Міністерство освіти і науки україни
Центральноукраїнський національний технічний університет

Минайленко Р.М.

Паралельні та розподілені обчислення

Навчальний посібник

Кропивницький
2021

УДК 004.75
ББК 32.973.2
М 61

Рекомендовано Вченою радою Центральноукраїнського національного технічного університету, протокол № 7 від 2 березня 2021 року

Рецензенти:

доктор технічних наук, професор Одарченко Р.С.
доктор технічних наук, професор Євсеєв С.П.

Минайленко Р.М.

М 61 Паралельні та розподілені обчислення: навч. посіб. — Кропивницький: Видавець Лисенко В. Ф., 2021. — 153 с.

В посібнику викладено основні поняття та концепції із області паралельних та розподілених обчислень.

Матеріал посібника супроводжується великою кількістю прикладів, що демонструють застосування методів та алгоритмів для вирішення конкретних завдань.

Навчальний посібник призначений для студентів, які навчаються за спеціальністю «Комп'ютерна інженерія», а також може бути корисним для спеціалістів в галузі інформатики, обчислювальної техніки та програмування в якості вступного курсу в проблематику паралельних та розподілених обчислень.

ББК 32.973.2

Зміст

ВСТУП

На теперішній час розвиток інформаційних та телекомунікаційних технологій знаходиться на тій стадії, коли в розподіленому інформаційно-телекомунікаційному середовищі все більшого значення набуває не тільки необхідність доступу та обміну інформацією, але й виконання різноманітних видів аналізу та обробки цієї інформації.

Широке впровадження комп'ютерів в усі види діяльності, постійне нарощування їх обчислювальної потужності, використання комп'ютерних мереж різного масштабу вимагає використання значного обсягу високопродуктивних розподілених обчислень, що в свою чергу призводить до дефіциту обчислювальних ресурсів при виконанні різноманітних обчислювальних процесів.

Ефективним шляхом вирішення цих проблем є використання паралельних та розподілених обчислень.

Метою навчального посібника є отримання студентами теоретичних і практичних знань в області паралельних та розподілених обчислень, оволодіння базовими концепціями програмування в рамках парадигм паралельного та розподіленого обчислення.

В посібнику викладено основні поняття та концепції із області паралельних та розподілених обчислень. Розглянуто технології розподілених систем їх переваги та недоліки. Приділено увагу моделюванню та аналізу паралельних обчислень. Наведено способи оцінювання максимально досяжного паралелізму та принципи розробки паралельних алгоритмів. Також велику увагу присвячено паралельному програмуванню на основі MPI. Матеріал посібника супроводжується прикладами, що демонструють застосування методів та алгоритмів для вирішення конкретних завдань.

Посібник може бути корисним для спеціалістів в галузі інформатики, обчислювальної техніки та програмування в якості вступного курсу в проблематику паралельних та розподілених обчислень.

1. ТЕХНОЛОГІЇ ПОБУДОВИ РОЗПОДІЛЕНИХ ОБЄ'КТНИХ СИСТЕМ

1.1. Розподілені об'єктні технології в інформаційних системах

За умови пошуку покращених виробничих процесів та швидкого розвитку обчислювальної техніки і прикладного програмного забезпечення, має місце швидке зростання складності інформаційних систем. З'являються нові напрямки, технології та архітектурні рішення побудови інформаційних систем (ІС). Здійснюється перехід до динамічної, гнучкої структури ІС, яка базується на розподілених системах обробки інформації. Сучасний рівень розвитку суспільства виводить індустрію інформаційних технологій (ІТ), на провідне і стратегічне місце, в якому зосереджуються величезні інтелектуальні та фінансові ресурси.

Складність створення, модифікації, супроводження та інтеграції ІС, особливість розв'язуваних з їх використанням задач, визначає такі класи ІС:

- малі;
- середні;
- великі (корпоративні на рівні загальнодержавних установ).
- До малих ІС відносяться системи, рівня невеликих підприємств, ознаками яких є:
- нетривалий життєвий цикл;
- орієнтація на масове використання;
- невисока ціна;
- практична відсутність засобів аналітичної обробки даних;
- відсутність можливості незначної модифікації без участі розробників;
- використання, головним чином, настільних СУБД (Clarion, FoxPro, Clipper, Paradox, Access та ін.);
- однорідність апаратного та системного забезпечення (недорогі персональні комп'ютери (ПК));

- практична відсутність засобів забезпечення безпеки;
- та ін.

Ознаками класу середніх ІС є:

- тривалий життєвий цикл і можливість зростання до великих ІС;
- наявність аналітичної обробки даних;
- наявність штату співробітників для здійснення функцій адміністрування апаратних та програмних засобів;
- наявність засобів забезпечення безпеки;
- тісна взаємодія з установами-розробниками програмного забезпечення з питань супроводження компонентів ІС;
- та ін.

До характерних ознак великих (корпоративних) ІС відносяться:

- тривалий життєвий цикл;
- міграція успадкованих систем
- різноманітність використовуваного апаратного забезпечення з меншим, порівняно із створюваною системою, життєвим циклом;
- різноманітність використовуваного програмного забезпечення (ПЗ);
- масштабність та складність розв’язуваних задач;
- перетин множини різних предметних галузей;
- орієнтація на аналітичну обробку даних;
- територіальний розподіл;
- та ін.

На даний час обговорюються питання стосовно опису продуктів, технологій та методологій створення малих та середніх ІС. Разом з тим, технології та методології побудови великих ІС, які об’єднують у собі множину локальних ІС, практично не розглядаються і не обговорюються. Це призводить до того, що, як технології створення великої ІС, вибираються ті, які з самого початку не це не розраховані. З цієї причини проекти, що реалізуються, не отримують належного розвитку.

Сучасний рівень розвитку суспільства визначає індустрію ІТ, як провідний і стратегічний напрямок зосередження інтелектуальних та фінансових ресурсів. Інформація та інструменти управління нею (програмні продукти різного функціонального призначення) набули статусу інформаційних ресурсів (ІР). Останні концентруються в рамках ІС. Об'єднання ресурсів на основі інформаційно-комунікаційної взаємодії ІС виводить їх на рівень корпоративних інформаційних ресурсів. Це об'єднання часто називають Єдиним Інформаційним Простором (ЄІП). Реалізація ЄІП на рівні держави, корпорації, підприємства можлива у разі створення з подальшим дотриманням стандарту на взаємодію між собою ІС, та їх окремих додатків (рис. 1.1)

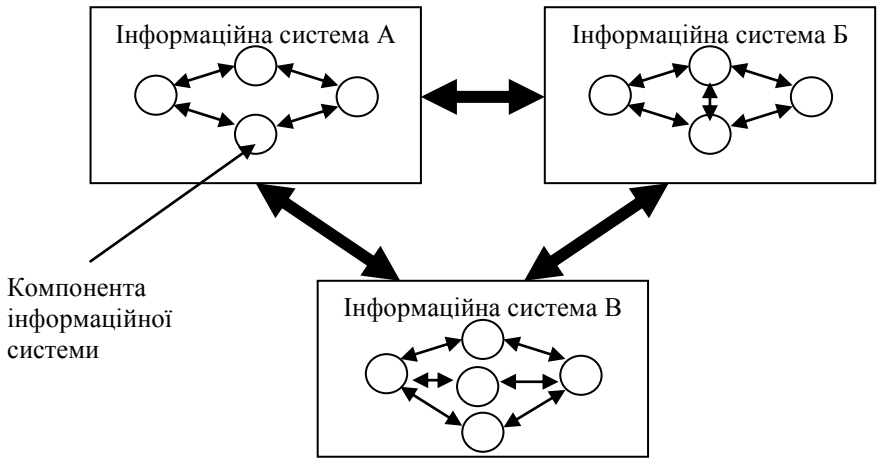


Рисунок 1.1 - Корпоративні інформаційні ресурси

У ряді випадків під ІР розуміють тільки дані, коли розв'язок проблеми ЄІП зводиться до Єдиного Простору Даних (ЄПД), рис. 1.2, а ІС виступають як клієнт та сервер і взаємодіють один з іншим у відповідності із схемою, наведеною на рис. 1.3.

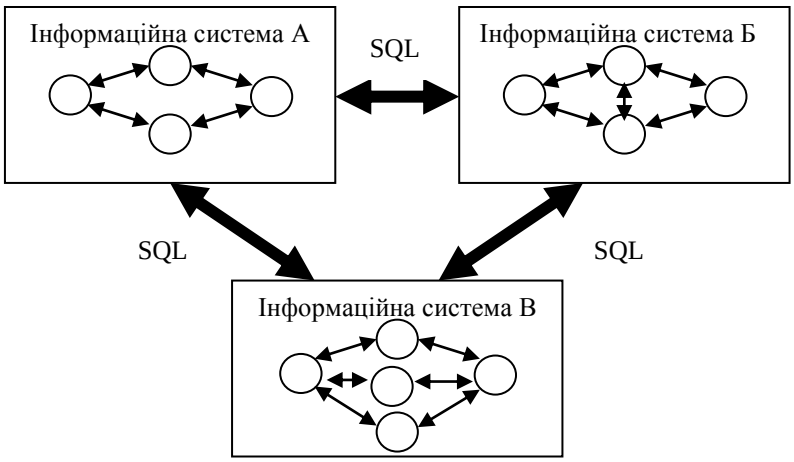


Рисунок 1.2 - Єдиний простір даних (ЄПД)

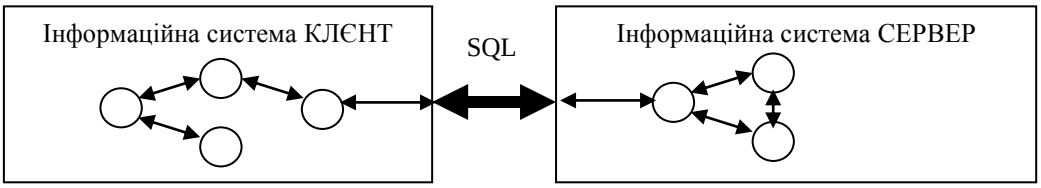


Рисунок 1.3 - Архітектура доступу до віддалених даних

Інформаційна система-клієнт (ІСК) надсилає інформаційній системі-сервер (ІСС) запит, отримуючи, як результат, дані, що підлягають подальшій обробці. Як запит, використовують мову SQL - стандарт поводження з реляційними системами управління базами даних. Доступ до віддалених баз даних (БД) у більшості випадків здійснюється з використанням продуктів, які підтримують протоколи ODBC (Open Data Base Connectivity), та JDBC (Java Data Base Connectivity), або використовуються шлюзи, які постачаються виробниками СУБД або третіми фірмами-виробниками. При побудові єдиного простору даних використовується архітектура доступу до віддалених даних, що є аналогом дворівневої архітектури клієнт-сервер. Ця архітектура припускає реалізацію на стороні клієнта як функцій введення та відображення даних, так і прикладних функцій додатку, тобто методів обробки даних. Клієнт направляє запити до сервера, який обробляє їх і повертає клієнту результат, оформлений як блок даних.

Описаному сценарію взаємодії систем притаманні всі недоліки, характерні для дворівневої архітектури клієнт-сервер:

- слід знати з боку ІСК особливості використовуваної СУБД та структуру віддаленої бази даних (БД) ІСС, що знижує рівень безпеки всієї системи в цілому;
- ускладнено супроводження та модифікація тих додатків ІСК, які спілкуються з БД ІСС, оскільки будь-яка зміна схеми віддаленої БД на стороні ІСС призводить до зміни додатків в ІСК, що ускладнює обслуговування, оновлення або заміну додатків, встановлених на великій кількості комп'ютерів;
- значно ускладнюється адміністрування БД ІСС, яке включає в себе управління правами доступу користувачів ІСК.

Істотним недоліком розглянутого сценарію є дублювання додатків ІСС в ІСК, що призводить до неефективного використання взаємодіючих ресурсів ІС. Зростання популярності глобальної мережі Internet та технології WWW останнім часом викликає підвищений інтерес до них з боку розробників корпоративних ІС. На самому початку WWW створювався як засіб, який надає графічний інтерфейс в Internet і спрощує доступ до інформації, розподіленої по мільйонам комп'ютерів у всьому світі. Основними компонентами були сторінки, вузли, браузери та сервери Web. Користувачам була надана можливість навігації по Internet з використанням технології гіпертексту, підтримуваної протоколами HTTP (Hypertext Transfer Protocol) та стандартом мови HTML (Hypertext Markup Language).

Додаток CGI (Common Gateway Interface) вирішив проблему обміну інформацією між сервером Web та програмами типу бази даних, які не можуть безпосередньо обмінюватися даними з браузерами Web. В результаті з'явилася можливість реалізації інтерактивної взаємодії кінцевого користувача з програмами Web - серверу, які обробляли інформацію, введену користувачем в браузері, і як в результаті повертали сформовану HTML -

сторінку. Багато з існуючих рішень доступні в середовищі Internet і базуються на такому підході.

Поява мови Java надала розробникам ІС абсолютно нові технологічні рішення побудови додатків у середовищі Internet. Проте не можна розглядати технологію Java тільки як частину технології WWW, оскільки Java дає змогу розв'язувати задачі більш широкого класу, порівняно з технологіями, які базуються на мові HTML, протоколи HTTP та CGI. Можливості, які надаються WWW - технологією розширили спектр рішень, якими керуються проектувальники при побудові ІС. Проте виникає питання: що собою представляють взаємодіючі системи ІС, які базуються на певній технології, і чи здатні вони розв'язати проблему ЄІП. Таке твердження пов'язане з тим, що в процесі розгляду взаємодії інформаційних систем, ІСК з браузером виступає в ролі компоненти зображення, а ІСС з WWW - сервером та додатками виступає як компонента, яка реалізує функціональну логіку та доступ до даних, що відповідає дворівневій архітектурі з інтелектуальним сервером (рис. 1.4). WWW - технологія здатна покращити ситуацію з імпортом/експортом даних між ІСК та ІСС, але є недоліки, що притаманні дворівневій архітектурі з інтелектуальним сервером.

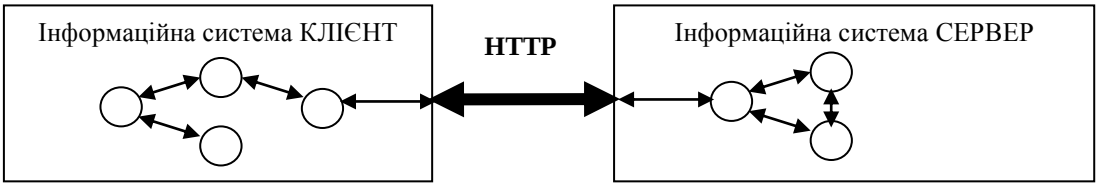


Рисунок 1.4 - Архітектура з інтелектуальним сервером

Одним з недоліків такої системи є відсутність можливості реалізації процесу обробки даних, які надаються WWW - сервером, збоку ІСК, оскільки остання отримує інформацію від ІСС у вигляді HTML - сторінок, що унеможливлює організацію процесу обробки отриманих даних компонентами ІСК. Це призводить до відсутності потрібної ефективності використання

обчислювальних ресурсів ІС. Разом з тим, гостро постає проблема підтримання безпеки системи в цілому, яка не має цілісного розв'язку у середовищі Internet, що неприпустимо для організацій, які висувають підвищені вимоги до безпеки. Крім того істотно ускладнюється адміністрування ресурсів ІСС, яке включає в себе управління правами доступу користувачів ІСС.

В концепції єдиного інформаційного простору потрібно передбачити, що інформаційні ресурси ІС, у відношенні до неї, виступають і як дані, і як різні додатки ІС. Тоді у кожній з ІС частина методів обробки даних реалізується як додатки, доступні з інших ІС, зокрема в разі взаємодії двох ІС, перша - використовується сервісами, які надаються другою, в результаті чого вона отримує вже оброблені дані, які можна піддати подальшій обробці компонентами першої ІС. Такий підхід відповідає розподіленій, одноранговій архітектурі взаємодії. Згідно цієї архітектури, будь-які додатки з різних ІС можуть виступати як в ролі клієнта, так і в ролі сервера по відношенню одна до одної, сумісно розв'язуючи ті чи інші задачі. Такий підхід мінімізує дублювання додатків. Розподіл додатків по різним ІС дає змогу досягти оптимального балансу завантаження додатків та апаратних засобів, що призведе до ефективного використання інформаційних ресурсів систем в цілому.

Знання схеми бази даних (БД) необхідно тільки тому додатку, який обробляє дані з цієї БД. Використання ІСС сервісів, які надаються ІС - сервером і реалізують методи обробки даних, дає змогу розв'язати проблему зміни схеми віддаленої бази даних. Статичність інтерфейсів компонентів. які надають ІСС набір сервісів, досягається застосуванням методологій об'єктно-орієнтованого аналізу та проектування, розподілених об'єктних технологій (CORBA, Java, DCOM) на різних етапах створення ІС. Більшість використовуваних ІС є додатками до дворівневої архітектури клієнт-сервер і як засіб спілкування клієнта та сервера часто використовуються неповністю стандартизовані механізми процедур. Специфіка їх реалізації,

невідокремлена від ядра системи управління БД призводить до необхідності наявності додаткових обчислювальних ресурсів на стороні сервера.

В разі збільшення виконуваних сервером робіт системи в дворівневій архітектурі клієнт-сервер стають все більше схожими на великі ЕОМ (мейнфрейми), а структури оброблюваних ними даних та способи їх представлення малодоступні для використання разом з іншими додатками. Як правило взаємодія розглянутих клієнт-сервер організується засобами СУБД, що перевантажує серверну частину. Разом з тим, сучасні технології дають змогу створити інтегроване середовище в рамках ІС, та в рамках концепції ЄІП. Таке середовище:

- не залежить від апаратних та системних програмних засобів;
- спирається на міжнародні та промислові стандарти;
- дає змогу розробити єдину інформаційну модель представлення підприємства як сукупності керованих ресурсів та потоків діяльності, настроєну на реалізацію правил управління колективною діяльностю кожного конкретного підприємства;
- забезпечує розширюваність системи, тобто простоту та легкість додавання нових компонентів в існуючі ІС;
- дає змогу інтегрувати старі додатки (legacy applications) в нові ІС;
- припускає природну інтегрованість створюваних ІС, що гарантує життєздатність та еволюційний розвиток;
- дає змогу накопичувати, тиражувати та розвивати формалізовані знання спеціалістів;
- істотно знижують сумарні витрати на створення ІС.

Технології RMI, CORBA, DCOM

Розглянемо три різні технології, які підтримують концепцію розподілених об’єктних систем: RMI, CORBA та DCOM.

Архітектура RMI (Remote Method Invocation - виклик віддаленого методу), яка інтегрована JDK1.1, є продуктом компанії Java Soft і реалізує розподілену модель. RMI та дає змогу клієнтським і серверним додаткам

через мережу викликати методи клієнтів/серверів, які виконуються Java Virtual Machine. Незважаючи на те, що RMI вважається "легкою" та менш потужною, порівняно з CORBA та DCOM, тим не менше, вона має ряд унікальних можливостей, оскільки має розподілене, автоматичне управління об’єктами та можливість пересилати самі об’єкти від машини до машини. На рис. 1.5 зображені основні компоненти архітектури RMI.

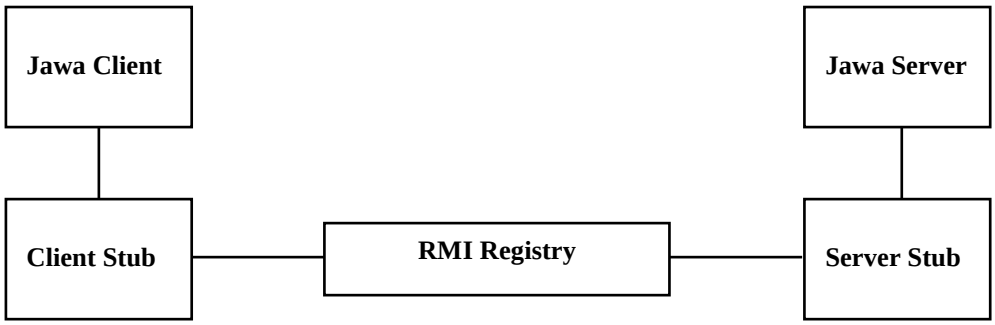


Рисунок 1.5 - Модель RMI

Client Stub (перехідник для клієнта) та Server Stub (перехідник для сервера) породжені від одного інтерфейсу, але різниця між ними полягає в тому, що client stub служить для під’єднання до RMI Registry, а server stub використовується для зв’язку безпосередньо з функціями сервера.

Технологія CORBA (Common Object Request Broker Architecture), яка розробляється OMG (Object Management Group) з 1990 року - це архітектура з брокером потрібних спільних об’єктів.

Dynamic Invocation Interface (DII): надає клієнту змогу знаходити сервери і викликати їх під час роботи системи. IDL Stubs: визначає, яким чином клієнт здійснює виклик сервера. ORB Interface: спільні для клієнта та для сервера сервіси. IDL Skeleton Interface: спільні інтерфейси для об’єктів, незалежно від їх типу, які не були визначені в IDL Object Adapter: здійснюють комунікаційну взаємодію між об’єктом та ORB.

На рис. 1.6 наведена основна структура CORBA 2.0 ORB:

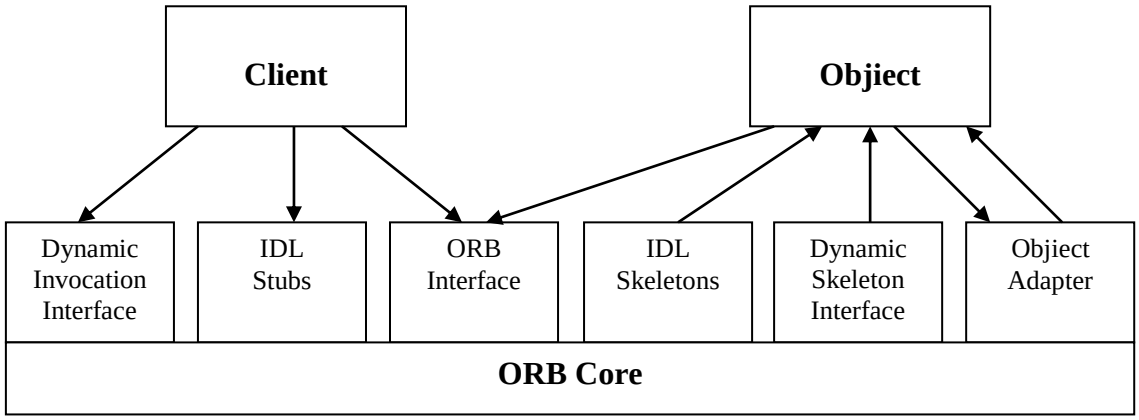


Рисунок 1.6 – Структура ORB (CORBA 2.0)

Технологія DCOM (Distributed Component Object Model), рис. 1.7, була розроблена компанією Microsoft як розв'язок для розподілених систем в 1996 році. DCOM є головним конкурентом CORBA, хоч і контролюється не Microsoft, а групою TOG (The Open Group), аналогічною OMG. DCOM є розширенням архітектури COM до рівня мережних додатків.

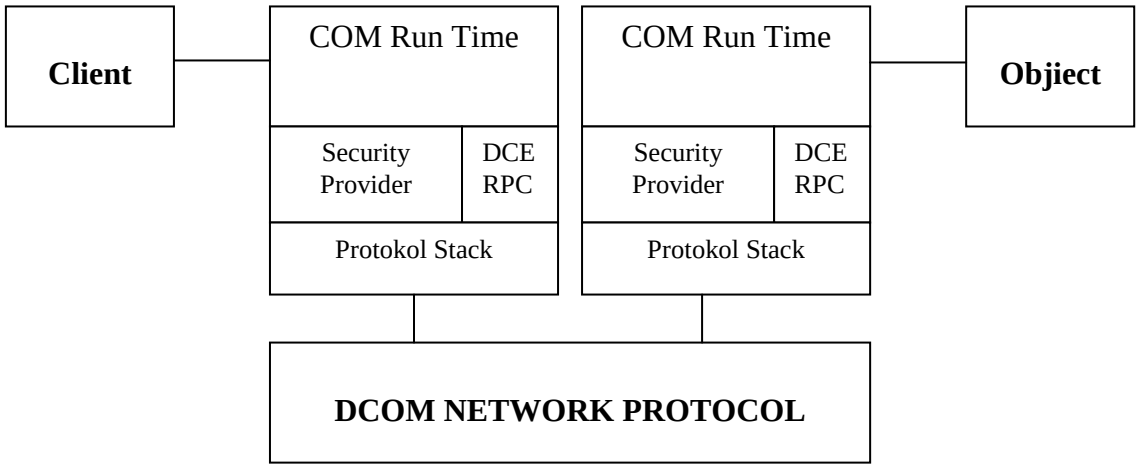


Рисунок 1.7 - Архітектура DCOM

У кожної з трьох розглянутих технологій (RMI, CORBA, DCOM) є свої унікальні особливості, які багато в чому характеризують можливість або неможливість її застосування для розв'язку конкретної поставленої задачі.

2 ПЕРЕВАГИ ТА НЕДОЛІКИ ТЕХНОЛОГІЙ

Технологія DCOM.

Переваги: мовна незалежність; динамічний/статичний виклик; динамічне находження об’єктів; масштабованість; відкритий стандарт (контроль з боку TOG).

Недоліки: складність реалізації; залежність від платформи; немає перевірки безпеки на рівні виконання ActiveX компонент. Крім того DCOM є лише частковим рішенням проблеми розподілених об’єктних систем. Це рішення добре підходить до Microsoft - орієнтованих середовищ. Як тільки в системі виникає необхідність працювати з архітектурою, яка відрізняється від Windows, DCOM перестає бути оптимальним рішенням проблеми.

Технологія RMI.

Переваги: швидке та просте створення; Java - оптимізація; динамічне завантаження компонентів - перехідників; можливість передачі об’єктів за значенням; внутрішня безпека.

Недоліки: підтримка тільки однієї мови - Java; свій власний, не IIOP - сумісний протокол взаємодії; труднощі інтегрування з існуючими додатками; погана масштабованість. Завдяки своїй Java - моделі, яка є легко використовуваною, RMI є самим простим і самим швидким способом створення розподілених систем. RMI є хорошим вибором для створення RAD - компонент та невеликих додатків на мові Java. Технологія RMI не є такою потужною, як DCOM або CORBA, зокрема RMI використовує свій (не CORBA/ IIOP) сумісний протокол передачі JRMP і може взаємодіяти лише з іншими Java об’єктами. Підтримка тільки однієї мови робить неможливою взаємодію з об’єктами, написаними не на мові Java. Тим самим, роль RMI у створенні великих, масштабованих промислових систем, знижується.

Технології CORBA.

Переваги: платформна незалежність; мовна незалежність; динамічні виклики; динамічне виявлення об’єктів; можливість масштабування;

CORBA-сервіси; широка індустріальна підтримка.

Недоліки: відсутність передачі параметрів "за значенням"; відсутність динамічного завантаження компонент-перехідників. До основних переваг CORBA можна віднести міжмовну та міжплатформенну підтримку. Незважаючи те, що CORBA - сервіси віднесені до переваг технології CORBA, їх в рівній мірі можна одночасно віднести до недоліків CORBA, внаслідок повної відсутності їх реалізації.

Технологія CORBA відноситься до найефективніших, придатних для корпоративних проектів, це технологія розподілених об’єктів. Причиною цього є те, що обидві технології CORBA та DCOM надзвичайно схожі за своєю функціональністю та за своїми можливостями (багатомовна підтримка, динамічний виклик, масштабованість та ін.) у DCOM відсутній важливий критичний елемент - мультиплатформна підтримка. Одного факту, що DCOM не підтримує цілком міжплатформену переносимість, достатньо, щоб не розглядати її як повноцінне, закінчене рішення.

Коло виробників продуктів, які підтримують дану технологію, ширше, порівняно аналогічного кола DCOM. Тобто виявляється, що саме CORBA є технологією, яка повністю призначена для промислових, відкритих, розподілених об’єктних систем.

Технологія CORBA

Наприкінці 1980 - х і на початку 1990 - х років багато провідних фірм-розробників займалися пошуком технологій, які принесли б відчутну користь на ринку комп’ютерних розробок. Такою технологією виявилася область розподілених комп’ютерних систем. Необхідно було розробити єдину структуру, яка б дала змогу здійснити повторне використання та інтеграцію коду, що важливо для розробників. Вартість повторного використання коду та інтеграція коду була високою, проте ніхто з розробників поодинці не міг втілити в реальність широко використовуваний, мовно-незалежний стандарт, який би включав в себе підтримку складних багатозв'язних додатків. У травні 1989 р. була сформована OMG (Object Management Group). OMG має велику

кількість користувачів (до OMG входять практично всі найбільші виробники програмного забезпечення (ПЗ), за виключенням Microsoft). Задачею консорціуму OMG є визначення набору специфікацій, які дають змогу будувати інтероперабельні інформаційні системи. Специфікація OMG - The Common Object Request Broker Architecture (CORBA) є індустріальним стандартом, який описує високорівневі засоби підтримки взаємодії об’єктів в розподілених гетерогенних середовищах. CORBA специфікує інфраструктуру взаємодії компонент (об’єктів) на представницькому рівні і на рівні додатків моделі OSI. Вона дає змогу розглядати всі додатки в розподіленій системі як об’єкти. Причому об’єкти можуть одночасно відігравати роль клієнта та сервера. Об’єкти-сервери зазвичай називають "реалізацією об’єктів". Практика показує, що більшість об’єктів одночасно виконують роль клієнтів і серверів, по черзі викликаючи методи на інших об’єктах і відповідають на зовнішні виклики. Використовуючи CORBA, тим самим, є можливість будувати більш гнучкі системи, ніж системи клієнт-сервер, засновані на дворівневій і трирівневій архітектурі.

Мова OMG IDL (Interface Definition Language - Мова Опису Інтерфейсу) являє собою технологічно незалежний синтаксис для опису інтерфейсу об’єктів. При описі програмної архітектури, OMG IDL добре використовується для окреслювання меж об’єкту, які визначають його поведінку у відношенні до інших компонентів ІС. OMG IDL дає змогу описати інтерфейси, які мають різні методи та атрибути. Мова також підтримує дослідження інтерфейсів, що необхідно для повторного використання об’єктів з можливістю їх розширення або конкретизації. IDL є чисто декларативною мовою, тобто вона не містить ніякої реалізації. IDL - специфікації можуть бути відкомпільовані (відображені) в файли заголовку та спеціальні прототипи серверів, які можуть використовуватися безпосередньо програмістом. Тобто IDL це визначені методи, що можуть бути написані, і далі виконані, на будь-якій мові, для якої існує відображення з IDL. До таких мов відносяться C, C++, SmallTalk, Java та Ada.

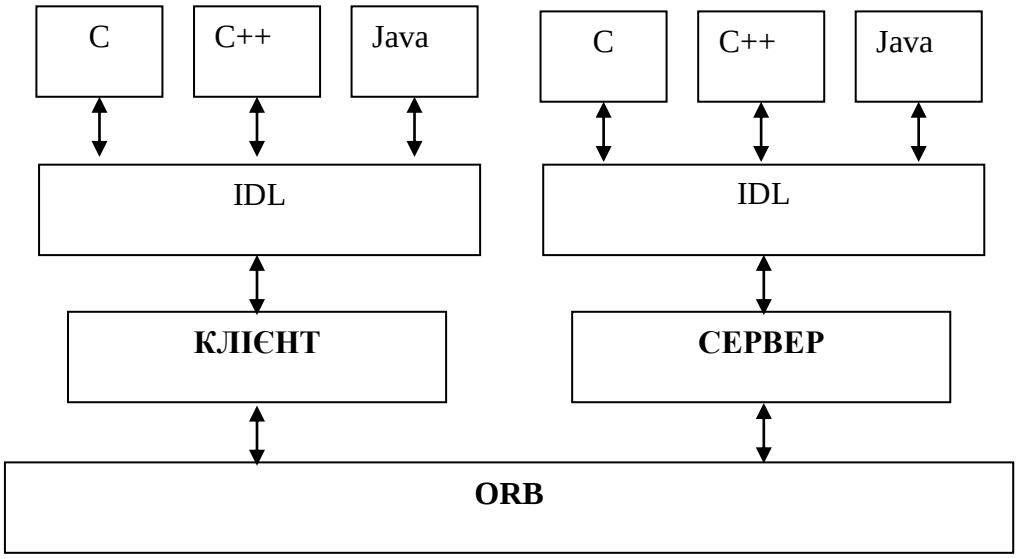


Рисунок 2.1 - CORBA IDL відображення в моделі Клієнт/Сервер

З використанням IDL можна описати також атрибути компоненти, і батьківські класи, які вона наслідує, і викликувані виключення, і методи, які визначають інтерфейс, причому з описом вхідних та вихідних параметрів. Структура CORBA IDL файлу має такий вигляд:

```
module <identifier> {
<type declarations>;
<constant declarations>;
<exception declarations>;
interface <identifier> [:<inheritance>] {
<type declarations>;
<constant declarations>;
<attribute declarations>;
<exception declarations>;
[<op_type>]<identifier>(<parameters>)
[raises exception] [context]
.
.
[<op_type>]<identifier>(<parameters>)
[raises exception] [context]
.
.
}
interface <identifier> [:<inheritance>]
.
.
}
```

Репозитарій Інтерфейсів (Interface Repository), який містить означення інтерфейсів на IDL, дає змогу бачити інтерфейси доступних серверів в мережі і програмувати їх використання в програмах-клієнтах.

Object Management Architecture. В 1990 році OMG вперше опублікувала документ Object Management Architecture Guide (OMA Guide). Він був відкоригований у вересні 1992 р. Деталі Common Facilities (Спільні засоби) були додані в січні 1995 р. На рис. 2.2 показані чотири основні елементи цієї архітектури:

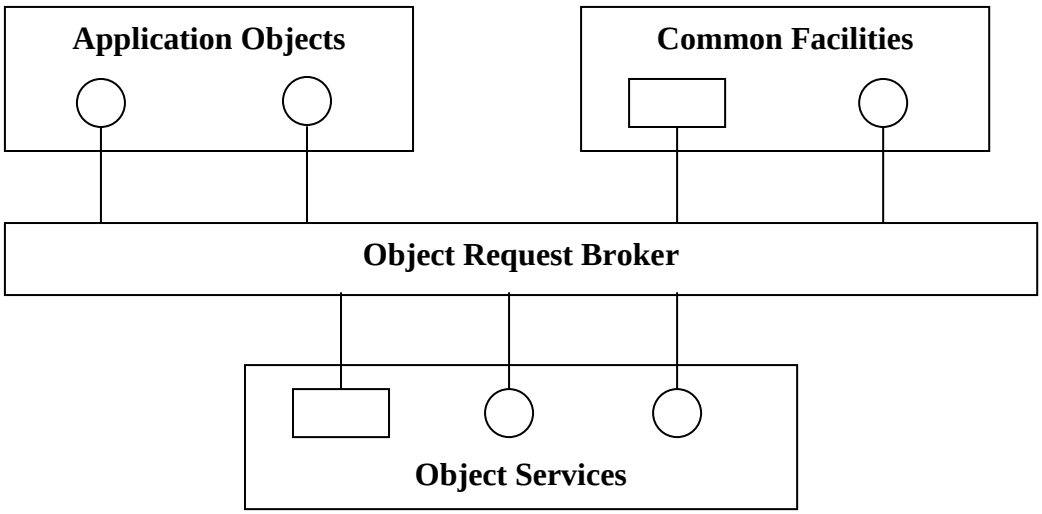


Рисунок 2.2 - OMG’s Object Management Architecture

1. Object Request Broker визначає об’єктну шину CORBA.
2. Common Object Services являють собою набір служб, із об’єктивними інтерфейсами, які забезпечують підтримку базових функцій об’єктів.
3. Common Facilities утворюють набір класів та об’єктів, які підтримують корисні в багатьох системах функції. Прикладні об’єкти представляють прикладні системи кінцевих користувачів і забезпечують функції, які є унікальними для даної прикладної системи.
4. Application Objects - це прикладні бізнес-об’єкти і додатки, які є основними споживачами всієї CORBA інфраструктури.

ORB, Object Request Broker (брокер об'єктних запитів) - це об'єктна шина, яка дає змогу об'єктам напряму виробляти і відповідати на запити інших об'єктів, розташованих як локально (на одному комп'ютері, але в різних процесах), так і віддалено. Клієнта не цікавлять комунікаційні та інші механізми, з використанням яких відбувається взаємодія між об'єктами, виклик і збереження серверних компонентів. CORBA - специфікації зачіпають лише IDL, відображення в інші мови, APIs для взаємодії з CORBA та сервіси, які надаються ORB. CORBA ORB не надає широкого набору розподілених middleware (проміжних) послуг. Шина ORB дає змогу об'єктам знаходити один одного прямо в процесі роботи і викликати один у іншого різноманітні служби. Вона є набагато тоншою системою, порівняно з іншими клієнт/сервер middleware - системами, такими, як RPC (Remote Procedure Calls) чи MOM (Message-Oriented Middleware).

Шлях від RPC до ORB. Стосовно відмінності механізму викликів CORBA та RPC можна відмітити наступне - ці механізми схожі, але між ними є серйозні відмінності. З використанням RPC можна викликати певну функцію, а з використанням ORB можна викликати метод у певного об'єкта. Різні об'єкти класів можуть по-різному відповідати на виклик одного і того ж методу. Оскільки кожний об'єкт керує своєю власною інформацією, то метод буде використаний для певних конкретних даних. У випадку RPC, буде виконана лише якась конкретна частина коду сервера, яка і взаємодіє з даними сервера. Всі функції з однаковими іменами будуть виконані абсолютно однаково. В RPC відсутня конкретизація викликів, в тому розумінні, в якому це відбувається в ORB. В ORB всі виклики функцій здійснюються до конкретних об'єктів, тим самим, результати цих функцій можуть бути абсолютно різними. Виклики функцій обробляються в специфічній для кожного окремого об'єкта формі.

Переваги ORB. Теоретично CORBA визначається як краща клієнт/сервер middleware - система, але на практиці її спроможність буде залежати від того, наскільки задовільні продукти, що її реалізують. До

основних комерційних ORB відносяться Orbix від фірми IONA Technologies; DSOM від IBM; Object Broker від Digital; JOE від Sun; Visibroker від Visigenic та Netscape; ORB+ від HP. Переваги CORBA ORB:

- статистичні та динамічні виклики методів - CORBA ORB надає можливість або статично визначити виклики методів прямо під час компіляції, або знаходити їх динамічно, але вже під час роботи програми.

- відображення та мови високого рівня - CORBA ORB дає змогу викликати методи у серверних компонент, використовуючи будь-який з певного списку мов високого рівня: C, C++, SmallTalk, Java, Ada. Абсолютно неважливо, на якій мові написані об'єкти - CORBA відділяє інтерфейси від реалізації і надає мовнонезалежні типи даних, що дає змогу здійснювати виклик методів, минаючи межі якоїсь мови програмування та конкретної операційної системи.

- система, що самоописується - з використанням своїх метаданих, CORBA дає змогу описувати інтерфейс будь-якого сервера, відомого системі. Кожна CORBA ORB повинна підтримувати Репозитарій Інтерфейсів, який зберігає необхідну інформацію, що описує синтаксис інтерфейсів, підтримуваних серверами. В своїй роботі клієнти використовують ці метадані для здійснення викликів до серверів.

- прозорість - ORB може виконуватися на портативному комп'ютері. ORB може здійснювати міжоб'єктну взаємодію для одного процесу і для кількох процесів, що виконуються на одній машині, а також для процесів, виконання яких відбувається в мережі, з різними операційними системами. Реалізація цих взаємодій ніяк не впливає на самі об'єкти. При використанні технології CORBA, розробник не має турбуватися про розташування серверів, запуск (активування) об'єктів, вирівнювання розміру змінних в залежності від платформи та операційної системи, та як здійснюється передача повідомлень. Рішення всіх цих задач бере на себе продукт, який підтримує стандарт CORBA.

- вбудована безпека - всі свої запити ORB доповнює певною контекстною інформацією, яка забезпечує збереження даних.

- поліморфізм при виклику методів - ORB не просто викликає віддалений метод - ORB викликає метод на віддаленому об’єкті. Тобто виконання одних і тих же функцій на різних об’єктах призведуть до різних дій, в залежності від типу об’єкта.

Object Services. CORBA Object Services являє собою набір сервісів системного рівня, які зображуються у вигляді компонентів з певними визначеними IDL - інтерфейсами. Ці компоненти доповнюють функціональність ORB, їх можна використати для створення, найменування компонентів, та ін. OMG визначає 14 стандартних сервісів. Практично всі комерційні ORB не підтримують жодного із сервісів (крім Visibroker).

Common Facilities. Common Facilities (спільні засоби) заповнюють простір між ORB та об’єктивними службами з одного боку, та прикладними службами з іншої. Тобто ORB забезпечує базову інфраструктуру, Object Services - фундаментальні об’єктивні інтерфейси, а задачею Common Facilities – є підтримка інтерфейсів сервісів високого рівня, які можуть включати спеціалізацію Object Services. Тобто операції, представлені Common Facilities, призначені, зокрема, для використання Object Services та прикладними об’єктами. Це реалізується шляхом наслідування стандартних Інтерфейсів. Спільні засоби поділяються на горизонтальні та вертикальні. До горизонтальних сервісів відносяться такі спільні сервіси, як, наприклад, управління інформацією, задачами, всією системою, тобто засоби, які не залежать від конкретних прикладних систем. До вертикальних сервісів відносяться сервіси, специфічні для якоїсь діяльності, наприклад, медицина, фінанси.

CORBA та WWW. Відповідь на поставлене раніше питання - як об’єднати ІС, засновані на технології WWW, з іншими (в тому числі розподілені) - полягає в наступному: слід пов’язати технологію розподілених об’єктів (тобто технологію CORBA) з технологією WWW. Метою такої

роботи є детальний розгляд зв'язки CORBA та WWW. Є два рішення цієї проблеми, перше - базується на застосуванні технології CGI, а друге - на застосуванні технології Java. Досліджено практичне застосування цих технологій, з використанням продуктів Orbix та OrbixWeb від IONA Technologies.

CORBA-CGI. В чистому вигляді, технологія CGI полягає в наступному: для формування HTML-сторінки WEB-сервер запускає деякий CGI-скрипт, який може реалізувати достатньо складну функціональну логіку і звертатися, наприклад, до бази даних. CGI - скрипт являє собою окремий виконуваний додаток і мова, на якій написаний скрипт, не відіграє великої ролі. Рішення CORBA- CGI, рис.2.3, базується на тому, що CGI-скрипт є одночасно однією з компонент розподіленої системи. Головна відмінність від технології CGI полягає в тому, що CGI-скрипт є не просто виконуваним модулем, а він є одночасно і CORBA - клієнтом. В певному сенсі, скрипт служить точкою входу і виходу в розподіленій системі, усередині якої можуть відбуватися різноманітні процеси.

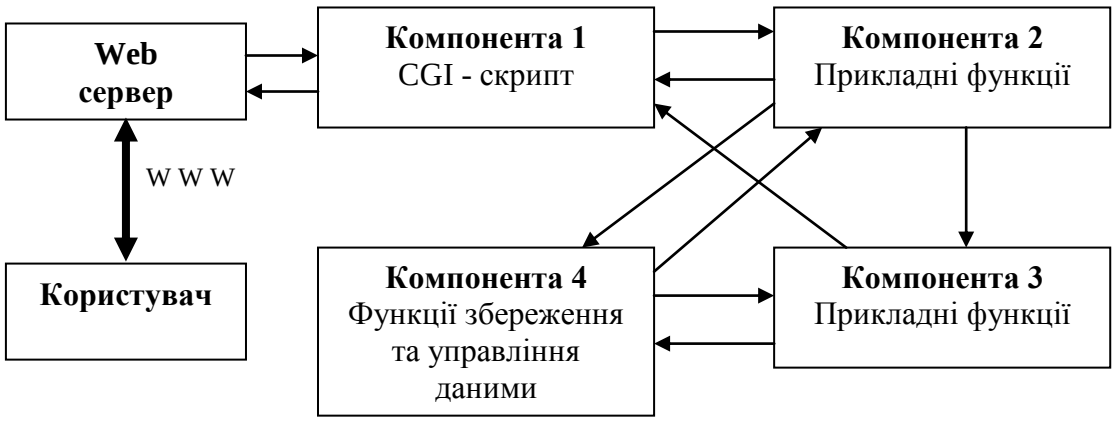


Рисунок 2.3 - CGI та CORBA

До переваг цієї технології можна віднести практично всі переваги використання CORBA. Крім того, користувач працює із звичними для нього HTML - сторінками, що особливо важливо при роботі з об'ємною текстовою інформацією. Стосовно недоліків цієї технології, практика свідчить, про

невелику ефективність системи. Кожного разу при перезавантаженні сторінки слід заново виконувати CGI-скрипт, заново встановлювати з'єднання з іншими компонентами системи. Це є неефективним і особливо виявляється, коли систему одночасно використовує кілька користувачів. З іншого боку, не знижуючи ефективність, неможливо своєчасно повідомляти користувача про зміни, що відбулися в інформації, яка ним використовується. Ці зміни будуть помітними лише за умови перезавантаження сторінки. Тобто користувач приймає участь в системі виключно в ролі клієнта. Окремим рішенням проблеми ефективності виконання CGI - запитів (особливо в багатокористувацькій системі), може бути розширення функціональності використовуваного WEB - сервера, шляхом додавання до нього відповідних функцій. Складності також виникають в разі створення, складних розгалужених інтерфейсів користувача. Справа в тому, що в системі, за умови виконання запитів окрім введеної у вікно браузера інформації, необхідна додаткова, схована від користувача інформація, яка характеризує поточний стан системи. Вся інтерфейсна частина системи прив'язана до вікна браузера, а та інформація, яка виводиться на екран створюється динамічно в залежності від параметрів запиту та внутрішнього стану інформаційної системи на певний момент часу. Внаслідок цього, якщо користувач виконує неконтрольовану навігацію вперед-назад за сторінками, що проглядаються, підвищується ризик десинхронізації інформації, яка проглядається користувачем і зберігається на сервері, що не є припустимим. Більше того, система повинна слідкувати і за тим, щоб вільні переходи від сторінки до сторінки мали б логічний сенс, що є важливо. Тим самим, у випадках складних користувацьких інтерфейсів необхідна наявність жорсткого контролю за поточним станом ІС.

Java-CORBA. Друге рішення проблеми зв'язування технологій CORBA та WWW - мова Java, рис. 2.4. Справа в тому, що OMG стандартизувала відображення з IDL в Java. Є програмні продукти, які

реалізують зв’язок CORBA та Java - наприклад, Java Virtual Machine (JVM), всередині якої і виконується завантажений аплет.

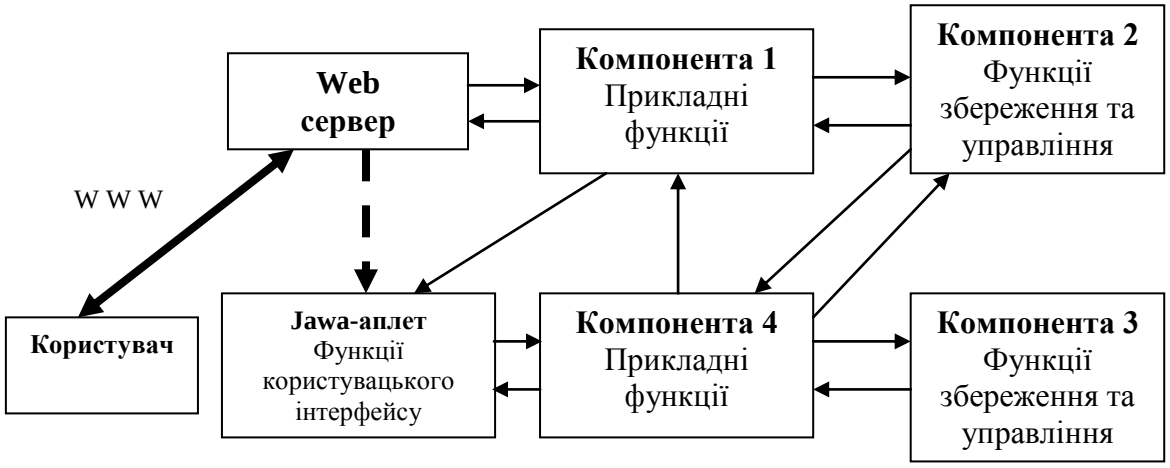


Рисунок 2.4 - Java та CORBA

Java - аплет, який є CORBA - клієнтом, встановлює всі необхідні з’єднання з іншими (серверними) додатками системи і саме через нього до користувача йде вся інформація. Аплет відіграє роль користувацького інтерфейсу для даної розподіленої системи. Кількість виконуваних аплетів нічим не обмежена - питання лише в достатніх обчислювальних ресурсах системи.

Переваги та недоліки технології Java- CORBA.

Переваги: Java - платформно-незалежна мова, всі аплети виконуватимуться однаково в будь-якій системі; існує можливість створення складних користувацьких інтерфейсів; аплети можуть виконувати роль клієнта та сервера; не виникає проблем за умови одночасної роботи декількох користувачів. Немає необхідності кожного разу заново завантажувати аплет, як це відбувається у випадку із CGI.

Недоліки: для ефективного виконання Java - додатків, система користувача повинна мати достатньо потужні обчислювальні ресурси, що особливо важливо для складних (насичених) користувацьких інтерфейсів; не всі браузери підтримують Java, або її останні версії.

Такі вбудовані в Java засоби як багатопотоковість, дають змогу легко реалізувати синхронну та асинхронну взаємодію аплетів з іншими додатками.

В технології Java - CORBA практично відсутні слабкі місця. Єдина проблема, яка може виникнути - необхідність наявності достатньої кількості обчислювальних ресурсів на стороні користувача, що є серйозним недоліком, який скоріше за все є недоліком мови Java.

Java - шлях до синхронізації інформації. В разі використання технології Java - CORBA, Java - аплети можуть відігравати роль клієнтів та серверів. Це дає змогу створювати "живі" сторінки - інформація на яких змінюється практично постійно. Наприклад, якщо аплет являє собою відображення стану певного пристрою, то за умови переходу пристроїв з одного стану до іншого, інформація в аплеті практично миттєво зміниться відповідним чином. Причому це може бути інформація будь-якого типу - графічна або текстова. Тоді виникає питання про те, яким способом відбувається обмін інформацією між аплетами та іншою частиною системи.

В CORBA існують два різних типи передачі повідомлень - механізм PUSH та механізм PULL .

Механізми PUSH та PULL. Механізм PULL, рис.2.5, являє собою наступне: коли клієнт готовий обробляти повідомлення, він опитує сервер на наявність у нього нових повідомлень. Якщо таких немає, то клієнт через певний проміжок часу повторює операцію. В залежності від контексту розв’язуваної задачі та пропускних здатностей мережних каналів, тип взаємодії клієнта та сервера може бути асинхронним та синхронним. Механізм PUSH, в певному розумінні, протилежний механізму PULL. В цьому випадку сервер повідомлень сам, у міру надходження нових повідомлень, буде інформувати про це клієнтів. Тобто клієнти самі є серверами, а сервер повідомлень лише викликає в них відповідні методи, надаючи їм повідомлення. Як і в моделі PULL, взаємодія клієнта та сервера повідомлень може бути асинхронною та синхронною.

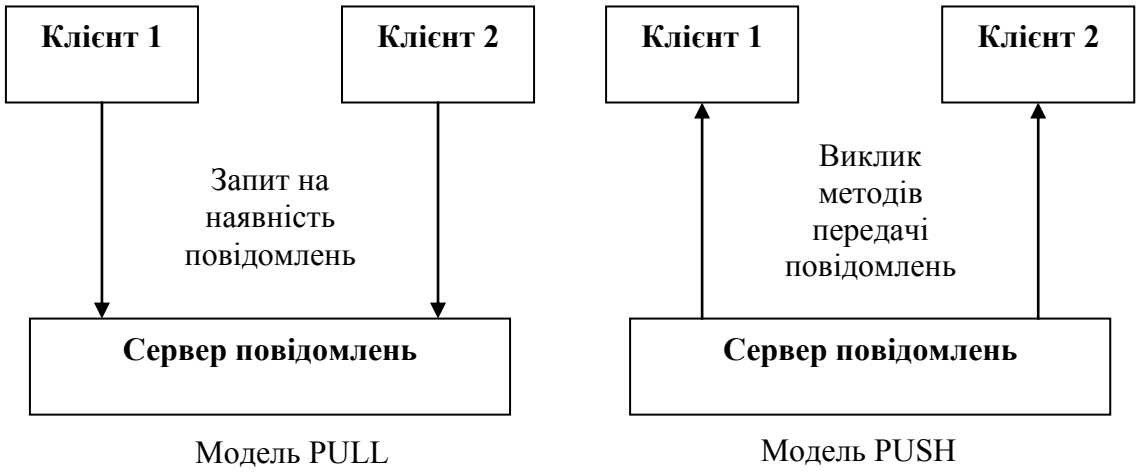


Рисунок 2.5 - Механізми PULL та PUSH

В разі використання механізму PULL, на обробку кожного запиту клієнта, сервер витрачає свої системні ресурси за наявності великої кількості клієнтів, які його регулярно опитують, що помітно знижує продуктивність. Багато залежить від того, як часто клієнти опитують сервер. За великої кількості клієнтів, які очікують, та нетривалого часу між двома запитами до сервера, продуктивність сервера знижується ще більше. У випадку, коли зв’язок між клієнтами та сервером незначний, то ефективність роботи механізму ще буде знижуватися у міру погіршення якості зв’язку. Тривалість виконання запитів буде збільшуватися, а канали зв’язку будуть зайнятими.

В моделі PUSH сервер повідомлень завантажений помітно менше. Сервер звільнений від необхідності регулярно реагувати на виклики клієнтів, що очікують. Тепер він взаємодіє з клієнтами-слухачами. “Виштовхування” повідомлень клієнту застосовується в тих випадках, коли повідомлення, які з’явилися на сервері повідомлень, повинні бути негайно оброблені клієнтом (клієнтами). За наявності неякісного зв’язку між вузлами, механізм PUSH набагато більше рентабельний, порівняно з PULL - він використовує канал лише один раз для кожного клієнта. За умови реальних розробок ІС, мають місце обидва представлених способи взаємодії компонентів. Розумна комбінація компонент ІС, які підтримують PUSH/ PULL моделі обміну

повідомленнями, дає змогу досягнути високого рівня гнучкості та продуктивності створюваної ІС.

Використання засобів Java. Використання PUSH - технології дає змогу практично миттєво і ефективно пересилати оновлену інформацію всім зацікавленим додаткам. За умови використання Java-CORBA реалізація PUSH - технології дуже проста. В зацікавленого у повідомленні клієнта (аплета) створюється спеціальний слухач, який запускається основним аплетом в окремому процесі з використанні механізму Treads (ниток) в Java. В разі запуску слухач, який реалізує специфічний для даного типу IDL - інтерфейс, інформує сервер повідомлень про свою зацікавленість в прийомі повідомлень даного типу, після чого починає очікувати. За умови отримання повідомлення клас-слухач може вже напряму взаємодіяти з клас-клієнтом.

Застосування технологій Java-CORBA та CORBA-CGI. Практика свідчить, що технологія Java-CORBA краще за все підходить для створення WWW CORBA-клієнтів, які: мають нестандартний, або не HTML - подібний користувацький інтерфейс; активно взаємодіють з іншими компонентами ІС протягом певного часу; повинні приймати участь в системі в ролі клієнта, та в ролі сервера. Технологію CORBA-CGI вигідно застосовувати у випадку, якщо: відбувається робота з великими обсягами текстової інформації і системні ресурси на стороні клієнта не мають достатньої потужності. Незважаючи на те, що переваги технології Java-CORBA над технологією CORBA-CGI значні, і область застосування ширша, обидві розглянуті технології добре підходять для об’єднання WWW - систем та клієнт/сервер - систем. Технологія CORBA-CGI розширює можливості CGI, а технологія Java-CORBA збільшує можливості всього WWW - до рівня розподілених об’єктних систем.

Тобто технології Java та CORBA добре доповнюють одна одну як потужний і універсальний засіб для захисту проблеми об’єднання систем, що базуються на технології WWW, з подібними та іншими, особливо розподіленими, ІС. Проте поява технології DCOM (Microsoft), суттєво

потіснила CORBA з ринку Windows - орієнтованих систем. Технологія RMI, навпаки, робить кроки назустріч CORBA. Починаючи з JDK 1.2, протокол RMI виконується поверх протоколу IIOP, що вигідно Java та CORBA розробникам. Масове використання технології Java-CORBA виведе Internet на новий рівень взаємодії. В такий спосіб відбудеться перехід від Web до нової, об’єктної мережі – Object Web.

3 ПАРАЛЕЛЬНІ ОБЧИСЛЮВАЛЬНІ СИСТЕМИ ТА ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ.

3.1 Побудова паралельних обчислювальних систем, аналіз і моделювання паралельних обчислень.

Шляхи досягнення паралелізму. Під паралельними обчисленнями розуміють процеси обробки даних, в яких комп'ютерна система одночасно здійснює декілька операцій. Досягнення паралелізму можливе за умови виконання наступних вимог до архітектурних принципів побудови обчислювального середовища:

- незалежність функціонування окремих пристроїв ЕОМ - ця вимога відноситься до всіх основних компонентів обчислювальної системи (пристрої введення-виведення, процесори, пристрої пам'яті);

- надлишковість елементів обчислювальної системи - організація надлишковості може здійснюватися у таких формах: використання спеціалізованих пристроїв (окремі процесори для специфічних задач, пристрої багаторівневої пам'яті - регістри, кеш); дублювання пристроїв ЕОМ шляхом використання декількох однотипних процесорів або кількох пристроїв оперативної пам'яті.

Окремою формою забезпечення паралелізму може бути конвеєрна реалізація пристроїв обробки інформації, коли виконання операцій в пристроях представлено як виконання послідовності складових операції команд. В результаті при обчисленнях в таких пристроях на різних стадіях обробки можуть знаходитися одночасно декілька різних елементів даних.

Розрізняють такі режими виконання незалежних частин програми:

- багатозадачний режим (режим розділення часу), коли для виконання декількох процесів використовується єдиний процесор. Такий режим є псевдопаралельним, коли активним, виконуваним, може бути один єдиний процес, а всі інші процеси знаходяться в стані очікування своєї черги;

застосування режиму розділення часу може підвищити ефективність організації обчислень. Наприклад, коли один з процесів не може виконуватися внаслідок очікування даних, що вводяться, а процесор може бути задіяний для виконання іншого процесу, готового до виконання. В цьому режимі з'являються багато ефектів паралельних обчислень (необхідність взаємного виключення та синхронізації процесів, та ін.), та, як результат, цей режим можна використати за умови початкової підготовки паралельних програм;

- паралельне виконання, коли в один і той же момент часу може виконуватися декілька команд обробки даних. Такий режим обчислень може бути забезпечений не тільки за наявності декількох процесорів, але й з використанням конвеєрних та векторних пристроїв обробки;

- розподілені обчислення; цей термін використовують як вказівник паралельної обробки даних, коли використовуються декілька пристроїв оброблення, достатньо віддалених один від одного, коли передача даних по мережі призводить до істотних часових затримок. Як результат, ефективне оброблення даних за умови такого способу організації обчислень можливе тільки для паралельних алгоритмів з низькою інтенсивністю потоків міжпроцесорної передачі даних. Ці умови характерні, наприклад, для обчислень в багатомашинних обчислювальних комплексах, утворених об'єднанням декількох окремих ЕОМ з використанням каналів зв'язку локальних або глобальних інформаційних мереж.

Далі розглядатимемо другий тип організації паралелізму, який реалізується у багатопроцесорних обчислювальних системах.

Приклади паралельних обчислювальних систем. Кількість паралельних обчислювальних систем величезна. Кожна така система унікальна, в кожній з них встановлюються різні апаратні складові: процесори (Intel, Power, AMD, HP, Alpha, Nec, Cray, та ін.), мережеві карти (Ethernet, Myrinet, Infiniband, SCI,...). Вони функціонують під управлінням різних операційних систем (версії Unix/Linux, версії Widjws,...) і використовують

різне прикладне програмне забезпечення. Здається, що практично неможливо знайти між ними щось спільне, але це не так. Далі будуть сформульовані варіанти класифікацій паралельних обчислювальних систем. Розглянемо декілька прикладів.

Суперкомп'ютери. Початком ери суперкомп'ютерів вважається 1976 рік з появою векторної системи Cray 1. Результати, показані цією системою на обмеженому в той час наборі додатків, були більш ніж вражаючими в порівнянні з іншими, тому система отримала назву "суперкомп'ютер" і протягом тривалого часу вона визначала розвиток всієї індустрії високопродуктивних обчислень. Проте в результаті сумісної еволюції архітектури та програмного забезпечення на ринку стали з'являтися системи з кардинально різними характеристиками, тому визначення "суперкомп'ютер" стало багатозначним і неодноразово переглядалося. Спроби дати означення терміну "суперкомп'ютер" базувалися не тільки на продуктивності. З альтернативних означень найбільш цікаві два: економічне та філософське. Перше означає, що суперкомп'ютер - це система, ціна якої вища за 1 - 2 млн. доларів США, друге - що суперкомп'ютер - це комп'ютер, потужність якого тільки на порядок менша необхідної для розв'язання сучасних задач.

Програма ASCI. Програма ASCI (http://www.llnl.gov/asci/) - Accelerated Strategic Computing Initiative, яка підтримується Міністерством енергетики США, в якості однієї з основних цілей - створення суперкомп'ютерів з продуктивністю в 100 TFlops (Терафлопс, означає 1 трильйон операцій за секунду). Перша система серії ASCI - ASCI Red, створена в 1996 р. компанією Intel, стала першим в світі комп'ютером з продуктивністю в 1 TFlops, яка надалі була доведена до 3 TFlops. Трьома роками пізніше з'явилися ASCI Blue Pacific від IBM та ASCI Blue Mountain від SGI, які стали першими на той час суперкомп'ютерами з швидкодією 3 TFlops. В червні 2000 року була введена в дію система ASCI White (http://www.llnl.gov/asci/platforms/white/) з піковою продуктивністю вище за

12 TFlops, реальна продуктивність на тесті LINPACK скала 4938 TFlops, пізніше цей показник був доведений до 7304 TFlops. Апаратно ASCI White є системою IBM RS/6000 SP з 512 симетричними мультипроцесорами (SMP) вузлами. Кожний вузол має 16 процесорів, система в цілому - 8192 процесорів. Оперативна пам'ять системи складає 4 ТВ, ємність дискового простору складає 180 ТВ. Всі вузли системи є симетричними мультипроцесорами IBM RS/6000 POWER 3 з 64 - розрядною архітектурою. Кожний вузол автономний, має власну пам'ять, операційну систему, локальний диск та 16 процесорів. Процесори POWER 3 є суперскалярними 64 - розрядними числами конвеєрної організації з двома пристроями обробки команд з плаваючою комою та трьома пристроями з обробки цілочислових команд. Вони здатні виконувати до 8 команд за тактовий цикл та до 4 операцій з плаваючою комою за такт. Тактова частота кожного такого процесора 375 MHz.

Програмне забезпечення ASCI White підтримує змішану модель програмування - передача повідомлень між вузлами та багатопотоковість всередині SMP - вузла. Операційна система є версією Unix - IBM AIX. AIX підтримує як 32 -, так і 64 - розрядні системи RS/6000. Підтримка паралельного коду на ASCI White включає паралельні бібліотеки, налагоджування, зокрема TotalView), профілювання, утиліти IBM та сервісні програми з аналізування ефективності виконання. Підтримуються бібліотеки MPI, OpenMP, потоки POSIX та транслятор директив IBM. Є паралельне налагодження IBM.

Система BlueGene - один з найпотужніших суперкомп'ютерів у світі створений фірмою IBM. Система має назву "BlueGene/L DD2 beta-System", вона є першою чергою повної обчислювальної машини. Згідно з прогнозами, на момент введення в дію в роботу її пікова продуктивність досягне 360 TFlops. Основні напрямки застосування є гідродинаміка, квантова хімія, моделювання клімату, та ін. Поточний варіант системи має такі характеристики: 32 стійки по 1024 двох ядерних 32-бітних процесори

PowerPC 440 з тактовою частотою 0.7 GHz; пікова продуктивність - порядку 180 TFlops; максимально показана продуктивність (на тесті LINPACK) - 135 TFlops.

МВС - 1000. Один з найпотужніших на теренах СНГ суперкомп'ютерів - багатопроцесорна обчислювальна система МВС-1000М встановлена у Міжвідомчому супекомп'ютерному центрі РАН, роботи з його створення проводилися з 2000 по 2001рік. Склад системи такий: 384 двох процесорних модулі на базі Alpha 21264 з тактовою частотою 667 MHz (кеш L2 - 4 Mb), зібрані у вигляді 6 базових блоків, по 64 модулі у кожному; керуючий сервер та файл-сервер NetApp F840; мережі Myrinet 2000 та Fast/Gigabit Ethernet; мережевий монітор; система безперебійного живлення. Кожний обчислювальний модуль має по 2 Gb оперативної пам'яті, HDD 20 GB, мережеві карти Myrinet (2000 Mbit). при обміні даними між модулями з використанням протоколів MP1 на мережі Myrinet пропускна здатність в МВС-1000М складає 110-150 Mb в секунду.

Кластер AC3 Velocity Cluster. Цей кластер встановлений в Корнельському університеті США і є результатом сумісної діяльності університету та консорціуму AC3 (Advanced Cluster Computing Consortium), утвореного компаніями Dell, Intel. Microsoft, Giganet та ще 15 виробників ПЗ з метою інтеграції різних технологій у створення кластерної архітектури для навчальних закладів та державних установ. Склад кластера: 64-ри чотирьох процесорних сервери Dell PowerEdge 6350 на базі Intel Pentium III Xeon MHz, 4 GB HDD, 100 MBit Ethernet card; і восьмипроцесорний сервер Dell PowerEdge 6350 на базі Intel Pentium III Xeon 550 MHz, 8 GB RAM, 36 GB HDD, 100 MBit Ethernet card.

Чотирьохпроцесорні сервери змонтовані по вісім штук на стійці і працюють під управлінням ОС Microsoft Windows NT 4.0 Server Enterprise Edition. Між серверами встановлено з'єднання на швидкості 100 Мбайт/с через Cluster Switch компанії Giganet.

Пікова продуктивність AC3 Velocity складає 122 GFlops з вартістю в 4-5 меншою, порівняно з суперкомп'ютерами з аналогічними показниками. На момент введення в дію, в 2000 році, кластер з показниками під час тестування на LINPACK в 47 GFlops займав 381 позицію в списку Top 500.

Кластер NCSA NT Supercluster. В 2000 році в Національному центрі суперкомп'ютерних технологій (NCSA - National Center for Supercomputing Applications) на основі робочих станцій Hewlett-Packard Kayak XU PC workstation (http://www.hp.com/desktops/kaeak/) був зібраний ще один кластер, для якого була вибрана операційна система (ОС) Microsoft Windows. Розробники назвали його NT Supercluster (http://archive.ncsa.uiuc. edu/SCD/Hardware/NTCluster/). На момент введення в дію кластер з показником на тесті LINPACK в 62 GFlops та піковою продуктивністю в 140 GFlops займав 207 стрічку списку Top 500. Кластер побудований з 38 двох процесорних серверів на базі Intel Pentium III Xeon 550 MHz, 1 Gb RAM, 7.5 Gb HDD, 100 MBit Ethernet card. Зв'язок між вузлами базується на мережі Myrinet (http://www.myri.com/murinet/index.html). Програмне безпечення кластера: операційна система - Microsoft Windows NT 4.0; компілятори - Fortran77, C/C++; рівень передачі повідомлень базується на HPVM (http://www-csag.ucsd.edu/projects/clusters.html).

Кластер Thunder. Чисельність систем, зібраних на основі процесорів корпорації Intel, які представлені в списку Top 500, складає 318. Самий потужний суперкомп'ютер, який являє собою кластер на основі Intel Itanium2, встановлений в Ліверморській національній лабораторії (США). Апаратна конфігурація кластера Thunder (http://www.llnl.gov/ linux/thunder/); 1024 сервери, по 4 процесори Intel Itanium 1.4GHz в кожному; 8 Gb оперативної пам'яті на вузол; загальна місткість дискової системи 150 Tb. Програмне забезпечення: операційна система CHOS 2.0; середовище паралельного програмування MPICH2; налагоджування паралельних програм TotalView; Intel та GNU Fortran, C/C++ компілятори. Кластер Thunder займає 5 позицію списку Top 500 (на момент встановлення - влітку 2004 року він

займав 2 стрічку) з піковою продуктивністю 22938 GFlops і максимально показаною на тесті LINPACK 19940 GFlops.

3.2. Класифікація обчислювальних систем.

Найпоширенішим способом класифікації ЕОМ є систематика Флінна (Flynn), в якій під час аналізу архітектури обчислювальних систем приділяється увага способам взаємодії послідовностей (потоків) виконуваних команд та оброблюваних даних В рамках цього підходу розглядаються такі основні типи систем:

- *SISD* **(**Single Instruction, Single Data) - системи, в яких існує одиночний потік команд та одиночний потік даних. До такого типу можна віднести звичайні послідовності ЕОМ;

- *SIMD* (Single Instruction, Multiple Data) - системи з одиночним потоком команд та множинним потоком даних. Подібний клас складають багатопроцесорні обчислювальні системи, в яких в кожний момент часу може виконуватися одна і та ж команда для обробки декількох інформаційних елементів; таку архітектуру мають зокрема багатопроцесорні системи з єдиним пристроєм управління. Цей підхід використовувався в попередні роки (системи ILLIAC IV або CM-1 компанії Thinking Machines), останнім часом його застосування обмежено, в основному, створенням спеціалізованих систем;

- *MISD* (Multiple Instruction, Single Data) - системи, в яких існує множинний потік команд та одиночний потік даних. Стосовно цього типу систем немає єдиної думки: ряд спеціалістів вважають, що прикладів конкретних ЕОМ, які відповідають цьому типу обчислювальних систем, не існує і введення такого класу застосовується для повноти класифікації; інші спеціалісти відносять до даного типу, наприклад, систолічні обчислювальні системи або системи з конвеєрною обробкою даних;

- *MIMD* (Multiple Instruction, Multiple Data) – системи з множинним потоком команд та множинним потоком даних. До цього класу відноситься більшість паралельних багатопроцесорних обчислювальних систем.

Систематика Флінна широко використовується при конкретизації типів комп'ютерних систем, але вона призводить до того, що практично всі типи паралельних систем, незважаючи на їх істотну різнорідність, виявляються віднесеними до однієї групи MIMD. В результаті цього вживаються заходи щодо деталізації систематики Флінна. Для класу MIMD запропонована загальновизнана структурна схема, в якій подальше розділення типів багатопроцесорних систем базується на використовуваних способах організації оперативної пам'яті в цих системах. Такий підхід дає змогу розрізнити два важливих типи багатопроцесорних систем - multiprocessors (мультипроцесори) та multicomputers (мультикомп'ютери).

Мультипроцесори. Для подальшої систематики мультипроцесорів враховується спосіб побудови спільної пам'яті. Перший можливий варіант - використання єдиної, централізованої, спільної пам'яті (shared memory). Такий підхід забезпечує однорідний доступ до пам'яті (uniform memory access або UMA) і є основою для побудови векторних паралельних процесорів (parallel vector processor або PVP) та симетричних мультипроцесорів (symmetric multiprocessor або SMP). Прикладами першої групи є суперкомп'ютер Cray T90, другої - IBM eServer, SunStarFire, HP Superdome, SGI Origin та ін. Однією з основних проблем, що виникають при організації паралельних обчислень з допомогою цих систем, є доступ з різних процесорів до спільних даних та забезпечення, у зв'язку з цим, однозначності (когерентності) вмісту різних кешів (cache coherence problem). Це відбувається тому, що за наявності спільних даних копії значень одних і тих же змінних можуть виявитися в кешах різних процесорів. Якщо за таких обставин, за наявності копій спільних даних, один з процесорів виконає зміну значення розділеної змінної, то значення копій в кешах інших процесорів виявляться такими, що не відповідають дійсності і їх використання призведе

до некоректності обчислень. Забезпечення однозначності кешів реалізується на апаратному рівні - для цього після зміни спільної змінної всі копії цієї змінної в кешах відмічаються як недійсні і подальший доступ до змінної потребує обов'язкового звертання до основної пам'яті. Необхідність забезпечення когерентності призводить до деякого зниження швидкості і ускладнює створення систем з достатньо великою кількістю процесорів.

Наявність спільних даних при паралельних обчисленнях приводить до необхідності синхронізації взаємодії одночасно виконуваних потоків команд. Так якщо зміна спільних даних потребує для свого виконання певної послідовності дій, то необхідно забезпечити взаємне виключення (mutual exclusion), щоб ці зміни в будь-який момент часу міг виконувати тільки один командний потік. Задачі взаємного виключення та синхронізації відносяться до числа класичних проблем, і їх розгляд при розробці паралельних програм є одним з основних питань паралельного програмування. Спільний доступ до даних можна забезпечити також при фізичному розподілі пам'яті (тривалість доступу вже не буде однаковою для всіх елементів пам'яті). Такий підхід іменується неоднорідним доступом до пам'яті (non-uniform memory access або NUMA). Серед систем з таким типом пам'яті виділяють:

а) системи, в яких для надання даних використовується тільки локальна кеш-пам'ять наявних процесорів (cache-only memory architecture або COMA); прикладами є KSR-1 та DDM;

б) системи, в яких забезпечується когерентність локальних кешів різних процесорів (cache-coherent NUMA або CC-NUMA); серед таких систем: SGI Origin 2000, Sun HPC 10000, IBM/Sequent NUMA-Q2000;

в) системи, в яких забезпечується спільний доступ до локальної пам'яті різних процесорів без підтримки на апаратному рівні когерентності кешу (non-cache coherent NUMA або NCC-NUMA); наприклад, система Cray T3E.

Використання розподіленої спільної пам'яті (distributed shared memory чи DSM) спрощує проблеми створення мультипроцесорів (відомі приклади

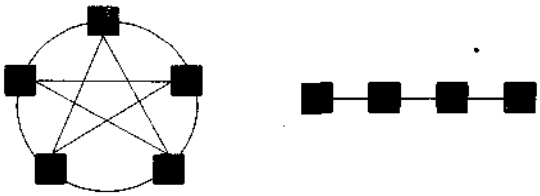
систем з кількома тисячами процесорів), проте проблеми ефективного використання розподіленої пам’яті (час доступу до локальної та віддаленої пам’яті може різнитися на декілька порядків) приводить до істотного підвищення складності паралельного програмування.

*Мультикомп'ютери***.** Це багатопроцесорні системи з розподіленою пам’яттю, які вже не забезпечують спільного доступу до всієї наявної в системі пам’яті (no-remote memory access або NORMA). Незважаючи на всю схожість подібної архітектури з системами з розподіленою спільною пам’яттю, мультикомп'ютери мають принципову відмінність: кожен процесор системи може використовувати тільки свою локальну пам’ять, в той час як для доступу до даних, розташованих на інших процесорах, слід явним чином виконати операції передачі повідомлень (message passing operations). Такий підхід застосовується за умови побудови двух важливих типів багатопроцесорних обчислювальних систем - масивно-паралельних систем (massively parallel processor) та кластерів (clusters). Серед представників першого типу систем - IBM RS/6000 SP2, Intel PARAGON, ASCI Red, трансп’ютерні системи Parsytec та ін.; прикладами кластерів є системи AC3 Velocity та NCSA NT Supercluster. Характеристикою обчислювальних систем кластерного типу є надзвичайно швидкий розвиток мікропроцесорних обчислювальних систем. Під кластером розуміють множину окремих комп’ютерів, об’єднаних в мережу, для яких за допомогою апаратно-програмних засобів забезпечується можливість уніфікованого управління (single system image), надійного функціонування (availability) та ефективного використання (performance). Кластери можуть бути утворені на основі вже наявних у споживачів окремих комп’ютерів або ж сконструйовані з типових комп’ютерних елементів, що не потребує істотних фінансових витрат. Застосування кластерів може певною мірою усунути проблеми, пов’язані з розробкою паралельних алгоритмів та програм, оскільки підвищення обчислювальної потужності окремих процесорів дає змогу будувати кластери з порівняно невеликої кількості (декілька десятків)

окремих комп’ютерів (lowly parallel processing). Тобто для паралельного виконання в алгоритмах розв’язку обчислювальних задач достатньо виділити тільки крупні незалежні частини розрахунків (coarse granularity), що в свою чергу, знижує складність побудови паралельних методів обчислень і зменшує потоки даних, що передаються, між комп’ютерами кластера. Разом з тим організація взаємодії обчислювальних вузлів кластера за допомогою передачі повідомлень приводить до значних часових затримок, що накладає додаткові обмеження на тип розроблюваних паралельних алгоритмів та програм. Звертається увага на відмінність поняття кластера від мережі комп’ютерів (network of workstations або NOW). Для побудови локальної комп’ютерної мережі використовують більш прості мережі передачі даних (порядку 100 Мбіт/сек). Комп’ютери мережі більше розосереджені, тому користувачі можуть застосувати їх для виконання якихось додаткових робіт. Слід зауважити, що існують також інші способи класифікації обчислювальних систем.

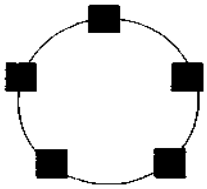
Приклади *топології мережі передачі даних.* Структура ліній комутації між процесорами обчислювальної системи (топологія мережі передачі даних) визначається, як правило, з врахуванням можливостей ефективної технічної реалізації. Важливу роль при виборі структури мережі відіграє також аналіз інтенсивності інформаційних потоків при паралельному вирішенні найпоширеніших обчислювальних задач. До числа типових топологій зазвичай відносять такі схеми комунікації процесорів (рис. 3.1):

- повний граф (completely-connected graph або clique) - система, в якій між будь-якою парою процесорів існує пряма лінія зв’язку. Така топологія забезпечує мінімальні затрати при передачі даних, проте вона має складну реалізацію за умови великої кількості процесорів;

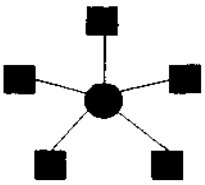


1 Повний граф

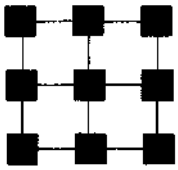
2 Лінійка



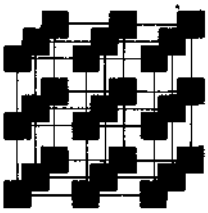
3 Кільце



4 Зірка



5 Двовимірна решітка



6 Тривимірна решітка

Рисунок 3.1 - Приклади топології багатопроцесорних обчислювальних систем

- лінійка (linear array або farm) - система, в якій всі процесори пронумеровані по порядку і кожний процесор, окрім першого і останнього, має лінії зв’язку тільки з двома сусідніми (з попереднім та наступним) процесорами. Така схема реалізується просто, а з іншого боку, відповідає структурі передачі даних при розв’язуванні багатьох обчислювальних задач, наприклад, при організації конвеєрних обчислень;

- кільце (ring) - ця топологія формується з лінійки процесорів шляхом з’єднанням першого і останнього процесора лінійки;

- зірка (star) - система, в якій всі процесори мають лінії зв’язку з певним керуючим процесором. Ця топологія ефективна, наприклад, при організації централізованих схем паралельних обчислень;

- решітка (mesh) - система, в якій граф ліній зв’язку утворює прямокутну сітку (двовимірну або тривимірну). Така технологія реалізується достатньо просто і може бути ефективно використана при паралельному виконанні багатьох чисельних алгоритмів (наприклад, при реалізації методів

аналізу математичних моделей, які описуються диференціальними рівняннями в часткових похідних);

- гіперкуб (hypercube) - ця топологія є окремим випадком структури решітки, коли за кожною розмірністю сітки є тільки два процесори (тобто гіперкуб містить *2N* процесорів при розмірності *N*). Такий варіант організації мережі передачі даних поширений на практиці і характеризується таким рядом розпізнавальних ознак: два процесори мають з'єднання, якщо двійкові зображення їх номерів мають тільки одну відмінну позицію; в *N*- вимірному гіперкубі кожний процесор зв'язаний рівно з *N* сусідами; *N* - вимірний гіперкуб можна розділити на два (*N-1*) - вимірних гіперкуби (всього можливі *N* таких варіантів розбиття); найкоротший шлях між двома будь-якими процесорами має довжини, яка співпадає з кількістю відмінних бітових значень в номерах процесорів (ця величина називається відстанню Хеммінга).

Топологія мережі обчислювальних кластерів. Для побудови кластерної системи в багатьох випадках використовують комутатор (switch), через який процесори кластера є повними графами, рис. 3.1, і у відповідності з яким передача даних може бути організована між будь-якими двома процесорами мережі. Одночасність виконання декількох комунікаційних операцій обмежена - в будь-який момент часу кожний процесор може приймати участь лише в одній операції прийому-передачі даних. В результаті паралельно можуть виконуватися тільки ті комунікаційні операції, в яких взаємодіючі пари процесорів не перетинаються між собою.

Характеристики топології мережі. Як основні характеристики топології мережі передачі даних використовуються такі показники:

- діаметр - показник, який визначається як максимальна відстань між двома процесорами мережі (такою відстанню є величина найкоротшого шляху між процесорами). Ця величина може характеризувати максимально необхідний час для передачі даних між процесорами, оскільки час передачі пропорційний довжині шляху;

- зв’язність (connectivity) - показник, який характеризує наявність різних маршрутів передачі даних між процесорами мережі. Конкретний вигляд цього показника можна визначити як мінімальну кількість дуг, які слід видалити для розділення мережі передачі даних на дві незв’язні області;

- ширина бінарного поділу (bisection width) - показник, який визначається як мінімальна кількість дуг, які слід видалити для розділу мережі передачі даних на дві незв’язні області однакового розміру;

- вартість - показник, який можна визначити, наприклад, як загальну кількість ліній передачі в багатопроцесорній обчислювальній системі.

Для порівняння в таблиці 3.1 наводяться значення перерахованих показників для різних топологій мережі передачі даних.

Таблиця 3.1. Характеристики топологій мережі передачі даних (p - кількість процесорів)

Топологія	**Діаметр**	**Ширина бісекції**	**Зв’язність**	**Вартість**
Повний граф	1	$p^2/4$	p-1	P(p-1)/2
Зірка	2	1	1	p-1
Повне двійкове дерево	2log((p+1)/2)	1	1	p-1
Лінійка	p-1	1	1	p-1
Кільце	[p/2]	2	2	P
Решітка N=2	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
Решітка-тор N=2	$2[\sqrt{p}/2]$	$2\sqrt{p}$	4	2p
Гіперкуб	log p	p/2	log p	(p log p)/2

4 МОДЕЛЮВАННЯ І АНАЛІЗ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

При розробці паралельних алгоритмів розв’язку складних задач важливим є аналіз ефективності використання паралелізму, тобто оцінка отримуваного прискорення процесу обчислень і скорочення тривалості розв’язування задачі.

Модель обчислень у вигляді графа "операції - операнди". Для опису існуючих інформаційних залежностей в алгоритмах рішення вибраних задач, можна використати модель у вигляді графа "операції-операнди". Для простоти вважатимемо, що тривалість виконання будь-яких обчислювальних операцій однакова і дорівнює 1 (одиниця виміру). Приймемо, що передача даних між обчислювальними пристроями виконується миттєво без будь-яких затрат часу (наприклад, за наявності спільної розділеної пам’яті в паралельній обчислювальній системі). Аналіз комунікаційної трудомісткості паралельних алгоритмів приводиться далі.

Зобразимо множину операцій, виконуваних в досліджуваному алгоритмі розв’язку обчислювальної задачі, та існуючі між операціями інформаційні залежності у вигляді ациклічного орієнтованого графа

$$G = (V, R), \qquad (4.1.1)$$

де $V = \{1,\ldots,|V|\}$ є множиною вершин графа, що є виконуваними операціями алгоритму, а R є множиною дуг графа (дуга $r = (i, j)$ належить графу тільки в тому випадку, якщо операція j не показує результату виконання операції i). Далі буде показано, що різні схеми обчислень мають різні можливості для розпаралелювання і, тим самим, при побудові моделі обчислень можна поставити задачу вибору обчислювальної схеми алгоритму, що найбільш підходить для паралельного обчислення.

Опис схеми паралельного виконання алгоритму. Операції, між якими немає шляху в рамках вибраної схеми обчислень, можна виконати

паралельно. Можливий спосіб опису паралельного виконання алгоритму полягає в наступному.

Нехай p є кількістю процесорів, використовуваних для виконання алгоритму. Тоді для паралельного виконання обчислень слід задати множину (розклад):

$$H_p = \{(i, P_i, t_i) : i \in V\}, \tag{4.1.2}$$

в якому для кожної операції $i \in V$ вказується номер використовуваного для виконання операції процесора P_i та тривалість виконання операції t_i. Для того щоб розклад міг бути реалізований, необхідно виконання таких вимог при задаванні множини H_p :

1) $\forall i, j \in V : t_i = t_j \Rightarrow P_i = P_j$, тобто один і той же процесор не повинен призначатися різним операціям в один і той же момент часу;

2) $\forall (i, j) \in R \Rightarrow t_j \geq t_i + 1$, тобто до призначеного моменту виконання операції всі необхідні дані вже повинні бути обчислені.

Визначення часу виконання паралельного алгоритму. Обчислювальна схема алгоритму G разом з розкладом H_p може розглядатися як модель паралельного алгоритму $A_p(G, H_p)$, виконуваного з використанням p процесорів. Тривалість виконання паралельного алгоритму визначається максимальним значенням часу, що використовується в розкладі:

$$T_p(G, H_p) = \max_{i \in V} (t_i + 1). \tag{4.1.3}$$

Для вибраної схеми обчислень бажано використання розкладу, який забезпечує мінімальну тривалість виконання алгоритму:

$$T_p(G) = \min_{H_p} T_p(G, H_p). \tag{4.1.4}$$

Зменшення тривалості виконання можна забезпечити шляхом підбору найкращої обчислювальної схеми:

$$T_p = \min_{G} T_p(G). \tag{4.1.5}$$

Оцінки $T_p(G,H_p)$, $T_p(G)$ та T_p можна застосувати як показники тривалості виконання паралельного алгоритму. Крім того, для аналізу максимально можливого паралелізму можна визначити оцінку найшвидшого виконання алгоритму:

$$T_\infty = \min_{p \geq 1} T_p. \tag{4.1.6}$$

Оцінку T_∞ можна розглядати як мінімально можливу тривалість виконання паралельного алгоритму при використанні необмеженої кількості процесорів (концепція обчислювальної системи з нескінченною кількістю процесорів, яка називається паракомп'ютером, широко застосовується при теоретичному аналізуванні паралельних обчислень). Оцінка T_1 визначає тривалість виконання алгоритму при використанні одного процесора і представляє, тим самим, тривалість виконання послідовного варіанту алгоритму рішення задачі. Побудова такої оцінки є важливою задачею під час аналізу паралельних алгоритмів, оскільки вона необхідна для визначення ефективності використання паралелізму (прискорення тривалості розв'язку задачі). Очевидно, що:

$$T_1(G) = \left|\overline{V}\right|, \tag{4.1.7}$$

де $\left|\overline{V}\right|$ - кількість вершин обчислювальної схеми без вершин введення. Важливо відмітити, що коли при визначення оцінки обмежитися розглядом тільки одного вибраного алгоритму розв'язку задачі і використати величину:

$$T_1 = \min_G T_1(G), \tag{4.1.8}$$

то отримувані при такій оцінці показники прискорення характеризуватимуть ефективність розпаралелювання вибраного алгоритму. Для оцінки ефективності паралельного рішення досліджуваної обчислювальної задачі тривалість послідовного рішення слід визначати з врахуванням різних послідовних алгоритмів, тобто використовувати величину:

$$T_1^* = \min T_1, \tag{4.1.9}$$

де операція мінімуму береться по множині всіх можливих послідовних алгоритмів розв'язку даної задачі.

Розглянемо (без доведення) теоретичні положення, які характеризують властивості оцінок тривалості виконання паралельного алгоритму.

Теорема 1. Мінімально можлива тривалість виконання паралельного алгоритму визначається довжиною максимального шляху обчислювальної схеми алгоритму, тобто:

$$T_\infty(G) = d(G). \qquad (4.1.10)$$

Теорема 2. Нехай для деякої вершини виведення в обчислювальній схемі алгоритму існує шлях з кожної вершини введення. Крім того, нехай вхідна ступінь вершин схеми (кількість вхідних дуг) не перевищує 2. Тоді мінімально можлива тривалість виконання паралельного алгоритму обмежена знизу значенням:

$$T_\infty(G) = \log_2 n, \qquad (4.1.11)$$

де n - кількість вершин введення в схемі алгоритму.

Теорема 3. При зменшенні кількості використовуваних процесорів тривалість виконання алгоритму збільшується пропорційно величині зменшення кількості процесорів, тобто:

$$\forall q = cp, 0 < c < 1 \Rightarrow T_p \le T_q. \qquad 4.1.12)$$

Теорема 4. Для будь-якої кількості використовуваних процесорів справедлива така верхня оцінка для тривалості виконання паралельного алгоритму:

$$\forall p \Rightarrow T_p < T_\infty + \frac{T_1}{p}. \qquad (4.1.13)$$

Теорема 5. Тривалості виконання алгоритму, яка порівнюється з мінімально можливою тривалістю T_∞, можна досягти при кількості процесорів порядку $p \sim \frac{T_1}{T_\infty}$, а саме:

$$p \geq \frac{T_1}{T_\infty} \Rightarrow T_p \leq 2T_\infty . \qquad (4.1.14)$$

При меншій кількості процесорів тривалість виконання алгоритму може перевищувати більш ніж в два рази найкращу тривалість обчислень при наявній чисельності процесорів, тобто:

$$p < \frac{T_1}{T_\infty} \Rightarrow \frac{T_1}{p} \leq T_p \leq 2\frac{T_1}{p} . \qquad (4.1.15)$$

Наведені вище твердження дають змогу дати наступні рекомендації з правил формування паралельних алгоритмів: при виборі обчислювальної схеми алгоритму повинен використовуватися граф з мінімальноможливим діаметром; для паралельного виконання доцільна кількість процесорів визначається величиною $p \sim \frac{T_1}{T_\infty}$; тривалість виконання паралельного алгоритму обмежується зверху величинами, наведеними в теоремах 4 та 5.

Показники ефективності паралельного алгоритму. Прискорення (speedup) отримуване при використанні паралельного алгоритму для p процесора, порівняно з послідовним варіантом виконання обчислень визначається величиною:

$$S_p(n) = T_1(n) / T_p(n), \qquad (4.1.16)$$

тобто як відношення тривалості рішення задач на скалярній ЕОМ до тривалості виконання паралельного алгоритму (величина n застосовується для параметризації обчислювальної складності розв'язуваної задачі і може розумітися, як кількість вхідних даних задачі).

Ефективність (efficiency) використання паралельним алгоритмом процесорів при розв'язку задачі визначається співвідношенням:

$$E_p(n) = T_1(n)/(pT_p(n)) = S_p(n)/p, \qquad (4.1.17)$$

(величина ефективності визначає середню частку часу виконання алгоритму, протягом якої процесори реально задіяні для розв'язку задачі). З цих співвідношень можна показати, що в найкращому випадку $S_n(p) = p$ і

$E_n(n)=1$. В разі практичного застосування цих показників для оцінки ефективності паралельних обчислень слід врахувати дві таких позиції:

1) За певних обставин прискорення може виявитися більша чисельність використовуваних процесорів $S_p(n)>p$ - в цьому випадку кажуть про існування понадлінійного (superliner) прискорення. На практиці понадлінійне прискорення можливе. Однією з причин цього може бути неоднорідність умов виконання послідовної та паралельної програм. Так за умови розв'язку задачі на одному процесорі може виявитись, що буде недостатньо оперативної пам'яті для зберігання всіх оброблюваних даних і тоді необхідно використати більш повільну зовнішню пам'яті (у випадку використання кількох процесорів оперативної пам'яті може виявитися достатньо за рахунок розділення даних між процесорами). Ще однією причиною понадлінійного прискорення може бути нелінійний характер залежності складності рішення задачі від об'єму оброблюваних даних. Так, наприклад, відомий алгоритм бульбашкового сортування характеризується квадратичною залежністю кількості необхідних операцій від числа впорядкованих даних. Як результат, при розподілі сортувального масиву між процесорами може бути отримане прискорення, що перевищує чисельність процесорів. Джерелом понадлінійного прискорення може бути також відмінність обчислювальних схем послідовного та паралельного методів.

2) Під час уважного аналізу можна звернути увагу, що спроби підвищення якості паралельних обчислень за одним із показників (прискорення або ефективності) можуть призвести до погіршення ситуації за другим показником, оскільки показники якості паралельних обчислень часто суперечливі. Так, наприклад, підвищення прискорення може бути забезпеченим за рахунок збільшення чисельності процесорів, що призводить, як правило, до падіння ефективності. І навпаки, підвищення ефективності досягається в багатьох випадках при зменшенні чисельності процесорів (в граничному випадку ідеальна ефективність $E_p(n)=1$ легко забезпечується в

разі використання одного процесора). Як результат, розробка методів паралельних обчислень часто передбачає вибір певного компромісного варіанту з врахуванням бажаних показників прискорення та ефективності.

В разі вибору належного паралельного способу розв'язку задачі може виявитись корисною оцінка вартості обчислень, яка визначається як добуток тривалості паралельного рішення задачі та чисельності використаних процесорів:

$$C_p = pT_p. \tag{4.1.18}$$

В зв'язку з цим можна означити поняття вартісно-оптимального (cost-optimal) паралельного алгоритму як методу, вартість якого пропорційна тривалості виконання найкращого паралельного алгоритму.

Розглянемо, приклад розв'язку задачі обчислення часткових сум для послідовності числових значень.

Приклад обчислення часткових сум послідовності числових значень

Розглянемо задачу знаходження частинних сум послідовності числових значень:

$$S_k = \sum_{i=1}^{k} x_i, 1 \le k \le n, \tag{4.1.19}$$

де n - кількість підсумованих значень (назва задачі - prefix sum problem) . Вивчення можливих паралельних методів розв'язку цієї задачі здійснимо на основі більш простого варіанту її постановки - з задачі обчислення загальної суми наявного набору значень (випадок загальної задачі редукції):

$$S = \sum_{i=1}^{n} x_i . \tag{4.1.20}$$

Послідовний алгоритм знаходження суми. Традиційний алгоритм розв'язку цієї задачі полягає в послідовності знаходження суми елементів числового набору:

$$S = 0, \tag{4.1.21}$$

$$S = S + x_1 +,... \tag{4.1.22}$$

Обчислювальну схему цього алгоритму можна подати наступним чином:

$$G_1 = (V_1, R_1), \tag{4.1.23}$$

де $V_1 = (v_{01},...,v_{0n},v_{11},...,v_{1n})$ є множиною операцій (вершини $v_{01},...,v_{0n}$ означають операції введення, кожна вершина $v_{1i}, 1 \le i \le n$, відповідає додаванню значення x_i до накопичуваної суми S), а:

$$R_1 = \{(v_{0i}, v_{1i}),(v_{1i}, v_{1i+1}), 1 \le i \le n-1\} \tag{4.1.24}$$

є множиною дуг, що визначають інформаційні залежності операцій. Можна помітити, що стандартний алгоритм знаходження суми допускає тільки певне послідовне виконання і не може бути розпаралелений.

Каскадна схема знаходження суми. Паралелізм алгоритму знаходження суми стає можливим тільки за іншого способу побудови процесу обчислень, який базується на використанні асоціативності операції додавання. Інший варіант знаходження суми (відомий, як каскадна схема) полягає в наступному:

- на першій ітерації каскадної схеми всі вихідні дані розбиваються на пари, і для кожної пари обчислюється сума їх значень;

- всі отримані суми також розбиваються на пари, і знову виконується знаходження суми значень пар, і т.д.

Таку обчислювальну схему можна визначити як граф (нехай $n = 2^k$):

$$G_2 = (V_2, R_2), \tag{4.1.25}$$

де $V_2 = \{(v_{i1},...,v_{li}), 0 \le i \le k, 1 \le l_i \le 2^{-1}n\}$ є вершини графа ($(v_{01},...,v_{0n})$ - операції введення, $(v_{1l},...,v_{1n/2})$ - операції знаходження суми першої ітерації, і т.д.) а множина дуг графа визначається співвідношеннями:

$$R_2 - \{(v_{i-1,2j-1}vij),(v_{i-1,2j}v_{ij}), 1 \le i \le k, 1 \le j \le 2^{-i}n\}. \tag{4.1.26}$$

Нескладно оцінити, кількість ітерацій каскадної схеми, яка виявляється рівною величині:

$$k = \log_2 n, \tag{4.1.27}$$

а загальна кількість операцій знаходження суми:

$$K_{посл} = n/2 + n/4 + ... + 1 = n - 1 \tag{4.1.28}$$

співпадає з кількістю операцій послідовного варіанту алгоритму знаходження суми. За умови паралельного виконання окремих ітерацій каскадної схеми загальна кількість паралельних операцій знаходження суми становить:

$$K_{пар} = \log_2 n. \tag{4.1.29}$$

Оскільки вважається, що тривалість виконання будь-яких операцій однакова і одинична, то $T_1 = K_{посл}, T_p = K_{пар}$, то показники прискорення та ефективності каскадної схеми алгоритму знаходження суми можна оцінити, як:

$$S_p = T_1 / T_p = (n-1)/\log_2 n, \tag{4.1.30}$$

$$E_p = T_1 / pT_p = (n-1)/(p\log_2 n) = (n-1)/((n/2)\log_2 n) \tag{4.1.31}$$

де $p = n/2$ - необхідна чисельність процесорів для виконання каскадної схеми.

Аналізуючи отримані характеристики, можна помітити, що тривалість паралельного виконання каскадної схеми співпадає з оцінкою для паракомп'ютера в теоремі 2. Проте ефективність використання процесорів зменшується із збільшенням чисельності значень суми:

$\lim E_p \to 0$ за умови $n \to \infty$.

Модифікована каскадна схема. Отримання асимптотичної ненульової ефективності можна забезпечити в разі використання модифікованої каскадної схеми. Для спрощення побудови оцінок можна припустити

$n = 2^k, k = 2^s$. Тоді в новому варіанті каскадної схеми всі обчислення виконуються в два послідовно виконуваних етапи знаходження суми:

- на першому етапі обчислень всі значення сум поділяються на $(n/\log_2 n)$ груп, в кожній з яких міститься $\log_2 n$ елементів; далі для кожної групи обчислюється сума значень з використанням послідовного алгоритму знаходження суми; обчислення в кожній групі можуть виконуватися незалежно один від одного, тобто для цього необхідна наявність не менше $(n/\log_2 n)$ процесорів);

- на другому етапі для отриманих $(n/\log_2 n)$ сум окремих груп застосовується звичайна каскадна схема.

Тоді для виконання першого етапу потрібно $\log_2 n$ паралельних операцій з використанням $p_1 = (n/\log_2 n)$ процесорів. Для виконання другого етапу необхідно:

$$\log_2 (n/\log_2 n) \le \log_2 n \qquad (4.1.32)$$

паралельних операцій для $p_2 = (n/\log_2 n)/2$ процесорів. Як результат, даний спосіб знаходження суми характеризується такими показниками:

$$T_p = 2\log_2 n, p = (n/\log_2 n). \qquad (4.1.33)$$

З урахуванням отриманих оцінок показники прискорення та ефективності модифікованої каскадної схеми визначаються співвідношеннями:

$$S_p = T_1 / T_{pn} = (n-1)/2\log_2 n, \qquad (4.1.34)$$

$$E_p = T_1 / pT_p = (n-1)/(2n/\log_2 n)\log_2 n) = (n-1)/2 \qquad (4.1.35)$$

Порівнюючи ці оцінки з показниками загальної каскадної схеми, можна помітити, що прискорення для запропонованого паралельного алгоритму зменшилось в 2 рази, проте для ефективності нового методу знаходження суми можна отримати асимптотичну ненульову оцінку знизу:

$E_p = (n-1)/2n \ge 0/25, \lim E_p \to 0.5$ за умови $n \to \infty$.

Можна помітити, що дані значення показників досягаються тоді, коли чисельність процесорів становить величину, визначену в теоремі 5. Слід підкреслити, що, на відміну від звичайної каскадної схеми, модифікований каскадний алгоритм є вартісним - оптимістичним, оскільки вартість обчислень в цьому випадку:

$$C_p = pT_p = (n/\log_2 n)(2\log_2 n) \qquad (4.1.36)$$

пропорційна тривалості виконання послідовного алгоритму.

Обчислення всіх часткових сум. Розглянемо задачу обчислення всіх часткових сум послідовності значень і проаналізуємо можливі способи послідовної та паралельної організації обчислень. Обчислення всіх часткових сум на скалярному комп'ютері може бути отримано з використанням звичайного послідовного алгоритму знаходження суми за умови тієї ж кількості операцій:

$$T_1 = n. \qquad (4.1.37)$$

За умови паралельного виконання застосування каскадної схеми в явному вигляді не призводить до бажаних результатів; досягнення ефективного розпаралелювання потребує залучення нових підходів для розробки нових паралельно - орієнтованих алгоритмів рішення задач. Так для розглянутої задачі знаходження всіх часткових сум алгоритм, що забезпечує отримання результату за $\log_2 n$ паралельних операцій (як і в випадку обчислень загальної суми), може полягати в наступному:

- перед початком обчислень створюється копія S вектора значень суми ($S = x$);

- далі на кожній ітерації сумування $i, 1 \le i \le \log_2 n$, формується допоміжний вектор Q шляхом зсуву праворуч вектор S на 2^{i-1} позицій (які вивільняються при зсуві позиції ліворуч встановлюються в нульове значення); ітерація алгоритму завершується паралельною операцією знаходження суми векторів S та Q.

Всього паралельний алгоритм виконується за $\log_2 n$ паралельних операцій додавання. На кожній ітерації алгоритму паралельно виконуються n скалярних операцій додавання, таким чином, загальна кількість скалярних операцій визначається величиною:

$$K_{пар} = n \log_2 n \qquad (4.1.38)$$

(паралельний алгоритм містить велику кількість операцій порівняно з послідовним способом знаходження суми). Необхідна кількість процесорів визначається кількістю значень, щодо яких визначається сума ($p = n$).

З урахуванням отриманих співвідношень показники прискорення та ефективності паралельного алгоритму визначення всіх часткових сум оцінюється наступним чином:

$$S_p = T_1 / T_p = n / \log_2 n . \qquad (4.1.39)$$

$$E_p = T_1 / pT_p = n/(\log_2 n) = n/(n \log_2 n) = 1/\log \qquad (4.1.40)$$

З побудованих оцінок випливає, що ефективність алгоритму також зменшується із збільшенням чисельності значень, щодо яких визначається сума, і за необхідності підвищення величини цього показника ефективнішою може виявитися модифікація алгоритму, як і у випадку звичайної каскадної схеми.

5 СПОСОБИ ОЦІНЮВАННЯ МАКСИМАЛЬНО ДОСЯЖНОГО ПАРАЛЕЛІЗМУ

Оцінка якості паралельних обчислень передбачає наявність найкращих (максимально досяжних) значень показників прискорення та ефективності. Проте отримання ідеальних величин $S_p = p$ для прискорення та $E_p = 1$ для ефективності може бути забезпечене не для всіх обчислювальних трудомістких задач. Розглянемо закономірності, корисні при побудові оцінок максимально досяжного паралелізму.

1. Закон Амдаля. Досягненню максимального прискорення може перешкоджати існування в виконуваних обчисленнях послідовних розрахунків, які можна розпаралелити. Нехай f - частка послідовних обчислень у вибраному алгоритмі обробки даних. Тоді у відповідності з законом Амдаля прискорення процесу обчислень з використанням p процесорів обмежується величиною:

$$S_p \le \frac{1}{f + (1-f)/p} \le S^* = \frac{1}{f}. \qquad (2.1.41)$$

Так за наявності 10% послідовних команд у виконуваних обчисленнях ефект використання паралелізму не може перевищувати 10-кратного прискорення обробки даних. В розглянутому вище прикладі обчислення суми значень для каскадної схеми частка послідовних розрахунків складає $f = \log_2 n / n$ і, як результат, величина можливого прискорення обмежена оцінкою $S^* = n / \log_2 n$.

Закон Амдаля характеризує одну з самих серйозних проблем в галузі паралельного програмування (алгоритмів без певної частки послідовних команд практично не існує). проте часто частка послідовних дій характеризує не можливість паралельного рішення задач, а послідовні властивості застосовуваних алгоритмів. Тому частка послідовних обчислень може бути

істотно знижена при виборі методів, що будуть більш придатними для розпаралелювання.

Розгляд закону Амдаля відбувається в припущенні, що частка послідовних розрахунків f є сталою величиною і не залежить від параметра n, який визначає обчислювальну складність розв'язуваної задачі. Проте для великого класу задач частка $f = f(n)$ є спадаючою функцією від n, і в цьому випадку прискорення для фіксованої кількості процесорів може бути збільшено за рахунок збільшення обчислювальної складності розв'язуваної задачі. Це зауваження можна сформулювати як твердження, що прискорення $S_p = S_p(n)$ є зростаючою функцією від параметра n (це твердження називається ефектом Амдаля). Так в прикладі, обчислення суми значень при використанні фіксованої чисельності процесорів p сумарний набір даних може бути розділений на блоки розміру n/p, для яких спочатку паралельно можна обчислити часткові суми, а далі ці суми можна скласти з використанням каскадної схеми. Тривалість послідовної частини виконуваних операцій (мінімально можлива тривалість паралельного виконання) в цьому випадку складає:

$$T_p = (n/p) + \log_2 p, \tag{2.1.42}$$

що призводить до оцінки частки послідовних розрахунків як величини:

$$f = (1/p) + \log_2 p/n \tag{2.1.43}$$

Як випливає з отриманого виразу, частка послідовних розрахунків f спадає із зростанням n і в граничному випадку маємо оцінку максимально можливого прискорення $S^* = p$.

2. Закон Густавсона-Баріса. Оцінимо максимально досяжне прискорення виходячи з наявної частки послідовних розрахунків у виконуваних паралельних обчисленнях:

$$g = \frac{\tau(n)}{\tau(n) + \pi(n)/p}, \tag{2.1.44}$$

де $\tau(n)$ і $\pi(n)$ - тривалість послідовної та паралельної частин виконуваних обчислень, відповідно, тобто:

$$T_1 = \tau(n) + \pi(n),\ T_p = \tau(n) + \pi(n)/p. \quad (2.1.45)$$

З врахуванням введеної величини g можна отримати:

$$\tau(n) = g(n) + \pi(n)/p,\ \pi(n) = (1-g)p(\tau(n) + \pi(n)/p), \quad (2.1.46)$$

що дає змогу побудувати оцінку для прискорення:

$$S_p = \frac{T_1}{T_p} = \frac{\tau(n) + \pi(n)}{\tau(n) + \pi(p)/p} = \frac{(\tau(n) + \pi(n)/p)(g + (1-g)p)}{\tau(n) + \pi(n)/p} \quad (2.1.47)$$

,

яка після спрощення приводиться до типу закону Густавсона-Барсіса (Gustafson - Barsis's law):

$$S_p = g + (1-g)p = p + (1-p)g. \quad (2.1.48)$$

Стосовно прикладу наведеного вище підсумовування значень при використанні p процесорів тривалість паралельного використання складає:

$$T_p = (n/p) + \log_2 p, \quad (2.1.49)$$

що відповідає послідовній частці

$$g = \frac{\log_2 p}{(n/p) + \log_2 p}. \quad (2.1.50)$$

За рахунок збільшення чисельності підсумованих значень величина g може бути зневажливо малою, забезпечуючи отримання ідеально можливого прискорення $S_p = p$.

При розгляді закону Густавсона-Барсіса слід враховувати те, що із збільшенням чисельності використовуваних процесорів темп зменшення тривалості паралельного розв'язку задач може спадати (після перевищення певного порогу). Проте за рахунок зменшення тривалості обчислень складність розв'язуваних задач може збільшуватись. Оцінку отримуваного при цьому прискорення можна визначити з використанням сформульованих закономірностей. Така аналітична оцінка корисна, оскільки розв'язок складніших варіантів задач на одному процесорі може виявитися достатньо

трудомістким, навіть неможливим, наприклад, по причині недостанього об’єму оперативної пам’яті. Враховуючи ці обставини, оцінку прискорення, отримувану у відповідності з законом Густавсона-Барсіса , ще називають прискоренням масштабування (scaled speedup), оскільки ця характеристика може показати, наскільки ефективно можна організувати паралельні обчислення при збільшенні складності розв’язуваних задач.

5.1. Аналіз масштабованості паралельних обчислень

Метою застосування паралельних обчислень є не тільки зменшення тривалості розрахунків, але й забезпечення можливості розв’язку складніших варіантів задач (таких постановок, розв’язок яких є неможливим при використанні однопроцесорних обчислювальних систем). Спроможність паралельного алгоритму ефективно використовувати процесори при підвищенні складності обчислень є важливою характеристикою виконуваних розрахунків. Тому паралельний алгоритм називають масштабованим (scalable), якщо в разі росту чисельності процесорів він забезпечує збільшення прискорення при збереженні постійного рівня ефективності використання процесорів. Можливий спосіб характеристики властивостей масштабованості полягає в наступному.

Оцінимо накладні витрати (total overhead), які мають місце при виконанні паралельного алгоритму:

$$T_0 = pT_p - T_1. \qquad (2.1.51)$$

Поточні витрати з’являються за рахунок необхідності організації взаємодії процесорів, виконання деяких додаткових дій, синхронізації паралельних обчислень і т.п. Використовуючи введене позначення, можна отримати нові вирази для тривалості паралельного вирішення задачі та відповідного прискорення:

$$T_p = \frac{T_1 + T_0}{p}, \; S_p = \frac{T_1}{T_p} = \frac{pT_1}{T_1 + T_0}. \qquad (2.1.52)$$

Застосовуючи отримані співвідношення, ефективність використання процесорів можна виразити як:

$$E_p = \frac{S_p}{p} = \frac{T_1}{T_1 + T_0} = \frac{1}{1 + T_0 / T_1}. \qquad (2.1.53)$$

Останній вираз показує, що коли складність розв'язуваної задачі є фіксованою ($T_1 = const$), то при зростанні чисельності процесорів ефективність, як правило, буде спадати за рахунок зростання накладних витрат T_0. При фіксації чисельності процесорів ефективність їх використання можна покращити шляхом підвищення складності розв'язуваної задачі T_1 (вважається, що при зростанні параметра складності *n* накладні витрати T_0 збільшуються повільніше, ніж об'єм обчислень T_1). Із цього слідує, що при збільшенні чисельності процесорів в більшості випадків можна забезпечити певний рівень ефективності з використанням відповідного підвищення складності розв'язуваних задач. Тому важливою характеристикою паралельних обчислень стає співвідношення необхідних темпів росту складності розрахунків та чисельності використовуваних процесорів.

Зробимо припущення , що $E = const$ є бажаний рівень ефективності виконуваних обчислень. З виразу для ефективності можна отримати:

$$\frac{T_0}{T_1} = \frac{1 - E}{E} \text{ або } T_1 = KT_0,\ K = E/(1 - E). \qquad (2.1.54)$$

Породжувану останнім співвідношенням залежність $n = E(p)$ між складністю розв'язуваної задачі та чисельністю процесорів називають функцією ізоефективності (isoefficiency function).

Отримаємо виведення виразу для функції ізоефективності для розглянутого прикладу знаходження суми числових значень. В цьому випадку:

$$T_0 = pT_p - T_1 = p(n/p) + \log_2 p) - n = p\log_2 p \qquad (2.1.55)$$

.

і вираз для функції ізоефективності набуде вигляду:

$$n = Kp\log_2 p\,. \qquad (2.1.56)$$

Як результат, наприклад, при чисельності процесорів $p = 16$ для забезпечення рівня ефективності $E = 0.5$ (тобто $K = 1$) кількість значень суми повинна бути не менше $n = 64$. Або ж, при зростанні чисельності процесорів з p до $q(q > p)$ для забезпечення пропорційного зростання прискорення $(S_q / S_p) = (q / p)$ необхідно збільшити чисельність значень суми n в $(q\log_2 q)/((p\log_p)$ разів.

5.2. Оцінка комунікаційної трудомісткості паралельних алгоритмів

Алгоритми маршрутизації. Алгоритми маршрутизації визначають шлях передачі даних від процесора - джерела повідомлення, до процесора, якому повідомлення повинно бути доставлено. розрізняють наступні можливі варіанти рішення такої задачі;

- оптимальні, які визначають завжди найкоротші шляхи передачі даних, та неоптимальні алгоритми маршрутизації;

- детерміновані і адаптовані методи вибору маршрутів (адаптивні алгоритми визначають шляхи передачі даних в залежності від існуючого завантаження комунікаційних каналів).

До числа найпоширеніших оптимальних алгоритмів відноситься клас методів по координатної маршрутизації (dimension-ordered routing), в яких пошук шляхів передачі даних здійснюється по черзі для кожної розмірності топології мережі комунікації. Так для двомірної решітки даний підхід приводить до маршрутизації, коли передача даних спочатку виконується по одному напрямку (наприклад, по горизонталі до досягнення вертикалі, на якій розташований процесор призначення), а далі дані передаються по другого напрямку (ця схема відома під назвою алгоритму XY-маршрутизації). Для гіперкуба покоординатна схема маршрутизації може полягати, наприклад, в циклічній передачі даних процесору, який визначається першою відмінною бітовою позицією в номерах процесорів -

того, на якому повідомлення розміщується в даний момент часу, та того, в який воно повинно бути передане.

Методи передачі даних. Тривалість передачі даних між процесорами визначає комунікаційну складову (communication latency) тривалості виконання паралельного алгоритму в багатопроцесорній обчислювальній системі. Основний набір параметрів, що описують тривалість передачі даних, складається з наступного ряду величин:

- тривалість початкової підготовки ($t_п$) характеризує тривалість підготовки повідомлення для передачі, пошуку маршруту в мережі і т.п.;

- тривалість передачі службових даних ($t_с$) між двома сусідніми процесорами (тобто для процесорів, між якими є фізичний канал передачі даних). До службових даних може відноситися заголовок повідомлення, блок даних для виявлення похибок передачі і т.п.;

- час передачі одного слова даних по одному каналу передачі даних ($t_к$). Тривалість подібної передачі визначається смугою пропускання комунікаційних каналів в мережі.

До числа найпоширеніших методів передачі даних відносяться два основних способи комунікації. Перший з них орієнтований на передачу повідомлень (метод передачі повідомлень, або МПС) як неподільних (атомарних) блоків інформації (store and-forward routing або SFR). При такому підході процесор, в якому знаходиться повідомлення для передачі, готує весь об'єм даних для передачі, визначає процесор, якому слід направити дані, і запускає операцію пересилання даних. Процесор, якому направлено повідомлення, в першу чергу здійснює прийом повністю всіх даних і тільки потім приступає до пересилання прийнятого повідомлення далі за означеним маршрутом. Тривалість пересилання даних $t_{пд}$ для методу передачі повідомлення розміром m байтів за маршрутом довжиною l визначається виразом:

$$t_{пд} = t_п + (mt_к + t_с)l\,. \qquad (2.2.1)$$

За умови достатньо довгих повідомлень тривалістю передачі службових даних можна знехтувати і вираз для тривалості передачі даних можна записати в простішому вигляді:

$$t_{пд} = t_н + mt_k l\,. \qquad (2.2.2)$$

Другий спосіб комунікації базується на представленні повідомлень, що пересилаються, у вигляді блоків інформації меншого розміру - пакетів, в результаті чого передача даних може бути зведена до передачі пакетів (метод передачі пакетів або МПП). За умови такого методу комунікації (cut-through routing або CTR) приймаючий процесор, може здійснювати пересилку даних за подальшим маршрутом безпосередньо відразу після прийому чергового пакету, не очікуючи завершення прийому даних всього повідомлення. Тривалість пересилання даних при використанні методу передачі пакетів визначається виразом:

$$t_{пд} = t_y + mt_k + t_c l\,. \qquad (2.2.3)$$

Порівнюючи отримані вирази, можна помітити, що в більшості випадків метод передачі пакетів приводить до більш швидкого пересилання даних; крім того, даний підхід знижує потребу в пам’яті для зберігання даних, що пересилаються, при організації прийому-передачі повідомлень, а для передачі пакетів можуть використовуватися одночасно різні комунікаційні канали. З іншого боку, реалізація пакетного методу потребує розробки більш складного апаратного та програмного забезпечення мережі і може збільшити накладні витрати (тривалість підготовки та тривалість передачі службових даних). Крім того, при передачі пакетів можливе виникнення конфліктних ситуацій (дедлоків).

Аналіз трудомісткості основних операцій передачі даних. При всьому розмаїтті виконуваних операцій передачі даних при паралельних способах розв’язку складних науково-технічних задач, певні процедури взаємодії процесів мережі можна віднести до числа основних комунікаційних дій, або найбільш широко розповсюджених на практиці паралельних

обчислень, або тих, до яких можна віднести багато інших процесів прийому-передачі повідомлень. Важливо відмітити, що в рамках подібного базового набору для більшості операцій комунікації існують процедури, обернені за дією вихідним операціям. Як результат, аналіз комунікаційних процедур доцільно виконувати попарно.

Передача даних між двома процесорами мережі. Складність цієї комунікаційної операції може бути наслідком підстановки довжини максимального шляху (діаметра мережі) в вирази для тривалості передачі даних при різних методах комунікації, табл. 5.1

Таблиця 5.1. Тривалість передачі даних між двома процесорами

Топологія	**Передача повідомлень**	**Передача пакетів**
Кільце	$t_н + mt_k \lfloor p/2 \rfloor$	$t_н + mt_k + t_c \lfloor p/2 \rfloor$
Решітка-тор	$t_н + 2mt_k \lfloor p/2 \rfloor$	$t_н + mt_k + 2t_c \lfloor p/2 \rfloor$
Гіперкуб	$t_н + mt_k \log_2 p$	$t_н + mt_k + t_c \log_2 p$

Передача даних від одного процесора всім іншим процесорам мережі.

Операція передачі даних (одного і того ж сполучення) від одного процесора всім іншим процесорам мережі (one-to-all broadcast або single-node broadcast) є однією з найчастіше виконуваних комунікаційних дій. Подібні операції використовуються, зокрема, при реалізації матрично-векторного множення, розв'язку систем лінійних рівнянь методом Гауса, розв'язку задачі пошуку найкоротших шляхів та ін.

Найпростіший спосіб реалізації операції розсилання полягає в її виконанні як послідовності попарних взаємодій процесорів мережі. Проте за такого підходу більша частина пересилань є надлишковою і можливе застосування більш ефективних алгоритмів комунікації.

Передача повідомлень. Для кільцевої технології процесор - джерело розсилання може ініціювати передачу даних відразу двом своїм сусідам, які, в свою чергу, прийнявши повідомлення, організують пересилання далі по

колу. Трудомісткість виконання операції розсилання в цьому випадку визначатиметься співвідношенням:

$$t_{пд} = (t_н + mt_k)\lceil \sqrt{p}/2 \rceil - 1. \qquad (2.2.4)$$

Для топології типу решітка-тор алгоритм розсилання можна отримати із способу передачі даних, застосованого для кільцевої структури мережі. Так, розсилка може бути виконана у вигляді двохетапної процедури. на першому етапі організується передача повідомлень всім процесорам мережі, розташованим на тій же горизонталі решітки, що й процесор - ініціатор передачі. На другому етапі процесори, які отримали копію даних під час виконання першого етапу, розсилають повідомлення по своїм відповідним вертикалям. Оцінка тривалості операції розсилки у відповідності з описаним алгоритмом визначається співвідношенням:

$$t_{пд} = 2(t_н + mt_k)\lceil \sqrt{p}/2 \rceil - 1. \qquad (2.2.5)$$

Для гіперкубу розсилка може бути виконана в ході N - етапної процедури передачі даних. На першому етапі процесор-джерело повідомлення передає дані одному з своїх сусідів (наприклад, по першій розмірності) - в результаті після першого етапу є два процесори, які мають копію даних, що пересилаються (цей результат можна інтерпретувати також як розбиття вихідного гіперкуба на два таких однакових за розміром гіперкуби з розмірністю *N-1,* тому що кожний з них має копію вихідного повідомлення). На другому етапі два процесори, задіяні на першому етапі, пересилають повідомлення своїм сусідам по другій розмірності і т.д. в результаті такого розсилання тривалість операції оцінюється з використанням виразу:

$$t_{пд} = (t_н + mt_k)\log_2 p. \qquad (2.2.6)$$

Порівнюючи отримані вирази для тривалості виконання операції розсилання, можна відмітити, що найкращі показники мають технології типу гіперкуб; більше того, можна показати, що такий результат є найкращим для вибраного способу комунікації з використанням повідомлень.

Передача пакетів. Для топології типу кільце алгоритм розсилання можна отримати шляхом логічного зображення кільцевої структури мережі у вигляді гіперкубу. В результаті на етапі розсилання процесор - джерело повідомлення передає дані процесору, який знаходиться на відстані $p/2$ від вихідного процесора. Далі, на другому етапі обидва процесори, які вже мають дані, що розсилаються, після першого етапу, передають повідомлення процесорам, які знаходяться на відстані $p/4$, і т.д. Трудомісткість виконання операції розсилання при використанні такого методу передачі даних визначається співвідношенням:

$$t_{пд} = \sum_{i=1}^{\log_2 p}(t_н + mt_k + t_c p/2^i) = (t_н + mt_k)\log_2 p + t_c(p-1) \qquad (2.2.7)$$

(як і раніше, за достатньо великих повідомленнях тривалістю передачі службових даних можна знехтувати).

Для топології типу решітка-тор алгоритм розсилання можна отримати із способу передачі даних, застосованого для кільцевої структури мережі, у відповідності з тим же способом, що і у випадку використання методу передачі повідомлень. Отриманий в результаті такого узагальнення алгоритм розсилання характеризується таким співвідношенням для оцінки тривалості виконання:

$$t_{пд} = (t_н + mt_k)\log_2 p + 2t_c(\sqrt{p} - 1). \qquad (2.2.8)$$

Для гіперкуба алгоритм розсилання (і часові оцінки тривалості виконання) при передачі пакетів не відрізняється від варіанту для методу передачі повідомлень.

Передача даних від всіх процесорів всім процесорам мережі. Операція передачі даних від всіх процесорів всім процесорам мережі (all-to-all broadcast або multinode broadcast) є звичайним узагальненням одиночної операції розсилки..

Можливий спосіб реалізації множинного розсилання полягає у виконанні відповідного набору операцій одиночного розсилання. Проте

такий підхід не є оптимальним для багатьох топологій мережі, оскільки частина необхідних операцій одиночного розсилання потенційно може бути виконана паралельно.

Передача повідомлень. Для кільцевої топології кожний процесор може ініціювати розсилання свого повідомлення одночасно (в якомусь вибраному напрямку по колу). В будь-який момент кожен процесор виконує прийом і передачу даних, завершення операції множинного розсилання відбудеться через $p-1$ циклів передачі даних.

Тривалість виконання операції розсилання оцінюється співвідношенням:

$$t_{nд} = (t_н + mt_k)(p-1). \qquad (2.2.9)$$

Для топології типу решітка-тор множинне розсилання повідомлень може бути виконана з використанням алгоритму, отриманого узагальненням способу передачі даних для кільцевої структури мережі. Алгоритм узагальнення полягає в наступному. На першому етапі організується передача повідомлень розділено по всім процесорам мережі, які розташовані на одних і тих же горизонталях решітки (в результаті на кожному процесорі однієї і тієї ж горизонталі формуються укрупнені повідомлення з розміром $m\sqrt{p}$, які об'єднують всі повідомлення горизонталі). Тривалість виконання етапу:

$$t'_{nд} = (t_н + mt_k)(\sqrt{p}-1). \qquad (2.2.10)$$

На другому етапі розсилання даних виконується тим процесорам мережі, які утворюють вертикалі решітки. Тривалість цього етапу:

$$t''_{nд} = (t_н + m\sqrt{p}t_k)(\sqrt{p}-1). \qquad (2.2.11)$$

Загальна тривалість операції розсилання визначається співвідношенням:

$$t_{nд} = 2t_н(\sqrt{p}-1) + mt_k(p-1). \qquad (2.2.12)$$

Для гіперкуба алгоритм множинного розсилання повідомлень можна отримати узагальненням раніше описаного способу передачі даних для

топології типу решітки на розмірність гіперкуба N. В результаті такого узагальнення схема комунікації полягатиме в наступному. На кожному етапі $i, 1 \le i \le N$, виконання алгоритму функціонують всі процесори мережі, які обмінюються своїми даними із своїми сусідами i-ої розмірності і формують об'єднані повідомлення. Тривалість операції розсилання може бути отримана з використанням виразу:

$$t_{n\partial} = \sum_{i=1}^{\log_2 p} (t_н + 2^{i-1} m t_k) = t_н \log_2 p + m t_k (p-1). \qquad (2.2.13)$$

Передача пакетів. Застосування більш ефективного для кільцевої структури та топології типу решітка-тор методу передачі даних не приводить до якогось поліпшення тривалості виконання операції одиночного розсилання, а у випадку множинного розсилання приводить до перевантаження каналів передачі даних (тобто, до існування ситуацій, коли в один і той же момент для передачі по одній і тій же лінії є декілька пакетів даних, що очікують). Перевантаження каналів приводить до затримок при пересиланні даних, що і не дозволяє проявити всі переваги методу передачі пакетів. Широко поширеним прикладом операції множинного розсилання є задача редукції (reduction), яка визначається в загальному вигляді як процедура виконання тієї чи іншої обробки даних, отриманих на кожному процесорі в ході множинного розсилання. Способи розв'язку задачі редукції полягають в наступному:

- безпосередній підхід полягає у виконанні операції множинного розсилання і, після цього, обробки даних на кожному процесорі зокрема;

- більш ефективний алгоритм можна отримати в результаті застосування операції одиночного прийому даних на окремому процесорі, виконання на цьому процесорі дій з обробки даних і розсилання отриманого результату обробки всім процесорам мережі;

- найкращий спосіб розв'язку задачі редукції полягає в суміщенні процедури множинного розсилання та дій з обробки даних, коли кожний процесор відразу після прийому чергового повідомлення реалізує потрібну

обробку отриманих даних (наприклад, виконує додавання отриманого значення з наявною на процесорі частковою сумою. Тривалість рішення задачі редукції при такому алгоритмі реалізації у випадку, наприклад, коли розмір даних, що пересилаються, має одиничну довжину ($m = 1$) і топологія мережі має структуру гіперкуба, визначається співвідношенням:

$$t_{пд} = (t_н + t_k)\log_2 p. \qquad (2.2.14)$$

Іншим типовим прикладом використання операції множинного розсилання є задача знаходження часткових сум послідовності значень S_i:

$$S_k = \sum_{i=1}^{k} x_i, 1 \le k \le p \qquad (2.2.15)$$

Вважається, що кількість значень співпадає з кількістю процесорів, значення x_i розташовується на i – у процесорі, а результат S_k повинен отримуватися у процесорі з номером k. Алгоритм розв'язку цієї задачі також можна отримати з використанням конкретизації загального способу виконання множинної операції розсилання, коли процесор виконує додавання отриманого значення (але тільки в тому випадку, якщо процесор – відправник має менший номер, ніж процесор – одержувач.

Узагальнена передача даних від одного процесора всім іншим процесорам мережі. Загальний випадок передачі даних від одного процесора всім іншим процесорам мережі полягає в тому, що всі повідомлення, які розсилаються є різними (one-to-all personalized communication чи single-node scatter). Операція передачі даних для даного типу взаємодії процесорів - узагальнений прийом повідомлень (single-node gather) на одному процесорі від всіх інших процесорів мережі (відмінність цієї операції від раніше розглянутої процедури збірки даних на одному процесорі полягає в тому, що узагальнена операція збірки не передбачає якоїсь взаємодії повідомлень (наприклад, редукції) в процесі передачі даних). Трудомісткість операції узагальненого розсилання порівнювана із складністю виконання процедури множинної передачі даних. Процесор - ініціатор розсилання посилає

кожному процесору мережі повідомлення з розміром m, і, тим самим, нижня оцінка тривалості виконання операції характеризується величиною $mt_k(p-1)$.

Проведемо аналіз трудомісткості узагальненої розсилки для випадку топології типу гіперкуб. Можливий спосіб виконання операції полягає в наступному. Процесор - ініціатор розсилання передає половину своїх повідомлень одному з своїх сусідів (наприклад, за першою розмірністю) - в результаті вихідний гіперкуб стає розділеним на два гіперкуби половинного розміру, в кожному з яких міститься рівно половина вихідних даних. Далі, дії з розсилання повідомлень можуть бути повторені, і загальна кількість повторень визначається вихідною розмірністю гіперкуба. Тривалість операції узагальненого розсилання можна характеризувати співвідношенням:

$$t_{по} = t_y \log_2 p + mt_k(p-1) \qquad (2.2.16)$$

(як і відмічалося раніше, трудомісткість операції співпадає з тривалістю виконання процедур множинного розсилання).

Узагальнена передача даних від всіх процесорів всім процесорам мережі. Цей тип передачі даних представляє собою найбільш загальний випадок комунікаційних дій. Необхідність виконання подібних операцій виникає в паралельних алгоритмах швидкого перетворення Фур'є, транспонування матриць та ін.

Передача повідомлень. Загальна схема алгоритму для кільцевої топології полягає в наступному. Кожний процесор здійснює передачу всіх своїх вихідних повідомлень своєму сусідові (в якомусь вибраному напрямку по колу). Далі процесори здійснюють прийом направлених до них даних, далі серед прийнятої інформації вибирають свої повідомлення, після чого виконують подальше розсилання частини даних, що розсилаються. Тривалість виконання подібного набору передачі даних оцінюється згідно виразу:

$$t_{по} = (t_н + \frac{1}{2} mpt_k)(p-1)\,. \qquad (2.2.17)$$

Спосіб отримання алгоритму розсилання даних для топології решітка-тор той же самий, що і у випадку розгляду інших комунікаційних операцій. На першому етапі організується передача повідомлень окремо по всім процесорам мережі, які розташовані на одних і тих же горизонталях решітки (кожному процесору по горизонталі передаються тільки ті вихідні повідомлення, що повинні бути направлені процесорам відповідної вертикалі решітки). Після завершення етапу на кожному процесорі збираються p повідомлень, призначених для розсилання по одній вертикалі решітки. На другому етапі розсилання даних виконується по процесорам мережі, які утворюють вертикалі решітки. Загальна тривалість всіх операцій розсилань визначається співвідношенням:

$$t_{пд} = (2t_н + mpt_k)(\sqrt{p} - 1). \qquad (2.2.18)$$

Для гіперкуба алгоритм узагальненого множинного розсилання повідомлень можна отримати шляхом узагальнення способу виконання операції для топології типу решітка на розмірність гіперкуба N. В результаті такого узагальнення схема комунікації полягатиме в наступному. На кожному етапі $i, 1 \le i \le N$, виконання алгоритму функціонують всі процесори мережі, які обмінюються своїми даними із своїми сусідами по i - й розмірності і формують об'єднані повідомлення. За організації взаємодії двох сусідів канал зв'язку між ними розглядається як сполучний елемент двох рівних за розміром полів гіперкубів вихідного гіперкуба, і кожний процесор пари посилає іншому процесору тільки ті повідомлення, що призначені для процесорів сусіднього гіперкуба. Тривалість операції розсилання можна отримати з використання виразу:

$$t_{пд} = (t_н + \frac{1}{2} mpt_k)\log_2 p \qquad (2.2.19)$$

(крім витрат на пересилання, кожний процесор виконує $mp\log_2 p$ операцій і сортування своїх повідомлень перед обміном інформацією з своїми сусідами).

Передача пакетів. Як і у випадку множинного розсилання, застосування методу передачі пакетів не приводить до покращення часових характеристик для операції узагальненого множинного розсилання. Для прикладу розглянемо застосування методу виконання даної комунікаційної операції для мережі з топологією типу гіперкуб. В цьому випадку розсилання можна виконати за $p-1$ ітерацію. Під час кожної ітерації всі процесори розбиваються на взаємодіючі пари процесорів, причому це розбиття на пари може бути виконане таким чином, щоб повідомлення, які передаються парами, не використовували одні і ті ж шляхи передачі даних. Як результат, спільну тривалість операції узагальненого розсилання можна визначити у відповідності з виразом:

$$t_{no} = (t_н + mt_k)(p-1) + \frac{1}{2} t_c p \log_2 p . \qquad (2.2.20)$$

Циклічний зсув. Окремий випадок узагальненого множинного розсилання є процедурою перестановки (permutation), який яавляє собою операцію перерозподілу інформації між процесорами мережі, в якій кожний процесор передає повідомлення визначеному певним способом іншому процесору мережі. Конкретний варіант перестановки - циклічний q - зсув (*circular q - shift*), коли кожний процесор $i, 1 \le i \le N$, передає дані процесору з номером $(i+q) \bmod p$. Подібна операція зсуву використовується, наприклад, при організації матричних обчислень.

Оскільки виконання циклічного зсуву для кільцевої топології може бути забезпечене із використанням простих алгоритмів передачі даних, розглянемо можливі способи виконання даної комунікаційної операції тільки для топологій решітка-тор та гіперкуб з різними методами передачі даних.

Передача повідомлень. Загальна схема алгоритму циклічного зсуву для топології решітка-тор полягає в наступному. Нехай процесори пронумеровані за стрічками решітки від 0 до $p-1$. На першому етапі організується циклічний зсув з кроком $q \bmod \sqrt{p}$ по кожному рядку зокрема (якщо при реалізації такого зсуву повідомлення передаються через праві

границі рядків, то після виконання кожної такої передачі необхідно здійснити компенсаційний зсув вгору на 1 для процесорів першого стовпця решітки). На другому етапі реалізується циклічний зсув вгору з кроком $\lfloor q/\sqrt{p} \rfloor$ для кожного стовпця решітки. Загальна тривалість всіх операцій розсилань визначається співвідношенням:

$$t_{пд} = (t_н + mt_k)(2\lfloor \sqrt{p}/2 \rfloor + 1). \qquad (2.2.21)$$

Для гіперкуба алгоритм циклічного зсуву можна отримати шляхом логічного представлення топології гіперкуба у вигляді кільцевої структури. Для отримання такого представлення встановимо взаємно-однозначну відповідність між вершинами кільця та гіперкуба. Необхідна відповідність може бути отримана, наприклад, з використанням відомого коду Грея. Детальніший виклад механізму встановлення такої відповідності здійснено нижче. На рис. 5.8 показано вигляд гіперкуба для розмірності $N = 3$ із зазначенням для кожного процесора гіперкуба відповідної вершини кільця.

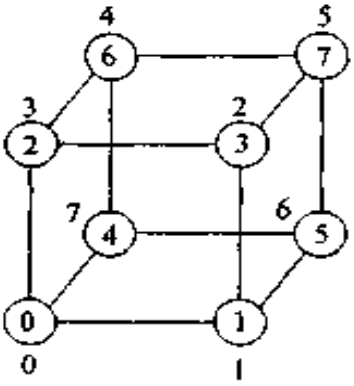


Рисунок 5.8 - Схема відображення гіперкуба на кільце (в кружках наведені номери процесорів гіперкуба)

Позитивною властивістю вибору такої відповідності є той факт, що для будь-яких двох вершин в кільці, відстань між якими дорівнює $l = 2^i$ для деякого значення *i*, шлях між відповідними вершинами в гіперкубі містить тільки дві лінії зв'язку (за виключенням випадку $i = 0$, коли шлях в гіперкубі має одиничну довжину).

Припустимо, що величина зсуву q має вигляд двійкового коду. Кількість ненульових позицій коду визначає кількість етапів в схемі реалізації операції циклічного зсуву. На кожному етапі виконується операція зсуву з величиною кроку, яка задається найстаршою ненульовою позицією значення q (наприклад, за умови вихідної величини зсуву $q = 5 = 101_2$ на першому етапі виконується зсув з кроком 4, на другому етапі крок зсуву дорівнює 1. Виконання кожного етапу (окрім зсуву з кроком 1) полягає в передачі даних по каналу, який складається з двох ліній зв'язку. Як результат, верхня оцінка для тривалості виконання операції циклічного зсуву визначається співвідношенням:

$$t_{пд} = (t_н + mt_k)(2\log_2 p - 1). \qquad (2.2.22)$$

Передача пакетів. Використання пересилання пакетів може підвищити ефективність виконання операції циклічного зсуву для топології гіперкуб. Реалізація всіх необхідних комунікаційних дій в цьому випадку може бути забезпечена шляхом відправлення кожним процесором всіх даних, що пересилаються, безпосередньо призначеним процесорам. Застосування методу по координатної маршрутизації дасть змогу уникнути колізій при використанні ліній передачі даних (в кожний момент часу для кожного каналу буде існувати не більше одного готового для відправки повідомлення). Довжина найбільшого шляху за умови такого розсилання даних визначається як $\log_2 p - \gamma(q)$, де $\gamma(q)$ - найбільше ціле значення j таке, що 2^j - дільник величини зсуву q. Тоді тривалість операції циклічного зсуву можна охарактеризувати з використанням виразу:

$$t_{пд} = t_н + mt_k + t_c(\log_2 p - \gamma(q)). \qquad (2.2.23)$$

(за умови достатньо великих розмірів повідомлень тривалістю передачі службових даних можна знехтувати і тоді тривалість виконання операції можна визначити як $t_{пд} = t_н + mt_k$).

Методи логічного зображення топології комунікаційного середовища. Ряд алгоритмів передачі даних припускає простіше рішення в разі використання певних топологій мережі міжпроцесорних з'єднань. Крім того, багато методів комунікації можна отримати з використанням того чи іншого логічного зображення використовуваної топології. Як результат, важливою при організації паралельних обчислень є можливість логічного зображення різноманітних топологій на основі конкретних, фізичних, міжпроцесорних структур.

Способи логічного зображення (відображення) топологій характеризується наступними трьома основними характеристиками:

- ущільнення дуг (congestion), яке виражається як максимальна кількість дуг логічної топології, які відображаються в одну лінію передачі фізичної топології;

- подовження дуг (dilation), яке визначається як шлях максимальної довжини фізичної топології, на якій відображається дуга логічної топології;

- збільшення вершин (expansion), що обчислюється як відношення кількості вершин в логічній та фізичній топологіях.

Для огляду топологій обмежимося питаннями відображення топологій кільця та решітки на гіперкуб.

Представлення кільцевої топології у вигляді гіперкуба. Встановлення відповідності між кільцевою топологією та гіперкубом можна реалізувати з використанням двійкового рефлексивного коду Грея $G(i,N)$ (binary reflected Gray code), який визначається у відповідності з виразами:

$$G(0,1)=0, G(1,1)=1,$$

$$G(i,s)=\begin{cases} G(i,s), i<2^s, \\ 2^s+G(2^{s+1}-1-i,s), i\geq 2^s, \end{cases} \qquad (2.2.24)$$

де i - задає номер значення в коді Грея, а N є довжиною цього коду. Для ілюстрації на рис. 5.8. показано відображення кільцевої топології на гіперкуб для мережі з $p=8$ процесорів.

Код Грея для N=1	Код Грея для N=1	Код Грея для N=1	Номери процесорів	
			гіперкуба	кільця
0	0 0	0 0 0	0	0
1	0 1	0 0 1	1	1
	1 1	0 1 1	3	2
	1 0	0 1 0	2	3
		1 1 0	6	4
		1 1 1	7	5
		1 0 1	5	6
		1 0 0	4	7

Важлива властивість коду Грея: сусідні значення $G(i,N)$ та $G(i+1,N)$ мають тільки одну бітову позицію, що різниться. Як результат, сусідні вершини в кільцевій технології відображаються на сусідні процесори в гіперкубі.

Відображення топології решітки на гіперкуб. Відображення топології решітки на гіперкуб можна виконати в рамках підходу, використаного для кільцевої структур мережі. Тоді для відображення решітки $2^r \times 2^s$ на гіперкуб розмірності $N = r + s$ можна прийняти правило, що елементу решітки з координатами (i, j) відповідає процесор гіперкуба з номером:

$G(i,r)\|G(j,s)$, де операція || означає конкатенацію кодів Грея.

Оцінка трудомісткості операцій передачі даних для кластерних систем. Для кластерних обчислювальних систем одним з широко застосовуваних способів побудови комунікаційного середовища є використання концентраторів (hub) або (switch) для об'єднання процесорних вузлів кластера в єдину обчислювальну мережу. В цих випадках топологія мережі кластера являє собою повний граф, в якому є певні обмеження на одночасність виконання комунікаційних операцій. Так, за умови використання концентраторів передача даних в кожний поточний момент може виконуватися тільки між двома процесорними вузлами; комунікатори можуть забезпечувати взаємодію декількох пар процесорів, що не перетинаються. Інший частовикористовуваний напрямок при створенні кластерів полягає у використанні методу передачі пакетів (часто реалізується

на основі стеку протоколів TCP/IP) в якості основного способу виконання комунікаційних операцій.

Якщо вибрати для подальшого аналізу кластери цього поширеного типу (топологія у вигляді повного графа, пакетний спосіб передачі повідомлень), то трудомісткість операції комунікації між двома процесорними вузлами можна оцінити у відповідності з виразом:

$$t_{по}(m) = t_н + m^* t_k + t_c; \qquad (2.2.25)$$

оцінка подібного типу випливає із співвідношень для методу передачі пакетів при одиничній довжині шляху передачі даних, тобто $l = 1$. Відмічаючи можливість подібного підходу, разам з тим слід зазначити, що в рамках розглядуваної моделі тривалість підготовки даних $t_н$ вважається сталою (не залежить від об'єму даних, що передаються), тривалість передачі службових даних t_c не залежить від кількості пакетів, що передаються, і т.п. Ці припущення не в повній мірі відповідають дійсності, і часові оцінки, що отримуються в результаті використання моделі, не можуть мати необхідну точність.

З врахуванням наведених зауважень, схема побудови часових оцінок може бути уточнена; в рамках нової розширеної моделі трудомісткість передачі даних між двома процесорами визначається у відповідності з такими виразами:

$$t_{пд} = \begin{cases} t_{поч0} + t_{поч1} + (m + V_c)t_k + (m + V_c)t_k, n = \\ t_{поч0} + (V_{\max} - V_c)t_{поч1} + (m + V_c n)t_k, n > \end{cases} \qquad (2.2.26)$$

де $n = \lfloor m/(V_{\max} - V_c) \rfloor$ є кількістю пакетів, на яку розбивається повідомлення, що передається, величина $V_{\max}$ визначає максимальний розмір пакета який може бути доставлений в мережу, а V_c - об'єм службових даних в кожному з пакетів, що пересилаються. В наведених співвідношеннях константа $t_{поч0}$ характеризує апаратну складову латентності і залежить від параметрів використовуваного мережевого обладнання, значення $t_{поч1}$ задає

тривалість підготовки одного байта даних для передачі по мережі. Як результат, величина латентності:

$$t_н = t_{поч0} + vt_{поч1}, \qquad (2.2.27)$$

збільшується лінійно в залежності від об'єму даних, що передаються. При цьому припускається, що підготовка даних для передачі другого та всіх інших пакетів може бути об'єднаною з пересиланням по мережі попередніх пакетів і латентність не може перевищувати величини:

$$t_н = t_{поч0} + (V_{\max} - V_c)t_{поч1}. \qquad (2.2.28)$$

Окрім латентності, в запропонованих виразах для оцінки трудомісткості комунікаційної операції можна уточнити також правило обчислення тривалості передачі даних:

$$(m + V_c n)t_k, \qquad (2.2.29)$$

це дає змогу враховувати ефект збільшення об'єму даних, що передаються, при зростанні числа пакетів, що пересилаються, за рахунок додавання службової інформації (заголовків пакетів).

Для практичного застосування перерахованих моделей необхідно виконати оцінку значень параметрів використовуваних співвідношень. В цьому відношенні корисним може виявитися використання більш простіших способів обчислення часових витрат на передачу даних - однією з відомих схем подібного типу є підхід, в якому трудоємність операції комунікації між двома процесорними вузлами кластера оцінюється у відповідності з виразом:

$$t_{пд}(m) = t_н + mt_k, \qquad (2.2.30)$$

дану модель , запропонована Хокні (the Hockney model).

6 ПРИНЦИПИ РОЗРОБКИ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

Розробка алгоритмів (особливо методів паралельних обчислень) для розв’язку складних задач часто є суттєвою проблемою. Ефективні способи організації паралельних обчислень можуть полягати в наступному:

- виконання аналізу наявних обчислювальних схем і здійснення їх розподілу (декомпозиції) на частини (підзадачі), які можна реалізувати з певним ступенем незалежно одна від одної;

- виділення для сформованого набору підзадач інформаційних взаємодій, які повинні здійснюватися під час вирішення вихідної поставленої задачі;

- визначення необхідної (або доступної) для розв’язку задачі обчислювальної системи і виконання розподілу наявного набору підзадач між процесорами системи.

За умови самого узагальненого аналізу зрозуміло, що об’єм обчислень для кожного використовуваного процесора повинен бути приблизно однаковий - це дасть змогу забезпечити рівномірне обчислювальне завантаження (балансування) процесорів. Крім того зрозуміло, що розподіл підзадач між процесорами повинен бути виконаний таким чином, щоб кількість інформаційних зв’язків (комунікаційних взаємодій) між підзадачами було мінімальним. Після виконання всіх перерахованих етапів проектування можна оцінити ефективність розроблюваних паралельних методів: для цього визначаються показники якості породжуваних паралельних обчислень (прискорення, ефективність, масштабованість). За результатами проведеного аналізу може виявитися, що виникне необхідність повторення окремих (або в деяких випадках всіх) етапів розробки. Повернення до попередніх кроків розробки може відбуватися на будь-якій стадії проектування паралельних обчислювальних схем.

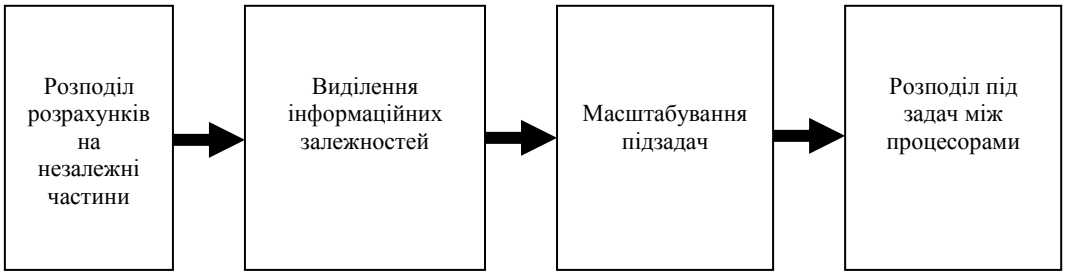


Рисунок 6.1 - Загальна схема розробки паралельних алгоритмів

Тому часто додатковою дією в наведеній вище схемі проектування, рис. 6.1, є коригування складу сформованої множини задач після визначення наявної кількості процесорів - підзадачі можуть бути укрупнені (агреговані) за наявності малої кількості процесорів, або навпаки, деталізовані в іншому випадку. В цілому, ці дії можна охарактеризувати як масштабування розроблюваного алгоритму або як окремий етап проектування паралельних обчислень.

Для застосування отриманого в кінцевому підсумку паралельного методу, необхідно виконати розробку програм для розв’язку сформованого набору підзадач і розмістити розроблені програми між процесорами у відповідності з вибраною схемою розподілу підзадач. Для проведення обчислень програми запускаються на виконання (програми на стадії виконання іменуються процесами), для реалізації інформаційних взаємодій програми повинні мати в своєму розпорядженні засоби обміну даними (канали передачі повідомлень). Слід відмітити, що кожний процесор виділяється для розв’язання єдиної підзадачі, проте за наявності великої кількості підзадач або використання обмеженої кількості процесорів це правило може не витримуватися і, в результаті, на процесорах може виконуватися одночасно декілька програм (процесів). Зокрема при розробці та початковій перевірці паралельної програми для виконання всіх процесів може використовуватися один процесор (за умови розташування на одному процесорі процеси виконуються в режимі розділення часу).

Слід відмітити, що такий підхід в значній мірі орієнтований на обчислювальні системи з розподіленою пам’яттю, коли необхідні інформаційні взаємодії реалізуються за допомогою передачі повідомлень по каналах зв’язку між процесорами. Тим не менше ця схема може бути застосована без втрати ефективності паралельних обчислень і для розробки паралельних методів для систем з загальною пам’яттю - в цьому випадку механізми передачі повідомлень для забезпечення інформаційних взаємодій повинні бути замінені операціями доступу до спільних (розділених) змінних.

Моделювання паралельних програм. Розглянута схема проектування і реалізації паралельних обчислень дає розуміння паралельних алгоритмів і програм. На стадії проектування паралельний метод може бути представлений у вигляді графа "підзадачі - повідомлення", який являє собою не що інше, як укрупнене (агреговане) представлення графа інформаційних залежностей (графа "операції - операнди").

Аналогічно на стадії виконання для опису паралельної програми можна використати модель у вигляді графа "процеси - канали", в якій замість підзадач використовується поняття процесор, а інформаційні залежності замінюються каналами передачі повідомлень. Додатково на базі цієї моделі можна показати розподіл процесів між процесорами обчислювальної системи, якщо кількість підзадач перевищує чисельність процесорів.

Використання двох моделей паралельних обчислень дають змогу краще розмежувати проблеми, які виникають при розробці паралельних методів. Перша модель - граф "підзадачі - повідомлення" - дає змогу зосередитися на питаннях виділення підзадач однакової обчислювальної складності, забезпечуючи при цьому низький рівень інформаційної складності між підзадачами. Друга модель - граф "процеси - канали" - концентрує увагу на питаннях розподілу підзадач між процесорами, забезпечуючи ще одну можливість зниження трудомісткості інформаційних взаємодій між підзадачами за рахунок розміщення на одних і тих же процесорах інтенсивно взаємодіючих процесів. Крім того, ця модель дає

змогу краще аналізувати ефективність розробленого паралельного методу і забезпечує можливість більш адекватного опису процесу виконання паралельних обчислень. Сутність додаткових пояснень для використовуваних понять в моделі "процеси-канали" наступна:

- під процесом розуміють виконувану в процесорі програму, яка використовує для своєї роботи частину локальної пам’яті процесора і містить ряд операцій прийому/передачі даних для організації інформаційної взаємодії з іншими виконуваними процесами паралельної програми;

- канал передачі даних з логічної точки зору можна розглядати як чергу повідомлень, в яку один або декілька процесів можуть відправляти дані, що пересилаються, і з яких процес - адресат може добувати повідомлення, які відправляються іншими процесами.

В загальному випадку, можна вважати, що канали виникають динамічно в момент виконання першої операції прийому/передачі каналом. В залежності від ступеню узагальненості канал може відповідати одній або декільком командам прийому даних процеса - одержувача; аналогічно при передачі повідомлень канал може використовуватись однією або кількома командами передачі даних одного чи кількох процесів. Для зниження складності моделювання і аналізу паралельних методів вважатимемо, що ємність каналів необмежена і, як результат, операції передачі даних виконуються практично без затримок простим копіюванням повідомлень в канал. З іншого боку, операції прийому повідомлень можуть приводити до затримок (блокування), якщо запитувані з каналу дані ще не були відправлені процесами - джерелами повідомлень. Слід мати на увазі важливу перевагу розглянутої моделі "процеси-канали" - тут проводиться чітке розмежування локальних (виконуваних на окремому процесорі) обчислень та дій з організації інформаційної взаємодії одночасно виконуваних процесів. Такий підхід значно знижує складність аналізу ефективності паралельних методів та істотно спрощує проблеми розробки паралельних програм.

Етапи розробки паралельних алгоритмів. Розглянемо детальніше викладену вище методику розробки паралельних алгоритмів. Ця методика включає етапи виділення підзадач, визначення інформаційних залежностей, масштабування і розподіл підзадач між процесорами обчислювальної системи.

Розділення обчислень на незалежні частини. Вибір способу розподілу на незалежні частини базується на аналізі обчислювальної схеми вирішення вихідної задачі. Вимоги, яким повинен задовольняти вибраний підхід - в забезпеченні рівного об'єму обчислень у виділених підзадачах, та мінімуму інформаційних залежностей між цими підзадачами (за інших рівних умов слід віддавати перевагу рідким операціям передачі повідомлень великого розміру порівняно з частими пересиланнями даних невеликого об'єму). В загальному випадку, проведення аналізу і виділення задач являє собою достатньо складну проблему. Вирішенню цього питання допомагає існування двох типів схем, рис.6.2.

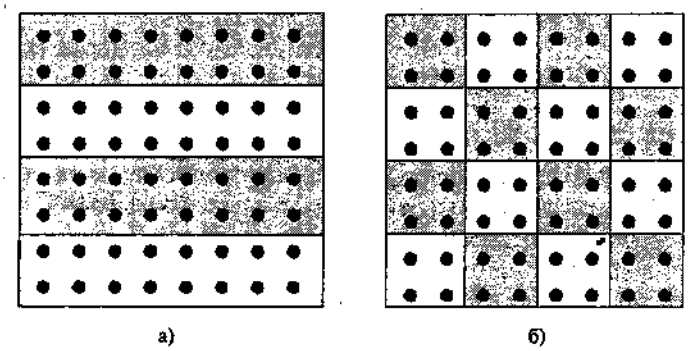


Рисунок 6.2 - Розділення даних матриці: а) стрічкова схема, б) блочна схема

Для великого класу задач обчислення зводяться до виконання однотипної обробки великого набору даних - до такого класу задач відносяться, наприклад, матричне числення, чисельні методи розв'язку рівнянь в частинних похідних та ін. В цьому випадку кажуть, що існує паралелізм за даними, і виділення підзадач зводиться до роздподілу наявних даних. Велика кількость задач роздподілу обчислень за даними приводить до

породження одно -, двох -, тривимірних наборів підзадач, в яких інформаційні зв’язки існують тільки між найближчими сусідами (такі схеми часто називаються сітками або решітками), рис. 6.3:

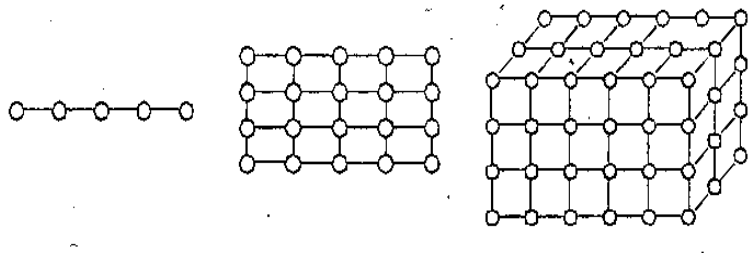


Рисунок 6.3 - Регулярні одно -, двох - та тривимірні структури базових підзадач після декомпозиції даних

Для іншої частини задач обчислення можуть полягати в виконанні різних операцій над одним і тим же набором даних - в цьому випадку кажуть про наявність функціонального паралелізму (як приклад, можна навести задачі обробки послідовності запитів до інформаційних баз даних, обчислення з одночасним застосуванням різних алгоритмів розрахунку та ін.). Часто функціональна декомпозиція використовується для організації конвеєрної обробки даних (наприклад, для виконання певних перетворень даних обчислення можна звести до функціональної послідовності введення, обробки і збереження даних). Важливим при виділенні підзадач є вибір потрібного рівня декомпозиції обчислень. Формування максимально можливої кількості підзадач забезпечує використання гранично досяжного рівня паралелізму розв’язуваної задачі, проте ускладнює аналіз паралельних обчислень. Застосування в разі декомпозиції обчислень тільки достатньо "великих" підзадач приводить до зрозумілої схеми паралельних обчислень, проте може завадити ефективному використанню достатньо великої кількості процесорів. Можливе розумне поєднання цих двох підходів може полягати в застосуванні як конструктивних елементів декомпозиції тільки тих підзадач, для яких методи паралельних обчислень відомі. Так при аналізі задачі матричного множення як підзадачі можна використовувати методи

скалярного добутку векторів або алгоритм матрично-векторного множення. Подібний проміжний спосіб декомпозиції обчислень дає змогу забезпечити простоту представлення обчислювальних схем та ефективність паралельних розрахунків. Підзадачі, що вибираються за умови такого підходу, далі іменуватимуться базовими, вони можуть бути елементарними (неподільними), якщо не припускають подальшого розділення, або складними , якщо буде подальше розділення.

Виділення інформаційних залежностей. За наявності обчислювальної схеми розв'язку задачі після виділення базових підзадач визначення інформаційних залежностей між ними не викликає ускладнень. Але етапи виділення підзадач та інформаційних залежностей достатньо складно піддаються розділенню. В разі проведення аналізу інформаційних залежностей між підзадачами слід розрізняти:

- локальні та глобальні схеми передачі даних - для локальних схем передачі даних в кожний момент часу виконується комунікація тільки між невеликою кількістю підзадач (які розташовані, як правило на сусідніх процесорах), для глобальних операцій передачі даних в процесі комунікації приймають участь всі підзадачі;

- структурні та довільні способи взаємодії - для структурних способів організації взаємодії приводить до формування певних стандартних схем комунікації (наприклад, у вигляді кільця, прямокутної решітки і т.д.), для довільних структур взаємодії схема виконання операцій передач даних не має характеру однорідності;

- статичні або динамічні схеми передачі даних - для статистичних схем моменти та учасники інформаційної взаємодії фіксуються на етапах проектування та розробки паралельних програм, для динамічного варіанту взаємодії структура операції передачі даних визначається в ході виконання обчислень;

- синхронні та асинхронні способи взаємодії - для синхронних способів операції передачі даних виконуються тільки за готовності всіх

учасників процесу взаємодії і завершуються тільки після повного закінчення всіх комунікаційних дій, за асинхронного виконання операцій учасники взаємодії можуть не очікувати повного завершення дій пов'язаних з передачею даних. Для представлення способів взаємодії достатньо складно виділити переважні форми організації передачі даних: синхронний варіант, як правило, більш простий для застосування, в той час як асинхронний часто дає змогу істотно знизити часові затримки, викликані операціями синхронної взаємодії.

Для оцінки правильності етапу виведення інформаційних залежностей можна скористатися такими контрольними питаннями:

- чи відповідає обчислювальна складність підзадач інтенсивності їх інформаційних взаємодій?

- чи є однаковою інтенсивність інформаційних взаємодій для різних підзадач?

- чи є схема інформаційної взаємодії локальною?

- чи не перешкоджає виявлена інформаційна залежність паралельному розв'язку підзадач?

Масштабування набору підзадач. Масштабування розробленої обчислювальної схеми паралельних обчислень проводиться у випадку, якщо кількість наявних підзадач відрізняється від числа запланованих до використання процесорів. Для скорочення кількості підзадач слід виконати укрупнення (агрегацію) обчислень. Застосовувані тут правила співпадають з рекомендаціями початкового етапу виділення підзадач: підзадачі, що визначаються, як і раніше, повинні мати однакову обчислювальну складність, а об'єм та інтенсивність інформаційних взаємодії між підзадачами повинні залишатися на мінімально можливому рівні. Як результат, першими претендентами на об'єднання є підзадачі з високим ступенем інформаційної взаємозалежності. За недостатньої кількості наявних підзадач для завантаження всіх доступних для використання процесорів слід виконати деталізацію (декомпозицію) обчислень. Як правило, проведення подібної

декомпозиції не викликає якихось ускладнень, якщо для базових задач методи паралельних обчислень відомі.

Виконання етапу масштабування обчислень повинно зводитися, в кінцевому результаті, до розробки правил агрегації та декомпозиції підзадач, які повинні параметрично залежать від кількості процесорів, використовуваних для обчислень.

Список контрольних питань для оцінки правильності етапу масштабування приблизно такий:

- чи не погіршиться локальність обчислень після масштабування наявного набору підзадач?

- чи мають підзадачі після масштабування однакову обчислювальну та комунікаційну складність?

- чи відповідає кількість задач кількості наявних процесорів?

- чи залежать параметричні правила масштабування від кількості процесорів?

Розподіл підзадач між процесорами. Розподіл підзадач між процесорами є завершальним етапом розробки паралельного методу. Слід відмітити, що управління розподілом навантаження для процесорів можливе тільки для обчислювальних систем з розподіленою пам’яттю, для мультипроцесорів (систем із спільною пам’яттю) розподіл навантаження, як правило, виконується операційною системою автоматично. Крім того, цей етап розподілу підзадач між процесорами є надлишковим, коли кількість підзадач співпадає з чисельністю наявних процесорів, а топологія мережі передачі даних обчислювальної системи є повним графом (тобто всі процесори пов’язані між собою прямими лініями зв’язку). Основний показник успішності виконання даного етапу - ефективність використання процесорів, яка визначається як відносний проміжок часу, протягом якого процесори використовувалися для обчислень, пов’язаних з розв’язком вихідної задачі. Шляхи досягнення позитивних результатів в цьому напрямку залишаються попередніми: як і раніше, слід забезпечити рівномірний

розподіл обчислювального навантаження між процесорами і мінімізувати кількість повідомлень, що передаються між ними. Як і в попередніх етапах проектування, оптимальне рішення проблеми розподілу підзадач між процесорами базується на аналізі інформаційної зв’язності графа "підзадачі - повідомлення". Зокрема, підзадачі, що мають інформаційні взаємодії, доцільно розміщувати на процесорах, між якими існують прямі лінії передачі даних. Вимога мінімізації інформаційних обмінів між процесорами може суперечити умові рівномірного завантаження. Можна розмістити всі підзадачі на одному процесорі і повністю усунути міжпроцесорну передачу повідомлень, проте завантаження більшості процесорів в цьому випадку буде мінімальним.

Розв’язком питань балансування обчислювального навантаження значно ускладнюється, якщо схема обчислень може змінюватися в ході розв’язування задачі. Причиною цього можуть бути неоднорідні сітки при розв’язуванні рівнянь в часткових похідних, розрідженість матриць та ін. Використовувані на етапах проектування оцінки обчислювальної складності рішення підзадач можуть мати наближений характер, а кількість підзадач може змінюватися в процесі обчислень. Тоді може знадобитися перерозподіл базових підзадач між процесорами вже безпосередньо під час виконання паралельної програми (тобто доведеться виконати динамічне балансування обчислювального навантаження).

Для прикладу охарактеризуємо спосіб динамічного управління розподілом обчислюваного навантаження, який називається схемою "менеджер-виконавець" (the manager-worker scheme), При використанні цього підходу припускається, що підзадачі можуть виникати і завершуватися в ході обчислень, при цьому інформаційні взаємодії між підзадачами або повністю відсутні, або мінімальні. У відповідності з цією схемою для управління розподілом навантаження в системі виділяється окремий процесор-менеджер, якому доступна інформація про всі наявні підзадачі. Інші процесори системи є виконавцями, які для отримання обчислювального навантаження

звертаються до процесора-менеджера. Породжувані в ході обчислень нові підзадачі передаються назад процесору-менеджеру і можуть бути отримані для розв'язку за умови наступних звертань процесорів-виконавців. Завершення обчислень відбувається в момент, коли процесори-виконавці завершили рішення всіх переданих їм підзадач, а процесор-менеджер не має яких-небудь обчислювальних навантажень для виконання. Доцільними є наступні питання для перевірки етапу розподілу підзадач:

- чи не призводить розподіл декількох задач на один процесор до зростання додаткових обчислювальних витрат?

- чи існує необхідність динамічного балансування обчислень?

- чи не є процесор-менеджер "вузьким" місцем при використанні схеми "менеджер-виконавець"?

Паралельне рішення гравітаційної задачі N тіл. Багато задач в галузі фізики приводяться до операцій обробки даних для кожної пари об'єктів наявної фізичної системи. Такою задачею є, зокрема, проблема, відома як гравітаційна задача N тіл (або просто задача N тіл).

В загальному вигляді ця задача описується наступним чином. Нехай є велика кількість тіл, для кожного з яких відома маса, початкове положення та швидкість. Під дією гравітації положення тіл змінюється, і потрібне рішення задачі полягає в моделюванні динаміки зміни системи N тіл протягом певного заданого інтервалу часу. Для проведення такого моделювання заданий інтервал часу розбивається на часові відрізки невеликої тривалості і далі на кожному кроці моделювання обчислюються сили, які діють на кожне тіло, а потім оновлюються швидкості та положення тіл. Очевидний алгоритм рішення задачі N тіл полягає в аналізуванні на кожному кроці моделювання всіх пар об'єктів фізичної системи і виконанні для кожної отримуваної пари всіх необхідних розрахунків. Як результат, за такого підходу тривалість виконання однієї ітерації моделювання складатиме

$$T_1 = \tau N(N-1)/2, \qquad (2.2.31)$$

де τ - тривалість пере обчислення параметрів однієї пари тіл.

Тобто обчислювальна схема розглянутого алгоритму є порівняно простою, що дає змогу використати задачу *N* тіл як ще одніу наочну демонстрацію застосування методики розробки паралельних алгоритмів.

Розділення обчислень на незалежні частини. Вибір способу розділення обчислень не викликає жодних ускладнень - очевидний підхід полягає у виборі як базової підзадачі всього набору обчислень, пов'язаних з обробкою даних якогось одного тіла фізичної системи.

Виділення інформаційних залежностей. Виконання обчислень, пов'язаних з кожною підзадачею, стає можливим тільки у випадку, коли в підзадачах є дані (положення та швидкості пересувань) про всі тіла фізичної системи. Як результат, перед початком кожної ітерації моделювання кожна підзадача повинна отримати всю необхідну інформацію від всіх інших підзадач системи. Така процедура передачі даних називається операцією збору даних. В досліджуваному алгоритмі ця операція повинна бути виконана для кожної підзадачі - такий варіант передачі даних називається операцією узагальненого збору даних. Визначення вимог до необхідних результатів інформаційного обміну не приводить до однозначного встановлення потрібного інформаційного обміну між підзадачами - досягнення потрібних результатів може бути забезпечене за допомогою різних алгоритмів виконання операції узагальненого збору даних Найпростіший спосіб виконання необхідного інформаційного обміну полягає в реалізації послідовності кроків, на кожному з яких всі наявні підзадачі розбиваються попарно і обмін даними здійснюється між підзадачами утворених пар. За належної організації попарного розділення підзадач *(N-1)* - кратне повторення описаних дій приведе до повної реалізації потрібної операції збору даних.

Розглянутий метод організації інформаційного обміну достатньо трудомісткий - для збору всіх необхідних даних слід провести *N-1* ітерацію, в кожній з яких виконується одночасно *N/2* операцій передачі даних. Для скорочення потрібної кількості ітерацій можна звернути увагу на факт, що

після виконання першого кроку операції збору даних підзадачі будуть вже містити не тільки свої дані, але й дані підзадач для обміну даних відразу про два тіла фізичної системи - тим самим, після завершення другої ітерації кожна підзадача міститиме відомості про чотири тіла системи і т.д. Цей спосіб реалізації обмінів дає змогу завершити необхідну процедуру за $\log_2 N$ ітерацій. За таких умов об'єм даних, що пересилаються в кожній операції обміну, подвоюється від ітерації до ітерації: В першій ітерації між підзадачами пересилаються дані про одне тіло системи, в другій - про два тіла і т.д.

Масштабування і розподіл підзадач між процесорами. Чисельність тіл системи N значно перевищує кількість процесорів p. Тому розглянуті раніше підзадачі слід укрупнити, об'єднавши в рамках однієї підзадачі обчислення для групи з N/p тіл. Після проведення такої агрегації чисельність підзадач і кількість процесорів співпадатиме і при розподілі підзадач між процесорами залишиться забезпечити наявність прямих комунікаційних ліній між процесорами з підзадачами, у яких змінюються інформаційні обміни при виконанні операції збору даних.

Аналіз ефективності паралельних обчислень. Оцінимо ефективність розроблених способів паралельних обчислень для розв'язку задачі N тіл. Оскільки запропоновані варіанти відрізняються тільки методами виконання інформаційних обмінів, для порівняння підходів достатньо визначити тривалість операції узагальненого збору даних. Використаємо для оцінки тривалості передачі повідомлень модель, запропоновану Хокні, - тоді тривалість виконання операції збору даних для першого варіанту паралельних обчислень можна виразити як:

$$T_1^p(comm) = (p-1)(\alpha + m(N/p)/\beta), \qquad (2.2.32)$$

де α, β - параметри моделі Хокні (латентність та пропускна здатність мережі передачі даних);

- m задає об’єм даних, що пересилаються, для одного тіла фізичної системи.

Для другого способу інформаційного обміну об’єм даних, що пересилаються в різних ітераціях збору даних, відрізняється. В першій ітерації об’єм повідомлень, що пересилаються, складає Nm/p, в другій ітерації цей об’єм збільшується вдвічі і становить $2Nm/p$ і т.д. В загальному випадку, для ітерації з номером i об’єм повідомлень оцінюється як $2^{i-1}Nm/p$. Як результат, тривалість виконання операції збору даних в цьому випадку визначається виразом:

$$T_p^2(comm) = \sum_{i=1}^{\log p}(\alpha + 2^{i-1}m(N/p)/\beta) = \alpha \log p + m(N/p)(p-1)/\beta \qquad (2.2.33)$$

.

Порівняння отриманих виразів показує, що другий спосіб паралельних обчислень має істотно вищу ефективність, потребує менших комунікаційних витрат і допускає кращу масштабованість за умови збільшення кількості використовуваних процесорів.

7 ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ НА ОСНОВІ MPI

В обчислювальних системах з розподіленою пам’яттю процесори функціонують незалежно один від одного. Для організації паралельних обчислень в таких умовах необхідно мати можливість розподіляти обчислювальне навантаження та організовувати інформаційну взаємодію (передачу даних) між процесорами. Рішення цих питань забезпечує інтерфейс передачі даних (message passing interface - MPI). В загальному випадку, для розподілу обчислень між процесорами необхідно: проаналізувати алгоритм розв’язку задачі,

- виділити інформаційні незалежні фрагменти обчислень;
- провести їх програмну реалізацію;
- розмістити отримані частини програми на різних процесорах.

В рамках MPI прийнятий простіший підхід - для рішення поставленої задачі розробляється одна програма, яка запускається одночасно на виконання на всіх наявних процесорах. Для уникнення ідентичності обчислень на різних процесорах, можна підставляти різні дані для програми на різних процесорах та використовувати наявні в MPI засоби для ідентифікації процесора, на якому виконується програма (тим самим надається можливість організувати в обчисленнях в залежності від використовуваного програмою процесора). Такий спосіб організації паралельних обчислень отримав назву моделі "одна програма множина процесів" (single program multiple processes or SPMP).

Для організації інформаційної взаємодії між процесорами в самому мінімальному варіанті достатньо операцій прийому і передачі даних (повинна існувати технічна можливість комунікації між процесорами - канали або лінії зв’язку). В MPI існує ціла множина операцій передачі даних, які забезпечують різні способи пересилання даних і реалізують практично всі раніше розглянуті комунікаційні операції. Саме ці можливості є найбільш

сильною стороною MPI. Спроби створення програмних засобів передачі даних між процесорами почали здійснюватись практично відразу з появою локальних комп'ютерних мереж. Проте ці засоби часто були незрозумілими і несумісними. Тобто одна з самих серйозних проблем в програмуванні - переносимість програм при переведенні програмного забезпечення на інші комп'ютерні системи. Як результат, вже з 90 - х років XX ст. стали вживатися заходи щодо стандартизації засобів організації передачі повідомлень в багатопроцесорних обчислювальних системах. Початком робіт, що призвели до появи MPI, послужило проведення робочої наради із стандартів для передачі повідомлень в середовищі розподіленої пам'яті (the Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia, USA, April 1992). За підсумками наради була утворена робоча група , пізніше перетворена в міжнародне товариство MPI Forum, результатом діяльності якого було створення і прийняття в 1994 р. стандарту інтерфейсу передачі повідомлень (message passing interface - MP) версії 1.0. В наступні роки стандарт MPI послідовно розвивався. В 1997 р. був прийнятий стандарт MPI версії 2.0.

Тепер можна пояснити значення поняття MPI. По-перше, MPI - це стандарт, якому повинні задовольняти засоби організації передачі повідомлень. По-друге - це програмні засоби, які забезпечують можливості передачі повідомлень і при цьому відповідають всі вимогам стандарту MPI. Так, за стандартом ці програмні засоби повинні бути організовані у вигляді бібліотек програмних функцій (бібліотека MPI) і повинні бути доступними для найбільш використовуваних алгоритмічних мов. Подібну "двоїстість" MPI слід враховувати при використанні термінології. Як правило, абревіатура MPI застосовується при згадуванні стандарту, а сполучення "бібліотека MPI" вказує ту чи іншу програмну реалізацію стандарту. Проте достатньо часто для скорочення позначення MPI використовується для бібліотек MPI, і, тим самим, для правильної інтерпретації терміну слід враховувати контекст.

Питання, пов’язані з розробкою паралельних програм з використанням MPI, достатньо добре розглянуто в літературі. Наведемо ряд важливих позитивних обставин:

- MPI дає змогу в значній мірі знизити гостроту проблеми перенесення паралельних програм між різними компонентами системи - паралельна програма, розроблена на алгоритмічній мові С з використанням бібліотеки MPI, як правило, працюватиме на різних обчислювальних платформах;

- MPI сприяє підвищенню ефективності паралельних обчислень, оскільки нині практично для кожного типу обчислювальних систем існують реалізації бібліотек MPI, які максимально враховують можливості комп’ютерного обладнання;

- MPI зменшує складність розробки паралельних програм, оскільки більша частина розглянутих вище основних операцій передачі даних передбачається стандартом MPI, з іншого боку, існує велика кількість бібліотек паралельних методів, створених з використанням MPI.

7.1. MPI: основні поняття та означення

Поняття паралельної програми. Під паралельною програмою в рамках MPI розуміють множину одночасно виконуваних процесів. Процеси можуть виконуватися на різних процесорах, але на одному процесорі можуть розташовуватися і декілька процесів (в цьому випадку їх виконання здійснюється в режимі розділення часу). В граничному випадку для виконання паралельної програми може використовуватися один процесор - як правило, такий спосіб застосовується для початкової перевірки правильності паралельної програми. Кожний процес програми породжується на основі копії одного і того ж програмного коду (модель SPMP). Цей програмний код, зображуваний у вигляді виконуваної програми, повинен бути доступним в момент запуску паралельної програми на всіх використовуваних процесорах. Вихідний програмний код для виконуваної

програми розроблюється на алгоритмічних мовах із застосуванням тієї чи іншої реалізації бібліотеки MPI.

Кількість процесів та чисельність використовуваних процесорів визначається в момент запуску паралельної програми засобами середовища виконання MPI - програм і в ході обчислень не може змінюватися без застосування спеціальних, але рідко задіяних засобів динамічного породження процесів та управління ними, які з'явилися в стандарті MPI версії 2.0. Всі процеси програми послідовно перенумеровані від) до $p-1$, де p є загальна кількість процесорів. Номер процесу іменується рангом процесу.

Операції передачі даних. Основу MPI складають операції передачі повідомлень. Серед передбачених в складі MPI функцій розрізняють парні (point-to-point) операції між двома процесами та колективні (collective) комунікаційні дії для одночасної взаємодії декількох процесів.

Для виконання парних операцій можна використовувати різні режими передачі, серед яких: синхронний, блокуючий та ін. До стандартів MPI включено більшість основних операцій передачі даних.

Поняття комунікаторів. Процеси паралельної програми об'єднуються в групи. Іншим важливим поняттям MPI, що описує набір процесів, є поняття комунікатора. Під комунікатором в MPI розуміють спеціально створюваний службовий об'єкт, який об'єднує в своєму складі групу процесів і ряд додаткових параметрів (контекст), використовуваних при виконанні операцій передачі даних.

Парні операції передачі даних виконуються тільки для процесів, які належать одному і тому ж комунікатору. Колективні операції застосовуються одночасно для всіх процесів одного комунікатора. Як результат, вказівка на використовуваний комунікатор є обов'язковою для операцій передачі даних в MPI. В ході обчислень можуть створюватися нові та видалятися існуючі групи процесів та комунікатори. Один і той же процес може належати різним групам і комунікаторам. Всі наявні в паралельній програмі процеси входять

до складу конструйованого за умовчанням комунікатора з ідентифікатором MPI_COMM_WORLD. У версії 2.0 стандарту з'явилася можливість створювати глобальні комунікатори (intercommunicator), які об'єднують в одну структуру пару груп за необхідності виконання колективних операцій між процесами з різних груп. Детальній розгляд можливостей MPI для роботи з групами і комунікаторами розглянемо далі.

Типи даних. При виконанні операцій передачі повідомлень для вказівки даних в функціях MPI, які передаються або отримуються, необхідно вказувати тип даних, що пересилаються. MPI містить великий набір базових типів даних, які багато в чому співпадають з типами даних в алгоритмічних мовах. Крім того, в MPI є можливості створення нових похідних типів даних для більш точного і короткого опису вмісту повідомлень, що пересилаються.

Віртуальні технології. Парні операції передачі даних можна виконати між будь-якими процесами одного і того ж комунікатора, а в колективній операції приймають участь всі процеси комунікатора. Логічна топологія ліній зв'язку між процесами має структуру повного графа (незалежно від наявності реальних фізичних каналів зв'язку між процесорами). Для подальшого викладу і аналізу ряду паралельних алгоритмів доцільним є логічне представлення наявної комунікаційної мережі у вигляді тих або інших топологій.

В MPI є можливість представлення множини процесів у вигляді решітки довільної розмірності. Граничні процеси решіток можуть бути оголошені сусідніми, і, тим самим, на основі решіток можуть бути визначені структури типу тор. Крім того, в MPI є також засоби для формування логічних (віртуальних) топологій будь-якого потрібного типу. Детальний розгляд можливостей MPI для роботи з топологіями буде здійснений далі. Перед початком розгляду MPI звернемо увагу на наступні зауваження:

- опис функцій та всі приклади програм, що наводитимуться, будуть представлені на алгоритмічній мові C;

- коротка характеристика наявних реалізацій бібліотек MPI та загальний опис середовища виконання MPI - програм будуть розглянуті далі;

- основне викладення можливостей MPI буде зорієнтовано на стандарт версії 1.2 (так званий MPI-1), нововведення стандарту версії 2.0 будуть розглянуті далі.

Починаючи знайомство з MPI, можна відмітити, що в стандарті MPI передбачається наявність більше ніж 120 функцій. З іншого боку, структура MPI є старанно продуманою - розробка паралельних програм може бути розпочата вже після розгляду 6 функцій MPI. Всі додаткові можливості MPI можна засвоїти у міру зростання складності розроблюваних алгоритмів і програм.

8 РОЗРОБКА ПАРАЛЕЛЬНИХ ПРОГРАМ З ВИКОРИСТАННЯМ MPI

Основи MPI. Наведемо мінімально необхідний набір функцій MPI, достатній для розробки порівняно простих паралельних програм.

Ініціалізація та завершення MPI - програм. Першою функцією MPI, що викликається, повинна бути функція:

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

де

- *argc* - вказівник на кількість параметрів командної стрічки,

- *argv* - параметри командної стрічки,

яка застосовується для ініціалізації середовища виконання MPI - програми. Параметрами функції є кількість аргументів в командній стрічці та адреса вказівника на масив символів тексту самого командного рядка.

Останньою функцією MPI, що викликається, обов'язково повинна бути функція:

int MPI_Finalize (void).

Як результат, можна відмітити, що структура паралельної програми, розроблена з використанням MPI, повинна мати наступний вигляд:

```
#include ''mpi. h''
int main(int  argc,  char  *argv[  ]  {
<програмний код без використання функцій MPI>
MPI_Init (&argc,  &argv);
<програмний код з використанням функцій MPI>
MPI_Finalize();
<програмний код без використання функцій MPI>
return  0;
}
```

Слід відмітити наступне:

- файл *mpi. h* містить означення іменованих констант, прототипів функцій та типів даних бібліотеки MPI;

- функції *MPI_Init* є обов'язковими і повинні бути виконані (тільки один раз) кожним процесом паралельної програми;

- перед викликом *MPI_Init* може бути виконана функція *MPI_Initialized* для означення того, чи був раніше виконаний виклик *MPI_Init*, а після виклику *MPI_Finalized* (ця функція з'явилася в стандарті MPI 2.0) аналогічного призначення.

Розглянуті приклади функцій дають представлення синтаксису іменування функцій в MPI. Імені функції передує префікс MPI, далі слідує одне чи декілька слів назви, перше слово в імені функції починається з заголовкового слова, слова розділяються знаком підкреслювання. Назви функцій MPI, як правило, пояснюють призначення виконуваних функцій дії.

Означення кількості та рангу процесів. Означення кількості процесів у виконуваній паралельній програмі з використанням функції:

int MPI_size(MPI_Comm, comm int *size),

де

- *comm* - комунікатор, розмір якого визначається,

-*size* - кількість процесів в комунікаторі, яка визначається.

Для визначення рангу процесу використовується функція:

int MPI_Com_rank(MPI_Comm comm, int *rank),

де -

- *comm* - комунікатор, в якому визначається ранг процесу,

- *rank* - ранг процесу в комунікаторі.

Розглянуті приклади функцій дають уявлення синтаксису іменування функцій в MPI. Імені функції передує префікс MPI, далі слідує одне або декілька слів назви, перше слово в назві функції починається із символу заголовку, слова розділяються знаком підкреслювання. Назви функції MPI, як правило, пояснюють призначення виконуваних функцією дій.

Визначення кількості і рангу процесів. Визначення кількості у виконуваній паралельній програмі здійснюється з використанням функції:

int MPI_Comm_size(MPI_Comm comm, int *size),

де

- *comm* - комунікатор, розмір якого визначається,

- *size* - кількість процесів в комунікаторі, що визначається.

Для визначення рангу процесу використовується функція:

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

де

- *comm* - комунікатор, в якому визначається ранг процесу,

- *rank* - ранг процесу в комунікаторі.

Як правило, виклик функцій *MPI_Comm_size* та *MPI_Comm_rank* виконується зразу після *MPI_ Init* для отримання загальної кількості процесів і рангу поточного процесу:

```
#include "mpi h"
int main(int argc, char *argv[ ] {
 int ProcNum, ProcRanc;
 <програмний код без використання функцій MPI>
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
 MPI_Comm_ranc(MPI_COMM_WORLD, &ProcRank);
     <програмний код з використанням функцій MPI>
 MPI_Finalize();
 <програмний код без використання функцій MPI>
 return 0;
}
```

Слід відмітити:

- комунікатор *MPI_COMM_WORLD*, як вже зазначалося, створюється за замовчуванням і представляє всі процеси виконуваної паралельної програми;

- ранг, отриманий з використанням функції *MPI_Comm_rank*, є рангом процесу, який виконав виклик цієї функції, тобто змінна *ProcRank* прийматиме різні значення в різних процесах.

Передача повідомлень. Для передачі повідомлень, відправник повинен виконати функцію:

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

де

- *buf* - адреса буфера пам’яті, в якому розташовані дані відправленого повідомлення;

- *count* - кількість елементів даних повідомлення;

- *type* - тип елементів даних повідомлення, що пересилається;

- *dest* - ранг процесів, якому відправляється повідомлення;

- *tag* - значення-тег, яке використовується для ідентифікації повідомлення;

- *comm* - комунікатор, в рамках якого виконується передача даних.

Для вказування типу даних що пересилаються в MPI, є ряд базових типів, повний список яких наведено в табл. 8.1.

Таблиця 8.1. Базові (визначені) типи даних MPI для алгоритмічної мови C

Тип даних MPI	**Тип даних C**
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI _NT	int
MPI_LONG	long
MPI_LONG DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Слід зауважити:

- повідомлення, що відправляється, визначається через вказівку блока пам’яті (буфера), в якому це повідомлення розміщене. Використовувана для вказівки буферу тріада:

(buf, count, type)

входить до складу практично всіх функцій передачі даних;

- процеси між якими виконується передача даних, в обов'язковому порядку повинні належати комунікатору, вказаному в функції *MPI_Send*;

- параметр *tag* використовується за необхідності розрізнювання повідомлень, що передаються, в іншому випадку в якості значення параметра можна використати довільне додатне ціле число (максимально можливе ціле число не може бути більшим ніж те, що визначається реалізацією константи *MPI_TAG_UB*) див. також опис функції *MPI_Recv*.

Відразу після завершення функції *MPI_Send* процес-відправник може почати повторно використовувати буфер пам'яті, в якому розташовувалось повідомлення, що відправлялось. Таким же чином слід розуміти, що в момент завершення функції *MPI_Send* стан самого повідомлення, що пересилається може бути різним: повідомлення може бути розташоване в процесі-відправнику, може знаходитися в процесі-одержувачі або може бути прийняте процесом одержувачем з використанням функції *MPI_Recv*. Тобто завершення функції *MPI_Send* означає лише те, що операція передачі початку виконується і пересилка повідомлення рано чи пізно буде виконана. Приклад використання функції буде представлений після опису функції *MPI_Recv*.

Прийом повідомлень. Для прийому повідомлення процес-отримувач повинен виконувати функцію:

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

де

- *buf, count, type* - буфер пам'яті для прийому повідомлення, призначення кожного окремого параметра відповідає опису в MPI_Send;

- *source* - ранг процесу, від якого повинен буде виконане прийняття повідомлення;

- *tag* - тег повідомлення, яке повинно бути прийняте для процесу;

- *comm* - комунікатор, в рамках якого виконується передача даних;

- *status* - вказівник на структуру даних з інформацією про результати виконання операції прийому даних.

Слід відмітити:

- буфер пам’яті повинен бути достатнім для прийняття повідомлення. При умові недостатності пам’яті частина повідомлення буде втрачена і під час завершення функції буде зафіксована похибка переповнення; з іншого боку, повідомлення, що приймається, може бути коротшим, порівняно з приймаючим буфером, в цьому випадку змінюються тільки ділянки буфера, які торкаються прийнятого повідомлення;

- типи елементів повідомлення, що передається та приймається, повинні співпадати;

- за необхідності прийому повідомлення від будь-якого процесу-відправника для параметра *source* може бути вказано значення *MPI_ANY_SOURCE* (на відміну від функції передачі *MPI_Send*, яка відсилає повідомлення тільки певному адресату);

- за необхідності прийому повідомлення з будь-яким тегом для параметра *tag* може бути вказано значення *MPI_ANY_TAG* (за умови використання функції *MPI_Send* повинно бути вказане конкретне значення тегу);

- на відміну від параметрів "процес-одержувач" та "тег", параметр "комунікатор" не має значення, яке означає "будь-який комунікатор";

- параметр *status* дає змогу визначити ряд характеристик прийнятого повідомлення:

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

де

- *status*- статус операції *MPI_Recv*;

- *type*- тип прийнятих даних;

- *count* - кількість елементів даних в повідомленні.

Виклик функції *MPI_Recv* не повинен узгоджуватись з часом виклику відповідної функції передачі повідомлення *MPI_Send* - прийом повідомлення може бути ініційований до моменту, на момент або після моменту початку

відправлення повідомлення. Після завершення функції *MPI_Recv* в заданому буфері пам'яті буде розміщено прийняте повідомлення. Принциповий момент в цьому випадку полягає в тому, що функція *MPI_Recv* є блокуючою для процесу одержувача, тобто його виконання призупиняється до завершення роботи функції. Якщо з якихось причин очікуване для прийому повідомлення буде відсутнє, то виконання паралельної програми буде блоковано.

Перша паралельна програма з використанням MPI. Розглянутий набір функцій виявляється достатнім для розробки паралельних програм (кількість функцій MPI, необхідних для початку розробки паралельних програм, становить 6). Програма, що наводиться нижче, є стандартним прикладом для алгоритмічної мови С.

Програма 6.1. Перша паралельна програма з використанням MPI.

```
#include <atdio. h>
#include "mpi.h"
int main(int argc, char* argv[ ]{
  int ProcRank, RecRank;
  MPI_Status Status;
  MPI_Init(&arc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcMum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  if ( ProcRanc ==0 ){
  // Дії виконувані тільки процесом з рангом 0
print("\n Hello from process %3d" , ProcRank);
        for ( int i = 1; i < ProcNum; 1++ ) {
          MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COM_WORLD, &Status);
          printf("\n Hello from process %3d" , RecvRank);
      }
  }
  else // Повідомлення, відправлене всіма процесми.
       // крім процесів з рангом 0
MPI_Send(&ProRank, 1 MPI_INT, 0, 0, MPI_COMM_W)RLD);
    MPI_Finalize();
}
```

Як випливає з тексту програми, кожний процес визначає свій ранг, після чого дії в програмі розділяються. Всі процеси, крім процесу з рангом 0, передають значення свого рангу нульовому процесу. Процес з рангом 0

спочатку друкує значення свого рангу, а далі послідовно приймає повідомлення з рангами процесів і також друкує їх значення. Слід відмітити, що порядок прийому повідомлень завчасно не визначений і залежить від умов виконання паралельної програми (цей порядок може змінюватися від запуску до запуску). Можливий варіант результатів друку процесу може полягати в наступному (для паралельної програми з чотирьох процесів):

```
Hello from process 0
Hello from process 2
Hello from process 1
Hello from process 3
```

Такий "плаваючий" тип отриманих результатів істотно ускладнює розробку, тестування та налагодження паралельних програм, оскільки в цьому випадку зникає один з основних принципів програмування - повторюваність виконуваних обчислювальних експериментів. Як правило, якщо це не приводить до втрати ефективності, слід забезпечувати однозначність розрахунків також при використанні паралельних обчислень. Для розгляднутого простого прикладу можна поновити постійність отримуваних результатів з використанням задавання рангу процесу-відправника в операції прийому повідомлення:

```
MPI_Recv(&RecvRank, 1, MPI_ANY_TAG, MPI_COMM_WORLD,
    &Status) .
```

Вказівка рангу процесу-відправника регламентує порядок прийому повідомлень та терміни друкування будуть з’являтися суворо в порядку зростання рангів процесів (таке регламентування в окремих ситуаціях) може призводити до уповільнення виконуваних паралельних обчислень). Розроблювана з використанням MPI програма, як в даному варіанті, так і в самому загальному випадку, використовується для породження всіх процесів паралельної програми і повинна визначати обчислення, виконувані всіма цими процесами. Тобто MPI - програма є певною "мікропрограмою", різні частини якої використовуються різними процесами. В наведеному прикладі програми виділені рамкою ділянки програмного коду не виконуються одночасно жодним з процесів, Перша з виділених ділянок з функцією

прийому *MPI_Recv* виконується тільки процесом з рангом 0, друга ділянка з функцією передачі *MPI_Send* задіється всіма процесами, за виключенням нульового процесу.

Для розділення фрагментів коду між процесами використовується підхід, який застосований у тільки розглянутій програмі. З використанням функції *MPI_Comm_rank* визначається ранг процесу, потім у відповідності з рангом виділяються необхідні для процесу ділянки програмного коду. Наявність в одній і тій же програмі фрагментів коду різних процесів також значно ускладнює розуміння і, в цілому розробку MPI - програми. В результаті можна рекомендувати при збільшенні об'єму розроблюваних програм виносити програмний код різних процесів в окремі програмні модулі (функції). Загальна схема MPI - програми в цьому випадку матиме вигляд:

```
MPI_Comm_rank(MPI_COMM_WORLD,  &ProcRank );
if  ( ProcRank  ==  0  ) DoProcess0( );
else  if  (  ProcRank  ==  1  ) DoProcess1)( );
else  if  (  ProcRank  ==  2  ) DoProcess2( );
```

В багатьох випадках, як і в розглянутому прикладі, виконувані дії різняться тільки для процесу з рангом 0. В цьому випадку загальна схема MPI - програми приймає простіший вигляд:

```
MPI_Comm_rank(MPI_COMM_WORLD,  &ProcRank);
if  (  ProcRank  ==  0  )  DoManagerProcess( );
else  DoWorkerProcesses( ) ;
```

На завершення обговорення прикладу потрібно пояснити застосований в MPI підхід для контролю правильності виконання функцій. Всі функції MPI (окрім *MPI_Write* та *MPI_Wtick*) повертають своє значення як код завершення. В разі успішного виконання функції код, що повертається, дорівнює *MPI_SUCCESS*. Інші значення коду завершення свідчать про виявлення тих чи інших помилкових ситуацій в ході виконання функцій. Для з'ясування типу виявленої похибки використовуються визначені іменовані константи, серед яких:

- MPI_ERR_BUFFER - невірний вказівник нв буфер;
- MPI_ERR_TRUNCATE - повідомлення перевищує розмір прийомного буфера;
- MPI_ERR_COMM - невірний комунікатор;
- MPI_ERR_RANK - невірний ранг процесора, та ін.

Повний список констант для перевірки вмісту коду завершення знаходиться в файлі *mpi.h*. Проте виникнення будь-якої похибки під час виконання функції MPI приводить до негайного завершення паралельної програми. Для того щоб мати можливість проаналізувати код завершення, що повертається, слід скористатися функціями MPI, що надаються, із обробників похибок і управління ними.

Визначення тривалості виконання MPI - програми. Відразу після розробки перших паралельних програм виникає необхідність визначення тривалості виконання обчислень для оцінки тривалості виконання обчислень, яка досягається, за рахунок використання паралелізму. Використовувані засоби для вимірювання тривалості роботи програм залежать від апаратної платформи, операційної системи, алгоритмічної мови і т.п. Стандарт MPI включає визначення спеціальних функцій для вимірювання часу, застосування яких дає змогу усунути залежність від середовища виконання паралельних програм. Отримання поточного моменту часу забезпечується з використанням функції:

```
double  MPI_Wtime(void) ,
```

результат її виклику це кількість одиниць часу, яка пройшла від певного моменту часу в минулому. Цей момент часу в минулому, від якого відбувається відлік часу, може залежати від середовища реалізації бібліотеки MPI, і для виходу від такої залежності функцію *MPI_Wtime* слід використати тільки для визначення тривалості виконання тих чи інших фрагментів коду паралельних програм. Можлива схема застосування функції *MPI_Wtime* полягає в наступному:

```
double  t1,  t2,  dt;
t1  =  MPI_Wtime( );
t2  =  MPI_Wtime( );
dt  =  t2  -  t1;
```

Точність виміру функції також може залежати від середовища виконання паралельної програми. Для визначення поточного значення точності може бути використана функція:

double MPI_Wtick(vold) ,

яка дає можливість визначити час в секундах між двома послідовними показниками часу апаратного таймера, що використовується в певній комп’ютерній системі.

9 КОЛЕКТИВНІ ОПЕРАЦІЇ ПЕРЕДАЧІ ДАНИХ

Функції *MPI_Send* та *MPI_Recv*, розглянуті раніше, забезпечують можливість виконання парних операцій передачі даних між двома процесами паралельної програмами. Для виконання комунікаційних колективних операцій, в яких задіяні всі процеси комунікатора, в MPI передбачений спеціальний набір функцій. Розглянемо три таких функції, які застосовуються при розробці порівняно простих паралельних програм. Для демонстрації застосування представлених функцій MPI використаємо початкову задачу знаходження суми елементів вектора *x*:

$$S = \sum_{i=1}^{n} x_i \ .$$

Розробка паралельного алгоритму для вирішення задачі є відносно нескладною задачею:

- потрібно розділити дані на рівні блоки;
- передати ці блоки процесам;
- виконати в процесах знаходження суми отриманих даних;
- зібрати значення обчислених часткових сум на одному з процесів;
- скласти значення окремих сум для отримання загального результату розв’язуваної задачі.

При подальшій розробці демонстраційних програм цей розглянутий алгоритм буде спрощений: процесам програми буде передаватися весь сумарний вектор, а не окремі блоки цього вектора.

Передача даних від одного процеса всім процесам системи. Перша задача при використанні розглянутого паралельного алгоритму знаходження суми полягає в необхідності передачі значень вектора *x* всім процесам паралельної програми. Для розв’язку цієї задачі можна скористатися розглянутими раніше функціями парних операцій передачі даних:

```
MPI_Comm_size(MPI_COMM_WORLD,  &ProcNum) ;
for (int i= 1; i < ProcNum;  1++)
```

```
MPI_Send(&x, n, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
```

Проте таке рішення буде вкрай не ефективним, оскільки повторення операцій передачі приводить до знаходження суми витрат (латентностей) на підготовку повідомлень, що передаються. Цю операцію можна виконати за $\log_2 p$ ітерацій передачі даних. Досягнення ефективного виконання операції передачі даних від одного процесу всім процесам програми може бути забезпечене з використанням функції MPI:

```
int MPI_Bcast(void *buf , int count, MPI_Datftype type, int root,
MPI_Comm comm),
```

де

- *buf, count, type* - буфер пам’яті з повідомленням, що відпраляється (для процесу з рангом 0) та для прийому повідомлень (для всіх інших процесів);

- *root* - ранг процесу, який виконує розсилання даних;

- *comm* - комунікатор, в рамках якого виконується передача даних.

Функція *MPI_Bcast* здійснює розсилку даних з буфера *buf*, який містить *count* елементів типу *type*, з процесу, який має номер *root*, всім процесам, які входять до комунікатора *comm*, рис. 9.1.

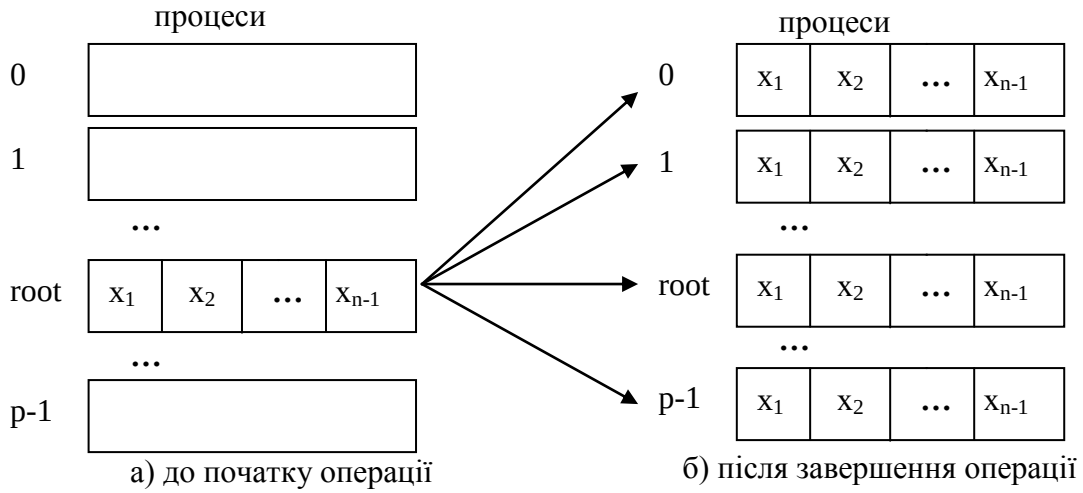


Рисунок 9.1 - Загальна схема операції передачі даних від одного процесу всім процесам

Слід відмітити:

- функція *MPI_Bcast* визначає колективну операцію, і при виконанні необхідних розсилок даних виклик функції *MPI_Bcast* повинен бути здійснений всіма процесами вказаного комунікатора;

- вказаний в функції *MPI_Bcast* буфер пам'яті має різне призначення в різних процесах: для процесу з рангом *root*, яким здійснюється розсилання даних, в цьому буфері повинно знаходитися повідомлення, що розсилається, а для всіх інших процесів вказаний буфер призначений для прийому даних, які передаються;

- всі колективні операції "несумісні" з парними операціям, якщо широкооголошуване повідомлення, відіслане з використанням *MPI_Bcast*, функцією *MPI_Recv* прийняти неможна, то для цього можна задіяти тільки *MPI_Bcast*.

Наведемо програму для розв'язку задачі знаходження суми елементів вектора з використанням розглянутої функції.

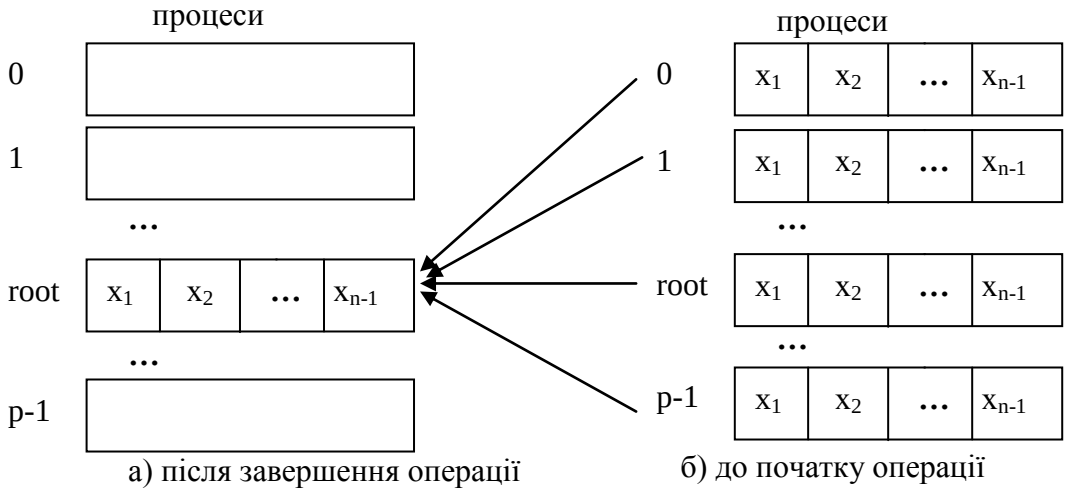
Програма 9.1. Паралельна програма знаходження суми числових значень

```
#include  <math. h>
#include  <stdio. h>
#include  <mp. i>
int  main(int  argc,  char*  argv[ ]){
MPI_Status  Status;
//  Ініціалізація
MPI_Init(&argc ,  &argv);
MPI_Comm_sizt(MPI_COMM_WORLD,  &ProcRum);
MPI_Comm_rank(MPI_COMM_WORLD.  &ProcRanc);
//  Підготовка  даних
 (  ProcRank  ==  0  )  DataInitialization(x, N);
//  Підготовка даних
if  {  ProcRank  ==  0  )  DataInitialization(z, H);
//  Розсилання даних на всі процеси
//MPI_Bcast(x,  N,  MPI_DOUBLE,  0,  MPI_Comm_WORLD);
//  Обчислення часткової суми на кожному з процесів
// на кожному процесі підсумовуються  елементи вектора хвід 11 до 12
k = N  /  ProcNum;
i 1  = k * ProcRank;
i 2 = k * (  ProcRank  +  1);
if  (  ProcRank  ==  ProcNum-1)  i2 =N;
for  (  int  i  =  11;  i  <  12;  i++  )
```

```
            ProcSum = ProcSum + x[i];
        // Збірка часткових сум на процесі з рангом 0
        if ( ProcRank == 0 ) {
           TotalSum = ProcSum;
           for ( int i=1; i < ProcSum; i++ ) {
             MPI_Recv(&ProcSum, 1, MPI_DOUBLE,MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD, &Status);
             TotalSum = TotalSum + ProcSum;
           }
        }
        else // Всі процеси відсилають свої часткові суми
         MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            // Виведення результату
           if ( ProcRank == 0 )
                printf("\nTotal Sum = %10.2f", TotflSum) ;
            MPI_Finalize( ) ;
          return 0;
        }
```

В наведеній програмі функція *DataInitialization* здійснює підготовку початкових даних. Необхідні дані можуть бути введені з клавіатури, прочитані з файлу або згенеровані з використанням датчика випадкових чисел.

Передача даних від всіх процесів одному процесу. Операція редукції. В розглянутій програмі підрахунок числових значень наявна процедура збору і наступного підсумовування даних є прикладом часто виконуваної колективної операції передачі даних від всіх процесів одному процесу рис. 9.2:



а) після завершення операції б) до початку операції

Рисунок 9.2 - Загальна схема операції збору і обробки на одному процесі даних від всіх процесів

Як і раніше, реалізація операції редукції з використанням звичайних парних операцій передачі даних є неефективною і достатньо трудомісткою. Для найкращого виконання дій, пов'язаних з редукцією даних, в MPI передбачена функція:

```
int MPI_Reduce(void *recvbuf, int count,
  MPI_Datatype type, MPI_Op op, int root, MPI-Comm comm,
```

де

- *sendbuf* - буфер пам'яті з повідомленням, яке відправляється;

- *recvbuf* - буфер пам'яті для результуючого повідомлення (тільки для процесу з рангом *boot)*

- *count* - якість елементів в повідомленнях;

- *type* - тип елементів повідомлень;

- *op* - операція, яка повинна бути виконана над даними;

- *root* - ранг процесу. на якому має бути результат;

- *comm* - комунікатор, в рамках якого виконується операція.

Як операції редукції даних можуть бути використані обумовлені в MPI операції, табл. 9.1.

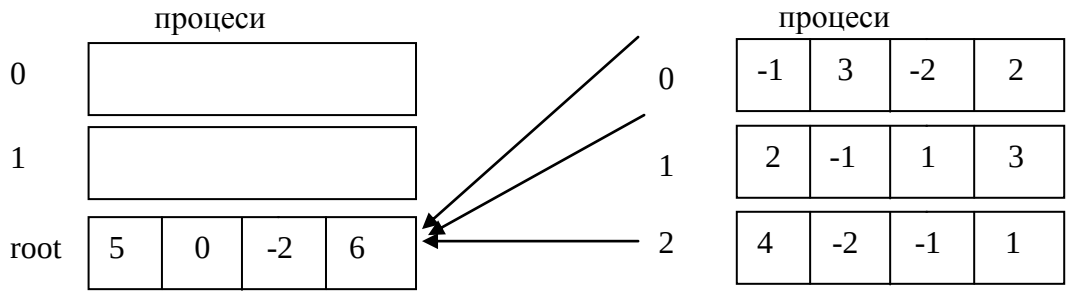
Таблиця 9.1. Базові (зумовлені) типи операцій MPI для функцій редукції даних

Операція	**Опис**
MPI_MAX	Означення максимального значення
MPI_MIN	Означення мінімального значення
MPI_SUM	Означення суми значень
MPI_PROD	Означення добутку
MPI_LAND	Виконання логічної операції "І" над значеннями повідомлень
MPI_BAND	Виконання бітової операції "І" над значеннями повідомлень
MPI_LOR	Виконання логічної операції "АБО" над значеннями повідомлень
MPI_BOR	Виконання бітової операції "АБО" над значеннями повідомлень
MPI_LXOR	Виконання логічної операції виключного "АБО" над значеннями повідомлень
MPI_BXOR	Виконання бітової операції виключного "АБО" над значеннями повідомлень
MPI_MAXLOC	Визначення максимальних значень та їх індексів
MPI_MINLOC	Визначення мінімальних значень та їх індексів

Окрім даного стандартного набору операцій можуть бути визначені і нові додаткові операції безпосередньо самим користувачем бібліотеки MPI. Елементи отримуваного повідомлення в процесі *root* являють собою результати обробки відповідних елементів, що передаються процесами повідомлень, тобто:

$$y_j = \otimes_{i=0}^{n-1} x_{ij},\ 0 \le j \le n,$$

де $\otimes$ - операція, яка задається під час виклику функції *MPI_Reduce* (для пояснення на рис.9.3 показаний приклад виконання функції редукції даних).



а) після завершення операції б) до початку операції

Рисунок 9.3 - Процес виконання операції редукції при підсумовуванні даних, які пересилаються, для трьох процесів (в кожному повідомленні 4 елементи, повідомлення збираються на процесі з рангом 2)

Слід відмітити:

- функція *MPI_Reduce* визначає колективну операцію, тим самим виклик функції повинен бути виконаний всіма процесами вказаного комунікатора. Всі виклики функції повинні містити однакові значення параметрів *count*, *type*, *op*, *root*, *comm*;

- виконання операції редукції здійснюється над окремими елементами повідомлень, що передаються. Якщо повідомлення містить по два елементи даних і виконується операція підрахунку *MPI_SUM*, то результат також буде складатися з двох значень, перше з яких міститиме суму перших елементів всіх відправлених повідомлень, а друге значення дорівнюватиме сумі других елементів повідомлень, відповідно;

- не всі сполучення типу *type* та операції *op* можливі. Дозволені сполучення наведені в табл.9.2.

Таблиця 9.2 Дозволені сполучення операції типу операнда в операції редукції

Операція	**Допустимий тип операндів для алгоритмічної мови С**
MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD	Цілий, дійсний
MPI_LAND, MPI_LOR, MPI_LXOR	Цілий
MPI_BAND, MPI_BOR, MPI_BXOR	Цілий, байтовий
MPI_MINLOC, MPI_MAXLOC	Цілий, дійсний

Застосуємо отриману інформацію для переробки раніше розглянутої програми підсрахунку: весь програмний код, виділений під рамкою, може бути замінений на виклик однієї лише функції MPI_Reduces:

```
// Збірка часткових сум на процесі з рангом 0
MPI_Reduce(&ProcSum, &TotalSum, 1, MPI_DOUBLE, MPI_SUM, 0,
  MPI_COMM_WORLD);
```

Синхронізація обчислень. В багатьох випадках незалежно виконувані обчислення в процесах необхідно синхронізувати. Зокрема для виміру часу початку роботи паралельної програми необхідно, щоб для всіх процесів одночасно були завершені всі підготовчі дії, перед закінченням роботи програми всі процеси повинні завершити свої обчислення і т.п. Синхронізація процесів, тобто одночасне досягнення процесами тих чи інших точок процесу обчислень, забезпечується за допомогою функції MPI:

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

де

- *comm* - комунікатор, в рамках якого виконується операція.

Функція *MPI_Barrier* визначає колективну операцію, і, тим самим, при використанні вона повинна викликатися всіма процесами використовуваного комунікатора. При виклику функції *MPI_Barrier* виконання процесу блокується, продовження обчислень процесу відбудеться тільки після виклику функції *MPI_Barrier* всіма процесами комунікатора.

Аварійне завершення паралельної програми. Для коректного завершення паралельної програми у випадку непередбачених ситуацій слід використати функцію:

```
int MPI_Abort(MPI_Comm comm, int errorcode ) ,
```

де

- *comm* - комунікатор, процеси якого необхідно аварійно зупинити;

- *errorcode* - код повернення та паралельної програми.

Ця функція коректно перериває виконання паралельної програми, сповіщаючи про цю подію середовище MPI, на відміну від функцій бібліотеки алгоритмічної мови С, таких, як *abort* чи *terminate*. Звичайне її використання полягає в наступному:

```
MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
```

Операції передачі даних між двома процесами. Режими передачі даних. Розглянута раніше функція *MPI_Send* забезпечує так званий стандартний режим відправки повідомлень, коли:

- на час виконання функції процес-відправник повідомлення блокується;

- після завершення функції буфер може бути використаний повторно;

- стан відправленого повідомлення може бути різним - повідомлення може розташовуватися на процесі-відправнику, може знаходитися в стані передачі, може зберігатися в процесі-одержувачі або може бути прийняте процесом-одержувачем з використанням функції *MPI_Recv*.

Крім стандартного режиму в MPI передбачаються також додаткові режими передачі повідомлень:

- синхронний режим полягає в тому, що завершення функції відправлення повідомлення відбувається тільки після отримання від процесу - одержувача підтвердження про початок прийому відправленого

повідомлення. Відправлене повідомлення або повністю прийняте процесом-одержувачем, або знаходиться в стані прийому;

- буферизований режим передбачає використання додаткових системних або заданих користувачем буферів для копіювання в них відправлених повідомлень. Функція відправки повідомлення завершується зразу ж після копіювання повідомлення до системного буферу;

- режим передачі за готовністю може бути використаний тільки, якщо операція прийому повідомлення вже ініційована. Буфер повідомлення після завершення функції відправки повідомлення може бути повторно використаний.

Для іменування функції відправки для різних режимів виконання в MPI застосовується назва функції *MPI_Send*, до якого як префікс додається початковий символ назви відповідного режиму роботи, тобто:

-*MPI_Ssent* - функція відправки повідомлення в синхронному режимі;

- *MPI_Bsent* - функція відправки повідомлення в буферному режимі;

- *MPI_Rsent* - функція відправки повідомлення в режимі за готовністю.

Список параметрів всіх перерахованих функцій співпадає із складом параметрів функції *MPI_Send*. Для застосування буферного режиму передачі може бути створений і переданий MPI буфер пам’яті, а використовувана для цього функція має вигляд:

```
int MPI_Buffer_attach(void *buf size),
```

де

- *buf* - адреса буфера пам’яті;

- *size* - розмір буфера.

Після завершення роботи з буфером він повинен бути відключеним від MPI з використанням функції:

```
int MPI_Buffer_detach(void *buf , int *size),
```

де

- *buf* - адреса буфера пам’яті;

- *size* - розмір буфера, що повертається.

З точки зору практичного використання режимів можна навести такі рекомендації:

- стандартний режим реалізується як буферизований або синхронний, в залежності від розміру повідомлення, що передається, такий режим часто є найбільш оптимізованим за продуктивністю;

- режим передачі за готовністю формально є найбільш швидким, але використовується достатньо рідко, оскільки складно гарантувати готовність операції прийому;

- буферизований режим також виконується достатньо швидко, але може приводити до великих витрат ресурсів пам'яті - в цілому може бути рекомендованим для передачі коротких повідомлень;

- синхронний режим є найбільш повільним, оскільки вимагає підтвердження прийому, проте не потребує додаткової пам'яті для зберігання повідомлення. Цей режим можна рекомендувати для передачі довгих повідомлень.

Для функції *MPI_Recv* не існує різноманітних режимів роботи.

Організація неблокуючих обмінів даних між процесами. Всі раніше розглянуті функції відправки і прийому повідомлень - блокуючі, тобто є такими, що призупиняють виконання процесів до моменту завершення роботи викликаних функцій. В той же час при виконанні паралельних обчислень частина повідомлень може бути відправлена і прийнята завчасно, до моменту реальної потреби в даних, що пересилаються. В таких ситуаціях небажано мати можливість виконання функцій обміну даними без блокування процесів для суміщення процесів передачі повідомлень та обчислень. Такий неблокуючий спосіб виконання обмінів є складнішим для використання, але за правильного застосування може в значній мірі зменшити втрати ефективності паралельних обчислень внаслідок повільних (порівняно з швидкодією процесорів) комунікаційних операцій. MPI забезпечує можливість неблокованого виконання операцій передачі даних

між двома процесами. Найменування неблокуючих аналогів утворюється з назв відповідних функцій шляхом додавання префікса *I* (Immediate). Список параметрів неблокуючих функцій містить звичайний набір параметрів вихідних функцій і один додатковий параметр *request* з типом *MPI_Request* (в функції *MPI_Irecv* відсутній також параметр *status*):

```
int MPI_Isend(void *buf. int count, MPI_batatype type, int dest
  int tag, MPI_Comm comm,  MPI_Request  *request),
int MPI_Inssent(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm, MPI_Request *request),
int MPI_Ibsennd(void *buf , int count , MPI_Dftftipe tupe, int dest,
  int tag, MPI_Comm comm, MPI_Request *request),
Int MPI_Irsend(void *buf , int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm, MPI_Request *request),
Int MPI_Irsend(void *buf , int count, MPI_Datatype type, int sourse,
  int tag, MPI_Comm comm, MPI_Request *request).
```

Виклик неблокуючої функції приводить до ініціації запрошеної операції передачі, після чого виконання функції завершується і процес може продовжувати свої дії. Перед завершенням неблокуюча функція визначає змінну *request*, яка далі може використовуватися для перевірки завершення ініційованої операції обміну. Перевірка стану виконуваної неблокуючої операції передачі даних здійснюється з використанням функції:

```
int MPI_Test(MPI_Request *request, int *flaq , MPI_status *status) ,
```

де

- *request* - дескриптор операції, визначений під час виклику неблокуючої функції;

- *flaq* - результат перевірки (*true*, якщо операція завершена);

- *status* - результат виконання операції обміну (тільки для завершеної операції).

Операція перевірки є неблокуючою, тобто процес може перевірити стан неблокуючої операції обміну і продовжити далі свої обчислення, якщо за результатами перевірки виявиться,що операція все ще не завершена. Можлива схема суміщення обчислень і виконання неблокуючої операції обміну може полягати в наступному:

```
MPI_Isend(buf, count, type, dest, taq, comm, &request );
```

```
...
do  {
...
  MPI_Test(&request ,  &flaq,  &status ) ;
}  while  (! flaq) ;
```

Якщо за умови виконання неблокуючої операції виявиться, що продовження обчислень неможливе без отримання даних, що передаються, то може бути використана блокуюча операція очікування завершення операції:

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

де

- *request* - дескриптор операції, визначений при виклику неблокуючої функції;
- *status* - результат виконання операції обміну (тільки для завершеної операції).

Окрім розглянутих, MPI містить ряд додаткових функцій перевірки і очікування неблокуючих операцій обміну:

- *MPI_SNSTALL* - перевірка завершення всіх перерахованих операцій обміну;
- *MPI_Waitall* - очікування завершення всіх операцій обміну;
- *MPI_Testany* - перевірка завершення хоча б однієї з перерахованих операцій обміну;
- *MPI_Waitany* - перевірка завершення будь-якої з перерахованих операцій обміну;
- *MPI_Testsome* - перевірка завершення кожної з перерахованих операцій обміну;
- *MPI_Waitsome* - очікування завершення хоч однієї з перерахованих операцій обміну та оцінка стану за всіма операціями.

Наведення простого прикладу використання неблокуючих функцій достатньо складно. Хорошою можливістю засвоєння розглянутих функцій можуть бути алгоритми матричного множення, які будуть розглянуті далі.

Одночасне виконання передачі і прийому. Однією з часто виконуваних форм інформаційної взаємодії в паралельних програмах є обмін даними між процесами, коли для продовження обчислень процесам необхідно відправити дані одним процесам і в той же час отримати повідомлення від інших. Найпростіший варіант цієї ситуації полягає в обміні даними між двома процесами. Реалізація таких обмінів з використанням звичайних парних операцій передачі даних може бути неефективна, крім того, така реалізація повинна гарантувати відсутність тупикових ситуацій, які можуть виникати, наприклад, коли два процеси починають передавати повідомлення один одному з використанням блокуючих функцій передачі даних. Досягнення ефективного і гарантованого одночасного виконання операцій передачі і прийому даних може бути забезпечене з використанням функції MPI:

```
int MPI_Sendrecv(void *sbuf , int scount , MPI_Datatype stype .
  int dest, int stag, void *rbuf , int rcount , MPI_Datatype etype,
  int source , int rtag, MPI_Comm comm, MPI_Status *status )
```

де

- *sbuf, scount, stype, dest, stag* - параметри повідомлення, що передається;

- *rbuf, rcount, rtype, source, rtag* - параметри повідомлення, що приймається;

- *comm* - комунікатор, в рамках якого виконується передача даних;

- *status* - структура даних з інформацією про результат виконання операції.

Функція *MPI_Sendrecv* передає повідомлення, яке описується параметрами (*sbuf, scount, stype, dest, stag*), процесу з рангом *dest* і приймає повідомлення в буфер, який визначається параметрами (*rbuf, rcount, rtype, source, rtag*), від процесу з рангом *source*. В функції *MPI_Sendrecv* для передачі і прийому повідомлень застосовуються різні буфери. У випадку, коли повідомлення, що відсилається більше не потрібне в процесі - відправнику, в MPI є можливість використання буфера:

```
int MPI_Sendrecv_replace(void *buf , int count , MPI_Datatyp type ,
```

```
int dest , int stag , int source , int rtaq , MPI_Comm comm ,
MPI_Status* status ) ,
```

де

- *buf, count, type* - параметри повідомлення, що передається;

- *dest* - ранг процесу, якому надсилається повідомлення;

- *stag* - тег для ідентифікації повідомлення, що надсилається;

- *source* - ранг процесу, від якого виконується приймаеея повідомлення;

- *rtag* - тег для ідентифікації повідомлення, що приймається;

- *comm* - комунікатор, в рамках якого виконується передавання даних;

- *status* - структура даних з інформацією про результат виконання.

Приклад використання функції для одночасного виконання операцій передачі і прийому наведений далі.

Колективні операції передачі даних. Під колективними операціями в МЗШ розуміють операції з даними, в яких приймають участь всі процеси використовуваного комунікатора.

Узагальнена передача даних від одного процеса всім процесам. Узагальнена операція передачі даних від одного процеса всім процесам (розподіл даних) відрізняється від широкомовної розсилки тим, що процес передає процесам дані, що розрізняються. Виконання даної операції забезпечується використанням функції:

```
int MPI_Scatter(void *sbuf , int scount , MPI_Datatype stype ,
  voide *rbuf , int rcount , MPI_Datatype rtype, int root ,
  MPI_Comm comm ) ,
```

де

- *sbuf, scount, stype* - параметри повідомлення, що передаються, (*scount* визначає кількість елементів, що передаються на кожний процес);

- *rbuf, rcount, rtype* - параметри повідомлення, яке приймається в процесах;

- *root* - ранг процесу, який виконує розсилання даних;

- *comm* - комунікатор, в рамках якого виконується передача даних.

В разі виклику цієї функції процес з рангом *root* здійснить передачу даних всім іншим процесам в комунікаторі. Кожному процесу буде відправлено *scount* елементів. Процес з рангом 0 отримає блок даних із *sbuf* елементів з індексами від 0 до *scount* - 1, процесу з рангом 1 буде відправлений блок з *sbuf* елементів з індексами від *scount* до *2* scount* 1 і т.д. Тим самим, спільний розмір повідомлення, що відправляється, повинен бути рівний *scount * p* елементів, де *p* є кількістю процесів в комунікаторі *comm*. Оскільки функція *MPI_Scatter* визначає колективну операцію, виклик цієї функції при виконанні розсилки даних може бути забезпечений в кожному процесі комунікатора. Функція *MPI_Scatter* передає всім процесам повідомлення однакового розміру. Виконання більш загального варіанту операції розподілу даних, коли розміри повідомлень для процесів можуть бути різні, забезпечується з використанням функції *MPI_Scatterv*.

Узагальнена передача даних від всіх процесів одному процесу. Для виконання цієї операції в MPI призначена функція:

```
int MPI_Gather(void *sbuf , int scount , MPI_Datatype stype ,
  void *rbuf , int rcount , MPI_Datatype rtype ,
  int root , MPI_Comm comm ) ,
```

де

- *sbuf, scount, stype* - параметри повідомлення, що передається;

- *rbuf, rcount, rtype* - параметри повідомлення, що приймається;

- *root* - ранг процесу, що виконує збирання даних;

- *comm* - комунікатор, в рамках якого виконується передача даних.

При виконанні функції *MPI_Gather* кожний процес в комунікаторі передає дані з буферу *sbuf* на процес з рангом *root*. Процес з рангом *root* збирає всі отримувані дані в буфері *rbuf* (розміщення даних в буфері здійснюється у відповідності з рангами процесів - відправників повідомлень). Для того, щоб розмістити всі дані, що надходять, розмір буфера *rbuf* повинен

бути рівним *scount * p* елементів, де *p* - кількість процесів в комунікаторі *comm*. Функція *MPI_Gather* також визначає колективну операцію, і її виклик при виконанні збору даних повинен бути забезпечений в кожному процесі комунікатора. при використанні функції *MPI_Gather* збірка даних здійснюється тільки в одному процесі. Для отримання всіх даних, що збираються, на кожному з процесів комунікатора слід застосовувати функцію збору і розсилання:

```
int  MPI_Allqather(void  *sbuf ,  int  scount ,  MPI_Datatype  stype ,
  void  *rbuf ,  int  rcount , MPI_Datatype  rtype ,  MPI_Comm  comm ) ,
```

де

- *sbuf, scount, stype* - параметри повідомлення, що передається;

- *rbuf, rcount, rtype* - параметри повідомлення, що приймається;

comm - комунікатор, в рамках якого виконується передача даних.

Виконання загального варіанту операції збору даних, коли розміри повідомлень, що передаються процесами, можуть бути різними, забезпечується з використанням функцій *MPI_Gather* та *MPI_Allgatherv*.

Загальна передача даних від всіх процесів всім процесам. Передача даних від всіх процесів всім процесам є найбільш загальною операцією передачі даних.

Виконання цієї операції забезпечується з використанням функції:

```
int  MPI_Alltoall(void  *sbuf , int  scount , MPI_Datatype  stype ,
vjid  *rbuf , int  rcount , MPI_Datatype  rtype , MPI_Comm  comm ),
```

де

- *sbuf, scount, stype* - параметри повідомлень, що передаються;

- *rbuf, rcount, rtype* - параметри повідомлень, що приймаються;

- *comm* - комунікатор, в рамках якого виконується передача даних.

При виконанні функції *MPI_Alltoall* кожний процес в комунікаторі передає дані з *scount* елементів кожному процесу (спільний розмір повідомлень, що надсилаються, в процесах повинен дорівнювати *scount * p* елементів, де *p* - кількістю процесів в комунікаторі *comm*) і приймає

повідомлення від кожного процесу. Виклик функції *MPI_Alltoall* при виконанні операції загального обміну даними повинен бути виконаний в кожному процесі комунікатора. Варіант операції загального обміну даними, коли розміри повідомлень, що передаються процесами, можуть бути різними, забезпечується з допомогою використанням функції *MPI_Alltoallv*.

Додаткові операції редукції даних. Розглянута вище функція *MPI_Reduce* забезпечує отримання результатів редукції даних тільки на одному процесі. Для отримання результатів редукції даних на кожному з процесів комунікатора слід використати функцію редукції і розсилки:

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

де

- *sendbuf* - буфер пам’яті з повідомленням, що відправляється;
- *recvbuf* - буфер пам’яті для результуючого повідомлення;
- *count* - кількість елементів в повідомленні;
- *type* - тип елементів повідомлень;
- *op* - операція, яка повинна бути виконана над даними;
- *comm* - комунікатор, в рамках якого виконується операція) ,

Функція *MPI_Allreduce* виконує розсилання між процесами всіх результатів операції редукції. Можливість управління розподілом цих даних між процесами надається функцією *MPI_Allreduce_scatter*. Ще один варіант операції збору і обробки даних, коли забезпечується отримання всіх результатів редукування, може бути реалізований з використанням функції:

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

де

- *sendbuf* - буфер пам’яті з повідомлення, що відправляється;
- *recvbuf* - буфер пам’яті для результуючого повідомлення;
- *count* - кількість елементів в повідомленні;
- *type* - тип елементів повідомлення;
- *op* - операція, яка повинна бути виконана над даними;

- *comm* - комунікатор, в рамках якого виконується операція.

Елементи отриманих повідомлень яввляють собою результати обробки відповідних елементів повідомлень, що передаються процесами. Для отримання результатів на процесі з рангом $i, 0 \le i \le n$ використовуються дані від процесів, ранг яких менший чи рівний i, тобто

$$y_{ij} = \otimes_{k=0}^{i} x_{kj}, 0 \le i, j \le n,$$

де $\otimes$ - операція, що задається при виклику функції *MPI_Scan*.

Поняття похідного типу даних. В самому загальному випадку під похідним типом даних в MPI розуміють опис набору значень передбаченого в MPI типу, причому в загальному випадку описувані значення не обов'язково неперервним чином розташовуються в пам'яті. Задавання типу в MPI прийнято здійснювати з використанням карти типу (*type map*) у вигляді послідовності описів, тобто

TypeMap$=\{(type_0, disp_0),...,(type_{n-1}, disp_{n-1})\}$.

Частина карти типу з вказівкою тільки типів значень іменується в MPI сигнатурою типу:

TypeSignature $=\{type_0,...,type_{n-1}\}$.

Сигнатура типу описує, які базові типи даних утворюють певний довільний тип даних MPI, і, тим самим, керує інтерпретацією елементів даних при передачі або отриманні повідомлень. Зміщення карти типу визначають, де знаходять значення даних. Пояснимо розглянуте поняття на прикладі. Нехай до повідомлення повинні входити значення змінних:

```
double a; /* адреса 24 */
double b; /* адреса 40 */
int    n; /* адреса 48 */
```

Тоді похідний тип для опису таких даних повинен мати карту типу такого вигляду:

```
{(MPI_DOUBLE, 0),
(MPI_DOUBLE, 16),
(MPI_INT ,    24)}
```

Додатково для похідних типів даних в MPI використовується наступний ряд нових понять:

* нижня межа типу

$lb(\mathbf{TypeMap}) = \min j(disp_j)$;

* верхня межа типу

$ub(\mathbf{TypeMap}) = \max j(disp_j + sizeo_f(type_j)) + \Delta$;

* протяжність типу

$extent(\mathbf{TypeMap}) = ub(\mathbf{TypeMap}) - lb(\mathbf{TypeMap})$.

Згідно з означенням, нижня межа є суміщення для першого байта значень розглядуваного типу даних. Відповідно, верхня межа є зміщення для байта, який розташовується слідом за останнім елементом розглядуваного типу даних. Величина зміщення для верхньої межі може бути округлена вгору з врахуванням вимог вирівнювання адрес. Одна з вимог, які накладають деякі реалізації мови С, полягає в тому, щоб адреса елемента була кратною довжині цього елемента в байтах. Наприклад, якщо тип i*nt* займає чотири байти, то адреса елемента типу *int* повинна націло ділитися на чотири. Саме ця вимога відображається в означення верхньої межі типу даних MPI. Пояснимо це на вже розглянутому прикладі набору змінних *a, b, n,* для якого нижня межа дорівнює 0, а верхня приймає значення 32 (величина округлення 6 або 4 в залежності від розміру типу *int*). Тут слід відмітити, що потрібне вирівнювання визначається за типом першого елемента даних в карті типу. Слід наголосити на різниці понять "протяжність" та "розмір типу". Протяжність - це розмір пам’яті в байтах, який слід відвести для одного елемента довільного типу. Розмір типу даних - це кількість байтів, які займають дані (різниця між адресами останнього і першого байтів даних). Тут різниця в значеннях протяжності і розміру в величині округлення для вирівнювання адрес. В прикладі розмір типу дорівнює 28, а протяжність - 32 (припускається, що тип *int* займає чотири байти).

Для отримання значення протяжності типу в MPI передбачена функція:

```
int  VHS_Type_extent(MPI_Datatype  type ,  MPI_Aint  *extent ) ,
```

де

- *type* - тип даних, протяжність якого знаходиться;
- *extent* - протяжність типу.

Розмір типу можна знайти, використовуючи функцію:

int MPI_Type_Size(MPI_Danatype type , MPI_Aint *size) ,

де

- *type* - тип даних, розмір якого знаходиться;

- *size* - розмір типу.

Визначення нижньої і верхньої меж типу може бути виконане з використанням функцій:

int MPI_Type_lb(MPI_Datatype type , MPI_Aint *disp) та
int MPI_Type_ub(MPI_Datatype type , MPI_Aint *disp) ,

де

- *type* - тип даних, нижня межа якого знаходиться;

- *disp* - нижня/верхня межа типу.

Важливою і необхідною при конструюванні довільних типів є функція отримання адреси змінної:

int MPI_Address(void *location , MPI_Aint *address) ,

де

- *location* - адреса пам'яті;

- *address* - адреса пам'яті в MPI - форматі, який є переносним

(слід відмітити, що ця функція є переносним варіантом засобів отримання адрес в алгоритмічних мовах таких як C).

Способи конструювання довільних типів даних. Для зниження складності в MPI передбачено декілька різних способів конструювання довільних типів:

- неперервний спосіб дає змогу визначити неперервний набір елементів наявного типу як новий довільний тип;

- векторний спосіб забезпечує створення нового довільного типу як набору елементів наявного типу, між елементами якого є регулярні проміжки. Розмір проміжків задається в кількості елементів вихідного типу, в той час як у варіанті *H* - векторного способу цей розмір вказується в байтах;

- індексний спосіб відрізняється від векторного методу тим, що проміжки між елементами вихідного типу можуть мати нерегулярний

характер (є *H* - індексний спосіб, який відрізняється способом задавання проміжків);

- структурний спосіб забезпечує самий загальний опис довільного типу через явну вказівку карти створюваного типу даних.

Безперервний спосіб конструювання. Для безперервного способу конструювання довільного типу даних в MPI використовуються функція:

```
int MPI_Type_contigous(int count , MPI_Data_type oldtype ,
   MPI_Datatype *newtype ),
```

де

- *count* - кількість елементів вихідного типу;

- *oldtype* - вихідний тип даних;

- *newtype* - новий тип даних, що визначається.

Як випливає з опису, новий тип *newtype* створюється, як *count* елементів вихідного типу *oldtype*.

Наприклад, якщо вихідний тип даних має карту типу

{(MPI_INT , 0) , (MPI_DOUBLE , 8)} ,

то виклик функції *MPI_Type_contiquos* з параметрами

```
MPI_Type_contiquous(2 , oldtype , &newtype ) ;
```

призведе до створення типу даних з картою типу

{(MPI_INT , 0) , (MPI_DOUBLE, 8) , (MPI_INT , 16) , (MPI_DOUBLE , 24)}.

В певному розумінні наявність безперервного способу конструювання є надлишковим, оскільки використання аргументу *count* в процедурах MPI рівнозначне використанню безперервного типу даних такого ж розміру.

Векторний спосіб конструювання. За умови векторного способу довільного типу в MPI застосовуються функції

```
int MPI_Type_vector(int count , snt blocklen , int stride ,
   MPI_Data_type oldtype , MPI_Datatype *newtype ) та
int MPI_Type_hvector(int count , int blocken , MPI_Aint stride ,
   MPI_Data_type oldtype , MPI_Datatype *newtype ) ,
```

де

- *count* - кількість блоків;

- *blocklen* - розмір кожного блока;

- *stride* - кількість елементів, розташованих між двома сусідніми блоками;

- *oldtype* - вихідний тип даних;

- *newtype* - новий тип даних, що визначається.

Відмінність способу конструювання, який визначається функцією *MPI_Type_hvector*, полягає лише в тому, що параметр *stride* для визначення інтервалу між блоками задається в байтах, а не в елементах вихідного типу даних. За умови векторного способу новий похідний тип створюється як набір блоків з елементів вихідного типу, між блоками можуть бути регулярні проміжки. Розглянемо приклади використання даного способу конструювання типів:

- конструювання типу для виділення половини (тільки парних або тільки непарних) стрічок матриці з розміром $n \times n$:

MPI_Type_vector(n / 2, n, 2 * n , &StripRowTypo, &ElemType) ,

- конструювання типу для виділення стовпця матриці розміром $n \times n$:

MPI_Type_vector(n . 1, n, &ColumnType, & ElemType) ,

- - конструювання типу для виділення головної діагоналі матриці розміром $n \times n$:

MPI_Type_vector(n , 1, n + 1 , &DiagonalType, & ElemType) .

З урахуванням характеру прикладів, що наводяться, можна згадати наявну в MPI можливість створення похідних типів для опису підмасивів з використанням функції (дана функція передбачається стандартом MPI - 2):

int MPI_Type_create_subarray(int ndims, int *sizes, int *subsizes,
int *starts, int order , MPI_Data_type oldtype, MPI_Datatype *newtype) ,

де

- *ndims* - розмірність масиву;

- *sizes* - кількість елементів в кожній розмірності вихідного масиву;

- *subsizes* - кількість елементів в кожній розмірності підмасиву, що визначається;

- *starts* - індекси початкових елементів в кожній розмірності підмасиву, що визначається;

- *order* - параметр для вказівки необхідності перевпорядкування;

- *oldtype* - тип даних елементів вихідного масиву;

- *newtype* - новий тип даних для опису підмасивів.

Індексний спосіб конструювання. В разі індексного способу конструювання довільного типу даних в MPI використовуються функції:

```
int MPI_Type_indexed(int court , int blocklens[ ] , int indices[ ] ,
  MPI_Data_type oldtype , MPI_Datatype *newtype ) та
int MPI_Type_hindexed(int court , int blocklens[ ] , MPI_Aint indices[ ],
  MPI_Data_type oldtype , MPI_Datatype *newtype ) ,
```

де

- *count* - кількість блоків;

- *blocklens* - кількість елементів в кожному блоці;

- *indices* - індексування кожного блоку від початку типу;

- *oldtype* - вихідний тип даних;

- *newtype* - новий тип даних, що визначається.

Як випливає з опису, в разі індексного способу новий похідний тип створюється як набір блоків різного розміру з елементів вихідного типу, причому між блоками можуть бути різні проміжки пам’яті. Для пояснення цього способу можна навести приклад конструювання типу для опису верхньої трикутної матриці розміром $n \times n$:

// Конструювання типу для опису верхньої трикутної матриці *for (i = 0, i < n; i++)*

```
  blocklens[i] = n - i;
  indices    [i] = i * n+i;
}
MPI_Type_indexed(n , blocklens, indices, &UTMatrixType , ElemType).
```

Як і раніше, спосіб конструювання, який визначається функцією *MPI_Type_hindexed*, відрізняється тим, що елементи *indices* для визначення інтервалів між блоками задаються в байтах, а не в елементах вихідного типу даних. Існує ще одна додаткова функція *MPI_Type_create_indexed_block*

індексного способу конструювання для визначення типів з блоками однакового розміру (дана функція передбачається стандартом MPI - 2).

Структурний спосіб конструювання. Цей спосіб є самим загальним методом конструювання довільного типу даних за умови явного задавання відповідної карти типу. Використання такого способу здійснюється з використанням функції:

```
int MPI_Type_struct( int count , int blocklens[ ] , MPI_Aint indices[ ],
        MPI_Data_type oldtypes[ ] , MPI_Datatype *newtype ) ,
```

де

- *count* - кількість блоків;
- *blocklens* - кількість елементів в кожному блоці;
- *indices* - зміщення кожного блоку від початку типу (в байтах);
- *oldtypes* - вихідні типи даних в кожному блоці зокрема;
- *newtype* - новий тип даних, що визначається.

Структурний спосіб додатково до індексного методу дає змогу вказувати типи елементів для кожного блоку зокрема.

Оновлення похідних типів та їх видалення. Розглянуті вище функції конструювання дають змогу визначати довільний тип даних. Додатково перед використанням створений тип повинен бути оголошений з використанням функції:

```
int MPI_Type_commit(MPI_Datatype *type ) ,
```

де

- *type* - оголошений тип даних.

За умови завершення використання похідний тип повинен бути анульований з використанням функції:

```
int MPI_Type_free(MPI_Datatype *type ) ,
```

де

- *type* - тип даних, що анулюється.

Формування повідомлень з використанням запаковування та розпаковування даних.. Для використання даного способу слід визначити буфер пам’яті достатнього розміру для збирання повідомлень. Дані, що

входять до складу повідомлення повинні бути запаковані в буфер з використанням функції:

```
int MPI_Pack(void *data , int count , MPI_Datatype type , void *buf ,
  int bufsize , int *bufpos , MPI_Comm comm ) ,
```

де

- *data* - буфер пам’яті з елементами для пакування;

- *count* - кількість елементів в буфері;

- *type* - тип даних для елементів, що запаковуються;

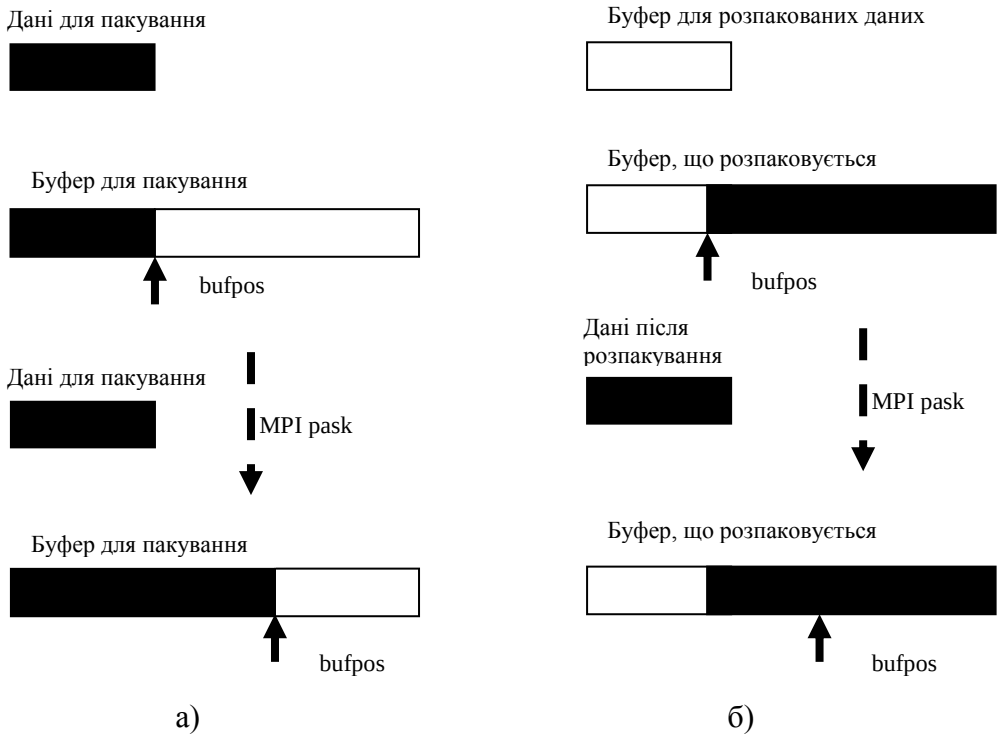
- *buf* - буфер пам’яті для пакування;

- *bufsize* - розмір буфера в пам’яті;

- *bufpos* - позиція для початку запису в буфер (в байтах від початку буфера);

- *comm* - комунікатор для запакованого повідомлення.

Функція MPI_Pack запаковує *count* елементів з буфера *data* до буфера пакування *buf*, починаючи з позиції *bufpos*. Загальна схема процедури а) - пакування і б) - розпаковування показана на рис. 9.4:



а) б)

Рисунок. 9.4 - Загальна схема а) пакування і б) розпакування даних

Початкові значення змінної *bufpos* повинно бути сформовано до початку запаковування і далі встановлюється функцією *MPI_Pack*. Виклик функції *MPI_Pack* здійснюється послідовно для запаковування всіх необхідних даних. Так, в раніше розглянутому прикладі набору змінних *a, b, n* для їх запаковування необхідно виконати:

```
bufpos = 0 ;
MPI_Pack(&a, 1 , MPI_DOUBLE , buf , buflen , &bufpos , comm ) ;
MPI_Pack(&b, 1 , MPI_DOUBLE , buf , buflen , &bufpos , comm ) ;
MPI_Pack(&n, 1 , MPI_DOUBLE , buf , buflen , &bufpos , comm ) ;
```

Для визначення необхідного розміру буфера для запаковування можна застосувати функцію:

```
int MPI_Pack_size(int count , MPI_Datatype type , MPI_Comm comm,
 int *size) ,
```

де

- *count* - кількість елементів в буфері;
- *type* - тип даних для запаковування елементів;
- *comm* - комунікатор для запакованого повідомлення;
- *size* - розрахований розмір буфера.

Після запаковування всіх необхідних даних підготовлений буфер можна використати в функціях передачі даних з вказівкою типу *MPI_PACKED*. Після отримання повідомлення з типом *MPI_PACKED* дані можна розпакувати з використанням функції:

```
int MPI_Unpack(void *buf , int bufsize , int *bufpos , void *data,
  int count , MPI_Datatype type , MPI_Comm comm ) ,
```

де

- *buf* - буфер пам’яті з запакованими даними;
- *bufsize* - розмір буфера в байтах;
- *bufpos* - позиція початку даних в буфері (в байтах від початку буфера);
- *data* - буфер пам’яті для розпакованих даних;
- *count* - кількість елементів в буфері;
- *type* - тип даних, що розпаковуються;

- *comm* -комунікатор для запакованого повідомлення.

Функція *MPI_Unpack* розпаковує, починаючи з позиції *bufpos*, чергову порцію даних з буфера *buf* і розміщує розпаковані дані в буфері *data*. Загальна схема процедури розпаковування показана на рис. 9.4 б). Початкове значення змінної *bufpos* повинно бути сформовано до початку розпаковування і далі встановлюється функцією *MPI_Unpack*. Виклик функції *MPI_Unpack* здійснюється послідовно для розпаковування всіх запакованих даних, за станом розпаковування повинен відповідати порядок запаковування. В раніше розглянутому прикладі запаковування для розпаковування запакованих даних необхідно виконати:

```
bufpos = 0 ;
MPI_Unpask(buf , buflen , &bufpos , &a , 1, MPI_DOUBLE , comm);
MPI_Unpack(buf , buflen , &bufpos , &b , 1, MPI_DOUBLE , comm);
MPI_Unpack(buf , buflen , &bufpos , &n , 1, MPI_DOUBLE , comm);
```

Оскільки такий підхід (застосування запаковування для формування повідомлень) приводить до появи додаткових дій з пакування і розпакування даних, то даний спосіб може бути виправданий за умови порівняно невеликих розмірів повідомлень та за малої кількості повторень. Запаковування та розпаковування може бути корисним за умови використання буферів для буферизованого способу передачі даних.

10 УПРАВЛІННЯ ГРУПАМИ ПРОЦЕСІВ І КОМУНІКАТОРАМИ.

Розглянемо можливості з управління групами процесів і комунікаторами. Процеси паралельної програми об’єднуються в групи. До групи можуть входити всі процеси паралельної програми; з іншого боку, в групі може знаходитися тільки частина наявних процесів. Один і той же процес може належати декільком групам. Управління групами процесів проводиться для створення на їх основі комунікаторів. Під комунікатором в MPI розуміють спеціально створюваний службовий об’єкт, який об’єднує в своєму складі групу процесів і ряд додаткових параметрів (контекст), які використовуються при виконанні операцій передачі даних. Парні операції передачі даних виконуються тільки для процесів, які належать одному і тому ж комунікатору. Колективні операції застосовуються одночасно для всіх процесів комунікатора. Комунікатори створюються для зменшення області дії колективних операцій і для усунення взаємного впливу різних виконуваних частин паралельної програми. Комунікаційні операції, що виконуються з використанням різних комунікаторів, є незалежними і не впливають одна на одну. Всі наявні в паралельній програмі процеси входять до складу створюваних за замовчуванням комунікатора та ідентифікатора MPI_COMM_WORLD. За необхідності передачі даних між процесами з різних груп необхідно створювати визначені в стандарті MPI - 2 глобальні комунікатори (*intercommunicator*). Взаємодія між процесами різних груп виявляється необхідною досить рідко.

Управління групами. Групи процесів можна створити з вже наявних груп. Як вихідну групу, можна використати групу, пов’язану з визначеним комунікатором MPI_COMM_WORLD. Іноді може бути корисним комунікатор MPI_COMM_SELF, визначений для кожного процесу

паралельної програми, який включає тільки цей процес. Для отримання групи, пов’язаної з існуючим комунікатором, застосовується функція:

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

де

- *comm* - комунікатор;

- *group* - група, пов’язана з комунікатором.

Далі, на основі наявних груп, можна створити нові групи:

- створення нової групи *newgroup* з наявної групи *oldgroup*, яка включатиме до себе *n* процесів - їх ранги перераховуються в масиві *ranks*:

```
int MPI_Group oldgroup, int n, int *ranks,
  MPI_Group *newgroup),
```

де

oldgroup - існуюча група;

- *n* - кількість елементів в масиві *ranks*;

- *ranks* - масив рангів процесів, які будуть включені до нової групи;

- *newgroup* - створена група. *newgroup* з групи *oldgroup*, яка включатиме:

- створення нової групи *n* процесів, ранги яких не співпадають з рангами, перерахованими в масиві *ranks*:

```
int MPI_Group_excl(MPI_Group olgroup, int n, int *ranks,
  MPI_Group *newgroup),
```

де

- *oldgroup* - наявна група;

- *n* - кількість елементів в масиві *ranks*;

- *ranks* - масив рангів процесів, які будуть видалені з нової групи;

-*newgroup* - створена група.

Для отримання нових груп з наявними групами процесів можна виконати операції об’єднання, перетину та різниці:

- створення нової групи *newgroup* як об’єднання груп *group1* та *group2*:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
```

MPI_Group *newgroup),

де

- *group1* - перша група;

- *group2* - друга група;

- *newgroup* - перетин груп;

- створення нової групи *newgroup* як перетину груп *group1* та *group2*:

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup),

де

- *group1* - перша група;

- *group2* - друга група;

- *newgroup* - перетин груп;

- створення нової групи *newgroup* як різниці груп *group1* та *group2*:

nt MPI_Group_вшааукутсу(MPI_Group group1, MPI_Group group2,
MPI_Group *newgroup),

де

- *group1* - перша група;

- *group2* - друга група;

- *newgroup* - перетин груп;

При конструюванні груп може виявитися корисною спеціальна порожня група MPI_COMM_EMPTY. Ряд функцій MPI забезпечує отримання інформації про групу процесів:

- отримання кількості процесів в групі:

int MPI_Group_size(MPI_Group group, int *size),

де

- *group* - група;

- *size* - кількість процесів в групі.

- отримання рангу поточного процесу в групі:

int MPI_Group_rank(MPI_Group group, int *rank),

де

- *group* - група;

- *rank* - ранг процесів в групі.

Після завершення використана група повинна бути видалена:

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

де

- *group* - група, яка підлягає видаленню (виконання цієї операції не впливає на комунікатори, в яких використовується група, що видаляється).

Управління комунікаторами. Розглянемо управління інтеркомунікаторами, які використовуються для операцій передачі даних всередині однієї групи процесів.. Для створення нових комунікаторів існують два основних способи:

1. Дублювання вже наявного комунікатора:

```
int MPI_Comm_dup(MPI_Comm oldcom, MPI_comm *newcom),
```

де

- *oldcom* - наявний комунікатор, копія якого створюється;

- *newcom* - новий комунікатор;

2. Створення нового комунікатора з підмножини процесів наявного комунікатора:

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

де

- *oldcom* - наявний комунікатор;

- *group* - підмножина процесів комунікатора *oldcom*;

- *newcom* - новий комунікатор.

Дублювання комунікатора може застосовуватись, наприклад, для усунення можливості перетину по тегам повідомлень в різних частинах паралельної програми (в тому числі і при використанні функцій різних програмних бібліотек). Операція створення комунікаторів колективна, тобто повинна виконуватися всіма процесами вихідного комунікатора. Для пояснення розглянутих функцій можна навести приклад створення комунікатора, в якому містяться всі процеси, окрім процесу, що має ранг 0 в комунікаторі MPI_Comm_WORLD (такий комунікатор може бути корисним

для підтримки схеми організації паралельних обчислень "менеджер-виконавці"):

```
MPI_Group WorldGroup, WorkerGroup;
MPI_Comm Workers;
int ranks[1];
ranks[0] = 0;
// Створення групи процесів в MPI_COMM_WORLD
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
// Створення групи без процесіу з рангом 0
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);
// Створення комунікатора по групі
MPI_Comm_create(MPI_COMM_WORLD, WorkerGroup, &Workers);
...
MPI_Group_Free(&WorkerGroup);
MPI_Comm_Free(&Workers);
```

Швидкий і корисний спосіб одночасного створення декількох комунікаторів забезпечує функція:

```
int MPI_Comm_split(MPI_Comm oldcom, int split, int key,
  MPI_Comm *newcomm),
```

де

- *nidcomm* - вихідний комунікатор:

- *split* - номер комунікатора, якому повинен належати процес:

- *key* - порядок рангу процеса в створюваному комунікаторі;

- *newcomm* - створюваний комунікатор.

Створення комунікатора належить до колективних операцій, тому виклик функції MPI_Comm_split повинен бути викликаний в кожному процесі комунікатора *oldcom.* В результаті виконання функції процеси розділяються на групи, що не перетинаються, з однаковим значенням параметра *split*. На основі сформованих груп створюється набір комунікаторів. Для того, щоб вказати, що процес не повинен входити до жодного із створюваних комунікаторів, слід скористатися константою MPI_UNDEFINDED як значення параметра *split*. При створенні комунікаторів для рангів процесів в новому комунікаторі вибирається такий порядок нумерації, щоб він відповідав порядку значень параметрів *key* (процес з великим значенням параметра *key* отримує великий ранг, процеси з однаковим значенням параметра *key* зберігають свою відносну нумерацію).

Як приклад, розглянемо задачу зображення набору процесів у вигляді двовимірної решітки. Нехай $p = q^* q$ є загальна кількість процесів: подальший наступний фрагмент програми забезпечує отримання комунікаторів для кожного рядка створюваної топології:

```
MPI_Comm comm;
int rank, row;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
row = rank / q;
MPI_Comm_split(MPI_COMM_WORLD, row, rank, &comm);
```

За умови виконання цього прикладу, наприклад, коли $p = 9$, процеси з рангами (0, 1, 2) утворюють перший комунікатор, процеси з рангами (3, 4, 5) - другий і т.д. Після завершення використання комунікатор повинен бути видалений, для чого використовується функція:

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

де

- *comm* - комунікатор, який підлягає видаленню.

Віртуальні топології. Під топологією обчислювальної системи розуміють структуру вузлів мережі і ліній між цими вузлами. Топологія може бути зображена у вигляді графа, в якому вершини є процесорами (процесами) системи, а дуги відповідають наявним лініям (каналам) зв'язку. Парні операції передачі даних можуть бути виконані між будь-якими процесами одного і того ж комунікатора, а в колективній операції приймають участь всі процеси комунікатора. Логічна топологія ліній зв'язку між процесами в паралельній програмі має структуру повного графа (незалежно від наявності реальних фізичних каналів зв'язку між процесами). Фізична топологія системи є такою, що апаратно реалізується, і зміні не підлягає (хоч існують засоби побудови мереж, що програмуються). Залишаючи незмінною фізичну основу, можна організувати логічне зображення будь-якої необхідної віртуальної топології. для цього достатньо, наприклад, сформувати той чи інший механізм додаткової адресації процесів. Використання віртуальних процесів може виявитися корисним з ряду причин. Віртуальна технологія, наприклад, може більше відповідати наявній структурі ліній передачі даних.

Застосування віртуальних технологій може помітно спростити зображення і реалізацію паралельних алгоритмів. В MPI підтримується два типи топологій - прямокутна решітка довільної розмірності (декартова топологія) та топологія графа довільного типу. Наявні в MPI функції забезпечують лише отримання нових логічних систем адресації процесів, що відповідають віртуальним топологіям, які формуються. Виконання всіх комунікаційних операцій повинно здійснюватися, як і раніше, з використанням звичайних функцій передачі даних з використанням вихідних рангів процесів.

Декартові топології (решітки). Декартові топології, в яких множина процесів представлена у вигляді прямокутної решітки, а для вказівки процесів використовується декартова система координат, широко застосовується в багатьох задачах для опису структури існуючих інформаційних залежностей. В числі прикладів таких задач - матричні алгоритми та мереєжеві методи розв'язування диференціальних рівнянь в частинних похідних. Для створення декартової топології (решітки) в MPI призначена функція:

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims,
  int *periods, int reorder, MPI_Comm *cartcom),
```

де

- *ndims* - вихідний комунікатор;

- *dims* - розмірність декартової решітки;

- *periods* - масив даних *ndims,* задає кількість процесів в кожному вимірі решітки;

- *reorder* - параметр допустимості зміни нумерації процесів;

- *cartcomm* - створюваний комунікатор з декартовою топологією процесів.

Операція створення топології є колективною і повинна виконуватися всіма процесами вихідного комунікатора. Для пояснення призначення параметрів функції MPI_Cart_create розглянемо приклад створення двомірної решітки 4×4, в якій стрічки і стовпці мають кільцеву структуру (за останнім процесом слідує перший процес):

```
// Створення  двомірної решітки 4×4
MPI-Comm  GridComm ;
int  dims[2],  periods[2],  reorder = 1;
dims[0]   = dims[1]     =4;
periods[0]  =  periods[1]  = 1;
MPT_Cart_create(MPT_COMM_WORLD,  2,  dims,  periods,  reorder,
    &GridComm) ;
```

Слід зауважити, що внаслідок кільцевої структури сформована в рамках прикладу топологія є тором. Для визначення декартових координат процесу за його рангом можна скористатися функцією:

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

де

- *comm* - комунікатор з топологією решітки;

- *rank* - ранг процесу, для якого визначаються декартові координати;

- *ndims* - розмірність решітки;

- *coords* - декартові координати процесу, що повертаються функцією.

Зворотна дія - визначення рангу процесу за його декартовими координатами, забезпечується використанням функції:

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

де

- *comm* - комунікатор з топологією решітки;

- *coords* - декартові координати процесу;

- *rank* - ранг процесу, що повертається функцією.

Важлива процедура розбиття решітки на підрешітки меншої розмірності забезпечується використанням функції:

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

де

- *comm* - вихідний комунікатор з топологією решітки;

- *subdims* - масив для вказівки, які виміри повинні залишитися в створюваній підрешітці;

- *newcomm* - створюваний комунікатор з підрешіткою.

Операція створення підрешіток також колективна і повинна виконуватися всіма процесами вихідного комунікатора. В ході виконання функція MPI_Cart_sub визначає комунікатори для кожного сполучення координат фіксованих вимірів вихідної решітки. Для пояснення функції MPI_Cart_sub доповнимо розглянутий раніше приклад створення двомірної решітки і визначимо комунікатори з декартовою топологією для кожного рядка і стовбчика решітки, зокрема:

```
// Створення комунікаторів для кожного рядка і стовбчика решітки
MPI_Comm  RowComm,  ColCom,
int  subdims[2];
// Створення  комунікаторів для рядків
subdims[0] = 0;  // фіксації виміру
subdims[1] = 1;  // наявність даного виміру в підрешітці
        RowComm);
// Створення комунікаторів для стовбчиків
subdims[0] =1;
subdims[1] =0;
MPI_Cart_sub(GridComm,  subdims,  & ColCom);
```

В наведеному прикладі для решітки з розміром 4×4 створюються 8 комунікаторів, по одному для кожного рядка та стовбчика решітки. Кожному процесу, що визначається комунікатором RawComm та ColComm, відповідають рядки та стовбчики, яким даний цей процес належить. Додаткова функція MPI_Cart_shift забезпечує підтримку процедури послідовної передачі даних по одному з вимірів решітки (операція зсуву даних). В залежності від періодичності виміру решітки, по якому виконується зсув, розрізняються два типи операції:

- циклічний зсув на k елементів впродовж виміру решітки - в *mod dim*, де *dim* є розміром виміру, впродовж якого відбувається зсув;

- лінійний зсув на k позицій впродовж виміру решітки - в цьому варіанті операції дані від процесу i пересилаються процесу $i+k$ (якщо такий існує).

Функція MPI_Cart_shift забезпечує отримання рангів процесів, з якими поточний процес (процес, який викликає функцію MPI_Cart_shift) повинен виконати обмін даними:

```
int MPI_Card_shift(MPI_Comm comm, int dir, int disp, int *source,
  int *dist),
```

де

- *comm* - комунікатор з топологією решітки;

- *dir* - номер виміру, по якому виконується зсув;

-*disp* - величина зсуву (за від'ємних значень зсув здійснюється до початку вимірювання);

- *source* - ранг процесу, від якого повинні бути отримані дані;

- *dst* - ранг процесу, якому повинні бути відправлені дані.

Слід зазначити, що функція MPI_Cart_shift тільки визначає ранги процесів, між якими повинен бути виконаний обмін даними в ході операції зсуву. Безпосередня передача даних може бути виконана, наприклад, з використанням функції MPI_Sendrecv.

Топології графа. Інформація про функції MPI для роботи з віртуальними топологіями типу граф будуть розглянуті більш коротко.

Для створення комунікатора з топологією типу граф в MPI призначена функція:

```
int MPI_Graph__create(MPI_Comm oldcom, int nnodes, int *index,
  int *edges, int reorder, MPI_Comm *graphcom),
```

де

oldcom - вихідний комунікатор;

nnodes - кількість вершин графа;

- *index* - кількість вихідних дуг для кожної вершини;

- *edges* - послідовний список дуг графа;

- *reoder* - параметр допустимості зміни нумерації процесів;

- *graphcom* - створюваний комунікатор з топологією типу граф.

Операція створення топології є колективною і, тим самим, повинна виконуватися всіма процесами вихідного комунікатора.

Для прикладу створимо топологію графа із структурою, поданою на рис. 10.1

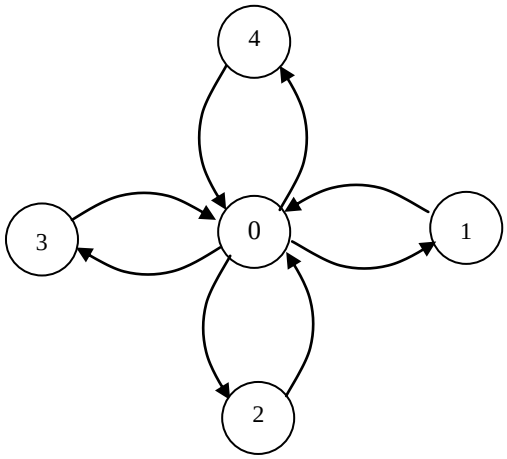


Рисунок 10.1 - Приклад графа для топології типу зірка

В цьому випадку кількість процесів становить 5, порядки вершин (кількості дуг, що виходять) приймають значення (4, 1. 1, 1, 1), а матриця інцидентності (номери вершин, для яких дуги є вхідними) має вигляд:

Процеси	*Лінії зв'язку*
0	1, 2, 3, 4
1	0
2	0
3	0
4	0

Для створення топології з графом даного типу необхідно виконати такий програмний код:

```
/*  Створення топології типу зірка   */
int   index[ ] = {4, 1, 1, 1, 1};
int   edges[ ] = {1, 2, 3, 4. 0, 0, 0, 0, };
MPI_Comm  StarComm  StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```

Наведемо ще дві корисні функції для роботи з топологіями графа. Кількість сусідніх процесів, в яких від процесу, що перевіряється, є вхідні дуги, можна отримати з використанням функцій:

```
int MPI_Graph_neighbors_count (MPI_Comm comm, int rank,
  int *nneighbors),
```

де:

- *comm* - комунікатор з топологією типу граф;
- *rank* - ранг процесу в комунікаторі;
- *nneighbors* - кількість сусідніх процесів.

Отримання рангів сусідніх вершин забезпечується функцією:

```
int MPI_Graph_neighbors(MPI_Comm comm, int rank,
  int *nneighbors),
```

де:

- *comm* - комунікатор з топологією типу граф;
- *rank* - ранг процесу в комунікаторі;
- *neighbors* - ранги сусідніх в графі процесів,
- *mneighbors* - розмір масиву *neighbors.*

Загальна характеристика середовища виконання MPI - програм.

Для проведення паралельних обчислень в обчислювальній системі повинно бути встановлене середовище виконання MPI - програм, яке б забезпечувало розробку, компіляцію, компонування та виконання паралельних програм. Для виконання першої частини перерахованих дій - розробки, компіляції і компонування достатньо звичайних засобів розробки програм (таких, як Microsoft Visual Studio), необхідна лише наявність тієї чи іншої бібліотеки MPI. Для виконання паралельних програм середовища розробки повинно мати додаткових можливостей, серед яких наявність засобів означення використовуваних процесорів, операцій віддаленого запуску програм і т.д. Бажаною є наявність в середовищі виконання засобів профілювання, трасування та відлагодження паралельних програм. На цьому стандартизація закінчується (стандарт MPI - 2 дає певні рекомендації про те,

як повинно бути організовано середовище MPI - програми, проте вони не є обов’язковими).

Існує декілька різних середовищ виконання МПІ – програм. В більшості випадків вони створюються сумісно з розробкою тих чи інших варіантів MPI - бібліотек. Як правило, вибір реалізації MPI – бібліотек, встановлення середовища виконання і підготування інструкцій з використання здійснюється адміністратором обчислювальної системи. Інформаційні ресурси Інтернету, на яких розміщено вільно використовувані реалізації MPI, та промислові версії MPI містять вичерпну інформацію про процедури встановлення MPI, і виконання всіх необхідних дій не є чимось складним. Запуск МПІ - програми також залежить від середовища виконання, але в більшості випадків ця дія відбувається з використанням команди *mpirum* (стандарт MPI - 2 рекомендує використовувати команду *mpiexec*). В числі можливих параметрів цієї команди:

- режим виконання - локальний або багатопроцесорний; локальний режим як правило відбувається з використанням ключа - *localony*, при виконанні паралельної програми в локальному режимі всі процеси програми знаходяться на комп’ютері, з якого був здійснений запуск програми. Такий спосіб виконання дуже корисний для початкової перевірки працездатності та відлагодження паралельної програми. Певний об’єм такої роботи може бути виконаний навіть на окремому комп’ютері за межами багатопроцесорної обчислювальної системи;

- виконуваний файл паралельної програми;

- командний рядок з параметрами для виконуваної програми.

Існує велика кількість інших параметрів, які використовуються при розробці достатньо складних паралельних програм - їх опис подається в довідково-інформаційній літературі з відповідного середовища виконання MPI - програм. Під час запуску програми на декількох комп’ютерах виконуваний файл програми слід скопіювати на ці комп’ютери або він

повинен знаходитися на загальному доступному для всіх комп'ютерів ресурсі.

Додаткові можливості стандарту MPI - 2. Стандарт MPI - 2 був прийнятий в 1997 р. Його використання до цього часу є не обмеженим. Основними причинами цього є певний консерватизм розробників програмного забезпечення, складність реалізації нового стандарту, та ін. Можливостей MPI - 1 достатньо для реалізації багатьох паралельних алгоритмів, а сфера застосування додаткових можливостей MPI - 2 виявляється не настільки широкою.

Наведемо коротку характеристику додаткових можливостей стандарту MPI - 2:

- динамічне породження процесів, коли процеси паралельної програми можуть створюватися і знищуватися під час виконання;

- однобічна взаємодія процесів, що дає змогу бути ініціатором операції передачі і прийому даних тільки одному процесу;

- паралельние введення/виведення, яке забезпечує спеціальний інтерфейс для роботи процесів з файловою системою;

- розширення можливостей колективних операцій, до числа яких входить (як приклад) взаємодія через глобальні комунікатори (intercommunicator);

- інтерфейс для алгоритмічних мов (C++).

Список використаних джерел

1. Корнеев В.В. Параллельные вычислительные системы. - М.: Нолидж, 1999 г.
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. - СПб.: БХВ-Петербург, 2002 г.
3. Гергель В.П. Теория и практика параллельных вычислений. Учебное пособие. - М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2007 г.
4. Кузьменко Б.В., Чайковська О.А. Технологія розподілених систем та паралельних обчислень. (конспект лекцій, частина 1. Розподілені об’єктні системи, паралельні обчислювальні системи та паралельні обчислення, паралельне програмування на основі MPI) Навчальний посібник. – К.: Видавничий центр КНУКІМ, 2011 – 126 с.
5. Гергель В.П. Теория и практика параллельных вычислений: учебное пособие /В.П. Гергель– М.:ИНТУИТ, 2016. – 501 с.
6. Дорошенко А.Ю. Паралельні обчислювальні системи. Методичний посібник і конспект лекцій. – Київ: Видавничий дім «КМ Академія», 2013.– 46 с.
7. Рольщиков В.Б. Технології розподілених систем та паралельних обчислень. Конспект лекцій. Одеса: ОДЕКУ 2016.155с
8. Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. – М.: ДМК Пресс, 2012. – 672 с:
9. Ashwin Pajankar. Raspberry Pi Supercomputing and Scientific Programming. – Nashik, Maharashtra, India, 2017. – 170 p.
10. Семеренко, В. П. Технології паралельних обчислень : навчальний посібник / Семеренко В. П. – Вінниця : ВНТУ, 2018. – 104 с.

11. Gropp, William. Using MPI : portable parallel programming with the Message-Passing Interface / William Gropp, Ewing Lusk, and Anthony Skjellum. Third edition. – Massachusetts Institute of Technology, 2014. – 330 с.

12. Таненбаум, Э. Распределенные системы: принципы и парадигмы : Пер. с англ.–СПб: «ПИТЕР», 2013. – 816 с.

13. Писарук Н.Н. Исследование операций.– Минск : БГУ, 2015. – 304 с.

14. Дистанційна освіта ЦНТУ. – URL: http://moodle.kntu.kr.ua/my/

15. Технології паралельного програмування URL: http://www.parallel.ru/tech

16. Сайт Української команди розподілених обчислень.– Режим доступу: http://distributed.org.ua/.

17. Стандарти MPI.– Режим доступу: http://www.mpiforum.org