МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ»
КАФЕДРА ПРИКЛАДНОЇ МАТЕМАТИКИ ТА ІНФОРМАТИКИ

І.А. Назарова, О.А. Дмитрієва

ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

НАВЧАЛЬНИЙ ПОСІБНИК

Покровськ
ДВНЗ «ДонНТУ»
2020

УДК 004.272.2:519.63
Н 19

Рекомендовано до друку Вченою радою ДВНЗ «Донецький національний технічний університет» (протокол № 7 від 24 вересня 2020 року).

Рецензенти:

Є.О. Башков, доктор технічних наук, професор, професор кафедри прикладної математики та інформатики Донецького національного технічного університету;

С.О. Ковальов, кандидат технічних наук, доцент, доцент кафедри комп'ютерної інженерії Донецького національного технічного університету, декан факультету комп'ютерних наук та технологій.

Назарова І.А., Дмитрієва О.А.

Н19 Паралельні обчислення: навчальний посібник / І.А. Назарова, О.А. Дмитрієва. – Покровськ: ДВНЗ «ДонНТУ», 2020. – 246с.

ISBN 978-966-377-237-0

Навчальний посібник «Паралельні обчислення» має своєю метою формування знань та вмінь ІТ-фахівців в області сучасних паралельних, багатопроцесорних комп'ютерних систем, методів розробки та оцінки ефективності алгоритмічного та програмного забезпечення для сучасних високопродуктивних комп'ютерів.

Посібник може бути корисним для студентів та аспірантів, що отримують освіту за спеціальностями галузі знань 12 Інформаційні технології.

УДК 004.272.2:519.63

ISBN 978-966-377-237-0

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

АМ	– Алгоритми маршрутизації
МД	– Матричний добуток
ММ	– Матричне множення
ОС	– Обчислювальні системи
ПЕ	– Процесорні елементи
ПОС	– Паралельні обчислювальні системи
CC-NUMA	– Cache-Coherent NUMA
COMA	– Cache-Only Memory Architecture
CTR	– Cut-through routing
DSM	– Distributed shared memory
FLOPS	– Floating Point Operations Per Second
MIMD	– Multiple Instruction, Multiple Data
MISD	– Multiple Instruction, Single Data
MPI	– Message Passing Interface
MPP	– Massively Parallel Processor
NCC-NUMA	– Non-Cache Coherent NUMA
NUMA	– Non-Uniform Memory Access
PVM	– Parallel Virtual Machine
PVP	– Parallel vector processor
SIMD	– Single Instruction, Multiple Data
SISD	– Single Instruction Single Data
SFR	– Store-and-forward routing
SMP	– Symmetric multiprocessor
RISC	– Reduced instruction set computer
UMA	– Uniform Memory Access

ВСТУП

Паралельні інформаційні системи – це один з найбільш нових та актуальних напрямків розвитку сучасного комп'ютінгу, тому важливою складовою у підготовці ІТ-спеціалістів є опанування методами та положеннями теорії паралельних обчислень. Матеріал, що представлений у навчальному посібнику, має своєю метою формування знань та вмінь в області сучасних паралельних, багатопроцесорних комп'ютерних систем, методів розробки та оцінки ефективності алгоритмічного і програмного забезпечення для сучасних суперкомп'ютерів.

Навчальний посібник «Паралельні обчислення» містить п'ять основних розділів. Перший розділ присвячено викладенню основ функціонування паралельних комп'ютерних систем та паралельних обчислювальних процесів. Приведено огляд сучасних архітектур та класів високопродуктивних обчислювальних систем, описано системи класифікації комп'ютерів, насамперед, таксономію Флінна, розглянуто класи ПОС за типом організації пам'яті.

Другий розділ містить матеріал, що присвячений питанням аналізу інформаційних потоків, що виникають при виконанні паралельних алгоритмів:

- дається загальна характеристика механізмів передачі даних;
- проводиться аналіз трудомісткості основних операцій обміну інформацією;
- розглядаються методи логічного представлення топологій міжпроцесорного зв'язку.

У третьому розділі розглянуті питання, що стосуються методів розробки та опису паралельних алгоритмів розв'язання реальних задач. Зокрема, введено поняття графів впливу та запропоновано поетапну

ієрархічну декомпозиційну методику для розпаралелювання послідовних алгоритмів. Докладно розглядаються системи метрик або динамічні характеристики оцінки якості використання отриманого паралелізму: тимчасові характеристики виконання арифметичних та обмінних операцій на p процесорах, коефіцієнти прискорення та ефективності, загальний ступінь паралелізму тощо. Розглядаються задачі визначення максимально досяжного паралелізму на базі законів Амдала та Густавсона-Барсіса. Вводиться поняття масштабованості паралельної системи: алгоритму у сукупності з паралельною архітектурою, на якій його реалізовано. Викладаються основи ізоефективного аналізу, вводиться функція ізоефективності, як метрика його масштабованості.

Четвертий розділ посвячено застосуванню загальних підходів, що описані у попередніх розділах, до паралельних обчислень у реалізації однієї з найбільш трудомістких операцій лінійної алгебри – матричного множення. В цьому розділі розглянуто найбільш ефективні класичні паралельні алгоритми для щільнозаповнених матриць з різними способами розподілу даних між процесорами. По-перше, це алгоритми з блоковим представленням даних: алгоритми Кеннона (узагальнення систолічного) та Фокса. По-друге, стрічкові алгоритми з горизонтальним та вертикальним розбиттям даних за процесорами. Для кожного з алгоритмів наведено загальний опис, визначено аналітичні вирази для основних динамічних характеристик таких, як прискорення, ефективність та загальні накладні витрати. Додатково, продемонстровано покрокове виконання алгоритмів з оцінками часу паралельного виконання обчислень та комунікацій. Окремий підрозділ присвячено паралельному алгоритму на основі швидкого рекурсивного матричного множення Штрассена-Копперсміта-Віноградa. Представлений алгоритм є поліалгоритмом, тобто комбінацією двох алгоритмів, Кеннона на верхньому рівні (між процесорами) та

рекурсивного на нижньому (всередині процесорів). До переваг його паралельної реалізації слід віднести високу масштабованість, а до недоліків певну чисельну нестійкість.

П’ятий розділ навчального посібника присвячено основам використання інтерфейсу передачі повідомлень, MPI, для організації паралельних обчислень у системах з розподіленою пам’яттю. Наведено короткий огляд історії розвитку стандарту MPI та його реалізацій. Детально розглянуто базові процедури та функції, двоточковий і колективний обмін даними, похідні типи, упаковка даних, групи процесів і комунікатори, віртуальні топології. У якості реалізації вибрано MS MPI від Microsoft, що відповідає стандарту MPI 1.0. Установка й настройка MS MPI в середовищі Windows наведена у додатку А.

В основу навчального посібника «Паралельні обчислення» частково покладено матеріали лекційних курсів, що викладаються авторами упродовж десятків років на кафедрі прикладної математики та інформатики факультету комп’ютерних наук та технологій ДВНЗ «Донецький національний технічний університет». Дослідження, що використані при написанні навчального посібника, проводились авторами на кафедрі ПМІ ДВНЗ «ДонНТУ» у рамках виконання науково-дослідних робіт за держбюджетними темами фундаментальних досліджень, затверджених МОН України. На майбутнє планується видати другу частину посібника, в яку увійдуть більш складні теми: організація паралельних обчислень при розв’язанні систем звичайних диференційних рівнянь та рівнянь з частинними похідними та паралельні алгоритми теорії графів та інше.

Автори висловлюють велику пошану своєму Вчителю, лауреату державної премії в галузі науки та техніки України, доктору технічних наук, професору Фельдману Льву Петровичу (1921-2017рр).

РОЗДІЛ 1
ОСНОВИ ПОБУДОВИ СУЧАСНИХ ПАРАЛЕЛЬНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

1.1 Класифікація паралельних обчислювальних систем

Сучасний етап розвитку обчислювальної техніки характеризується існуванням величезної кількості високопродуктивних паралельних обчислювальних систем. Велика різноманітність паралельних систем спричинила необхідність введення класифікації, що дозволяє диференціювати різні високопродуктивні обчислювальні засоби [18-24].

Вимоги до ефективної і коректної системи класифікації:

- можливість класифікації всіх, як існуючих, так і створюваних обчислювальних систем;
- диференціація суттєво різних обчислювальних структур;
- однозначність класифікації будь-якої ЕОМ або обчислювальної структури;
- наочність, простота і практична доцільність системи класифікації.

Однією із перших та найбільш поширених схем класифікації архітектур обчислювальних систем є так звана систематика або таксономія Майкла Флінна (*Flynn's taxonomy*). В її основу покладено поняття потоків виконуваних команд та оброблюваних даних і способи їх взаємодії [18-19].

В результаті такого підходу розрізняють наступні основні типи обчислювальних (ОС) систем:

1) SISD (Single Instruction, Single Data) – поодинокий потік команд та поодинокий потік даних;

2) SIMD (Single Instruction, Multiple Data) – поодинокий потік команд та множинний потік даних;

3) MISD (Multiple Instruction, Single Data) – множинний потік команд та поодинокий потік даних;

4) MIMD (Multiple Instruction, Multiple Data) – множинний потік команд та множинний потік даних.

Перший клас за систематикою Флінна, SISD (Single Instruction, Single Data, «Поодинокий потік команд та поодинокий потік даних»), містить комп'ютери, які мають рівно один потік команд, всі команди обробляються послідовно одна за одною і кожна команда ініціює одну операцію з одним потоком даних (рис. 1.1-1.2).

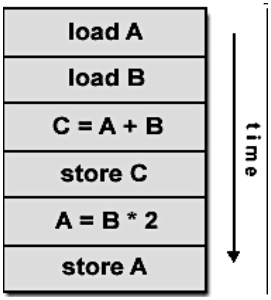


Рисунок 1.1 – Схема обробки даних у ОС класу SISD

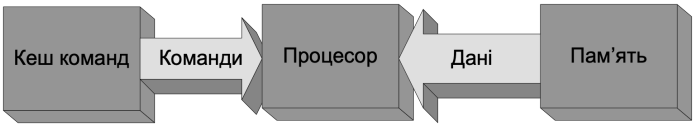


Рисунок 1.2 – Структура обчислювальних систем класу SISD за таксономією Флінна

Власне кажучи, в даному класі не використовується паралелізм, ані за даними, ані за командами, з цього приводу, SISD-машина не є паралельною. До SISD-класу відносяться класичні, традиційні послідовні

машини або машини фон-неймановського типу, наприклад, PDP-11 або VAX 11/780. Для збільшення швидкості обробки команд і виконання арифметичних операцій може застосовуватися конвеєрна обробка. У такому випадку в цей клас потраплять і такі системи, як CRAY-1, CYBER 205, машини сімейства FACOM VP і багато інших.

Другий клас за таксономією Флінна, SIMD (Single Instruction, Multiple Data, «Поодинокий потік команд та множинний потік даних»), ще називають *синхронною паралельністю* (рис. 1.3-1.4).

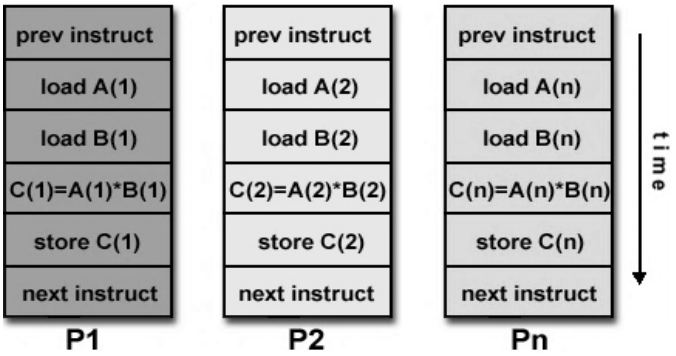


Рисунок 1.3 – Схема обробки даних у ОС класу SIMD

Подібний клас систем складають обчислювальні системи, в яких в кожен момент часу може виконуватися одна і та ж команда для обробки декількох інформаційних елементів. Підхід широко використовувався в попередні роки, зараз «чистих» представників класу SIMD зовсім небагато. Класичні SIMD-системи – ILLIAC IV або CM-1 компанії Thinking Machines. Саме представники класу SIMD вперше досягли продуктивності порядку GFLOPS.

Основні представники класу SIMD: векторні процесори, матричні процесори і процесори з архітектурою VLIW.

Найбільш популярна ідея класу SIMD – векторне процесування. Векторний процесор підтримує обробку не тільки скалярних, але і векторних операндів. Ефективне декодування інструкцій і зручні дані позначаються на продуктивності вкрай позитивно. У матричних процесорів немає аналогії з векторними. Матричний процесор – це масив процесорів з єдиним потоком команд. Великий вихідний масив даних поділяють на частини, що підлягають ідентичною обробці. Кожен процесор масиву обробляє відповідну частину даних, виконуючи єдиний потік інструкцій.

Перспективним представником класу SIMD є архітектура VLIW (дуже довге командне слово). Одна інструкція в такій системі команд є кортеж з кількох RISC інструкцій, які є незалежними за даними між собою. Один з найпотужніших VLIW процесорів – Intel Itanium 2.

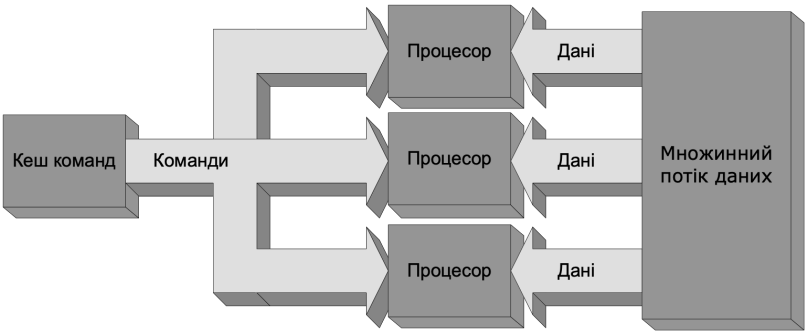


Рисунок 1.4 – Структура обчислювальних систем класу SIMD за таксономією Флінна

Третій клас за систематикою Флінна, MISD (Multiple Instruction, Single Data, «Множинний потік команд і поодинокий потік даних») (рис. 1.5) вважається деякими авторами пустим. Ряд фахівців вважають, що якщо прикладів конкретних ЕОМ даного типу ОС не існує зовсім, то

введення подібного класу робиться тільки для повноти системи класифікації. Інші ж відносять до даного типу ОС, наприклад, систолічні ОС або системи з конвеєрною обробкою даних.

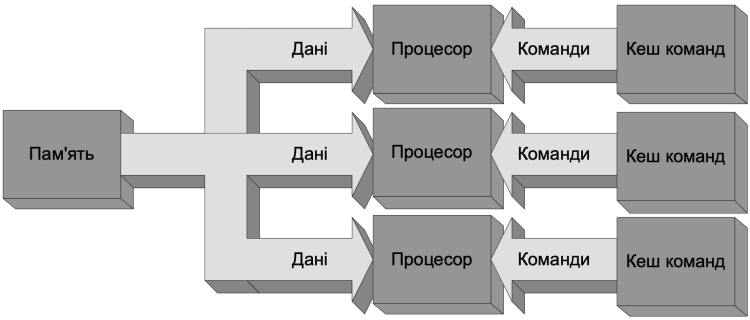


Рисунок 1.5 – Структура обчислювальних систем класу MISD за таксономією Флінна

Четвертий клас за систематикою Флінна – MIMD (Multiple Instruction, Multiple Data, «Множинний потік команд та множинний потік даних») є основним і найбільш заповненим (рис. 1.6-1.7).

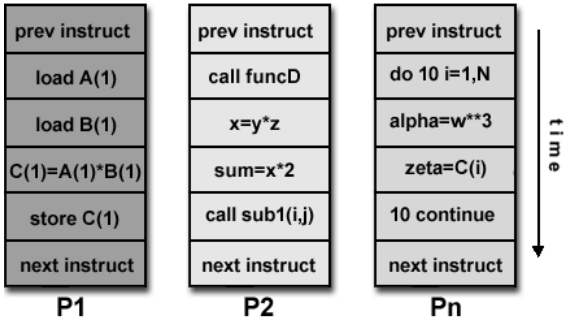


Рисунок 1.6 – Схема обробки даних у ОС класу MIMD

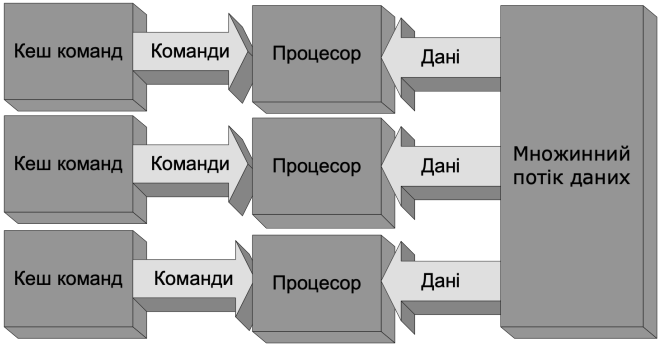


Рисунок 1.7 – Структура обчислювальних систем класу MIMD за таксономією Флінна

Клас MIMD включає в себе багатопроцесорні системи, де процесори обробляють множинні потоки даних. Сюди відносять і традиційні мультипроцесорні машини, багатоядерні і багатопотокові процесори, а також комп'ютерні кластери.

Класифікація Флінна широко використовується при початковій характеристиці комп'ютерних систем, проте вона має певні недоліки. Практично усі види паралельних комп'ютерів, незважаючи на їх суттєву різнорідність, належать до однієї MIMD-групи. Як результат, багатьма дослідниками робилися неодноразові спроби деталізації цієї систематики [21-24]. Недоліком класифікації Флінна є ще й те, що деякі важливі системи, наприклад dataflow та векторно-конвеєрні машини, чітко не вписуються у дану класифікацію.

Існують кілька інших способів класифікації обчислювальних систем для паралельної обробки даних [1-3, 23]. Більш детальна систематизація комп'ютерів класу MIMD запропонована Р. Хокні [2, 21]. Е. Джонсон проводить класифікацію MIMD-архітектур на основі структури пам'яті і реалізації механізму взаємодії і синхронізації між

процесорами. Т. Фенг класифікує обчислювальні системи на основі двох простих характеристик, числа біт n в машинному слові, що обробляються паралельно і числа слів m, оброблюваних одночасно даною обчислювальною системою. На особливу увагу заслуговує спосіб структурної нотації, що дозволяє з високою точністю описати багато характерних особливостей комп'ютерних систем. Слід зазначити, що запропоновано ще значне число інших способів класифікації: Хендлера, Шнайдера, Скіллкорна і т. д. [1-3, 23].

Таким чином, величезне різноманіття архітектурних рішень для паралельних обчислювальних систем призводить до того, що створення ефективного і масштабованого алгоритмічного і програмного забезпечення для них є трудомістким процесом.

1.2 Основні класи сучасних паралельних комп'ютерів

MIMD-архітектури далі класифікуються у залежності від фізичного способу організації пам'яті, тобто, чи має процесор свою власну локальну пам'ять і звертається до інших блоків пам'яті, використовуючи комутуючу мережу, або комутуюча мережа підключає усі процесори до загальнодоступної пам'яті [1-3].

Виходячи з цього розрізняють наступні типи паралельних MIMD-архітектур (рис. 1.8):

1) комп'ютери із розподіленою пам'яттю (*distributed memory*);

2) комп'ютери із загальною пам'яттю (*true shared memory*);

3) комп'ютери із віртуальною загальною пам'яттю (*virtual shared memory*).

Деякі автори [2, 7] обчислювальні системи із загальною пам'яттю називають *мультипроцесорами*, а системи із розподіленою пам'яттю – *мультикомп'ютерами*. У відповідності із типом доступу до пам'яті

вводяться основні класи паралельних комп'ютерів [24]: SMP, MPP, PVP, NUMA-архітектури та кластери (рис. 1.8).

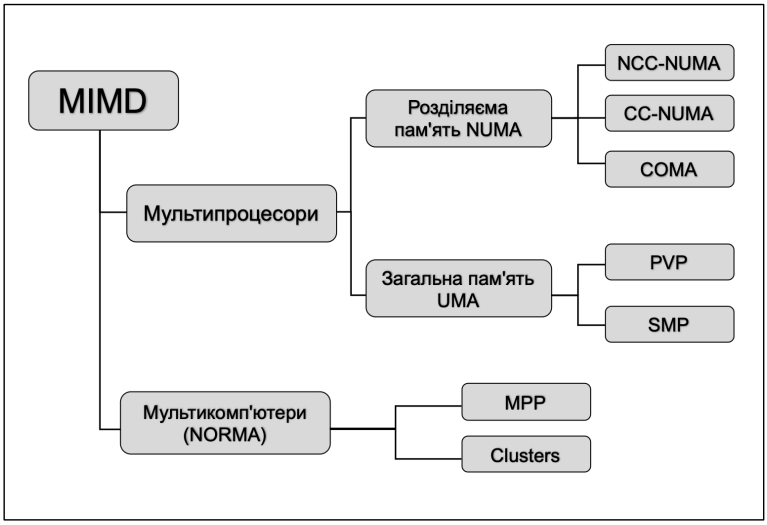


Рисунок 1.8 – Схема класифікації MIMD-систем

Мультипроцесори із загальною пам’яттю в свою чергу поділяються на два класи (рис. 1.9):

1) симетричні мультипроцесорні системи: SMP – *symmetric multiprocessor*;

2) паралельні векторні системи: PVP – *parallel vector processor*.

Однорідний доступ до пам'яті: UMA – *Uniform Memory Access.*

Симетричні мультипроцесорні системи [20-24] зазвичай складаються із декількох однакових процесорів та масиву загальної пам'яті. Усі процесори мають рівні можливості щодо доступу до єдиного адресного простору, такі ОС називають *системами із однорідним доступом до пам'яті.*

SMP-системи мають ряд недоліків:

– доступ з різних процесорів до загальних даних і забезпечення в зв'язку з цим однозначності (когерентності) кешей (*cache coherent problem*);

– необхідність синхронізації взаємодії одночасно виконуваних потоків команд.

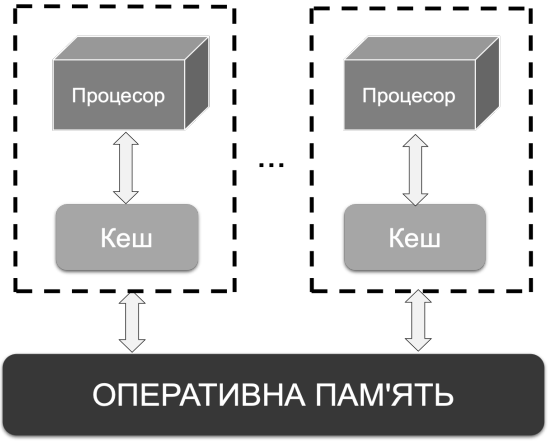


Рисунок 1.9 – Мультипроцесори із загальною пам’яттю

MIMD-системи

Наявність загальної пам'яті спрощує взаємодію процесорів між собою, однак накладає сильні обмеження на їх кількість, дана архітектура не є добре масштабованою і відмовостійкою.

Прикладами SMP-систем можуть служити HP 9000 V-class, N-class; SMP-сервера і робочі станції на базі процесорів Intel: IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu.

Основною ознакою паралельних *векторних систем* є наявність спеціальних векторно-конвеєрних процесорів, в яких передбачені команди однотипної обробки векторів незалежних даних, що ефективно виконуються на конвеєрних функціональних пристроях.

Як правило, кілька таких процесорів працюють одночасно над загальною пам'яттю (аналогічно SMP) в рамках багатопроцесорних конфігурацій. Декілька таких вузлів можуть бути об'єднані за допомогою комутатора (аналогічно MPP). Приклади PVP-систем: NEC SX-4/SX-5, CRAY J90/T90.

Наступні класи: мультипроцесори із загальною пам'яттю, що розділяється (*distributed shared memory* – DSM) або розділяємою пам'яттю (рис. 1.10). Основна характеристика – загальний доступ до даних при фізично розподіленій пам'яті або неоднорідний доступ до пам'яті – *Non-Uniform Memory Access* (NUMA).

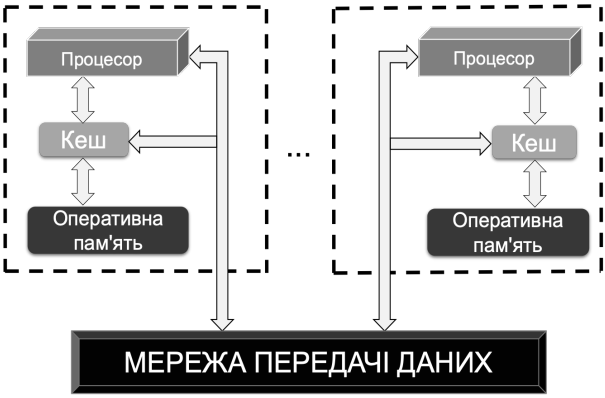


Рисунок 1.10 – Мультипроцесори із загальною розділяємою пам'яттю MIMD-системи (NUMA)

Мультипроцесори NUMA:

1) системи, що використовують тільки локальну кеш-пам'ять процесорів, *Cache-Only Memory Architecture* – COMA (KSR-1, DDM);

2) системи, в яких забезпечується когерентність локальних кешей різних процесорів, *Cache-Coherent NUMA* – CC-NUMA (SGI Origin 2000, Sun HPC-10000, IBM/Sequent NUMA-Q 2000) ;

3) системи, в яких забезпечується загальний доступ до локальної пам'яті різних процесорів без підтримки на апаратному рівні когерентності кешей – *Non-Cache Coherent NUMA* – NCC-NUMA (Gray T3E).

Мультикомп'ютери (рис. 1.11):

– масивно-паралельні системи: MPP – *Massively Parallel Processor*;

– кластери: *Clusters.*

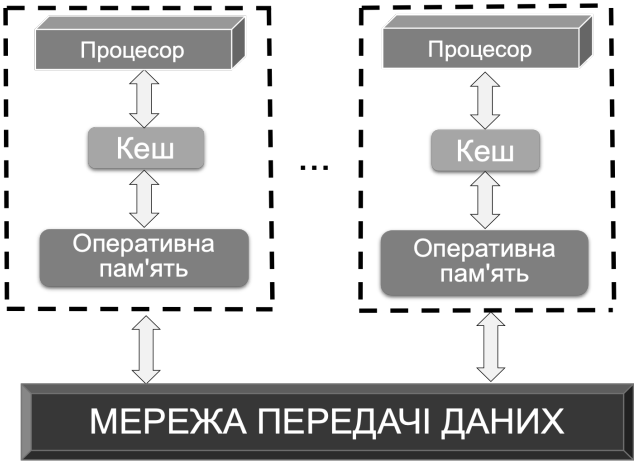


Рисунок 1.11 – Мультикомп’ютер або система MIMD-типу розподіленою пам’яттю

Масивно-паралельні системи (MPP) [20-24] складаються з декількох однорідних обчислювальних вузлів, як правило, RISC-архітектури. Кожен такий вузол має свою локальну пам'ять, один або декілька центральних процесорів та іноді жорсткий диск, причому прямий доступ до пам'яті інших вузлів неможливий. Вузли зазвичай пов'язані через комунікаційне середовище.

Доступ до віддаленої пам'яті реалізується у рамках моделі передачі повідомлень на основі бібліотек MPI (Message Passing Interface), PVM (Parallel Virtual Machine).

Системи масового паралелізму добре масштабуються, загальне число процесорів може досягати декількох тисяч і значно більше. Основний недолік таких систем полягає в складності організації обміну інформацією між процесорами. Прикладами MPP-систем служать: IBM RS/6000 SP2, Intel PARAGON, комп'ютери серії БОС.

Системи із неоднорідним доступом до пам'яті (NUMA – *Non Uniform Memory Access*) представляють собою щось середнє між SMP та MPP. У NUMA пам'ять фізично розподілена, але є логічно загальнодоступною. Підтримується єдиний адресний простір, апаратно підтримується доступ до віддаленої пам'яті. При цьому доступ до локальної пам'яті процесорів в кілька разів швидше, ніж до віддаленої пам'яті. Масштабованість NUMA-систем обмежується обсягом адресного простору та можливостями операційної системи. Найбільш відомі NUMA-системи: SGI Origin 2000, Sun HPC 10000, IBM / Sequent NUMA-Q 2000, SNI RM600.

В останні роки широке поширення одержали *кластерні системи*, як дешева альтернатива суперкомп'ютерам [2-3,24].

Кластер – група комп'ютерів, об'єднаних в локальну обчислювальну мережу і здатних працювати в якості єдиного ресурсу.

Кластер передбачає більш високу надійність і ефективність, ніж локальні мережі, і істотно більш низьку вартість в порівнянні з іншими паралельними ОС (за рахунок типових апаратних і програмних рішень).

Кластери є логічним продовженням ідей, що закладені в архітектурі MPP-систем. Розвиток комунікаційних технологій, а саме, поява високошвидкісного мережевого обладнання та спеціального програмного забезпечення, що реалізує механізм передачі повідомлень

над стандартними мережевими протоколами, зробили кластерні технології загальнодоступними.

При об'єднанні комп'ютерів різної потужності чи архітектури, говорять про *неоднорідні* або *гетерогенні* кластери, в іншому разі про *однорідні* або *гомогенні* [2-3]. Зазвичай використовуються або прості однопроцесорні персональні комп'ютери, або двох- або чотирьох - процесорні SMP-сервери. При цьому не накладається ніяких обмежень на склад і архітектуру вузлів. Кожен з вузлів може функціонувати під управлінням своєї власної операційної системи. Найчастіше використовуються стандартні операційні системи: Linux, Tru64 UNIX, Windows NT. Для зв'язку вузлів використовується одна зі стандартних мережевих технологій таких, як Fast / Gigabit Ethernet на базі шинної архітектури або комутатора. Ряд фірм пропонують спеціалізовані кластерні рішення на основі більш швидкісних мереж, таких як SCI фірми Scali Computer, а також Mirynet і InfiniBand.

В списку TOP500 найбільш високопродуктивних систем кластери становлять більшу частину, це Beowulf-кластери, NCSA NT Supercluster [10, 24]. Відносно мала вартість кластерних систем обертається великими накладними витратами на взаємодію паралельних процесів, що звужує потенційний клас розв'язуваних задач.

Ще одним напрямком розвитку сучасної паралельної індустрії є комбінування різних архітектур в одній системі і побудова *гібридних систем.* Гібридна архітектура втілює в собі зручності систем із загальною пам'яттю і відносну дешевизну систем з розподіленою пам'яттю. Суть цієї архітектури – в методі організації пам'яті, а саме: пам'ять є фізично розподіленою по різних частинах системи, але такою, що логічно розділяється, так що користувач бачить єдиний адресний простір. Система утворюється з однорідних базових модулів (плат), що складаються з невеликого числа процесорів і блоку пам'яті.

Модулі об'єднані за допомогою високошвидкісного комутатора. Підтримується єдиний адресний простір, апаратно підтримується доступ до віддаленої пам'яті, тобто до пам'яті інших модулів. При цьому доступ до локальної пам'яті здійснюється у декілька разів швидше, ніж до віддаленої. По суті архітектура NUMA є MPP-архітектурою (масивно-паралельною), де як окремі обчислювальні елементи беруться SMP-вузли.

На завершення усього сказаного важливо відзначити, що величезна кількість архітектурних рішень для паралельних обчислювальних систем призводить до того, що створення ефективного та масштабованого алгоритмічного та програмного забезпечення для них є трудомістким процесом. Тому на сьогодні не викликає сумнівів, що реалізувати потенційні можливості паралельних комп'ютерів можна тільки на основі проведення цілеспрямованої теоретичної роботи із побудови нових та адаптації існуючих алгоритмів розв'язання складних науково-технічних задач.

1.3 Завдання для самостійної роботи

1. Навести приклади паралельних обчислювальних систем зі списку TOP-500 або TOP-50 за останньою редакцією. Класифікувати їх за систематикою Флінна.
2. Провести огляд та аналіз сучасної редакції TOP-500 на предмет існування паралельних комп’ютерів типу SIMD або їх особливостей на мікрорівні.
3. Провести огляд та аналіз списку TOP-500 на предмет існування паралельних комп’ютерів типу MISD або таких, що мають деякі риси цього класу.
4. Провести огляд та аналіз (потужність, розробник, країна, тощо) сучасної редакції списків TOP-500 та TOP-50 на предмет

приналежності паралельних комп'ютерів до типу MIMD, визначити тип фізичного способу організації пам'яті.

5. Провести огляд та аналіз (країна, розробник, потужність, топології тощо) суперкомп'ютерів класу SMP у сучасній редакції списків TOP-500 та TOP-50.

6. Провести огляд та аналіз (країна, розробник, потужність, топології тощо) суперкомп'ютерів класу PVP у сучасній редакції списків TOP-500 та TOP-50.

7. Розгляньте способи забезпечення когерентності кешей в системах із загальною пам'яттю.

8. Провести огляд та аналіз (країна, розробник, потужність, топології тощо) суперкомп'ютерів класу MPP у сучасній редакції списків TOP-500 та TOP-50.

9. Провести огляд та аналіз (країна, розробник, потужність, топології тощо) кластерних систем у сучасній редакції списків TOP-500 та TOP-50.

10. Виконайте огляд та аналіз додаткових способів класифікації комп'ютерних систем. Наведіть приклади застосування класифікаторів Фенга, Хендлера, Шнайдера, Скіллкорна тощо.

1.4 Контрольні питання

1. У чому можуть полягати відмінності паралельних обчислювальних систем?

2. Що покладено в основу класифікації Флінна? Що таке SISD, SIMD, MISD, MIMD типи обчислювальних систем? Дайте визначення потоку команд та даних.

3. На чому базується деталізація класу MIMD за моделлю Хокні? Опишіть особливості MIMD-систем із загальною, віртуальною загальною та розподіленою пам'яттю.

4. У чому полягає зміст поділу багатопроцесорних систем на мультипроцесори та мультикомп'ютери?

5. В чому полягають принципи доступу UMA та NUMA?

6. Що таке симетричні мультипроцесорні системи? Які переваги та недоліки мають представники класу SMP? Які обчислювальні системи із списку TOP-500 є представниками цього класу? У чому полягають позитивні і негативні сторони симетричних мультипроцесорних систем?

7. Що таке паралельні векторні системи? Які переваги та недоліки мають представники класу PVP? Які обчислювальні системи із списку TOP-500 є представниками цього класу? У чому полягають позитивні і негативні сторони симетричних мультипроцесорних систем?

8. Що таке масивно паралельні системи? Які переваги та недоліки мають представники класу MPP? Які обчислювальні системи із сучасної версії списку TOP-500 є представниками цього класу? У чому полягають позитивні і негативні сторони масивно паралельних систем?

9. Що таке кластерні системи? Які переваги та недоліки мають представники цього класу? Які обчислювальні системи із сучасної версії списку TOP-500 є представниками цього класу? У чому полягають позитивні і негативні сторони кластерних систем? Які системні платформи можуть бути використані для побудови кластерів?

10. Які паралельні системи належать до класу мультипроцесорів NUMA? Що таке когерентність локальних кешей? Дати визначення мультипроцесорним системам типу COMA, CC-NUMA, NCC-NUMA. Навести приклади обчислювальних систем із сучасної версії списку TOP-500, що є представниками цих типів мультипроцесорів.

РОЗДІЛ 2
ХАРАКТЕРИСТИКА ТИПОВИХ ТОПОЛОГІЙ ТА ОЦІНКА КОМУНІКАЦІЙНОЇ ТРУДОМІСТКОСТІ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

Паралельні обчислювальні системи мають у своєму складі поле процесорних елементів/процесорів, а також один або декілька модулів пам'яті. Для організації обміну даними ці функціональні компоненти повинні бути пов'язані через відповідні комутаційні мережі. Всі високорівневі концепції комунікацій реалізуються в паралельних системах деякими фізичними структурами.

Система зв'язку між процесорами характеризує топологію зв'язку процесорів паралельного комп'ютеру, прикладами якої є: шинний зв'язок; статична топологія (лінійка, кільце, зірка, сітка-матриця, тор, кліка, гіперкуб тощо), динамічна топологія (зі змінними зв'язками), комутатори.

2.1 Основні характеристики топології мережі передачі даних

У силу специфіки задач, що вирішуються на паралельних ОС [1-4], топологія мережі передачі даних повинна задовольняти певним вимогам. По-перше, забезпечувати зв'язок між будь-якими двома процесорами або модулями пам'яті. З другого боку, підтримувати максимальну кількість одночасних зв'язків, щоб затрати на комутацію не обмежували паралельну обробку інформації.

Основні характеристики топологій та їх позначення:

- p – кількість процесорних елементів (ПЕ) або процесорів в мережі;
- V – кількість ліній зв'язку у одного процесора або ПЕ.

Діаметр або *комунікаційна довжина* (*diameter*), *A* – максимальна відстань між двома процесорами мережі (під відстанню зазвичай розуміється величина найкоротшого шляху). Дана величина характеризує максимально-необхідний час для передачі даних між двома процесорами, оскільки час передачі прямо пропорційний довжині шляху між ними.

Зв'язність (*connectivity*) – показник, що характеризує наявність різних маршрутів передачі даних між процесорами мережі. Конкретний вид даного показника може бути визначений, наприклад, як мінімальна кількість дуг, які треба видалити для розподілу мережі передачі даних на дві незв'язні області.

Ширина бінарного розподілу (*bisection width*) – мінімальна кількість дуг, які треба видалити для поділу мережі передачі даних на дві незв'язні області однакового розміру.

Вартість (*cost*) – для оцінювання вартості мережі використовують багато критеріїв [4], найчастіше, наприклад, загальну кількість ліній передачі даних в паралельній ОС.

Порівняння ефективності застосування комутаційних мереж може бути проведено на основі введених характеристик. При зменшенні діаметру та вартості, очевидно, зростає ефективність використання тієї чи іншої топології, але бажано мати мережу з високим рівнем зв'язності, оскільки вона знижує конкуренцію за комунікаційні ресурси. Параметри, число процесорів і кількість ліній зв'язку характеризують витрати на виготовлення системи, а діаметр мережі – витрати експлуатації.

Як видно з наведених вище міркувань, кількість зв'язків на один процесор і дистанція між двома процесорами повинні бути мінімальними. Структура зв'язку також повинна бути розширюваною, тобто малі комутаційні мережі повинні мати можливість збільшуватися за рахунок доповнень.

2.2 Типові топології міжпроцесорного зв'язку

До числа найбільш популярних топологій міжпроцесорного зв'язку за типом «точка-точка» зазвичай відносять наступні схеми: лінійка/кільце, сітка/тор, гіперкуб, кліка або повний граф, зірка, деревоподібні топології тощо [1-4].

Лінійка (*linear array* або *farm*) представляє собою лінійний масив процесорів (рис. 2.1). Кожен процесор з'єднаний з двома сусідніми (з попереднім і наступним) крім кінцевих процесорів, які мають по одному з'єднанню. Перевага цієї схеми – простота. Недолік полягає в тому, що дані, перш ніж досягти свого кінцевого призначення, повинні пройти через кілька проміжних процесорів, $p - 1$.

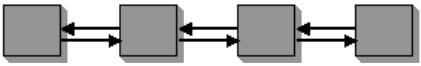


Рисунок 2.1 – Топологія лінійний масив

Топологія *кільце*/1-D *тор* (*ring*) (рис. 2.2) – модифікація лінійки шляхом з'єднання першого і останнього процесорів. Зв'язки у топології кільце можуть бути *односпрямованими* (дані можуть переміщатися тільки, наприклад, за годинниковою стрілкою) або *двоспрямованими*.

Кільцева структура зберігає переваги лінійки: число зв'язків на один процесор дорівнює 2 та скорочує максимальну відстань між процесорами – $p / 2$.

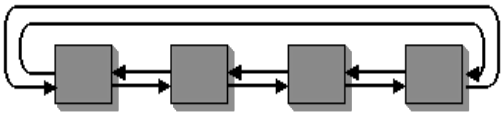


Рисунок 2.2 – Топологія кільце або 1D-тор

Сітка (2-D *сітка, матриця – mesh*) являє собою систему, в якій граф ліній зв'язку утворює прямокутну сітку, тобто процесорні елементи розташовані у вигляді правильної двовимірної таблиці та кожен процесор (крім крайніх) з'єднаний із чотирма сусідами (рис. 2.3).

Перевагою схеми є простота, а недолік полягає в тому, що при обміні між віддаленими процесорами дані повинні пройти через ряд проміжних процесорів. Всі квадратні сітки мають максимальну відстань між процесорами порядку $O(\sqrt{p})$. Така схема з'єднань "північ-південь-схід-захід" була застосована в комп'ютері IILIAC IV, 64 процесори якого становили масив 8×8.

Більшість реально побудованих сіткових масивів використовують двовимірні схеми з'єднання. Крім двовимірних є схеми з тривимірним масивом процесорів [2-4], в яких кожен процесор з'єднаний з шістьма найближчими сусідами. У кубічній, 3D-сітці, процесори розміщені у просторі кубу, максимальна відстань між процесорними елементами пропорційна кореню кубічному із p: $\sqrt[3]{p}$.

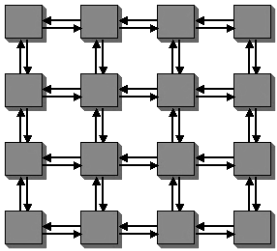


Рисунок 2.3 – Топологія матриця або 2-D сітка

Якщо у сітці граничні процесори з'єднати лініями зв'язку, то вийде замкнутий варіант сітки або *тор* (*torus*). Тороїдальні 2-D схеми з'єднання мають діаметр, пропорційний $\sqrt{p}$ (рис. 2.4).

Квадратний тор з чотирма лініями зв'язків у кожного процесора має максимальну відстань між процесорними елементами: $\sqrt{p}$. Аналогічна схема комутації з 8 лініями зв'язків має діаметр, що дорівнює $2\lfloor\sqrt{p}/2\rfloor$.

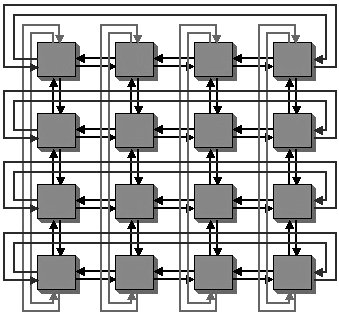


Рисунок 2.4 – Топологія замкнута сітка або 2-D тор

Комунікаційна структура *кліка* або *повний граф* (*clique, completely-connected*) – передбачає існування ліній зв'язку між будь-якими двома процесорами в системі (рис. 2.5).

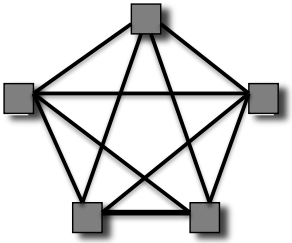


Рисунок 2.5 – Топологія кліка або повний граф

При повній зв’язності системи з p процесорів, з кожного процесора виходить $p-1$ лінія зв’язку, що робить вартість такої схеми досить великою при великому p.

Перевагою такої схеми з'єднання є мінімальна величина діаметра, що дорівнює одиниці.

Зірка (*star*) – топологічна система, в якій всі процесори мають лінії зв'язку з деяким управляючим процесором (рис. 2.6). Дана топологія є ефективною, наприклад, при організації централізованих схем паралельних обчислень.

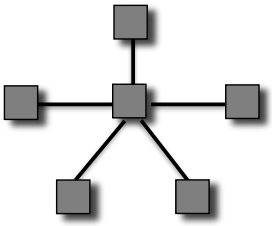


Рисунок 2.6 – Топологія зірка

Дерево (*tree*) – це ціла сукупність топологій, в яких кожен вузол вищого рівня пов'язаний з вузлами нижчого рівня зіркоподібним зв'язком. Деревоподібні топології ще називають ієрархічними зірками. На рисунку 2.7 наведено приклад такої топології – повне двійкове дерево (кожна вершина має рівно нуль або два вузла нижчого рівня).

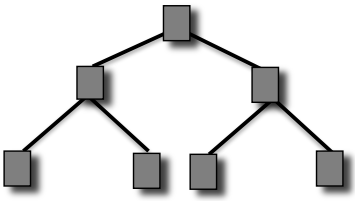


Рисунок 2.7 – Топологія повне бінарне дерево

Основна проблема дерева полягає в тому, що пропускна здатність між секціями дорівнює пропускній здатності каналів. Зазвичай у корні дерева спостерігається дуже великий потік обміну інформацією, тому

верхні вузли стають перешкодою для підвищення продуктивності. Можна вирішити цю проблему, збільшивши пропускну здатність верхніх каналів.

Топологія «*товсте дерево*» (*fat tree*) – топологія комп'ютерної мережі, що була винайдена Чарльзом Лейзерсоном в 1985 році, і є однією з найдешевших та ефективніших для суперкомп'ютерів [4]. Процесори локалізовані в листях дерева, внутрішні вузли дерева скомпоновані у внутрішню мережу. Піддерева можуть спілкуватися між собою, не зачіпаючи більш високих рівнів мережі.

На відміну від класичної топології дерево, в якій всі зв'язки між вузлами однакові, зв'язки в «товстому дереві» стають ширшими (товстими, продуктивними за пропускною здатністю) з кожним рівнем у міру наближення до кореня дерева. Часто використовують подвоєння пропускної здатності на кожному рівні.

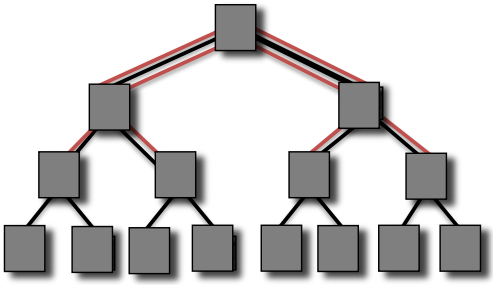


Рисунок 2.8 – Топологія «товсте дерево»/fat tree

Наприклад, самі нижні канали матимуть пропускну здатність x, наступний рівень – пропускну здатність $2x$, а кожен канал верхнього рівня – пропускну здатність $4x$. Така схема є одним з варіантів реалізації топології «товсте дерево», вона, наприклад, була застосована в комерційних мультикомп'ютерах Thinking Machines CM-5.

В існуючих обчислювальних системах однією з найбільш часто використовуваних та ефективних є топологія міжпроцесорних зв'язків – гіперкуб. Структура *гіперкуб* (*hypercube*) є окремим випадком сітки, коли за кожним розміром сітки є тільки два процесори, тобто гіперкуб містить $p = 2^N$ процесорів при розмірності N. При тривимірній схемі з'єднання процесори розміщені у вершинах тривимірного куба, а ребра куба грають роль локальних зв'язків між процесорами. Кожен процесор з'єднаний з трьома найближчими сусідами. Схема з'єднання чотиривимірного гіперкуба та її розгортка на площині наведені на рис. 2.9-2.10.

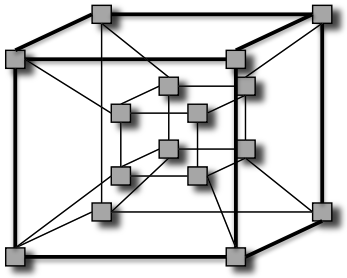


Рисунок 2.9 – Топологія гіперкуб $p = 2^4$

Топологія гіперкуб надає можливість логічно моделювати деякі важливі типи зв'язків: лінійка, кільце, сітка. Недоліком описаної архітектури є існування практичної межі збільшення розміру гіперкуба. При гіперкубовій архітектурі кількість зв'язків між процесорами невелика: у N -мірному гіперкубі кожний процесор має N сусідів. Невеликим порівняно з кількістю процесорів виявляється й діаметр системи, рівний $log_2 p$.

Гіперкуб є універсальною і одною з найбільш ефективніших систем зв'язку, її концепції використані в обчислювальних системах Intel iPSC, Ncube Corp. та інших.

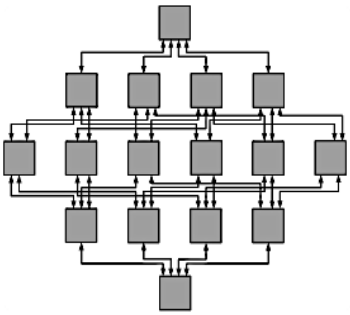


Рисунок 2.10 – Розгортка на площині схеми з'єднання процесорів в чотиривимірному гіперкубі

Характеристики описаних вище топологій з'єднання процесорів у мережу для передачі даних наведено у таблиці 2.1.

Таблиця 2.1 – Характеристики топологій «точка-точка»

Топологія	Діаметр	Ширина бісекції	Зв'язність	Вартість (кількість лінків)
Лінійка	$p-1$	1	1	$p-1$
Кільце/1D-тор	$\lfloor p/2 \rfloor$	2	2	p
Двовимірна сітка	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2D-тор	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
Гіперкуб	$\log(p)$	$p/2$	$\log(p)$	$p\log(p)/2$
Повнозв'язний 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	$p-1$
Замкнутий k-арний d-куб	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	dp

Реальні паралельні обчислювальні системи зазвичай використовують кілька різних схем з'єднання процесорів. Це дозволяє поєднувати кращі якості відомих топологій і мінімізувати недоліки. Так, наприклад:

- паралельна ОС MasPar MP-1 функціонує з використанням двох мереж: сіткової структури і триступеневої мережі Клосса;
- гібридна система Finite Element Mashine використовує сітку для зв'язків між сусідніми процесорами і шину для більш віддалених;
- комп'ютер Connection Mashine має 16 повнозв’язних процесорів в групі і групи об'єднуються мережею топології гіперкуб.

Іншим способом поліпшити характеристики схем комутації є побудова систем із змінною конфігурацією. Змінювати шаблон з'єднань можна як статично (до виконання програми), так і динамічно (в ході її виконання).

2.3 Аналіз трудомісткості основних операцій передачі даних

Час передачі даних між процесорами визначає *комунікаційну складову* (time communication) тривалості виконання паралельного алгоритму для паралельних систем з розподіленою пам’яттю [2-4].

Існує два принципово різних метода передачі даних в паралельних мультикомп’ютерах:

1) метод передачі повідомлень;
2) метод передачі пакетів.

Метод передачі повідомлень (*store-and-forward routing or SFR*) здійснює передачу даних як неподільних/атомарних блоків інформації (рис. 2.11):

- процесор, що містить повідомлення для передачі, готує весь обсяг даних для передачі, визначає процесор, якому необхідно надіслати дані, і запускає операцію пересилання даних,

- процесор, якому направлено повідомлення, в першу чергу здійснює прийом всіх даних, що пересилаються, повністю і тільки потім приступає до пересилання прийнятого повідомлення далі за маршрутом.

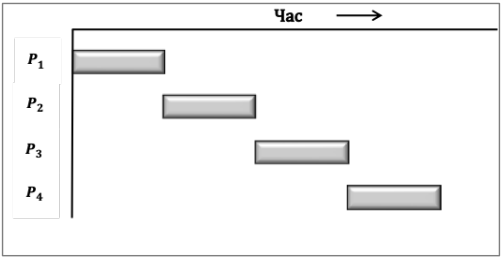


Рисунок 2.11 – Ілюстрація методу передачі повідомлень

Другий спосіб комунікації, *метод передачі пакетів* (*cut-through routing or CTR*), ґрунтується на представленні повідомлень, що пересилаються, у вигляді блоків інформації меншого розміру – *пакетів*, в результаті чого передача даних зведена до передачі пакетів (рис. 2.12). Таким чином, цілісне повідомлення розбивається на d частин-пакетів, далі дані пересилаються пакетами.

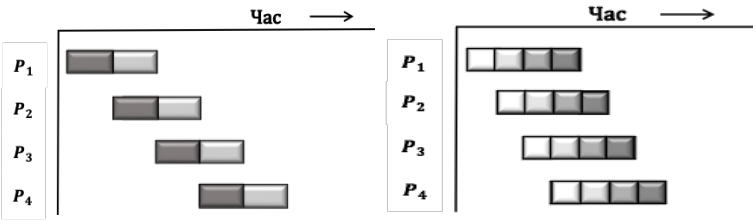


Рисунок 2.12 – Ілюстрація методу передачі пакетів, $d = 2 \vee d = 4$

Приймаючий процесор може здійснювати пересилання даних щодо подальшого маршруту безпосередньо відразу після прийому чергового пакета, не чекаючи завершення прийому даних повного повідомлення.

Переваги методу передачі пакетів:

- призводить до прискорення процесу пересилання даних;

- знижує потребу в пам'яті для зберігання даних, що пересилаються, при організації прийому-передачі повідомлень;

- для передачі можуть використовуватися одночасно різні комунікаційні канали.

Недоліки:

- вимагає розробки більш складного апаратного і програмного забезпечення мережі;

- може збільшити накладні витрати (час підготовки і час передачі службових даних);

- при передачі пакетів можливе виникнення конфліктних ситуацій (дедлоків).

Для оцінки часу виконання операції передачі одного повідомлення обсягом V байтів між двома процесами, локалізованими на різних процесорах/вузлах при розподіленій пам'яті, використовується наступна спрощена лінійна модель, запропонована Хокні [20]:

$$T_{p-p} = t_s + t_w \cdot V \cdot l, t_w = \frac{y}{B}, \quad (2.1)$$

де t_s – латентність, (час на підготовку повідомлення для передачі, пошук маршруту у мережі, тощо), на практиці величину латентності визначають за часом передачі повідомлення нульової довжини;

l – довжина маршруту (залежить від топології та алгоритму маршрутизації);

t_w – час передачі одного байту, тривалість подібної передачі визначається пропускною здатністю комунікаційних каналів мережі;

y – кількість байтів у слові;

B – пропускна здатність каналу передачі даних (байт/секунда).

Зауваження: в загальному випадку основними характеристиками швидкодії мережі є латентність та пропускна здатність. Припустимо, що на двох процесорах паралельної системи працюють два процеси, між якими за допомогою мережі пересилаються повідомлення. У передачі інформації, крім апаратних пристроїв, вочевидь бере участь і програмне забезпечення, наприклад реалізація інтерфейсу передачі повідомлень MPI. Латентністю вважається час, що витрачається програмним забезпеченням і пристроями мережі на підготовку до передачі інформації з даного каналу, тобто повна латентність складається з програмної і апаратної складових.

Розрізняють такі види пропускної здатності мережі:

1) пропускна здатність однонаправлених пересилань ("точка-точка", *uni-directional bandwidth*), що дорівнює максимальній швидкості, з якою процес на одному вузлі може передавати дані іншому процесу на іншому вузлі;

2) пропускна здатність двонаправлених пересилань (*bidirectional bandwidth*), що дорівнює максимальній швидкості, з якою два процеси можуть одночасно обмінюватись інформацією в мережі.

Модель Хокні для методу передачі повідомлень можна зробити більш точною за рахунок доданку, пов'язаного з часом передачі службових даних:

$$T_{p-p} = t_s + (t_w \cdot V + t_h) \cdot l, \tag{2.2}$$

де t_h – час передачі службових даних для двох сусідніх процесорів (між якими є фізичний канал), до службових даних відноситься заголовок повідомлення, блок даних для виявлення помилок передачі тощо.

Для методу передачі пакетів модель Хокні має наступний вигляд:

$$T_{p-p} = t_s + t_w \cdot V + t_h \cdot l. \tag{2.3}$$

Зауваження: при досить довгих повідомленнях часом передачі службових даних можна знехтувати і вираз для часу передачі даних може бути записано в більш простому вигляді – лінійна модель Хокні.

2.3.1 Оцінка трудомісткості комунікацій при режимі передачі повідомлень

Оскільки паралельні методи розв'язання динамічних задач часто використовують структурну інформаційну взаємодію за типом комунікацій, надається можливість розробити єдині комунікаційні примітиви для різних операцій передачі даних у різних топологіях.

Розглянемо топології, що найбільш поширені у використанні: лінійка/кільце, сітка/тор, гіперкуб та три основні комунікаційні операції:

1) передача даних між двома процесорами мережі – операція типу "точка-точка" або "point-point";

2) передача даних від одного процесора всім іншим процесорам мережі – операція "один-усім" або "one-to-all";

3) передача даних від усіх процесорів мережі усім процесорам мережі – множинна пересилка "усі-усім" або "all-to-all".

Трудомісткість одиночної операції пересилання даних між двома процесорами може бути отримана шляхом підстановки довжини максимального шляху (діаметра мережі) в модель Хокні (2.1).

Для обчислення часу виконання множинної пересилки в умовах тієї ж моделі необхідно вибрати алгоритм маршрутизації.

Алгоритми маршрутизації (АМ**)** визначають шлях передачі даних від процесора-джерела інформації до процесора, до якого ці дані повинні бути доставлені.

Розрізняються:

1) оптимальні (що визначають найкоротші шляхи) і неоптимальні АМ;

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

До числа найбільш поширених детермінованих оптимальних алгоритмів передачі даних відносять *методи оптимальної покоординатної маршрутизації* (*dimension-ordering routing*) [2-4].

Ідея цих методів полягає в тому, що пошук шляхів передачі даних здійснюється послідовно для кожної розмірності розглянутої топології.

Для двовимірної сітки: передача даних спочатку виконується по одному напрямку (наприклад, горизонталі X), а потім дані передаються вздовж іншого напрямку (відповідно, вертикалі Y). Для сіток метод має назву – алгоритм XY- маршрутизації. Для гіперкуба: циклічна передача даних процесору, який визначається першою відмінною бітовою позицією в номерах процесорів, на якому повідомлення розташовується в даний момент часу і на який повідомлення має бути передано.

2.3.1.1 Оцінка трудомісткості комунікацій для топології «кільце» при режимі передачі повідомлень

Для кільцевої топології (рис. 2.2) кожен процесор може ініціювати розсилку повідомлення в будь-якому обраному напрямку за кільцем. Без істотних обмежень припускаємо, що будь-який процесор має можливість здійснювати одночасно прийом та передачу даних (так звані, двонаправлені лінки). Таким чином, для топології кільце час виконання поодинокої пересилки даних складає:

$$T_{p-p}^{R} = t_s + t_w \cdot V \cdot \lfloor p/2 \rfloor. \quad (2.4)$$

Передача даних від одного процесора всім іншим для топології кільце в режимі передачі повідомлень визначається співвідношенням:

$$T_{one-to-all}^{R} = (t_s + t_w \cdot V) \cdot \lceil p/2 \rceil. \quad (2.5)$$

Для кільцевої топології процесор-джерело розсилки може ініціювати передачу даних послідовно двом своїм сусідам, які, в свою чергу, прийнявши повідомлення, організовують пересилання далі по кільцю. Так, на рисунку 2.13 кількість поодиноких пересилок сусіднім процесорам дорівнює $\lceil p/2 \rceil$=4, процесор-ініціатор має 0 номер.

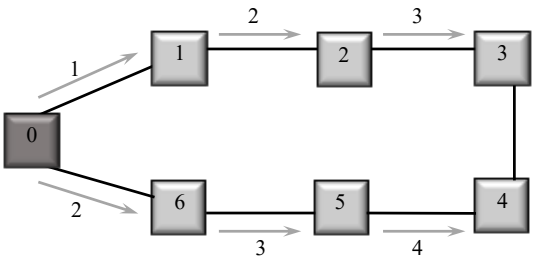


Рисунок 2.13 – Ілюстрація операції «один-усім» для кільцевої топології в режимі передачі повідомлень, $p = 7$

На першому кроці процесор з номером 0 передає процесору з номером 1 дані обсягом V, потім одночасно процесор 0 передає 6, а процесор 1 передає 2 ті ж самі дані і так далі.

Час виконання пересилки "усі-усім" для топології кільце в режимі передачі повідомлень становить:

$$T^R_{all-to-all} = (t_s + t_w \cdot V) \cdot (p - 1). \quad (2.6)$$

Для виконання множинної пересилки кожен процесор може ініціювати розсилку свого повідомлення одночасно в будь-якому обраному напрямку за кільцем. Завершення операції множинної розсилки відбудеться через $(p - 1)$ цикл передачі даних.

Наприклад, на рисунках 2.14-2.15 наведено ілюстрацію виконання перших двох та останнього кроку ($p = 8$, $p - 1 = 7$) операції за типом «усі- усім» в режимі передачі повідомлень у кільцевій топології.

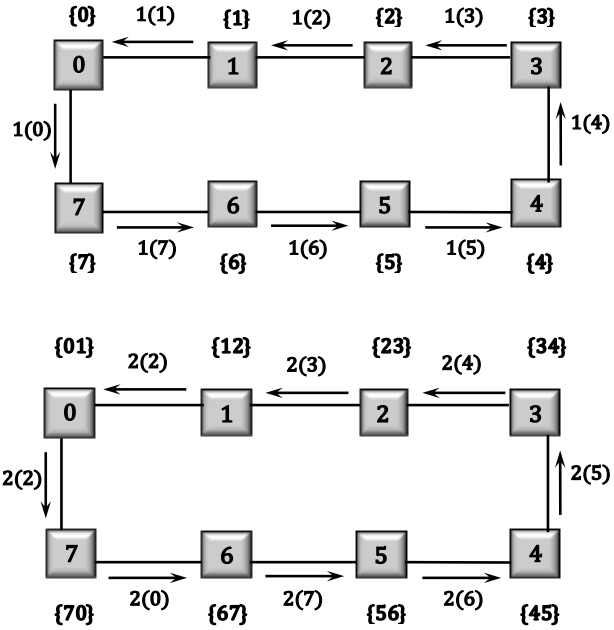


Рисунок 2.14 –Ілюстрація 1 та 2 кроків виконання операції «усі-усім» для топології кільце в режимі передачі повідомлень, $p = 8$

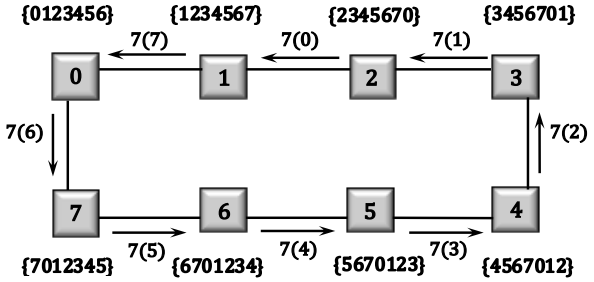


Рисунок 2.15 – Ілюстрація останнього $p - 1 = 7$ кроку виконання операції «усі-усім» для топології кільце в режимі передачі повідомлень, $p = 8$

У фігурних дужках наведено номери повідомлень, що спочатку складали вміст поточного процесору на кроці. Так, на першому кроці, кожен процесор має тільки одне повідомлення, що співпадає з номером цього процесору: на першому процесорі – повідомлення {1}, на другому – повідомлення {2} і так далі. Пара $x(y)$ на рисунках означає наступне: x – номер кроку, y – номер повідомлення, що передається на цьому кроці. Таким чином, за сім ($p - 1 = 7$) кроків кожен процесор отримає дані від кожного іншого процесору.

2.3.1.2 Оцінка трудомісткості комунікацій для топології «2D-тор» при режимі передачі повідомлень

Для топології замкнута двовимірна сітка або 2D-тор (рис. 2.4) поодинока операція пересилки даних вимагає наступного часу для виконання з урахуванням моделі (2.1) та діаметра топології:

$$T_{p-p}^{M} = t_s + t_w \cdot 2V \cdot \left\lfloor \sqrt{p}/2 \right\rfloor \tag{2.7}$$

Операція передачі «один-усім» для топології 2D-тор виконується за алгоритмом покоординатної маршрутизації. Ілюстрація її виконання в умовах, що процесор-ініціатор має номер 0, наведена на рисунку 2.16.

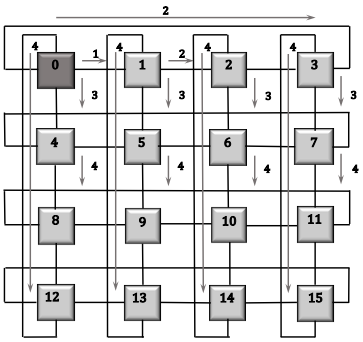


Рисунок 2.16 – Ілюстрація виконання операції «один-усім» для топології 2D-тор в режимі передачі повідомлень, $p = 16$

Пересилка даних від одного процесора всім іншим на топології двовимірний тор отримана з цього ж способу передачі даних для кільцевої топології, але виконаного у два етапи.

На першому етапі виконується передача даних всім процесорам мережі, розташованим на тій же горизонталі сітки, що і процесор-ініціатор передачі:

$$T^M_{one-to-all,1} = (t_s + t_w \cdot V) \cdot \lceil \sqrt{p}/2 \rceil.$$

На другому етапі процесори, які отримали копію даних на першому етапі, розсилають повідомлення за своїми відповідними вертикалями:

$$T^M_{one-to-all,2} = (t_s + t_w \cdot V) \cdot \lceil \sqrt{p}/2 \rceil.$$

Таким чином, загальний час на операцію «один-усім» для топології 2D-тор в режимі передачі повідомлень складає:

$$T^M_{one-to-all} = 2(t_s + t_w \cdot V) \cdot \lceil \sqrt{p}/2 \rceil. \quad (2.8)$$

Множинне розсилання повідомлень «усі-усім» для топології типу 2D-тор також може бути виконано за допомогою алгоритму, одержаного узагальненням способу передачі даних для кільцевої мережі на основі ідеї оптимальної покоординатної маршрутизації:

$$T^M_{all-to-all} = 2t_s \cdot \left(\sqrt{p} - 1\right) + t_w \cdot V \cdot (p - 1). \quad (2.9)$$

На першому етапі виконується передача повідомлень окремо по всім процесорам мережі, розташованим на одних і тих же горизонталях тору. В результаті на кожному процесорі однієї і тієї ж горизонталі формуються укрупнені повідомлення розміру $\sqrt{p} \cdot V$, які об'єднують всі повідомлення горизонталі.

Час виконання першого етапу операції «усі-усім» складає:

$$T^M_{all-to-all,1} = (t_s + t_w \cdot V) \cdot \left(\sqrt{p} - 1\right). \quad (2.10)$$

На другому етапі розсилка даних виконується за процесорами мережі, що створюють вертикалі сітки. Час виконання другого етапу множинного розсилання «усі-усім»:

$$T^{M}_{all-to-all,2} = (t_s + t_w \cdot \sqrt{p} \cdot V) \cdot \left(\sqrt{p} - 1\right). \quad (2.11)$$

Загальна тривалість операції множинної розсилки «усі-усім» для 2D- тору складається з двох доданків:

$$T^{M}_{all-to-all} = T^{M}_{all-to-all,1} + T^{M}_{all-to-all,2},$$

$$T^{M}_{all-to-all} = 2(t_s + t_w \cdot \sqrt{p} \cdot V) \cdot \left(\sqrt{p} - 1\right). \quad (2.12)$$

Ілюстрація виконання операції наведена на рисунках 2.17-2.22. Зауваження: кожен етап реалізує обмін у рамках топологій «кільце» спочатку за вертикаллю, потім – горизонталлю і виконується за $\sqrt{p}$ -1 кроків.

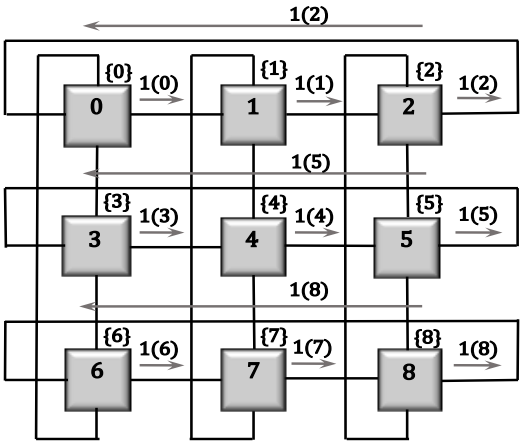


Рисунок 2.17 – Ілюстрація виконання 1 кроку 1 етапу операції «усі-усім» для топології 2-D тор в режимі передачі повідомлень, $p = 9$

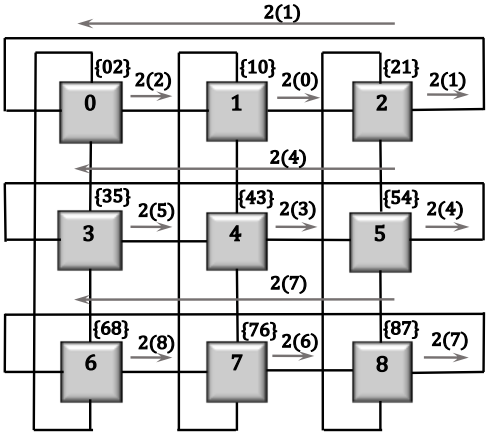
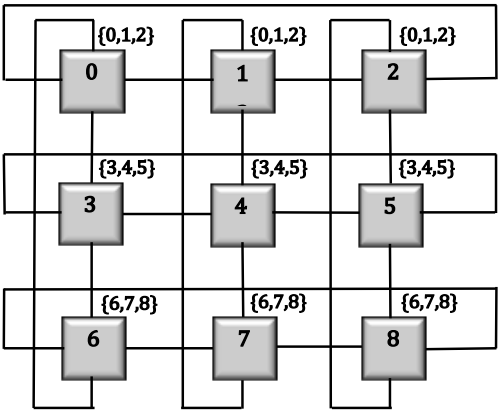


Рисунок 2.18 – Ілюстрація виконання 2 кроку 1 етапу операції «усі-усім» для топології 2-D тор в режимі передачі повідомлень, $p = 9$



$$T^{M}_{all-to-all,1} = (t_s + t_w \cdot V) \cdot \left(\sqrt{p} - 1\right)$$

Рисунок 2.19 – Результат виконання 1 етапу операції «усі-усім» для топології 2-D тор в режимі передачі повідомлень, $p = 9$

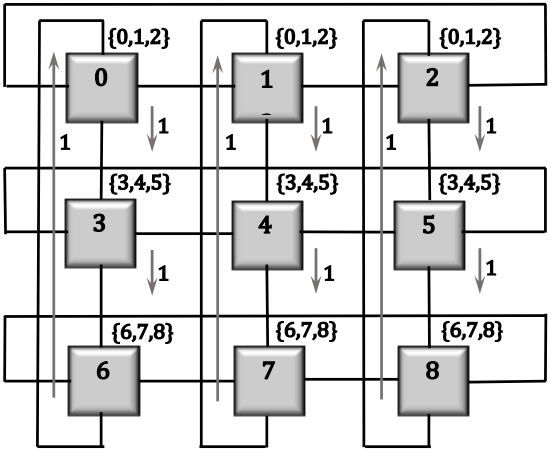


Рисунок 2.20 – Ілюстрація виконання 1 кроку 2 етапу операції «усі-усім» для топології 2-D тор в режимі передачі повідомлень, $p = 9$

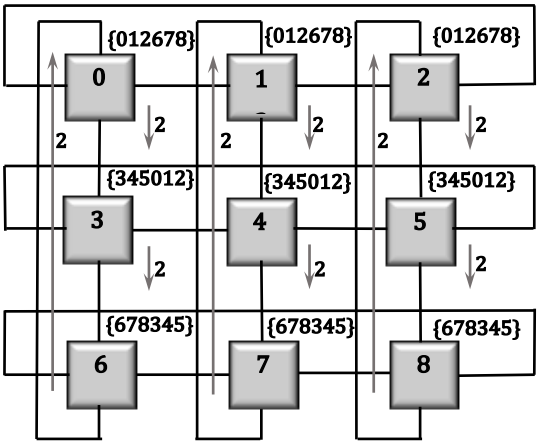
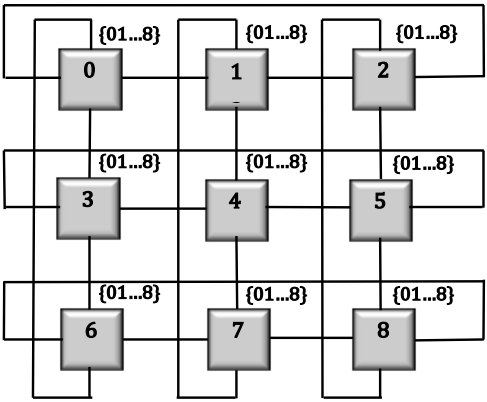


Рисунок 2.21 – Ілюстрація виконання 2 кроку 2 етапу операції «усі-усім» для топології 2-D тор в режимі передачі повідомлень, $p = 9$



$$T^M_{all-to-all,2} = (t_s + t_w \cdot \sqrt{p} \cdot V) \cdot \left(\sqrt{p} - 1\right),$$

$$T^M_{all-to-all} = 2(t_s + t_w \cdot \sqrt{p} \cdot V) \cdot \left(\sqrt{p} - 1\right).$$

Рисунок 2.22 – Результат виконання 2 етапу і всієї операції «усі-усім» для топології 2-D тор в режимі передачі повідомлень, $p = 9$

2.3.1.3 Оцінка трудомісткості комунікацій для топології «гіперкуб» при режимі передачі повідомлень

Для топології гіперкуб (рис. 2.9) час виконання одиночної операції пересилки даних дорівнює ($A = log_2 p$):

$$T^H_{p-p} = t_s + t_w \cdot V \cdot log_2 p. \quad (2.13)$$

Час виконання операції передачі даних від одного процесора всім іншим у топології гіперкуб становить:

$$T^H_{one-to-all} = (t_s + t_w \cdot V) \cdot log_2 p. \quad (2.14)$$

Для гіперкуба розсилка може бути виконана в ході N-етапної процедури передачі даних. На першому етапі процесор-джерело повідомлення передає дані одному зі своїх сусідів – в результаті після

першого етапу є два процесори, що мають копію даних, що пересилаються. На другому етапі два процесори, задіяні на першому етапі, пересилають повідомлення своїм сусідам по другій розмірності і так далі (рис. 2.23).

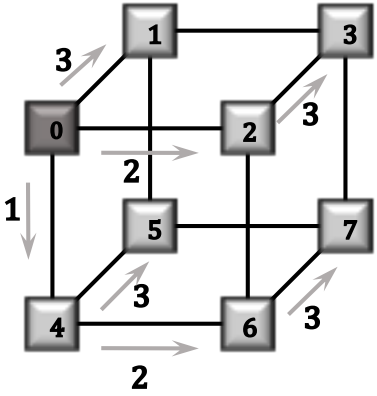


Рисунок 2.23 – Ілюстрація виконання операції «один-усім» для топології гіперкуб в режимі передачі повідомлень, $p = 8$

Алгоритм виконання множинної розсилки повідомлень для гіперкуба може бути отриманий шляхом узагальнення способу передачі даних "усі-усім" для топології сітка на розмірність гіперкуба $lp = log_2\, p$.

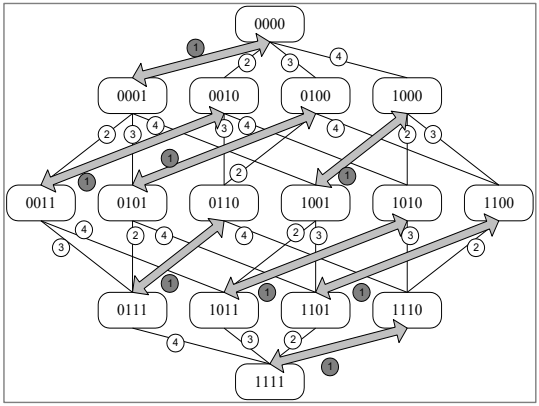
Кожному процесору ставиться у відповідність двійковий еквівалент його номера. Процесори, що мають безпосереднє з'єднання у гіперкубі, будуть мати номери, що відрізняються один від одного тільки одним розрядом. На кожному етапі $i, i = \overline{1,lp}$ алгоритму функціонують усі процесори мережі, які обмінюються даними зі своїми сусідами за i-тою розмірністю та формують об'єднані повідомлення:

$$T^{H}_{all-to-all} = \sum_{i=1}^{lp}\left(t_s + 2^{i-1} \cdot t_w \cdot V\right) = \tag{2.15}$$

$$= t_s \cdot log_2 p + t_w \cdot V \cdot (p-1).$$

Покрокове виконання колективної операції обміну даними за типом «all-to-all» на гіперкубі розміру $lp = 4$ із застосуванням найбільш поширеного методу покоординатної оптимальної маршрутизації наведено на рисунках 2.24-2.27. На рисунках показано процес формування об'єднаних повідомлень для процесору с номером 0, лінки двосторонні. На першому етапі усі процесори передають свої повідомлення усім процесорам мережі з номерами, що відрізняються першим справа або молодшим бітом: 0 обмінюється з 1 процесором, 2 з 3, 4 з 5, 8 з 9, 6 з 7, 10 з 11, 12 з 13, 14 з 15. Таким чином, по завершенні 1 етапу на кожному процесорі будуть «подвоєні» повідомлення. На другому етапі, знову кожен процесор буде обмінюватися своїми укрупненими повідомленнями з процесорами мережі з номерами, що відрізняються другим бітом або розрядом: 0 з 2, 1 з 3, 4 з 6, 8 з 10 тощо. Процес повторюється N раз за розміром гіперкубу.

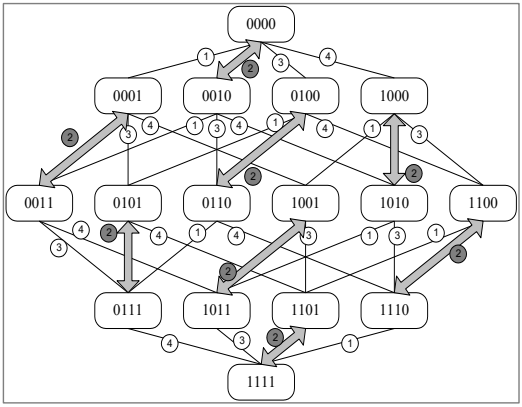
$$1 \text{ етап: } 0 \Leftrightarrow 1$$



$$0\text{: } 0000 + 0001$$

Рисунок 2.24 – Ілюстрація першого етапу виконання операції «all-to-all» на 4-х мірному гіперкубі, в режимі передачі повідомлень, $p = 16$

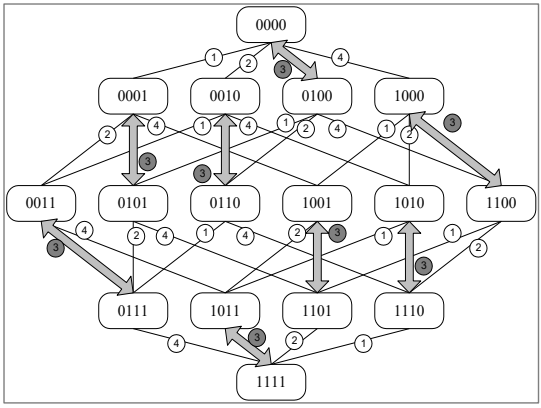
2 етап: $0 \Leftrightarrow 2$



$$0: 0000 + 0001$$
$$2: 0010 + 0011$$
$$\Downarrow$$
$$0: 0000 + 0001 + 0010 + 0011$$

Рисунок 2.25 – Ілюстрація 2 етапу виконання операції «all-to-all»

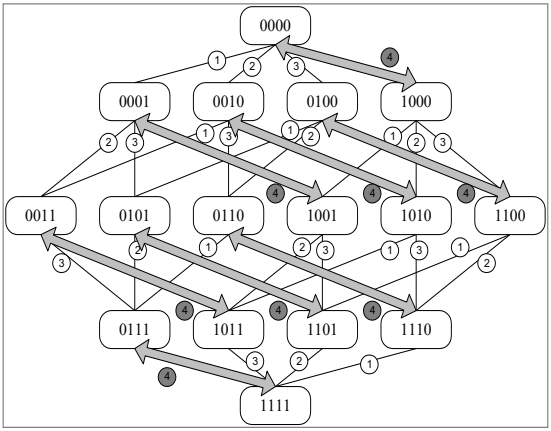
3 етап: $0 \Leftrightarrow 4$



$$0: 0000 + 0001 + 0010 + 0011$$
$$4: 0100 + 0101 + 0110 + 0111$$
$$\Downarrow$$
$$0: 0000 + 0001 + 0010 + 0011 + 0100 + 0101 + 0110 + 0111$$

Рисунок 2.26 – Ілюстрація 3 етапу виконання операції «all-to-all»

4 етап: $0 \Leftrightarrow 8$



$$0: 0000 + 0001 + 0010 + 0011 + 0100 + 0101 + 0110 + 0111$$
$$8: 1000 + 1001 + 1010 + 1011 + 1100 + 1101 + 1110 + 1111$$

$$\Downarrow$$

$$0: 0000 + 0001 + 0010 + 0011 + 0100 + 0101 + 0110 + 0111 + 1000 + 1001 + 1010 + 1011 + 1100 + 1101 + 1110 + 1111$$

Рисунок 2.27 – Ілюстрація 4 етапу виконання операції «all-to-all»

2.3.2 Оцінка трудомісткості комунікацій при режимі передачі пакетів

Трудомісткість комунікаційної операції «точка-точка» може бути отримана шляхом підстановки довжини максимального шляху в вираз для часу передачі даних, тобто у відповідну модель Хокні (2.3), при різних методах комунікації, в тому числі і для передачі пакетів.

В цьому сенсі, для топології кільце час для виконання поодинокої операції пересилання даних дорівнює:

$$T_{p-p}^{R} = t_s + t_w \cdot V + t_h \cdot \lfloor p/2 \rfloor. \quad (2.16)$$

Передача даних від одного процесора всім іншим процесорам мережі в режимі передачі пакетів для топології кільце може бути виконана шляхом логічного представлення кільцевої структури мережі у вигляді гіперкубу.

Алгоритм операції «один-усім» в цьому випадку (рис. 2.28):

1) на етапі розсилки процесор-джерело повідомлення передає дані процесору, що знаходиться на відстані $p/2$ від заданого процесору;

2) на другому етапі обидва процесори, що вже мають дані після першого етапу, передають повідомлення процесорам, що знаходяться на відстані $p/4$ від кожного з них і так далі.

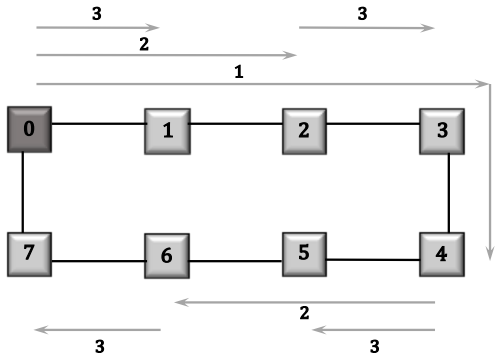


Рисунок 2.28 – Ілюстрація виконання операції «один-усім» для топології кільце в режимі передачі пакетів, $p = 8$

Трудомісткість здійснення операції розсилки при такому методі передачі даних визначається наступним співвідношенням:

$$T^R_{one-to-all} = \sum_{i=1}^{lp}(t_s + t_w \cdot V + t_h \cdot p/2^i) = \\ = (t_s + t_w \cdot V) \cdot log_2 p + t_h \cdot (p-1). \quad (2.17)$$

Застосування режиму передачі пакетів, як більш ефективного методу передачі даних, для кільцевої структури і топології типу сітка-

тор, не призводить до якого-небудь поліпшення часу виконання операції множинної розсилки «усі-усім».

Узагальнення алгоритмів виконання операції поодинокої розсилки на випадок множинної розсилки призводить до перевантаження каналів передачі даних (тобто для існування ситуацій, коли в один і той же момент часу для передачі по одній і тій лінії передачі є кілька пакетів даних, що очікують пересилання). Перевантаження каналів призводить до затримок при пересиланнях даних, що і не дозволяє проявитися всім перевагам методу передачі пакетів.

Для топології 2-D тор час для виконання поодинокої операції пересилання даних в режимі передачі пакетів дорівнює ($l = A$):

$$T_{p-p}^{M} = t_s + t_w \cdot V + 2t_h \cdot \left\lfloor \sqrt{p}/2 \right\rfloor. \quad (2.18)$$

Передача даних від одного процесора всім іншим процесорам мережі у режимі передачі пакетів для топології типу 2-D тор може бути отримана модифікацією способу передачі даних, застосованого для кільцевої структури мережі, відповідно до того ж способу узагальнення, що і в разі використання методу передачі повідомлень.

На рис. 2.29 наведено ілюстрацію процесу виконання операції «один-усім», процесор-джерело даних має номер 1, кількість процесорів $p = 16, \sqrt{p} = 4$. На першому кроці процесор, що пересилає повідомлення пересилає його процесору, що знаходіться від нього на відстані $\sqrt{p}/2 = 2$ по горизонталі, тобто процесорові з номером 3.

На другому кроці процесори, що вже мають повідомлення, пересилають його процесорам, що знаходяться від них на відстані $\sqrt{p}/4 = 1$ теж по горизонталі, це процесори 2 та 4. Загалом, всі процесори першого рядка сітки вже мають повідомлення. Далі, цей алгоритм виконання пересилок повторюється, але вже за вертикаллю.

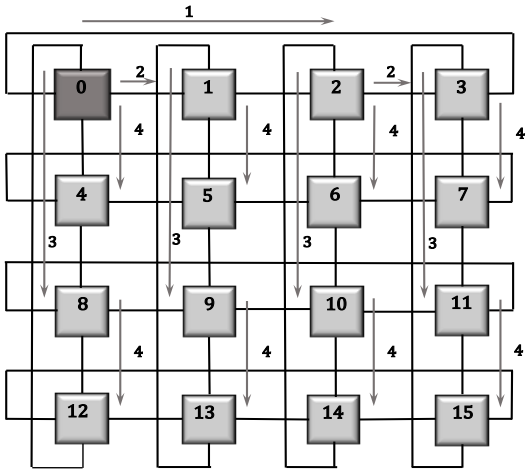


Рисунок 2.29 – Ілюстрація виконання операції «один-усім» для топології 2-D тор в режимі передачі пакетів, $p = 16$

Отриманий в результаті такого узагальнення алгоритм розсилки характеризується наступним співвідношенням для оцінки часу виконання операції «один-усім» для тору:

$$T^{M}_{one-to-all} = (t_s + t_w \cdot V) \cdot log_2 p + 2t_h \cdot \left(\sqrt{p} - 1\right). \quad (2.19)$$

Для топології гіперкуб трудомісткість поодинокої операції передачі даних для пакетів отримана аналогічно кільцю та тору і складає:

$$T^{H}_{p-p} = t_s + t_w \cdot V + t_h \cdot log_2 p. \quad (2.20)$$

Для гіперкуба алгоритм розсилки «усі-усім» (і, відповідно, трудомісткість виконання) при передачі пакетів не відрізняється від варіанту для методу передачі повідомлень. Але застосування режиму пакетної обробки, як і для інших розглянутих топологій, в цьому випадку не приводить до якого-небудь поліпшення часу виконання операції множинної розсилки.

2.4 Логічне представлення топології комунікаційного середовища

Важливим моментом при організації паралельних обчислень є можливість логічного представлення різноманітних топологій на основі наявних фізичних міжпроцесорних структур. Багато реальних чисельних алгоритмів допускають більш простий виклад при використанні цілком визначених мережевих топологій. Так, наприклад, блокові паралельні алгоритми матричного добутку ефективно відображаються на двовимірний тор, а алгоритми зі стрічковим розбиттям даних – на кільце, тощо. Таким чином, при переході на комутаційну мережу іншої топології необхідно мати гнучкий алгоритмічний/програмний механізм логічного відображення однієї топології на іншу. Математичним апаратом, здатним вирішити поставлене завдання, є так звані, *бінарні рефлексивні коди Грея* (*binary reflected Gray code*) [2-3, 25].

Як відомо, з'єднання процесорів в комунікаційну мережу по типу гіперкуб одне з найбільш використовуваних топологічних рішень в реальних паралельних архітектурах, тому далі наведено механізми логічного відображення її у 2-D тор та кільце.

Розглядається паралельна обчислювальна система з p процесорами і топологією гіперкуб розмірності N. Побудова відображення паралельних методів, що повністю використовують внутрішній паралелізм матричних операцій, на топологію гіперкуб не є тривіальним завданням, тому виникає необхідність матричного подання мережі гіперкуб.

Гіперкуб розглядається як сітка, стовпці і рядки якої формуються підкубами або гіперкубами меншого розміру. Така матрична візуалізація разом з серією властивостей, що випливають з рекурсивного визначення гіперкуба, становить корисний інструмент конструювання паралельних алгоритмів для комп'ютерів описаної архітектури.

Гіперкуб $H(N)$ розмірності N містить $p = 2^N$ вузлів. Кожен вузол ідентифікується бінарним кодом довжиною у N біт. Вузол i є суміжним всім вузлам, код яких відрізняється від коду i одним розрядом. Кожен вузол гіперкубу є суміжним $N = log_2 p$ іншим вузлам.

Позначимо значення коду Грея, як: $G(i, N)$, де $i -$ порядковий номер значення в коді, $N -$ його довжина.

Тоді, $G(0,1) = 0,\ G(1,1) = 1 -$ список кодів довжини 1.

Алгоритм генерації двійкового коду Грея довжини $k + 1$ полягає у виконанні наступної примітивно рекурсивної процедури:

1) розмістити 0 перед кожним кодом в k-списку кодів Грея;

2) розмістити 1 перед кожним кодом в реверсному k-списку;

3) записати послідовно перший і другий отримані списки, отримаємо код Грея довжини $k + 1$.

Логічне відображення двовимірної замкнутої сітки на гіперкуб виконується за правилом: процесорному елементу сітки з координатами (i, j) буде відповідати процесор гіперкубу з номером $G(i, p)||\ G(j, p)$, де $||$ – операція конкатенація кодів Грея.

Наприклад, рисунок 2.30 ілюструє відображення 2D-тора або замкнутої сітки розміром $\sqrt{p} \times \sqrt{p} = 4 \times 4$ на чотирьохвимірний гіперкуб $H(4)$.

Зауважимо, що в результаті виконання такого відображення процесори, які є "безпосередніми сусідами" в топології тор, матимуть ті ж властивості в топології гіперкуб. Так, процесор з номером <0,0> у торі має наступних «сусідів» – <0,1>, <1,0>, <0,3>, <3,0>. Відповідно, процесор гіперкубу з номером 0000=#<0,0> безпосередньо зв'язаний з процесорами 0001=#<0,1>, 0100=#<1,0>, 0010=#<0,3> та #<3,0>=1000.

Таким чином, всі паралельні алгоритми, розроблені для замкнутої сітки або двовимірного тору, тільки використанням логічного

перетворення, можуть бути перенесені без зміни алгоритму на комунікаційну мережу гіперкуб.

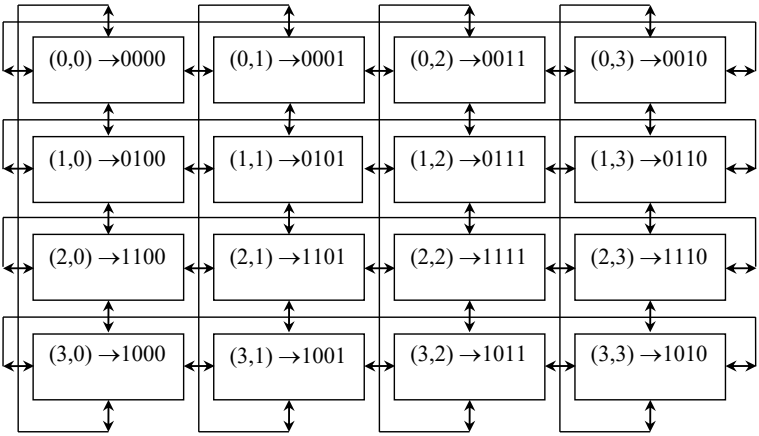


Рисунок 2.30 – Схема логічного відображення топології 2D-тор розмірності 4 × 4 на топологію гіперкуб

Встановлення відповідності між кільцевою топологією і гіперкубом також виконується за допомогою кодів Грею на основі наступних перетворень:

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

$$G(i, s+1) = \begin{cases} G(i,s), i < 2^s, \\ 2^s + G(2^{s+1} - 1 - i, s), i \geq 2^s. \end{cases} \quad (2.21)$$

Наприклад, у таблиці 2.2 наведено побудоване відображення кільця з 8 процесорів на гіперкуб $H(3)$.

Зауваження: важливою властивістю кодів Грея є той факт, що сусідні значення $G(i,N)$ і $G(i+1,N)$ розрізняються тільки однією бітовою позицією. Як результат, аналогічно попередньому випадку усі «сусіди» у кільці є «сусідами» у гіперкубі.

Таблиця 2.2 – Відображення кільця на гіперкуб $H(3)$

Код Грея N=1	Код Грея N=2	Код Грея N=3	Номера процесорів	
			гіперкуб	кільце
0	00	000	0	0
1	01	001	1	1
	11		3	2
	10	010	2	3
		110	6	4
		111	7	5
		101	5	6
		100	4	7

Таким чином, огляд властивостей типових топологій з'єднання процесорів та їх числових характеристик, введення комунікаційних примітивів операцій передачі даних для топологій, розробка логічного представлення топологій комунікаційного середовища, дають підстави проводити теоретичний порівняльний аналіз виконання різних паралельних алгоритмів на основі параметрів, характерних для реальних високопродуктивних багатопроцесорних систем та розробляти ефективне паралельне алгоритмічне і програмне забезпечення.

2.5 Завдання для самостійної роботи

1. Розробити алгоритми виконання основних операцій передачі даних для топології мережі у вигляді 3-мірної сітки в режимах передачі повідомлень та пакетів. Проілюструвати їх схемами поетапного виконання.

2. Розробити алгоритми виконання основних операцій передачі даних для топології мережі у вигляді двійкового дерева в режимах передачі повідомлень та пакетів. Проілюструвати їх схемами поетапного виконання.

3. Розробити алгоритми виконання основних операцій передачі даних для топології мережі у вигляді товстого дерева в режимах передачі

повідомлень та пакетів. Проілюструвати їх схемами поетапного виконання.

4. Розробити алгоритми виконання основних операцій передачі даних для топології зірка в режимах передачі повідомлень та пакетів. Проілюструвати їх схемами поетапного виконання.

5. Розробити алгоритми виконання основних операцій передачі даних для топології мережі у вигляді повний граф в режимах передачі повідомлень та пакетів. Проілюструвати їх схемами поетапного виконання.

6. Отримати аналітичні вирази для оцінки тимчасової складності операцій передачі даних («точка-точка, «один-усім», «усі-усім») для топології 3-вимірної сітки в умовах лінійної та уточненої моделі Хокні.

7. Отримати аналітичні вирази для оцінки тимчасової складності операцій передачі даних («точка-точка, «один-усім», «усі-усім») для топології «повне двійкове дерево» в умовах лінійної та уточненої моделі Хокні.

8. Отримати аналітичні вирази для оцінки тимчасової складності операцій передачі даних («точка-точка, «один-усім», «усі-усім») для топології «зірка» в умовах лінійної та уточненої моделі Хокні.

9. Отримати аналітичні вирази для оцінки тимчасової складності операцій передачі даних («точка-точка, «один-усім», «усі-усім») для топології «кліка» в умовах лінійної та уточненої моделі Хокні.

10. Розробити алгоритми логічного представлення двійкового дерева для різних фізичних топологій мережі.

2.6 Контрольні питання

1. Які основні характеристики використовуються для оцінки топології мережі передачі даних?

2. Наведіть значення характеристик для конкретних типів комунікаційних структур (повний граф, лінійка, сітка тощо).

3. Які основні методи застосовуються при маршрутизації даних, що передаються за мережею? Що таке детерміновані та оптимальні маршрутизатори? Що таке метод покоординатної маршрутизації?

4. В чому полягають основні методи передачі даних? Наведіть для цих методів аналітичні оцінки часу виконання?

5. Які операції передачі даних може бути виділено в якості основних? Що таке операція «точка-точка»/«point-point», «один-усім»/«one-to-all», «усі-усім»/«all-to-all»?

6. Записати лінійну модель Хокні та її модифікації. Що таке латентність, час на передачу слова та службових даних?

7. У чому полягають алгоритми виконання передачі даних від одного процесора іншому процесорові мережі для топологій кільця, сітки та гіперкуба? Наведіть оцінки тимчасової трудомісткості для цих алгоритмів.

8. У чому полягають алгоритми виконання передачі даних від одного процесора всім процесорам мережі для топологій кільця, решітки та гіперкуба? Наведіть оцінки тимчасової трудомісткості для цих алгоритмів.

9. У чому полягають алгоритми виконання передачі даних від всіх процесорів всім процесорам мережі для топологій кільця, решітки та гіперкуба? Наведіть оцінки тимчасової трудомісткості для цих алгоритмів.

10. У чому полягає корисність використання логічних топологій? Наведіть приклади алгоритмів логічного представлення структури комунікаційної мережі.

РОЗДІЛ 3
РОЗРОБКА ТА АНАЛІЗ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ ПАРАЛЕЛЬНИХ АЛГОРИТМІВ

В даному розділі наведено методи розробки і опису паралельних алгоритмів, а також метричні характеристики високопродуктивних обчислень у системах з розподіленою пам’яттю.

3.1 Ієрархічна декомпозиційна технологія розробки паралельного алгоритму

Розвиненим і загальновизнаним математичним апаратом розробки паралельних алгоритмів є графові моделі: графи впливу, залежностей, тобто деякі інформаційні графи алгоритмів [1-4]. Однак для реальних задач застосування цієї моделі пов'язане з великими труднощами. Для прямого опису всіх вершин і дуг відповідного графу необхідні великі витрати комп'ютерної пам'яті, а для аналізу використовуються алгоритми, які практично всі мають високу обчислювальну складність, найчастіше експоненційну. Тому для розробки паралельних алгоритмів задач практичного рівня складності використовується багатоетапний процес, що починається аналізом задачі, вибором моделі обчислень, декомпозицією задачі на незалежні паралельні процеси та закінчується питаннями дослідження ефективності. Така технологія розпаралелювання отримала назву – *ієрархічної декомпозиційної методики* [2-3, 5].

Спочатку обчислювальна задача розбивається на підзадачі, аналізуються інформаційні взаємозв'язки між ними, визначається множина макрооперацій, оцінюється ефективність потенційного та реального паралелізму. У разі незадовільного результату на будь-якому з етапів схема розбиття змінюється, та весь процес повторюється з початку

до тих пір, поки характеристики отриманого паралельного алгоритму не стануть задовільними або шляхом повного перебору не стане ясно, що домогтися якісного розпаралелювання неможливо (рис. 3.1).

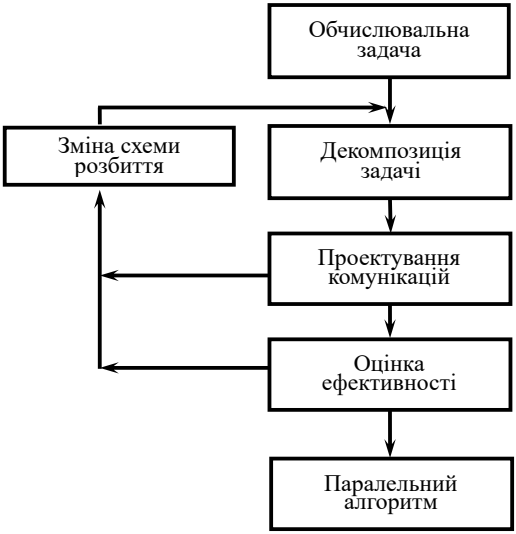


Рисунок 3.1 – Схема виконання ієрархічної декомпозиційної методики

Графові моделі використовуються для побудови паралельного алгоритму як на верхньому рівні для аналізу взаємодії підзадач, так і на нижньому рівні для розпаралелювання окремих макроопераций [1-4].

Графи впливу [1] – це спеціального виду орієнтовані графи.

У графі впливу вершинам зіставляються операції алгоритму, що виконуються. Вершини з'єднуються дугами, тоді і тільки тоді, коли результат виконання однієї операції впливає на результат суміжної. За визначенням операція x впливає на операцію y, якщо зміною значення параметру, який обчислює операція x можна змінити значення

параметру, який обчислює операція y. За вказаним правилом будується орієнтований граф, причому операції алгоритму вважаються вершинами графу. Дугами з'єднують кожну з пар вершин, в якій одна з відповідних їм операцій впливає на іншу.

Наприклад, наведемо графи впливу для операції множення матриці $G:\{g_{ij}\}, i,j=\overline{1,m}$ на вектор $X:\{x_i, i=1,m\}$, $\sum_{j=1}^{m} x_j \times g_{ij}, i=\overline{1,m}$. На рисунках 3.2 і 3.3 зображено графи впливу алгоритмів здвоювання та модифікованого каскадного підсумування [1-3].

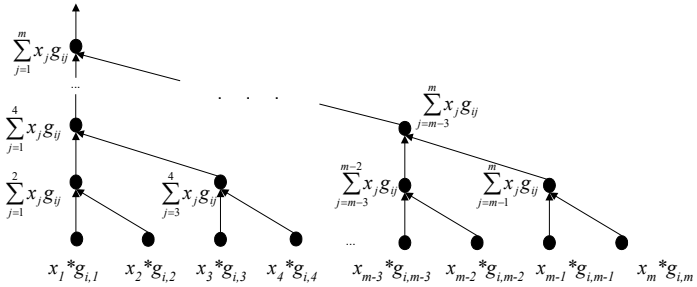


Рисунок 3.2 – Граф впливу алгоритму здвоювання при множенні матриці на вектор

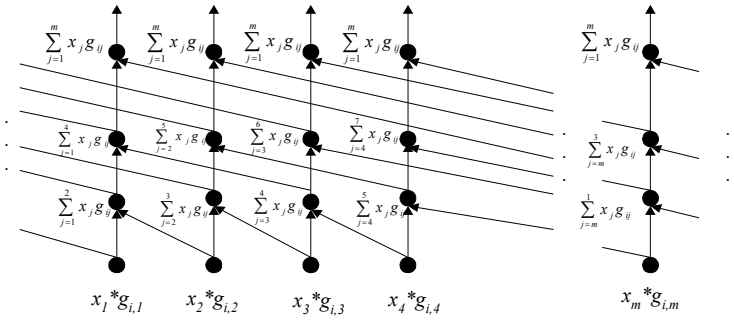


Рисунок 3.3 – Модифікований граф впливу алгоритму здвоювання при множенні матриці на вектор (каскадна схема)

При застосуванні методики виконується введення макрооперацій, що призводить:

- до більш компактного подання графу алгоритму;
- значного спрощення проблеми вибору ефективних способів розпаралелювання обчислень;
- використання типових паралельних методів виконання макрооперацій, як конструктивних елементів при розробці паралельних способів вирішення більш складних обчислювальних задач.

Процес введення макрооперацій здійснюється поетапно з послідовно зростаючим рівнем деталізації операцій, що використовуються.

Існує 2 принципово різних способів розбиття процесу обчислень на незалежні частини: функціональне розбиття та розбиття за даними. Функціональне розбиття, в першу чергу, вимагає поділу процесу обчислень на окремі процедури і тільки після цього розбиття даних в залежності від виділених незалежних процедур обробки. Розбиття за даними має на увазі розподіл оброблюваних даних на блоки, найчастіше на основі геометричного розбиття області рішення задачі, а потім, в залежності від способу розбиття даних, декомпозицію процедур обробки розподіленої інформації.

Незалежно від обраного способу розбиття процесу на частини, наступним етапом побудови паралельного алгоритму, є визначення інформаційних потоків. Розрізняють такі типи інформаційної взаємодії паралельно виконуваних процесів: локальна і глобальна взаємодія за кількістю взаємодіючих процесів, структурна і взаємодія "нетипового" виду за схемами комунікаційних топологій.

Організація взаємодії процесів, що призводить до формування цілком певних, типових схем комунікації (кільце, тор, гіперкуб, зірка або

дерево) – є структурною. Неструктурна взаємодія виникає, коли схема виконуваних операцій передач даних має граф "нетипового" вигляду.

Залежно від способу реалізації міжпроцесорних обмінів розрізняють синхронний і асинхронний паралелізм (модель паралельної ОС типу SIMD і типу MIMD).

Останніми етапами побудови паралельного алгоритму є:

- розподіл обчислювального навантаження по процесорам паралельної комп'ютерної системи; рішення задачі балансування, є NP-повною проблемою;
- побудова відображення паралельного алгоритму на реальну архітектуру ПОС.

Треба відзначити, що управління розподілом навантаження для процесорів можливе тільки для обчислювальних систем з розподіленою пам'яттю, для систем зі спільною пам'яттю розподіл навантаження зазвичай виконується операційною системою автоматично.

Приклади відображення паралельних алгоритмів на ОС із заданою топологією схематично показано на рис. 3.4-3.5 (топології кільце та 2-D тор).

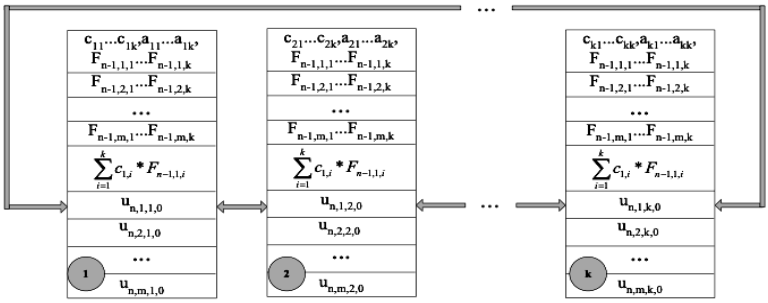


Рисунок 3.4 – Приклад відображення паралельного алгоритму на ОС з топологією кільце

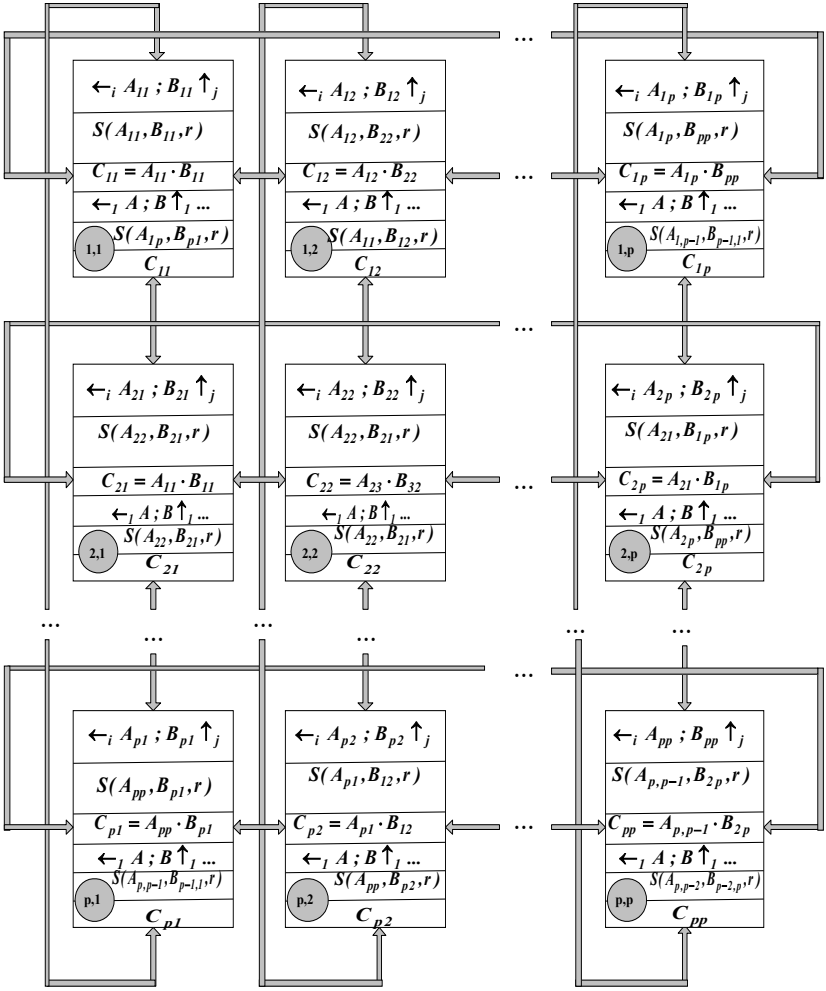


Рисунок 3.5 – Приклад відображення паралельного алгоритму на ОС з топологією 2-D замкнута сітка

Таким чином, розглянуто наступні етапи розробки паралельних алгоритмів із використанням поетапної декомпозиційної технології:

1) аналіз задачі та виявлення її потенційного паралелізму;

2) вибір моделі програмування та схеми розпаралелювання;

3) розподіл процесу обчислень на частини, які можуть бути виконані одночасно та визначення необхідних інформаційних взаємодій для паралельних процесів обробки даних;

4) визначення необхідних інформаційних взаємодій для виділених паралельних процесів обробки даних;

5) розподіл сформованого набору процесів обробки даних по процесорам (відображення паралельної схеми виконання завдання на архітектуру комп'ютерної обчислювальної системи).

Існує і альтернативний підхід для розбиття вхідної обчислювальної задачі на незалежні частини – агрегаційний. Якщо декомпозиційний підхід – це розбиття "зверху-вниз", то агрегаційний – "знизу-вгору" [5].

Вибір декомпозиційної методики обумовлений перевагами цього підходу, а саме: взаємозалежністю та ітеративністю етапів побудови паралельного алгоритму, можливістю забезпечити необхідний рівень масштабованості обчислень за рахунок варіювання детальності декомпозиції. Даний підхід найчастіше використовується для обчислювальних систем з розподіленою пам'яттю і моделлю передачі повідомлень. Однак немає ніяких обмежень для застосування описаної методики в системах із загальною пам'яттю, якщо для забезпечення інформаційної взаємодії використовується доступ до поділюваного адресного простору.

3.2 Високопродуктивні обчислення та динамічні характеристики паралельних алгоритмів

Метрики, динамічні характеристики паралельних обчислень – це система показників, що дозволяє оцінювати переваги, отримані при паралельному розв’язанні завдань на p процесорах, у порівнянні з послідовним вирішенням тих же задач на одному процесорі [1-4].

З іншого боку, вони дозволяють судити про обґрунтованість застосування даного числа процесорів для вирішення конкретних завдань або визначають, якого розміру задачу можна ефективно розв'язати на даному паралельному комп'ютері тощо.

Базисом для визначення метрик паралелізму є наступні характеристики послідовних і паралельних обчислень/систем та задачі:

1) $m -$ розмір входу або розмірність вирішуваної задачі;

2) $p -$ кількість процесорів, що використовуються для організації паралельних обчислень;

3) $O_p -$ обсяг обчислень, що виражається через кількість операцій, виконуваних p процесорами у ході вирішення завдань;

4) $T_p -$ загальний час обчислення із використанням p процесорів.

Розмір входу, m – це розмірність вхідних даних для завдання, яке розв'язується. Цей параметр може бути або очевидним чином визначеним, як для:

- цілочисельних алгоритмів – абсолютне значення скаляра;
- одновимірних масивів – максимальна розмірність масиву;
- матриць –максимальна розмірність матриці;
- систем рівнянь – розмірність системи;

або невизначеним зовсім, як для одного рівняння.

Наведемо загальноприйняті і найбільш використовувані в паралельних обчисленнях показники та їх позначення:

1) T_1 – час, необхідний для розв'язання задачі певного розміру на одному процесорі за допомогою найкращого послідовного алгоритму;

2) $T_{p,comp}$ – час для розв'язання задачі певного розміру з використанням паралельного алгоритму на комп'ютері з p процесорів без обліку обмінних операцій (час реалізації обчислень);

3) $T_{p,comm}$ – комунікаційна трудомісткість/складність алгоритму або час виконання операцій міжпроцесорного обміну;

4) T_p – загальний час реалізації паралельного алгоритму на паралельній архітектурі: $T_p = T_{p,comp} + T_{p,comm}$ (іноді містить ще один доданок: $T_{p,idle}$ – загальний час непродуктивні витрати;

5) Z – доля часу операцій обміну в загальному часі виконання паралельного алгоритму:

$Z = T_{p,comm}/T_p$;

6) Dop – ступінь паралелізму або максимальна кількість процесорів, що можуть бути використані при виконанні паралельного алгоритму.

У однопроцесорній системі $T_1 = O_1$. У загальному випадку, $T_p < O_p$, якщо в одиницю часу p процесорами виконується більш ніж одна команда, де $p > 2$. Останнє співвідношення формулює твердження: час обчислень можна скоротити за рахунок розподілу обсягу обчислень за декількома процесорами.

На сьогоднішній день величезна кількість робіт [1-4, 16-17, 26-33] присвячена аналізу існуючих і введенню нових динамічних характеристик, що визначають якість паралельного алгоритму. Загалом можна виділити чотири групи існуючих метрик (рис. 3.6).

Перша характеризує швидкість обчислень:

- індекс паралелізму: $PI_p = O_p/T_p$;
- прискорення: $S_p = T_1 / T_p$.

Другу групу утворюють метрики, що дають можливість мати уявлення про ефективність залучення до вирішення завдання додаткових процесорів:

- ефективність: $E_p = S_p/p = T_1 / (p \times T_p)$;
- утилізація: $U_p = R_p \times E_p = O_p/ (p \times T_p)$.

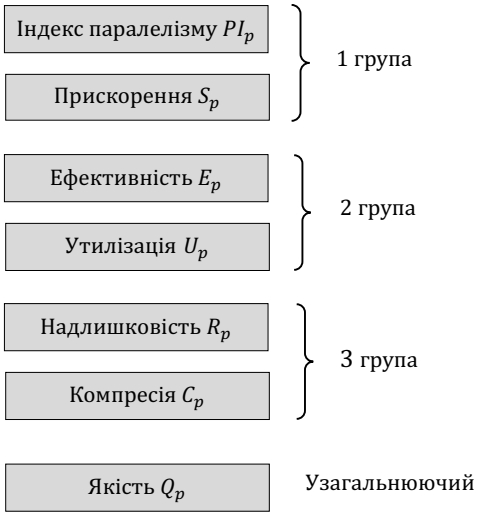


Рисунок 3.6 – Групи метричних характеристик
для паралельних обчислень

Третя група метрик характеризує ефективність паралельних обчислень шляхом порівняння обсягів обчислень, що отримані при паралельному і послідовному виконанні завдання.

- надлишковість: $R_p = O_p / O_1$;
- компресія: $C_p = O_1 / O_p$.

Четверту групу утворює узагальнена метрика якість паралелізму: $Q_p = S_p \times E_p \times C_p$. Оскільки ця метрика пов'язує метрики прискорення, ефективність і стиснення, вона є більш об'єктивним показником поліпшення продуктивності за рахунок паралельних обчислень.

Індекс паралелізму (*parallel index)* характеризує середню швидкість паралельних обчислень через кількість виконаних операцій:

$$PI_p = O_p / T_p. \quad (3.1)$$

Коефіцієнт прискорення (speedup) – одна з найбільш важливих динамічних характеристик паралельного алгоритму. Коефіцієнт прискорення визначається, як відношення часу розв’язання задачі на однопроцесорній обчислювальної системі до часу виконання паралельного алгоритму. Розрізняють потенційний і реальний паралелізм, а, отже, і відповідні коефіцієнти прискорення.

Коефіцієнт потенційного прискорення враховує тільки внутрішній паралелізм методу розв'язання задачі без урахування операцій обміну та інших накладних витрат:

$$S_{pot} = T_1 / T_{p,comp},\ S_p = T_1 / T_p. \quad (3.2)$$

Наступною важливою характеристикою паралельного алгоритму є коефіцієнт ефективності (*efficiency*) [2-4], що визначає середню частку часу виконання алгоритму, протягом якого процесори реально використовуються для розв'язання задачі, тобто якість завантаження обладнання:

$$E_p = S_p/p = T_1 / (p \times T_p). \quad (3.3)$$

В ідеальному випадку досягається лінійне прискорення й одинична ефективність: $1 \le S \le p,\ 1/p \le E \le 1$, за певних обставин може спостерігатися явище надлінійного прискорення: $S > p$.

В якості причин існування *надлінійного прискорення* називають наступні [2, 9]:

1) не ідентичні умови виконання послідовної і паралельної програм (наприклад, недолік оперативної пам'яті в однопроцесорній системі);

2) нелінійний характер залежності складності рішення задачі від обсягу оброблюваних даних (деякі методи сортування);

3) відмінність обчислювальних схем послідовного і паралельного методів.

Утилізація (*utilization*) враховує внесок кожного процесора при паралельному обчисленні у вигляді кількості операцій, виконаних процесором в одиницю часу.

Коефіцієнт корисного використання або утилізації:

$$U_p = O_p / (p \times T_p). \quad (3.4)$$

Утилізація може бути виражена через метрики надмірності і ефективності:

$$U_p = R_p \times E_p, \quad (3.5)$$

з урахуванням того, що виконується співвідношення $T_1 = O_1$.

Надлишковість (*redundancy*) – це відношення обсягу паралельних обчислень до обсягу еквівалентних послідовних обчислень:

$$R_p = O_p / O_1. \quad (3.6)$$

Важливість даної метрики в тому, що вона виходить не з відносних показників прискорення і ефективності, отриманих з часу обчислень, а з абсолютних показників, які враховують обсяг виконаної обчислювальної роботи. Надлишковість відображає ступінь відповідності між програмним і апаратним паралелізмом.

Компресія (compression) обчислюється як величина, зворотна надмірності:

$$C_p = O_1 / O_p. \quad (3.7)$$

Метрика *якість* (*quality*) об'єднує прискорення, ефективність і надлишковість і визначається наступним чином:

$$Q_p = S_p \times E_p \times C_p. \quad (3.8)$$

Ціна або *вартість* (*cost*) обчислень визначається, як добуток часу виконання алгоритму і числа використаних процесорів.

Виходячи з викладеного, ціна послідовних обчислень дорівнює часу їх виконання: $Cost_1 = 1 \cdot T_1 = T_1$, а ціна паралельних обчислень становить: $Cost_p = p \cdot T_p$.

З іншого боку, $T_p = T_1/S_p$, тому ціна паралельних обчислень може бути обчислена в такий спосіб: $Cost_p = p \cdot T_p = p \cdot T_1/S_p = T_1/E_p$. Алгоритми, вартість яких пропорційна часу виконання найкращого послідовного алгоритму ≡ вартості, позиціонуються, як *вартісно-оптимальні*.

3.3 Оцінка максимально досяжного паралелізму

Оцінки якості паралельних обчислень припускають знання найкращих, максимально можливих значень показників прискорення і ефективності. Одним з чинників, що перешкоджають досягненню гранично максимальному значенню прискорення, є наявність в алгоритмі послідовної частини, яка не може бути розпаралеленою. При розгляді наступних закономірностей не враховується вплив комунікаційної складової алгоритму і інших непродуктивних витрат [1-4, 26-27].

В ідеальному випадку система з p процесорів могла б прискорити обчислення в p раз. В реальності досягти такого показника за рядом причин не вдається. Головна з них полягає в неможливості повного розпаралелювання ні одного із завдань. Як правило, в кожному алгоритмі є такий фрагмент, який принципово повинен виконуватися послідовно і тільки одним з процесорів. Деякі проблеми виникають і з тією частиною завдання, яка піддається розпаралелюванню. Тут ідеальним був би варіант, коли паралельні гілки алгоритму постійно завантажували б усі процесори системи, причому так, щоб навантаження на кожен процесор було б однаковим. Обидві ці умови на практиці важко реалізувати.

Таким чином, орієнтуючись на паралельну обчислювальну систему необхідно чітко усвідомлювати, що домогтися прямо пропорційного числу процесорів збільшення продуктивності не вдасться, і, природньо, постає питання про те, на яке реальне прискорення можна розраховувати.

Відповідь залежить від того, яким чином використовувати обчислювальні потужності ОС, які зростають в результаті збільшення числа процесорів.

Найбільш характерними є три варіанти:

- обсяг обчислень не змінюється, а головна мета розширення ОС – скоротити час обчислень, досяжне в цьому випадку прискорення визначається законом Амдала;
- час обчислень з розширенням системи не змінюється, але при цьому збільшується обсяг розв'язуваної задачі, мета такого підходу – за заданий час виконати максимальний обсяг обчислень, цю ситуацію характеризує закон Густавсона;
- даний варіант схожий на попередній, але з однією умовою: збільшення обсягу розв'язуваної задачі обмежене ємністю доступної пам'яті, прискорення в такому формулюванні визначає закон Сана і Ная.

3.3.1 Закон Амдала про обмеження прискорення паралельних обчислень

Джин Амдал (Gene Amdahl) – один з розробників всесвітньо відомої системи IBM 360 в 1967 році запропонував формулу, яка відображає залежність прискорення обчислень, що досягається на багатопроцесорній ОС, як від числа процесорів, так і від співвідношення між послідовною і розпаралеленою частинами алгоритму [26].

Проблема розглядалася Амдалом виходячи з положення, що обсяг розв'язуваної задачі (робоче навантаження – число виконуваних операцій) зі зміною числа процесорів, які беруть участь в її вирішенні, залишається незмінним. Така постановка характерна для випадків, коли основною вимогою є швидкість обчислень, наприклад, для завдань, які виконуються в реальному часі.

Нехай маємо деякий алгоритм розв'язання реальної задачі і нехай f – частка послідовних обчислень у розглядаємому алгоритмі. Тоді, відповідно, частка алгоритму, яка може бути розподілена між процесорами, дорівнює: $1 - f$.

При перенесенні обчислень на паралельну машину час обчислень розподілиться таким чином:

1) $f \cdot T_1$ – час виконання тієї частини алгоритму, яку не можна розпаралелити;

2) $(1 - f) \cdot T_1/p$ – час, який буде витрачено на розпаралелену частину алгоритму.

Загальний час на реалізацію алгоритму на паралельній машині з p процесорами, буде визначатися за формулою:

$$T_p = f \cdot T_1 + (1 - f) \cdot T_1/p. \tag{3.9}$$

Отже, коефіцієнт прискорення приймає наступний вигляд:

$$S_p = T_1/T_p = p/(fp + 1 - f), \tag{3.10}$$

$$S_p = 1/(f + (1 - f)/p). \tag{3.11}$$

Ця формула отримала назву закону Амдала про обмеження прискорення паралельних обчислень. Закон Амдала пов'язує прискорення з часткою операцій, які виконуються послідовно і служить для оцінки максимально досяжного прискорення. Закон визначає, що навіть, якщо послідовна частка обчислень мала, максимальне прискорення паралельного алгоритму для необмеженого числа процесорів не перевищує $1/f$.

Для істотного збільшення коефіцієнта прискорення необхідно мінімізувати ту частину операцій f, яка виконується послідовно: $f \ll 1$. Однак, навіть в цьому випадку, величина $f \times p$ може бути досить значною при великому числі процесорів.

При $f \ll 1$ закон Амдала має вигляд:

$$S_p = p/(1 + f \cdot p). \quad (3.12)$$

Якщо частка послідовних обчислень є великою (близька до одиниці), то збільшення числа процесорних елементів не призводить до істотного прискорення процесу виконання завдання.

При заданому значенні f величина S_p наближається до свого асимптотичного значення, наближено рівного $1 / f$, тому існує деяке критичне значення числа процесорів після якого нарощування числа процесорів не призводить до збільшення коефіцієнту прискорення.

Відповідно до закону Амдала, прискорення процесу обчислень при використанні p процесорів обмежується величиною: $S_p \ll \frac{1}{f+(1-f)/p}$, однак, часто частка послідовних обчислень може бути істотно знижена при виборі найбільш підхожих для розпаралелювання алгоритмів (рис. 3.7-3.8).

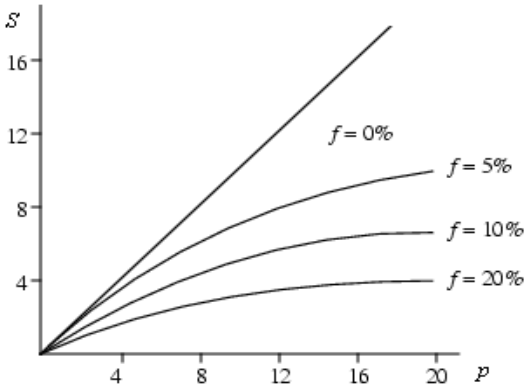


Рисунок 3.7 – Графіки залежності коефіцієнта прискорення від кількості процесорів

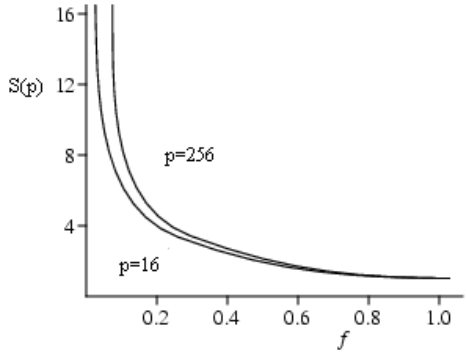


Рисунок 3.8 – Графіки залежності коефіцієнта прискорення від величини частки послідовних обчислень алгоритму

Розгляд закону Амдала відбувається в припущенні, що частка послідовних розрахунків f є постійною величиною і не залежить від параметра m, що визначає обчислювальну складність розв'язуваної задачі. Для великого ряду завдань частка $f = f(m)$ є спадною функцією від m, і в цьому випадку прискорення для фіксованого числа процесорів може бути збільшено за рахунок збільшення обчислювальної складності розв'язуваної задачі. В цьому випадку, прискорення $S_p = S_p(m)$ є зростаючою функцією від параметра m. Дане твердження часто іменується як ефект Амдала.

Закон Амдала характеризує одну з найсерйозніших проблем в області паралельного програмування, бо алгоритмів без певної частки послідовних команд практично не існує. Однак часто частка послідовних дій характеризує не можливість паралельного вирішення завдань, а послідовні властивості застосованих алгоритмів. Як результат, частка послідовних обчислень може бути істотно знижена при виборі більш підходящих для розпаралелювання методів.

3.3.2 Закон Густавсона-Барсіса про масштабування прискорення

Джон Густавсон із NASA Ames Research прийшов до висновку, що нарощування загального обсягу програми стосується в основному розпаралеленої частини [27]. Отже, прискорення можна підвищити завдяки тому, що, залишаючись практично незмінною, послідовна частина в загальному обсязі збільшеної програми має вже меншу питому вагу.

Вводиться величина частки послідовних розрахунків в паралельних обчисленнях, яка в оригінальній нотації має наступний вигляд:

$$g = \tau(m)/(\tau(m) + \pi(m)), \tag{3.13}$$

де $\tau(m)$ і $\pi(m)$ – час, відповідно, на послідовну і паралельну частини виконуваних обчислень.

Оскільки $T_1 = \tau(m) + \pi(m)$, $T_p = \tau(m) + \pi(m)/p$, тоді коефіцієнт прискорення набуває вигляду:

$$S_p = \frac{T_1}{T_p} = \frac{\tau(m)+\pi(m)}{\tau(m)+\frac{\pi(m)}{p}} = \frac{(\tau(m)+\frac{\pi(m)}{p})(g+(1-g)p)}{\tau(m)+\frac{\pi(m)}{p}}. \tag{3.14}$$

Тоді, для оцінки можливості прискорення, яке може бути отримано на ОС з p процесорів, коли обсяг обчислень збільшується зі зростанням кількості процесорів (при сталості загального часу обчислень), використовується вираз, відомий як закон Густавсона або закон Густавсона-Барсіса).

Отже, спрощуючи отриману оцінку, маємо:

$$S_p = \frac{T_1}{T_p} = \ g + (1-g)p = p + (1-p)g. \tag{3.15}$$

Таким чином, закон Густавсона характеризує ситуацію, при якій час обчислень з розширенням системи не змінюється, але збільшується обсяг розв'язуваної задачі. Мета такого підходу – за заданий час виконати

максимальний обсяг обчислень. Закон Густавсона не суперечить закону Амдала. Різниця полягає лише в формі утилізації додаткової потужності ОС, що виникає при збільшенні числа процесорів.

Оцінка прискорення, що отримується відповідно до закону Густавсона-Барсіса, отримала назву прискорення масштабування (*scaled speedup*). Дана характеристика може показати, наскільки ефективно можуть бути організовані паралельні обчислення при збільшенні складності розв'язуваної задачі.

3.4 Масштабованість паралельних алгоритмів та основи ізоефективного аналізу

У зв’язку з інтенсивним розвитком паралельної комп’ютерної індустрії масштабованість на сьогодні стає одною з найбільш важливих характеристик при проектуванні паралельних алгоритмів та архітектур. Терміни “масштабованість” та “масштабований” широко і досить часто використовуються при дослідженнях стосовно паралельної обробки інформації [2-4, 28-33]. Проте не існує досить адекватного, загально-прийнятого визначення цього терміну. Для вимірювання та порівняння масштабованості комп’ютерних систем і програм важко отримати не тільки кількісні характеристики, але й якісні оцінки. Тому введення формального визначення та теоретичного обґрунтування метрик масштабованості для високопродуктивних обчислень є актуальною і перспективною науково-практичною задачею. Не менш важливими являються завдання розробки практичного інструментарію для оцінки масштабованості систем та порівняння цієї нової оцінки якості паралельних обчислень з існуючими динамічними характеристиками такими, як прискорення та ефективність.

Розробка паралельних програм в якості своєї мети може мати не тільки зменшення часу виконання, але й забезпечення можливості

вирішення задач більшої розмірності. Здатність паралельного алгоритму ефективно використовувати процесори при збільшенні складності розрахунків є важливою характеристикою паралельних обчислень і називається мірою масштабованості [2-4]. Паралельний алгоритм є масштабованим, якщо при зростанні кількості процесорів він забезпечує збільшення прискорення при збереженні постійного рівня ефективності використання процесорів.

3.4.1 Поняття масштабованості паралельних обчислень

Однією з найважливіших властивостей паралельного алгоритму в комбінації з паралельною архітектурою, є її *масштабованість*. У літературі можна знайти достатньо велику кількість визначень цієї властивості [2-4, 28-33], які в більшості своїй вказують на той факт, що для програми при зміні «масштабів» її роботи ряд чинників, що знижують ефективність, починають мати більший або менший вплив. Результатом цього стає зміна таких показників якості роботи паралельної програми, як прискорення і ефективність, що характеризує здатність використовувати додаткові обчислювальні ресурси системи.

Поняття масштабованості паралельних алгоритмів може бути визначено по-різному. У декількох джерелах [2-4] визначення масштабованості можуть досить значно відрізнятися один від одного, відбиваючи той факт, що поняття масштабованості складне і вимагає ретельного аналізу.

Здатність паралельного алгоритму ефективно використовувати процесори при збільшенні складності розрахунків є важливою характеристикою паралельних обчислень і називається *масштабованістю*. Паралельний алгоритм є *масштабованим (scalable)*, якщо при зростанні числа процесорів він забезпечує збільшення

прискорення при збереженні постійного рівня ефективності використання процесорів.

Аналіз масштабованості може бути використаний для різних цілей. Наприклад, щоб вибрати найкращу комбінацію «алгоритм − архітектура» для завдання при різних обмеженнях на зростання розміру задачі і числа процесорів. Також може бути використаний для прогнозування ефективності паралельного алгоритму і паралельної архітектури, що складається з великої кількості процесорів відомої продуктивності, на менше число процесорів. Для фіксованого розміру завдання аналіз масштабованості може бути використаний, щоб визначити оптимальну кількість використаних процесорів, та максимальне можливе прискорення. Аналіз масштабованості може передбачити вплив зміни апаратної технології на продуктивність і, таким чином, допомогти в розробці паралельних архітектур для розв'язання різних завдань.

Часто властивість масштабованості паралельної програми зображують графічно у вигляді кривої, яка показує залежність прискорення або продуктивності програми від числа використаних процесорів. Відзначається вплив на характер залежності інших факторів, тому обумовлюється, за яких параметрів була отримана залежність.

Дослідити масштабованість паралельних програм виключно важливо, тому що використання найпотужніших суперкомп'ютерів виправдано тільки у випадку, якщо програма ефективно розподіляє обчислення. Знаючи масштабованість, можна зробити висновки про роботу паралельної програми і визначити найбільш вузькі місця зв'язування паралельної програми – обчислювальна система, що знижує продуктивність обчислювального процесу.

Необхідно враховувати багато факторів, такі як архітектура обчислювальної системи, алгоритм, реалізація в коді і використання

даних. На підставі даних масштабованості можна зробити висновки про необхідність оптимізації програми і причини виникаючих проблем.

Масштабованість паралельних програм можна розділити на кілька основних типів [3]:

– сильна масштабованість (*strong scaling*) – залежність продуктивності від кількості процесорів p при фіксованій обчислювальній складності задачі ($m = const$);

– масштабованість в ширину (*wide scaling*) – залежність продуктивності від обчислювальної складності задачі m при фіксованому числі процесорів ($p = const$);

– слабка масштабованість (*weak scaling*) – залежність продуктивності від кількості процесорів p при фіксованій обчислювальній складності задачі в перерахунку на один вузол ($W/p = const$).

3.4.2 Основи ізоефективного аналізу

Існує декілька підходів для кількісної оцінки властивостей масштабованості алгоритму. Найбільш уживаною є метрика, запропонована в [4,-28-33] та заснована на введенні функції рівної ефективності або ізоефективності *(isoefficiency function)*.

Визначається нова динамічна характеристика – *накладні витрати на паралелізм (total overhead)*:

$$T_0(m,p) = p \cdot T_p - T_1. \tag{3.16}$$

Величина загальних накладних витрат включає сумарні витрати всіх процесорів паралельної системи: на реалізацію обмінів, послідовну частину розпаралеленого алгоритму, непродуктивні витрати на синхронізацію та час простою через незбалансованість завантаження процесорів.

Використовуючи введені позначення, отримують нові вирази для коефіцієнтів прискорення та ефективності:

$$S = T_1/T_p = {(p \cdot T_1)}\Big/{(T_1 + T_0)},$$

$$E = S/p = T_1/(T_1 + T_p) = 1/(1 + T_0/T_1). \quad (3.17)$$

Останній вираз показує, що, якщо складність розв'язуваної задачі являється фіксованою ($T_1 = const$), то при зростанні числа процесорів ефективність, як правило, буде спадати за рахунок зростання накладних витрат T_0.

Нехай $E = const$ задає необхідний рівень ефективності виконуваних обчислень, тоді:

$$T_0/T_1 = (1 - E)/E, \quad (3.18)$$

$$T_1 = E/(1 - E) \cdot T_0 = K(pT_p - T_1), \quad (3.19)$$

де $K = E/(1 - E)$ – є коефіцієнт, що залежить тільки від значення показника ефективності.

Вираз $T_1 = (E/(1 - E)) \cdot T_0 = K \cdot T_0$ (3.19), є центральним/основним співвідношенням *ізоефективного аналізу*.

Величина $K = E/(1 - E)$ називається *коефіцієнтом масштабування*. Наприклад, $E = 0{,}5 \implies K = 1$ або $E = 0{,}9 \implies K = 9$.

Породжену останнім співвідношенням залежність $m = f_E(p)$ між складністю розв'язуваної задачі та кількістю процесорів називають *функцією ізоефективності*. Функція f_E визначає, як треба збільшувати розмір задачі при збільшенні кількості процесорів для підтримки постійної ефективності.

Ізоефективна функція деяких паралельних систем є поліномом від розмірності процесорного поля: $O(p^x)$, де $x \geq 1$. При $x = 1$ отримуємо *лінійно масштабований* паралельний алгоритм. Малий ступінь p у функції ізоефективності свідчить про високу масштабованість. Якщо ж

мається експоненційна (або будь-яка інша швидко зростаюча функція) залежність, то мова йде про слабо масштабовані системи.

Близьким до лінійно масштабованих алгоритмів вважається *квазілінійний алгоритм* із функцією ізоефективності порядку $O(log_x p)$ [2-4]. Таким чином, побудова функції ізоефективності – це спроба поєднати в єдиному аналітичному виразі характеристики задачі та паралельної архітектури.

Під час аналізу ефективності паралельних алгоритмів необхідно враховувати тимчасові машинно-залежні параметри, що впливають на швидкість виконання паралельних обчислень. Такими параметрами є час виконання операції додавання/добутку t_{ad}, t_{mul}, або час виконання довільної операції з рухомою точкою. Кількість операцій з рухомою точкою в секунду (*Floating Point Operations Per Second – FLOPS*) є характеристикою продуктивності процесора.

Час виконання операції з рухомою точкою будемо позначати:

$$t_{op} = 1/FLOPS. \tag{3.17}$$

При підрахунку динамічних характеристик алгоритмів часто вважається, що будь-яка арифметична операція з рухомою точкою виконується за один і той же час t_{op} незалежно від виду операції. Це припущення справедливе для більшості комп'ютерів RISC архітектури.

Покажемо в якості ілюстрації розрахунок функції ізоефективності для прикладу підсумовування n числових значень на p процесорах, якщо $T_1 = n, T_p = n/p + log_2 p$. Часові дані приведено до t_{op}.

Коефіцієнти прискорення та ефективності розраховуються як:

$$S_p = \frac{n}{n/p + log_2 p},$$

$$E_p = \frac{1}{1 + p\, log_2 p/p},$$

загальні накладні витрати: $T_0 = p \cdot T_p - T_1 = p\, log_2 p$.

Отже, основне співвідношення ізоефективного аналізу буде мати наступний вигляд:

$$T_1 = K \cdot T_0 = K \cdot p\ log_2 p \implies n = K \cdot p\ log_2 p. \quad (3.18)$$

В даному випадку, функція ізоефективності і є вираз (3.18):

$f_E = K \cdot p\ log_2 p.$

Як результат, при числі процесорів $p = 32$, маємо:

$E = 0{,}5 \implies K = 1 \implies n \geq 32 \cdot log_2 32 = 160,$

$E = 0{,}9 \implies K = 9 \implies n \geq 9 \cdot 32 \cdot log_2 32 = 1440.$

Кількість значень, що підсумовуються для досягнення заданої ефективності, визначається отриманою нерівністю, тобто при числі процесорів $p = 16$ для забезпечення рівня ефективності $E = 0.5$ (тобто $K = 1$) кількість значень повинно бути не менше $n = 160$. Або ж, при зростанні числа процесорів з p до q $(q > p)$ для забезпечення пропорційного зростання прискорення $(Sq\ /\ Sp) = (q\ /\ p)$ необхідно збільшити число доданків n у $(q \cdot log_2 2q)\ /\ (p \cdot log_2 2p)$ разів.

Другий приклад – чисельне визначення числа π за рахунок обчислення інтегралу: $\pi = \int_0^1 \frac{4}{1+x^2} dx$ [2-3].

Задля чисельного інтегрування застосуємо метод прямокутнику (рис. 3.9-3.10).

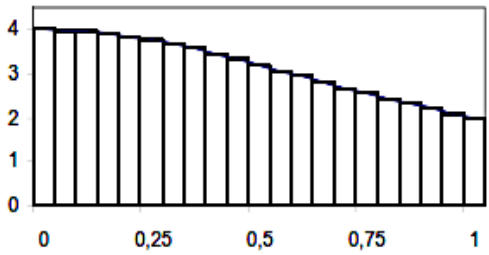


Рисунок 3.9 – Метод прямокутнику для обчислення числа π

Ідея паралельного методу прямокутнику: розподілимо обчислення поміж трьох процесорів, отримані на окремих процесорах часткові суми повинні бути підсумовані. Нехай при виконанні паралельного алгоритму отримані наступні дані: $n-$ кількість розбиття відрізку $[0,1]$, $m-$ кількість вузлів сітки на окремому процесорі, $m = \left\lceil \frac{n}{p} \right\rceil \leq \frac{n}{p}+1$.

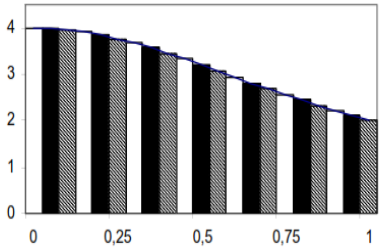


Рисунок 3.10 – Метод прямокутнику для обчислення числа

Вирази для тимчасових показників алгоритму приведено до t_{op}: $T_1 = 6n, T_p = 6n/p + 6 + log_2 p$. Тоді, $S_p = 6n/(6n/p + 6 + log_2 p), E_p = 6n/(6n + 6p + plog_2 p), T_0 = 6p + plog_2 p.$

Функція ізоефективності: $T_1 = K\left(p \cdot T_p - T_0\right) = K(6p + plog_2 p) \Longrightarrow n = [K(6p + plog_2 p]/6. E = 0{,}5 \Longrightarrow K = 1, p = 8 \Longrightarrow n > 12 \vee p = 64 \Longrightarrow n > 128.$

Для визначення областей пріоритетного використання паралельних алгоритмі або діапазону значень m і p , при яких один алгоритм працює краще ніж інший, прирівнюють їх накладні витрати і виводять m, як функцію від числа процесорів p, якщо це можливо:

$$T_0^{Alg1} = T_0^{Alg2} \Longrightarrow m = f_{Alg1\_to\_Alg2}(p). \quad (3.19)$$

Якщо $m > f_{Alg1\_to\_Alg2}(p)$, то алгоритм 1 має менші накладні витрати ніж алгоритм 2 та навпаки.

Якщо побудувати графік функції $m = f_{Alg1\_to\_Alg2}(p)$, то для частини площі зліва від функції алгоритм 1 має менші накладні витрати ніж 2, праворуч – алгоритм 2 переважає над алгоритмом 1.

3.5 Завдання для самостійної роботи

1. Розробити паралельний алгоритм і виконати оцінку динамічних показників паралелізму для підсумовування заданого набору числових даних (каскадний алгоритм, алгоритм здвоювання тощо).
2. Розробити паралельний алгоритм і виконати оцінку динамічних показників паралелізму для отримання мінімального або максимального значення серед заданого набору числових даних.
3. Розробити паралельний алгоритм і виконати оцінку динамічних показників паралелізму середнього значення для заданого набору числових даних.
4. Розробити паралельний алгоритм і виконати оцінку динамічних показників паралелізму для обчислення добутку вектору на скаляр.
5. Розробити паралельний алгоритм і виконати оцінку динамічних показників паралелізму скалярного добутку двох векторів.
6. Розробити паралельний алгоритм і виконати виконати оцінку динамічних показників паралелізму скалярного добутку матриці на вектор.
7. Розробити паралельний алгоритм і виконати оцінку динамічних показників паралелізму скалярного матричного множення.
8. Виконати відповідно до закону Амдала оцінку максимально досяжного прискорення для паралельного алгоритму підсумовування, визначення максимального, мінімального або середнього значення заданого набору числових даних.

9. Виконати оцінку прискорення масштабування для паралельного алгоритму підсумовування, визначення максимального, мінімального або середнього значення заданого набору числових даних.

10. Виконати побудову функцію ізоефективності та проаналізувати масштабованість паралельного алгоритму підсумовування, визначення максимального, мінімального або середнього значення заданого набору числових даних.

3.6 Контрольні питання

1. Які графові методи опису паралельних алгоритмів існують? Що таке графи впливу та їх особливості?

2. Які методи або технології розробки паралельних алгоритмів існують?

3. В чому полягає суть ієрархічної декомпозиційної методики?

4. Що таке метричні характеристики паралельних обчислень? На які групи їх можна поділити?

5. Як визначаються поняття прискорення і ефективності? Чи можливе досягнення зверхлінійного прискорення? У чому полягає суперечливість показників прискорення і ефективності?

6. Як формулюється закон Амдала? Який аспект паралельних обчислень дозволяє врахувати цей закон?

7. Які припущення використовуються для обґрунтування закону Густавсона-Барсіса?

8. Що таке масштабованість паралельного алгоритму? Наведіть приклади методів з різним рівнем масштабованості.

9. Загальні накладні витрати на паралелізм – вміст та засіб визначення.

10. Мета та основне співвідношення ізоефективного аналізу? Як визначається функція ізоефективності і що вона означає?

РОЗДІЛ 4
ПАРАЛЕЛЬНІ АЛГОРИТМИ МАТРИЧНОГО МНОЖЕННЯ

Матриці та операції над ними широко використовуються при математичному моделюванні найрізноманітніших процесів, явищ і систем, а також складають основу багатьох наукових розрахунків. Будучи обчислювально трудомісткими, матричні обчислення є класичною областю застосування паралельних обчислень. З одного боку, використання високопродуктивних паралельних систем дозволяє істотно підвищити складність завдань, які розв'язуються. З іншого боку, через своє достатньо просте формулювання матричні операції надають прекрасну можливість для демонстрації багатьох прийомів і методів паралельного програмування.

У даному розділі аналізуються методи паралельних обчислень для операції *матричного множення* (ММ) або *добутку* (МД).

Обчислення матричного добутку $C = A \times B$, скалярного типу для двох квадратних матриць A та B розміру $m \times m$ виконується наступним чином (рис. 4.1):

$$C = A \times B, c_{ij} = \sum_{r=1}^{m} a_{ir} \times b_{rj}, \tag{4.1}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix} \times \begin{bmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mm} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mm} \end{bmatrix},$$

де $A = \|a_{ij}\|, B = \|b_{ij}\|, C = \|c_{ij}\|, i, j = \overline{1, m}.$

Зауваження: наведене обмеження щодо квадратної форми матриць не є суттєвим, бо усі алгоритми, що будуть розглядатися, описано саме для таких умов і легко може бути модифіковано при їх порушенні (наприклад, операцією доповнення нулями за стовбцями та рядками без змін у алгоритмі).

$$
\begin{aligned}
&for\ \ i = 1; i \le m; i++ \\
&\{\, for\ \ j = 1; j \le m; j++ \\
&\{\, c_{ij} = 0 \\
&for\ \ k = 1; k \le m; k++ \\
&c_{ij} = c_{ij}\ + a_{ij} \times b_{ij} \\
&\}\}
\end{aligned}
$$

$$T_1 = m^2(mt_{mul} + mt_{ad}) = 2m^3 t_{op}$$

Рисунок 4.1 – Послідовний стандартний алгоритм ММ

Алгоритми матричного множення суттєво розрізняються за ступенем заповнювання матриць:

– щільні (*dense*) або щільно-заповнені – число нульових елементів є незначним у порівнянні із загальною кількістю елементів матриць;

– розріджені (*spars*e) – більша частина елементів матриці є нульовими, такі матриці у свою чергу можуть бути загального або спеціального виду, як то: трикутні, стрічкові тощо.

Послідовний алгоритм(и), що реалізує цю операцію за формулою (4.1), називають *стандартним* або *традиційним* на відміну від *швидкого множення* на основі рекурсивного алгоритму Штрассена (п. 4.3).

Обчислювальна складність послідовних алгоритмів ММ $C = A \times B$ для вхідних даних розміру $m \times m$ складає:

1) традиційні алгоритми – $\Theta(m^3)$;

2) швидке множення за методом Штрассена (1969) – $\Theta\left(m^{log_2^7}\right) = \Theta(m^{2,81})$;

3) швидке множення за методом Штрассена-Копперсміта-Віноградa (1990) – $\Theta\left(m^{log_2^4}\right) = \Theta(m^{2,376})$.

Класифікація алгоритмів матричного множення за типом послідовного алгоритму, що було покладено в основу паралельного аналогу, наведена на рисунку 4.2.

Рисунок 4.2 – Класифікація алгоритмів множення щільно-заповнених матриць

Зауваження: далі, якщо особливо не обумовлено, при викладі матеріалу будемо вважати, що розглядаються матриці які є щільно-заповненими та квадратними і базуються на традиційному послідовному алгоритмі множення, окрім пункту 4.3.

Для багатьох методів матричних обчислень характерним є повторення одних і тих же обчислювальних дій для різних елементів матриць. Даний момент свідчить про наявність паралелізму за даними при виконанні матричних розрахунків і, як результат, розпаралелювання матричних операцій зводиться в більшості випадків до поділу оброблюваних матриць між процесорами використовуваної обчислювальної системи.

Вибір способу поділу матриць призводить до визначення конкретного методу паралельних обчислень; існування різних схем розподілу даних породжує цілий ряд паралельних алгоритмів матричних обчислень [34-45].

Найбільш загальні і широко використовувані способи поділу матриць складаються у разі необхідності розділення даних на смуги (за вертикаллю або горизонталлю) або на прямокутні фрагменти (блоки).

Таким чином, за способом розбиття даних методи матричного множення поділяються на такі класи (рис. 4.3):

– блокові (*chessboard block*);
– стрічкові (*block-striped*):
  – горизонтальні (*rowwise*);
  – вертикальні (*columnwise)*.

Горизонтальне розбиття Вертикальне розбиття Блокове розбиття

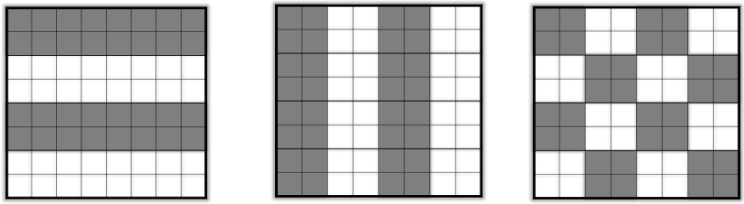


Рисунок 4.3 – Приклади розбиття елементів матриць

4.1 Паралельні блокові алгоритми матричного множення

Нехай маємо замкнуту сітку процесорів або 2D-тор розміру: $p \times p$, тобто загальна кількість процесорів складає: p^2.

Порядок перемножуваних матриць дорівнює m, вхідні матриці A та B, матриця-результат C є квадратними і, в свою чергу, розбиваються на квадратні блоки розміром:

$$k = \lceil m/p \rceil, \tag{4.2}$$

відповідно, загальна кількість отриманих блоків обчислюється як q^2, де $q = \lceil m/k \rceil$. (4.3)

Операцію множення матриць $C = A \times B$ порядку m в блоковому вигляді можна представити таким чином:

$$\begin{bmatrix} A_{11} & \cdots & A_{1q} \\ \vdots & \ddots & \vdots \\ A_{q1} & \cdots & A_{qq} \end{bmatrix} \times \begin{bmatrix} B_{11} & \cdots & B_{1q} \\ \vdots & \ddots & \vdots \\ B_{q1} & \cdots & B_{qq} \end{bmatrix} = \begin{bmatrix} C_{11} & \cdots & C_{1q} \\ \vdots & \ddots & \vdots \\ C_{q1} & \cdots & C_{qq} \end{bmatrix}, \quad (4.4)$$

де кожен блок C_{ij} матриці C визначається відповідно до виразу:

$$C_{ij} = \sum_{r=1}^{q} A_{ir} \times B_{rj}. \quad (4.5)$$

Вид першого блоку матриці A:

$$A_{11} = \begin{bmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{k1} & \cdots & a_{kk} \end{bmatrix}. \quad (4.6)$$

Зауваження: розмір вхідних матриць не обов'язково кратний до розміру процесорної сітки (при необхідності недостатня кількість елементів покладається нулями), тому далі q, k вважаємо цілими числами.

4.1.1 Алгоритм Кеннона

Алгоритм Кеннона (*Lynn Elliot Cannon*) – блоково-систолічний алгоритм множення матриць [2-4]. Алгоритм Кеннона – паралельний алгоритм матричного множення, заснований на блоковому розбитті даних, блоковий аналог систолічного алгоритму множення матриць.

4.1.1.1 Загальний опис алгоритму Кеннона та схема відображення на паралельну топологію

Етап ініціалізації алгоритму або попередньої підготовки полягає в виконанні двох кроків:

– розбиття матриць, що перемножуються, на блоки;
– відображення блоків на топологію процесорного поля.

Перед виконанням алгоритму кожен процесор замкнутої 2D-сітки з номером (i, j) містить (рис. 4.4) два блоки матриць вхідних даних: A_{ij} та B_{ij} та формує блок матриці результату C_{ij} (спочатку нульовий).

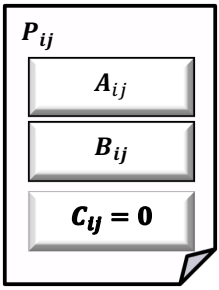


Рисунок 4.4 – Розподіл вхідних даних на процесорі P_{ij} за алгоритмом Кеннона

Алгоритм Кеннона починається з підготовчого етапу, що складається з двох операцій косого переміщування.

Крок 1. Блоки матриці A «косо переміщуються» вліво за рядками $\underset{i}{\leftarrow} A$: блоки i-того рядка матриці A передаються циклічно вдовж рядка замкнутої сітки процесорів вліво на $(i-1)$ позицію:

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & & A_{2q} \\ \vdots & & \ddots & \vdots \\ A_{q1} & A_{q2} & \cdots & A_{qq} \end{bmatrix} \Rightarrow \underset{i}{\leftarrow} A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{22} & A_{23} & & A_{21} \\ \vdots & & \ddots & \vdots \\ A_{qq} & A_{q1} & \cdots & A_{qq-1} \end{bmatrix}. \quad (4.7)$$

Крок 2. Блоки матриці B «косо переміщуються» вгору за стовпцями: блоки j-того стовпця матриці B передаються циклічно вдовж стовпця замкнутої сітки процесорів вгору на $(j-1)$ позицію:

$$B = \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1q} \\ B_{21} & B_{22} & & B_{2q} \\ \vdots & & \ddots & \vdots \\ B_{q1} & B_{q2} & \cdots & B_{qq} \end{bmatrix} \Rightarrow \; B \uparrow^{j} = \begin{bmatrix} B_{11} & B_{22} & \dots & B_{qq} \\ B_{21} & B_{32} & & B_{1q} \\ \vdots & & \ddots & \vdots \\ B_{q1} & B_{12} & \cdots & B_{q-1q} \end{bmatrix}. \quad (4.8)$$

Крок 3. Основний етап алгоритму Кеннона виконується $q = p$ разів на всіх процесорах паралельно. Кожного разу на процесорі з номером (i, j) виконується множення поточних блоків матриць A_{ij}, B_{ij} та додавання матричних блоків $C_{ij} := C_{ij} + A_{ij} \times B_{ij}$. Потім проводиться одноразове переміщення вліво для блоків матриці A і вгору для блоків матриці B, описані дії повторюються. В результаті на кожному процесорі з номером (i, j) виходить блок матриці результату – C_{ij}.

Таким чином, в алгоритмі Кеннона паралельно над блоками виконуються в точності ті ж дії, які виконуються систолічним алгоритмом над елементами матриць, причому додавання і множення матричних блоків виконуються послідовно одним процесором на основі стандартного алгоритму множення матриць (рядок на стовпець всередині блоку).

Вочевидь, що за типом даних та засобом їх розбиття, алгоритм Кеннона природньо відображається на топологію замкнута двомірна сітка або 2D-тор (рис. 4.5).

4.1.1.2 Динамічні характеристики алгоритму Кеннона

Отримаємо аналітичні вирази для динамічних характеристик алгоритму Кеннона. Час паралельного виконання алгоритму Кеннона складається з двох доданків – часу обчислень та комунікацій:

$$T_p^{Cannon} = T_{p,comp}^{Cannon} + T_{p,comm}^{Cannon}. \quad (4.9)$$

Порахуємо ці доданки за окремими кроками $(si, i = \overline{1,3})$ алгоритму. На підготовчому етапі (кроки $s1$ та $s2$) ніяких обчислень не проводилося, тобто: $T_{p,comp}^{s1+s2} = 0$.

Таким чином, треба порахувати тільки час на обмін даними: час на «косе переміщення» блоків матриці A вліво за рядками і на «косе переміщення» блоків матриці B вгору за стовпцями.

За змістом обидві операції є стандартними операціями пересилки даних для топології – кільце (рядок/стовпець 2D-тору розміру p = кільце довжиною p) [16 − 17,38 − 39].

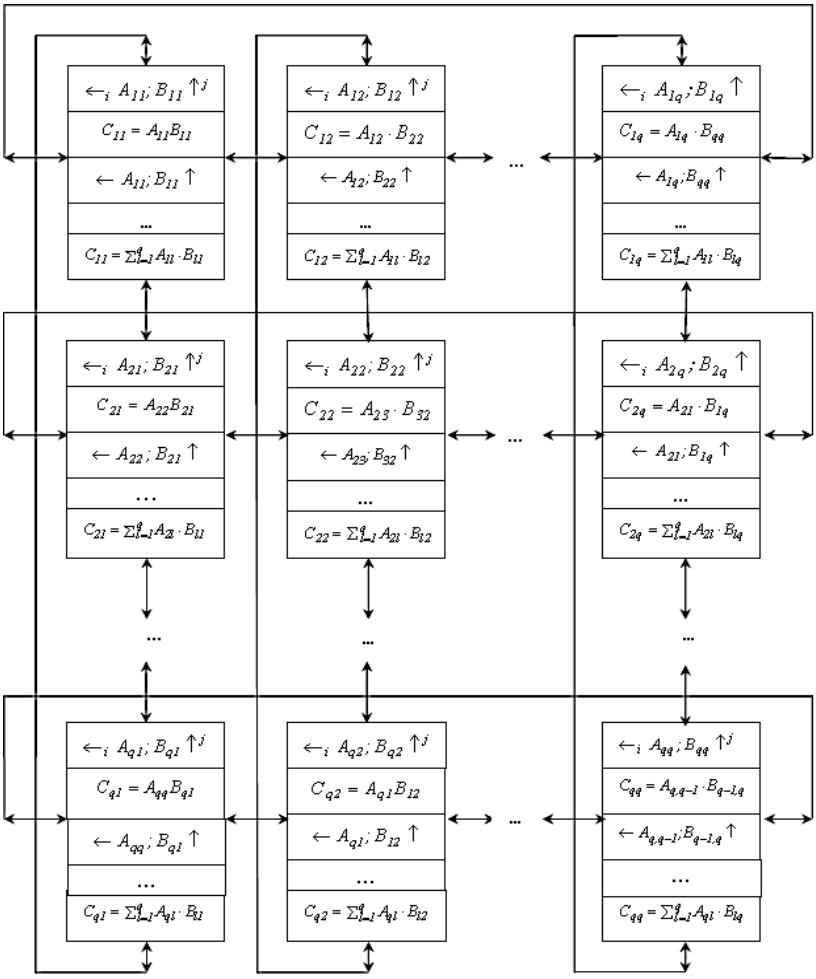


Рисунок 4.5 – Схема відображення алгоритму Кеннона на топологію двовимірна замкнута сітка або 2D-тор

Скористаємось формулою (2.1). За простою моделлю Хокні для режиму передачі повідомлень при $l = 1$ (дані передаються «сусіду»), максимально $(p - 1)$ раз для останнього рядка, маємо:

$$T_{p,comm}^{s1} = (p - 1) \cdot (t_s + V t_w), \tag{4.10}$$

де V – обсяг даних, що переміщуються, в даному випадку це кількість елементів одного блоку матриці або обсяг блоку:

$V = k^2 = m^2/p^2$.

Аналогічно, для другого етапу алгоритму Кеннона маємо (на останньому кроці передачі даних не буде):

$$T_{p,comm}^{s2} = (p - 1) \cdot (t_s + V t_w). \tag{4.11}$$

Таким чином, загальний час на обмін на підготовчому етапі складає:

$$T_{p,comm}^{s1+s2} = 2(p - 1) \cdot \left(t_s + \frac{m^2}{p^2} t_w\right). \tag{4.12}$$

На основному кроці виконання алгоритму ($s3$), що повторюється p раз, час виконання визначається на основі реалізації послідовного алгоритму матричного множення блоків розміру $k \times k$ на одному процесорі:

$$T_{p,comp}^{s3} = T_{A_{ij} \times B_{ij}}, \tag{4.13}$$

де $T_{A_{ij} \times B_{ij}}$ – час на обчислення добутку поточних блоків матриць A та B, що розташовані на (i, j) процесорі.

Тоді, на кожному з p кроків час на обчислення розраховується, як:

$$T_{p,comp}^{s3} = T_{A_{ij} \times B_{ij}} = k^3 t_{mul} + k^3 t_{ad} = 2 \frac{m^3}{p^3} t_{op}, \tag{4.14}$$

а загальний час на реалізацію обчислень на всіх кроках алгоритму Кеннона складає:

$$p \cdot T_{p,comp}^{s3} = 2 \frac{m^3}{p^2} t_{op}. \tag{4.15}$$

(4.16)

Час на виконання обмінів основного етапу складається з часу на одноразове переміщення вліво для блоків матриці A і вгору для блоків матриці B:

$$T_{p,comm}^{s3} = 2(p-1)\cdot\left(t_s + \frac{m^2}{p^2}t_w\right).$$

Загальний час на усі операції обміну даними за блоковим алгоритмом Кеннона обчислюється за виразом:

$$T_{p,comm}^{Cannon} = T_{p,comm}^{s1+s2} + T_{p,comm}^{s3} = 4(p-1)\cdot\left(t_s + \frac{m^2}{p^2}t_w\right). \quad (4.17)$$

Отже, $T_p^{Cannon} = 2\frac{m^3}{p^2}t_{op} + 4(p-1)\cdot\left(t_s + \frac{m^2}{p^2}t_w\right).$ (4.18)

Основними динамічними характеристиками, що визначають якість паралельного, є коефіцієнти прискорення та ефективності:

$$S_p^{Cannon} = \frac{T_1^{Cannon}}{T_p^{Cannon}} = \frac{2m^3 t_{op}}{2\frac{m^3}{p^2}t_{op} + 4(p-1)\left(t_s + \frac{m^2}{p^2}t_w\right)}, \quad (4.19)$$

$$E_p^{Cannon} = \frac{S_p^{Cannon}}{p^2} = \frac{2m^3\cdot t_{op}}{2m^3 t_{op} + 4(p-1)\cdot(p^2 t_s + m^2 t_w)}. \quad (4.20)$$

Загальні накладні витрати на паралелізм:

$$T_0^{Cannon} = p^2\cdot T_p^{Cannon} - T_1^{Cannon} =$$

$$= p^2\left[2\frac{m^3}{p^2}t_{op} + 4(p-1)\cdot\left(t_s + \frac{m^2}{p^2}t_w\right)\right] - 2m^3 t_{op} =$$

$$= 4(p-1)\cdot(p^2 t_s + m^2 t_w).$$

$$T_0^{Cannon} = 4(p-1)\cdot(p^2 t_s + m^2 t_w). \quad (4.21)$$

Нагадаємо, що отримані аналітичні вирази метрик виведено при умові використання механізму передачі повідомлень для топології двовимірний тор.

4.1.1.3 Демонстрація покрокового виконання алгоритму Кеннона

На наступних рисунках 4.6-4.16 показано покрокове виконання алгоритму Кеннона для паралельної системи топології 2D-тор розміру $p \times p$. Вхідні матриці мають блоковий вид з параметрами: $p = q = 4$. На кожному рисунку наведено час паралельного виконання поточного i-го етапу T_p^{si} та його складові. Операції вводу та виводу з пам'яті не розглядаються.

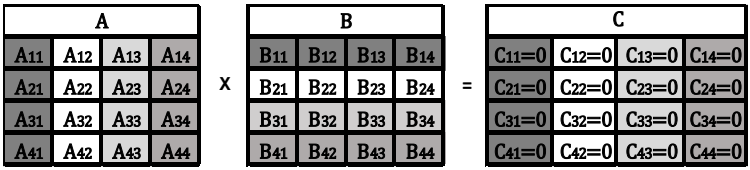
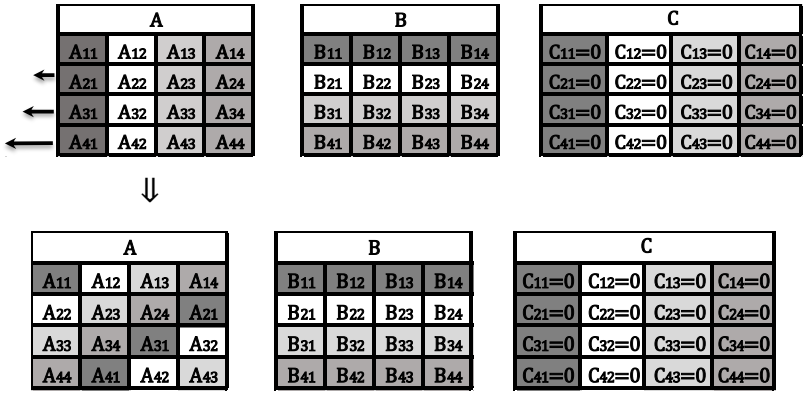
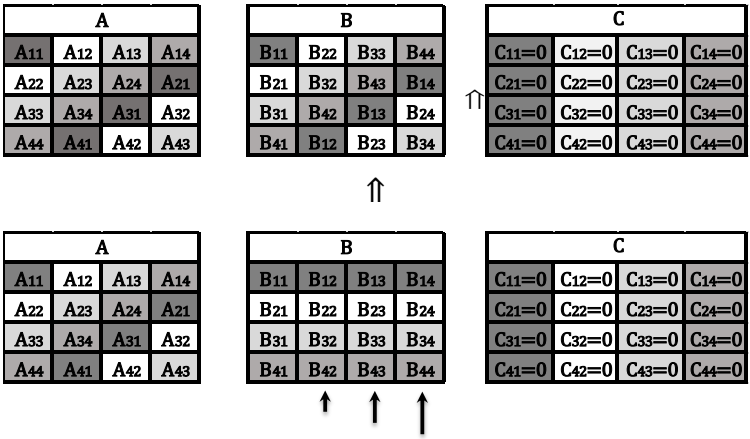


Рисунок 4.6 – Ініціалізація алгоритму: розподіл вхідних даних



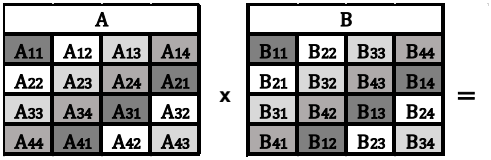
$$T_{p,comp}^{s1} = 0,\ T_{p,comm}^{s1} = (p-1) \cdot \left(t_s + \frac{m^2}{p^2} t_w\right)$$

Рисунок 4.7 – Підготовчий етап: «косе переміщення» блоків матриці A вліво



$$T_{p,comp}^{s2} = 0, T_{p,comm}^{s2} = (p-1) \cdot \left(t_s + \frac{m^2}{p^2} t_w\right)$$

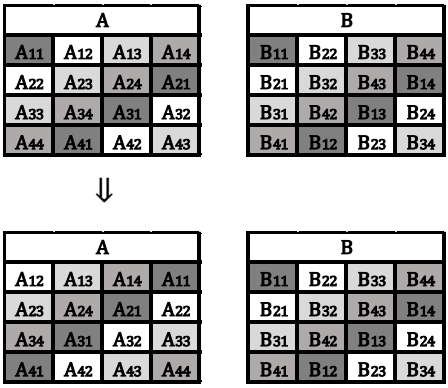
Рисунок 4.8 – Підготовчий етап: «косе переміщення» блоків матриці B вгору



C			
C11=0 + A11 * B11	C12=0 + A12 * B22	C13=0 + A13 * B33	C14=0 + A14 * B44
C21=0 + A22 * B21	C22=0 + A23 * B32	C23=0 + A24 * B43	C24=0 + A21 * B14
C31=0 + A33 * B31	C32=0 + A34 * B42	C33=0 + A31 * B13	C34=0 + A32 * B24
C41=0 + A44 * B41	C42=0 + A41 * B12	C43=0 + A42 * B23	C44=0 + A43 * B34

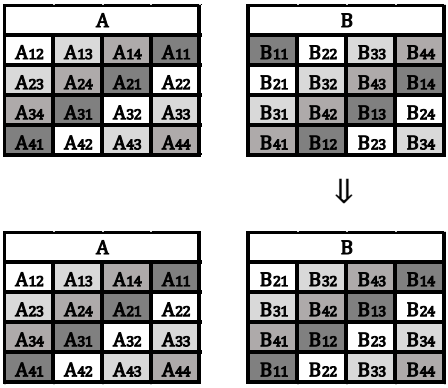
$$T_{p,comp}^{s3} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.9 – Основний етап: крок 1, поелементне множення поточних блоків матриць A і B за стандартним алгоритмом ММ



$$T_{p,comm}^{s3} = t_s + \frac{m^2}{p^2} t_w$$

Рисунок 4.10 – Основний етап: крок 1, циклічне переміщення блоків матриці A вліво



$$T_{p,comm}^{s3} = t_s + \frac{m^2}{p^2} t_w$$

Рисунок 4.11 – Основний етап: крок 1, циклічне переміщення блоків матриці B вгору

C			
$C_{11}= A_{11} * B_{11} + A_{12} * B_{21}$	$C_{12}= A_{12} * B_{22} + A_{13} * B_{32}$	$C_{13}= A_{13} * B_{33} + A_{14} * B_{43}$	$C_{14}= A_{14} * B_{44} + A_{11} * B_{14}$
$C_{21}= A_{22} * B_{21} + A_{23} * B_{31}$	$C_{22}= A_{23} * B_{32} + A_{24} * B_{42}$	$C_{23}= A_{24} * B_{43} + A_{21} * B_{13}$	$C_{24}= A_{21} * B_{14} + A_{22} * B_{24}$
$C_{31}= A_{33} * B_{31} + A_{34} * B_{41}$	$C_{32}= A_{34} * B_{42} + A_{31} * B_{12}$	$C_{33}= A_{31} * B_{13} + A_{32} * B_{23}$	$C_{34}= A_{32} * B_{24} + A_{33} * B_{34}$
$C_{41}= A_{44} * B_{41} + A_{41} * B_{11}$	$C_{42}= A_{41} * B_{12} + A_{42} * B_{22}$	$C_{43}= A_{42} * B_{23} + A_{43} * B_{33}$	$C_{44}= A_{43} * B_{34} + A_{44} * B_{44}$

$$T_{p,comp}^{s3} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.12 – Основний етап: крок 2, поелементне множення поточних блоків матриць A і B за стандартним алгоритмом ММ

A			
A_{12}	A_{13}	A_{14}	A_{11}
A_{23}	A_{24}	A_{21}	A_{22}
A_{34}	A_{31}	A_{32}	A_{33}
A_{41}	A_{42}	A_{43}	A_{44}

B			
B_{21}	B_{32}	B_{43}	B_{14}
B_{31}	B_{42}	B_{13}	B_{24}
B_{41}	B_{12}	B_{23}	B_{34}
B_{11}	B_{22}	B_{33}	B_{44}

A			
A_{13}	A_{14}	A_{11}	A_{12}
A_{24}	A_{21}	A_{22}	A_{23}
A_{31}	A_{32}	A_{33}	A_{34}
A_{42}	A_{43}	A_{44}	A_{41}

B			
B_{31}	B_{42}	B_{13}	B_{24}
B_{41}	B_{12}	B_{23}	B_{34}
B_{11}	B_{22}	B_{33}	B_{44}
B_{21}	B_{32}	B_{43}	B_{14}

$$T_{p,comm}^{s3} = 2\left(t_s + \frac{m^2}{p^2} t_w\right)$$

Рисунок 4.13 – Основний етап: крок 2, циклічне переміщення блоків матриці A вліво та циклічне переміщення блоків матриці B вгору

C			
$C_{11} = A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$	$C_{12} = A_{12} * B_{22} + A_{13} * B_{32} + A_{14} * B_{42}$	$C_{13} = A_{13} * B_{33} + A_{14} * B_{43} + A_{11} * B_{13}$	$C_{14} = A_{14} * B_{44} + A_{11} * B_{14} + A_{12} * B_{24}$
$C_{21} = A_{22} * B_{21} + A_{23} * B_{31} + A_{24} * B_{41}$	$C_{22} = A_{23} * B_{32} + A_{24} * B_{42} + A_{21} * B_{12}$	$C_{23} = A_{24} * B_{43} + A_{21} * B_{13} + A_{22} * B_{23}$	$C_{24} = A_{21} * B_{14} + A_{22} * B_{24} + A_{23} * B_{34}$
$C_{31} = A_{33} * B_{31} + A_{34} * B_{41} + A_{31} * B_{11}$	$C_{32} = A_{34} * B_{42} + A_{31} * B_{12} + A_{32} * B_{22}$	$C_{33} = A_{31} * B_{13} + A_{32} * B_{23} + A_{33} * B_{33}$	$C_{34} = A_{32} * B_{24} + A_{33} * B_{34} + A_{34} * B_{44}$
$C_{41} = A_{44} * B_{41} + A_{41} * B_{11} + A_{42} * B_{21}$	$C_{42} = A_{41} * B_{12} + A_{42} * B_{22} + A_{43} * B_{32}$	$C_{43} = A_{42} * B_{23} + A_{43} * B_{33} + A_{44} * B_{43}$	$C_{44} = A_{43} * B_{34} + A_{44} * B_{44} + A_{41} * B_{14}$

$$T_{p,comp}^{s3} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.14 – Основний етап: крок 3, поелементне множення поточних блоків матриць A і B за стандартним алгоритмом ММ

A			
A_{13}	A_{14}	A_{11}	A_{12}
A_{24}	A_{21}	A_{22}	A_{23}
A_{31}	A_{32}	A_{33}	A_{34}
A_{42}	A_{43}	A_{44}	A_{41}

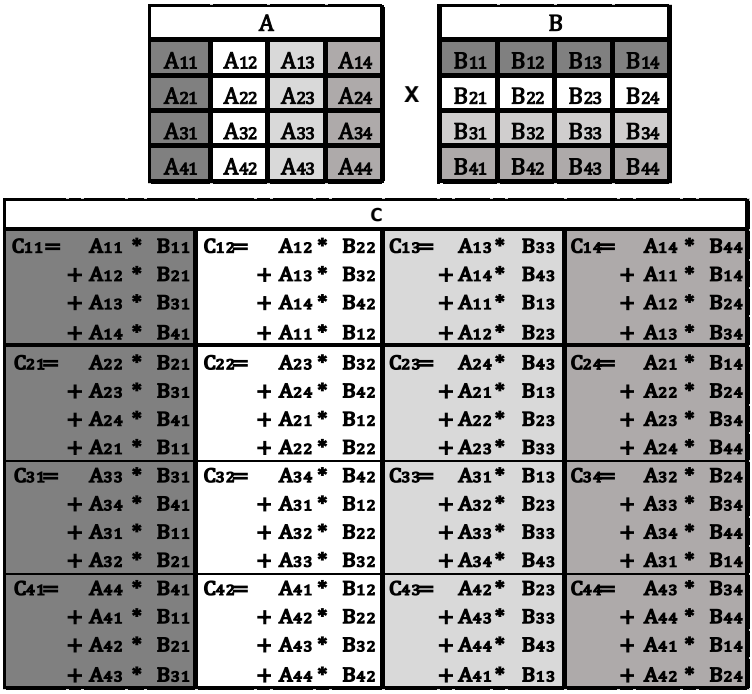
B			
B_{31}	B_{42}	B_{13}	B_{24}
B_{41}	B_{12}	B_{23}	B_{34}
B_{11}	B_{22}	B_{33}	B_{44}
B_{21}	B_{32}	B_{43}	B_{14}

A			
A_{14}	A_{11}	A_{12}	A_{13}
A_{21}	A_{22}	A_{23}	A_{24}
A_{32}	A_{33}	A_{34}	A_{31}
A_{43}	A_{44}	A_{41}	A_{42}

B			
B_{41}	B_{12}	B_{23}	B_{34}
B_{11}	B_{22}	B_{33}	B_{44}
B_{21}	B_{32}	B_{43}	B_{14}
B_{31}	B_{42}	B_{13}	B_{24}

$$T_{p,comm}^{s3} = 2\left(t_s + \frac{m^2}{p^2} t_w\right)$$

Рисунок 4.15 – Основний етап: крок 3, циклічне переміщення блоків матриці A вліво та циклічне переміщення блоків матриці B вгору



$$T_{p,comp}^{s3} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.16 – Основний етап: крок 4, поелементне множення поточних блоків матриць *A* і *B* за стандартним алгоритмом ММ

4.1.2 Алгоритм Фокса

Алгоритм Фокса (*Geoffrey Fox*), як і алгоритм Кеннона – паралельний алгоритм множення матриць з блоковим представленням даних [2-4].

При використанні однакової схеми розбиття матриць алгоритми Фокса і Кеннона відрізняються характером виконуваних операцій передачі даних. Для алгоритму Фокса в ході обчислень здійснюється

розсилка і циклічне переміщення блоків матриць, в алгоритмі Кеннона виконується тільки операція циклічного переміщення.

4.1.2.1 Загальний опис алгоритму Фокса

Відповідно до алгоритму Фокса в ході обчислень на кожному процесорі (i, j) розташовується чотири матричних блока (рис. 4.17):

– блок C_{ij} матриці-результату C;

– блок A_{ij} матриці A, що розміщується перед початком обчислень;

– поточні блоки A'_{ij}, B'_{ij} матриць A і B, що одержані в ході виконання обчислень.

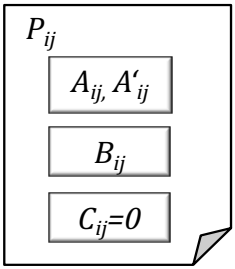


Рисунок 4.17 – Розподіл вхідних даних на процесорі P_{ij} за алгоритмом Фокса

Етап ініціалізації: кожному процесору (i, j) передаються блоки матриць аргументів МД A_{ij}, B_{ij} й обнуляються блоки матриці-результату C_{ij} на всіх процесорах.

Етап обчислення, в межах якого на кожній ітерації $n, 0 \leq n < q$, здійснюються такі операції:

– формування A'_{ij}: для кожного рядка i, $1 \leq i \leq q$ блок матриці A_{ij} процесору 2D-тору с номером (i, j) пересилається на всі процесори

того ж рядку i сітки, індекс j, що визначає положення блоку в рядку, обчислюється відповідно до виразу: $j = (i - 1 + n)\ mod\ q + 1$;

– отримані в результаті пересилань блоки A'_{ij}, B'_{ij} кожного процесору (i, j) перемножуються і додаються до блоку C_{ij}:

$$C_{ij} := C_{ij} + A'_{ij} \times B'_{ij}; \tag{4.22}$$

– блоки B'_{ij} кожного процесору (i, j) пересилаються процесорам, що є сусідами зверху в стовпцях сітки (блоки з першого рядка сітки пересилаються до останнього рядка сітки).

4.1.2.2 Динамічні характеристики алгоритму Фокса

Час паралельного виконання алгоритму Фокса складається з двох доданків, тобто часу на обчислення та передачу даних:

$$T_p^{Fox} = T_{p,comp}^{Fox} + T_{p,comm}^{Fox}. \tag{4.23}$$

На кожній n-тій ітерації алгоритму, $n = \overline{0, p-1}, p = q$, паралельно на кожному з p^2 процесорів виконуються операції обчислення добутку та додавання поточних блоків матриць A'_{ij}, B'_{ij}. Час виконання розраховується, аналогічно як і у випадку алгоритму Кеннона:

$$T_{p,comp}^{Fox,n=i} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}. \tag{4.24}$$

Таким чином, повний час на виконання обчислень за алгоритмом Фокса дорівнює:

$$T_{p,comp}^{Fox} = q\left(2\frac{m^3}{p^3}\right) \cdot t_{op} = p\left(2\frac{m^3}{p^3}\right) \cdot t_{op} = 2\frac{m^3}{p^2} t_{op}. \tag{4.25}$$

Час на обмінні операції за блоковим алгоритмом Фокса на кожній $r-$тій ітерації складається з виконання двох різних операцій передачі даних. По-перше, це операція «один-усім» в рядку матриці A': $T_{p,comm,1}^{Fox,n=r}$ та операція циклічний зсув вгору в стовпцях матриці B': $T_{p,comm,2}^{Fox,n=r}$.

Для обчислення часу виконання описаних операцій комунікації скористуємся результатами, отриманими у главі 2. Перша операція виконується в рядку матриці A', яка відображена на двовимірну сітку, тобто передача даних буде відбуватися за «кільцем». Обсяг даних, що передаються – це блок елементів матриці розміру: $k^2 = \frac{m^2}{p^2}$, тоді за формулою 2.5 маємо: $T^R_{one-to-all} = (t_s + Vt_w) \cdot \lceil p/2 \rceil$.

$$T^{Fox,n=r}_{p,comm,1} = T^R_{one-to-all} = \left(t_s + \frac{m^2}{p^2} t_w\right) \cdot \left\lceil \frac{p}{2} \right\rceil, \tag{4.26}$$

тоді як час на другу операцію обміну (у стовпці матриці B') розраховується аналогічно алгоритму Кеннона:

$$T^{Fox,n=r}_{p,comm,2} = t_s + \frac{m^2}{p^2} t_w. \tag{4.27}$$

Загальний час виконання алгоритму Фокса дорівнює:

$$T^{Fox}_p = T^{Fox}_{p,comp} + T^{Fox}_{p,comm} = 2\frac{m^3}{p^2} t_{op} + p\left(t_s + \frac{m^2}{p^2} t_w\right) \cdot \left\lceil \frac{p}{2} \right\rceil +$$

$$+(p-1) \cdot \left(t_s + \frac{m^2}{p^2} \cdot t_w\right). \tag{4.28}$$

Коефіцієнти прискорення та ефективності, відповідно, обчислюються наступним чином:

$$S^{Fox}_p = \frac{2m^3 t_{op}}{2\frac{m^3}{p^2} t_{op} + p\left(t_s + \frac{m^2}{p^2} t_w\right)\left\lceil \frac{p}{2} \right\rceil + (p-1)\left(t_s + \frac{m^2}{p^2} t_w\right)}. \tag{4.29}$$

$$E^{Fox}_p = \frac{2m^3 t_{op}}{2m^3 t_{op} + (p + p\left\lceil \frac{p}{2} \right\rceil - 1)(p^2 t_s + m^2 t_w)}. \tag{4.30}$$

Тоді, як загальні накладні витрати на паралелізм для алгоритму Фокса розраховуються як:

$$T^{Fox}_0 = p(p^2 t_s + m^2 t_w)\left\lceil \frac{p}{2} \right\rceil + (p-1)(p^2 t_s + m^2 t_w). \tag{4.31}$$

4.1.2.3 Демонстрація покрокового виконання алгоритму Фокса

Нехай маємо дві квадратні матриці A та B порядку $m = 8$ та паралельний комп'ютер з топологією 2D-тор порядку $p = 4$, кількість процесорів дорівнює 16.

Обчислити матрицю $C = A \times B$ за допомогою алгоритму Фокса.

Розбиття матриці A на блоки (рис. 4.18):

1) порядок блоку – $k = m/p = 2$ (підматриця 2×2);

2) кількість блоків – $q^2 = (m/k)^2 = 16$ ($q -$ розмір блокового представлення A, $q = 4$).

Аналогічно розбиваються на блоки матриця B та C.

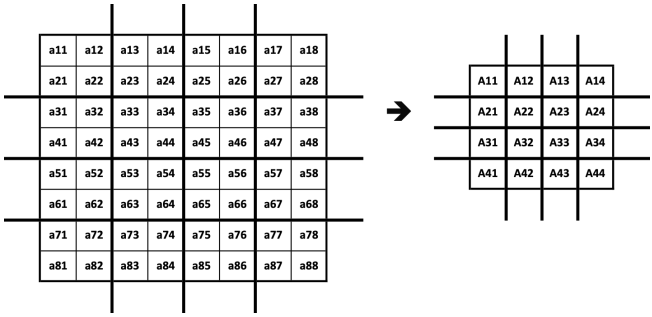


Рисунок 4.18 – Розбиття матриці A на блоки

Вхідні матриці для алгоритму Фоксу, як і у випадку алгоритму Кеннона, мають блоковий вигляд (рис. 4.19).

A			
A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

B			
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44

Рисунок 4.19 – Вхідні матриці A та B для алгоритму Фокса

На наступних рисунках 4.20-4.31 показано покрокове виконання алгоритму Фокса для паралельної системи топології 2D-тор розміру $p \times p$, де $n -$ номер ітерації.

A
A11
A22
A33
A44

A'			
A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

B'			
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44

n=0; i=1,..,q; q=4; j=(i-1+n)%q+1
i=1; A1j: j=(1-1+0)%4+1=1
i=2; A2j: j=(2-1+0)%4+1=2
i=3; A3j: j=(3-1+0)%4+1=3
i=4; A4j: j=(4-1+0)%4+1=4

$$T_{p,comm,1}^{Fox,n=0} = \left(t_s + \frac{m^2}{p^2} t_w\right) \cdot \left\lceil \frac{p}{2} \right\rceil$$

Рисунок 4.20 – Ітерація $n = 0$, пересилання даних в матрицю A′

A'			
A11	A11	A11	A11
A22	A22	A22	A22
A33	A33	A33	A33
A44	A44	A44	A44

B'			
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44

C			
C11=0 + A11 * B11	C12=0 + A11 * B12	C13=0 + A11 * B13	C14=0 + A11 * B14
C21=0 + A22 * B21	C22=0 + A22 * B22	C23=0 + A22 * B23	C24=0 + A22 * B24
C31=0 + A33 * B31	C32=0 + A33 * B32	C33=0 + A33 * B33	C34=0 + A33 * B34
C41=0 + A44 * B41	C42=0 + A44 * B42	C43=0 + A44 * B43	C44=0 + A44 * B44

$$T_{p,comp}^{Fox,n=0} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.21 – Ітерація $n = 0$, поблокове множення і додавання в матрицю C

A'			
A11	A11	A11	A11
A22	A22	A22	A22
A33	A33	A33	A33
A44	A44	A44	A44

B'			
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44

A'			
A11	A11	A11	A11
A22	A22	A22	A22
A33	A33	A33	A33
A44	A44	A44	A44

B'			
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14

$$T_{p,comm,2}^{Fox,n=0} = t_s + \frac{m^2}{p^2} t_w$$

Рисунок 4.22 – Ітерація $n = 0$, циклічне зміщення вгору за стовпцями матриці B'

A
A12
A23
A34
A41

A'			
A11	A11	A11	A11
A22	A22	A22	A22
A33	A33	A33	A33
A44	A44	A44	A44

B'			
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14

n=1; i=1,..,q; q=4; j=(i-1+n)%q+1
i=1; A1j: j=(1-1+1)%4+1=2
i=2; A2j: j=(2-1+1)%4+1=3
i=3; A3j: j=(3-1+1)%4+1=4
i=4; A4j: j=(4-1+1)%4+1=1

A'			
A12	A12	A12	A12
A23	A23	A23	A23
A34	A34	A34	A34
A41	A41	A41	A41

B'			
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14

$$T_{p,comm,1}^{Fox,n=1} = \left(t_s + \frac{m^2}{p^2} t_w\right) \cdot \left\lceil \frac{p}{2} \right\rceil$$

Рисунок 4.23 – Ітерація $n = 1$, пересилання даних в матрицю A'

A'			
A12	A12	A12	A12
A23	A23	A23	A23
A34	A34	A34	A34
A41	A41	A41	A41

×

B'			
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14

C			
C11=C11 + A12 * B21	C12=C12 + A12 * B22	C13=C13 + A12 * B23	C14=C14 + A12 * B24
C21=C21 + A23 * B31	C22=C22 + A23 * B32	C23=C23 + A23 * B33	C24=C24 + A23 * B34
C31=C31 + A34 * B41	C32=C32 + A34 * B42	C33=C33 + A34 * B43	C34=C34 + A34 * B44
C41=C41 + A41 * B11	C42=C42 + A41 * B12	C43=C43 + A41 * B13	C44=C44 + A41 * B14

$$T_{p,comp}^{Fox,n=1} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.24 – Ітерація $n = 1$, поблокове множення і додавання в матрицю C

A'			
A12	A12	A12	A12
A23	A23	A23	A23
A34	A34	A34	A34
A41	A41	A41	A41

B'			
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14

A'			
A12	A12	A12	A12
A23	A23	A23	A23
A34	A34	A34	A34
A41	A41	A41	A41

B'			
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24

$$T_{p,comm,2}^{Fox,n=1} = t_s + \frac{m^2}{p^2} t_w$$

Рисунок 4.25 – Ітерація $n = 1$, циклічне зміщення вгору за стовпцями матриці B'

A
A13
A24
A31
A42

A'			
A12	A12	A12	A12
A23	A23	A23	A23
A34	A34	A34	A34
A41	A41	A41	A41

B'			
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24

n=2; i=1,..,q; q=4; j=(i-1+n)%q+1
i=1; A1j: j=(1-1+2)%4+1=3
i=2; A2j: j=(2-1+2)%4+1=4
i=3; A3j: j=(3-1+2)%4+1=1
i=4; A4j: j=(4-1+2)%4+1=2

A'			
A13	A13	A13	A13
A24	A24	A24	A24
A31	A31	A31	A31
A42	A42	A42	A42

B'			
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24

$$T_{p,comm,1}^{Fox,n=2} = \left(t_s + \frac{m^2}{p^2} t_w\right) \cdot \left\lceil \frac{p}{2} \right\rceil$$

Рисунок 4.26 – Ітерація $n = 2$, пересилання даних в матрицю A'

A'			
A13	A13	A13	A13
A24	A24	A24	A24
A31	A31	A31	A31
A42	A42	A42	A42

×

B'			
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24

C			
C11=C11 + A13 * B31	C12=C12 + A13 * B32	C13=C13 + A13 * B33	C14=C14 + A13 * B34
C21=C21 + A24 * B41	C22=C22 + A24 * B42	C23=C23 + A24 * B43	C24=C24 + A24 * B44
C31=C31 + A31 * B11	C32=C32 + A31 * B12	C33=C33 + A31 * B13	C34=C34 + A31 * B14
C41=C41 + A42 * B21	C42=C42 + A42 * B22	C43=C43 + A42 * B23	C44=C44 + A42 * B24

$$T_{p,comp}^{Fox,n=2} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.27 – Ітерація $n = 2$, поблокове множення і додавання в матрицю C

A'			
A13	A13	A13	A13
A24	A24	A24	A24
A31	A31	A31	A31
A42	A42	A42	A42

B'			
B31	B32	B33	B34
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24

A'			
A13	A13	A13	A13
A24	A24	A24	A24
A31	A31	A31	A31
A42	A42	A42	A42

B'			
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34

$$T_{p,comm,2}^{Fox,n=2} = t_s + \frac{m^2}{p^2} t_w$$

Рисунок 4.28 – Ітерація $n = 2$, циклічне зміщення вгору за стовпцями матриці B'

A
A14
A21
A32
A43

A'			
A13	A13	A13	A13
A24	A24	A24	A24
A31	A31	A31	A31
A42	A42	A42	A42

B'			
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34

n=3; i=1,..,q; q=4; j=(i-1+n)%q+1
i=1; A1j: j=(1-1+3)%4+1=4
i=2; A2j: j=(2-1+3)%4+1=1
i=3; A3j: j=(3-1+3)%4+1=2
i=4; A4j: j=(4-1+3)%4+1=3

$$T_{p,comm,1}^{Fox,n=3} = \left(t_s + \frac{m^2}{p^2} t_w\right) \cdot \left\lceil \frac{p}{2} \right\rceil$$

Рисунок 4.29 – Ітерація $n = 3$, пересилання даних в матрицю A'

A'			
A14	A14	A14	A14
A21	A21	A21	A21
A32	A32	A32	A32
A43	A43	A43	A43

×

B'			
B41	B42	B43	B44
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34

C			
C11=C11 + A14 * B41	C12=C12 + A14 * B42	C13=C13 + A14 * B43	C14=C14 + A14 * B44
C21=C21 + A21 * B11	C22=C22 + A21 * B12	C23=C23 + A21 * B13	C24=C24 + A21 * B14
C31=C31 + A32 * B21	C32=C32 + A32 * B22	C33=C33 + A32 * B23	C34=C34 + A32 * B24
C41=C41 + A43 * B31	C42=C42 + A43 * B32	C43=C43 + A43 * B33	C44=C44 + A43 * B34

$$T_{p,comp}^{Fox,n=3} = T_{A_{ij} \times B_{ij}} = 2\frac{m^3}{p^3} t_{op}$$

Рисунок 4.30 – Ітерація $n = 3$, поблокове множення і додавання в матрицю C

A			
A11	A12	A13	A14
A21	A22	A23	A24
A31	A32	A33	A34
A41	A42	A43	A44

×

B			
B11	B12	B13	B14
B21	B22	B23	B24
B31	B32	B33	B34
B41	B42	B43	B44

C			
C11=C11 + A11 * B11 + A12 * B21 + A13 * B31 + A14 * B41	C12=C12 + A11 * B12 + A12 * B22 + A13 * B32 + A14 * B42	C13=C13 + A11 * B13 + A12 * B23 + A13 * B33 + A14 * B43	C14=C14 + A11 * B14 + A12 * B24 + A13 * B34 + A14 * B44
C21=C21 + A22 * B21 + A23 * B31 + A24 * B41 + A21 * B11	C22=C22 + A22 * B22 + A23 * B32 + A24 * B42 + A21 * B12	C23=C23 + A22 * B23 + A23 * B33 + A24 * B43 + A21 * B13	C24=C24 + A22 * B24 + A23 * B34 + A24 * B44 + A21 * B14
C31=C31 + A33 * B31 + A34 * B41 + A31 * B11 + A32 * B21	C32=C32 + A33 * B32 + A34 * B42 + A31 * B12 + A32 * B22	C33=C33 + A33 * B33 + A34 * B43 + A31 * B13 + A32 * B23	C34=C34 + A33 * B34 + A34 * B44 + A31 * B14 + A32 * B24
C41=C41 + A44 * B41 + A41 * B11 + A42 * B21 + A43 * B31	C42=C42 + A44 * B42 + A41 * B12 + A42 * B22 + A43 * B32	C43=C43 + A44 * B43 + A41 * B13 + A42 * B23 + A43 * B33	C44=C44 + A44 * B44 + A41 * B14 + A42 * B24 + A43 * B34

Рисунок 4.31 – Алгоритм Фокса, вхідні дані та результат роботи алгоритму, матриця $C = A \times B$

4.2 Паралельні стрічкові алгоритми матричного множення

В даному підрозділі представлено паралельні алгоритми матричного множення із *стрічковим* розбиттям, в яких вхідні матриці A і B та матриця-результат C розбиваються на безперервні послідовності рядків або стовпців (смуги) шириною: $k = \lceil m/p \rceil$. У найпростішому випадку при $k = 1, m = p$ смугою може служити окремий рядок або стовпець. Задля забезпечення рівномірного розподілу обчислювального навантаження за процесорами, що становлять паралельну систему, розмір смуги вибирають рівним $k = m/p$ (в припущенні, що m кратне p), якщо це звичайно можливо забезпечити [2-4].

У розглянутих нижче алгоритмах кожен процесор використовується для обчислення одного рядка/смуги результуючого добутку матриць A і B. В цьому випадку процесор повинен мати доступ до відповідного рядку/смуги матриці A і всієї матриці B. Оскільки одночасне зберігання всієї матриці B у всіх процесорах паралельного комп'ютеру вимагає надмірних витрат пам'яті, обчислення організовуються таким чином, щоб в кожен момент часу процесори містили лише частину елементів матриці B (одну вертикальну або горизонтальну смугу, один стовпець або один рядок). Доступ до решти забезпечується за допомогою операцій передачі даних.

Таким чином, кожний процесор завжди містить рядок/смугу матриць A і C. А в залежності від розбиття другого аргументу матричного множення, тобто матриці B, стрічкові паралельні алгоритми поділяються на два наступні класи:

1) з *вертикальним* стрічковим розбиттям даних: кожен процесор містить смугу *стовпця* матриці B (рис. 4.32);

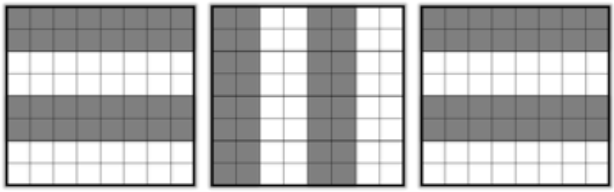
2) з *горизонтальним* стрічковим розбиттям даних: кожен процесор містить смугу *рядка* матриці B (рис. 4.45).

Зауваження: стрічкові алгоритми ММ природним чином відображаються на ОС топології кільце.

4.2.1 Паралельний стрічковий алгоритм ММ з вертикальним розбиттям даних

4.2.1.1 Загальний опис стрічкового вертикального алгоритму ММ

Загальна схема розміщення даних для паралельного алгоритму ММ із стрічковим вертикальним розбиттям наведена на рисунку 4.32. За алгоритмом спочатку процесор з номером i містить елементи i-го рядка (горизонтальної смуги) матриці A і елементи i-го стовпця (вертикальної смуги) матриці B. Елементи рядка (горизонтальної смуги) добутку $C = A \times B$ обнуляються.



A B C

Рисунок 4.32 – Загальна схема розміщення даних для паралельного алгоритму ММ із стрічковим вертикальним розбиттям

Потім запускається циклічний процес (число ітерацій дорівнює p) в ході якого виконуються наступні дві дії:

1) виконується скалярне множення відповідних елементів рядків/смуг матриці A і стовпців/смуг матриці B, що розташовані на процесорах з однаковими номерами, результати додаються до відповідного елементу рядка/смуги матриці C того ж процесору;

2) виконується циклічне пересилання рядків/смуг матриці B в сусідні процесори в рамках топології кільце (напрямок пересилки може бути довільним: як за зростанням номерів процесорів, так і за їх зменшенням).

Після завершення циклу в кожному процесорі міститься відповідний рядок/смуга добутку $A \times B$.

4.2.1.2 Динамічні характеристики стрічкового вертикального алгоритму ММ

Отримаємо динамічні характеристики стрічкового вертикального алгоритму ММ в умовах його відображення на обчислювальну систему з p процесорами та топологією кільце.

Час паралельного виконання даного алгоритму складається з двох доданків, тобто часу на обчислення та передачу даних:

$$T_p^{Str-Column} = T_{p,comp}^{Str-Column} + T_{p,comm}^{Str-Column}. \quad (4.32)$$

При виконанні однієї з p ітерацій алгоритму проводиться скалярне множення рядка/смуги на відповідний стовпець/смугу довжиною m, що призводить до отримання відповідних елементів результуючої матриці C. На кожен елемент C, що розраховується, необхідно m операцій добутку та m операцій додавання елементів матриць. Отже, через величину флопу, загальний час на виконання арифметичних операцій на один елемент матриці C складає: $2m \cdot t_{op}$. Кількість елементів матриці-результату, що обчислюються на одній ітерації в загальному випадку дорівнює: $k^2 = (m/p)^2$.

Наприклад, нехай $m = 4, p = 2 \implies k = 2$, тобто довжина смуги дорівнює 2 (рис. 4.33-4.36). Спочатку кожен процесор містить 2 рядка матриці $A: a_{ij}, i = \overline{1,k} = \overline{1,2}\,, j = \overline{1,m} = \overline{1,4}$ та 2 стовпця матриці $B: a_{ij}, j = \overline{1,k} = \overline{1,2}\,, i = \overline{1,m} = \overline{1,4}$.

За результатом, матриця C буде містити також 2 своїх рядка: $c_{ij}, i = \overline{1,k} = \overline{1,2}$, $j = \overline{1,m}$=$\overline{1,4}$.

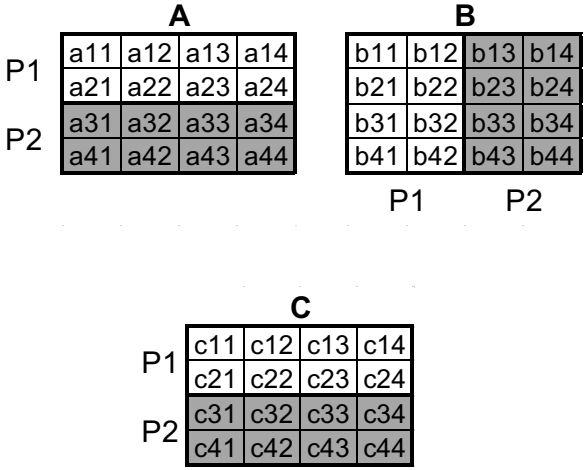


Рисунок 4.33 – Початкове розбиття матриць-аргументів та очікуване розбиття матриці-результату $C = A \times B$

Паралельний стрічковий алгоритм з вертикальним розподілом даних буде мати $p = 2$ ітерації. На першій ітерації процесори обчислюють ті елементи матриці C, для яких вони мають відповідні дані, на $P1 - c_{11}, c_{12}, c_{21}, c_{22}$ та $P2 - c_{33}, c_{34}, c_{43}, c_{44}$ (рис. 4.34).

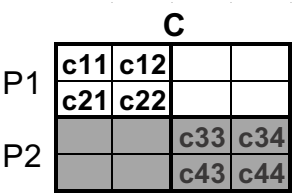


Рисунок 4.34 – Перша ітерація алгоритму, проміжна матриця-результат $C = A \times B$

Нагадуємо, що при завершенні обчислень, наприкінці кожної ітерації, стовпці матриці B повинні бути передані між процесорами. Мета цієї операції передачі даних, щоб у кожному процесорі з'явилися нові стовпці матриці B і тоді могли бути обчислені нові елементи матриці C. При цьому дана передача стовпців між процесорами повинна бути організована таким чином, щоб по завершенні алгоритму в кожному процесорі послідовно з'являлися усі стовпці матриці B. Для прикладу в результаті виконання цієї комунікаційної операції розташування смуг рядків матриці A не зміниться, а результат розташування смуг стовпців матриці B продемонстровано на рис. 4.35.

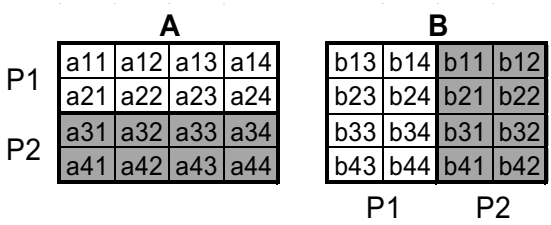


Рисунок 4.35 – Перша ітерація алгоритму, циклічне пересилання стовпців матриці B за зростанням номеру процесора (вправо)

Наступний крок алгоритму – друга ітерація, виконання обчислень нових, відсутніх елементів матриці-результату C, відповідно, на $P1 - c_{13}, c_{14}, c_{23}, c_{24}$ та $P2 - c_{31}, c_{32}, c_{41}, c_{42}$ (рис. 4.36).

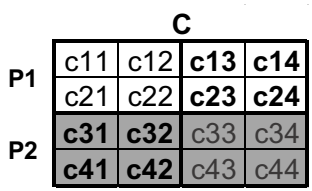


Рисунок 4.36 – Друга ітерація алгоритму, матриця-результат $C = A \times B$

Кількість елементів C, що обчислює кожен процесор послідовно, незалежно від номеру ітерації дорівнює: $k^2 = (m/p)^2 = 4$.

Таким чином, час на реалізацію обчислень за стрічковим вертикальним алгоритмом ММ визначається за наступним виразом:

$$T_{p,comp}^{Str-Column} = p[k^2(2m)] \cdot t_{op} = p\left[\frac{m^2}{p^2}(2m)\right] \cdot t_{op} =$$

$$= \frac{2m^3}{p} t_{op}. \qquad (4.33)$$

В свою чергу, час на реалізацію обмінів на кожній з p ітерацій алгоритму складається з часу на виконання циклічного пересилання рядків/смуг матриці B в сусідні процесори в рамках топології кільце. Для оцінки комунікаційної складності паралельних обчислень будемо припускати, що всі операції передачі даних між процесорами в ході однієї ітерації алгоритму можуть бути виконані паралельно.

Обсяг переданих даних між процесорами визначається розміром смуг і становить m/p рядків або стовпців довжини m. Загальна кількість паралельних операцій передачі повідомлень на одиницю менше числа ітерацій алгоритму (на останній ітерації передача даних немає сенсу).

За моделлю Хокні для режиму передачі повідомлень одна така комунікаційна операція потребує ($l = 1$): $T_{p,comm}^{Str-Column,i} = \left(t_s + \frac{m^2}{p} t_w\right) \cdot$

Отже, комунікаційна складова паралельного алгоритму ММ із стрічковим вертикальним розбиттям складає:

$$T_{p,comm}^{Str-Column} = (p-1)\left(t_s + \frac{m^2}{p} t_w\right) \cdot \qquad (4.34)$$

Таким чином, загальний час на паралельне виконання стрічкового вертикального матричного множення на ОС з p процесорами та топологією кільце, обчислюється за наступним виразом:

$$T_p^{Str-Column} = T_{p,comp}^{Str-Column} + T_{p,comm}^{Str-Column} =$$

$$= \frac{2m^3}{p} t_{op} + (p-1)\left(t_s + \frac{m^2}{p} t_w\right). \quad (4.35)$$

Коефіцієнти прискорення та ефективності, відповідно:

$$S_p^{Str-Column} = \frac{2m^3 t_{op}}{\frac{2m^3}{p} t_{op} + (p-1)\left(t_s + \frac{m^2}{p} t_w\right)}. \quad (4.36)$$

$$E_p^{Str-Column} = \frac{2m^3}{2m^3 t_{op} + (p-1)(pt_s + m^2 t_w)}. \quad (4.37)$$

Тоді, загальні накладні витрати на паралелізм для цього алгоритму розраховуються як:

$$T_0^{Str-Column} = (p-1)(p^2 t_s + m^2 t_w). \quad (4.38)$$

4.2.1.3 Демонстрація покрокового виконання алгоритму ММ із стрічковим вертикальним розбиттям даних

На наступних рисунках 4.37-4.44 наведена покрокова демонстрація виконання паралельного алгоритму ММ із стрічковим вертикальним розбиттям даних за умови, що циклічне пересилання рядків/смуг матриці B виконується в напрямку зростання номерів процесорів. Нехай для наочності: $m = 4, p = 4, k = 1$, тобто кожний процесор містить рівно один рядок матриць A, C та стовпець відповідно матриці B.

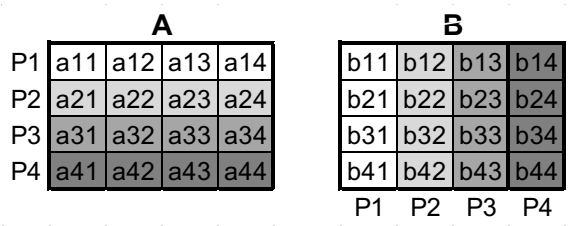
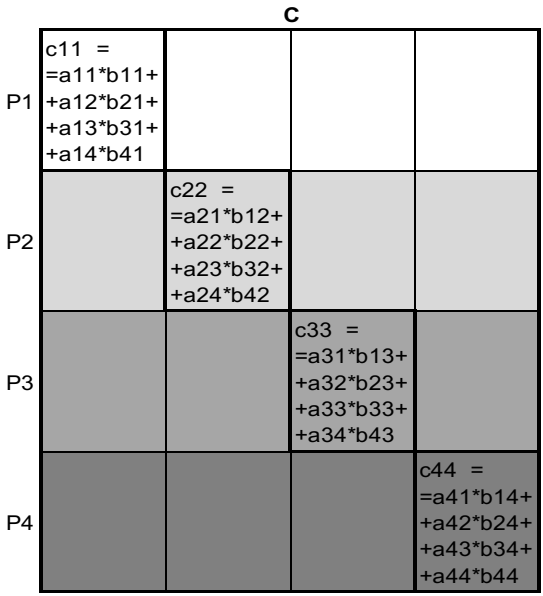
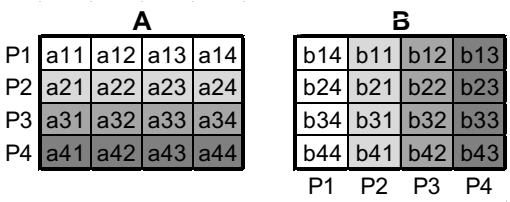


Рисунок 4.37 – Початкове розміщення даних паралельного стрічкового вертикального алгоритму ММ



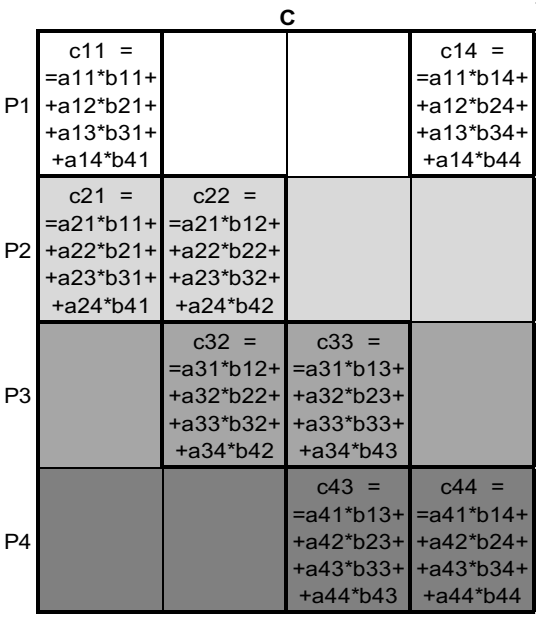
$$T_{p,comp}^{Str-Column,i=1} = 2\frac{m^3}{p^2}t_{op}$$

Рисунок 4.38 – Перша ітерація, $i = 1$, кожний процесор обчислює один скалярний добуток для отримання відповідних елементів матриці C: $c_{11}, c_{22}, c_{33}, c_{44}$



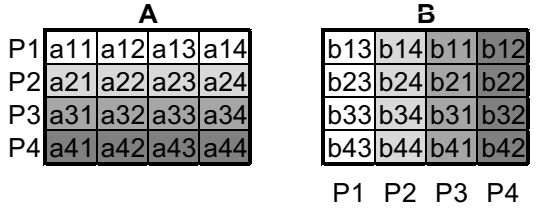
$$T_{p,comm}^{Str-Column,i=1} = t_s + \frac{m^2}{p}t_w$$

Рисунок 4.39 – Перша ітерація $i = 1$, виконання передачі даних: кожен процесор циклічно передає відповідний стовпець матриці B сусіду справа ($\text{P1} \longrightarrow \text{P2}, \text{P2} \longrightarrow \text{P3}, \text{P3} \longrightarrow \text{P4}, \text{P4} \longrightarrow \text{P1}$)



$$T_{p,comp}^{Str-Column,i=2} = 2\frac{m^3}{p^2}t_{op}$$

Рисунок 4.40 – Друга ітерація, $i = 2$, виконання скалярного добутку для обчислення відповідних елементів матриці C: c_{14}, c_{21}, c_{32}, c_{43}



$$T_{p,comm}^{Str-Column,i=2} = t_s + \frac{m^2}{p}t_w$$

Рисунок 4.41 – Друга ітерація, $i = 2$, виконання передачі даних: кожен процесор циклічно передає відповідний стовпець матриці B сусіду справа

	C			
P1	c11 = =a11*b11+ +a12*b21+ +a13*b31+ +a14*b41		c13 = =a11*b13+ +a12*b23+ +a13*b33+ +a14*b43	c14 = =a11*b11+ +a12*b21+ +a13*b31+ +a14*b41
P2	c21 = =a21*b11+ +a22*b21+ +a23*b31+ +a24*b41	c22 = =a21*b12+ +a22*b22+ +a23*b32+ +a24*b42		c24 = =a21*b14+ +a22*b24+ +a23*b34+ +a24*b44
P3	c31 = =a31*b11+ +a32*b21+ +a33*b31+ +a34*b41	c32 = =a31*b12+ +a32*b22+ +a33*b32+ +a34*b42	c33 = =a31*b13+ +a32*b23+ +a33*b33+ +a34*b43	
P4		c42 = =a41*b12+ +a42*b22+ +a43*b32+ +a44*b42	c43 = =a41*b13+ +a42*b23+ +a43*b33+ +a44*b43	c44 = =a41*b14+ +a42*b24+ +a43*b34+ +a44*b44

$$T_{p,comp}^{Str-Column,i=3} = 2\frac{m^3}{p^2}t_{op}$$

Рисунок 4.42 – Третя ітерація, виконання скалярного добутку для обчислення відповідних елементів матриці C: c_{13}, c_{24}, c_{31}, c_{42}

	A			
P1	a11	a12	a13	a14
P2	a21	a22	a23	a24
P3	a31	a32	a33	a34
P4	a41	a42	a43	a44

B			
b12	b13	b14	b11
b22	b23	b24	b21
b32	b33	b34	b31
b42	b43	b44	b41
P1	P2	P3	P4

$$T_{p,comm}^{Str-Column,i=3} = t_s + \frac{m^2}{p}t_w$$

Рисунок 4.43 – Третя ітерація, виконання передачі даних: кожен процесор циклічно передає відповідний стовпець матриці B сусіду справа

C

P1	c11 = =a11*b11+ +a12*b21+ +a13*b31+ +a14*b41	c12 = =a11*b12+ +a12*b22+ +a13*b32+ +a14*b42	c13 = =a11*b13+ +a12*b23+ +a13*b33+ +a14*b43	c14 = =a11*b11+ +a12*b21+ +a13*b31+ +a14*b41
P2	c21 = =a21*b11+ +a22*b21+ +a23*b31+ +a24*b41	c22 = =a21*b12+ +a22*b22+ +a23*b32+ +a24*b42	c23 = =a21*b13+ +a22*b23+ +a23*b33+ +a24*b43	c24 = =a21*b14+ +a22*b24+ +a23*b34+ +a24*b44
P3	c31 = =a31*b11+ +a32*b21+ +a33*b31+ +a34*b41	c32 = =a31*b12+ +a32*b22+ +a33*b32+ +a34*b42	c33 = =a31*b12+ +a32*b22+ +a33*b32+ +a34*b42	c34 = =a31*b13+ +a32*b23+ +a33*b33+ +a34*b43
P4	c41 = =a41*b11+ +a42*b21+ +a43*b31+ +a44*b41	c42 = =a41*b12+ +a42*b22+ +a43*b32+ +a44*b42	c43 = =a41*b13+ +a42*b23+ +a43*b33+ +a44*b43	c44 = =a41*b14+ +a42*b24+ +a43*b34+ +a44*b44

$$T_{p,comp}^{Str-Column,i=4} = 2\frac{m^3}{p^2}t_{op}$$

Рисунок 4.44 – Четверта ітерація, виконання скалярного добутку для обчислення відповідних елементів матриці C: c_{12}, c_{23}, c_{34}, c_{41}

Таким чином, за чотири ітерації алгоритму стрічкового вертикального множення матриць, отримано усі елементи матриці C. На останній четвертій ітерації операція передачі даних не виконується тому, що в ній не має потреби.

4.2.2 Паралельний стрічковий алгоритм ММ з горизонтальним розбиттям

Другий варіант стрічкового паралельного алгоритму ММ використовує так зване горизонтальне розбиття даних [2-4]. Нагадаємо, що цей факт стосується тільки розміщення рядків/смуг матриці B.

4.2.2.1 Загальний опис стрічкового вертикального алгоритму ММ

Загальна схема розподілу даних для паралельного стрічкового горизонтального алгоритму ММ наведена на рис. 4.45. На відміну від попереднього стрічкового алгоритму ММ, в алгоритмі з горизонтальним розбиттям на кожному процесорі розташовуються не стовпці, а рядки матриці B. Тоді, добуток даних на кожному процесорі зводиться не до скалярного множення його наявних векторів, а до поелементного множення. За результатом подібного множення в кожному процесорі виходить рядок часткових результатів матриці C.

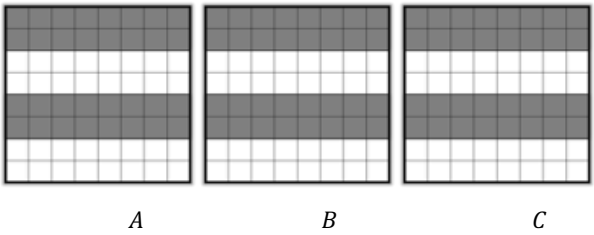


Рисунок 4.45 – Загальна схема розміщення даних для паралельного алгоритму ММ із стрічковим горизонтальним розбиттям

За алгоритмом спочатку процесор з номером i містить елементи i-го рядка (горизонтальної смуги) матриць A та B. Елементи рядка (горизонтальної смуги) добутку $C = A \times B$ обнуляються.

Далі виконується циклічний процес, кількість ітерацій дорівнює числу процесорів ОС ($i = 1, \dots, p$):

1) поелементне множення даних відповідних рядків матриць A та B з метою отримання часткових сум для елементів матриці C, підсумовування знову одержаних значень з раніше обчисленими результатами;

2) виконання циклічної операції передачі рядків/смуг матриці B між процесорами з використанням кільцевої структури в напрямку зростання або убування номерів процесорів.

4.2.2.2 Динамічні характеристики стрічкового горизонтального алгоритму ММ

Динамічні характеристики стрічкового горизонтального алгоритму ММ, як і для попереднього алгоритму, порахуємо в умовах його відображення на паралельну систему з p процесорами та кільцевою топологією. Якщо порядок перемножуваних та результуючої матриць більш ніж число процесорів, то ширина смуги елементів на одному процесорі дорівнює: $k = \frac{m}{p}, m : p$, вважаємо, що k – ціле число. Тоді, кількість елементів c_{ij}, що будуть отримані на одному довільному процесорі, P_i: $m \cdot k$.

Наприклад, нехай порядок матриць $m = 4$, кількість процесорів становить $p = 2$, тоді довжина смуги дорівнює 2 (рис. 4.46-4.49). Спочатку кожен процесор містить 2 рядка матриці A: $a_{ij}, i = \overline{1,k} = \overline{1,2}, j = \overline{1,m}$=$\overline{1,4}$ та 2 рядка матриці B: $a_{ij}, j = \overline{1,k} = \overline{1,2}, i = \overline{1,m}$=$\overline{1,4}$. За результатом, матриця C буде містити також 2 своїх рядка: $c_{ij}, i = \overline{1,k} = \overline{1,2}, j = \overline{1,m}$=$\overline{1,4}$.

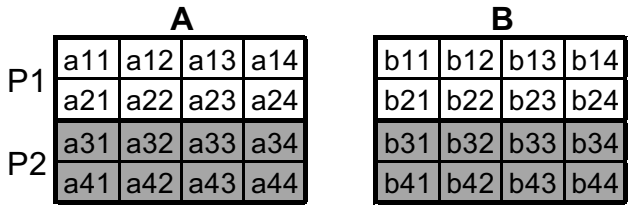


Рисунок 4.46 – Початкове розбиття матриць-аргументів

$$C = A \times B$$

Паралельний стрічковий алгоритм з горизонтальним розподілом даних буде мати $p = 2$ ітерації. На першій ітерації процесори обчислюють часткові суми для тих елементів матриці C, для яких вони мають відповідні дані, на P1 – $c_{11}, c_{12}, c_{13}, c_{14}, c_{21}, c_{22}, c_{23}, c_{24}$ та P2 – $c_{31}, c_{32}, c_{33}, c_{34}, c_{41}, c_{42}, c_{43}, c_{44}$ (рис. 4.47).

C

P1	c11 = =a11*b11+ +a12*b21	c12 = =a11*b12+ +a12*b22	c13 = =a11*b13+ +a12*b23	c14 = =a11*b14+ +a12*b24
	c21 = =a22*b21+ +a21*b11	c22 = =a22*b22+ +a21*b12	c23 = =a22*b23+ +a21*b13	c24 = =a22*b24+ +a21*b14
P2	c31 = =a33*b31+ +a34*b41	c32 = =a33*b32+ +a34*b42	c33 = =a33*b33+ +a34*b43	c34 = =a33*b34+ +a34*b44
	c41 = =a44*b41+ +a43*b31	c42 = =a44*b42+ +a43*b32	c43 = =a44*b43+ +a43*b33	c44 = =a44*b44+ +a43*b34

Рисунок 4.47 – Перша ітерація алгоритму,
проміжна матриця-результат $C = A \times B$

A

P1	a11	a12	a13	a14
	a21	a22	a23	a24
P2	a31	a32	a33	a34
	a41	a42	a43	a44

B

b31	b32	b33	b34
b41	b42	b43	b44
b11	b12	b13	b14
b21	b22	b23	b24

Рисунок 4.48 – Перша ітерація алгоритму,
циклічне пересилання рядків матриці B
за зростанням номеру процесора (вниз)

C

P1	c11 = =a11*b11+ +a12*b21+ +a13*b31+ +a14*b41	c12 = =a11*b12+ +a12*b22+ +a13*b32+ +a14*b42	c13 = =a11*b13+ +a12*b23+ +a13*b33+ +a14*b43	c14 = =a11*b11+ +a12*b21+ +a13*b31+ +a14*b41
	c21 = =a21*b11+ +a22*b21+ +a23*b31+ +a24*b41	c22 = =a21*b12+ +a22*b22+ +a23*b32+ +a24*b42	c23 = =a21*b13+ +a22*b23+ +a23*b33+ +a24*b43	c24 = =a21*b14+ +a22*b24+ +a23*b34+ +a24*b44
P2	c31 = =a31*b11+ +a32*b21+ +a33*b31+ +a34*b41	c32 = =a31*b12+ +a32*b22+ +a33*b32+ +a34*b42	c33 = =a31*b12+ +a32*b22+ +a33*b32+ +a34*b42	c34 = =a31*b13+ +a32*b23+ +a33*b33+ +a34*b43
	c41 = =a41*b11+ +a42*b21+ +a43*b31+ +a44*b41	c42 = =a41*b12+ +a42*b22+ +a43*b32+ +a44*b42	c43 = =a41*b13+ +a42*b23+ +a43*b33+ +a44*b43	c44 = =a41*b14+ +a42*b24+ +a43*b34+ +a44*b44

Рисунок 4.49 – Друга ітерація алгоритму, заключна матриця-результат $C = A \times B$

Час паралельного виконання обчислень на кожній з ітерацій складається з часу на реалізацію операції часткового добутку для кожного з $m \cdot \frac{m}{p}$ елементів матриці C та підсумовування одержаних значень з раніше обчисленими результатами для тих же елементів:

$$T_{p,comp}^{Str-Row,i} = mk(kt_{mul}+kt_{ad}) = 2\frac{m^3}{p^2}t_{op}. \quad (4.39)$$

Тоді, загальний час на обчислення на усіх ітераціях алгоритму розраховується, як:

$$T_{p,comp}^{Str-Row} = p \cdot 2\frac{m^3}{p^2} \cdot t_{op} = 2\frac{m^3}{p} t_{op}. \quad (4.40)$$

Обсяг даних, що передаються, та їх загальна кількість співпадають з такими ж характеристиками попереднього стрічкового алгоритму ММ з вертикальним розбиттям, тому одна така комунікаційна операція потребує:

$$T_{p,comm}^{Str-Row,i} = t_s + \frac{m^2}{p} t_w \cdot$$

Таким чином, повна комунікаційна складова паралельного алгоритму ММ із стрічковим горизонтальним розбиттям складає:

$$T_{p,comm}^{Str-Row} = (p-1)\left(t_s + \frac{m^2}{p} t_w\right) \cdot \quad (4.41)$$

Час паралельного виконання даного алгоритму містить два доданки, а саме, час на обчислення та передачу даних:

$$T_p^{Str-Row} = T_{p,comp}^{Str-Row} + T_{p,comm}^{Str-Row} =$$

$$= 2\frac{m^3}{p} \cdot t_{op} + (p-1)\left(t_s + \frac{m^2}{p} t_w\right). \quad (4.42)$$

Коефіцієнти прискорення та ефективності, відповідно, обчислюються за наступними виразами:

$$S_p^{Str-Row} = \frac{2m^3 t_{op}}{2\frac{m^3}{p} t_{op} + (p-1)\left(t_s + \frac{m^2}{p} t_w\right)}. \quad (4.43)$$

$$E_p^{Str-Row} = \frac{2m^3}{2m^3 t_{op} + (p-1)(pt_s + m^2 t_w)}. \quad (4.44)$$

Тоді, загальні накладні витрати на паралелізм для цього алгоритму розраховуються як:

$$T_0^{Str-Row} = (p-1)(pt_s + m^2 t_w). \quad (4.45)$$

4.2.2.3 Демонстрація покрокового виконання алгоритму ММ із стрічковим горизонтальним розбиттям даних

На наступних рисунках 4.50-4.57 традиційно наведена покрокова демонстрація виконання паралельного алгоритму ММ із стрічковим горизонтальним розбиттям даних за умови, що циклічне пересилання рядків/смуг матриці B виконується в напрямку зростання номерів процесорів (вниз циклічно). Нехай для наочності: $m = 4, p = 4, k = 1$, тобто кожний процесор містить по одному рядку відповідної матриці.

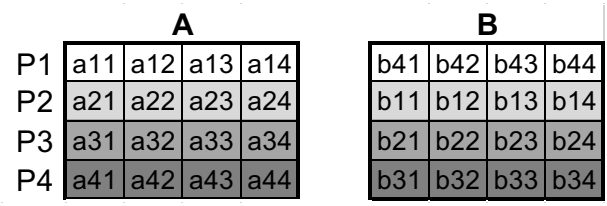
	A				B			
P1	a11	a12	a13	a14	b11	b12	b13	b14
P2	a21	a22	a23	a24	b21	b22	b23	b24
P3	a31	a32	a33	a34	b31	b32	b33	b34
P4	a41	a42	a43	a44	b41	b42	b43	b44

Рисунок 4.50 – Початкове розміщення даних паралельного стрічкового горизонтального алгоритму ММ

	C			
P1	c11 = =a11*b11	c12 = =a11*b12	c13 = =a11*b13	c14 = =a11*b14
P2	c21 = =a22*b21	c22 = =a22*b22	c23 = =a22*b23	c24 = =a22*b24
P3	c31 = =a33*b31	c32 = =a33*b32	c33 = = a33*b33	c34 = =a33*b34
P4	c41 = =a44*b41	c42 = =a44*b42	c43 = =a44*b43	c44 = =a44*b44

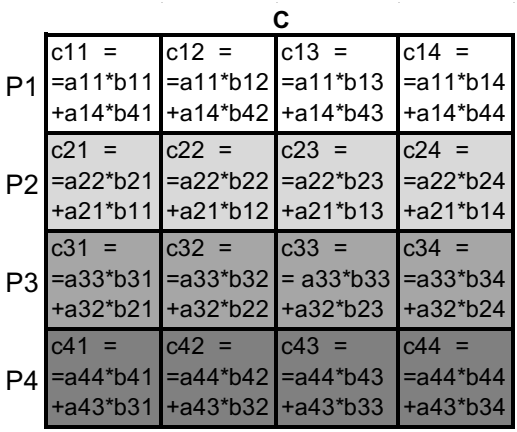
$$T_{p,comp}^{Str-Row,i=1} = 2\frac{m^3}{p^2}t_{op}$$

Рисунок 4.51 – Перша ітерація, $i = 1$, кожний процесор обчислює часткові суми відповідних елементів (на першій ітерації часткові суми додаються до 0)



$$T_{p,comm}^{Str-Row,i=1} = t_s + \frac{m^2}{p} t_w$$

Рисунок 4.52 – Перша ітерація, $i = 1$, виконання передачі даних: кожен процесор j циклічно передає відповідний рядок матриці B процесору з номером $(j \ mod \ p) + 1$
$(\mathrm{P1} \longrightarrow \mathrm{P2}, \mathrm{P2} \longrightarrow \mathrm{P3}, \mathrm{P3} \longrightarrow \mathrm{P4}, \mathrm{P4} \longrightarrow \mathrm{P1})$



$$T_{p,comp}^{Str-Row,i=2} = 2\frac{m^3}{p^2} t_{op}$$

Рисунок 4.53 – Друга ітерація, $i = 2$, кожний процесор обчислює часткові суми і додає їх до відповідних елементів

	A					B			
P1	a11	a12	a13	a14		b31	b32	b33	b34
P2	a21	a22	a23	a24		b41	b42	b43	b44
P3	a31	a32	a33	a34		b11	b12	b13	b14
P4	a41	a42	a43	a44		b21	b22	b23	b24

$$T_{p,comm}^{Str-Row,i=2} = t_s + \frac{m^2}{p} t_w$$

Рисунок 4.54 – Друга ітерація, $i = 2$, виконання передачі даних: кожен процесор j циклічно передає відповідний рядок матриці B процесору з номером $(j \ mod \ p) + 1$
$(\text{P1} \longrightarrow \text{P2}, \text{P2} \longrightarrow \text{P3}, \text{P3} \longrightarrow \text{P4}, \text{P4} \longrightarrow \text{P1})$

	C			
P1	c11 = =a11*b11 +a14*b41 +a13*b31	c12 = =a11*b12 +a14*b42 +a13*b32	c13 = =a11*b13 +a14*b43 +a13*b33	c14 = =a11*b14 +a14*b44 +a13*b34
P2	c21 = =a22*b21 +a21*b11 +a24*b41	c22 = =a22*b22 +a21*b12 +a24*b42	c23 = =a22*b23 +a21*b13 +a24*b43	c24 = =a22*b24 +a21*b14 +a24*b44
P3	c31 = =a33*b31 +a32*b21 +a31*b11	c32 = =a33*b32 +a32*b22 +a31*b12	c33 = = a33*b33 +a32*b23 +a31*b13	c34 = =a33*b34 +a32*b24 +a31*b14
P4	c41 = =a44*b41 +a43*b31 +a42*b21	c42 = =a44*b42 +a43*b32 +a42*b22	c43 = =a44*b43 +a43*b33 +a42*b23	c44 = =a44*b44 +a43*b34 +a42*b24

$$T_{p,comp}^{Str-Row,i=3} = 2\frac{m^3}{p^2} t_{op}$$

Рисунок 4.55 – Третя ітерація, $i = 3$, кожний процесор обчислює часткові суми і додає їх до відповідних елементів

	A				B			
P1	a11	a12	a13	a14	b21	b22	b23	b24
P2	a21	a22	a23	a24	b31	b32	b33	b34
P3	a31	a32	a33	a34	b41	b42	b43	b44
P4	a41	a42	a43	a44	b11	b12	b13	b14

$$T_{p,comm}^{Str-Row,i=3} = \left(t_s + \frac{m^2}{p} t_w\right)$$

Рисунок 4.56 – Третя ітерація, $i = 3$, виконання передачі даних: кожен процесор j циклічно передає відповідний рядок матриці B процесору з номером $(j \ mod \ p) + 1$

$(\text{P1} \longrightarrow \text{P2}, \text{P2} \longrightarrow \text{P3}, \text{P3} \longrightarrow \text{P4}, \text{P4} \longrightarrow \text{P1})$

	C			
P1	c11 = =a11*b11 +a14*b41 +a13*b31 +a12*b21	c12 = =a11*b12 +a14*b42 +a13*b32 +a12*b22	c13 = =a11*b13 +a14*b43 +a13*b33 +a12*b23	c14 = =a11*b14 +a14*b44 +a13*b34 +a12*b24
P2	c21 = =a22*b21 +a21*b11 +a24*b41 +a23*b31	c22 = =a22*b22 +a21*b12 +a24*b42 +a23*b32	c23 = =a22*b23 +a21*b13 +a24*b43 +a23*b33	c24 = =a22*b24 +a21*b14 +a24*b44 +a23*b34
P3	c31 = =a33*b31 +a32*b21 +a31*b11 +a34*b41	c32 = =a33*b32 +a32*b22 +a31*b12 +a34*b42	c33 = = a33*b33 +a32*b23 +a31*b13 +a34*b43	c34 = =a33*b34 +a32*b24 +a31*b14 +a34*b11
P4	c41 = =a44*b41 +a43*b31 +a42*b21 +a41*b11	c42 = =a44*b42 +a43*b32 +a42*b22 +a41*b12	c43 = =a44*b43 +a43*b33 +a42*b23 +a41*b13	c44 = =a44*b44 +a43*b34 +a42*b24 +a41*b14

$$T_{p,comp}^{Str-Row,i=4} = 2\frac{m^3}{p^2} t_{op}$$

Рисунок 4.57 – Четверта ітерація, $i = 4$, кожний процесор обчислює часткові суми і додає їх до відповідних елементів

4.3 Швидке рекурсивне множення матриць і поліалгоритми на його основі

Матричний добуток (МД) – домінуюча обчислювальна частина багатьох методів розв'язання реальних динамічних задач. Тому прискорення паралельної реалізації цієї базової операції лінійної алгебри, означає зменшення терміну виконання методу вирішення задачі в цілому. При величезній різноманітності методів обчислення матричного добутку [2-3] для щільнозаповнених матриць є два принципово різні класи послідовних алгоритмів: традиційні та рекурсивні методи швидкого множення Штрассена [40-45]. У оригіналі алгоритм Штрассена – це алгоритм множення блокових матриць половинного розміру, де кожен блок квадратний, тобто розміри матриць є парними числами. Метод Штрассена-Копперсміта-Віноградa [42] наведено на рис. 4.58.

$$
\begin{array}{lll}
S_1 = A_{21} + A_{22}, & M_1 = S_2 S_6, & T_1 = M_1 + M_3, \\
S_2 = S_1 - A_{11}, & M_2 = A_{11} B_{11}, & T_2 = T_1 + M_4, \\
S_3 = A_{21} - A_{12}, & M_3 = A_{12} B_{21}, & T_3 = M_5 + M_{6,} \\
S_4 = A_{12} - S_2, & M_4 = S_3 S_7, & C_{11} = M_2 + M_3, \\
S_5 = B_{12} - B_{11}, & M_5 = S_1 S_5, & C_{12} = T_1 + T_3, \\
S_6 = B_{22} - S_5, & M_6 = S_4 B_{22}, & C_{21} = T_2 - M_7, \\
S_7 = B_{22} - B_{12,} & M_7 = A_{22} S_8, & C_{22} = T_2 + M_5, \\
S_8 = S_6 - B_{12}, & &
\end{array}
$$

$$
A = \left\langle \frac{A_{11}}{A_{21}} \middle| \frac{A_{12}}{A_{22}} \right\rangle, \; B = \left\langle \frac{B_{11}}{B_{21}} \middle| \frac{B_{12}}{B_{22}} \right\rangle, \; C = \left\langle \frac{C_{11}}{C_{21}} \middle| \frac{C_{12}}{C_{22}} \right\rangle.
$$

Рисунок 4.58 – Метод швидкого рекурсивного множення матриць Штрассена- Копперсміта-Віноградa

Далі, якщо це буде не принципово, будемо називати алгоритм – алгоритмом Штрассена. Ідея Штрассена може бути застосована рекурсивно для знаходження добутків блоків матриць $M_i, i = 1, \ldots, 7$.

Якщо початкові матриці A і B порядку m, то теоретично алгоритм швидкого множення можна використовувати багато разів, отримуючи на самому нижньому рівні рекурсії блоки розміру 1×1. Проте немає необхідності опускатися вниз до рівня блоків одиничного порядку. При досить малих розмірах блоку ($m \le m_{min}$) може виявитися корисним обчислення блоків із використанням стандартного алгоритму МД. Підсумовуючи викладене, наведемо обчислювальну схему послідовного алгоритму Штрассена (рис. 4.59).

$$
\begin{aligned}
&function\ C = Strassen(A, B, m_{min})\\
&m = row(A)\\
&if\ m \le m_{min}\ then\\
&\quad C = A \underset{ordinary}{\times} B\\
&else\\
&\quad n = m/2;\ u = 1:n;\ v = n+1:m;\\
&\quad M_1 = Strassen(A(u,u) + A(v,v), B(u,u) + B(v,v), m_{min})\\
&\quad M_2 = Strassen(A(v,u) + A(v,v), B(u,u), m_{min})\\
&\quad M_3 = Strassen(A(u,u), B(u,v) - B(v,v), m_{min})\\
&\quad M_4 = Strassen(A(v,v), B(v,u) - B(u,u), m_{min})\\
&\quad M_5 = Strassen(A(u,u) + A(u,v), B(v,v), m_{min})\\
&\quad M_6 = Strassen(A(v,u) - A(u,u), B(u,u) + B(u,v), m_{min})\\
&\quad M_7 = Strassen(A(u,v) - A(v,v), B(v,u) + B(v,v), m_{min})\\
&\quad C(u,u) = M_1 + M_4 - M_5 + M_7\\
&\quad C(u,v) = M_3 + M_5\\
&\quad C(v,u) = M_2 + M_4\\
&\quad C(v,v) = M_1 + M_3 - M_2 + M_6\\
&end\\
&end\ Strassen
\end{aligned}
$$

Рисунок 4.59 – Обчислювальна схема послідовного рекурсивного алгоритму Штрассена

Обчислювальна складність запропонованої схеми алгоритму швидкого множення визначається функцією порядку вхідних матриць і мінімального порядку перемножуваних блоків: m, m_{min}.

Нехай при обчисленні матричного множення алгоритм Штрассена рекурсивно викликався d раз, тоді порядок матриць,що перемножуються, дорівнює $m_{min} = m/2^d$. На першому кроці алгоритм передбачає 7 звернень до самого себе з матрицями порядку половинного порядку: $m/2$ та 15 операцій типу алгебраїчне додавання матриць того ж порядку. Далі йде розгортка рекурсії до досягнення мінімального розміру блоку й множення блоків за традиційним алгоритмом матричного множення.

Час реалізації послідовного алгоритму при цьому складає:

$$T_1^{Str}(m) = \left[2\left(\frac{7}{8}\right)^d m^3 + \frac{15}{4}m^2\left(1 + \frac{7}{4} + \frac{7^2}{4^2} + \cdots + \frac{7^{d-1}}{4^{d-1}}\right)\right] t_{op.} \qquad (4.46$$

Відомий паралельний алгоритм класичного методу Штрассена був застосований на CRAY-2 і показав добрі результати в порівнянні з традиційними алгоритмами МД [43-44]. Істотним недоліком цього алгоритму є відсутність масштабованості, оскільки необхідна кількість процесорів для нього кратна сімці (по кількості множень блоків матриць $M_i, i = \overline{1,7}$). Це обмеження є достатньо жорстким і не природним для більшості паралельних архітектур.

Для подолання вказаного недоліку скористаємося *поліалгоритмічним* підходом. Під *поліалгоритмом* мається на увазі комбінація двох або більше алгоритмів в одну обчислювальну схему, що реалізує поставлену задачу з метою скорочення обчислювальних і/або ємкісних витрат [16-17]. Алгоритм швидкого множення рекурсивний, тому є можливість побудувати поліалгоритм з деякого традиційного алгоритму множення матриць на верхньому рівні рекурсії і методу Штрассена-Віноградa на нижньому рівні, і навпаки. Алгоритм Штрассена є блоковим, тому природно комбінувати його з алгоритмом матричного добутку, що використовує відповідне розбиття даних.

Побудуємо поліалгоритм із блокового систолічного алгоритму матричного множення, алгоритма Кеннона, між процесорами і серії вживань рекурсивного методу Штрассена на кожному процесорі (рис. 4.60).

Нехай є замкнута двовимірна сітка процесорів розміру $p \times p$, початкові матриці A і B, що розподілені на блоки $k = \lceil m/p \rceil$, кількість блоків дорівнює $q^2 = p^2$. Блоки початкових матриць і результату з координатами $< i, j >$ зберігаються у відповідному процесорі з тими ж координатами. Як і раніше, передбачаємо, що $m \, mod \, p = 0$, інакше використовується доповнення нульовими елементами.

Спочатку, за обчислювальною схемою блокового систолічного множення, виконується косе зрушення вліво по рядках для блоків матриці A і косе зрушення вгору по стовпцях для блоків матриці B. На кожному з p кроків алгоритму проводиться множення блоків матриць A і B, що зберігаються в процесорі з номером $< i, j >$, і складання із вже обчисленим значенням блоку матриці C_{ij}, розташованим на цьому ж процесорі. Для першого кроку це значення дорівнює нульовому блоку. Потім проводиться одиночне зрушення вліво по рядках паралельно для всіх блоків матриці A: $A_{ij} \longleftarrow A_{ij+1}$ і одиночне зрушення вгору по стовпцях також паралельно для всіх блоків матриці B: $B_{ij} \longleftarrow B_{i+1j}$.

Множення блоків матриць виконується усередині одного процесора за рекурсивним алгоритмом Штрассена-Віноградa, що дозволяє уникнути додаткових пересилок даних. На нижньому рівні рекурсії застосовується стандартний алгоритм множення матриць, глибина рекурсії дорівнює: d. Поліалгоритм Кеннона-Штрассена має достатньо великий ступінь паралелізму рівний розміру вхідних матриць $Dop = m$, і є таким, що масштабується для будь-якого числа процесорів і будь-якого порядку матриць, що перемножуються.

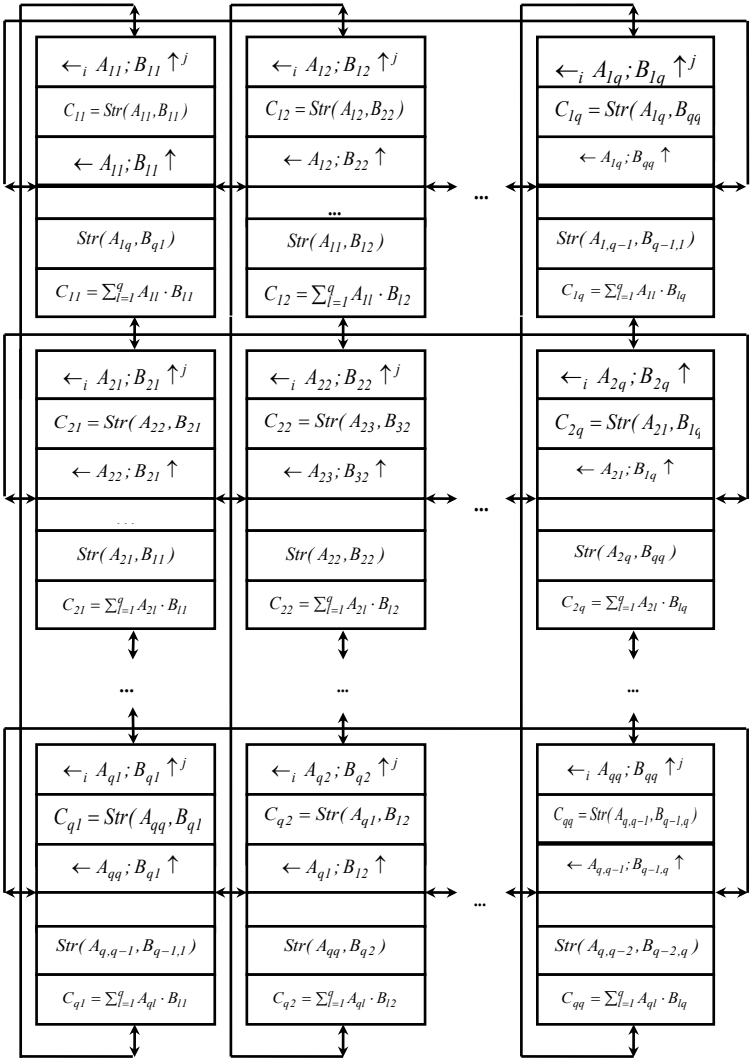


Рисунок 4.60 – Обчислювальна схема комбінованого алгоритму множення матриць: алгоритм Кеннона + алгоритм Штрассена-Віноградa для топології 2D-тор

Час реалізації блокового рекурсивно-систолічного алгоритму включає час виконання арифметичних і обмінних операцій:

$$T_p^{Cannon-Str}=T_{p,comp}^{Cannon-Str}+T_{p,comm}^{Cannon-Str}. \quad (4.47)$$

При цьому час виконання обчислень за схемою рис. 4.60 дорівнює:

$$T_{p,comp}^{Cannon-Str}=q\left(T_{A_{ij}\underset{Str}{\times}B_{ij}}+T_{A_{ij}+B_{ij}}\right), \quad (4.48)$$

де

1) $T_{A_{ij}\underset{Str}{\times}B_{ij}}$ – час множення блоків матриць порядку $k=m/p$ за рекурсивним алгоритмом Штрассена-Віноградa;

2) $T_{A_{ij}+B_{ij}}=k^2t_{ad}=(m^2/p^2)t_{op}$ – час складання блоків матриць того ж розміру.

Час на множення блоків, що виконується за алгоритмом Штрассена-Віноградa, задовольняє рекурентному співвідношенню:

$$T_{A_{ij}\underset{Str}{\times}B_{ij}}=T_p^{Str}(k)=7T_p^{Str}\left(\frac{k}{2}\right)+15\left(\frac{k}{2}\right)^2t_{op.}. \quad (4.49)$$

Час реалізації алгоритму для матриць порядку $k/2,\dots,k/2^{d-1}$:

$$T_p^{Str}\left(\frac{k}{2}\right)=7T_p^{Str}\left(\frac{k}{4}\right)+15\left(\frac{k}{4}\right)^2t_{op},$$

…

$$T_p^{Str}\left(\frac{k}{2^{d-1}}\right)=7T_p^{Str}\left(\frac{k}{2^d}\right)+15\left(\frac{k}{2^d}\right)^2t_{op}.$$

Виконавши підстановки та елементарні перетворення, отримаємо:

$$T_p^{Str}(k)=7^dT_p^{Str}\left(\frac{k}{2^d}\right)+\frac{15}{4}k^2\left(1+\frac{7}{4}+\frac{7^2}{4^2}+\cdots+\frac{7^{d-1}}{4^{d-1}}\right)t_{op.} \quad (4.50)$$

Вираз (4.5) дає обчислювальну складність тільки для розгортки рекурсії. Розглянемо внутрішній рівень рекурсії, оскільки саме там виконуються операції множення блоків матриць мінімального порядку за стандартним, не рекурсивним методом матричного множення:

$$T_p^{Str}\left(\frac{k}{2^d}\right) = k_{min}^3(t_{mul} + t_{ad}) = 2\left(\frac{k}{2^d}\right)^3 t_{op.} \quad (4.51)$$

Таким чином, час реалізації швидкого рекурсивного множення блоків матриць складає:

$$T_p^{Str}(k) = \left[2\left(\frac{7}{8}\right)^d k^3 + \frac{15}{4}k^2\left(\frac{1-\frac{7^d}{4^d}}{1-\frac{7}{4}}\right)\right] t_{op.}. \quad (4.52)$$

Тоді загальний час виконання арифметичних операцій для комбінації блокового систолічного алгоритму і рекурсивного матричного множення дорівнює:

$$T_{p,comp}^{Cannon-Str} = q\left(T_p^{Str}\left(\frac{m}{p}\right) + T_{A_{ij}+B_{ij}}\right) =$$

$$= \left[2\left(\frac{7}{8}\right)^d \frac{m^3}{p^2} + \left(5\frac{7^d}{4^d} - 4\right)\frac{m^2}{p}\right] t_{op.} \quad (4.53)$$

Час обмінних операцій для описаної схеми (рис. 4.60) визначається, як і для блокового систолічного алгоритму:

$$T_{p,comm}^{Cannon-Str} = 4(p-1)\left(t_s + \frac{m^3}{p^2} t_w\right). \quad (4.54)$$

Очевидно, що динамічні характеристики паралельних алгоритмів матричного добутку залежать від співвідношення між кількістю процесорів та порядком матриць. Для рекурсивного алгоритму множення матриць додатковим істотним параметром є також величина глибини рекурсії, d (рис. 4.61-4.64).

```
(* Визначення оптимального значення глибини рекурсії *)

Round[Simplify[Log[2, 2 * m * (Log[8 / 7])] - Log[2, (5 * Log[7 / 4])]]]

d_opt = Round[(Log[2 m Log[8/7]] - Log[5 Log[7/4]]) / Log[2]]

m_opt = N[m / 2^d_opt] = 8
```

Рисунок 4.61 – Визначення оптимального значення глибини рекурсії

Визначення оптимальної величини глибини рекурсії, а, отже, й розміру мінімального оброблюваного блоку матриць проводилося у пакеті *Mathematica*. Для цього необхідно знайти значення, що доставляє мінімум функції T_1^{Str} , причому діапазон зміни глибини рекурсії обмежений наступною нерівністю:

$$m_{opt} = m_{min} = (m/2^d) \geq 1 \implies m \geq 2^d \implies d \leq \log_2 m. \qquad (4.55)$$

Очевидно, що при великих розмірах задачі кожна з функцій $V(d) = T_1^{Str}/t_{op}$ рисунку 4.62 має явно виражений локальний мінімум, відношення ж розміру системи до глибини рекурсії при цьому залишається постійним. Наприклад, при $m = 1024$, $d_{min} = 7$ і $\frac{m}{2^{d_{min}}} = 8$, а при $m = 256$ маємо $d_{min} = 5 \implies m/2^{d_{min}} = 8$. У той же час при $m = 16$ мінімальна глибина рекурсії дорівнює 1, тобто для досягнення мінімуму обчислювальних витрат необхідний всього один крок розгортки рекурсії, а для $m = 8$ розгортки рекурсії немає, оскільки $d_{min} = 0$.

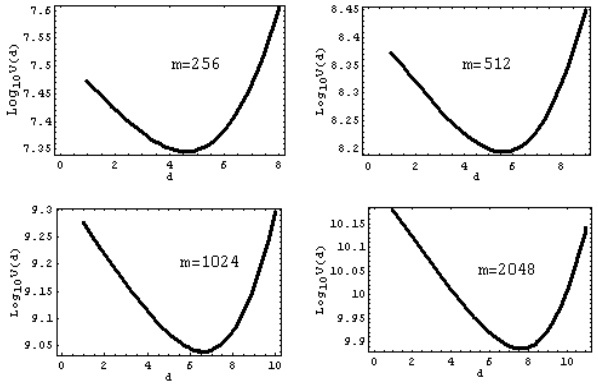


Рисунок 4.62 – Обчислювальна складність рекурсивно-систолічного алгоритму від глибини рекурсії у логарифмічному масштабі

Аналітичні результати підтверджуються результатами експериментів, на рис. 4.63 приведено залежність часу виконання матричного добутку від величини блоку матриць, що перемножуються.

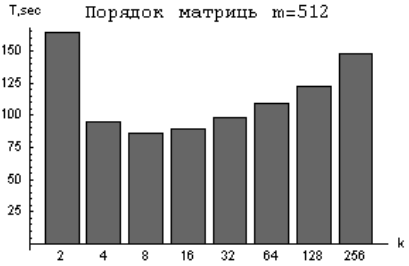


Рисунок 4.63 – Експериментальне визначення величини мінімального блоку для рекурсивно-систолічного алгоритму множення матриць

Для паралельного варіанту запропонованого алгоритму блокового рекурсивно-систолічного множення окрім глибини рекурсії необхідно враховувати співвідношення між кількістю процесорів p і порядком перемножуваних матриць m (рис. 4.64). Обмеження на діапазон зміни глибини рекурсії визначаються наступною нерівністю:

$$m_{min} = \frac{m}{2^d p} \geq 1 \implies \frac{m}{p} \geq 2^d \implies d \leq log_2 \frac{m}{p}.$$

```
(* Визначення оптимального значення глибини рекурсії *)

pd_opt = Round[Simplify[Log[2, 2 * m / p * (Log[8 / 7])] - Log[2, (5 * Log[7 / 4])]]]

pd_opt = Round[(Log[2 m Log[8/7] / p] - Log[5 Log[7/4]]) / Log[2]]

(m / p)_opt = 8 * 2^pd_opt
```

Рисунок 4.64 – Визначення величини оптимальної глибини рекурсії для паралельного блокового рекурсивно-систолічного алгоритму

При оцінюванні глибини рекурсії тимчасові характеристики приведені до часу реалізації однієї операції з рухомою точкою, для різних мультикомп'ютерів величина оптимальної глибини рекурсії має бути поправлена з урахуванням цієї машинно-залежної константи.

Аналіз математичних виразів, що характеризують виконання паралельних алгоритмів, а також проведений чисельний експеримент, дозволяють зробити наступні висновки (рис. 4.65-4.66):

1) запропонована паралельна обчислювальна схема поліалгорітма на основі швидкого множення Штрассена-Віноградy має кращі характеристики паралелізму для матриць великих розмірностей при постійній ширині процесорної сітки і певному рівні рекурсії;

2) коефіцієнти прискорення і ефективності зростають з ростом розмірності задачі: $m \uparrow \Longrightarrow S_p^{Cannon-Str} \uparrow \Longrightarrow E_p^{Cannon-Str}$ і зменшуються з ростом числа процесорів: $p \downarrow \Longrightarrow S_p^{Cannon-Str} \uparrow \Longrightarrow E_p^{Cannon-Str}$;

3) високий рівень рекурсії негативно позначається, як на часу виконання, так і на обсязі використаної пам'яті.

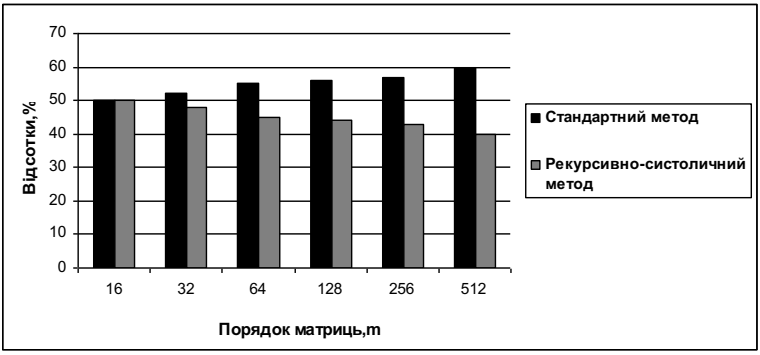


Рисунок 4.65 – Співвідношення часу реалізації паралельних методів стандартного і рекурсивно-систолічного матричного добутку

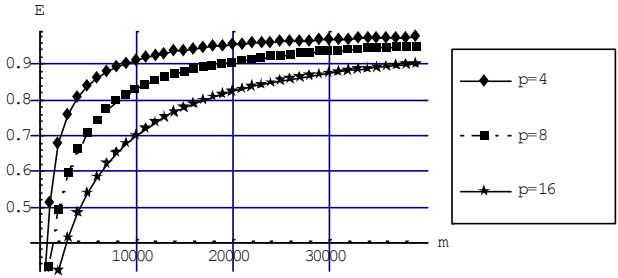


Рисунок 4.66 – Коефіцієнти ефективності паралельного блокового рекурсивно-систолічного методу для мультикомп'ютерів із шириной процесорної сітки, рівною p

Крім того, рекурсивний характер розробленого алгоритму дозволяє звести багатовимірну початкову задачу до вирішення підзадач меншої розмірності, і, за рахунок використання швидкої пам'яті комп'ютера, досягти прискорення операції матричного добутку. До недоліків цього підходу слід віднести декілька гіршу чисельну стійкість, хоча й достатню для більшості практичних задач [40-45].

4.4 Завдання до самостійної роботи

1. Виконати модифікацію блокового паралельного алгоритму Кеннона з побудовою відображення на топологію гіперкуб. Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність, загальні накладні витрати. Провести ізоефективний аналіз побудованого алгоритму. Порівняти його з алгоритмом Кеннона для 2D-тору.

2. Виконати модифікацію блокового паралельного алгоритму Фокса з побудовою відображення на топологію гіперкуб. Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність, загальні накладні

витрати. Провести ізоефективний аналіз побудованого алгоритму. Порівняти його з алгоритмом Кеннона для 2D-тору.

3. На основі ізоефективного аналізу провести порівняння ефективності двох паралельних алгоритмів з блоковим розбиттям даних, Кеннона та Фокса. Отримати області пріоритетного застосування цих алгоритмів на основі аналізу накладних витрат на паралелізм.

4. Виконати модифікацію блокового паралельного алгоритму Кеннона/Фокса для множення прямокутних щільнозаповнених матриць. Оцінити якість отриманих паралельних алгоритмів на основі динамічних характеристик таких, як прискорення та ефективність, загальні накладні витрати.

5. Виконайте модифікацію методу матричного множення з використанням раніше описаних паралельних алгоритмів (Кеннона, Фокса, стрічкових) для реалізації добутку матриці на вектор. Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність, загальні накладні витрати.

6. Розробити паралельний алгоритм матричного множення для топології зірка. Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність, накладні витрати на паралелізм.

7. Розробити паралельний алгоритм матричного множення для тривимірної сітки. Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність, накладні витрати на паралелізм.

8. Розробити паралельний алгоритм матричного множення для топології повне бінарне дерево. Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність, накладні витрати на паралелізм.

9. Розробити паралельний поліалгоритм на основі алгоритму Фокса та рекурсивного швидкого множення Штрассена-Винограда (верхній рівень, між процесорами – алгоритм Фокса, нижчий рівень, всередині процесору – Штрассена-Винограда). Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність.

10. Розробити паралельний поліалгоритм на основі алгоритму Фокса та рекурсивного швидкого множення Штрассена-Винограда (верхній рівень, між процесорами – Штрассена-Винограда, нижчий рівень, всередині процесору – алгоритм Кеннона або Фокса). Оцінити якість отриманого паралельного алгоритму на основі його динамічних характеристик таких, як прискорення та ефективність.

4.5 Контрольні питання

1. Дати математичну постановку задачі скалярного матричного множення/добутку.
2. Дати математичну постановку задачі скалярного матричного множення/добутку у блоковому вигляді.
3. Навести приклади класифікації методів матричного множення за типом перемножуваних матриць, за видом послідовного алгоритму, що покладено в основу паралельної схеми, за способом розбиття даних.
4. Навести приклади послідовних алгоритмів виконання матричного множення традиційний та на основі швидкого рекурсивного алгоритму Штрассена. Чи відрізняється їх обчислювальна трудомісткість?
5. Які способи розділення даних використовуються при розробці паралельних алгоритмів матричного множення? Опишіть блокове і стрічкове розбиття та їх модифікації.

6. Наведіть загальну схему блокового паралельного алгоритму матричного добутку Кеннона з відображенням на топологію 2D-тор. Проаналізуйте якість паралельного алгоритму Кеннона на основі динамічних характеристик.

7. Наведіть загальну схему блокового паралельного алгоритму матричного добутку Фокса з відображенням на топологію 2D-тор. Проаналізуйте якість паралельного алгоритму Кеннона на основі динамічних характеристик.

8. Наведіть загальну схему паралельного алгоритму матричного добутку зі стрічковим горизонтальним розбиттям при відображенні на топологію 2D-тор. Проаналізуйте якість паралельного алгоритму Кеннона на основі динамічних характеристик.

9. Наведіть загальну схему паралельного алгоритму матричного добутку зі стрічковим вертикальним розбиттям при відображенні на топологію 2D-тор. Проаналізуйте якість паралельного алгоритму Кеннона на основі динамічних характеристик.

10. Наведіть загальну схему паралельного поліалгоритму матричного добутку Кеннона-Штрассена з відображенням на топологію 2D-тор. Проаналізуйте якість паралельного алгоритму Кеннона на основі динамічних характеристик.

РОЗДІЛ 5
ОСНОВИ ПРОГРАМУВАННЯ В MPI

Одним з відомих способів збільшення продуктивності обчислень є використання паралельних обчислювальних систем. Основна характеристика при класифікації паралельних комп'ютерів – спосіб організації пам'яті. Всю множину архітектур ПОС з точки зору організації пам'яті можна розділити на дві групи: системи із загальною та розподіленою пам'яттю. У обчислювальних системах з розподіленою пам'яттю кожен процесор має власну локальну пам'ять, і прямий доступ до пам'яті інших процесорів неможливий (рис. 5.1).

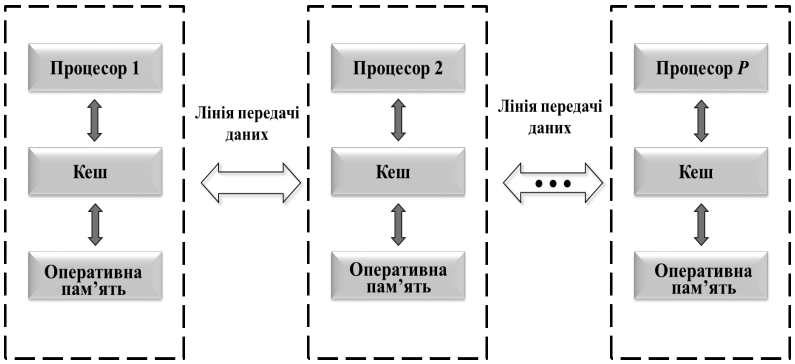


Рисунок 5.1 – Паралельна обчислювальна система
з розподіленою пам'яттю

Така паралельна система або мультикомп'ютер зазвичай складається з декількох самостійних обчислювальних вузлів, для зв'язку яких використовується певне комунікаційне обладнання та технологія програмування, що забезпечує обмін даними, наприклад, MPI (Message Passing Interface), PVM (Parallel Virtual Machine), Cluster OpenMP тощо.

Даний розділ присвячений розгляду методів паралельного програмування для паралельних систем розподіленої пам’яті на базі інтерфейсу передачі повідомлень – MPI.

MPI на сучасному етапі розвитку суперкомп’ютерів є одним з основних підходів для розробки паралельних програм для систем з розподіленою пам'яттю. Використання MPI дозволяє розподілити обчислювальне навантаження і організувати інформаційну взаємодію (передачу даних) між процесорами. Термін MPI означає, з одного боку, стандарт, якому повинні задовольняти засоби організації передачі повідомлень, а, з іншого боку, позначає програмні бібліотеки, які забезпечують можливість передачі повідомлень і при цьому відповідають всім вимогам стандарту MPI [46-80].

5.1 Стандарт MPI та його реалізації

MPI (*Message Passing Interface*) – програмний інтерфейс передачі повідомлень, який дозволяє обмінюватися даними між процесами в паралельних обчисленнях для розподілених комп’ютерних систем. Розробка та супровід стандартів та реалізацій MPI проводиться міжнародним консорціумом MPI Forum [46].

Історично еволюція стандартів ведеться від MPI-1.0, що був затверджений у 1994 році (рис. 5.2) та отримав розвиток у версіях MPI-1.1 (1995 р.), MPI-1.2 (1997 р.) та MPI-1.3 (2008 р.). Паралельно, у 1997 році, з'явився проект стандарту MPI-2. Реалізацією стандарту MPI-2 стала бібліотека MPI-2 з версіями MPI-2.1 (2008 р.) та MPI-2.2 (2009 р.). Далі у 2012 році була опублікована версія 3.0 стандарту MPI. Стандарти MPI 2.0 і 3.0 надають ряд додаткової функціональності, але поки не підтримуються так широко, як MPI 1.0. Версія MPI-3.1 (2015р.) додала незначні розширення до MPI-3.0 і група MPI Forum анонсувала розробку стандарту MPI-4.0.

Спроби зі стандартизації MPI 4.0 спрямовані на додавання нових методів, підходів чи концепцій для використання MPI у архітектурах нового покоління.

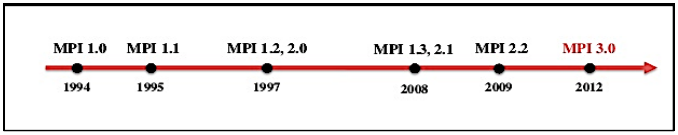


Рисунок 5.2 – Стандарт MPI та його реалізації

На сьогодні MPI є найбільш поширеним та розвиненим засобом організації обміну інформацією в паралельних системах. Існують програмні реалізації MPI (бібліотеки) для більшості комп'ютерних платформ і мов програмування, що дозволяє широко використовувати цей стандарт при розробці паралельних програм. Так, існують стандартні "прив'язки" MPI до мов C / C ++, Fortran 77/90, а також безкоштовні і комерційні реалізації MPI майже для всіх суперкомп'ютерних платформ, а також для UNIX і Windows NT обчислювальних кластерів. Однак, відзначимо, що інтерфейс MPI громіздкий і складний як для прикладного програміста, так і для реалізації, тому варіанти, в яких повною мірою забезпечується поєднання обмінів з обчисленнями, практично відсутні.

MPICH – це одна з перших розроблених реалізацій MPI [47]. Поточна версія бібліотеки – 3.2.1 – відповідає стандарту MPI 3.0, однак починаючи з версії 1.4.1 розробники оголосили про припинення підтримки Windows і рекомендують користувачам Windows замість MPICH використовувати іншу реалізацію – MS MPI від Microsoft (відповідає стандарту MPI 1.0) [48-51].

Основна мета, яку ставили перед собою розробники MPI – забезпечення повної незалежності програм, написаних з використанням бібліотеки MPI, від архітектури багатопроцесорної системи.

Процеси MPI-програми взаємодіють один з одним за допомогою передачі повідомлень шляхом виклику комунікаційних функцій MPI-бібліотеки. Як правило, кожен процес виконується в своєму власному адресному просторі, проте допускається і режим поділу пам'яті.

MPI-бібліотека реалізована у вигляді приблизно 130 функцій, в число яких входять такі основні набори функцій [51-55]:

- функції для роботи з групами процесів і комунікаторами;
- функції, що реалізують комунікаційні операції типу «точка-точка»;
- функції, що реалізують колективні комунікаційні операції.

Набір функцій бібліотеки MPI далеко виходить за рамки набору функцій, мінімально необхідного для підтримки механізму передачі повідомлень. Майже будь-яка MPI-програма може бути написана з використанням всього 6 MPI-функцій, а досить повне і зручне середовище програмування становить набір з 24 функцій.

MPI не забезпечує механізмів завдання початкового розміщення процесів за процесорами. Передбачається, що для цього є інші засоби, які повинні дозволяти задати число процесорів і процесів, розмістити процеси і дані на кожному процесорі.

Стандарт MPI вимагає, щоб MPI-програми могли виконуватися на гетерогенних багатопроцесорних системах, в яких різні процесори мають різні формати даних одного і того ж типу. MPI-програма повинна працювати також на мультипроцесорних обчислювальних системах.

Основними поняттями стандарту MPI є:

- паралельна програма;
- процес;
- група процесів;
- комунікатор.

Паралельна програма в рамках MPI – множина одночасно виконуваних процесів. Процеси можуть виконуватися на різних процесорах, але на одному процесорі можуть розташовуватися і кілька процесів (в цьому випадку їх виконання здійснюється в режимі поділу часу). У граничному випадку для виконання паралельної програми може використовуватися один процесор, як правило, такий спосіб застосовується для початкової перевірки правильності паралельної програми.

Процес – виконання програми на одному процесорі. Процес може містити послідовний код, паралельні гілки, операції введення / виведення та ін.

Група процесів – сукупність процесів, кожний з яких має всередині групи унікальне ім'я. Процеси в групі взаємодіють за допомогою комунікатора групи. Процеси в групі мають номер від 0 до (P-1), де P – кількість процесів у групі.

Комунікатор – реалізує обміни даними між процесами та їх синхронізацію. Для прикладної програми комунікатор виступає в якості комунікаційного середовища. Розрізняють внутрішньо *групові комунікатори* (*intra*) і *міжгрупові комунікатори* (*inter*). Повідомлення, що використовують різні комунікатори, не впливають один на одного і не взаємодіють.

5.2 Базові процедури та функції MPI

Базовий і мінімально повний набір процедур/функцій містить 6 функцій MPI, яких буде достатньо при розробці зовсім простих програм [51-57].

Для опису параметрів процедур/функцій введено наступні позначення:

1) `IN` – вхідні параметри процедури;

2) `OUT` – вихідні параметри процедури;

3) `INOUT` – вхідні параметри, що модифікуються процедурою.

Перша функція, `MPI_Init`, служить для ініціалізації середовища виконання MPI-програми. Параметрами функції є кількість аргументів у командному рядку і текст самого командного рядка:

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

Всі інші MPI-процедури можуть бути викликані тільки після виклику `MPI_Init`. Складний тип аргументів `MPI_Init` передбачений для того, щоб передавати всім процесам аргументи `main`. В результаті виконання цієї функції створюється група процесів, в якій розміщуються всі процеси додатку, і також створюється область зв'язку, що описується визначеним комунікатором `MPI_COMM_WORLD`.

`MPI_Finalize` – завершення паралельної частини програми. Всі подальші звернення до будь-яких MPI-процедур, в тому числі до `MPI_Init`, заборонено. Функція закриває всі MPI-процеси і ліквідує всі області зв'язку:

```
int MPI_Finalize (void).
```

Загальну структуру MPI-програми представлено на рисунку 5.3.

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

Рисунок 5.3 – Загальна структура MPI-програми

Визначення загальної кількості виконуваних процесів паралельної програми здійснюється за допомогою функції `MPI_Comm_size`:

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

1) `IN comm` – ідентифікатор комунікатору;
2) `OUT size` – число процесів у області зв'язку комунікатору.

Визначення номера процесу, що викликав цю функцію, в `comm`:

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

Значення, що повертається за адресою `&rank`, лежить в діапазоні від 0 до `size-1`.

Зазвичай, виклик функцій `MPI_Comm_size` і `MPI_Comm_rank` виконується відразу після `MPI_Init` (рис. 5.4).

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

Рисунок 5.4 – Визначення кількості та рангу процесів

Комунікатор `MPI_COMM_WORLD` створюється за замовчуванням і представляє всі процеси виконуваної паралельної програми. Ранг, одержуваний за допомогою функції `MPI_Comm_rank`, є рангом процесу, який виконав виклик цієї функції, і, тим самим, змінна `Procrank` буде приймати різні значення в різних процесах.

Найпростіші функції передачі/прийому повідомлень [57]. Для передачі повідомлення процес-відправник повинен виконати наступну функцію передачі повідомлення:

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

Функція `MPI_Send` з вказаними параметрами виконує передачу `count` елементів типу `datatype` повідомлення з ідентифікатором `tag` процесу `dest` в області зв'язку комунікатора `comm`.

Параметри функції `MPI_Send`:

1) `IN buf` – адреса буфера пам'яті, початку розташування даних, що пересилаються;

2) `IN count` – кількість елементів даних, що пересилаються;

3) `IN datatype` – тип елементів, що пересилаються;

4) `IN dest` – номер або ранг процесу-одержувача в групі, що пов'язана з комунікатором `comm`;

5) `IN tag` – ідентифікатор повідомлення;

6) `IN comm` – комунікатор області зв'язку.

Повідомлення, що відправляється, визначається через покажчик блоку пам'яті (буфера), в якому це повідомлення розташовується. Використовувана трійка (`buf, count, datatype`) входить до складу параметрів практично всіх функцій передачі даних. Процеси, між якими виконується передача даних, обов'язково повинні належати комунікатору `comm`, який вказаний у цій функції. Параметр `tag` використовується тільки при необхідності розрізнення переданих повідомлень, в іншому випадку в якості параметра може бути використане довільне число.

Для прийому повідомлення процес-отримувач повинен виконати наступну функцію отримання повідомлення:

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

Параметри функції `MPI_Recv`:

1) `OUT buf` – адреса буфера пам'яті, початку розташування даних, що приймаються;

2) `IN count` – максимальна кількість елементів даних, що приймаються;

3) `IN datatype` – тип елементів, що приймаються;

4) `IN tag` – ідентифікатор повідомлення;

5) `IN comm` – комунікатор області зв'язку.

6) `IN source` – номер процесу-відправника;

7) `OUT status` – атрибути прийнятого повідомлення.

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

Буфер пам'яті повинен бути достатнім для прийому повідомлення, а типи елементів відправленого і прийнятого повідомлення повинні збігатися. При нестачі пам'яті частина повідомлення буде втрачена і в коді завершення функції буде зафіксована помилка переповнення. При необхідності прийому повідомлення від будь-якого процесу-відправника для параметра `source` може бути вказано значення `MPI_ANY_SOURCE`. При необхідності прийому повідомлення із будь-яким тегом для параметра `tag` може бути вказано значення `MPI_ANY_TAG`.

Параметр `status` дозволяє визначити ряд характеристик прийнятого повідомлення [58]:

1) `status.MPI_SOURCE` – ранг процесу-відправника прийнятого повідомлення;

2) `status.MPI_TAG` – тег прийнятого повідомлення.

`MPI_Get_count (MPI_Status *status, MPI_Datatype datatype, int *count)` повертає в змінній `count` кількість елементів типу `datatype` в прийнятому повідомленні.

Функція `MPI_Recv` реалізує режим з блокуванням для процесу-отримувача, тобто його виконання припиняється до завершення роботи функції. Таким чином, якщо, з якихось причин, очікуване для прийому повідомлення буде відсутнім, виконання паралельної програми буде блоковано.

Функції визначення часу виконання `MPI`-програми (таймер). При виконанні паралельної програми із `MPI` іноді необхідно визначати час реалізації обчислень або комунікацій, наприклад, для оцінки прискорення, що досягається за рахунок використання паралелізму. Отримання астрономічного часу (у секундах) поточного моменту виконання програми забезпечується за допомогою `MPI`-функції:

```
double MPI_Wtime(void).
```

Функція повертає астрономічний час в секундах, що минув з певного моменту в минулому. Якщо деяку ділянку в програмі оточити викликами цієї функції, то різниця між повернутими значеннями покаже час роботи даного фрагменту програми. Хоча точка відліку функції не контролюється розробником, гарантується, що вона не буде змінена за час існування процесу. Для хронометражу ділянки програми виклик `MPI`- функції проілюстровано в фрагменті програми на мові С (рис. 5.5).

Для визначення поточного значення точності може бути використана функція:

```
double MPI_Wtick(void),
```

(час в секундах між двома послідовними показниками апаратного таймера використаної системи, тобто мінімальне значення кванту часу).

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

Рисунок 5.5 – Хронометраж ділянки MPI-програми

Відповідність між типами даних MPI і типами даних мови С наведена в таблиці 5.1.

Таблиця 5.1 – Основні типи даних MPI

Тип MPI	Тип мови С
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Тип MPI_BYTE використовується для передачі двійкової інформації без будь-якого перетворення. Тип MPI_PACKED використовується для передачі в одному повідомленні різнотипних даних, запакованих спеціальними функціями. Крім того, програмісту надаються засоби створення власних типів даних на базі стандартних. Повний список допустимих типів MS MPI доступний в [56].

У разі успішного виконання всі функції повертають код MPI_SUCCESS, інакше – код помилки. У MPI прийняті стандартні угоди про коди завершення викликів підпрограм:

1) `MPI_SUCCESS` – успішне завершення виклику;
2) `MPI_ERR_OTHER` – помилка, зазвичай повторний виклик процедури `MPI_INIT`;
3) `MPI_ERR_BUFFER` – невірний покажчик на буфер;
4) `MPI_ERR_COMM` – невірний комунікатор;
5) `MPI_ERR_RANK` – невірний ранг;
6) `MPI_ERR_OP` – невірна операція;
7) `MPI_ERR_ARG` – невірний аргумент;
8) `MPI_ERR_UNKNOWN` – необізнана помилка;
9) `MPI_ERR_TRUNCATE` – повідомлення, обрізане при прийомі;
10) `MPI_ERR_INTERN` – внутрішня помилка, зазвичай системі не вистачає пам'яті.

Повний список констант для перевірки коду завершення міститься у файлах `mpi.h`.

5.3 Операції передачі даних між двома процесами

Функції передачі даних між процесами паралельної програми із застосуванням MPI поділяються на дві групи. В одну входять функції, що

призначені для взаємодії двох процесів програми [54-57]. Такі комунікаційні операції називаються парними або операціями типу «точка-точка»/ «point-point».

До іншої групи належать функції, що реалізують взаємодію між усіма процесами в межах деякого комунікатора (колективні операції).

Розглянемо детальніше деякі функції передачі і прийому повідомлень *між двома процесами*. Всі функції даного типу також діляться на дві групи: *з блокуванням* і *без блокування*. Блокування гарантує коректність повторного використання всіх параметрів після повернення з підпрограми (після повернення з відповідної функції можна використовувати будь-які присутні в ній змінні без ризику «зіпсувати» значення переданих повідомлень і додаткових перевірок цих повідомлень).

У комунікаційних операціях типу «точка-точка» завжди беруть участь не більше двох процесів: той, що передає повідомлення, та той, що отримує. Додатково до *стандартного режиму* можливе використання *синхронної*, *буферизованої* або *узгодженої* передачі, як *з блокуванням*, так і *без блокування*.

Синхронний (*Synchronous*) режим полягає в тому, що завершення функції відправки повідомлення відбувається тільки при отриманні від процесу-одержувача підтвердження про початок прийому надісланого повідомлення. Відправлене повідомлення або повністю прийнято процесом-одержувачем або знаходиться в стані прийому.

Режим передачі по готовності, *узгоджений або підготовлений режим* (*Ready*) може бути використаний тільки в тому випадку, якщо операція прийому повідомлення вже ініційована. Буфер повідомлення після завершення функції відправки повідомлення може бути використаний повторно.

Буферизований (*Buffered*) режим передбачає використання додаткових системних буферів для копіювання в них повідомлень, що відправляються. Як результат, функція відправки повідомлення завершується відразу ж після копіювання повідомлень до системного буферу.

Для іменування функцій відправки повідомлення для різних режимів виконання в MPI використовується функція `MPI_Send`, до назви якої як префікс додається початковий символ назви відповідного режиму роботи, тобто:

1) `MPI_Ssend` – функція відправки повідомлення в синхронному режимі;

2) `MPI_Bsend` – функція відправки повідомлення в буферизованому режимі;

3) `MPI_Rsend` – функція відправки повідомлення в режимі по готовності.

Список параметрів всіх перерахованих функцій (таблиця 5.2) збігається зі складом параметрів функції `MPI_Send`.

Таблиця 5.2 – Режими для парних комунікаційних операцій

Режими виконання посилання	З блокуванням	Без блокування
Стандартний	`MPI_Send`	`MPI_Isend`
Синхронний	`MPI_Ssend`	`MPI_Issend`
Буферизований	`MPI_Bsend`	`MPI_Ibsend`
Узгоджений	`MPI_Rsend`	`MPI_Irsend`
Прийом інформації	`MPI_Recv`	`MPI_Irecv`

Для використання буферизованого режиму передачі повинен бути створений і переданий в MPI буфер пам'яті для буферизації повідомлень. Використана для цього функція має вигляд:

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

де `buf` – буфер пам'яті для буферизації повідомлень,

`size` – розмір буферу.

Після завершення роботи з буфером він повинен бути відключений від MPI за допомогою функції:

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

Щодо практичного використання режимів можна навести такі рекомендації. Режим передачі по готовності формально є найбільш швидким, але використовується дуже рідко, тому що зазвичай складно гарантувати готовність операції прийому. Стандартний і буферизований режими також виконуються досить швидко, але можуть призводити до великих витрат ресурсів (пам'яті). В цілому вони можуть бути рекомендовані для передачі коротких повідомлень. Синхронний режим є найбільш повільним, тому що вимагає підтвердження прийому. У той же час, цей режим найбільш надійний – можна рекомендувати його для великих повідомлень. На закінчення відзначимо, що для функції прийому `MPI_Recv` не існує різних режимів роботи.

5.3.1 Функції передачі повідомлень з блокуванням

Розглянута раніше функція передачі повідомлень з блокуванням `MPI_Send` забезпечує так званий стандартний (*Standard*) режим відправки, при якому:

- на час виконання функції процес-відправник повідомлення блокується;

- після завершення функції буфер може бути використаний повторно;

- стан надісланого повідомлення може бути різним – повідомлення може розташовуватися в процесі-відправника, може перебувати в процесі передачі, може зберігатися в процесі-одержувачі або ж може бути прийнято процесом-одержувачем за допомогою функції MPI_Recv.

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

Параметри функції `MPI_Send`:

1) `IN buf` – адреса початку розташування даних, що пересилаються;

2) `IN count` – число елементів, що пересилаються;

3) `IN datatype` – тип MPI елементів, що пересилаються;

4) `IN dest` – номер процесу-одержувача в групі, пов'язаної з комунікатором comm;

5) `IN tag` – ідентифікатор повідомлення (часто є порядковим номером повідомлення в послідовності);

6) `IN comm` – комунікатор області зв'язку.

Функція `MPI_Send` виконує відправку `count` елементів типу `datatype` повідомлення з ідентифікатором `tag` процесу з номером `dest` в області зв'язку комунікатора `comm`. Змінна `buf` – це, як правило, масив або скалярна змінна. В останньому випадку значення `count` = 1.

Повернення з функції не означає ані того, що повідомлення отримано процесом `dest`, ні того, що повідомлення покинуло процесорний елемент, на якому виконується процес, який викликав функцію. Оскільки `MPI_Send` є функцією з блокуванням, гарантується безпека подальшої зміни змінних, використаних в якості аргументів.

Функція прийому повідомлень з блокуванням `MPI_Recv`:

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

Параметри функції `MPI_Recv`:

1) `OUT buf` – адреса початку розташування прийнятого повідомлення;
2) `IN count` – максимальне число прийнятих елементів;
3) `IN datatype` – тип елементів прийнятого повідомлення;
4) `IN source` – номер процесу-відправника в групі, пов'язаної з комунікатором comm;
5) `IN tag` – ідентифікатор повідомлення;
6) `IN comm` – комунікатор області зв'язку;
7) `OUT status` – атрибути прийнятого повідомлення.

Функція виконує прийом `count` елементів типу `datatype` повідомлення з ідентифікатором `tag` від процесу `source` в області зв'язку комунікатора `comm`. Число елементів в прийнятому повідомленні не повинно перевищувати значення `count`. Якщо число прийнятих елементів менше, гарантується, що в буфері `buf` зміняться тільки відповідні їм елементи. Блокування гарантує, що після повернення з функції всі елементи повідомлення будуть прийняті і розташовані в `buf`. Точне число елементів в прийнятому повідомленні (для виділення необхідного обсягу пам'яті при розташуванні повідомлення) можна дізнатися за допомогою функції `MPI_Get_count`:

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

Параметри функції `MPI_Get_count`:

1) `IN status` – параметри прийнятого повідомлення;
2) `IN datatype` – тип елементів повідомлення;
3) `OUT count` – число елементів повідомлення.

Структура `status` дозволяє визначити параметри прийнятого повідомлення, такі як:

- `status.MPI_SOURCE` – номер процесу відправника;
- `status.MPI_TAG` – ідентифікатор повідомлення;
- `status.MPI_ERROR` – код помилки [58].

У функції `MPI_Recv` при визначенні номера процесу-відправника можливе використання спеціального параметра `MPI_ANY_SOURCE` («приймай від кого завгодно»), а в якості ідентифікатора одержуваного повідомлення – `MPI_ANY_TAG` («приймай, що завгодно»). Це так звані параметри-джокери, MPI резервує для них негативні цілі числа (в той час як реальні ідентифікатори процесів і повідомлень завжди лежать в діапазоні від 0 до 32767).

Користуватися джокерами слід з обережністю, тому що таким викликом `MPI_Recv` може бути помилково захоплено повідомлення, яке повинно прийматися в іншій частині процесу-одержувача. Якщо логіка програми досить складна, використовувати джокери можна тільки в функціях, що перевіряють наявність повідомлення для процесу (`MPI_Probe`), щоб перед фактичним прийомом дізнатися тип і кількість даних в повідомленні. Незважаючи на те, що ми хочемо отримати «що завгодно», тип прийнятих даних в функції `MPI_Recv` повинен бути вказаний явно, а він може бути різним в повідомленнях з різними ідентифікаторами.

Функція `MPI_Probe` дозволяє отримати інформацію про структуру повідомлення, що очікується (в структурі `OUT status`), з блокуванням:

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

Повернення з функції не відбудеться, поки повідомлення з відповідним ідентифікатором (`tag`) і номером процесу-відправника (`source`) не буде доступним для отримання. Функція визначає факт приходу повідомлення, не отримуючи його самостійно.

Описані вище функції `MPI_Send/MPI_Recv` реалізують стандартний (*Standard*) режим з блокуванням. Повернення з такої функції передачі даних ще не означає, що передача завершена, гарантується тільки можливість повторного використання буфера даних для внесення в нього змін, які вже не вплинуть на дані, що відправляються.

5.3.2 Додаткові функції передачі повідомлень

У комунікаційних операціях типу «точка-точка» завжди беруть участь не більше двох процесів: той, що передає, та той, отримує. Додатково до стандартного режиму можливе використання синхронної, буферизованої або узгодженої передачі, як з блокуванням, так і без блокування [54-55,57].

Невизначеність функції `MPI_Send` не завжди допустима, тому для розширення можливостей обміну повідомленнями можна використовувати такі функції (параметри функцій еквівалентні `MPI_Send`):

`MPI_Bsend` – передача повідомлення з буферизацією. Якщо прийом повідомлення, що посилається, не був ініціалізований процесом-одержувачем, то повідомлення буде записано в буфер, і станеться негайне повернення з функції. Виконання даної функції ніяк не залежить від виклику функції прийому повідомлення. Функція поверне код помилки, якщо місця під буфер недостатньо.

`MPI_Ssend` – передача повідомлення з синхронізацією. Повернення з даної функції станеться тільки тоді, коли виконається прийом повідомлення, що посилається, процесом-одержувачем. Таким чином, завершення передачі з синхронізацією говорить не тільки про можливість повторного використання буфера, але і про гарантоване досягнення процесом-одержувачем точки прийому повідомлення в програмі.

`MPI_Rsend` – передача повідомлення за готовністю. Цією функцією можна користуватися тільки в тому випадку, якщо процес-одержувач вже ініціював прийом повідомлення. В іншому випадку виклик функції є помилковим і результат її виконання не визначений. Функція скорочує протокол взаємодії між відправником і отримувачем, зменшуючи накладні витрати на організацію передачі.

5.3.3 Прийом/передача повідомлень без блокування

MPI забезпечує можливість виконання операцій передачі даних між двома процесами без блокування [57]. Найменування неблокувальних аналогів утворюється з назв відповідних функцій шляхом додавання префікса I (*Immediate*).

На відміну від функцій з блокуванням, повернення з функцій даної групи відбувається відразу без будь-якого блокування процесів. Паралельно з подальшим виконанням програми відбувається і обробка асинхронно запущеної операції. Передача повідомлення аналогічна `MPI_Send`, але повернення з неї відбувається відразу після ініціалізації процесу передачі без очікування обробки всього повідомлення, що знаходиться в буфері `buf`. Це означає, що не можна використовувати цей буфер повторно для інших цілей без отримання додаткової інформації, яка б підтверджувала завершення операції передачі (інакше можна зіпсувати передане повідомлення).

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

Параметри функції `MPI_Isend`:

1) `IN buf` – адреса початку буфера з посланим повідомленням;
2) `IN count` – число переданих елементів;
3) `IN datatype` – тип елементів;
4) `IN dest` – номер процесу-одержувача;
5) `IN tag` – ідентифікатор повідомлення;
6) `IN comm` – ідентифікатор комунікатора;
7) `OUT request` – ідентифікатор асинхронної операції.

Щоб визначити момент часу, коли можна повторно використовувати буфер, використовується параметр `request` або функції `MPI_Wait` і `MPI_Test`.

Аналогічно модифікаціям функції `MPI_Send` (`MP_Bsend`, `MPI_Ssend`, `MPI_Rsend`) передбачені модифікації функції без блокування: `MPI_Ibsend`, `MPI_Issend`, `MPI_Irsend`.

Функція прийому повідомлення без блокування `MPI_Irecv`:

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

Прийом повідомлення аналогічний `MPI_Recv`, але повернення з функції відбувається відразу після ініціалізації процесу прийому без завершення отримання всього повідомлення і запису в буфері `buf`. Закінчення процесу можна також визначити за допомогою параметра request, процедур `MPI_Wait` та `MPI_Test`.

Функція очікування завершення неблокувальної операції `MPI_Wait`:

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

Функція очікує завершення асинхронної операції, асоційованої з ідентифікатором `request` і запущеної функцією `MPI_Isend` або `MPI_Irecv`, і блокує подальше виконання процесу, що її викликав.

Параметри функції `MPI_Wait`:

1) `INOUT request` – ідентифікатор операції асинхронного прийому або передачі;

2) `OUT status` – параметри повідомлення про закінчену операцію.

Функція перевірки завершення неблокувальної операції `MPI_Test`:

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

Параметри функції `MPI_Test`:

1) `INOUT request` – ідентифікатор операції асинхронного прийому або передачі;

2) `OUT flag` – ознака завершеності операції, що перевіряється;

3) `OUT status` – ідентифікатор операції асинхронного прийому або передачі.

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

Функція `MPI_Waitall` блокує виконання процесу, поки всі асинхронні операції обміну, асоційовані з ідентифікаторами `requests`, не будуть завершені:

```
int MPI_Waitall (int count, MPI_Request *requests, MPI_Status *statuses).
```

Функція `MPI_Waitany` блокує виконання процесу, поки якась асинхронна операція обміну, асоційована з ідентифікаторами `requests`, не буде завершена:

```
int MPI_Waitany (int count, MPI_Request *requests, int *index, MPI_Status *status).
```

Параметр `index` вказує номер елемента в масиві `requests`, що відповідає операції, яка повинна завершитись.

Крім розглянутих, MPI містить ряд додаткових функцій перевірки та очікування неблокуючих операцій обміну:

1) `MPI_Testall` – перевірка завершення всіх перерахованих операцій обміну;
2) `MPI_Waitall` – очікування завершення всіх операцій обміну;
3) `MPI_Testany` – перевірка завершення хоча б однієї з перерахованих операцій обміну;
4) `MPI_Waitany` – очікування завершення будь-якої з перерахованих операцій обміну;
5) `MPI_Testsome` – перевірка завершення кожної з перерахованих операцій обміну;
6) `MPI_Waitsome` – очікування завершення хоча б однієї з перерахованих операцій обміну і оцінка стану по всіх операціях.

5.3.4 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 процесів. Найпростіший варіант цієї ситуації полягає, наприклад, в обміні даними між двома процесами [54-55, 57] .

Реалізація таких обмінів за допомогою звичайних парних операцій передачі даних неефективна і досить трудомістка. 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 забезпечено за допомогою функції `MPI_Sendrecv`:

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

де

`IN sendbuf, IN sendcount, IN sendtype, IN dest, IN sendtag` – параметри переданого повідомлення,

`OUT recvbuf, IN recvcount, IN recvtype, IN source, IN recvtag` – параметри прийнятого повідомлення,

`IN comm` – комунікатор, в рамках якого відбуватиметься передача даних,

`OUT status` – структура даних з інформацією про результат виконання операції.

Як випливає з опису, функція `MPI_Sendrecv` передає повідомлення, що описується параметрами (`sendbuf, sendcount, sendtype, dest, sendtag`), процесу з рангом `dest` і приймає повідомлення в буфер, який визначається параметрами (`recvbuf, recvcount, recvtype, source, recvtag`), від процесу з рангом `source`.

Функція `MPI_Sendrecv` для передачі та прийому повідомлень використовує різні буфери. У разі ж, коли повідомлення мають однаковий тип, в MPI є можливість використання єдиного буфера:

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

Повний список функцій MS MPI можна знайти в [51, 59].

5.4 Колективні операції передачі даних

Парні операції або операції типу «point- point» («точка-точка»), що наведені у попередньому підрозділі, забезпечують розробку паралельних алгоритмів довільної складності, але можливості MPI цим не обмежені. Якщо виникає необхідність розіслати деяку інформацію від одного процесу всім іншим, набагато зручніше використовувати колективні операції (наприклад, `MPI_Bcast`) замість багаторазового використання функцій `Send/Recv` між парами процесів [59-60].

Головна відмінність колективних операцій від операцій типу «точка-точка» полягає в тому, що в них завжди беруть участь всі процеси в межах певного комунікатора. Недотримання цього правила призводить або до аварійного завершення завдання, або до його зависання.

Колективні операції повинні бути виконані кожним процесом, можливо, зі своїм набором параметрів. Повернення з процедури колективної взаємодії може відбутися в той момент, коли участь процесу в даній операції вже закінчено. Як і у випадку з блокуючими процедурами операцій «точка-точка», повернення означає, що доступ до буферу прийому/передачі дозволений і не загрожує цілісності інформації, що передається. Асинхронних колективних операцій в MPI немає. У колективних операціях, на відміну від операцій «точка-точка», не використовуються ідентифікатори повідомлень.

5.4.1 Типи колективних операцій

Набір колективних операцій включає:

а) синхронізацію всіх процесів за допомогою бар'єрів (`MPI_Barrier`);

б) колективні комунікаційні операції, в число яких входять:

- розсилка інформації від одного процесу всім іншим членам деякої області зв'язку (`MPI_Bcast`);

- збір (`gather`) розподіленого по процесам масиву в один масив зі збереженням його в адресному просторі виділеного (`root`) процесу (`MPI_Gather, MPI_Gatherv`);

- збір (`gather`) розподіленого масиву в один масив з розсилкою його всім процесам певної області зв'язку (`MPI_Allgather`, `MPI_Allgatherv`);

- розбиття масиву і розсилка його фрагментів (`scatter`) всім процесам області зв'язку (`MPI_Scatter, MPI_Scatterv`);

- поєднана операція `Scatter/Gather` (`All-to-All`), кожен процес ділить дані зі свого буфера передачі і надсилає фрагменти всім іншим процесам, одночасно збираючи фрагменти, послані іншими

процесами, в свій буфер прийому (`MPI_Alltoall`, `MPI_Alltoallv`);

в) глобальні обчислювальні операції (`sum`, `min`, `max` та ін.) над даними, розташованими в адресних просторах різних процесів:

- зі збереженням результату в адресному просторі одного процесу (`MPI_Reduce`);

- з розсилкою результату всім процесам (`MPI_Allreduce`);

- поєднана операція `Reduce` та `Scatter` (`MPI_Reduce_scatter`);

- префіксна редукція (`MPI_Scan`).

Всі комунікаційні підпрограми, за винятком `MPI_Bcast`, представлені в двох варіантах:

- простий варіант, коли всі частини переданого повідомлення мають однакову довжину і займають суміжні області в адресному просторі процесів;

- «векторний» варіант, який надає більш широкі можливості по організації колективних комунікацій, знімаючи обмеження, властиві простому варіанту, як стосовно довжин блоків, так і пов'язані з розміщенням даних в адресному просторі процесів.

Векторні варіанти відрізняються додатковим символом «v» наприкінці імені функції.

Відмінні риси або особливості колективних операцій:

- колективні комунікації не взаємодіють з комунікаціями типу «точка-точка»;

- колективні комунікації виконуються в режимі з блокуванням;

- повернення з підпрограми в кожному процесі відбувається тоді, коли його участь у колективній операції завершилась, однак це не означає, що інші процеси завершили операцію;

- кількість одержаних даних має дорівнювати кількості посланих даних;

- типи елементів повідомлень, що посилаються та одержуються, повинні збігатися;

- повідомлення не мають ідентифікаторів.

Всі колективні операції, доступні в реалізації MS MPI, можна знайти в [59-60]. Нижче наведені основні з них.

Функція `MPI_Bcast` виконує розсилку повідомлення `buffer` довжиною `count` від процесу `source` всім процесам комунікатора `comm`, включаючи процес, що розсилає:

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

Параметри функції `MPI_Bcast`:

- `OUT buffer` – адреса початку буфера посилки повідомлення;

- `IN count` – число переданих елементів в повідомленні;

- `IN datatype` – тип переданих елементів;

- `IN source` – номер процесу, що розсилає повідомлення;

- `IN comm` – ідентифікатор групи [61].

При поверненні з функції вміст буфера `buffer` буде скопійовано в локальний буфер кожного процесу. Значення `count`, `datatype`, `source` і `comm` повинні бути однакові у всіх процесів (рис. 5.6).

Наведемо приклад:

```
    MPI_Bcast(my_array, 100, MPI_INT, 0, comm).
```

В результаті такого виклику функції усіма процесами комунікатора `comm` будуть отримані перші 100 цілих чисел з масиву `my_array` нульового процесу в локальні буфери `my_array`.

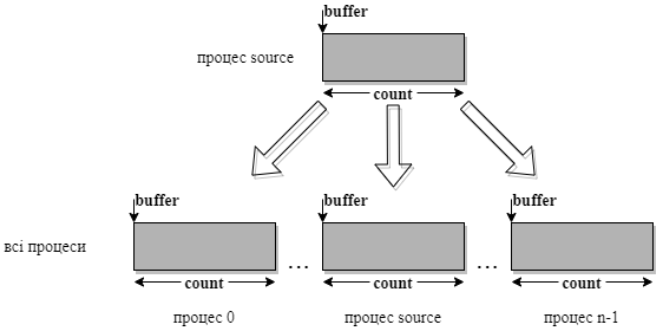


Рисунок 5.6 – Ілюстрація виконання операції `MPI_Bcast`

Функція `MPI_Barrier` блокує роботу процесів, що її викликали, до того часу, поки вся решта процесів групи `comm` також не виконають її. Всі процеси повинні викликати цю функцію, хоча її виклик для різних процесів може перебувати в різних частинах програми:

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

де `IN comm` – ідентифікатор групи.

Функція `MPI_Gather`:

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

Параметри функції `MPI_Gather`:

1) `IN sendbuf` – адреса початку розміщення даних, що посилаються;

2) `IN sendcount` – число елементів, що посилаються;

3) `IN sendtype` – тип елементів;

4) `OUT recvbuf` – адреса початку буфера прийому (використовується тільки в процесі-одержувачі `root`);

5) `IN recvcount` – число елементів, отриманих від кожного процесу (використовується тільки в процесі-одержувачі `root`);

6) `IN recvtype` – тип одержуваних елементів;

7) `IN root` – номер процесу-одержувача;

8) `IN comm` – ідентифікатор групи.

Функція `MPI_Gather` виконує збірку блоків даних `sendbuf`, що посилаються усіма процесами групи, в один масив процесу з номером `root`. Довжина блоків вважається однаковою. Об'єднання відбувається в порядку збільшення номерів процесів-відправників. Тобто дані, послані процесом і зі свого буфера `sendbuf`, поміщаються в і-у порцію буфера `recvbuf` процесу `root`. Довжина масиву `sendcount`, в якому збираються дані, повинна бути достатньою для їх розміщення. Значення `root` і `comm` повинні бути однакові у всіх процесів (рис. 5.7).

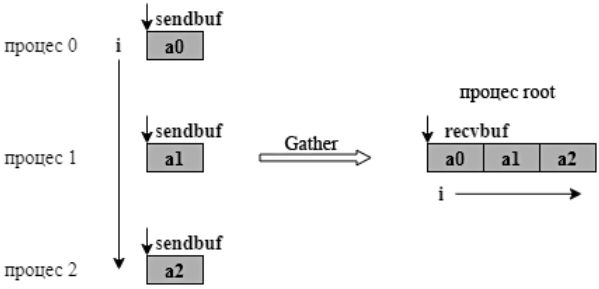


Рисунок 5.7 – Ілюстрація виконання операції `MPI_Gather`

За допомогою аналогічної функції `MPI_Gatherv` можна приймати від процесів масиви даних різної довжини.

Функція `MPI_Allgather` виконується так само, як `MPI_Gather`, але одержувачами є всі процеси групи. Дані, надіслані процесом і зі свого буфера `sendbuf`, поміщаються в і-у порцію буфера `recvbuf` кожного

процесу (рис. 5.8). Після завершення операції зміст буферів прийому `recvbuf` у всіх процесів однаковий:

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

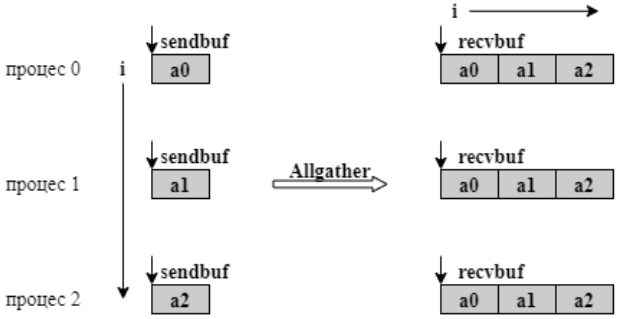


Рисунок 5.8 – Ілюстрація виконання операції `MPI_Allgather`

Параметри функції `MPI_Allgather`:

1) `IN sendbuf` – адреса початку буфера посилки;
2) `IN sendcount` – число елементів, що посилаються;
3) `IN sendtype` – тип елементів, що посилаються;
4) `OUT recvbuf` – адреса початку буфера прийому;
5) `IN recvcount` – число елементів, що одержуються від кожного процесу;
6) `IN recvtype` – тип одержуваних елементів;
7) `IN comm` – ідентифікатор групи.

Функція `MPI_Scatter` є зворотною до функції `MPI_Gather`. Вона розбиває повідомлення з буфера процесу `root` на рівні частини

розміром `sendcount` та посилає і-ту частину в буфер прийому процесу з номером і (в тому числі і собі):

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

Параметри функції `MPI_Scatter`:

1) `IN sendbuf` – адреса початку розміщення блоків даних, що розподіляються (використовується тільки в процесі-відправнику `root`);

2) `IN sendcount` – число елементів, що посилаються кожному процесу (використовується тільки в процесі-відправнику `root`);

3) `IN sendtype` – тип елементів, що посилаються (використовується тільки в процесі-відправнику `root`);

4) `OUT recvbuf` – адреса початку буфера прийому;

5) `IN recvcount` – число одержуваних елементів;

6) `IN recvtype` – тип одержуваних елементів;

7) `IN root` – номер процесу-відправника;

8) `IN comm` – ідентифікатор комунікатора.

Процес `root` використовує обидва буфери (посилки і прийому), тому у підпрограмі, що викликається ним, всі параметри є суттєвими. Решта процесів групи є лише одержувачами, тому для них параметри, які визначають буфер посилки, не істотні. Тип елементів `sendtype`, що посилаються, повинен збігатися з типом `recvtype` одержуваних елементів, також число елементів `sendcount`, що посилаються, повинно дорівнювати числу прийнятих `recvcount`. Параметри `root` і `comm` повинні бути однаковими на всіх процесах (рис. 5.9).

За допомогою аналогічної функції `MPI_Scatterv` можна відсилати процесам порції даних різної довжини.

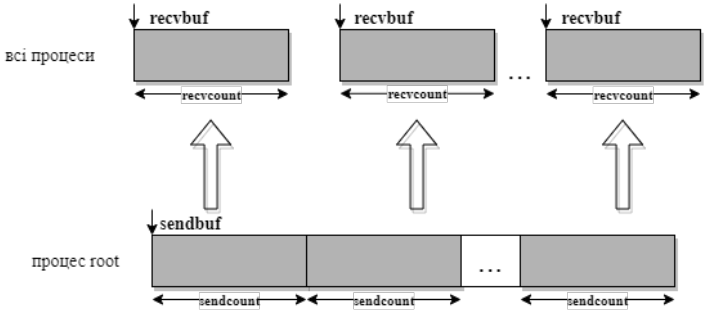


Рисунок 5.9 – Ілюстрація виконання операції `MPI_Scatter`

На рисунку 5.10 показано використання функції для розсилки рядків масиву.

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4
int main(int argc, char **argv) {
    int numtasks, rank, sendcount, recvcount, source;
    float sendbuf[SIZE][SIZE] = { {1.0, 2.0, 3.0, 4.0},
                                  {5.0, 6.0, 7.0, 8.0},
                                  {9.0, 10.0, 11.0, 12.0},
                                  {13.0, 14.0, 15.0, 16.0} };
    float recvbuf[SIZE];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    if (numtasks == SIZE) {
        source = 1;
        sendcount = SIZE;
        recvcount = SIZE;
        MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,
                    recvcount,MPI_FLOAT,source,MPI_COMM_WORLD);
        printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
                recvbuf[1],recvbuf[2],recvbuf[3]);
    }
    else
        printf("Число процессоров должно быть равно %d. \n",
        SIZE);

    MPI_Finalize();
}
```

Рисунок 5.10 – Розсилка рядків масиву

До колективних операцій належать і редукційні операції. Вони припускають, що на кожному процесі зберігаються деякі дані, над якими слід виконати певну операцію, наприклад, складання чисел або пошук максимального значення. Перелік визначених операцій наведено нижче, крім того, можуть виконуватися і операції, задані користувачем [62].

Функція `MPI_Reduce` виконує `count` незалежних глобальних операцій `op`. Передбачається, що в буфері `sendbuf` кожного процесу знаходяться `count` аргументів типу `datatype`. Перші елементи масивів `sendbuf` беруть участь в першій операції `op`, другі – в другій тощо (рис. 5.11). Результати виконання всіх операцій записуються в буфер `recvbuf` кореневого процесу `root`:

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

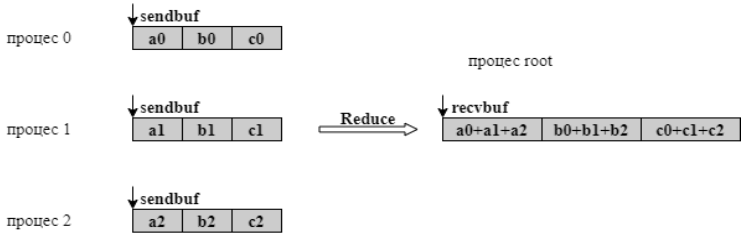


Рисунок 5.11 – Ілюстрація виконання операції `MPI_Reduce`

Параметри функції `MPI_Reduce`:

1) `IN sendbuf` – адреса буфера, що передається;
2) `OUT recvbuf` – адреса буфера прийому;
3) `IN count` – кількість переданих елементів;
4) `IN datatype` – тип елементів, що посилаються;

5) `IN op` – операція редукції;

6) `IN root` – ранг кореневого процесу;

7) `IN comm` – комунікатор.

Передбачається, що операція має властивості асоціативності і комутативності, щоб задовольнити вимогам ефективності.

Функція `MPI_Allreduce` аналогічна функції `MPI_Reduce`, але результати виконання операцій записуються в буфер на кожному процесі (5.12):

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

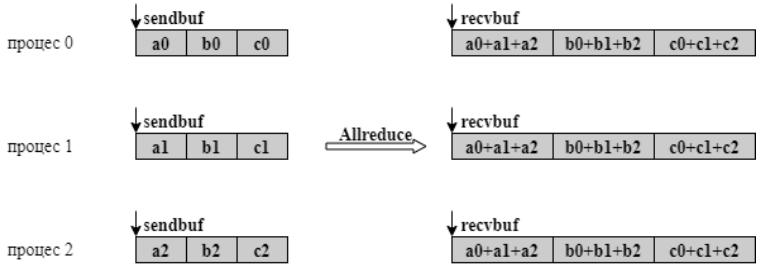


Рисунок 5.12 – Ілюстрація виконання операції `MPI_Allreduce`

Функція `MPI_Alltoall` збирає дані `sendbuf` в кількості `sendcount` елементів типу `datatype` з усіх процесів групи `comm` і передає `recvcount` з них в буфер `recvbuf`:

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

5.4.2 Колективні операції у функціях редукції

Існуючі операції `MPI_Op` [63], що зустрічаються у функціях редукції, представлені в таблиці 5.3.

Таблиця 5.3 – Можливі операції в функціях редукції

Назва	Функція	Дозволені типи
MPI_MAX MPI_MIN	Максимум Мінімум	C integer, FORTRAN integer, Floating point
MPI_SUM MPI_PROD	Сума Добуток	C integer, FORTRAN integer, Floating point, Complex
MPI_LAND MPI_LOR MPI_LXOR	Логічне І Логічне Або Логічне виключаюче Або	C integer, FORTRAN Logical
MPI_BAND MPI_BOR MPI_BXOR	Порозрядне І Порозрядне Або Порозрядне виключаюче Або	C integer, FORTRAN integer, Byte
MPI_MAXLOC MPI_MINLOC	Максимальне (мінімальне) значення та його індекс	Спеціальні функції для цих функцій

5.5 Похідні типи та пакування даних в MPI

5.5.1 Похідні типи даних

При розробці паралельних програм іноді виникає потреба передавати дані різних типів (наприклад, структури) або дані, розташовані в несуміжних областях пам’яті (частини масивів, які не утворюють безперервну послідовність елементів).

MPI надає два механізми ефективного пересилання даних в цих випадках:

- створення похідних типів для використання в комунікаційних операціях замість визначених типів MPI;

- пересилання пакованих даних (процес-відправник пакує дані перед їх відправкою, а процес-одержувач розпаковує їх після отримання).

Похідні типи MPI не є в повному розумінні типами даних, як це розуміється в мовах програмування. Вони не можуть використовуватися ні в яких інших операціях, крім комунікаційних. Похідні типи MPI слід розуміти як дескриптори розташування в пам'яті елементів базових типів. Похідний тип MPI являє собою прихований (`opaque`) об'єкт [64], який специфікує дві речі: послідовність базових типів і послідовність зсувів. Послідовність таких пар визначається як відображення (карта) типу:

```
Typemap ={(type0, disp0),...,(typen-1, dispn-1)}.
```

Значення зсувів не обов'язково повинні бути невід'ємними, різними і впорядкованими за зростанням. Відображення типу разом з базовою адресою початку розташування даних `buf` визначає комунікаційний буфер обміну. Цей буфер буде містити n елементів, а i-й елемент матиме адресу `buf + desp` і буде мати базовий тип `type`. Стандартні типи MPI мають зумовлені відображення типів. Наприклад, `MPI_INT` має відображення: `{(int, 0)}`.

Використання похідного типу в функціях обміну повідомленнями можна розглядати як трафарет, накладений на область пам'яті, яка містить передане або прийняте повідомлення. Стандартний сценарій визначення і використання похідних типів включає наступні кроки [65]:

а) похідний тип будується з визначених типів MPI і раніше визначених похідних типів за допомогою спеціальних функцій-конструкторів [66]:

- `MPI_Type_contiguous`;
- `MPI_Type_vector`;
- `MPI_Type_hvector`;
- `MPI_Type_indexed`;
- `MPI_Type_hindexed`;
- `MPI_Type_struct`;

б) новий похідний тип реєструється викликом функції `MPI_Type_commit`, тільки після реєстрації новий похідний тип можна використовувати в комунікаційних підпрограмах і при конструюванні інших типів (визначені типи MPI вважаються зареєстрованими);

в) коли похідний тип стає непотрібним, він знищується функцією `MPI_Type_free`.

Будь-який тип даних в MPI має дві характеристики: протяжність і розмір, виражені в байтах:

- протяжність типу визначає, скільки байт змінна даного типу займає в пам'яті; ця величина може бути обчислена як: «адреса останньої клітинки даних - адреса першої клітинки даних + довжина останньої клітинки даних» (опитується підпрограмою `MPI_Type_extent`);

- розмір типу визначає кількість реально переданих байт в комунікаційних операціях; ця величина дорівнює сумі довжин всіх базових елементів типу, що визначається (опитується підпрограмою `MPI_Type_size`).

Для простих типів протяжність і розмір збігаються.

5.5.2 Функції для роботи з похідними типами даних

Функція `MPI_Type_extent` визначає протяжність (`OUT extent`) елемента деякого типу `datatype`:

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent).
```

Функція `MPI_Type_size` визначає «чистий» розмір (`OUT size`) елемента деякого типу `datatype` (за вирахуванням порожніх проміжків):

```
int MPI_Type_size (MPI_Datatype datatype, int *size).
```

Функція `MPI_Type_contiguous` є найпростішим конструктором нового типу даних, елементи якого складаються із зазначеного числа елементів базового типу, що займають суміжні області пам'яті:

```
int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype).
```

Параметри функції `MPI_Type_contiguous`:

- `IN count` – число повторень;
- `IN oldtype` – старий тип даних;
- `OUT newtype` – новий тип даних.

Новий тип `newtype` отримується конкатенацією (зчепленням) `count` копій старого типу `oldtype`. Наприклад, нехай `oldtype` має карту `type map {(double, 0), (char, 8)}` з довжиною 16, і нехай `count` = 3.

Карта нового типу буде мати вигляд:
`{ (double, 0), (char, 8), (double, 16), (char, 24), (double, 32), (char, 40) }`,тобто містити 3 копії вихідного типу, що йдуть поспіль, зі зсувами 0, 8, 16, 24, 32, 40.

Функція `MPI_Type_vector` є більш універсальним конструктором:

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

Параметри функції `MPI_Type_vector`:

- `IN count` – число блоків;
- `IN blocklength` – число елементів в кожному блоці;
- `IN stride` – крок між початками сусідніх блоків, виміряний числом елементів базового типу;
- `IN oldtype` – старий тип даних;
- `OUT newtype` – новий тип даних.

Вона створює новий тип `newtype`, який являє собою кілька (`count`) рівновіддалених один від одного блоків з однаковим числом `blocklength` суміжних елементів базового типу `oldtype`. Крок `stride` між початком блоку і початком наступного блоку усюди однаковий і кратний протяжності базового типу. Кожен блок виходить конкатенацією деякої кількості копій старого типу. Простір між блоками кратний розміру `oldtype`.

Графічна інтерпретація прикладу роботи конструктора MPI_Type_vector наведена на рисунку 5.13.

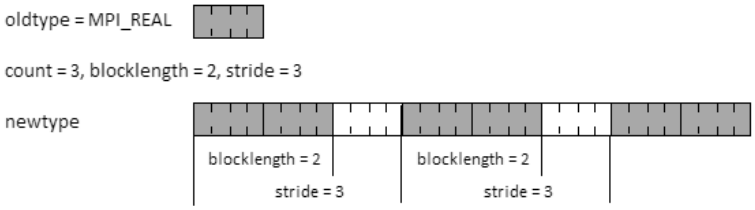


Рисунок 5.13 – Ілюстрація роботи конструктора `MPI_Type_vector`

Конструктор типу `MPI_Type_hvector` розширює можливості конструктора `MPI_Type_vector`, дозволяючи задавати довільний крок `stride` між початками блоків в байтах:

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

Конструктор типу `MPI_Type_indexed` є більш універсальним конструктором в порівнянні з `MPI_Type_vector`, оскільки елементи створюваного типу складаються з довільних по довжині блоків `array_of_blocklengths` з довільним зсувом блоків `array_of_displacements` від початку розміщення елементу. Зсув вимірюються в елементах старого типу.

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

Конструктор типу `MPI_Type_hindexed` ідентичний конструктору `MPI_Type_indexed` за винятком того, що зміщення вимірюються в байтах:

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

Конструктор типу `MPI_Type_struct` – найбільш універсальний з усіх конструкторів типів. Створюваний ним тип є структурою, що складається з довільного числа блоків, кожен з яких може містити

довільне число елементів одного з базових типів і може бути зміщений на довільне число байтів від початку розміщення структури:

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

Функція `MPI_Type_commit` реєструє створений похідний тип `datatype`. Тільки після реєстрації новий тип може використовуватися в комунікаційних операціях:

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

Функція `MPI_Type_free` знищує дескриптор похідного типу:

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

5.5.3 Пакування даних

Для здійснення пересилання елементів різного типу з кількох областей пам'яті, їх слід попередньо запакувати в один масив, послідовно звертаючись до функції пакування `MPI_Pack` [67-69]. При першому виклику функції пакування параметр `position`, як правило, встановлюється в 0, щоб паковані дані розміщувалися з початку буфера.

Для безперервного заповнення буфера необхідно в кожному наступному виклику використовувати значення параметра `position`, отримане з попереднього виклику. Пакований буфер пересилається будь-якими комунікаційними операціями із зазначенням типу `MPI_PACKED` і комунікатора `comm`, який використовувався при зверненнях до функції `MPI_Pack`.

Отримане паковане повідомлення розпаковується в різні масиви або змінні. Це реалізується послідовними викликами функції

розпакування MPI_Unpack із зазначенням числа елементів, яке слід витягти при кожному виклику, і з передачею значення position, повернутого попереднім викликом. При першому виклику функції параметр position слід встановити в 0.

У загальному випадку, при першому зверненні має бути встановлено то значення параметра position, яке було використано при першому зверненні до функції пакування даних. Очевидно, що для правильного розпакування даних, черговість вилучення даних повинна бути тією ж самою, як і пакування.

Функції для передачі пакованих даних.

Функція MPI_Pack_size допомагає визначити розмір буфера, необхідний для пакування деякої кількості даних типу datatype:

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

Короткі відомості про функції передачі пакованих даних.

Функція MPI_Pack пакує incount елементів визначеного або похідного типу datatype, поміщаючи їх побайтне уявлення inbuf в вихідний буфер outbuf, починаючи з позиції position:

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

Функція MPI_Unpack витягує задане число елементів деякого типу з побайтного уявлення елементів у вхідному масиві:

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

Приклад розсилки різнотипних даних з 0-го процесу з використанням функцій MPI_Pack й MPI_Unpack наведено на рисунку 5.14.

```
    char buff[100];
    double x, y;
    int position, a[2];
    {
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
    { /* Пакування даних в 0-му процесі */
    position = 0;
    MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position,
MPI_COMM_WORLD);
    MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position,
MPI_COMM_WORLD);
    MPI_Pack(a, 2, MPI_INT, buff, 100, &position,
MPI_COMM_WORLD);
    }
    /*Розсилка пакованого повідомлення всім процесам
від 0-го */
    MPI_Bcast(buff, position, MPI_PACKED, 0,
MPI_COMM_WORLD);
    /* Розпакування повідомлення у всіх процесах */
    if (myrank != 0)
    position = 0;
    MPI_Unpack(buff, 100, &position, &x, 1,
MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buff, 100, &position, &y, 1,
MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buff, 100, &position, a, 2, MPI_INT,
MPI_COMM_WORLD);
    }
```

Рисунок 5.14 – Приклад розсилки різнотипних даних

5.6 Робота з групами процесів і комунікаторами

Часто в додатках виникає потреба обмежити область комунікацій деяким набором процесів, які складають підмножину загальної

сукупності процесів [70-75]. Для виконання будь-яких колективних операцій всередині цієї підмножини з них повинна бути сформована своя область зв'язку, що описується своїм комунікатором. Для вирішення таких завдань MPI підтримує два взаємопов'язаних механізми:

а) по-перше, існує набір функцій для роботи з *групами* процесів як впорядкованими множинами;

б) по-друге, є набір функцій для роботи з *комунікаторами* для створення нових комунікаторів як дескрипторів нових областей зв'язку.

Група є впорядкованою множиною процесів. Кожен процес ідентифікується змінною цілого типу. Ідентифікатори процесів утворюють безперервний ряд, що починається з 0. У MPI вводиться спеціальний тип даних `MPI_Group` і набір функцій для роботи зі змінними і константами цього типу.

Існує дві передвизначені групи:

- `MPI_GROUP_EMPTY` – група, яка не містить жодного процесу;

- `MPI_GROUP_NULL` – значення, що повертається в разі, коли група не може бути створена.

Створена група не може бути модифікована (розширена або усічена), може бути тільки створена нова група. Цікаво відзначити, що при ініціалізації MPI не створюється група, що відповідає комунікатору `MPI_COMM_WORLD`. Вона повинна створюватися явно.

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

При ініціалізації MPI створюється два наперед визначених комунікатора:

- `MPI_COMM_WORLD` – описує область зв'язку, що містить всі процеси;

- `MPI_COMM_SELF` – описує область зв'язку, що складається з одного процесу [28].

5.6.1 Основні функції роботи з групами

Нижче наведені лише деякі функції для роботи з групами. Список функцій, доступний в реалізації MS MPI, можна знайти в [71-72].

5.6.1.1 Конструктори груп

MPI не має механізму для формування групи з нуля, група може формуватися тільки на основі іншої, попередньо визначеної групи. Базова група, на основі якої визначаються всі інші групи, є групою, пов'язаною з початковим комунікатором `MPI_COMM_WORLD` (доступна через функцію `MPI_COMM_GROUP`).

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

Функція `MPI_Comm_group` повертає в `group` дескриптор групи, пов'язаної з комунікатором з `comm`:

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

Функція `MPI_Group_union` об'єднує задані групи в одній новій `newgroup`. Функція `MPI_Group_intersection` утворює нову групу

`newgroup` перетином двох груп, а функція `MPI_Group_difference` – різницею.

Об'єднання (union) – містить всі елементи першої групи (group1) і наступні за ними елементи другої групи (group2), що не входять в першу групу.

Перетин (intersect) – містить всі елементи першої групи, які також знаходяться в другій групі, впорядковані як в першій групі.

Різниця (difference) – містить всі елементи першої групи, які відсутні в другій групі, впорядковані як в першій групі.

Інші функції для створення груп [72]:

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

5.6.1.2 Доступ до групи

Функція `MPI_Group_size` використовується для визначення розміру `size` групи `group`:

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

Функція `MPI_Group_rank` використовується для визначення номера процесу `rank` в групі `group`:

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

Функція `MPI_Group_translate_ranks` виконує перетворення рангу процесу в одній групі в його ранг щодо іншої групи:

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

Функція `MPI_Group_translate_ranks` використовується для визначення відносної нумерації `ranks2` однакових процесів в двох різних групах `group1` і `group2`. Наприклад, якщо відомі номери деяких процесів в групі комунікатора `MPI_COMM_WORLD`, то можна дізнатися їх номери в підмножині цієї групи.

Параметри функцій:

1) `IN group1` – група 1;
2) `IN n` – кількість елементів в масивах `ranks1` та `ranks2`;
3) `IN ranks1` – масив з нуля або більшої кількості правильних номерів з групи 1;
4) `IN group2` – група 2;
5) `OUT ranks2` – масив відповідних номерів в групі 2, `MPI_UNDEFINED`, якщо відповідність відсутня.

5.6.1.3 Деструктор груп

Функція `MPI_Group_free` маркує об'єкт групи для видалення:

```
MPI_Group_free(MPI_Group *group).
```

Дескриптор `group` встановлюється в стан `MPI_GROUP_NULL`. Будь-яка операція, яка використовує цю групу, завершиться нормально.

5.6.2 Основні функції роботи з комунікаторами

Функції роботи з комунікаторами поділяються на функції *доступу до комунікаторів* і функції *створення комунікаторів*. Функції доступу є

локальними і не вимагають комунікацій, на відміну від функцій створення, які є колективними і можуть вимагати міжпроцесорних комунікацій. Нижче наведені деякі функції для роботи з комунікаторами. Список функцій, доступний в реалізації MS MPI, можна знайти в [73-74]. Деякі з них вже використовувалися.

Функція `MPI_Comm_size` визначає кількість процесів `size` в групі `comm`, функція `MPI_Comm_rank` вказує номер процесу, що її викликає:

```
    int MPI_Comm_size (MPI_Comm comm, int *size),
    int MPI_Comm_rank (MPI_Comm comm, int *rank).
```

Функція `MPI_Comm_compare` (як і подібна функція, але для порівняння груп `MPI_Group_compare`) порівнює задані комунікатори на еквівалентність:

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

Якщо члени комунікатора (групи) і їх порядок абсолютно однакові, повертається результат `result=MPI_IDENT`. Якщо однакові, але порядок різний, то повертається результат `MPI_SIMILAR`. В інших випадках повертається значення `MPI_UNEQUAL`.

Функції, що описані нижче, є колективними і викликаються всіма процесами в групі, пов'язаної з `comm`.

```
    MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm),
    MPI_Comm_create(MPI_Comm comm, MPI_Group group,
MPI_Comm *newcomm),
    MPI_Comm_split(MPI_Comm comm, int color, int
key, MPI_Comm *newcomm).
```

Функція `MPI_COMM_dup` дублює існуючий комунікатор `comm` разом з пов'язаними з ним значеннями ключів в новий – `newcomm`.

Функція `MPI_Comm_create` створює новий комунікатор `newcomm` з комунікаційною групою, визначеною аргументом `group` і новим контекстом. Запит повинен бути виконаний всіма процесами в `comm`, навіть якщо вони не належать новій групі. Цей запит застосовується тільки до *інтра-комунікаторів* [75].

Функція `MPI_Comm_split` ділить групу, пов'язану з `comm`, на непересічні підгрупи, по одній для кожного значення кольору `color`. Кожна підгрупа містить всі процеси того ж самого кольору. У межах кожної підгрупи процеси пронумеровані в порядку, визначеному значенням аргументу `key`, зі зв'язками, розділеними відповідно до їх номеру в старій групі. Для кожної підгрупи створюється новий комунікатор і повертається в аргументі `newcomm`. Процес може мати значення кольору `MPI_UNDEFINED`, тоді `newcomm` повертає `MPI_COMM_NULL`. Це колективна операція, але кожному процесу дозволяється мати різні значення для `color` і `key`.

Звернення до `MPI_COMM_CREATE (comm, group, newcomm)` еквівалентно зверненням до `MPI_COMM_SPLIT(comm, color, key, newcomm)`, де всі члени `group` мають `color=0` і `key`=номеру в `group`, і всі процеси, які не є членами `group`, мають `color = MPI_UNDEFINED`.

Функція `MPI_COMM_SPLIT` допускає більш загальний поділ групи на одну або кілька підгруп з необов'язковим зміненим упорядкуванням:

```
MPI_Comm_free (MPI_Comm *comm).
```

Деструктор `MPI_Comm_free` маркує комунікаційний об'єкт для видалення. Дескриптор встановлюється в `MPI_COMM_NULL`. Будь-які очікувані операції, які використовують цей комунікатор, будуть

завершуватися нормально; об’єкт фактично видаляється тільки в тому випадку, якщо немає ніяких інших активних посилань на нього.

5.7 Віртуальні топології процесів в MPI

Під фізичною топологією паралельної обчислювальної системи, як відомо, розуміється структура вузлів мережі і ліній зв'язку між цими вузлами. Зрозуміло, що фізична топологія є апаратно реалізованою і зміні не підлягає (хоча існують і програмні засоби побудови мереж). Але, залишаючи незмінною фізичну основу, можна організувати логічне уявлення будь-якої необхідної віртуальної топології. Для цього досить, наприклад, сформувати той чи інший механізм додаткової адресації процесів.

Віртуальна топологія – це механізм зіставлення процесів деякого комунікатора певній схемі адресації. У MPI всі топології віртуальні, тобто вони не пов’язані з реальною топологією комунікаційної мережі. Топологія процесів є одним з необов’язкових атрибутів комунікатора, такий атрибут може бути присвоєно лише intra-комунікаторам [76-77].

За замовченням передбачається *лінійна* топологія, в якій процеси пронумеровані в діапазоні від 0 до p-1, де p – число процесів в групі. Однак для багатьох завдань лінійна топологія неадекватно відображає логіку комунікаційних зв’язків між процесами. MPI надає засоби для створення досить складних віртуальних топологій у вигляді *графів*, де вершини є процеси, а ребра – канали зв’язку між ними [77].

Звичайно ж, слід розрізняти логічну топологію процесів, яку дозволяє формувати MPI, і фізичну топологію з’єднання процесорів у паралельній обчислювальній системі. В ідеалі логічна топологія процесів повинна враховувати як особливості алгоритму розв’язання задачі, так і фізичну топологію процесорів.

Для дуже широкого кола завдань найбільш адекватною топологією процесів є *двовимірна* або *тривимірна сітка* або їх замкнені еквіваленти – *тори*. Такі структури повністю визначаються числом вимірювань і кількістю процесів уздовж кожного координатного напрямку, а також способом накладення процесів на координатну сітку. У MPI, як правило, використовується *row-major* нумерація процесів, тобто використовується нумерація вздовж рядка на відміну від *column-major*, нумерації вздовж стовпців (рис. 5.15).

Узагальненням *лінійної* і *матричної* топологій на довільне число вимірювань є *декартова топологія* [77]. Для створення комунікатора з декартовою топологією використовується функція `MPI_Cart_create`. За допомогою цієї функції можна створювати топології з довільним числом вимірів, причому по кожному виміру окремо можна накладати періодичні граничні умови.

row-major	
0 (0,0)	**1 (0,1)**
2 (1,0)	**3 (1,1)**

column-major	
0 (0,0)	**2 (0,1)**
1 (1,0)	**3 (1,1)**

Рисунок 5.15 – Приклади нумерації процесів
за рядками (row-major) та стовпцями (column-major)

Таким чином, для *одновимірної* топології можемо отримати або *лінійну* структуру, або *кільце* в залежності від того, які граничні умови будуть накладені. Для *двовимірної* топології, відповідно, або *прямокутник*, або *циліндр*, або *тор*. Зауважимо, що не потрібна спеціальна підтримка для *гіперкубової* структури, оскільки вона являє собою *n-мірний тор* з двома процесами уздовж кожного координатного напрямку.

У MPI передбачено два типи топологій:

- *декартова топологія* (прямокутна сітка довільної розмірності);

- *топологія графа*.

5.7.1 Функції для роботи з комунікаторами декартової топології

Функція `MPI_Cart_create` – створює комунікатор з декартовою топологією, тобто повертає дескриптор нового комунікатора, до якого підключається топологічна інформація:

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

Параметри функцій `MPI_Cart_create`:

1) `IN comm_old` – вхідний (старий) комунікатор;

2) `IN ndims` – кількість вимірів у декартовій топології;

3) `IN dims` – цілочисельний масив розміром `ndism`, що визначає кількість процесорів у кожному вимірі;

4) `IN periods` – масив розміром `ndism` логічних значень, який визначає періодичність (`true` – періодичні, `false` – неперіодичні) у кожному вимірі;

5) `IN reorder` – логічна змінна, що вказує ранги пронумеровані (`true`) чи ні (`false`);

6) `OUT comm_cart` – комунікатор нової (створеної) декартової топології.

Якщо `reorder=false`, тоді ранг кожного процесора в новій групі ідентичний її рангу в старій групі, інакше функція може змінювати порядок процесів з метою оптимізації комунікацій.

Якщо повна розмірність декартової сітки менше, ніж розмір групи комунікаторів, то деякі процеси повертаються з результатом

MPI_COMM_NULL за аналогією з MPI_COMM_SPLIT. Виклик буде невірним, якщо він задає сітку більшого розміру, ніж розмір групи.

Функція MPI_Cart_create являється колективною: це означає, що вона повинна запускатись на всіх процесах, що належать вхідному (старому) комунікатору.

Функція MPI_Dims_create – визначення оптимальної конфігурації сітки, здійснює розбиття всіх процесів групи в N-мірну топологію:

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

Параметри функції MPI_Dims_create:

1) IN nnodes – кількість вузлів в сітці;

2) INOUT dims – цілочисельний масив розміром ndims, що визначає кількість вузлів за кожною розмірністю.

Елементи масиву dims описують декартову сітку координат з числом розмірностей ndims і загальною кількістю вузлів nnodes.

Кількість вузлів за розмірностями потрібно зробити якомога близькими одна одній, використовуючи відповідний алгоритм поділення вузлів між процесорами.

Підпрограма може далі обмежити виконання цієї функції, задаючи кількість елементів масиву dims. Якщо розмірність dims[i] є позитивним числом, функція не змінюватиме число вузлів в напрямку i; будуть змінені тільки ті елементи, для яких dims[i]=0.

Приклад впливу функції на масив dims показаний у таблиці 5.4.

Визначити декартові координати процесу за його рангом в групі можна за допомогою функції MPI_Cart_coords:

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

Таблиця 5.4 – Результат виклику функції

`dims` перед викликом	Виклик функції	`dims` після повернення функції
(0,0)	MPI_DIMS_CREATE(6,2,dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7,2,dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6,3,dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7,3,dims)	помилковий виклик

Функція `MPI_Cart_coords` переводить ранг процесу в координати процесу в декартовій топології.

Параметри функції `MPI_Cart_coords`:

1) `IN comm` – комунікатор з декартовою топологією;

2) `IN rank` – ранг процесора в топології `comm`;

3) `IN ndims` – максимальний розмір масивів `dims`, `periods` і `coords` в програмі;

4) `OUT coords` – цілочисельний масив, що визначає координати потрібного процесу в декартовій топології.

Функція зсуву даних `MPI_Cart_shift`:

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

Параметри функції `MPI_Cart_shift`:

1) `IN comm` – комунікатор з декартовою топологією;

2) `IN direction` – номер вимірювання (в топології), де робиться зміщення;

3) `IN disp` – напрямок зсуву (> 0 зміщення в бік збільшення номерів координати `direction`, <0 зміщення в бік зменшення номерів координати `direction`);

4) `OUT rank_source` – ранг процесу джерела;

5) `OUT rank_dest` – ранг процесу призначення.

Координати маркуються від 0 до `ndims-1`, де `ndims` – число розмірностей. Якщо використовується декартова топологія, то операцію `MPI_Sendrecv` можна виконати шляхом зсуву даних вздовж напрямку координати. В якості вхідного параметра `MPI_Sendrecv` використовує номер процесу-відправника для прийому і номер процесу-одержувача – для передачі.

Якщо функція `MPI_Cart_shift` виконується для декартової групи процесів, то вона передає процесу, що її викликає, ці номери, які потім можуть бути використані для `MPI_Sendrecv`. Користувач визначає напрямок координати і величину кроку (позитивний або негативний). Функція `MPI_Cart_shift` є локальною.

Функції `MPI_Cartdim_get` та `MPI_Cart_get` повертають інформацію про декартову топологію, яка була пов'язана з функцією `MPI_Cart_create`:

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

Зворотною дією по відношенню до `MPI_Cart_coords` володіє функція `MPI_Cart_rank`. З її допомогою можна визначити ранг процесу за його декартовими координатами у відповідному комунікаторі.

Для групи процесів з декартовою структурою функція `MPI_Cart_rank` переводить логічні координати `coords` процесів в номери `rank`, які використовуються в процедурах парного обміну:

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

Функція `MPI_Cart_sub` виконує розщеплення комунікатора на підгрупи, відповідні декартовим підсіткам меншого розміру. Якщо декартова топологія була створена за допомогою `MPI_Cart_create`, то функція `MPI_Cart_sub` може використовуватися для декомпозиції групи в підгрупи (які і є декартовими підсітками):

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

5.7.2 Функції для роботи з комунікаторами топології граф

Функція `MPI_Graph_create` являє собою конструктор універсальної (графової) топології:

```
    int MPI_Graph_create(MPI_Comm comm, int nnodes,
int *index, int *edges, int reorder, MPI_Comm
*comm_graph).
```

Параметри функції `MPI_Graph_create`:

1) `IN comm` – вхідний комунікатор;
2) `IN nnodes` – кількість вершин/вузлів графа;
3) `IN index` – масив цілочисельних значень, що описує ступені вершин;
4) `IN edges` – масив цілочисельних значень, що описує ребра графу;
5) `IN reorder` – номера можуть бути переупорядковані (true) чи ні (false);

6) `OUT comm_graph` – побудований комунікатор з графовою топологією.

Функція `MPI_Graph_create` передає дескриптор нового комунікатора, до якого приєднується інформація про графову топологію. Якщо `reorder=false`, то номер кожного процесу в новій групі ідентичний його номеру в старій групі. В іншому випадку функція може змінювати порядок процесів. Якщо розмір декартової сітки `nnodes` менше, ніж розмір групи комунікатора, то деякі процеси повертаються значення `MPI_COMM_NULL`. Виклик буде невірним, якщо він визначає граф більшого розміру, ніж розмір групи вихідного комунікатора.

Структуру графа визначають три параметри: `nnodes, index` і `edges`. `nnodes` – число вершин або вузлів графа, вузли маркуються від 0 до `nnodes-1`. `i`-тий елемент масиву `index` зберігає загальне число сусідів перших `i` вершин графа. Списки сусідів вершин `0,1,...,nnodes-1` зберігаються в послідовності осередків масиву `edges`. Загальна кількість об'єктів в `index` є `nnodes`, а загальне число елементів в `edges` дорівнює числу ребер графа.

Функції `MPI_Graphdims_get` та `MPI_Graph_get` відшукують графо-топологічну інформацію, яка була пов'язана з комунікатором за допомогою функції `MPI_Graph_create`:

```
    MPI_Graphdims_get (MPI_Comm comm, int *nnodes,
int *nedges),
    MPI_Graph_get (MPI_Comm comm, int nnodes, int
nedges, int *index, int *edges).
```

Інформація, яку надає `MPI_Graphdims_get`, може бути використана для коректного визначення розміру векторів `index` й `edges` для подальшого виклику функції `MPI_Graph_get`.

5.7.3 Функція для визначення типу топології

Функція `MPI_Topo_test` визначає тип топології, що пов'язана із комунікатором `comm`:

```
MPI_Topo_test (MPI_Comm comm, int *toptype).
```

Параметри функції `MPI_Topo_test`:

1) `IN comm` – комунікатор;
2) `OUT toptype` – тип топології.

Функція `MPI_Topo_test` через змінну `toptype` повертає наступні значення:

1) `MPI_CART` – для декартової топології;
2) `MPI_GRAPH` – для графової топології;
3) `MPI_INDEFINED` – для незаданої топології.

5.8 Завдання до самостійної роботи

1. Розробити паралельну програму, в якій кожен k-тий процес виводить на екран свій номер і повідомлення «Hello world», а всі інші процеси – тільки свій номер (k – номер варіанта студента).

2. Змоделювати послідовний обмін повідомленнями між 2 процесами протягом k секунд, де k – номер варіанту. Виміряти час на 1 ітерацію обміну, визначити залежність часу від довжини повідомлення.

3. Порівняти ефективність реалізації 2 різних видів пересилань даних між 2 процесами.

Таблиця 5.5

Варіант	Тип 1	Тип 2
1	Стандартний режим з блокуванням	Стандартний режим без блокування
2	Синхронна передача з блокуванням	Синхронна передача без блокування

Продовження таблиці 5.5

Варіант	Тип 1	Тип 2
3	Буферизована передача з блокуванням	Буферизована передача без блокування
4	Узгоджена передача з блокуванням	Узгоджена передача без блокування
5	Стандартний режим без блокування	Синхронна передача без блокування
6	Синхронна передача з блокуванням	Буферизована передача з блокуванням
7	Буферизована передача з блокуванням	Узгоджена передача з блокуванням
8	Узгоджена передача без блокування	Буферизована передача з блокуванням
9	Синхронна передача без блокування	Буферизована передача без блокування
10	Стандартний режим з блокуванням	Синхронна передача з блокуванням

4. Написати програму, в якій всі процеси передають деяке повідомлення наступному за номером процесу і приймають повідомлення від попереднього процесу (останній передає повідомлення першому). Після передачі кожен k -ий (номер варіанту) процес зупиняється (не діє), а інші продовжують ті ж операції передачі повідомлень по колу (2-я ітерація). Ітерації тривають, поки не залишаться $k-1$ процес.

5. Змоделювати бар'єрну синхронізацію за допомогою операцій типу «точка-точка» для виконання довільної задачі (наприклад, передача

повідомлення). Обчислити швидкість роботи і порівняти з аналогічною програмою, що використовує відповідні колективні операції.

6. Написати програму, що виконує підсумовування елементів матриці з використанням колективних функцій. Порівняйте її ефективність з програмою, яка виконує ту ж задачу без розпаралелювання. Реалізуйте аналогічний паралельний алгоритм, який використовує тільки операції «точка-точка», порівняйте ефективність трьох програм.

7. Розробити послідовну програму множення матриць. Розробити програмний додаток з використанням MPI відповідно з обраним/заданим для паралельного алгоритму матричного множення: Кеннона, Фокса, стрічкового з горизонтальним або вертикальним розбиттям, поліалгоритмів на базі рекурсивного швидкого множення Штрассена-Копперсмітта-Віноградa (в комбінації з алгоритмом Кеннона або Фокса на верхньому рівні та рекурсивного – на нижньому). Порівняти динамічні характеристики розроблених паралельних алгоритмів.

8. Реалізуйте операцію редукції для виконання довільної (не визначеної в MPI_Op) операції, наведеної в таблиці 5.4. Використовуйте функції MPI_Op_create, MPI_Op_free, MPI_Reduce (або деякий аналог редукції).

Таблиця 5.6

Варіант	Операція
1	Підсумовування модулів цілих чисел
2	Пошук максимального від’ємного елементу
3	Пошук мінімального залишку від ділення на задане число
4	Пошук мінімального позитивного елементу

Продовження таблиці 5.6

Варіант	Операція
5	Підсумовування парних елементів
6	Добуток від'ємних елементів
7	Пошук максимального за модулем числа

9. Розробити довільний суміжний (безперервний) і векторний тип даних, привести приклади їх використання.

10. Створити тип даних «Точка» (на площині). Реалізувати паралельну програму, що дозволяє визначити і вивести відстань між кожною парою точок, мінімальне з отриманих значень і довжину довільного шляху, що з'єднує всі точки.

11. Реалізуйте трансляцію і деяку обробку власного типу даних за допомогою відповідних комунікаційних операцій. Перетворіть алгоритм обраного завдання, використавши операції пакування/розпаковування. Порівняйте ефективність програм.

12. Створіть три комунікатора. Перший комунікатор нехай включає всі процеси, другий – тільки парні, а третій – непарні. У межах кожного комунікатора створіть 2-3 групи, використовуючи різні конструктори груп. Нехай k-ий процес кожної групи, де k – номер варіанту за списком групи студенту, виведе на екран кількість процесів в своїй групі. Порівняйте між собою групи першого комунікатора на рівність, відобразіть результат.

13. Розбити всі процеси додатку випадковим чином на $k+3$ довільних груп і надрукувати ранги в `MPI_COMM_WORLD` тих процесів, що потрапили в першу групу, але не потрапили в інші.

14. Написати програму, в якій з усіх процесів `MPI_COMM_WORLD` відібрати випадковим чином k процесів в першу групу, потім, таким же

випадковим чином, в другу, де k – номер варіанта + 5. На основі отриманих груп створити нові, використовуючи операції об'єднання, перетину та різниці. Відобразити ранги процесів `MPI_COMM_WORLD`, що потрапили в кожну з 5 груп.

15. Написати програму, в якій виконати розбиття процесів на дві групи, в одній реалізувати обмін повідомленнями між усіма процесами в декартовій топології, а в іншій – в топології графа. Виконати декілька ітерацій, порівняти швидкість виконання завдання в різних топологіях.

5.9 Контрольні питання

1. Дати визначення MPI. Які стандарти та реалізації MPI відомі?

2. Що таке процес? група процесів? комунікатор?

3. Які поля включає структура MPI_Status? У яких випадках вони використовуються? Між викликом яких процедур повинен розташовуватися паралельний код MPI? Які у них аргументи?

4. Які базові функції MPI вам відомі? На які групи можна розділити операції взаємодії між двома процесами?

5. Які існують режими передачі повідомлень з блокуванням і без блокування? У чому їх відмінності?

6. Назвіть відомі функції передачі і прийому повідомлень. Як визначити точне число елементів у прийнятому повідомленні? Що таке параметри-джокери?

7. Як (за допомогою якої функції, структури) можна отримати інформацію про очікуване повідомлення?

8. Які допоміжні функції і з якою метою використовують при асинхронній передачі даних? Яким чином можна реалізувати передачу повідомлення від одного процесу – всім іншим?

9. Що таке операція редукції? Як вона виконується? Які колективні операції можна застосовувати в редукції? Які функції за це відповідають?

10. Як реалізувати операцію взаємодії процесів «усі-усім»? За допомогою яких функцій реалізується розподіл і збір даних? Яка функція дозволяє реалізувати синхронізацію процесів при колективній розсилці?

11. Якими способами можна передавати дані різних типів (структури,…) або дані, розташовані в несуміжних областях пам'яті?

12. Який порядок використання довільних типів даних? Які існують конструктори довільних типів? У чому їх відмінності?

13. Для чого потрібні пакування/розпакування даних? Які функції їх реалізують?

14. Які існують способи роботи з підмножинами процесів? Що таке група? Які наперед визначені групи існують?

15. Що таке комунікатор? Які наперед визначені комунікатори ви знаєте? Які функції реалізують роботу з групами і комунікаторами?

16. Які типи конструкторів груп існують? Що відбувається при виклику деструктора групи/комунікатора?

17. Що таке логічна і фізична топологія мережі? Які топології реалізовані в MPI? У чому їх особливості? Яка топологія використовується в MPI за замовчуванням?

18. Які функції дозволяють реалізувати топологію декартової сітки, графа? Як реалізувати топологію гіперкуба або кільця в MPI?

ПЕРЕЛІК ЛІТЕРАТУРИ

1.Воеводин В.В., Воеводин Вл. В. Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608с.

2. Гергель В.П. Теория и практика параллельных вычислений. – Москва: Бином. Лаборатория знаний, 2007. – 423с.

3. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие. – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2003. – 184 с.

4. Grama A., Gupta A., Karypis G., Kumar V. Introduction to Parallel Computing. – Addison Wesley, 2003. – 856p.

5. Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. – СПб.: БХВ-Петербург, 2002. – 396с.

6. Барский А.Б. Параллельные информационные технологии. – Москва: ИУИТ. Бином. Лаборатория знаний, 2007. – 503с.

7. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем. – СПб.: Питер, 2004. – 668с.

8. Богачев К.Ю. Основы параллельного программирования. – М.: Бином. Лаборатория знаний, 2003. – 342с.

9. Букатов А.А., Дацюк В.Н., Жегуло А.И. Программирование многопроцессорных ВС. – Ростов-на Дону: ООО ЦВВР, 2003. –208с.

10. Информационно-аналитические материалы по параллельным вычислениям научно-исследовательского вычислительного центра (НИВЦ) МГУ [Електронний ресурс]. – Режим доступу: http://parallel.ru.

11. Дмитрієва О.А. Паралельне моделювання динамічних об’єктів зі сконцентрованими параметрами. – Харків: Ноулідж, 2014. – 336с.

12. Числові методи моделювання динамічних об'єктів в мультипроцесорних системах / О.А. Дмитрієва, Н.Г. Гуськова, Є.О. Башков, І.А. Назарова: монографія. – Покровськ: ДВНЗ «ДонНТУ», 2020. – 268 с.

13. Dmitrieva O., Feldman L. Parallel Step Control. Development of parallel algorithms of the step variation for simulation of stiff dynamic systems. – Lambert Academic Publishing, 2013. – 72 p.

14. Дмитрієва О. А. Паралельні різницеві методи розв'язання завдання Коші. – Донецьк: ДонНТУ, 2011. – 265 с.

15. Фельдман Л.П., Назарова И.А. Современные параллельные методы численного решения задачи Коши. – Донецк: ГВУЗ “ДонНТУ”, 2013. – 206с.

16. Паралельні однокрокові методи чисельного розв’язання задачі Коші: монографія / Л.П. Фельдман, І.А. Назарова. – Донецьк: «ДВНЗ» ДонНТУ, 2011. – 185 с.

17. Flynn M. Very high-speed computing systems. Proceeding of the IEEE №54 (12), 1966. – P.1901-1909.

18. Flynn M. Some Computer Organisations and Their Effectiveness // IEEE Trans. Computers. 1972. – V.21. N 9. – P.948-960.

19. Хокни Р., Джессхоуп К. Параллельные ЭВМ: Архитектура, программирование и алгоритмы. – М.: Радио и связь, 1986. – 392с.

20. Корнеев В.В. Параллельные вычислительные системы. – М.: Нолидж, 1999. – 320с.

21. Foster I. Designing and Building Parallel Programs. – Addison-Wesley, 1999. – 302с.

22. Классификация архитектур вычислительных систем [Електронний ресурс]. – Режим доступу: https://parallel.ru/computers/taxonomy/

23. Основные классы современных параллельных компьютеров [Електронний ресурс]. – Режим доступу: http://parallel.ru/computers/classes.html

24. Андерсон Дж. Дискретная математика и комбинаторика. – М.: Мир, 2001. – 960с.

25. Amdahl G. Validity of the single processor approach to achieving large scale computing capabilities // In AFIPS Conference proceeding, Washington, D.C.: Thompson Books, Vol. 30. – P.483-485.

26. Gustavson J.L. Reevaluating Amdahl's law // Communications of the ACM. 31(5). – P. 532-533.

27. Grama A., Gupta A., Kumar V. Isoefficiency: Measuring the scalability of parallel algorithms and architectures // IEEE Parallel and Distributed technology, 1993. – P. 12-21.

28. Kumar V., Gupta A. Analyzing scalability of parallel algorithms and architectures // Journal of Parallel and Distributed Computing, 22(3), 1994. – P. 379-391(2nd edn., 2003).

29. Назарова И.А. Оценка масштабируемости параллельного решения СОДУ // Збірник наукових праць. Матеріали міжнародної наукової конференції «Моделювання -2010». Київ. 2010. т.2. – С. 282-290.

30. Назарова И.А., Фельдман Л.П. Масштабируемость параллельных алгоритмов и архитектур при численном решении задачи Коши на основе явных разностных схем // Наукові праці Донецького національного технічного університету. Серія: Інформатика, кібернетика та обчислювальна техніка, випуск №1 (24): – Покровськ, ДонНТУ, 2017. – С. 100-108.

31. Назарова І.А., А.О. Чорна Дослідження масштабованості паралельних систем на основі функції ізоефективності // Наукові праці Донецького національного технічного університету. Серія: Інформатика,

кібернетика та обчислювальна техніка, випуск №2 (21): – Красноармійськ, ДонНТУ, 2015. – С. 56-62.

32. Назарова И.А. Анализ масштабируемости параллельных алгоритмов численного решения задачи Коши // Наукові праці Донецького національного технічного університету. Серія: Інформатика, кібернетика та обчислювальна техніка, випуск №10 (153): – Донецьк, ДонНТУ, 2009. – С. 21-26.

33. Голуб Д., Ван Лоун Ч. Матричные вычисления: Пер. с англ. – М.: Мир, 1999. – 548с.

34. Деммель Дж. Вычислительная линейная алгебра. Теория и приложения. Пер. с англ. – М.: Мир, 2001. – 430с.

35. Demmel J., Higham N.J. Stability of block algorithms with fast Level 3 BLAS // ACM Trans. Math. Software, 18, 1992. – P. 274-291.

36. Choi J., Dongarra J., Walker D. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. – Режим доступа: http://citeseer.ist.psu.edu/choi93pumma.html

37. Фельдман Л.П., Назарова І.А., Шматько А.Е. Паралельний блоковий рекурсивно-систолічний метод реалізації матричного добутку // Материалы международной научно-технической конференции “Искусственный интеллект. Интеллектуальные системы: ИИ-2008. – Таганрог-Донецк-Минск: Изд-во ТРТУ, т.2, 2008. – С. 380–382.

38. Шматько О.Є., Назарова І.А., Фельдман Л.П. Бібліотека паралельного чисельного аналізу: масштабовані алгоритми блокового матричного добутку // Матеріали V всеукраїнської науково-технічної конференції студентів, аспірантів та молодих вчених. – Донецьк: ДонНТУ, 2009. – С. 368-370.

39. Strassen V. Gaussian elimination is not optimal // Numer. Math. 13. – P. 354-356.

40. Pan V. How can we speed up matrix multiplication // SIAM Rev., 26, 1984. – P. 393-416.

41. Winograd S. A new algorithm for inner product // IEEE Trans. Comp. C-17. – P. 693-694.

42. Bailey D.H. Extra high-speed matrix multiplication on CRAY-2 // SIAM J. Sci. and Stat. Comp. 9. – P. 603-607.

43. Bailey D.H., Lee K., Simon H.D. Using Strassen's algorithm to accelerate the solution of linear systems. J. Supercomputing, 4: 97-371, 1991.

44. Назарова І.А. Підвищення ефективності паралельного розв'язання лінійної задачі Коші на основі методу рекурсивного множення матриць // Научно-теоретический журнал ИПИИ НАН Украины «Искусственный интеллект», №3, 2008. – Донецк: ИПИИ, 2008. – С. 706-714.

45. Сайт MPI Forum [Електронний ресурс]. – Режим доступу: https://www.mpi-forum.org/ (дата звернення: 12.10.2020).

46. MPICH (MPICH 1.4.1 – остання версія, що підтримується для Windows) [Електронний ресурс]. – Режим доступу: http://www.mpich.org/static/downloads/1.4.1p1/

47. MS MPI: веб-сайт. URL: https://blogs.msdn.microsoft.com/ru-hpc/2009/12/28/ms-mpi-visual-studio-windows-hpc-server/

48. MS MPI [Електронний ресурс]. – Режим доступу: https://blogs.technet.microsoft.com/windowshpc/2015/02/02/how-to-compile-and-run-a-simple-ms-mpi-program/

49. MS MPI [Електронний ресурс]. – Режим доступу: http://www.math.ucla.edu/~mputhawala/PPT.pdf

50. Документація MS MPI [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473458(v=vs.85).aspx

51. Посилання на навчальні та інші матеріали по курсу MPI (документація, література, реалізації та ін.) [Електронний ресурс]. – Режим доступу: http://parallel.ru/tech/tech_dev/mpi

52. Приклади MPI програм [Електронний ресурс]. – Режим доступу: http://parallel.ru/vvv/examples.html

53. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие. – М.: Изд-во МГУ, 2004. – 71с.

54. Антонов А. С. Вычислительный практикум по технологии MPI [Електронний ресурс]. – Режим доступу: http://parallel.ru/tech/tech_dev/MPIcourse

55. Можливі значення MPI_Datatype [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473290(v=vs.85).aspx

56. Функції MS MPI для реалізації передачі повідомлень між двома процесами («точка-точка») [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473446(v=vs.85).aspx

57. Структура MPI_Status [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473475(v=vs.85).aspx

58. Колективні операції MS MPI [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473253(v=vs.85).aspx

59. Колективні операції [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node67.html, http://www.intuit.ru/studies/courses/584/440/lecture/9854?page=3

60. Можливі значення MPI_Comm [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473254(v=vs.85).aspx

61. Операції, що визначаються користувачем [Електронний ресурс]. – Режим доступу:
https://msdn.microsoft.com/en-us/library/dn520588(v=vs.85).aspx,
https://www.opennet.ru/docs/RUS/mpi-1/node84.html

62. Можливі значення MPI_Op [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473436(v=vs.85).aspx

63. Приховані типи даних [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node17.html

64. Похідні типи даних [Електронний ресурс]. – Режим доступу:
https://www.opennet.ru/docs/RUS/linux_parallel/node153.html
https://www.opennet.ru/docs/RUS/mpi-1/node57.html
http://www.ccas.ru/mmes/educat/lab04k/07/Derived.html

65. Функції MS MPI для роботи з типами даних [Електронний ресурс]. – Режим доступу:
https://msdn.microsoft.com/en-us/library/dn473291(v=vs.85).aspx

66. Пакування і розпакування даних [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node66.html

67. Приклад використання довільного типу даних [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node84.html

68. Похідні типи даних і передача пакованих даних [Електронний ресурс]. – Режим доступу: http://rsusu1.rnd.runnet.ru/tutor/method/m2/page14.html

69. Групи, контексти і комунікатори [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node90.html
http://www.intuit.ru/studies/courses/4447/983/lecture/14927?page=7#sect39
https://parallel.ru/sites/default/files/info/parallel/antonov/groups.pdf

70. Функції MS MPI для роботи з групами [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473399(v=vs.85).aspx

71. Конструктори груп [Електронний ресурс]. – Режим доступу:https://www.opennet.ru/docs/RUS/mpi-1/node101.html

72. Функції MS MPI для роботи з комунікаторами [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473255(v=vs.85).aspx

73. Комунікатори: базові концепції [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node94.html

74. Інтра-комунікатори [Електронний ресурс]. – Режим доступу: https://www.opennet.ru/docs/RUS/mpi-1/node97.html

75. Функції MS MPI для роботи з топологіями [Електронний ресурс]. – Режим доступу: https://msdn.microsoft.com/en-us/library/dn473449(v=vs.85).aspx

76. Топології процесів. Декартова топологія, графи. [Електронний ресурс]. – Режим доступу:

https://www.opennet.ru/docs/RUS/mpi-1/node123.html,

http://www.intuit.ru/studies/courses/4448/984/lecture/14945?page=2

77. Завантажити MS-MPI та MS-MPI SDK (на даний момент остання версія MS-MPI v10.1.2) [Електронний ресурс]. – Режим доступу:

https://msdn.microsoft.com/en-us/library/bb524831.aspx

78. Розширені інструкції по налаштуванню MS MPI (настройка, запуск на кластері, додаткові інструменти, інтеграція з Visual Studio [Електронний ресурс]. – Режим доступу: https://blogs.technet.microsoft.com/windowshpc/2015/02/02/how-to-compile-and-run-a-simple-ms-mpi-program/

ПРЕДМЕТНИЙ ПОКАЖЧИК

А

Б

В

Г

Д

Е

З

І

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ш

Щ

Я

A

B

C

D

M

N

O

P

Q

R

S

T

U

V

W

Додаток А

Установка й настройка MS MPI в середовищі Windows

Для розробки паралельних програм за допомогою MS-MPI в середовищі Windows необхідно встановити і налаштувати відповідне програмне забезпечення.

Скачайте файли MSMpiSetup.exe і msmpisdk.msi з офіційного сайту [78-79] і встановіть. Параметри установки можна залишити за замовчуванням. На поточний момент актуальна версія 8.1.

Після установки можна перевірити, що змінні оточення правильно встановлені, виконавши команду `set MSMPI` в командному рядку (запущеному від імені адміністратора). Значення змінних будуть використані для настройки проекту. Команда має відобразити значення, наведені на рисунку А.1.

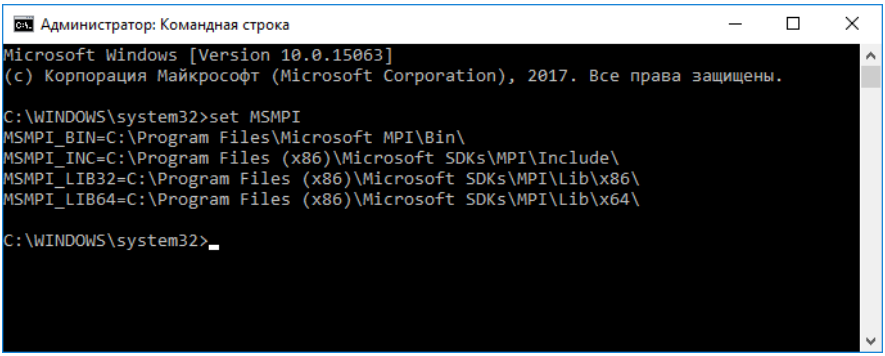


Рисунок А.1 – Перевірка змінних оточення

Запустіть Visual Studio і створіть новий консольний додаток. Назвемо його, наприклад, `MpiTest`. Переконайтеся, що значення

параметра «Платформа рішення» (x64 або x86) відповідає потрібній конфігурації, вибір платформи показаний на рисунку А.2.

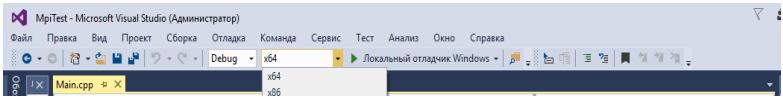


Рисунок А.2 – Вибір платформи рішення

Перейдіть у властивості проекту, щоб налаштувати MPI. Відкрийте вкладку «Каталоги VC++» (VC++ Directories) і в полі «Каталоги включення» (Include Directories) задайте шлях до заголовних файлів MS MPI (див. значення змінної `MSMPI_INC`): C:\Program Files (x86)\Microsoft SDKs\MPI\Include\.

У цій же вкладці в поле «Каталоги бібліотек» (Library Directories) додайте шлях до бібліотеки MPI (див. значення змінної `MSMPI_LIB64` або `MSMPI_LIB32` в залежності від платформи, для якої буде будуватися проект), в нашому випадку було задано значення C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64\, як показано на рисунку А.3.

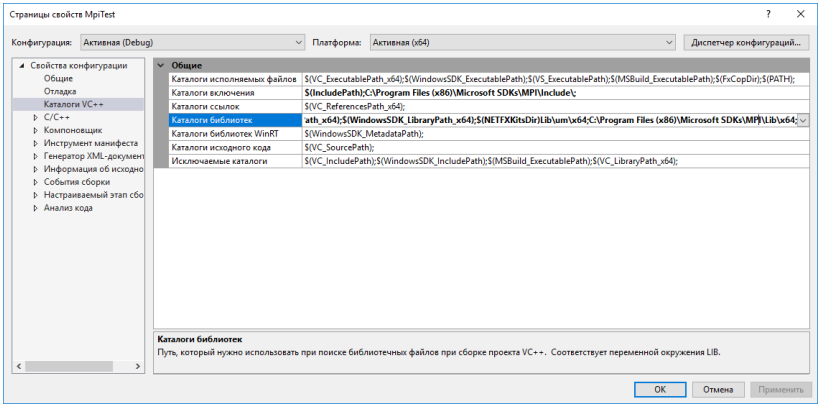


Рисунок А.3 – Налаштування каталогів, що підключаються

Перейдіть у вкладку «Компоновщик» -> «Всі параметри» (Linker -> All Options). Тут в поле «Додаткові залежності» (Additional Dependencies) додайте значення `msmpi.lib`, як наведено на рисунку А4.

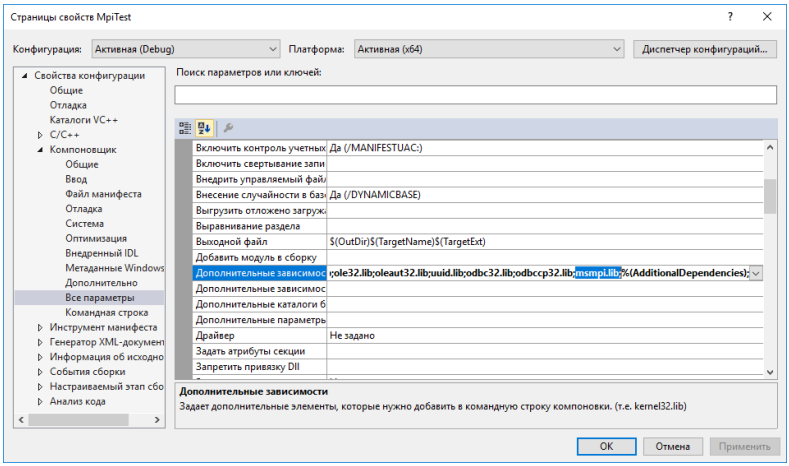


Рисунок А.4 – Налаштування параметрів компоновщика

Щоб переконатися, що необхідне ПЗ успішно встановлено і готове до роботи, напишемо і скомпілюємо невелику тестову програму, що наведена на рисунку А.5. Програма виконує передачу повідомлення “Hello” процесу з номером 0 від всіх інших процесів. Процес 0 перевіряє довжину отриманого повідомлення та виводить його на екран.

```
#include <mpi.h>          // заголовок бібліотеки MS MPI
#include <iostream>       // бібліотека введення-виведення
using namespace std;
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv); // ініціалізація паралельної частини програми
    int rank, size;
```

Рисунок А.5 – Тестова програма

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // визначення номера поточного процесу
MPI_Comm_size(MPI_COMM_WORLD, &size); // визначення кількості процесів
    if (rank) // передача повідомлення «Hello» процесами 1..n-1 процесу 0
    {
        char buf[] = "Hello!";
        MPI_Send(buf, sizeof(buf), MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
    else {  // отримання і виведення повідомлення нульовим процесом
        cout << "Process 0 started" << endl;
        for (int i(1); i<size; ++i)
        {
            MPI_Status s;
            // перевірити вхідні повідомлення
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &s);
            int count;
            // дізнатися обсяг повідомлення
            MPI_Get_count(&s, MPI_CHAR, &count);
            char *buf = new char[count];
            // отримати вхідне повідомлення
            MPI_Recv(buf, count, MPI_CHAR,
                MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &s);
            // вивести отримане повідомлення
            cout << "Message from process " << s.MPI_SOURCE << ": "
                << buf << endl;
            delete[] buf;
        }
        cout << "Done." << endl;
    }
    MPI_Finalize();
    return 0;
}
```

Рисунок А.5, лист 2

Скомпілювати додаток можна стандартним способом в Visual Studio. Запуск програми можна виконати через командний рядок наступним чином:

mpiexec –n < кількість процесів > < назва програми >

У даному випадку запускається програма MpiTest.exe з 4 паралельними процесами, що видно з рисунку А.6. Процеси 1-3 передають повідомлення «Hello» нульовому процесу, який отримує їх і виводить на екран.

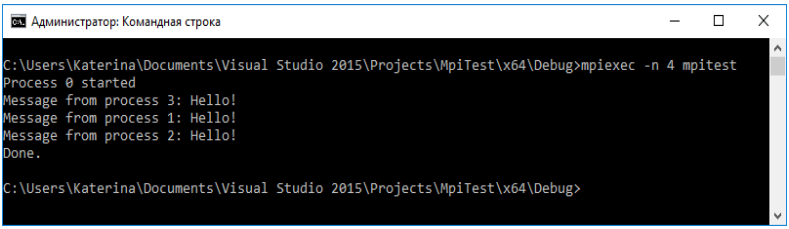


Рисунок А.6 – Запуск і результат роботи програми

Для того, щоб дізнатися про основні параметри команди mpiexec, виконайте її без будь-яких параметрів. Для отримання повної довідки по команді, виконайте mpiexec /help2.

Додаток Б
Довідка з MPI

Таблиця Б.1 містить довідку по функціям, командам та атрибутам MPI [46-77].

Таблиця Б.1 – Довідка з MPI

Команда/функція/атрибут	Сторінка
MPI_Allgather	176
MPI_Allgatherv	176
MPI_Allreduce	177
MPI_Alltoall	177
MPI_Alltoallv	177
MPI_ANY_SOURCE	159
MPI_ANY_TAG	159
MPI_Barrier	176
MPI_Bcast	175
MPI_Bsend	164
MPI_Buffer_attach	165
MPI_Buffer_detach	165
MPI_BYTE	162
MPI_CART	209
MPI_Cartdim_get	206
MPI_Cart_coords	204
MPI_Cart_create	203
MPI_Cart_get	206
MPI_Cart_rank	207

Продовження таблиці Б.1

Команда/функція/атрибут	Сторінка
MPI_Cart_shift	205
MPI_Cart_sub	207
MPI_Comm_compare	199
MPI_Comm_dup	199
MPI_Comm_group	196
MPI_COMM_RANK	157
MPI_COMM_SIZE	156
MPI_COMM_SELF	196
MPI_Comm_split	199
MPI_COMM_WORLD	196
MPI_Dims_create	204
MPI_Ibsend	164
MPI_INDEFINED	209
MPI_INIT	156
MPI_Irecv	164
MPI_Irsend	164
MPI_Isend	164
MPI_Issend	164
MPI_Finalize	156
MPI_Gather	176
MPI_Gatherv	176
MPI_GET_COUNT	160
MPI_GRAPH	209
MPI_Graph_create	207
MPI_Graphdims_get	208

Продовження таблиці Б.1

Команда/функція/атрибут	Сторінка
MPI_Graph_get	208
MPI_Group	195
MPI_Group_difference	196,197
MPI_GROUP_EMPTY	195
MPI_Group_excl	197
MPI_Group_compare	199
MPI_Group_free	198
MPI_Group_incl	197
MPI_Group_intersection	196
MPI_Group_range_excl	197
MPI_Group_range_incl	197
MPI_GROUP_NULL	195,198
MPI_Group_rank	197
MPI_Group_size	197
MPI_Group_translate_ranks	198
MPI_Group_union	196
MPI_Op	186
MPI_PACKED	162
MPI_Pack	192-194
MPI_Pack_size	193
MPI_Probe	168
MPI_RECV	157
MPI_Reduce	177
MPI_Reduce_scatter	177
MPI_Rsend	164

Продовження таблиці Б.1

Команда/функція/атрибут	Сторінка
MPI_Scan	177
MPI_Scatter	176
MPI_Scatterv	176
MPI_Send	158
MPI_Sendrecv	174
MPI_Sendrecv_replace	175
MPI_Ssend	164
MPI_Test	171
MPI_Testall	173
MPI_Testany	173
MPI_Testsome	173
MPI_Topo_test	209
MPI_Type_commit	188,192
MPI_Type_contiguous	188,189
MPI_Type_extent	188,189
MPI_Type_free	188,192
MPI_Type_hindexed	188,191
MPI_Type_hvector	188,191
MPI_Type_indexed	188,191
MPI_Type_size	188,189
MPI_Type_struct	188,191
MPI_Type_vector	188-190
MPI_Unpack	193,194
MPI_Wait	171
MPI_Waitall	173

Продовження таблиці Б.1

Команда/функція/атрибут	Сторінка
`MPI_Waitany`	173
`MPI_Waitsome`	173
`MPI_WTIME`	160
`MPI_WTICK`	160

Навчальне видання

Назарова Ірина Акопівна
Дмитрієва Ольга Анатоліївна

Паралельні обчислення
Навчальний посібник

Робота друкується в авторській редакції

Формат 60x84/16. Ум. друк. арк. 14. Обл.-вид. 8
Тираж 300 прим. Замовлення № 6914

Видавець: Державний вищий навчальний заклад «Донецький національний технічний університет». пл. Шибанкова, 2, м. Покровськ Донецької обл., 85300.

Свідоцтво про державну реєстрацію суб'єкта видавничої справи:
серія ДК № 4911 від 09.06.2015.

Віддруковано з оригіналів замовника.
ФОП Корзун Д.Ю.
21027, м. Вінниця, вул. Келецька, 51а, прим. 143.
Тел.: (0432) 603-000, (096) 97-30-934, (093) 89-13-852.
e-mail: info@tvoru.com.ua
http://www.tvoru.com.ua