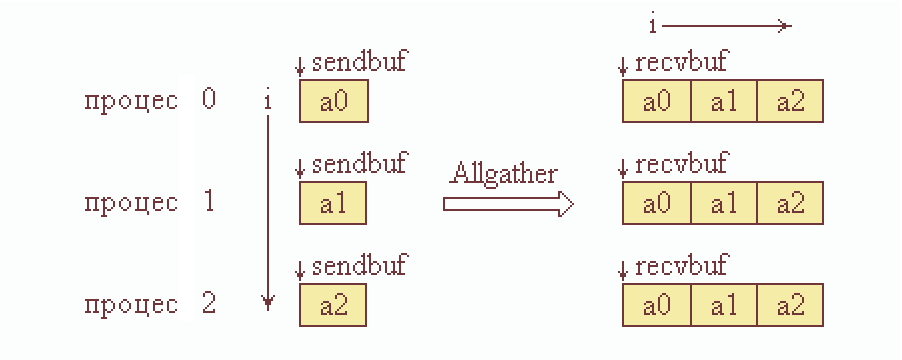

Рис 4.3. Графічна інтерпретація операції Allgather

Функція MPI_Gatherv дозволяє обирати блоки з різним числом елементів від кожного процесу, тому що кількість елементів, отриманих від кожного процесу, задається індивідуально за допомогою масиву recvcounts. Ця функція забезпечує також більшу гнучкість під час розміщення даних у процесі-отримувачі завдяки введенню як параметра масиву зміщень displs.

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* rbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype,int root, MPI_Comm comm)
```

IN sendbuf - адреса початку буфера передачі;
IN sendcount - кількість елементів, що відсилаються;
IN sendtype - тип елементів, що відсилаються;
OUT rbuf - адреса початку буфера прийому;
IN recvcounts - цілочисловий масив (розмір дорівнює числу процесів у групі), i-й елемент якого визначає число елементів, яке повинно бути отримане від процесу i;
IN displs - цілочисловий масив (розмір дорівнює числу процесів у групі), i-те значення визначає зміщення i-го блока даних відносно початку rbuf;
IN recvtype - тип отриманих елементів;
IN root - номер процесу-отримувача;
IN comm - комунікатор.

Повідомлення розміщуються в буфері процесу-отримувача root відповідно до номерів процесів, які їх посилають, а саме, дані, послані процесом i, розміщуються в адресному просторі процесу root, починаючи з адреси rbuf + displs[i]. Графічна інтерпретація операції Gatherv подана на рис. 4.4.

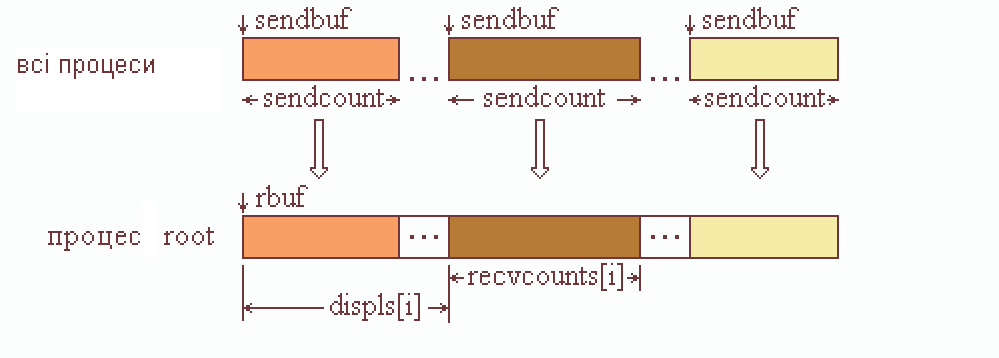

Рис. 4.4. Графічна інтерпретація операції Gatherv

Функція MPI_Allgatherv є аналогом функції MPI_Gatherv, але збирання даних здійснюється усіма процесами групи. Саме тому у списку параметрів відсутній параметр root.

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* rbuf, int *recvcounts,  int *displs,
MPI_Datatype recvtype, MPI_Comm comm)
```

IN sendbuf - адреса початку буфера передачі;
IN sendcount - кількість надісланих елементів;
IN sendtype - тип надісланих елементів;
OUT rbuf - адреса початку буфера прийому;
IN recvcounts - цілочисловий масив (розмір дорівнює числу процесів у групі), який містить число елементів, яке повинно бути отримане від кожного процесу;
IN displs - цілочисловий масив (розмір дорівнює числу процесів у групі), i-те значення визначає заміщення відносно початку rbuf i-го блока даних;
IN recvtype - тип отриманих елементів;
IN comm - комунікатор.

4.3. Функція розподілу блоків даних по всіх процесах групи

Сім'я функцій розподілу блоків даних по всіх процесах групи складається із двох підпрограм MPI_Scatter и MPI_Scaterv.

Функція MPI_Scatter розбиває повідомлення з буфера відсилки процесу root на однакові частини розміром sendcount та відсилає i-ту частину до буфера прийому процесу з номером i (в тому числі й самому собі). Процес root використовує обидва буфери (відсилки й прийому), саме тому в підпрограмі, що викликається, всі параметри є дійсними. Інші процеси групи з комунікатором comm є тільки одержувачами, саме тому для них параметри, які вказують буфер відсилки, не є дійсними .

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype
recvtype,int root, MPI_Comm comm)
```

IN sendbuf - адреса початку розміщення блоків розподілених даних (застосовується тільки в процесі-відправнику root);
IN sendcount - число елементів, які посилаються кожному процесу;
IN sendtype - тип елементів, які посилаються;
OUT recvbuf - адреса початку буфера прийому;
IN recvcount - число отримуваних елементів;
IN recvtype - тип отримуваних елементів;
IN root - номер процесу-відправника;
IN comm - комунікатор.

Тип надісланих елементів sendtype повинен збігатися з типом recvtype отримуваних елементів, а кількість надісланих елементів sendcount повинна дорівнювати кількості отримуваних recvcount. Слід звернути увагу, що значення sendcount у виклику з процесу root - це кількість надісланих кожному процесу елементів, а не загальна їх кількість. Операція Scatter є зворотною щодо Gather. На рис. 4.5 подана графічна інтерпретація операції Scatter.

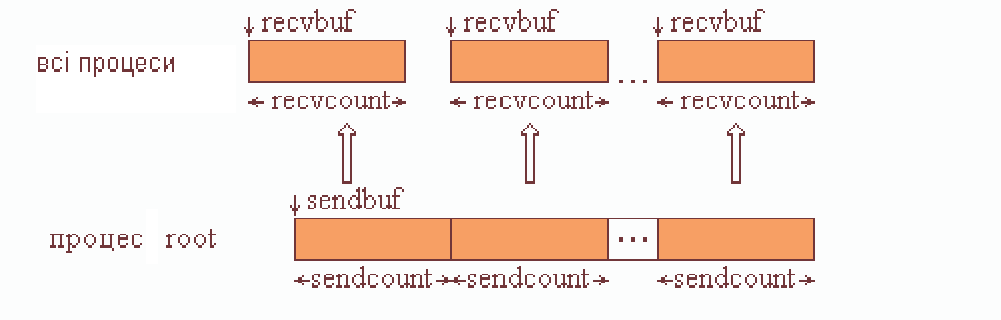

Рис. 4.5. Графічна інтерпретація операції Scatter

Приклад використання функції

```
 MPI_Scatter MPI_Comm comm;
 int  rbuf[100], gsize;
 int  root, *array;
. . . . . .
 MPI_Comm_size(comm, &gsize);
 array = (int *) malloc(gsize * 100 * sizeof(int));
. . . . . .
 MPI_Scatter(array,100,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

Функція MPI_Scaterv є векторним варіантом функції MPI_Scatter, що дозволяє відсилати кожному процесу різну кількість елементів. Початок розміщення елементів блока, що відправляються і-му процесу, задається в масиві зміщень displs, а кількість надісланих елементів - в масиві sendcounts. Ця функція є зворотною стосовно функції MPI_Gatherv.

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs,
MPI_Datatype sendtype, void* recvbuf, int
recvcount,MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN sendbuf - адреса початку буфера відсилки (застосовується тільки в процесі-відправнику root);
IN sendcounts - цілочисловий масив (розмір дорівнює числу процесів у групі), який містить кількість елементів, які надсилаються кожному процесу;
IN displs - цілочисловий масив (розмір дорівнює числу процесів у групі), i-те значення визначає зміщення відносно початку sendbuf для даних, що посилаються процесу i;
IN sendtype - тип елементів, що посилаються;
OUT recvbuf - адреса початку буфера прийому;
IN recvcount - кількість отриманих елементів;
IN recvtype - тип отриманих елементів;
IN root - номер процесу-відправника;
IN comm - комунікатор.

На рис. 4.6 подана графічна інтерпретація операції Scatterv.

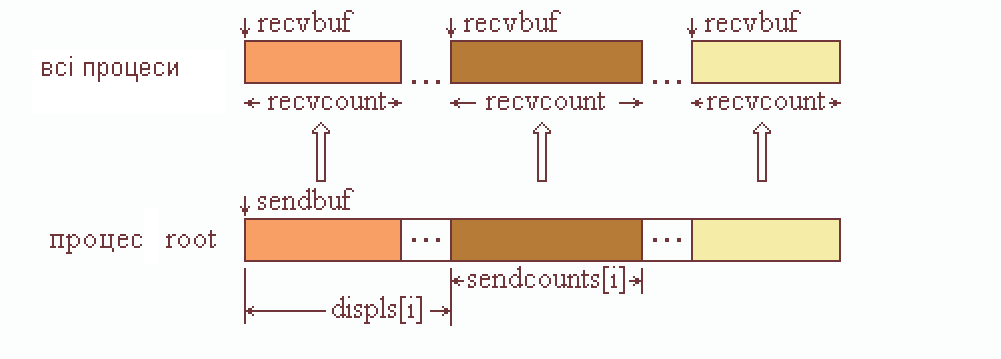

Рис. 4.6. Графічна інтерпретація операції Scatterv

4.4. Зміщені колективні операції

Функція MPI_Alltoall містить операції Scatter і Gather та є фактично розширенням операції Allgather, коли кожен процес відправляє різні дані різним одержувачам. Процес i відправляє j-й блок свого буфера sendbuf процесу j, який розміщує його до i-го блока свого буфера recvbuf. Кількість відправлених даних повинна дорівнювати кількості отриманих даних для кожної пари процесів.

```
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype,
MPI_Comm comm)
```

IN sendbuf - адреса початку буфера відправки;
IN sendcount - кількість відправлених елементів;
IN sendtype - тип відправлених елементів;
OUT recvbuf - адреса початку буфера прийому;
IN recvcount - кількість елементів, отриманих від кожного процесу;
IN recvtype - тип отриманих елементів;
IN comm - комунікатор.

Графічна інтерпретація операції Alltoall подана на рис. 4.7.

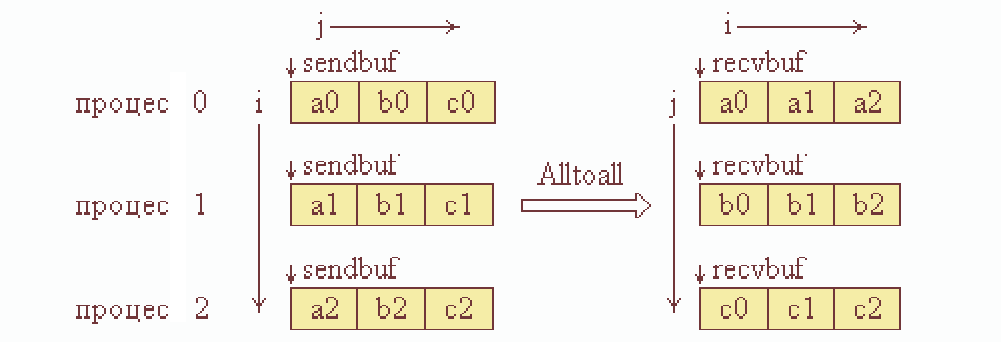

Рис. 4.7. Графічна інтерпретація операції Alltoall

Функція MPI_Alltoallv реалізує векторний варіант операції Alltoall, яка дозволяє передачу й прийом блоків різної довжини з більш гнучким розміщенням даних.

4.5. Глобальні розрахункові операції над розподіленими даними

У паралельному програмуванні математичні операції над блоками даних, розподілених по процесорах, називають ***глобальними операціями редукції***. У звичайному випадку ***операцією редукції*** називається операція, аргументом якої є вектор, а результатом - скалярна величина, отримана застосуванням деякої математичної операції до всіх компонентів вектора. У випадку, якщо компоненти вектора розміщені в адресних просторах процесів, які виконуються на різних процесорах, то говорять про ***глобальну (паралельну) редукцію***. Наприклад, нехай в адресному просторі всіх процесів деякої групи є копії змінної var (вони не обов'язково повинні мати одне й те саме значення), тоді застосовуючи до неї операції обчислення глобальної суми або, іншими словами, операцію редукції SUM, можна повернути одне значення, яке буде містити суму всіх локальних значень цієї змінної. Використання цих операцій є одним з основних засобів організації розподілених обчислень.

У MPI глобальні операції редукції подані декількома варіантами:

- зі збереженням результату в адресному просторі одного процесу (MPI_Reduce).
- зі збереженням результату в адресному просторі одного/усіх процесів (MPI_Allreduce).
- Префіксна операція редукції, що як результат операції повертає вектор, і-та компонента якого є результатом редукції перших і компонент розподіленого вектора (MPI_Scan).
- зміщена операція Reduce/Scatter (MPI_Reduce_scatter).

Функція MPI_Reduce виконується таким чином. Операція глобальної редукції, вказана параметром op, виконується над першими елементами вхідного буфера, і результат відправляється в перший елемент буфера прийому процесу root. Потім те саме виконується для другого елемента буфера й т.д.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```
IN sendbuf - адреса вхідного буфера;
OUT recvbuf - адреса буфера результатів (застосовується тільки в процесі-отримувачі root);
IN count - кількість елементів у вхідному буфері;
IN datatype - тип елементів у вхідному буфері;
IN op - операція, за якою виконується редукція;
IN root - номер процесу, який одержує результат операції;
IN comm - комунікатор.

На рис. 4.8 подана графічна інтерпретація операції Reduce. На даній схемі операція "+" означає будь-яку допустиму операцію редукції.

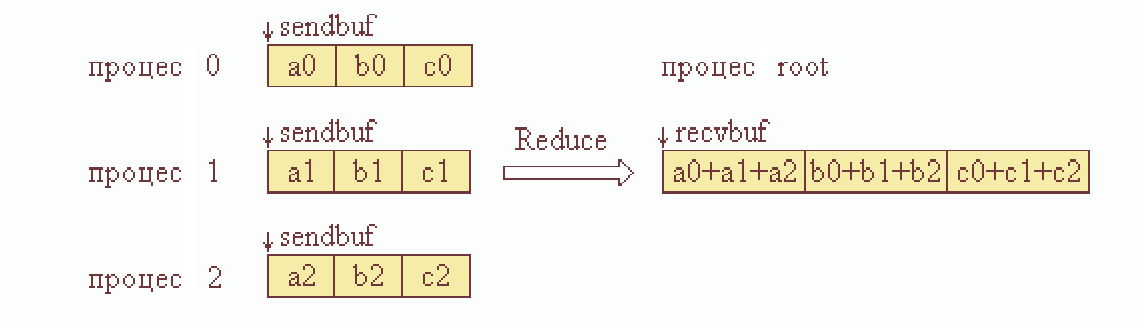

Рис. 4.8. Графічна інтерпретація операції Reduce

Як op можна використовувати одну з визначених операцій або операцію, створену користувачем. Всі перевизначені операції є асоціативними й комутативними. Створена користувачем операція повинна бути асоціативною. Порядок редукції визначається номерами процесів у групі. Тип datatype елементів повинен бути сумісним з операцією op. У табл. 6 представлено перелік наперед визначених операцій, які можуть бути використані у функціях редукції MPI.

Таблиця 6. Наперед визначені операції у функціях редукції MPI

Назва	Операція	Дозволені типи
MPI_MAX MPI_MIN	Максимум Мінімум	Integer, Floating point
MPI_SUM MPI_PROD	Сума Добуток	Integer, Floating point
MPI_LAND MPI_LOR MPI_LXOR	Логічне AND Логічне OR Логічне виключне OR	Integer, Logical
MPI_BAND MPI_BOR MPI_BXOR	Порозрядне AND Порозрядне OR Порозрядне виключне OR	Integer, Byte
MPI_MAXLOC MPI_MINLOC	Індекси максимального та мінімального значення	Спеціальні типи для цих функцій

Функція MPI_Allreduce зберігає результат редукції в адресному просторі всіх процесів, саме тому в списку параметрів функції відсутній ідентифікатор процесу root. Інші параметри такі ж, як і в попередньої функції.

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
      MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN sendbuf - адреса початку вхідного буфера;
OUT recvbuf - адреса початку буфера прийому;
IN count - кількість елементів у вхідному буфері;
IN datatype - тип елементів у вхідному буфері;
IN op - операція, за якою виконується редукція;
IN comm - комунікатор.

На рис. 4.9 подана графічна інтерпретація операції Allreduce.

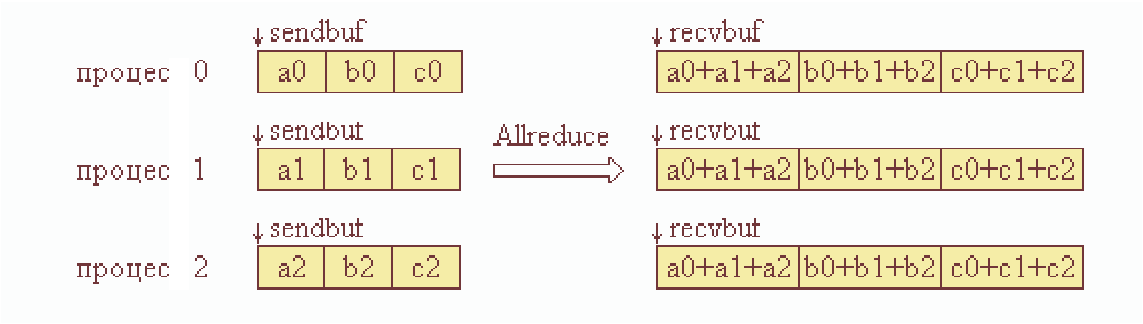

Рис. 4.9. Графічна інтерпретація операції Allreduce

Функція MPI_Reduce_scatter виконує операції редукції й розподілу результату по процесах.

```
MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int
*recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN sendbuf - адреса початку вхідного буфера;
OUT recvbuf - адреса початку буфера прийому;
IN revcount - масив, в якому задаються розміри блоків, які відправляються процесам;
IN datatype - тип елементів у вхідному буфері;
IN op - операція, за якою виконується редукція;
IN comm - комунікатор.

Функція MPI_Reduce_scatter відрізняється від MPI_Allreduce тим, що результат операції ділиться на частини відповідно до кількості процесів в групі, і-та частина відсилається і-му процесу в його буфер прийому. Довжина цих частин задає третій параметр, який є масивом. На рис. 4.10 подана графічна інтерпретація операції Reduce_scatter.

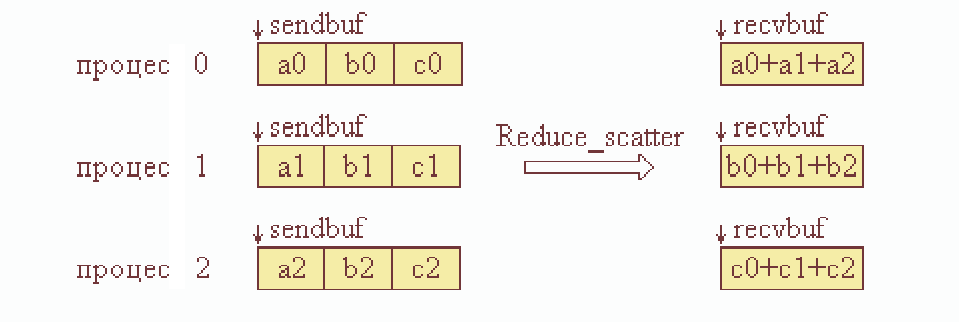

Рис. 4.10. Графічна інтерпретація операції Reduce_scatter

Функція MPI_Scan виконує префіксну редукцію. Параметри такі ж, як в MPI_Allreduce, але отримані кожним процесом результати відрізняються один від одного. Операція пересилає до буфера прийому i-го процесу редукцію значень із вхідних буферів процесів з номерами 0, ... і включно.

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

IN sendbuf - адреса початку вхідного буфера;
OUT recvbuf - адреса початку буфера прийому;
IN count - кількість елементів у вхідному буфері;
IN datatype - тип елементів у вхідному буфері;
IN op - операція, за якою виконується редукція;
IN comm - комунікатор.

На рис. 4.11 подана графічна інтерпретація операції Scan.

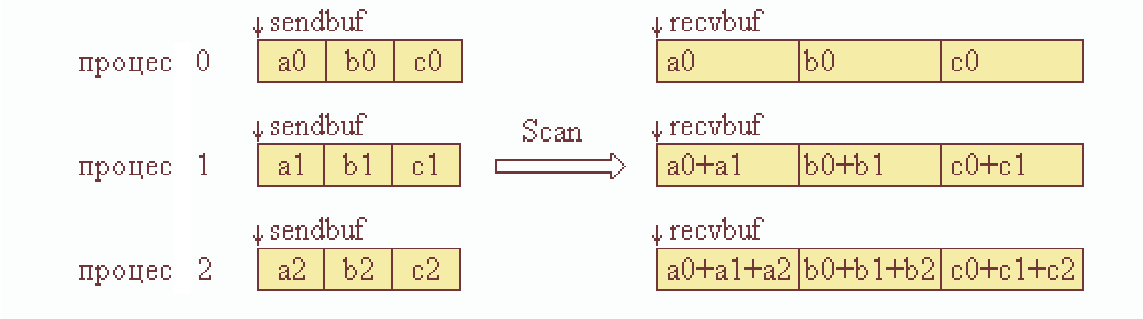

Рис. 4.11. Графічна інтерпретація операції Scan

5. Похідні типи даних і передача упакованих даних

Розглянуті раніше комунікаційні операції дозволяють відправляти й отримувати послідовність елементів одного типу, які займають суміжні області пам'яті. У процесі розробки паралельних програм іноді виникає потреба передавати дані різних типів (наприклад, структури) або дані, розміщені в несуміжних областях пам'яті (частини масивів, які не створюють неперервну послідовність елементів). MPI дає два механізми пересилання даних у згаданих випадках:

- шляхом створення похідних типів для використання в комунікаційних операціях замість наперед визначених типів MPI;
- пересилання упакованих даних (процес-відправник упаковує дані перед їх відправкою , а процес-одержувач розпаковує їх після отримання).

У більшості випадків обидва ці механізми дозволяють отримати бажаний результат, але в певних випадках більш вигідним може бути або один, або другий підхід.

5.1. Похідні типи даних

Похідні типи MPI не є в повному значенні типами даних, як це розуміється в мовах програмування. Вони не можуть використовуватися в жодних інших операціях, крім комунікаційних. Похідні типи MPI слід розуміти як ті, що описують розміщення в пам'яті елементів базових типів. Похідний тип MPI являє собою прихований (opaque) об'єкт, який специфікує дві речі: послідовність базових типів і послідовність зміщень. Послідовність таких пар визначається як *відображення (карта) типу*:

$\mathtt{Typemap} = \{(\mathtt{type}_0, \mathtt{disp}_0), \ldots, (\mathtt{type}_{n-1}, \mathtt{disp}_{n-1})\}$

Значення зміщень не обов'язково повинно бути не від'ємним, розрізнятися й упорядковуватися за збільшенням. Відображення типу разом з базовою адресою початку розміщення даних buf визначає комунікаційний буфер обміну. Цей буфер буде містити n елементів, а i-й елемент буде мати адресу buf+disp й базовий тип type. Стандартні типи MPI мають наперед визначене відображення типів. Наприклад, MPI_INT має відображення {(int,0)}.

Використання похідного типу у функціях обміну повідомленнями можна розглядати як трафарет, накладений на область пам'яті, яка містить повідомлення.

Стандартний сценарій визначення й використання похідних типів містить такі кроки:

- Похідний тип будується з наперед визначених типів MPI і раніше визначених похідних типів за допомогою спеціальних функцій-конструкторів MPI_Type_contiguous, MPI_Type_vector, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_hindexed, MPI_Type_struct.

- Новий похідний тип реєструється викликом функцій.
- MPI_Type_commit. Тільки після реєстрації новий похідний тип можна використовувати в комунікаційних підпрограмах під час створення інших типів. Наперед визначені типи MPI вважаються зареєстрованими.
- Коли похідний тип стає непотрібним, він знищується функцією MPI_Type_free.

Будь-який тип даних у MPI має дві характеристики: довжину й розмір, виражені в байтах:

- *Довжина типу* визначає, скільки байт змінна даного типу займає в пам'яті. Ця величина може бути підрахована як: адреса останньої чарунки даних - адреса першої чарунки даних + довжина останньої чарунки даних (запитується підпрограмою MPI_Type_extent).
- *Розмір типу* визначає реальну кількість байт, що передаються в комунікаційних операціях. Ця величина дорівнює сумі довжин всіх базових елементів визначеного типу (запитується підпрограмою MPI_Type_size).

Для звичайних типів довжина й розмір збігаються.

Функція MPI_Type_extent визначає довжину елемента деякого типу

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```
IN datatype - тип даних;
OUT size - розмір елемента заданого типу.

Функція MPI_Type_size визначає "чистий" розмір елемента деякого типу (з відніманням порожніх проміжків)

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```
IN datatype - тип даних;
OUT size - розмір елемента заданого типу.

Як зазначалось раніше, для створення похідних типів у MPI існує набір спеціальних функцій-конструкторів. Розглянемо їх у послідовності від простого до складного .

К*онструктор* типу MPI_Type_contiguous створює новий тип, елементи якого складаються з указаної кількості елементів базового типу, які займають суміжні області пам'яті.

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
MPI_Datatype *newtype)
```
IN count - кількість елементів базового типу;
IN oldtype - базовий тип даних;
OUT newtype - новий похідний тип даних.
Графічна інтерпретація роботи конструктора MPI_Type_contiguous наведена на рис. 5.1.

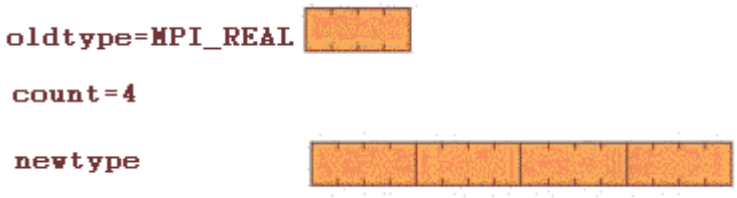

Рис. 5.1. Графічна інтерпретація роботи конструктора MPI_Type_contiguous

Конструктор типу MPI_Type_vector створює тип, елемент якого являє собою декілька рівновіддалених один від одного блоків з однакового числа суміжних елементів базового типу.

```
int MPI_Type_vector(int count, int blocklength, int stride,
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

IN count - кількість блоків;
IN blocklength - кількість елементів базового типу в кожному блоці;
IN stride - крок між початком сусідніх блоків, заміряний числом елементів базового типу;
IN oldtype - базовий тип даних;
OUT newtype - новий похідний тип даних.

Функція створює тип newtype, елемент якого складається з count блоків, кожен з яких містить однакове число blocklength елементів типу oldtype. Крок stride між початком блока і початком наступного блока скрізь однаковий і повинен бути кратним довжині базового типу. Графічна інтерпретація роботи конструктора MPI_Type_vector показана на рис. 5.2.

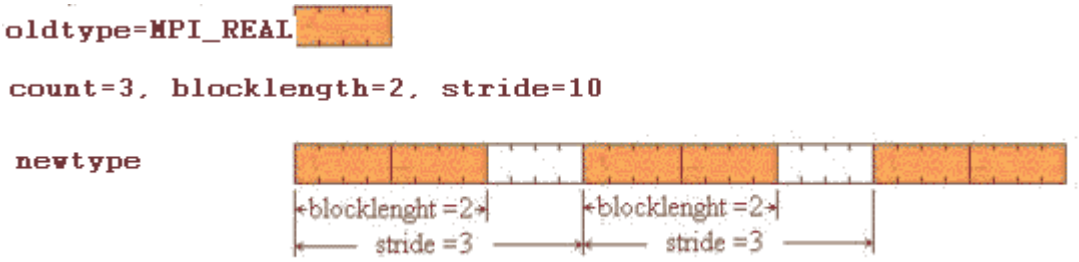

Рис. 5.2. Графічна інтерпретація роботи конструктора MPI_Type_vector

Конструктор типу MPI_Type_hvector розширює можливості конструктора MPI_Type_vector, дозволяючи задавати довільний крок між початками блоків у байтах.

```
int MPI_Type_hvector(int count, int blocklength, MPI_int
stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

IN count - кількість блоків;
IN blocklength - кількість елементів базового типу в кожному блоці;
IN stride - крок між початками сусідніх блоків у байтах;
IN oldtype - базовий тип даних;
OUT newtype - новий похідний тип даних.

Графічна інтерпретація роботи конструктора MPI_Type_hvector показана на рис. 5.3.

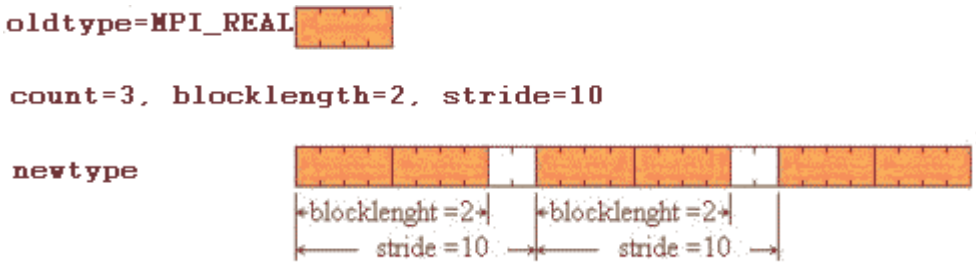

Рис. 5.3. Графічна інтерпретація роботи конструктора MPI_Type_hvector

Конструктор типу MPI_Type_indexed є більш загальним конструктором порівняно з MPI_Type_vector, оскільки елементи створеного типу складаються з довільних за довжиною блоків з довільним зміщенням блоків від початку розміщення елемента. Зміщення вимірюються в елементах старого типу.

```
int MPI_Type_indexed(int count, int *array_of_blocklengths, int
*array_of_displacements, MPI_Datatype oldtype,MPI_Datatype
*newtype)
```

IN count - кількість блоків;

IN array_of_blocklengths - масив, який містить кількість елементів базового типу в кожному i-му блоці;

IN array_of_displacements - масив зміщення кожного блока від початку розміщення елемента нового типу, зміщення вимірюються кількістю елементів базового типу;

IN oldtype - базовий тип даних;

OUT newtype - новий похідний тип даних.

Ця функція створює тип newtype, кожен елемент якого складається з count блоків, де i-й блок містить array_of_blocklengths[i] елементів базового типу і зміщення від початку розміщення елемента нового типу на array_of_displacements[i] елементів базового типу. Графічна інтерпретація роботи конструктора MPI_Type_indexeda показана на рис. 5.4.

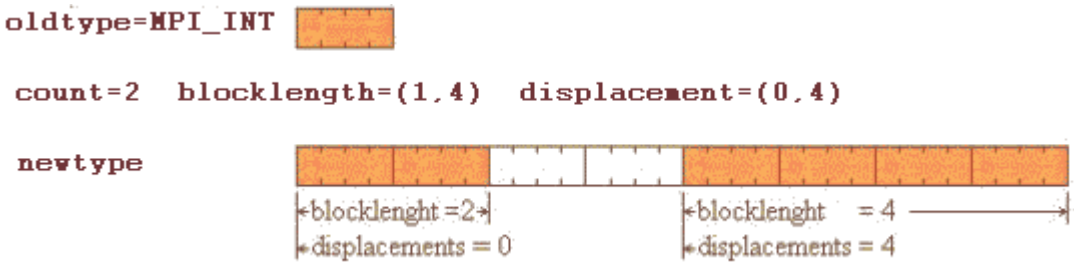

Рис. 5.4. Графічна інтерпретація роботи конструктора MPI_Type_indexed

Конструктор типу MPI_Type_hindexed подібний до конструктора MPI_Type_indexed, відмінність полягає в тому, що зміщення `array_of_displacements` вимірюються в байтах.

```
int MPI_Type_hindexed(int count, int *array_of_blocklengths,
 MPI_Aint *array_of_displacements, MPI_Datatype oldtype,
MPI_Datatype *newtype)
```

Елемент нового типу складається з count блоків, де i-й блок містить array_of_blocklengths[i] елементів старого типу і є зміщеним від початку

розміщення елемента нового типу на array_of_displacements[i] байт. Графічна інтерпретація роботи конструктора MPI_Type_hindexed показана на рис. 5.5.

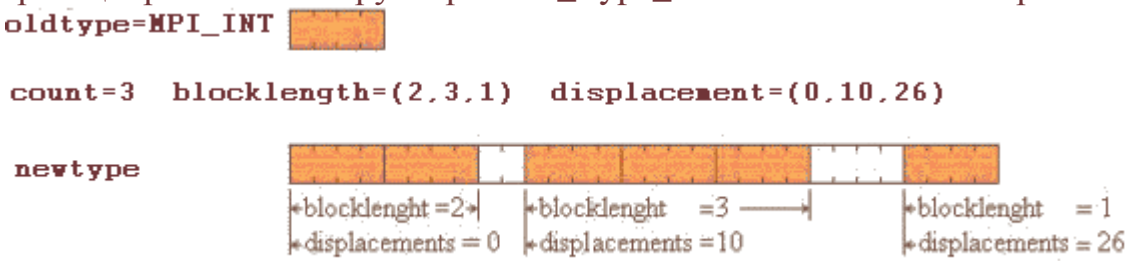

Рис. 5.5. Графічна інтерпретація роботи конструктора MPI_Type_hindexed

Конструктор типу MPI_Type_struct - найбільш загальний з усіх конструкторів типу. Створюваний ним тип є структурою, яка містить довільну кількість блоків, кожен з яких може містити довільну кількість елементів одного з базових типів і може бути зміщений на довільну кількість байтів від початку розміщення структури.

```
int MPI_Type_struct(int count, int *array_of_blocklengths,
MPI_Aint *array_of_displacements, MPI_Datatype
*array_of_types,MPI_Datatype *newtype)
```

IN count - кількість блоків;
IN array_of_blocklength - масив, який містить кількість елементів одного з базових типів у кожному блоці;
IN array_of_displacements - масив зміщень кожного блока від початку розміщення структури, зміщення вимірюються в байтах;
IN array_of_type - масив, який містить тип елементів в кожному і-му блоці;
OUT newtype - новий похідний тип даних.

Функція створює тип newtype, елемент якого складається з count блоків, де і-й блок містить array_of_blocklengths[i] елементів типу array_of_types[i]. Зміщення і-го блока від початку розміщення елемента нового типу вимірюється в байтах і задається в масиві array_of_displacements[i].

Графічна інтерпретація роботи конструктора MPI_Type_struct показана на рис. 5.6.

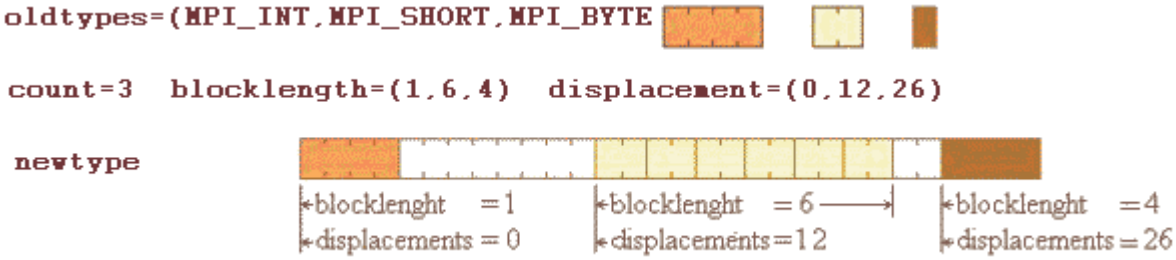

Рис. 5.6. Графічна інтерпретація роботи конструктора MPI_Type_struct

Функція MPI_Type_commit реєструє створений похідний тип. Тільки після реєстрації новий тип може використовуватись у комунікаційних операціях.

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

INOUT datatype - новий похідний тип даних.

Функція MPI_Type_free знищує описувач похідного типу.

```
int MPI_Type_free(MPI_Datatype *datatype)
```

INOUT datatype - похідний тип, що знищується.

Функція MPI_Type_free встановлює описувач типу у стан MPI_DATATYPE_NULL. Це не вплине на комунікаційні операції з цим типом даних, що виконуються в даний момент, та на похідні типи, які раніше були визначені через знищений тип.

Для визначення довжини повідомлення застосовуються дві функції: MPI_Get_count і MPI_Get_elements. Для повідомлень з простими типами вони повертають однакове число. Підпрограма MPI_Get_count повертає число елементів типу datatype, вказаного в операції. Якщо отримане не ціле число елементів, то вона поверне константу MPI_UNDEFINED (функція MPI_Get_count розглядалась у розділі 3.2, присвяченому комунікаційним операціям типу точка-точка).

Функція MPI_Get_elements повертає число елементів простих типів, що містяться у повідомленні.

```
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype,
int *count)
```

IN status - статус повідомлення;
IN datatype - тип елементів повідомлення;
OUT count - число елементів простих типів, що містяться в повідомленні.

5.2. Передача упакованих даних

Функція MPI_Pack упаковує елементи заздалегідь визначеного або похідного типу MPI, розміщуючи їх у вихідний буфер.

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype datatype,
void *outbuf, int outsize, int *position, MPI_Comm comm)
```

IN inbuf - адреса області пам'яті з елементами, які потрібно упакувати;
IN incount - число елементів, які потрібно упакувати;
IN datatype - тип елементів, які потрібно упакувати;
OUT outbuf - адреса початку вихідного буфера упакованих даних;
IN outsize - розмір вихідного буфера (у байтах);
INOUT position - поточна позиція у вихідному буфері (у байтах);
IN comm - комунікатор.

Функція MPI_Pack упаковує incount елементів типу datatype з області пам'яті з початковою адресою inbuf. Результат упакування розміщується у вихідний буфер з початковою адресою outbuf і розміром outsize байт. Параметр position вказує поточну позицію у байтах, починаючи з якої будуть розміщуватись упаковані дані. На виході з підпрограми значення position збільшується на число упакованих байт, вказуючи на перший вільний байт.

Функція MPI_Unpack вилучає задане число елементів деякого типу з побайтного подання елементів у вхідному масиві.

```
int MPI_Unpack(void* inbuf, int insize, int *position, void
*outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
```

IN inbuf - адреса початку вхідного буфера з упакованими даними;
IN insize - розмір вхідного буфера (у байтах);
INOUT position - поточна позиція у вхідному буфері (у байтах);
OUT outbuf - адреса початку області пам'яті для розміщення розпакованих елементів;
IN outcount - кількість елементів, що вилучаються;
IN datatype - тип елементів, що вилучаються;
IN comm - комунікатор.

Функція MPI_Unpack вилучає outcount елементів типу datatype з побайтного подання елементів у масиві inbuf, починаючи з адреси position. Після повернення з функції параметр position збільшується на розмір розпакованого повідомлення. Результат розпакування поміщається в область пам'яті з початковою адресою outbuf.

Для посилки елементів різного типу з декількох областей пам'яті їх слід попередньо запакувати в один масив, послідовно звертаючись до функції упаковки MPI_Pack. Під час першого виклику функції упаковки параметр position, як правило, встановлюється в 0, щоб упаковані дані розміщувались з початку буфера. Для неперервного заповнення буфера необхідно в кожному наступному викликові використовувати значення параметра position, отримане з попереднього виклику.

Упакований буфер пересилається будь-якими комунікаційними операціями з покажчиками типу MPI_PACKED і комунікатора comm, який використовувався під час звернення до функції MPI_Pack.

Отримане упаковане повідомлення розпаковується в різні масиви або змінні. Це реалізується послідовними викликами функції розпакування MPI_Unpack з указанням числа елементів, яке потрібно вилучити під час кожного виклику, і з передачею значення position, поверненого попереднім викликом. Під час першого виклику функції параметр position слід встановити в 0. У загальному випадку, під час першого звернення має бути встановлене те значення параметра position, яке було використане під час першого звернення до функції упаковки даних. Очевидно, що для правильного розпакування даних черговість вилучення даних має бути такою самою, як і під час упакування.

Функція MPI_Pack_size допомагає визначити розмір буфера, необхідний для упаковки деякої кількості даних типу datatype.

```
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm
comm, int *size)
```

IN incount - кількість елементів, що підлягають упаковці;
IN datatype - тип елементів, що підлягають упаковці;
IN comm - комунікатор;
OUT size - розмір повідомлення (в байтах) після його упаковки.

Перші три параметри функції MPI_Pack_size такі ж, як у функції MPI_Pack. Після звернення до функції параметр size буде містити розмір повідомлення (у байтах) після його упаковки.

Розглянемо приклад розсилки різнотипних даних з 0-го процесу з використанням функцій MPI_Pack і MPI_Unpack.

```
char buff[1000];
double x, y;
int position, a[2];
{
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if (myrank == 0){
  /* Упаковка даних*/
  position = 0;
  MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position,
           MPI_COMM_WORLD);
  MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position,
           MPI_COMM_WORLD);
  MPI_Pack(a, 2, MPI_INT, buff, 100, &position,
           MPI_COMM_WORLD);
 }
   /* Розсилання упакованого повідомлення */
 MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD);

   /* Розпакування повідомлення у всіх процесах */
  if (myrank != 0){
    position = 0;
    MPI_Unpack(buff, 1000, &position, &x, 1, MPI_DOUBLE,
             MPI_COMM_WORLD);
    MPI_Unpack(buff, 1000, &position, &y, 1, MPI_DOUBLE,
             MPI_COMM_WORLD);
    MPI_Unpack(buff, 1000, &position, a, 2, MPI_INT,
             MPI_COMM_WORLD);
  }
}
```

6. Робота з групами і комунікаторами

Часто в програмах виникає необхідність обмежити область комунікацій деяким набором процесів, які складають підмножину початкового набору. Для виконання будь-яких колективних операцій всередині цієї підмножини з них має бути сформована своя область зв'язку, яка описується своїм комунікатором. Для таких задач MPI підтримує два взаємозв’язані механізми. По-перше, існує набір функцій для роботи з групами процесів як впорядкованими множинами, і, по-друге, є набір функцій для роботи з комунікаторами для створення нових комунікаторів як описувачів нових областей зв'язку.

6.1. Основні поняття

Група являє собою впорядковану множину процесів. Кожний процес ідентифікується змінною цілого типу. Ідентифікатори процесів утворюють неперервний ряд, який починається з 0. У MPI вводиться спеціальний тип даних MPI_Group і набір функцій для роботи зі змінними й константами цього типу. Є дві наперед визначені групи:

- MPI_GROUP_EMPTY - група, що не містить жодного процесу;
- MPI_GROUP_NULL – значення, що повертається, коли група не може бути створена.

Створена група не може бути модифікована (розширена або зменшена), може бути лише створена нова група. Цікаво відзначити, що у процесі ініціалізації MPI не створюється група, відповідна до комунікатора MPI_COMM_WORLD. Вона має створюватись спеціальною функцією явним чином.

Комунікатор являє собою об'єкт з деяким набором атрибутів, а також правилами його створення, використання та знищення. Комунікатор описує деяку область зв'язку. Одній і тій самій області зв'язку може відповідати декілька комунікаторів, але навіть у цьому випадку вони не є тотожними і не можуть брати учать у взаємному обміні повідомленнями. Якщо дані посилаються через один комунікатор, процес-отримувач може отримати їх тільки через той самий комунікатор.

У MPI існує два типи комунікаторів:

- intracommunicator - описує область зв'язку деякої групи процесів;
- intercommunicator - слугує для зв'язку між процесами двох різних груп.

Тип комунікатора можна визначити за допомогою спеціальної функції **MPI_Comm_test_inter.**

```
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
```

IN comm - комунікатор;
OUT flag - повертає true, якщо comm - intercommunicator.

Функція повертає значення "істина", якщо комунікатор є inter комунікатором.

У процесі ініціалізації MPI створюються два наперед визначені комунікатори:

- MPI_COMM_WORLD - описує область зв'язку, що містить всі процеси;
- MPI_COMM_SELF - описує область зв'язку, яка складається з одного процесу.

6.2. Функції роботи з групами

Функція визначення числа процесів у групі MPI_Group_size

```
int MPI_Group_size(MPI_Group group, int *size)
```

IN group - група;
OUT size - кількість процесів у групі.
Функція повертає кількість процесів у групі. Якщо group = MPI_GROUP_EMPTY, тоді size = 0.

Функція визначення номера процесу у групі MPI_Group_rank

```
int MPI_Group_rank(MPI_Group group, int *rank)
```

IN group - група;
OUT rank - номер процесу у групі.
Функція MPI_Group_rank повертає номер у групі процесу, який викликав функцію. Якщо процес не є членом групи, то повертається значення MPI_UNDEFINED.

Функція встановлення відповідності між номерами процесів у двох групах MPI_Group_translate_ranks

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int
*ranks1, MPI_Group group2, int *ranks2)
```

IN group1 - група1;
IN n - кількість процесів, для яких встановлюється відповідність;
IN ranks1 - масив номерів процесів із 1-ї групи;
IN group2 - група2;
OUT ranks2 - номери тих же процесів у другій групі.
Функція визначає відносні номери одних і тих самих процесів у двох різних групах. Якщо процес у другій групі відсутній, то для нього встановлюється значення MPI_UNDEFINED.

Для створення нових груп у MPI існує 8 функцій. Група може бути створена або за допомогою комунікатора, або за допомогою операцій над множинами процесів інших груп.

Функція створення групи за допомогою комунікатора MPI_Comm_group

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

IN comm - комунікатор;
OUT group - група.

Функція створює групу group для множини процесів, які входять в область зв'язку комунікатора comm.

Наступні три функції мають однаковий синтаксис і створюють нову групу як результат операції над множиною процесів двох груп.

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup)
```

IN group1 - перша група;
IN group2 - друга група;
OUT newgroup - нова група.

Операції визначаються таким чином:

- Union - формує нову групу з елементів 1-ї групи і з елементів 2-ї групи, які не входять у 1-шу (об'єднання множин).
- Intersection - нова група формується з елементів 1-ї групи, які входять також і в 2-гу. Впорядковування як у 1-й групі (перетин множин).
- Difference - нову групу утворюють всі елементи 1-ї групи, які не входять у 2-гу. Впорядковування як у 1-й групі (доповнення множин).

Створена група може бути пустою, що еквівалентно MPI_GROUP_EMPTY.

Нові групи можуть бути створені за допомогою різних вибірок з існуючої групи. Наступні дві функції мають однаковий синтаксис, але є додатковими одна щодо одної.

```
int MPI_Group_incl(MPI_Group group, int n, int *ranks,
MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks,
MPI_Group *newgroup)
```

IN group - існуюча група;
IN n - число елементів у масиві ranks;
IN ranks - масив номерів процесів;
OUT newgroup - нова група.

Функція MPI_Group_incl створює нову групу, яка складається з процесів існуючої групи, перерахованих у масиві ranks. Процес з номером і в новій групі – це процес з номером ranks[i] в існуючій групі. Кожний елемент у масиві ranks

повинен мати коректний номер у групі group, і серед цих елементів не повинно бути збіжних.

Функція MPI_Group_excl створює нову групу з тих процесів group, які не перелічені в масиві ranks. Процеси впорядковуються як у групі group. Кожний елемент у масиві ranks повинен мати коректний номер у групі group, і серед них не повинно бути збіжних.

Дві наступні функції за змістом збігаються з попередніми, але використовують більш складне формування вибірки. Масив ranks замінюється двовимірним масивом ranges, який являє собою набір триплетів для задання діапазонів процесів.

```
int MPI_Group_range_incl(MPI_Group group, int n, int
ranges[][3], MPI_Group *newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int
ranges[][3], MPI_Group *newgroup)
```

Кожний триплет має вигляд: нижня границя, верхня границя, крок.

Знищення створених груп виконується функцією **MPI_Group_free.**

```
int MPI_Group_free(MPI_Group *group)
```

де INOUT group - група, що знищується.

6.3. Функції роботи з комунікаторами

У даному підрозділі розглядаються функції роботи з комунікаторами. Вони діляться на функції доступу до комунікаторів і функції створення комунікаторів. Функції доступу є локальними і не потребують комунікації, на відміну від функцій створення, які є колективними і можуть вимагати міжпроцесорних комунікацій.

Дві головні функції доступу до комунікатора, MPI_Comm_size - опитування кількості процесів в області зв'язку і MPI_Comm_rank - опитування ідентифікатора номера процесу в області зв'язку, були розглянуті серед базових функцій MPI.

Крім них є **функція порівняння двох комунікаторів MPI_Comm_compare.**

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,int *result)
```

IN comm1 - перший комунікатор;
IN comm2 - другий комунікатор;
OUT result - результат порівняння.

Можливі значення результату порівняння:

- MPI_IDENT - комунікатори ідентичні, представляють один і той самий об'єкт;
- MPI_CONGRUENT - комунікатори конгруентні, дві області зв'язку з одними й тими ж атрибутами групи;
- MPI_SIMILAR - комунікатори подібні, групи містять однакові процеси, але інше впорядкування;
- MPI_UNEQUAL - у всіх інших випадках.

Створення нового комунікатора можливе за допомогою однієї з трьох функцій: `MPI_Comm_dup, MPI_Comm_create, MPI_Comm_split.`

Функція дублювання комунікатора MPI_Comm_dup

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)
```

IN comm - комунікатор;
OUT newcomm - копія комунікатора.
Функція корисна для створення комунікаторів з новими атрибутами.

Функція створення комунікатора MPI_Comm_create

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm
*newcomm)
```

IN comm - батьківський комунікатор;
IN group - група, для якої створюється комунікатор;
OUT newcomm - новий комунікатор.

Ця функція створює комунікатор для групи group. Для процесів, які не є членами групи, повертається значення MPI_COMM_NULL. Функція повертає код помилки, якщо група group не є підгрупою батьківського комунікатора.

Функція розщеплення комунікатора MPI_Comm_split

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm
*newcomm)
```

IN comm - батьківський комунікатор;
IN color - ознака підгрупи;
IN key - керування упорядкуванням;
OUT newcomm - новий комунікатор.

Функція розщеплює групу, пов'язану з батьківським комунікатором, на непересічні підгрупи по одній на кожне значення ознаки підгрупи color. Значення color має бути невід'ємним. Кожна підгрупа містить процеси з одним і тим же значенням color. Параметр key керує упорядкуванням всередині нових груп: меншому значенню key відповідає менше значення ідентифікатора процесу. У

випадку рівності параметру key для декількох процесів упорядкування відбувається відповідно до порядку в батьківській групі.

Наведемо алгоритм розщеплення групи з восьми процесів на три підгрупи і його графічну інтерпретацію (рис. 6.1).

```
MPI_comm comm, newcomm;
int myid, color;
 . . . . . .
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```

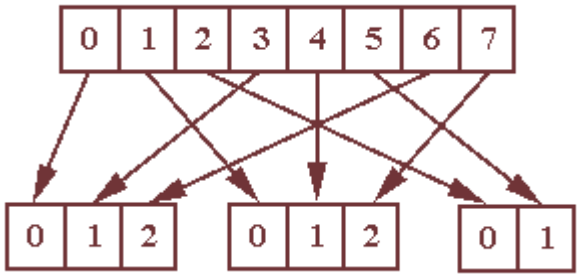

Рис. 6.1. Розбиття групи з восьми процесів на три підгрупи

У даному прикладі першу підгрупу утворили процеси, номери яких діляться на 3 без остачі, другу - для яких остача дорівнює 1 і третю - для яких остача дорівнює 2. Відзначимо, що після виконання функції MPI_Comm_split значення комунікатора newcomm у процесах різних підгруп будуть відрізнятись.

Функція знищення комунікатора MPI_Comm_free

```
int MPI_Comm_free(MPI_Comm *comm)
```

IN comm - комунікатор, який знищується.

Після виклику даної функції значення комунікатора буде MPI_COMM_NULL, але будь-які операції, що виконувались з даним комунікатором, закінчаться нормально. Дана функція може бути використана як для intra- так і для inter-комунікаторів.

Примітка: За межами даного посібника ми залишимо розгляд inter-комунікаторів і питання, пов'язані зі зміною або додаванням нових атрибутів комунікаторів.

7. Топологія процесів

Топологія процесів є одним із необов'язкових атрибутів комунікатора. Такий атрибут може бути присвоєний тільки intra-комунікатору. За замовчуванням передбачається лінійна топологія, в якій процеси пронумеровані в діапазоні від 0 до n-1, де n - число процесів у групі. Однак для багатьох задач лінійна топологія неадекватно відображає логіку комунікаційних зв'язків між процесами. MPI надає засоби для створення досить складних "віртуальних" топологій у вигляді графів, де вузли є процесами, а грані - каналами зв'язку між процесами. Так, слід розрізняти логічну топологію процесів, яку дозволяє формувати MPI, і фізичну топологію процесорів.

В ідеалі логічна топологія процесів повинна враховувати як алгоритм розв'язування задачі, так і фізичну топологію процесорів. Для дуже широкого кола задач найбільш адекватною топологією процесів є двовимірна або тривимірна сітка. Такі структури повністю визначаються числом змін і кількістю процесів вздовж кожного координатного напрямку, а також способом розкладки процесів на координатну сітку. У MPI, як правило, застосовується row-major нумерація процесів, тобто застосовується нумерація вздовж рядка. На рис. 7.1 зображена відповідність між нумераціями шести процесів в одновимірній і двовимірній (2x3) топологіях.

Row-major

0 (0,0)	1 (0,1)	2 (0,2)
3 (1,0)	4 (1,1)	5 (1,2)

Column-major

0 (0,0)	2 (0,1)	4 (0,2)
1 (1,0)	3 (1,1)	5 (1,2)

Рис. 7.1. Співвідношення між ідентифікатором процесу (верхнє число) і координатами у двовимірній сітці 2x3 (нижня пара чисел)

7.1. Декартова топологія

Узагальненням лінійної і матричної топології на довільне число вимірів є декартова топологія. Для створення комунікатора з декартовою топологією застосовується функція MPI_Cart_create. За допомогою цієї функції можна створювати топології з довільним числом вимірів, причому за кожним виміром окремо можна накладати періодичні граничні умови. Таким чином, для одновимірної топології ми можемо отримати або лінійну структуру, або кільце залежно від того, які граничні умови будуть накладені. Для двовимірної топології відповідно або прямокутник, або циліндр, або тор. Відзначимо, що немає потреби

в спеціальній підтримці гіперкубової структури, оскільки вона являє собою n-вимірний тор з двома процесами вздовж кожного координатного напряму.

Функція створення комунікатора з декартовою топологією

```
int  MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)
```

IN comm_old - батьківський комунікатор;
IN ndims - кількість вимірів;
IN dims - масив розміру ndims, в якому задається число процесів вздовж кожного виміру;
IN periods - логічний масив розміру ndims для задання граничних умов (true - періодичні, false - неперіодичні);
IN reorder - логічна змінна, яка вказує, проводити перенумерацію процесів (true) чи ні (false);
OUT comm_cart - новий комунікатор.

Функція є колективною, тобто повинна запускатись на всіх процесах, що входять у групу комунікатора comm_old. При цьому, якщо якісь процеси не попадають у нову групу, то для них повертається результат MPI_COMM_NULL. У випадку, коли розміри замовленої сітки більше числа процесів, яке міститься в групі, функція завершується аварійно. Значення параметра reorder=false означає, що ідентифікатори всіх процесів у новій групі будуть такими ж, як у старій групі. Якщо reorder=true, то MPI буде намагатись перенумерувати їх з метою оптимізації комунікацій.

Інші функції, які будуть розглянуті в цьому розділі, мають допоміжний або інформаційний характер.

Функція визначення оптимальної конфігурації сітки

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
```

IN nnodes - загальна кількість вузлів у сітці;
IN ndims - кількість вимірів;
INOUT dims - масив цілого типу розмірності ndims, в який поміщається рекомендована кількість процесів уздовж кожного виміру.

На вході у процедуру в масив dims мають бути занесені цілі невід'ємні числа. Якщо елементу масиву dims[i] присвоєне додатне число, то для цієї розмірності розрахунки не проводяться (число процесів вздовж цього напрямку вважається заданим). Розраховуються лише ті компоненти dims[i], яким перед викликом процедури були присвоєні значення 0. Функція намагається створити максимально рівномірний розподіл процесів вздовж напрямків, розподіляючи їх за убуванням, тобто для 12 процесів вона побудує тривимірну сітку 4 x 3 x 1. Результат роботи цієї процедури може бути використаний як вхідний параметр для процедури MPI_Cart_create.

Функція опитування числа вимірювань декартової топології MPI_Cartdim_get

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
```

IN comm - комунікатор з декартовою топологією;
OUT ndim - число вимірювань у декартовій топології.

Функція повертає число вимірювань у декартовій топології ndims для комунікатора comm. Результат може бути використаний як параметр для виклику **функції MPI_Cart_get**, яка служить для отримання детальнішої інформації.

```
int MPI_Cart_get(MPI_Comm comm, int ndims, int *dims, int
*periods, int *coords)
```

IN comm - комунікатор з декартовою топологією;
IN ndims - число вимірювань;
OUT dims - масив розміру ndims, в якому повертається кількість процесів уздовж кожного вимірювання;
OUT periods - логічний масив розміру ndims, в якому повертаються накладені граничні умови (true - періодичні, false - неперіодичні);
OUT coords - координати в декартовій сітці процесу, що викликається.

Дві наступні функції встановлюють відповідність між ідентифікатором процесу і його координатами в декартовій сітці. Під ідентифікатором процесу розуміється його номер у початковій області зв'язку, з якої була створена декартова топологія.

Функція отримання ідентифікатора процесу за його координатами MPI_Cart_rank

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

IN comm - комунікатор з декартовою топологією;
IN coords - координати в декартовій системі;
OUT rank - ідентифікатор процесу.

Для вимірювань з періодичними граничними умовами буде виконуватись приведення до основної області визначення 0 <= coords(i) < dims(i).

Функція визначення координат процесу за його ідентифікатором MPI_Cart_coords

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int
*coords)
```

IN comm - комунікатор з декартовою топологією;
IN rank - ідентифікатор процесу;
IN ndim - число вимірювань;
OUT coords - координати процесу в декартовій топології.

У багатьох числових алгоритмах застосовується операція зсуву даних уздовж якихось напрямів декартової сітки. У MPI існує спеціальна функція MPI_Cart_shift, що реалізовує цю операцію. Точніше кажучи, зсув даних здійснюється за допомогою функції MPI_Sendrecv, а функція MPI_Cart_shift обчислює для кожного процесу параметри для функції MPI_Sendrecv (source і dest).

Функція зсуву даних MPI_Cart_shift

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int
*rank_source, int *rank_dest)
```

IN comm - комунікатор з декартовою топологією;
IN direction - номер вимірювання, уздовж якого виконується зсув;
IN disp - величина зсуву (може бути як додатною, так і від'ємною);

OUT rank_sourcea - номер процесу, від якого повинні бути одержані дані;
OUT rank_dest - номер процесу, якому повинні бути відправлені дані.

Номер вимірювання і величина зсуву не повинні бути однаковими для всіх процесів. Залежно від граничних умов зсув може бути або циклічним, або з урахуванням граничних процесів. В останньому випадку для граничних процесів повертається MPI_PROC_NULL або для змінної rank_source, або для rank_dest. Це значення також може бути використане під час виклику функції MPI_sendrecv.

Інша операція, яка часто застосовується, – виділення в декартовій топології підпросторів меншої розмірності і зв'язування з ними окремих комунікаторів.

Функція виділення підпростору в декартовій топології MPI_Cart_sub

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm
*newcomm)
```

IN comm - комунікатор з декартовою топологією;
IN remain_dims - логічний масив розміру ndims, який вказує, чи входить і-те вимірювання в нову підрешітку (remain_dims[i]= true);
OUT newcomm - новий комунікатор, що описує підрешітку, яка містить процес, що її викликає.

Функція є колективною. Дію функції проілюструємо таким прикладом. Припустимо, що є декартова решітка 2 x 3 x 4, тоді виклик функції MPI_Cart_sub з масивом remain_dims (true, false, true) створить три комунікатори з топологією 2x4. Кожний з комунікаторів описуватиме область зв'язку, що складається з 1/3 процесів, що входили в початкову область зв'язку.

Крім розглянутих функцій, у MPI входить набір з 6 функцій для роботи з комунікаторами з топологією графів, які в даному посібнику розглядатися не будуть.

Для визначення топології комунікатора служить функція MPI_Topo_test

```
int MPI_Topo_test(MPI_Comm comm, int *status)
```

IN comm - комунікатор;
OUT status - топологія комунікатора.
Функція MPI_Topo_test повертає через змінну status топологію комунікатора comm. Можливі значення:

- MPI_GRAPH - топологія графа;
- MPI_CART - декартова топологія;
- MPI_UNDEFINED - топологія не задана.

8. Приклади програм

8.1. Визначення числа π

Як перший приклад застосування комунікаційної бібліотеки MPI розглянемо програму визначення числа π.

Для обчислення числа π застосовується числове інтегрування функції $(1+x^2)^{-½}$ від -1 до 1, яке здійснюється методом прямокутників. У випадку паралельної програми цілком очевидно, що можна розбити інтервал інтегрування на N інтервалів і обчислювати інтеграл за кожним інтервалом на окремому процесорі, а потім обчислити суму отриманих величин.

Нижче наведено приклад програми на Фортрані, яка виконує даний алгоритм.

```
      program calc_pi
      use MPI
        integer, parameter :: dp=selected_real_kind(12,50)
      integer :: i, n, np, myid, Ierr
      real(kind=dp) :: w, gsum, sum, v, time, mflops
      real(kind=dp) :: time1, time2, dsecnd
! Ініціалізація MPI та визначення процесорної конфігурації
      Ierr = MPI_INIT(0,0)
      Ierr = MPI_COMM_RANK( MPI_COMM_WORLD, myid )
      Ierr = MPI_COMM_SIZE( MPI_COMM_WORLD, np )
! Інформацію з клавіатури зчитує 0-й процесор
      if ( myid .eq. 0 ) then
        print *, 'Введіть кількість точок розбиття інтервалу: '
        read (*,*) n
        time1 = MPI_Wtime()
      endif
! Розсилка кількості точок розбиття всім процесорам
      Ierr = MPI_BCAST(loc(n), 1, MPI_INTEGER, 0,MPI_COMM_WORLD)
! Обчислення часткової суми на процесорі
      w = 1._dp/ n
      sum = 0.0_dp
      do i = myid+1, n, np
        v = (i - 0.5_dp) * w
        v = 4.0_dp / (1.0_dp + v * v)
        sum = sum + v
      end do
! Підсумування часткових сум зі збереженням результату в 0-му
! процесорі
      Ierr = MPI_REDUCE(loc(sum), loc(gsum), 1, MPI_DOUBLE, &
                        MPI_SUM, 0, MPI_COMM_WORLD)
! Друкування вихідної інформації з 0-го процесора
      if (myid .eq. 0) then
        time2 = MPI_Wtime()
        time   = time2 - time1
        mflops = 9 * n / (1000000.0 * time)
        print *, 'pi is approximated with ', gsum *w
        print *, 'time = ', time, ' seconds'
        print *, 'mflops = ', mflops, ' on ', np, ' processors'
```

```
        print *, 'mflops = ', mflops/np, ' for one processor'
      endif
! Закриття MPI
      Ierr = MPI_FINALIZE()
      end program calc_pi
```

8.2. Перемноження матриць

Розглянутий у попередньому підрозділі приклад являє собою найпростіший для розпаралелювання тип задач, в яких у процесі виконання підзадач не вимагається виконувати обмін інформацією між процесорами. Така ситуація має місце завжди, коли змінна циклу, що розпаралелюється, не індексує які-небудь масиви (типовий випадок - параметричні задачі). У задачах лінійної алгебри, в яких обчислення пов'язані з обробкою масивів, часто виникають ситуації, коли необхідні для обчислення матричні елементи відсутні на оброблювальному процесорі. Тоді процес обчислення не може бути продовжений доти, поки вони не будуть передані в пам'ять процесора, який їх потребує. Як простий приклад задач цього типу розглянемо задачу перемножування матриць.

Існує безліч варіантів розв’язання цієї задачі на багатопроцесорних системах. Алгоритм розв’язання істотним чином залежить від того, проводиться чи ні розподіл матриць по процесорах і яка топологія процесорів при цьому застосовується. Як правило, задачі такого типу розв'язуються або на одновимірній сітці процесорів, або на двовимірній. На рис. 8.1 зображені можливі розподіли матриці в пам'яті 4 процесорів, які утворюють одновимірну і двовимірну сітки.

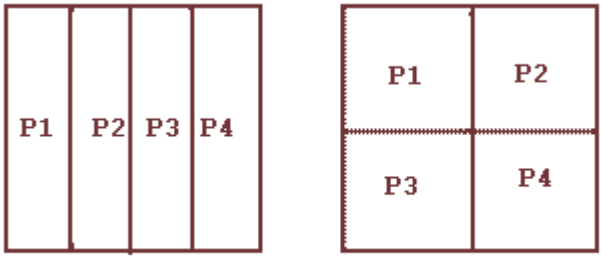

Рис. 8.1. Приклади розподілу матриці на одновимірну і двовимірну сітки процесорів

Перший варіант значно простіший у використанні, оскільки дозволяє працювати із заданим за замовчанням комунікатором. У разі двовимірної сітки процесорів потрібно буде описати створювану топологію і комунікатори для кожного напряму сітки. Кожна з трьох матриць (A, B і C) може бути розподілена одним з 4 способів:

- не розподілена по процесорах;
- розподілена на двовимірну сітку;
- розподілена по стовпцях на одновимірну сітку;
- розподілена по рядках на одновимірну сітку.

Звідси виникає 64 можливі варіанти розв’язування цієї задачі. Більшість із цих варіантів погано відображають специфіку алгоритму і відповідно явно

неефективні. Той або інший спосіб розподілу матриць однозначно визначає, які з трьох циклів обчислювального блока краще використати для процедури редукції.

Нижче пропонується варіант програми розв'язання цієї задачі, який повною мірою враховує специфіку алгоритму. Оскільки для обчислення кожного матричного елемента матриці C необхідно обчислити скалярний добуток рядка матриці A на стовпець матриці B, то матриця A розкладена на одновимірну сітку процесорів по рядках, а матриця B - по стовпцях. Матриця C розкладена по рядках, як матриця A (рис. 8.2).

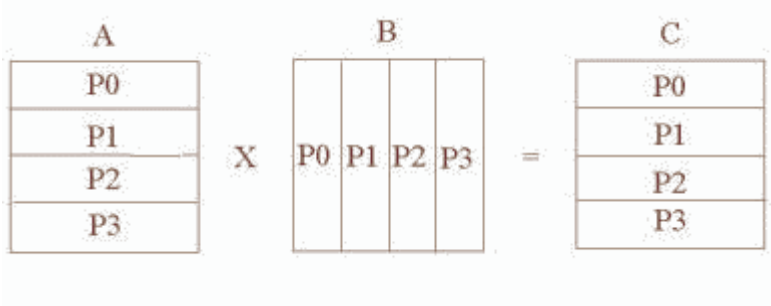

Рис. 8.2. Розподіл матриць на одновимірну сітку процесорів

За такого розподілу рядок, необхідний для обчислення деякого матричного елемента, гарантовано знаходиться в даному процесорі, а стовпець хоча й може бути відсутнім, але цілком розподілений у деякому процесорі. Тому алгоритм розв'язання задачі повинен передбачати визначення, в якому процесорі знаходиться потрібний стовпець матриці B, і пересилання його в той процесор, який його потребує в даний момент. Насправді, кожен стовпець матриці B бере участь в обчисленні всього стовпця матриці C, і тому його слід розсилати у всі процесори.

```
PROGRAM MATMUL_MPI
    use MPI
! Параметри:
! NM    - повна розмірність матриці;
! NPMIN - мінімальна кількість процесорів для виконання задачі;
! NPS   - розмірність локальної частини матриць
    integer, PARAMETER :: NM = 500, NPMIN=4, NPS=NM/NPMIN+1
    REAL(8) :: A(NPS,NM), B(NM,NPS), C(NPS,NM), COL(NM), TIME
! В масивах NB и NS інформація про декомпозицію матриць:
! NB  - кількість рядків матриці в кожному процесорі;
! NS  - номер рядка, починаючи з якого зберігається матриця в даному
процесорі;
! Передбачається, що процесорів не більше ніж 64.
    INTEGER NB(0:63), NS(0:63)
! Ініціалізація MPI
    Ierr = MPI_INIT(0,0)
    Ierr = MPI_COMM_RANK(MPI_COMM_WORLD, IAM)
    Ierr = MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS)
    IF(IAM.EQ.0) WRITE(*,*) 'NM = ',NM,' NPROCS = ',NPROCS
! Обчислення параметрів декомпозиції матриць
! Алгоритм реалізує максимально рівномірний розподіл
    NB1 = NM/NPROCS
    NB2 = MOD(NM,NPROCS)
    DO I = 0,NPROCS-1
```

```
      NB(I) = NB1
    END DO
    DO I = 0,NB2-1
      NB(I)= NB(I)+1
    END DO
    NS(0)=0
    DO I = 1,NPROCS-1
      NS(I)= NS(I-1) + NB(I-1)
    END DO
     ! Заповнення матриці A
    DO J = 1,NM
      DO I = 1,NB(IAM)
        A(I,J) = DBLE(I+NS(IAM))
      END DO
    END DO
!    Заповнення матриці B(I,J)=1/J
    DO I = 1,NM
      DO J = 1,NB(IAM)
       B(I,J) =1./DBLE(J+NS(IAM))
      END DO
    END DO
    TIME = MPI_WTIME()   ! Виклик таймера
!---------------- Блок обчислення, ----------------------------
! Цикли по рядках та по стовпцях переставлені місцями і цикл по
! стовпцях розбитий на дві частини: по процесорах J1 і по
! елементах всередині процесора J2. Це зроблено для того, щоб не
! обчислювати, в якому процесорі знаходиться даний стовпець.
! Змінна J виконує наскрізну нумерацію стовпців. ---------------
    J = 0
    DO J1 = 0,NPROCS-1
      DO J2 = 1,NB(J1)
       J = J + 1
! Процесор, що зберігає черговий стовпець, розсилає його всім
! іншим процесорам
       IF(IAM.EQ.J1) THEN
         DO N = 1,NM
            COL(N) = B(N,J2)
         END DO
       END IF
       Ierr=MPI_BCAST(loc(COL),NM,MPI_DOUBLE,J1, MPI_COMM_WORLD)
! Цикл по рядках (саме він скорочений)
       DO I = 1,NB(IAM)
          C(I,J) = 0.0
! Внутрішній цикл
          DO K = 1,NM
             C(I,J) = C(I,J) + A(I,K)*COL(K)
          END DO
       END DO
      END DO
    END DO
    TIME = MPI_WTIME() - TIME
! Друк контрольних кутових матричних елементів матриці C
! з тих процесорів, де вони зберігаються
```

```
    IF(IAM.EQ.0) WRITE(*,*) IAM,C(1,1),C(1,NM)
    IF(IAM.EQ.NPROCS-1) WRITE(*,*) IAM, C(NB(NPROCS-1),1),&
                          C(NB(NPROCS-1), NM)
    IF(IAM.EQ.0) WRITE(*,*) ' TIME COMPUTATION: ', TIME
    Ierr = MPI_FINALIZE()
    STOP
  END PROGRAM MATMUL_MPI
```

На відміну від програми визначення числа π, у цій програмі практично неможливо виділити зміни порівняно з однопроцесорним варіантом. По суті справи, це абсолютно нова програма, що має дуже мало спільного з прототипом. Розпаралелювання в даній програмі полягає в тому, що кожен процесор обчислює свій блок матриці C, який складає приблизно 1/NPROCS частин повної матриці. Неважко помітити, що пересилання даних були б не потрібні, якби матриця B не розподілялася по процесорах, а цілком зберігалася в кожному процесорі. У деяких випадках така асиметрія в розподілі матриць буває дуже вигідна.

8.3. Розв'язування крайової задачі методом Якобі

Ще одна область, для якої достатньо добре розроблена технологія паралельного програмування, - це крайові задачі. У цьому випадку застосовується техніка декомпозиції по процесорах розрахункової області, як правило, з перекриттям підобластей. На рис. 8.3 зображене таке розкладання початкової розрахункової області на 4 процесори з топологією одновимірної сітки. Заштриховані області на кожному процесорі позначають ті вузли сітки, в яких розрахунок не проводиться, але вони необхідні для виконання розрахунку в примежових вузлах. У них повинна бути заздалегідь поміщена інформація з примежового шару сусіднього процесора.

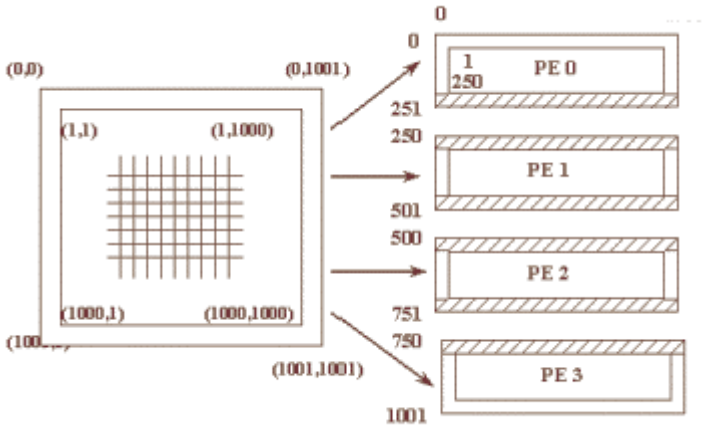

Рис. 8.3. Приклад декомпозиції двовимірної розрахункової області

Наведемо приклад програми розв'язання рівняння Лапласа методом Якобі на двовимірній регулярній сітці. У програмі декомпозиція області виконана не по рядках, як на рис. 8.3, а по стовпцях. Так зручніше у разі програмування на мові

FORTRAN, на С зручніше робити розбиття по рядках (це визначається способом розміщення матриць в пам'яті комп'ютера).

```
! Програма числового розв’язування рівняння Лапласа методом
! Якобі з використанням функції MPI_Sendrecv та NULL процесів
    program JACOBI
    use MPI
! n      - кількість точок області в кожному напрямі;
! npmin  - мінімальне число процесорів для розв’язання задачі;
! npsaa  - число стовпців локальної частини матриці.
! itmax  - максимальна кількість ітерацій
    INTEGER, PARAMETER :: n=400,npmin=1,nps=n/npmin+1,itmax=1000
    REAL(8) :: A(0:n+1,0:nps+1),B(n,nps), diff, gdiff, eps, time
    INTEGER :: left,right,i,j,k,itcnt,status(0:3),tag,IAM,NPROCS
!Визначення кількості процесорів NPROCS та номера процесора IAM
    Ierr = MPI_INIT(0,0)
    Ierr = MPI_COMM_SIZE(MPI_COMM_WORLD, NPROCS)
    Ierr = MPI_COMM_RANK(MPI_COMM_WORLD, IAM)
    eps = 0.001    ! Установка критерію досягнення збіжності
! Обчислення кількості стовпців, які оброблюються процесором
    m = n/NPROCS
    IF (IAM.LT.(n-NPROCS*m))  m = m+1
    time = MPI_Wtime()
! Задавання початкових та граничних значень
    do j = 0,m+1
       do i = 0,n+1
          a(i,j) = 0.0
       end do
    end do
    do j = 0,m+1
       A(0,j) = 1.0
       A(n+1,j) = 1.0
    end do
    IF(IAM.EQ.0) then
      do i = 0,n+1
         A(i,0) = 1.0
      end do
    end if
    IF(IAM.EQ.NPROCS-1) then
      do i = 0,n+1
         A(i,m+1) = 1.0
      end do
    end if
! Визначення номерів процесорів зліва та справа. Якщо такі
! відсутні, то їм присвоюється значення MPI_PROC_NULL
    IF (IAM.EQ.0) THEN
      left = MPI_PROC_NULL
    ELSE
      left = IAM - 1
    END +IF
    IF (IAM.EQ.NPROCS-1) THEN
      right = MPI_PROC_NULL
    ELSE
```

```
      right = IAM+1
    END IF
    tag = 100
    itcnt = 0
! Цикл по ітераціях
    DO k = 1,itmax
      diff = 0.0
      itcnt = itcnt + 1
! Обчислення нових значень функції за 5-точковою схемою
      DO j = 1, m
        DO i = 1, n
          B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
          diff = diff + (B(i,j)-A(i,j))**2
        END DO
      END DO
! Переприсвоєння внутрішніх точок області
      DO j = 1, m
        DO i = 1, n
          A(i,j) = B(i,j)
        END DO
      END DO
! Пересилання граничних значень у сусідні процесори
      Ierr=MPI_SENDRECV(loc(B(1,1)),n,MPI_DOUBLE,left,tag, &
       loc(A(1,0)),n,MPI_DOUBLE,left,tag,MPI_COMM_WORLD,   &
       status)
       Ierr = MPI_SENDRECV(loc(B(1,m)), n, MPI_DOUBLE, right, &
         tag, loc(A(1,m+1)), n, MPI_DOUBLE, right, tag,       &
         MPI_COMM_WORLD, status)
! Обчислення відхилу та перевірка умови досягнення збіжності
    Ierr = MPI_Allreduce( loc(diff), loc(gdiff), 1, MPI_DOUBLE,&
       MPI_SUM, MPI_COMM_WORLD )
      gdiff = sqrt( gdiff)
      if(gdiff.LT.eps) exit
    END DO
    time = MPI_Wtime() - time
    IF(IAM.EQ.0) then
     WRITE(*,*) ' At iteration ', itcnt, 'a diff is ', gdiff
     WRITE(*,*) ' Time calculation: ', time
    END IF
    Ierr = MPI_Finalize()
    stop
    end program JACOBI
```

9. Що таке OpenMP?

OpenMP (Open specifications for Multi-Processing) - це набір специфікацій для паралелізації програм у середовищі із загальною пам'яттю. Інтерфейс OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах (SSMP, ccNUMA) в моделі загальної пам'яті (shared memory model). У стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища.

OpenMP дозволяє легко і швидко створювати багатопоточні додатки на алгоритмічних мовах Fortran і C/C++. При цьому директиви OpenMP аналогічні директивам препроцесора для мови C/C++ і є аналогом коментарів у алгоритмічній мові Fortran. Це дозволяє в будь-який момент розробки паралельної реалізації програмного продукту при необхідності повернутися до послідовного варіанту програми.

9.1. Спільна пам'ять

OpenMP - це стандартна модель для паралельного програмування в середовищі зі спільною пам'яттю. У даній моделі всі процеси спільно використовують загальний адресний простір, до якого вони асинхронно звертаються із запитами на читання і запис. У таких моделях для управління доступом до загальної пам'яті використовуються різні механізми синхронізації типу семафорів та блокування процесів. Перевага цієї моделі з точки зору програмування полягає в тому, що поняття монопольного володіння даними відсутнє, отже, не потрібно явно задавати обмін даними між потоками, що їх задають, та потоками, що їх використовують. Ця модель, з одного боку, спрощує розробку програми, але, з іншого боку, ускладнює розуміння і управління локальністю даних, написання детермінованих програм. В основному вона використовується при програмуванні для архітектур зі спільною пам'яттю.

Прикладами систем зі спільною пам'яттю, які мають велике число процесорів, можуть служити суперкомп'ютери Cray Origin2000 (до 128 процесорів), HP 9000 V-class (до 32 процесорів в одному вузлі, а в конфігурації з 4 вузлів - до 128 процесорів), Sun Starfire (до 64 процесорів).

9.2. Хто розробляє стандарт?

Розробкою стандарту займається організація OpenMP ARB (ARchitecture Board), до якої увійшли представники найбільших компаній - розробників SMP-архітектур і програмного забезпечення. Перша версія специфікації OpenMP (www.openmp.org) з'явилася в 1997 році і призначалася для мови програмування Фортран. Біля витоків OpenMP стояли такі відомі компанії, як IBM, Intel, Sun і Hewlett-Packard. У 1998 році з'явилися варіанти OpenMP для мов C/C++, і на даний момент останньою є версія 3.0 [5].

В даний час OpenMP підтримується більшістю розробників паралельних обчислювальних систем: компаніями Intel, Hewlett-Packard, Silicon Graphics, Sun,

IBM, Fujitsu, Hitachi, Siemens, Bull і іншими. Багато відомих компаній в галузі розробки системного програмного забезпечення також приділяють значну увагу розробці системного програмного забезпечення з OpenMP. Серед цих компаній відзначимо Intel, KAI, PGI, PSR, APR, Absoft і деякі інші. Значне число компаній і науково-дослідних організацій, які розробляють прикладне програмне забезпечення, в даний час використовує OpenMP при розробці своїх програмних продуктів. Серед цих компаній і організацій відзначимо ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software.

Компілятори gcc і gfortran мають вбудовану підтримку паралелізації OpenMP починаючи з версії 4.2. Для використання цієї можливості слід додати ключ `-fopenmp` при компіляції. OpenMP версії 3.0 підтримується починаючи з версії gcc 4.4. OpenMP 2.5 підтримується також компілятором Microsoft Visual C + + 2005 (слід використовувати ключ /openmp) і компіляторами Intel C або Intel Fortran починаючи з версії 10.1 (ключ -openmp).

9.3. Основи OpenMP

Будь-яка програма, послідовна або паралельна, складається з набору областей двох типів: послідовних областей і областей розпаралелювання. При виконанні послідовних областей породжується тільки один головний потік виконання (процес). У цьому потоці ініціюється виконання програми, а також відбувається її завершення. У послідовної програми цей потік є єдиним протягом виконання всієї програми. У паралельній програмі в областях розпаралелювання породжується цілий ряд паралельних потоків. Породжені паралельні потоки можуть виконуватися як на різних процесорах, так і на одному процесорі обчислювальної системи. В останньому випадку паралельні процеси (потоки) конкурують між собою за доступ до процесора. Управління конкуренцією здійснюється планувальником операційної системи за допомогою спеціальних алгоритмів. В операційній системі Linux планувальник завдань здійснює обробку процесів за допомогою стандартного карусельного (round-robin) алгоритму. При цьому тільки адміністратори системи мають можливість змінити або замінити цей алгоритм системними засобами. Таким чином, у паралельних програмах в областях розпаралелювання виконується ряд паралельних потоків. Принципова схема паралельної програми зображена на рис. 9.1.

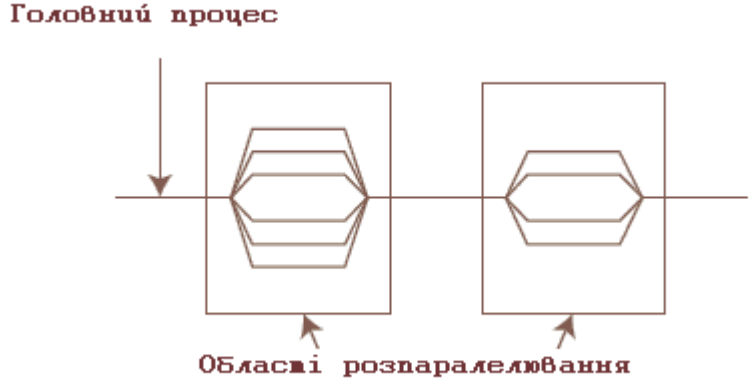

Рис. 9.1. Принципова схема паралельної програми

При виконанні паралельної програми робота починається з ініціалізації і виконання головного потоку (процесу), який у міру необхідності створює і виконує паралельні потоки виконання, передаючи їм необхідні дані. Паралельні потоки з однієї паралельної області програми можуть виконуватися як незалежно один від одного, так і з пересиланням та отриманням повідомлень від інших паралельних потоків. Остання обставина ускладнює розробку програми, оскільки в цьому випадку програмістові доводиться займатися плануванням, організацією і синхронізацією посилки повідомлень між паралельними потоками. Таким чином, при розробці паралельної програми бажано виділяти такі області розпаралелювання, в яких можна організувати виконання незалежних паралельних потоків. Для обміну даними між паралельними процесами (потоками) в OpenMP використовуються загальні змінні. При зверненні до загальних змінних у різних паралельних потоках можливе виникнення конфліктних ситуацій при доступі до даних. Для запобігання конфліктів можна скористатися процедурою синхронізації. При цьому треба мати на увазі, що процедура синхронізації - дуже дорога операція по тимчасових витратах і бажано по можливості уникати її або застосовувати якомога рідше. Для цього необхідно дуже ретельно продумувати структуру даних програми.

Виконання паралельних потоків в паралельній області програми починається з їх ініціалізації. Вона полягає у створенні дескрипторів породжуваних потоків і копіюванні всіх даних з області даних головного потоку в області даних паралельних потоків, що створюються. Ця операція надзвичайно трудомістка - вона еквівалентна приблизно трудомісткості не менше 1000 машинних команд. Ця оцінка надзвичайно важлива при розробці паралельних програм с допомогою OpenMP, оскільки її ігнорування веде до створення неефективних паралельних програм, які виявляються часто повільнішими ніж їх послідовні аналоги. Справді: для того щоб отримати виграш у швидкодії паралельної програми, необхідно, щоб трудомісткість паралельних процесів в областях розпаралелювання програми істотно перевершувала б трудомісткість породження паралельних потоків. В іншому випадку ніякого виграшу за швидкодією отримати не вдасться, а часто можна опинитися навіть і в програші.

Після завершення виконання паралельних потоків управління програмою знову передається головному потоку. При цьому виникає проблема коректної передачі даних від паралельних потоків головного. Тут важливу роль грає синхронізація завершення роботи паралельних потоків, оскільки в силу цілого ряду обставин час виконання навіть однакових за трудомісткістю паралельних потоків непередбачувано (воно визначається як історією конкуренції паралельних процесів, так і поточним станом обчислювальної системи). При виконанні операції синхронізації паралельні потоки, вже завершили своє виконання, простоюють і чекають завершення роботи самого останнього потоку. Природно, при цьому неминуча втрата ефективності роботи паралельної програми. Крім того, операція синхронізації має трудомісткість, порівнянну з трудомісткістю ініціалізації паралельних потоків.

9.4. Які переваги OpenMP дає розробнику?

1. За рахунок ідеї **"інкрементального розпаралелювання"** OpenMP ідеально підходить для розробників, хто хоче швидко розпаралити свої обчислювальні програми з великими циклами. Розробник не створює нову паралельну програму, а просто послідовно додає в текст послідовної програми OpenMP-директиви.
2. При цьому, OpenMP - досить **гнучкий механізм**, що надає розробнику великі можливості контролю над поведінкою паралельного програми.
3. Передбачається, що OpenMP-програма на однопроцесорній платформі може бути використана **як послідовна** програма, тобто немає необхідності підтримувати послідовну і паралельну версії. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs), текст яких наведений у специфікаціях [17].

Далі наведено короткий опис специфікації OpenMP та наведено приклади використання в програмах на алгоритмічних мовах C і Фортран. Матеріал включає короткий опис основних директив OpenMP, процедур і змінних середовища. Для вивчення всіх тонкощів використання OpenMP слід звертатися до керівництва користувача [17] та книг [1-4].

10. Прагми OpenMP

Специфікація OpenMP визначає набір прагм. Прагма - це директива компілятора що вказує, як обробляти код, який слідує за нею. Найбільш суттєвою є прагма `#pragma omp parallel`, що визначає область паралельності.

Коли прагми OpenMP використовуються в програмі, вони дають вказівку компілятору, що підтримує OpenMP, створити виконуваний модуль, який буде виконуватися паралельно з використанням декількох потоків. При цьому в вихідний код необхідно внести достатньо невеликі зміни. Прагми OpenMP дозволяють використовувати красивий, одноманітний і переносимий інтерфейс для паралелізації програм на різних архітектурах та системах. Специфікація OpenMP прийнята багатьма і підтримується такими постачальниками, як Sun, Intel, IBM і SGI. (Посилання на веб-сайт OpenMP, де є документ з останньою версією специфікації OpenMP, знаходиться нижче у розділі Література [17].)

Модель OpenMP дозволяє паралельному програмуванню піднятися на наступний рівень, створюючи для програміста потоки виконання і керуючи ними. Все, що необхідно, це вставити відповідні прагми у вихідний код програми і потім скомпілювати програму компілятором, який підтримує OpenMP, з відповідним ключем. Компілятор інтерпретує прагми і параллелізует код. При використанні компіляторів, що не підтримують OpenMP, прагми OpenMP ігноруються без додаткових повідомлень.

Основні конструкції OpenMP - це директиви компілятора або прагма (директиви препроцесора) мови C/C++. Нижче наведено загальний вигляд директиви OpenMP прагма.

```
#Pragma omp конструкція [пропозиція [пропозиція] ...]
```

У мові Фортран директиви OpenMP є коментарями і починаються з комбінації символів "!$OMP". Нижче наведені приклади таких рядків у загальному вигляді

```
!$OMP PARALLEL
!$OMP END PARALLEL
!$OMP PARALLEL DO ...
```

Для звичайних послідовних програм директиви OpenMP не змінюють структуру і послідовність виконання операторів. Таким чином, звичайна послідовна програма зберігає свою працездатність. У цьому й полягає гнучкість розпаралелювання за допомогою OpenMP.

Директиви можна розділити на 3 категорії:

- визначення паралельної секції,
- розподіл роботи,
- синхронізація.

Кожна директива може мати кілька додаткових атрибутів - *клауз (clause)* . Окремо специфікуються клаузи для призначення класів змінних, які можуть бути атрибутами різних директив.

Більшість директив OpenMP застосовується до структурних блоків. Структурні блоки - це послідовності операторів з однією точкою входу на початку блоку і однією точкою виходу в кінці блоку.

У OpenMP використовується модель паралельного виконання "розгалуження-злиття". Програма OpenMP починається як єдиний потік виконання (так званий початковий потік). Коли потік зустрічає паралельну конструкцію, він створює нову групу потоків, що складається з себе і невід'ємного числа додаткових потоків, і стає головним у новій групі. Всі члени нової групи (включаючи головний потік) виконують код всередині паралельної конструкції. У кінці паралельної конструкції є неявний бар'єр. Після паралельної конструкції виконання користувацького коду продовжує тільки головний потік.

Кількість потоків у групі, що виконуються в області паралельності, можна контролювати декількома способами. Один з них - використання змінної середовища `OMP_NUM_THREADS`. Інший спосіб - виклик процедури `omp_set_num_threads()`. Ще один спосіб - використання клаузи `num_threads` у поєднанні з прагмою `parallel`.

У OpenMP підтримуються дві основних конструкції розділення праці для зазначення того, що роботу в області паралельності слід розділити між потоками групи. Ці конструкції розділення праці - цикли і розділи. прагма `#pragma omp for` використовується для циклів, а прагма `#pragma omp sections`

використовується для розділів - блоків коду, які можуть бути виконані паралельно.

Прагма `#pragma omp barrier` дає всіх потоків вказівку чекати один одного перед тим, як вони продовжать виконання за бар'єром. Як було зазначено вище, в кінці області паралельності є неявний бар'єр. Прагма `#pragma omp master` дає компілятору вказівку про те, що наступний блок коду повинен виконуватися тільки головним потоком. Прагма `#pragma omp single` показує, що наступний блок коду повинен виконуватися тільки одним потоком групи; цей потік не обов'язково повинен бути головним. Прагма `#pragma omp critical` може використовуватися для захисту блоку коду, який повинен виконуватися одночасно тільки одним потоком. Звичайно, всі ці прагма мають сенс тільки в контексті прагма parallel (області паралельності).

10.1. Породження потоків виконання

`PARALLEL ... END PARALLEL`

Визначає *паралельну область* програми. При вході в цю область породжуються нові (N-1) потоки виконання, утворюється "команда" з N потоків, а головний процес отримує номер 0 і стає основним потоком команди (так званим "master thread"). При виході з паралельної області основний потік виконання чекає завершення інших потоків, і продовжує виконання в єдиному екземплярі. Передбачається, що в SMP-системі потоки будуть розподілені по різних процесорам. Однак це, як правило, перебуває у віданні операційної системи.

Яким чином між породженими потоками **розподіляється робота** - визначається директивами `DO`, `SECTIONS` і `SINGLE`. Можливо також явна управління розподілом роботи (*подібно MPI*) за допомогою функцій, що повертають номер даного потоку і загальне число потоків. За замовчуванням (за цими директивами), код всередині `PARALLEL` виконується всіма потоками однаково.

Разом з `PARALLEL` може використовуватися клауза `IF (умова)`. При цьому паралельна робота ініціюється тільки при виконанні зазначеного в ній умови.

Паралельні області можуть динамічно вкладеними. За замовчуванням (якщо вкладений паралелізм не дозволений явно), внутрішня паралельна область виконується одним потоком.

10.2. Розподіл роботи (work-sharing constructs)

Паралельні цикли

`DO ... [ENDDO]`

Визначає паралельний цикл.

Клауза `SCHEDULE` визначає спосіб розподілу ітерацій по нитках:

- `STATIC,m` - статично, блоками по m ітерацій

- DYNAMIC,m - динамічно, блоками по m (кожна нитка бере на виконання перший ще нездоланих вершин блок ітерацій)
- GUIDED,m - розмір блоку ітерацій зменшується експоненціально до величини m
- RUNTIME - вибирається під час виконання.

За замовчуванням, в кінці циклу відбувається неявна синхронізація; цю синхронізацію можна заборонити з допомогою клаузи ENDDO NOWAIT.

Паралельні секції

SECTIONS ... END SECTIONS

Не-ітеративна паралельна конструкція. Визначає набір незалежних секцій коду (так званий "кінцевий" паралелізм). Секції відокремлюються один від одного директивою SECTIONS.

Примітка. Якщо всередині PARALLEL міститься тільки одна конструкція DO або тільки одна конструкція SECTIONS, то можна використовувати скорочений запис: PARALLEL DO або PARALLEL.

Виконання одним потоком

SINGLE ... END SINGLE

Визначає блок коду, який буде виконано тільки одним потоком (першим, що дійде до цього блоку).

Явне управління розподілом роботи

За допомогою функцій OMP_GET_THREAD_NUM() і OMP_GET_NUM_THREADS() потік виконання може взнати свій номер і загальну кількість потоків, а потім виконувати свою частину роботи залежно від свого номера. (Цей підхід широко використовується в програмах на базі інтерфейсу MPI).

10.3. Директиви синхронізації

MASTER ... END MASTER

Визначає блок коду, який буде виконанувати тільки master (нульовий потік).

CRITICAL ... END CRITICAL

Визначає критичну секцію, тобто блок коду, який не повинен виконуватися одночасно двома або більше потоками.

BARRIER

Визначає точку бар'єрної синхронізації, в якій кожен потік чекає всіх інших.

ATOMIC

Визначає змінну в лівій частині оператора "атомарного" присвоювання, яка повинна коректно оновлюватися декількома потоками.

ORDERED ... END ORDERED

Визначає блок усередині тіла циклу, який повинен виконуватися в тому порядку, в якому ітерації йдуть у послідовному циклі. Може використовуватися для впорядкування виведення від паралельних потоків виконання.

FLUSH

Явно визначає точку, в якій реалізація повинна забезпечити однаковий зміст пам'яті для всіх потоків. Неявно FLUSH присутній у наступних директивах: BARRIER, CRITICAL, END CRITICAL, END DO, END PARALLEL, END SECTIONS, END SINGLE, ORDERED, END ORDERED.

Для синхронізації можна також користуватися механізмом замків (locks).

10.4. Класи змінних

В OpenMP змінні в паралельних областях програми розділяються на два основні класи:

- SHARED (глобальні; під ім'ям А всі потоки бачать одну змінну)
- PRIVATE (приватні; під ім'ям А кожен потік бачить свою змінну).

Окремі правила визначають поведінку змінних при вході і виході з паралельної області або паралельного циклу: REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYIN.

За замовчуванням, всі COMMON-блоки, а також змінні, породжені поза паралельною областю, при вході в цю область залишаються глобальними (SHARED). Виняток становлять змінні - лічильники ітерацій в циклі, з очевидних причин. Змінні, породжені всередині паралельної області, є приватними (PRIVATE). Явно призначити клас змінних за замовчуванням можна за допомогою клаузи DEFAULT.

SHARED

Застосовується до змінних, які необхідно зробити глобальними.

PRIVATE

Застосовується до змінних, які необхідно зробити приватними. При вході в паралельну область для кожного потоку виконання створюється окремий екземпляр змінної, який не має жодного зв'язку з оригінальною змінною поза паралельною областю.

THREADPRIVATE

Застосовується до COMMON-блоків, які необхідно зробити приватними. Директива повинна застосовуватися після кожної декларації COMMON-блоку.

FIRSTPRIVATE

Приватні копії змінної при вході в паралельну область ініціалізуються значенням оригінальної змінної.

LASTPRIVATE

Після закінчення паралельно циклу або блоку паралельних секцій, потік, який виконав останню ітерацію циклу або останню секцію блоку, оновлює значення оригінальної змінної.

`REDUCTION(+:A)`

Вказує на змінну, з якою в циклі здійснюється операція редукції (наприклад, в даному випадку підсумовування). При виході з циклу, дана операція проводиться над копіями змінної у всіх потоках, і результат присвоюється оригінальній змінній.

`COPYIN`

Застосовується до COMMON-блоків, які позначені як THREADPRIVATE. При вході в паралельну область приватні копії цих даних ініціалізуються оригінальними значеннями.

11. Процедури і змінні середовища

У OpenMP передбачений також набір бібліотечних процедур, які дозволяють:

- під час виконання контролювати і запитувати різні параметри, що визначають поведінку програми (такі як число потоків і процесорів, можливість вкладеного паралелізму); процедури призначення параметрів мають пріоритет над відповідними змінними середовища.
- використовувати синхронізацію на базі замків (locks).

З метою створення можливості запуску паралельних програм в різних умовах, в OpenMP визначено також ряд змінних середовища, які контролюють поведінку програми.

11.1. Процедури OpenMP

OpenMP надає ряд процедур, які можна використовувати для отримання інформації про потоки у програмі. До їх число належать `omp_get_num_threads()`,`mp_set_num_threads()`,`omp_get_max_threads()`, `omp_in_parallel()` та інші. Крім того, OpenMP надає ряд процедур блокування, які можна використовувати для синхронізації потоків, а також процедури для визначення поточного часу.

11.2. Процедури для контролю/запиту параметрів середовища виконання

`OMP_SET_NUM_THREADS`

Дозволяє призначити максимальне число потоків для використання в наступній паралельної області (якщо це число дозволено міняти динамічно). Викликається з послідовної області програми.

`OMP_GET_MAX_THREADS`

Повертає максимальне число потоків.

`OMP_GET_NUM_THREAD`

Повертає фактичне число потоків в паралельній області програми.

OMP_GET_NUM_PROCS

Повертає число процесорів, доступних програмі.

OMP_IN_PARALLEL

Повертає. TRUE., якщо викликана з паралельної області програми.

OMP_SET_DYNAMIC/OMP_GET_DYNAMIC

Встановлює / запитує стан прапора, що дозволяє динамічно змінювати число потоків.

OMP_GET_NESTED / OMP_SET_NESTED

Встановлює / запитує стан прапора, що дозволяє вкладений паралелізм.

11.3. Процедури для синхронізації на базі замків

В якості замків використовуються загальні змінні типу INTEGER (Розмір повинен бути достатнім для зберігання адреси). Дані змінні повинні використовуватися лише як параметри примітивів синхронізації.

OMP_INIT_LOCK(var) / OMP_DESTROY_LOCK(var)

Ініціалізує замок, звязаний зі змінною var.

OMP_SET_LOCK

Заставляє потік виконання, що її викликав, чекати звільнення замка, а потім захоплює його.

OMP_UNSET_LOCK

Звільняє замок, якщо він був захоплений потоком.

OMP_TEST_LOCK

Пробує звільнити вказаний замок. Якщо це неможливо, вертає .FALSE.

11.4. Змінні середовища OpenMP

OpenMP надає кілька змінних середовища, які можна використовувати для управління поведінкою програми, що використовує OpenMP.

Важливою змінною середовища є змінна OMP_NUM_THREADS, яка вказує кількість потоків у групі, яка повинна використовуватися для виконання в області паралельності (включаючи головний потік групи). Інша широко застосовувана змінна середовища - OMP_DYNAMIC. Щоб відключити динамічна зміна числа потоків реалізацією у час виконання, для цієї змінної слід встановити значення FALSE . Загальним правилом є не робити число потоків більшим, ніж число процесорних ядер в системі.

OMP_SCHEDULE

Визначає спосіб розподілу ітерацій в циклі, якщо в директиві DO використана клауза SCHEDULE(RUNTIME).

OMP_NUM_THREADS

Визначає число потоків для виконання паралельних областей програми.

OMP_DYNAMIC

Дозволяє або забороняє динамічну зміну числа потоків.

OMP_NESTED

Дозволяє або забороняє вкладений паралелізм.

Крім змінних середовища, передбачених стандартом OpenMP, компілятори gcc надають додатковий набір специфічних змінних середовища, що забезпечує додаткові можливості управління середовищем часу виконання. Ці змінні описані в керівництві користувача по gcc.

12. Приклади використання OpenMP

Потужність і простоту OpenMP найкраще можна продемонструвати на прикладі. Наступний цикл перетворює 32-розрядний піксель в колірній моделі RGB в 8-розрядний піксель в моделі Grayscale. Все, що потрібно для паралельного виконання в цьому випадку - одна прагма, вставлена безпосередньо перед циклом.

```
#pragma omp parallel for
for (i=0; i < numPixels; i++){
   pGrayScaleBitmap[i] = (unsigned BYTE)
            (pRGBBitmap[i].red * 0.299 +
             pRGBBitmap[i].green * 0.587 +
             pRGBBitmap[i].blue * 0.114);
}
```

Давайте уважно розглянемо цей цикл. По-перше, у цьому прикладі використовується "розподіл навантаження" - загальний термін, який застосовується у OpenMP для опису розподілу робочого навантаження між потоками. Якщо розподіл навантаження застосовується з директивою for, як показано в прикладі, ітерації циклу розподіляються між декількома потоками, так що кожна ітерація циклу виконується тільки один раз, паралельно одним або декількома потоками. OpenMP визначає, скільки потоків слід створити, а також найкращий спосіб створення, синхронізації та знищення потоків. Все, що потрібно від програміста - вказати OpenMP, який саме цикл слід розпаралелити.

У OpenMP існують п'ять обмежень на те, які цикли можна розпаралелити:

- Змінна циклу повинна мати тип signed integer. Беззнакові цілі числа, такі як DWORD, працювати не будуть.
- Операція порівняння повинна мати такий формат: `змінна_цикла <=,>,>= інваріант_цикла_цілого_типу`.
- Третє вираз (або інкрементна частина циклу `for`) повинно бути або цілочисловим додаванням, або цілочисловим відніманням і має практично збігатися зі значенням інваріанта циклу.
- Якщо використовується операція порівняння < або <=, змінна циклу повинна збільшуватися при кожній ітерації, а при використанні операції > або > = змінна циклу повинна зменшуватися.
- Цикл повинен бути базовим блоком. Це означає, що не дозволені переходи з циклу, за винятком оператора exit, який завершує роботу всього додатку. Якщо використовуються оператори goto або break, вони повинні приводити

до переходів усередині циклу, а не поза ним. Те ж саме відноситься до обробки винятків; винятку повинні перехоплюватися всередині циклу.

Хоча ці обмеження здаються стримуючими, нестандартні цикли можна легко змінити відповідним чином.

Якщо цикл відповідає всім обмеженням і компілятор розпаралелив цикл, це ще не гарантує правильної роботи, оскільки може існувати залежність даних. Залежність даних існує, якщо різні ітерації циклу (точніше кажучи, ітерація, яка виконується в іншому потоці) виконують читання або запис спільної пам'яті. Розглянемо наступний приклад, в якому обчислюються факторіали

```
// Не робіть так. Це не працює внаслідок залежності даних.
#pragma omp parallel for
for (i=2; i < 10; i++){
   factorial[i] = i * factorial[i-1];
}
```

Компілятор створює з цього циклу потік, який, проте, завершується помилкою, оскільки принаймні одна з ітерацій циклу залежить від даних іншої ітерації. Подібна ситуація називається станом гонки або гонками даних. Стан гонки виникає тільки при використанні загальних ресурсів (наприклад, пам'яті) при паралельному виконанні. Для вирішення цієї проблеми слід або змінити цикл, або вибрати інший алгоритм, який не приведе до змагання потоків.

Стан гонки важко виявити, оскільки, в даному екземплярі, змінні можуть "вигравати гонку" в тому порядку, який забезпечує правильну роботу програми. Але те, що програму вдалося виконати один раз, не означає, що вона буде працювати завжди. Гарною відправною точкою є тестування програми в різних системах, в одних з яких застосовується технологія Hyper-Threading, а в інших - кілька фізичних процесорів. Також можуть виявитися корисними такі інструменти, як Intel Thread Checker. Традиційні програми налагодження не в змозі виявити гонки даних, оскільки змушують один потік зупинити "гонку", в той час як інші потоки продовжують вносити значні зміни в динамічну поведінку.

На прикладі простої програми множення матриць можна побачити, як слід використовувати OpenMP для паралелізації програми. Розглянемо наступний невеликий фрагмент коду множення двох матриць. Це дуже простий приклад, і якщо дійсно необхідно написати хорошу підпрограму для перемноження матриць, слід взяти до уваги ефекти кешування або використовувати найкращий алгоритм (Штрассе, або Копперсміта і Винограду, або якийсь інший).

```
for (ii = 0; ii < dim; ii++) {
  for (jj = 0; jj < dim; jj++) {
    for (kk = 0; kk < dim; kk++) {
      array[ii][jj] += array1[ii][kk] * array2[kk][jj];
    }
  }
}
```

Зверніть увагу, що на кожному рівні вкладеності циклів тіла циклів можуть виконуватися незалежно один від одного. Так що паралелізувати наведений вище код дуже просто: вставте прагма `#pragma omp parallel for` перед самим зовнішнім циклом (wиклом по змінній `ii`). Краще всього вставити прагма перед самим зовнішнім циклом, так як це дасть найбільший виграш в продуктивності. У параллелізованном циклі змінні `array`, `array1`, `array2` і `dim` є загальними для потоків, а змінні `ii`, `jj` і `kk` є приватними для кожного потоку. Наведений вище код тепер стає таким:

```
#pragma omp parallel for shared(array,array1,array2,dim)
private(ii,jj,kk)
for (ii = 0; ii < dim; ii++) {
  for (jj = 0; jj < dim; jj++) {
    for (kk = 0; kk < dim; kk++) {
      array[ii][jj] = array1[ii][kk] * array2[kk][jj];
    }
  }
}
```

В якості іншого прикладу розглянемо наступний фрагмент коду, призначений для обчислення суми значень `f(x)` для цілих аргументів з інтервалу `0 <= x < n`.

```
for (ii = 0; ii < n; ii++) {
  sum = sum + some_complex_long_fuction(a[ii]);
}
```

Для паралелізації наведеного вище фрагмента в якості першого кроку можна зробити наступне:

```
#pragma omp parallel for shared(sum,a,n) private(ii,value)
for (ii = 0; ii < n; ii++) {
  value = some_complex_long_fuction(a[ii]);
  #pragma omp critical
  sum = sum + value;
}
```

або, що краще, можна використовувати оператор приведення, в результаті чого вийде

```
#pragma omp parallel for shared(a,n) private(ii)
reduction(+:sum)
for (ii = 0; ii < n; ii++) {
  sum = sum + some_complex_long_fuction(a[ii]);
}
```

12.1. З чого почати

Паралелізувати програму можна декількома способами. По-перше, слід визначити, чи потрібна паралелізації. Деякі алгоритми не годяться для

паралелізації. Якщо створюється новий проект, можна вибрати алгоритм, який може бути паралелізований. До початку спроб паралелізації дуже важливо переконатися, що код виконується правильно в послідовному режимі. Визначте тимчасові характеристики при послідовному виконанні, за якими можна буде прийняти рішення про доцільність паралелізації.

Скомпілюйте послідовну версію в декількох режимах оптимізації. Взагалі кажучи, компілятор може виконати набагато більш серйозну оптимізацію, ніж сам програміст.

Коли програма готова для паралелізації, можна застосувати ряд функцій і засобів компілятора, які допоможуть досягти мети. Вони коротко описані нижче.

12.2. Автоматична паралелізація

Можно спробувати застосувати ключ автоматичної паралелізації компіляторів Intel (`-Parallel`). Щоб використовувати цю опцію слід також дозволити оптимізація з допомогою ключів `-O2` або `-O3`. Довіривши паралелізації компілятору, програміст може паралелізувати програму без будь-яких зусиль зі свого боку. Крім того, автоматичний параллелізатор може допомогти визначити фрагменти коду, які можна паралелізувати за допомогою прагм OpenMP, і особливості, які можуть перешкодити паралелізації (наприклад, залежності між повтореннями тіла циклу).

12.3. Автовизначення області дії

Одним з найбільш поширених типів помилок у програмуванні з використанням OpenMP є помилки в визначенні області дії, коли область дії змінної вказується невірно, наприклад, змінна визначається як глобальна (приватна), у той час як її треба визначити як приватну (глобальну). Правильне визначення області дії змінної є непростою задачею і тому деякі компілятори мають спеціальні вбудовані засоби, що дозволяють автоматично визначити область дії деяких змінних.

Простий приклад: обчислення числа **π**. У послідовну програму на Фортрані вставлена **одна строчка**, і вона розпаралелена!

```
   program compute_pi
   parameter (n = 1000)
   integer i
   double precision w,x,sum,pi,f,a
   f(a) = 4.d0/(1.d0+a*a)
   w = 1.0d0/n
   sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x), SHARED(w), REDUCTION(+:sum)
   do i=1,n
       x = w*(i-0.5d0)
       sum = sum + f(x)
   enddo
```

```
pi = w*sum
print *,'pi = ',pi
stop
end
```

В даному прикладі змінна `x` є приватною, а `w` глобальною. Слід зазначити, що змінна `sum` також є приватною, різною в кожному потоці виконання. Після закінчення паралельного циклу обчислюється сума всіх цих змінних і результат зберігається в головному потоці.

Висновки

Іншою моделлю паралельного програмування є інтерфейс MPI (інтерфейс передачі повідомлень). На відміну від OpenMP, інтерфейс MPI породжує кілька процесів, які потім взаємодіють (по протоколу TCP/IP або іншим чином) для передачі повідомлень. Так як ці процеси не використовують загальне адресний простір, вони можуть виконуватися на віддалених комп'ютерах (або на кластері комп'ютерів). Важко сказати, що краще - OpenMP або MPI. У кожного з них є свої переваги і недоліки. Цікавіше те, що OpenMP може використовуватися спільно з MPI. Зазвичай MPI використовується для розподілу роботи між кількома комп'ютерами, а OpenMP потім використовується для паралелізації на одному комп'ютері з багатоядерним процесором.

Наведені приклади показують, що під час написання паралельних програм з використанням механізму передачі повідомлень алгоритми розв’язання навіть простих задач, таких як, наприклад, перемножування матриць, перестають бути тривіальними. І зовсім вже нетривіальною стає задача написання ефективних програм для розв’язання складніших задач лінійної алгебри. Складність програмування із застосуванням механізму передачі повідомлень довгий час залишалася основним стримувальним чинником на шляху широкого використання багатопроцесорних систем з розподіленою пам'яттю.

Останніми роками ситуація значно змінилася завдяки появі досить ефективних бібліотек підпрограм для розв’язання широкого кола задач [10]. Такі бібліотеки позбавляють програмістів від рутинної роботи з написання підпрограм для розв’язання стандартних задач числових методів і дозволяють концентруватися на конкретній предметній області. Проте використання цих бібліотек не позбавляє необхідності ясного розуміння принципів паралельного програмування і вимагає виконання досить об’ємної підготовчої роботи.

Завдання для лабораторних робіт

Завдання №1 **Налаштування MPI API в ОС Windows**

- Встановіть MPICH останньої версії, який можна загрузити з сайту http://www.mcs.anl.gov/research/projects/mpich2/
- Вивчіть правила виклику програми mpiexec
- Відкомпілюйте і виконайте тестову програму pi.c з різною кількість процесів на одному компютері.
- Опишіть процес перевірки встановленого програмного забезпечення.

Завдання №2 **Написання простих програм MPI**

- Напишіть програму на мові С, яка визначить кількість процесів, які виконуються, та імена компютерів, на яких вони виконуються.
- Обчислити обєм доступної памяті кожного процесу.
- Перевірити, чи можна ввести дані з зовнішнього файлу, розташованому на мережному драйві.

Завдання №3 **Передача повідомлень між процесами MPI**

- Напишіть програму на мові С, яка буде виконуватись двома процесами.
- Організуйте передачу одновимірного масиву з нульового процесу до першого.
- Обчисліть суму всіх елементів масиву і розішліть її всім процесам.
- Організуйте видачу результату на нульовому процесі.
- Використати фунціх MPI_Send та MPI_Recv.

Завдання №4 **Робота з матрицями**

- Напишіть програму на мові С, яка буде зберігати квадратну матрицю на M процесах. Розмір матриці N обчисліть таким чином, щоб кожний процес зберігав K рядків матриці і N = k*M, де M - число запущених процесів.
- Напишіть підпрограми для перемноження матриці на вектор і навпаки. Результуючий вектор розішліть на всі процеси.
- Обчисліть час виконання даних процедур. Упевніться, що він буде приблизно однаковий для обох випадків.

Завдання №5 **Колективні операції**

- Організуйте розсилку інформації від одного процесса всім остальным членам деякої області зв'язку (MPI_Bcast).

- Зберіть (MPI_Gather) розподілені по процессам массиви в один массив зі збереженням його в адресном просторі виділеного (root) процесса (MPI_Gather).
- Проведіть глобальні обчислювальні операції (sum, min, max и min) над даними, які розташовані в адресних просторах різних процесів: зі збереженням результата в адресном просторі одного процесса (MPI_Reduce);
- Організуйте видачу відповідних результатів на нульовому процесі.

Завдання №6 **Використання OpenMP**

- Напишіть програму на мові С для обчислення скалярного добутку двох векторів типу double. Розмір векторів виберіть таким чином, щоб вони займали половину оперативної памяті.
- Модифікуйте даний алгоритм з допомогою прагм OpenMP і виконайте програму з використанням різної кількості нитей.
- Обчисліть час виконання і проаналізуйте отримані результати.

Список використаної літератури та електронних ресурсів

1. Бройдо В.Л., Ильина О.П. Архитектура ЭВМ и систем. СПб.:Питер, 2005, 720с.
2. Богачев К.Ю. Основы параллельного программирования. – М.: БИНОМ. Лаб. знаний, 2003. – 342 с.
3. Воеводин, В. В. Параллельные вичисления [Текст]/ В. В. Воєводин, Вл. В. Воеводин - СПб.: БХВ-Петербург, 2002. - 600 с.
4. Гергель, В.П. Основы параллельных вычислений для многопроцессорных вычислительных систем [Текст] / В.П. Гергель, Р.Г. Стронгин – 2-е изд. - Н.Новгород, ННГУ, 2003. – 356с.
5. Гэри, М. Вычислительные машины и труднорешаемые задачи [Текст] / М.Гэри, Д.Джонсон - М.: Мир, 1982. - 416 с.
6. Жуков І., Корочкін О. Паралельні та розподілені обчислення – К.:Корнійчук, 2005. – 226 с.
7. Королев Л.Н. Архитектура ЭВМ. М.: Научный мир, 2005, 272 с.
8. C-DVM - язык разработки мобильных параллельных программ [Текст] / Н.А.Коновалов, В. А. Крюков, А.А.Погребцов, Ю.Л. Сазанов – Программирование, 1999. - С. 20-28.
9. Немнюгин, С. Параллельное программирование для многопроцессорных вычислительных систем [Текст] / С.Немнюгин, О. Стесик - СПб.: БХВ-Петербург, 2002. – 286с.
10. Эндрюс Г. Основы многопоточного, параллельного и распределенного програмирования.: Пер. с англ. – М.: Изд. Дом «Вильямс», 2003. – 512с.
11. Amdahl, G.Validity of the single-processor approach to achieving large-scale computing capabilities. / [Text] / G. Amdahl //Proc. 1967 AFIPS Conf., AFIPS Press. - 1967. - Vol. 30. - P. 483.
12. Grama, A. Introduction to Parallel Computing [Text] / A.Grama, A.Gupta, V.Kumar - Harlow, Addison-Wesley, 2003/ - 452pp.
13. MPI: The Complete Reference [Electronic resource] – Режим доступу: http://www.netlib.org/ papers/mpi-book/mpi-book.html
14. Foster, Ian. Designing and Building Parallel Programs [Text] / Ian Foster - Addison-Wesley, Inc. Boston, MA, 1995 - 381pp.
15. ScaLAPACK Users' Guide. [Electronic resource] – Режим доступу: http://www.netlib.org/scalapack/slug/
16. Quinn M.J. Parallel Programming in C with MPI and OpenMP. McGrawHill, 2004, 544 p.
17. OpenMP Application Program Interface. Version 3.0 May 2008