МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

РОЛЬЩИКОВ В.Б.

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ (ЗМІСТОВНИЙ МОДУЛЬ №1)

Конспект лекцій

Одеса
Одеський державний екологічний університет
2018

УДК 681.518
Р68

Рекомендовано методичною радою Одеського державного екологічного університету Міністерства освіти і науки України як конспект лекцій (протокол №7 від 31.03. 2016 р.)

Рольщиков В. Б.
Технології розподілених систем та паралельних обчислень. Змістовний модуль №1: конспект лекцій. Одеса, Одеський державний екологічний університет, 2018. 181 с.

В конспекті лекцій з курсу «Технології розподілених систем та паралельних обчислень» розглянуті основні питання архітектури систем паралельних обчислень. Паралельні обчислення є перспективною (і дуже привабливою) областю застосування обчислювальної техніки і являють собою складну науково-технічну область діяльності. Тим самим, знання сучасних тенденцій розвитку комп'ютерів і апаратних засобів для досягнення паралелізму, уміння розробляти моделі, методи й програми паралельного рішення завдань обробки даних варто віднести до числа важливих кваліфікаційних характеристик сучасного фахівця із прикладної математики, інформатиці й обчислювальній техніці.

Конспект лекцій призначений для студентів четвертого курсу Одеського державного екологічного університету, які навчаються за кваліфікаційним рівнем «бакалавр» та спеціальністю 07.05010101 «Інформаційні управляючі системи та технології (за галузями)».

ISBN 978-966-186-084-0

© Рольщиков В. Б., 2018
© Одеський державний екологічний університет, 2020

ЗМІСТ

Перелік скорочень, умовних позначень і термінів 5
Вступ 7
Сфери застосування паралельних обчислень і суперкомп'ютерів 9
Питання і завдання для самоперевірки 20
1 Паралельні обчислювальні методи 22
1.1 Організація паралельних обчислень з використанням наявних технологій (PVM, MPI) 29
1.2 Паралельні перетворення арифметичних виразів 41
1.3 Питання і завдання для самоперевірки 83
2 Базові алгоритми паралельних обчислень 89
2.1 Алгоритми паралельного обчислення рекурентних співвідношень 89
2.1.1 Алгоритм викреслювання стовпців 90
2.1.2 Алгоритм логарифмічного підсумовування (алгоритм здвоювання) 94
2.1.3 Алгоритм рекурентного добутку 96
2.1.4 Блоковий алгоритм 97
2.2 Паралельні методи розв'язання СЛАР 98
2.2.1 Метод простої ітерації розв'язання СЛАР 99
2.2.2 Метод Гаусса-Зейделя розв'язання СЛАР 106
2.3 Паралельні алгоритми роботи з матрицями 115
2.3.1 Обчислення добутку матриці на вектор 115
2.3.2 Самоплануючий алгоритм множення матриць 118
2.3.3 Клітинний алгоритм множення квадратних матриць 120
2.4 Питання і завдання для самоперевірки 127
3 Паралельні методи розв'язання систем нелінійних рівнянь 130
3.1 Паралельне розв'язання нелінійних рівнянь 130
3.2 Вирішення питання паралельного розв'язання системи нелінійних рівнянь 131

3.3 Питання і завдання для самоперевірки 135
4 Ефективність паралельних обчислювальних методів під час розв'язання нелінійної задачі Коші для ЗДР 137
4.1 Питання і завдання для самоперевірки 145
5 Паралельні методи чисельного розв'язання жорстких ЗДР та їх реалізація в багатопроцесорних структурах... 146
5.1 Питання і завдання для самоперевірки 152
6 Паралельні алгоритми чисельного розв'язання задачі Пуассона............... 154
6.1 Питання і завдання для самоперевірки 171
Алфавітний покажчик.. 173
Перелік ілюстрацій.. 173

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

БД – бази даних

ВІЛ – вірус імунодефіциту людини

ГіБ – Гібібайт – 2^{30} байтів

Дедлок – тупикова ситуація в паралельній програмі, яка виникає через конфлікт потоків (процесів) при володінні суспільним ресурсом

ЕОМ – електронна обчислювальна машина

ЗДР – звичайне диференціальне рівняння

ІМ – інформаційна модель

ІТ – інформаційні технології

Мегафлопс – 10^6 операцій із плаваючою крапкою в секунду

МДУ – Московський державний університет

МСЦ РАН – Міжвідомчий суперкомп’ютерний центр Російської академії наук

ОМ – операційна модель

ОС – операційна система

Петафлопс –10^{12} операцій із плаваючою крапкою в секунду

ПК – персональний комп’ютер

РАН Російська академія наук

СЗДР – система звичайних диференціальних рівнянь

СКБД – система керування базами даних

СЛАР – система лінійних алгебраїчних рівнянь

СНАР – система нелінійних алгебраїчних рівнянь

СОІ система оборонної ініціативи США

Терафлопс – 10^9 операцій із плаваючою крапкою в секунду

ТіБ – Тебібайт – 2^{40} байтів

Флопс – одиниця вимірювання швидкодій комп’ютерів – одна операція з плаваючою крапкою в секунду

ЦНМРК – цілком неявні методи типу Рунге-Кутти

ЯПФ – ярусно-паралельна форма програми

API – Application Program Interface – інтерфейс створення програмних застосувань

BMB – Beijing Meteorological Bureau – Пекінське метеорологічне бюро

FLOPS – one floating point operation per second (див. флопс)

MFLOPS – Mega floating point operations per second (див. Мегафлопс)

MPI – Message Passing Interface – інтерфейс передачі повідомлень

MPMD Multiple Program, Multiple Data stream – багато паралельних програм, багато потоків даних

NASA – національне агентство з космічних досліджень

PFLOPS – Peta floating point operations per second (див. Петафлопс)

POSIX – Portable OS Interface based on uniX) – стандарт для операційних систем, оснований на операційній системі UNIX

PVM – Parallel Virtual Machine – паралельна віртуальна машина

SPMD – Single Program, Multiple Data stream – одна паралельна програма, багато потоків даних

SQL – Structured Query Language – мова структурованих запитів

TFLOPS – Tera floating points operations per second (див. Терафлопс)

TID – Task IDentifier – ідентифікатор задачі

UML – Unified Modeling Language – уніфікована мова моделювання

ВСТУП

Мета: надати студентам основні відомості про архітектуру сучасних паралельних обчислювальних систем, принципи функціонування окремих складових систем та структуру взаємозв'язків між складовими. Ознайомити студентів з методами розпаралелювання різних чисельних методів. Навчити студентів працювати із засобами паралельного програмування як для суперкомп'ютерів, локальних обчислювальних мереж, так і для GRID-систем.

Завдання: завданням дисципліни є роз'яснення найбільш складних питань дисципліни шляхом читання курсу лекцій, винесенням менш складних питань на їхнє самостійне вивчення під час самостійної роботи студентів, а, також, закріплення теоретичного матеріалу за допомогою і під час виконання циклу лабораторних робіт, котрі охоплюють найбільш важливі теми дисципліни.

За результатами вивчення навчальної дисципліни студент повинен:

знати: архітектуру сучасних паралельних обчислювальних систем як мережевих, в тому числі й GRID, так й багатопроцесорних систем з пам'яттю загального використання, основні принципи розпаралелювання задач та паралельні алгоритми, які застосовуються для вирішення складних задач моделювання в науці і техніці;

вміти: засобами мови програмування mpC та засобами бібліотеки функцій MPI створювати паралельні програми різного призначення, тобто розв'язання СЛАР і ЗДР, складати завдання для вирішення складних задач у GRID-системах.

У конспекті розглядаються основні питання проблематики паралельних обчислень, які є надзвичайно широкою галуззю теоретичних досліджень і практично виконуваних робіт і за звичаєм підрозділяються на такі напрямки діяльності:

- формування загальних принципів розробки паралельних алгоритмів для розв'язання складних задач, які мають велику трудомісткість обчислень;

– аналіз ефективності паралельних обчислень для оцінки *одержуваного прискорення* обчислень і *рівня використання* всіх можливостей комп’ютерного встаткування при паралельних способах розв’язання складних задач;
– розробка паралельних обчислювальних систем – огляд принципів побудови паралельних систем;
– створення й розвиток системного програмного забезпечення для паралельних обчислювальних систем;
– створення й розвиток паралельних алгоритмів для розв’язання прикладних задач у різних галузях.

СФЕРИ ЗАСТОСУВАННЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ І СУПЕРКОМП'ЮТЕРІВ

Зі зростанням продуктивності настільних ПК і робочих станцій, а також серверів може здатися, що потреба в застосуванні такої дуже дорогої обчислювальної техніки як супереом повинна постійно знижуватися. Однак це не так. З одного боку, цілий ряд застосувань може тепер успішно виконуватися на робочих станціях, але з іншого боку, час довів, що стійкою тенденцією є поява все нових застосувань, для яких необхідно використовувати супереом.

Сучасні засоби ІТ сприяють прискоренню фундаментальних досліджень і зростанню наукового й технологічного потенціалу.

Від самого їхнього народження й до сьогодення традиційною сферою застосування суперкомп'ютерів були та й залишаються наукові дослідження:

- фізика плазми й статистична механіка;
- фізика конденсованих середовищ;
- молекулярна й атомна фізика;
- теорія елементарних часток;
- газова динаміка й теорія турбулентності;
- астрофізика;
- квантова хімія (включаючи розрахунки електронної структури для цілей конструювання нових матеріалів, наприклад, каталізаторів і надпровідників);
- молекулярна динаміка;
- хімічна кінетика;
- теорія поверхневих явищ і хімія твердого тіла;
- конструювання ліків.

Але сьогодні вже існує безліч технічних проблем, вирішення яких неодмінно потребує використання суперкомп'ютерів:

- задачі екології;
- проблеми, пов'язані з вивченням і розвитком нанотехнологій;

- задачі аерокосмічної промисловості;
- розробки в автомобільній промисловості;
- підвищення ефективності й безпеки ядерної енергетики;
- прогнозування й розробка родовищ корисних копалин;
- задачі нафтовидобувної й газової промисловості (у тому числі проблеми ефективної експлуатації родовищ, особливо тривимірні задачі їхнього дослідження);
- конструювання нових мікропроцесорів і комп'ютерів, у першу чергу саме суперкомп'ютерів.

Величезна кількість галузей застосування суперкомп'ютерів перебуває на стиках різних наук. Так, наприклад, у метеорології – вивчення атмосферних явищ і, у першу чергу, задача довгострокового прогнозу погоди, для розв'язання якої постійно не вистачає потужностей сучасних супереом, тісно пов'язані з вирішенням низки названих вище проблем фізики. Роботи з хімії й біології перекриваються з технічними додатками. Кажучи про останні, доречно відзначити результати, отримані при моделюванні шумопоглинальних покриттів і конструкцій автомобілів, нових пристроїв напівпровідникової наноелектроніки, нових ліків, імуномодуляторів. Ресурси суперкомп'ютерів в числі іншого використовуються при обчисленні проектів, пов'язаних з розробкою нових матеріалів, пошуком лікарських препаратів такого покоління, а також при моделюванні змін клімату, поведінки елементарних часток, складних хімічних реакцій і т.ін. Все це було б неможливо без розробки й побудови нових обчислювальних потужностей, орієнтованих на застосування високопродуктивної багатоядерної обчислювальної техніки.

Застосування технологій високопродуктивних обчислень дозволяє «стискати» або «розтягувати» час будь-якого віртуального експерименту. Це актуально при дослідженні скороминучих або, навпаки, таких процесів, що протікають дуже повільно. Тепер можна моделювати параметри систем у їхніх прикордонних станах, будь то вкрай високі температури й тиск у гідро- і газодинаміці або граничні напруги в матеріалах при деформаціях.

За звичаєм на суперкомп'ютери дивляться головним чином як на лабораторний інструмент для вирішення складних наукових задач великого обсягу. Зараз коло їхніх застосувань розширюється. Безумовно, основними замовниками суперЕОМ ще продовжують вважати їхніх традиційних користувачів – науковців. Але поряд з вирішенням наукових задач сучасні суперкомп'ютери набувають все більшого застосування для вирішення нових, комерційних задач.

СуперЕОМ поступово проникають у раніш практично недоступну для них комерційну сферу. Насамперед мова йде про задачі, що припускають інтенсивну (у тому числі, і оперативну) обробку транзакцій для надвеликих БД. До цього класу задач можна віднести також системи підтримки прийняття рішень і створення інформаційних баз. Необхідно відзначити, що для роботи з подібними застосуваннями в першу чергу потрібна висока продуктивність введення/виведення й швидкодія при виконанні лише цілочисельних операцій, то застосування суперкомп'ютерів у цьому випадку можна вважати не зовсім виправданим. Але аналогічні вимоги виникають, зокрема, і з боку ряду додатків ядерної фізики, наприклад, при обробці результатів експериментів на прискорювачах елементарних часток. Але ж ядерна фізика – класична область застосування суперЕОМ від дня їхнього виникнення.

Аналізуючи потенційні потреби використання суперЕОМ для розв'язання існуючих сьогодні завдань, їх можна умовно розбити на два класи. До першого за звичаєм відносять застосування, для яких відомо, який рівень продуктивності треба досягти в кожному конкретному випадку, наприклад, довгостроковий прогноз погоди. До другого – відносять задачі, для яких характерне швидке зростання обчислювальних витрат зі збільшенням розміру досліджуваного об'єкта. Наприклад, у квантовій хімії неемпіричні розрахунки електронної структури молекул потребують обчислювальних ресурсів, витрати яких визначаються деяким параметром N, що умовно характеризує розмір молекули, і які пропорційні N у четвертому або навіть у п'ятому степені. Зараз багато молекулярних систем вимушено досліджуються в спрощеному модельному баченні. Маючи в перспективі вивчення й моделювання поведінки ще більш великих

молекулярних утворень (біологічні системи, кластери й т.ін.), квантова хімія дає приклад застосування, яке є «потенційно нескінченним» користувачем суперкомп'ютерних ресурсів.

У зв'язку зі сказаним є сенс більш докладно розглянути приклади застосування сучасних суперкомп'ютерів для розв'язання різноманітних обчислювальних задач – у наукових і космічних дослідженнях, конструкторських розробках, розробці нових ліків, метеорології…

Суперкомп'ютери традиційно застосовуються для військових цілей. Крім очевидних задач розробки зброї масового знищення й конструювання літаків і ракет, можна згадати приклад конструювання безшумних підводних човнів і ін. найбільш відомий приклад – це американська програма СОІ. У Міністерстві енергетики США суперкомп'ютер застосовується для моделювання ядерної зброї, що дозволяє взагалі відмінити ядерні випробування.

Суперкомп'ютери відіграють найважливішу роль при розв'язанні багатьох задач, які стоять перед агентством NASA, включаючи проектування нових космічних апаратів, вивчення глобального клімату й здійснення астрофізичних досліджень. Система, установлена в суперкомп'ютерному центрі NASA Advanced Supercomputing при Еймсовскому дослідницькому центрі в Моффет-Філд (шт. Каліфорнія), укомплектована 640 процесорними ядрами й має пікову продуктивність близько 5,6. Згідно із численними прогнозами, потреби NASA у комп'ютерах вищого класу надалі будуть зростати, тому необхідно вчасно переходити на більш досконалі технології. У масштабі всієї країни центр NASA підтримує вчених і інженерів, які працюють у таких галузях, як проектування космічних апаратів, удосконалення моделей для прогнозування кліматичних змін і ураганів, дослідження механізмів сонячної активності і т.ін. Багато проектів NASA потребують масштабних і складних обчислень, детального математичного моделювання. Ефективне вирішення цих задач можливо тільки за допомогою суперкомп'ютерів. Так дослідження, які проведені вченими NASA, дозволяють інженерам прискорити проектування й створення безпечніших і досконаліших космічних літальних апаратів, а комп'ютерне імітаційне моделю-

вання дає можливість одержувати адекватні прототипи для здійснення віртуальних тестів, а це істотно скорочує потребу в коштовних натурних випробуваннях.

У свою чергу Міжвідомчий суперкомп'ютерний центр Російської академії наук (МСЦ РАН) для здійснення наукових обчислень установив суперкомп'ютер з піковою продуктивністю 100 TFLOPS (трильйонів операцій із плаваючою крапкою в секунду). Систему спільно створювали фахівці МСЦ РАН, корпорацій HP і Intel. Завдяки новітнім рішенням HP і Intel, а також багаторічному досвіду МСЦ РАН в галузі високопродуктивних обчислень, у суперкомп'ютерному центрі встановлена найпотужніша в Росії обчислювальна система, яка за оцінками фахівців на момент її створення входила в п'ятірку найпотужніших суперкомп'ютерів Європи й, навіть, у число 50 найпродуктивніших систем у світі. Основна задача МСЦ РАН – забезпечення російських учених сучасними обчислювальними, інформаційними й телекомунікаційними ресурсами. МСЦ РАН в своєму розпорядженні має декілька високопродуктивних систем на базі різних платформ, серед яких, зокрема, кластери продуктивністю 7,7 TFLOPS з 320 двоядерними процесорами Intel Xeon 5160 і з 256 процесорами Intel Itanium 2 продуктивністю 1,64 TFLOPS. Комунікаційні канали зі швидкістю передачі даних до 10 Гіб/с зв'язують МСЦ РАН з російськими й закордонними науковими й освітніми інститутами. Ресурсами МСЦ РАН користуються 958 користувачів з 87 наукових і освітніх інститутів.

Крім суперкомп'ютерів, встановлених у МСЦ, корпорація IBM і провідний ВУЗ Росії – Московський державний університет ім. Ломоносова оприлюднили угоду про постачання суперкомп'ютера Blue Gene/P для факультету обчислювальної математики й кібернетики МДУ. Ця новітня обчислювальна система – перший суперкомп'ютер із всесвітньо відомої серії Blue Gene, установлений у Росії. Він застосовується для проведення фундаментальних досліджень в галузях нанотехнологій, моделювання нових матеріалів, біомедицини, моделювання діяльності мозку й інших. МДУ придбало дві апаратні стійки системи Blue Gene/P, які містять 8192 щільно впакованих мікропроцесори. Продуктив-

ність цієї конфігурації суперкомп’ютера 27,8 TFLOPS, що в 2600 разів перевищує продуктивність найбільш швидкодіючого суперкомп’ютера, що у цей час експлуатується в МДУ. Модульна і масштабована конструкція комп’ютера передбачає можливість додавання стійок у міру зростання потреб в обчислювальних потужностях. ОС суперкомп’ютера основана на ОС Linux з відкритим вихідним кодом. У цьому середовищі виконуються застосування, написані на розповсюджених мовах програмування, таких, як Fortran, C і C++, і які використовують комунікаційні протоколи основані на стандартах MPI (Message Passing Interface – інтерфейс передачі повідомлень). Комп’ютер підтримує різноманітні застосування, які зараз використовуються при проведенні досліджень у багатьох галузях науки, охоплюючи фізику, хімію, біологію, астрофізику, генетику, космологію, сейсмологію…

Сьогодні у світі як екологічно чиста альтернатива традиційним джерелам енергії активно розвивається гідроенергетика. Дослідницька компанія Turboinstitut, яка спеціалізується в галузі гідроенергетичних технологій, у недавно відкритому в Любляні (Словенія) Центрі високопродуктивних обчислень (Ljubljana Supercomputing Center) разом з IBM, установила новий потужний суперкомп’ютер на базі ОС Linux, названий Adria, який став найпотужнішим суперкомп’ютером у Південно-Східній Європі. Turboinstitut використовує його для прискореного моделювання складних процесів у роботі гідротурбін. Виконуючи обчислення в 50 разів швидше, ніж у існуючій системі, Adria скорочує тривалість експериментів з тижнів до декількох годин. Нова обчислювальна система сприяє прискоренню реалізації програми досліджень і розробок Turboinstitut в галузі гідроенергетики й впровадженню цих інноваційних технологій енергетичними компаніями в усьому світі як альтернативних існуючим джерелам енергії.

Цікаво відзначити використання високопродуктивного суперкомп’ютера командою «Формули-1» AT&T Williams. Комп’ютер установлений компанією Lenovo для проведення випробувань гоночного боліда в аеродинамічних трубах на базі цієї команди у Великобританії. Аеродинаміка відіграє вирішальну роль

у визначенні того, наскільки успішним буде виступ команди в кожному Гран-прі. Оптимальний баланс між притискною силою й опором варіюється від траси до траси. Так, аеродинаміка автомобіля в гонці в Монако, де багато крутих поворотів з невеликою кількістю прямих, істотно відрізняється від аеродинаміки, наприклад у Монці, де мало поворотів, але багато довгих прямих відрізків. Висока продуктивність суперкомп'ютера Lenovo дозволяє вивчити можливості настроювання автомобіля між трасами, що підвищує результативність команди. Суперкомп'ютер використовується для досліджень в галузі обчислювальної гідрогазодинаміки. Вивчається вплив на аеродинаміку таких факторів, як рельєф поверхні, поведінка коліс і рельєф траси. Команда інженерів може аналізувати вплив зміни кривизни поверхні боліда з метою збільшення притискної сили й зменшення опору середовища машині. Нова система виконує мільйони операцій, моделюючи повітряний потік на треку навколо 3D-моделі гоночного боліда. Подібний процес допомагає прогнозувати вплив незначних змін компонентної бази на опір і притискну силу боліда, що безпосередньо впливає на швидкість і керованість автомобіля. Моделювання аеродинамічних процесів виконується в комбінації з випробуваннями автомобілів у двох аеродинамічних трубах. Комп'ютерні показники аеродинамічних процесів дозволять команді AT&T Williams істотно скоротити час на дослідження й сконцентруватися на розробці кращих рішень для випробувань болідів у трубі й на треку. Суперкомп'ютер, використовуваний командою, має максимальну продуктивність до 8 TFLOPS і прискорює моделювання аеродинамічних процесів приблизно на 75%.

Дослідники з Університету Единбурга й наукового центра IBM на ім'я Т.Дж. Уотсона оголосили про запуск спільного проекту, мета якого – прискорити розробку ліків, що перешкоджають поширенню вірусу імунодефіциту людини (HIV, ВІЛ). Поряд із проведенням лабораторних експериментів проект передбачає застосування суперкомп'ютера для моделювання процесів на клітинному рівні. У проекті використовуються потужні обчислювальні технології, зокрема, суперкомп'ютер, у сполученні з новою експериментальною методи-

кою. Основні зусилля сконцентровані на дослідженні власно процесу інфікування шляхом розробки інгібіторів (сповільнювачів хімічних реакцій і біологічних процесів) для тієї частини вірусу імунодефіциту людини, яка відповідає за введення генетичного матеріалу ВІЛ у людську клітину. Новий аспект співробітництва вчених полягає в спробі розробити серію різних інгібіторів для їхнього застосування з одночасною можливістю запобігання мутації «хитрого» вірусу у відповідь на лікарську терапію з використанням одиночних інгібіторів. Перші отримані результати надзвичайно обнадіюють і підтверджують, що за допомогою моделювання процесів на комп'ютері можна дізнатися, які молекули інгібіторів здатні зупинити вірус ВІЛ і запобігти зараженню людей.

Університет штату Алабама в Бірмінгемі придбав у корпорації IBM суперкомп'ютер Blue Gene/L, у три рази збільшивши свої обчислювальні ресурси. Суперкомп'ютер із продуктивністю 5,6 TFLOPS істотно розширює можливості університету в галузі обчислювальної біології й молекулярного моделювання. Комп'ютер допомагає дослідникам здійснювати поглиблене імітаційне моделювання таких біологічних процесів, як протікання крові в артеріях і капілярах у районі новотворів. Ця система використовується при здійсненні медичних досліджень і в імітаційному моделюванні, а також для знаходження способів стримування й повної зупинки біологічної активності, яка зумовлює появу новотворів і інших захворювань у тканинах людини, які є загрозою для життя. Суперкомп'ютер підтвердив свої можливості як найбільш багатообіцяючий інструмент для здійснення імітаційного моделювання на рівні мікросекунд і менших відрізків часу.

Пекінське метеорологічне бюро (Beijing Meteorological Bureau, BMB) у своїй роботі використовує суперкомп'ютер для прогнозування погоди й контролю повітряного середовища. Система здатна охопити до 44 тис. кв. км території й надавати щогодинні кількісні прогнози погоди по кожному квадратному кілометру. Обчислювальна потужність комп'ютера в 10 разів перевершує аналогічний показник системи прогнозування погоди, використовуваної раніше пекінським метеоцентром. Поряд з погодинними прогнозами погоди супер-

комп'ютер також застосовується для прогнозування характеристик повітряного середовища. Головна задача нової обчислювальної системи ВМВ – підвищення точності метеорологічних прогнозів в областях, прилеглих до Пекіна. Суперкомп'ютер дозволив більш точно прогнозувати кількісні характеристики погоди в районі Пекіна під час Олімпійських ігор, і після їхнього завершення. Обчислювальна система пекінського метеоцентра входить у першу десятку найбільш швидкодіючих суперкомп'ютерів у Китаї.

Моделювання об'ємів природних нафтових резервуарів з метою визначення запасів нафти в нафтоносній області виконуються за результатами вимірів геофізичних характеристик верхнього шару земної поверхні приблизно в 100×100×100 точках. У кожній точці необхідно обчислити від 5 до 20 різних функцій (швидкість звуку, тиск, концентрацію, температуру і т.ін.). Для обчислення кожної функції в кожній точці виконується десь від 200 до 1000 операцій, для яких потрібно зробити від 100 до 1000 кроків у часі. У результаті, з огляду на середні значення вказаних діапазонів, можна оцінити обсяг необхідних обчислень:

$$10^6 \text{ (точок сітки)} \times 10 \text{ (функцій)} \times 500 \text{ (операцій)} \times 500 \text{ (кроків)} = 2{,}5 \cdot 10^{12},$$

або 2,5 трильйони операцій! Цілком очевидно, що виконати таку кількість обчислень за допомогою звичайних комп'ютерів дуже важко.

Як приклад фірми, що використовує суперкомп'ютер для розв'язання таких комерційних задач, можна назвати американську фірму Arco Oil Gas Co, яка застосовує орендований нею суперкомп'ютер Cray-1S для тривимірного моделювання басейнів залягання нафти. За словами наукового керівника розробки математичного забезпечення в дослідницькому центрі цієї фірми (Плано, шт. Техас) Джона Кіллоу, завдяки оптимізації процесу буравлення свердловин на нафтових родовищах півночі Аляски, фірма Arco очікує одержати додатковий дохід на загальну суму декількох мільярдів доларів. «Без ЕОМ Cray нам би не вистачило часу, щоб упоратися із цією задачею, – сказав Кіллоу. – За один тиждень ЕОМ Cray виконує такий же об'єм роботи, який установлені раніше в

нас машини спромоглися б зробити тільки за півроку».

Центральний економіко-математичний інститут РАН, з використанням, так званих, великомасштабних агент-орієнтованих моделей, здійснює роботи з моделювання складних систем, якими є соціально-економічні системи. Цікаво, що в роботі виконується порівняння розрахунків на звичайному персональному комп’ютері з досить високою продуктивністю й на суперкомп’ютері, який навіть не входить в число 500 кращих суперкомп’ютерів у світі. Так відзначається, що звичайний персональний комп’ютер цілком здатний здійснювати обчислення із задовільною швидкістю над сукупністю агентів числом до 20 тис. (поведінка кожного з них задається приблизно 20-ю функціями), і при цьому середній час розрахунку одиниці модельного часу (один рік) становить біля хвилини. При більшому числі агентів, наприклад 100 тис., комп’ютер попросту зависає (скоріш за все експериментаторам не стало сил дочекатися закінчення розрахунків). Використання 800 процесорів суперкомп’ютера й виконання оптимізованого коду дозволило збільшити число агентів до 5 млн. При цьому весь масив обчислень за 16 років був виконаний за період часу, приблизно 1 хв. 10 сек.

У спеціалізованій періодичній пресі регулярно з’являються повідомлення про різні застосування суперкомп’ютерів.

Так, наприклад, повідомляється про проведення аеродинамічного розрахунку компресора авіаційного двигуна. У машинному моделюванні використовувалася розрахункова сітка з кількістю вузлів більш ніж 50 мільйонів. Стаціонарні й нестаціонарні обчислення вироблялися на 130 процесорах, загальна кількість використаної пам’яті склала 495 ГіБ. З такою кількістю процесорів була досягнута продуктивність 3000 ітерацій у день, усього розрахунки проводились 6 місяців для нестаціонарного обчислення, яке містить 128 кроків у фізичному часі. Зрозуміло, що на однопроцесорному комп’ютері подібні розрахунки провести неможливо.

В іншій роботі викладені результати з чисельного моделювання кругового зриву потоку в компресорі. Модель компресора складалася з 188 елементарних об’ємів. У цьому випадку проводився нестаціонарний розрахунок, обчис-

лення тривали 40 діб на 1860 процесорах. Загальний об'єм роботи оцінювався в 1,8 мільйона годин процесорного часу.

За своїм масштабом дуже цікаві роботи з моделювання ракетного двигуна. У періодиці наводиться низка задач з галузі космонавтики й оцінюється їхня обчислювальна складність. Як правило, дані задачі мають відношення до галузі обчислювальної газової динаміки. Однієї із задач є задача розрахунку бафтингу основи ракети Ariane 5. Модель у цьому випадку містить 12 мільйонів вузлів і потребує більш ніж 350000 годин процесорного часу для моделювання 100 мілісекунд реального фізичного часу. Моделювання взаємодії факелів двигунів у космосі потребує 2 мільйони даних для 80 мільйонів часток в 120 субдоменах і потребує близько 35000 годин роботи процесора для виконання 130000 кроків розрахунку методом Монте-Карло.

Ще одна нова галузь виконання робіт, які зараз абсолютно немислимі без застосування суперкомп'ютерів – це розробка графічних застосувань для кіно й телебачення, де потрібно така ж сама висока продуктивність на операціях із плаваючою комою. Як приклад, можна оцінити необхідні обчислювальні ресурси для створення повноформатного повністю комп'ютерного мультиплікаційного фільму Шрек 3. Сучасний устояний стандарт кінематографії потребує наявності 24 кадрів у секунду. Для повнометражного фільму тривалістю приблизно 90 хвилин потрібно більше 120000 кадрів. Кожен із цих кадрів повинен бути оброблений незалежно один від одного. При цьому обробка одного кадру одним процесором у середньому потребує дві години на один варіант (дубль) кадру. Виконання нескладних розрахунків часу, який потрібен для створення фільму, дає значення більше 20 млн. процесорогодин і необхідний для цього обсяг даних перевищує 30 ТіБ (тебібайт – 2^{40}, або в старій термінології терабайт – $\sim 10^{12}$)! Тому творці фільму використовували суперкомп'ютер, який має більше 8000 процесорних ядер.

У цей час великі компанії, такі, як Google і Microsoft, використовують для підтримки працездатності своїх Web-сервісів комп'ютерні кластери. Такий підхід дозволяє сформувати досить потужну обчислювальну інфраструктуру на

основі відносно недорогих комп'ютерних вузлів і мережного встаткування. Однак у кластерів є й недоліки – у міру нарощування їхньої потужності зростають енергоспоживання й площі для розміщення кластерів. Як альтернативу кластерам IBM пропонує свої суперкомп'ютери лінійки Blue Gene. Ці обчислювальні комплекси відрізняються високою масштабованістю, завдяки чому теоретично зможуть справлятися з обслуговуванням Web-сервісів будь-якої складності. У цей час дослідники IBM експериментують з встановленням на суперкомп'ютери ОС Linux у комплекті з традиційним набором Web-застосувань (сервер Apache і СКБД MySQL).

Взагалі на сьогодні IBM, займаючи лідируючу позицію в галузі суперкомп'ютерів, впритул наблизилися до чергового бар'єра продуктивності, вимірюваного в петафлопсах (PFLOPS), що еквівалентно здатності виконувати 1000 трильйонів (1 квадрильйон) обчислень за секунду. Як очікується, петафлопс-системи будуть сприяти революційним проривам у науці й техніці завдяки можливостям високоточного прогнозування й імітаційного моделювання з високим ступенем деталізації. Наприклад, при моделюванні землетрусів вдається відслідковувати зсуви земної кори на територіях уздовж так званого розламу Сан-Андреас у Каліфорнії буквально від будівлі до будівлі, що дозволить поліпшити проектування сейсмостійких будівель і конструкцій.

Нема рації й далі продовжувати наведення прикладів використання паралельних обчислень на суперкомп'ютерах і розподілених системах у різних галузях природних і соціальних наук і в техніці, оскільки роль паралелізму і його вплив на розвиток архітектури комп'ютерів уже очевидна.

Питання і завдання для самоперевірки

1) Назвіть деякі традиційні сфери застосування суперкомп'ютерів і відповідного паралельного програмного забезпечення для них.
2) Які властивості характерні для технологій високопродуктивних паралельних обчислень при дослідженні скороминучих процесів або, навпаки, процесів, які змінюються дуже повільно.

3) Назвіть сучасні, раніш практично недоступні, сфери застосування паралельних технологій і суперкомп'ютерів.
4) На які два умовно існуючих класи підрозділяють потенційні потреби у використанні суперεом? Наведіть деякі приклади завдань, які вирішуються тим чи іншим класом.
5) Прокоментуйте приклади застосування паралельних технологій, описаних в цьому розділі конспекту і наведіть низку власних відомих Вам прикладів.

1 ПАРАЛЕЛЬНІ ОБЧИСЛЮВАЛЬНІ МЕТОДИ

Всі наведені раніше приклади, передбачають використання ***паралельних*** або ***розподілених*** обчислень на відповідному апаратному забезпеченні. Дуже часто ці два поняття вважаються синонімами того самого обчислювального процесу, але це не зовсім правильно.

Паралельні обчислення – спосіб комп'ютерних обчислень, при організації якого програми розробляються як набір взаємодіючих обчислювальних процесів, що працюють паралельно (одночасно). Термін охоплює сукупність питань паралелізму в програмуванні, а також створення ефективно діючих апаратних реалізацій.

А ***розподілені обчислення*** – це спосіб розв'язання трудомістких обчислювальних задач із використанням декількох комп'ютерів, найчастіше об'єднаних у паралельну обчислювальну систему, у тому числі й за допомогою звичайної локальної обчислювальної мережі. Розподілені обчислення застосовні також у розподілених системах керування.

Іншими словами, розподілені обчислення *обов'язково* передбачають виконання окремих частин загальної задачі на окремих процесорах або комп'ютерах, тим чи іншим чином з'єднаних між собою. Паралельні ж обчислення *можуть бути* виконані як на багатопроцесорних системах, так і на комп'ютерах з одиночним процесором, виконуючись у режимі розподілу часу для окремих потоків одного процесу.

Але, не дивлячись на відмінності, ці два види обчислень поєднує та обставина, що в обох випадках загальна *задача повинна бути розпаралелена* для виконання в окремих потоках на одному процесорі або на ансамблі паралельно працюючих процесорів.

У цей час існують два основних підходи до розпаралелювання обчислень. Це ***паралелізм даних*** і ***паралелізм задач***.

При здійсненні ***паралелізму даних*** *одна операція виконується відразу над всіма елементами цілого масиву даних*. При цьому різні фрагменти такого ма-

сиву обробляються на різних процесорах паралельної машини однаковою послідовністю операторів програми. У такій парадигмі програмування бажано забезпечити автоматичний розподіл даних між процесорами, тому ця задача покладається на програму. У цьому випадку за звичаєм один з процесорів виконує роботу з розподілу даних рівними фрагментами між іншими процесорами системи. ***Векторизація*** або ***розпаралелювання*** – розподіл того самого виконуваного коду по процесорах у цьому випадку найчастіше виконуються вже під час трансляції, тобто перекладу вихідного тексту програми в машинні команди. Роль програміста при створенні таких програм за звичаєм мінімальна й зводиться до завдання опцій паралельної оптимізації компілятора, директив паралельної компіляції, використання спеціалізованих мов для паралельних обчислень. Навіть при програмуванні складних обчислювальних алгоритмів можна використовувати бібліотеки підпрограм, спеціально розроблених і оптимізованих для конкретної архітектури комп'ютера. В останньому випадку програміст навіть може нічого й не знати про те, що програма виконується паралельно, прикладом таких бібліотек програм може служити добре відомий пакет математичного моделювання MATLAB. Мало хто знає, що мова реляційних баз даних SQL (Structured Query Language) є мовою з неявною паралельністю, яка може використовуватися відповідним компілятором у розподіленій системі баз даних, не потребуючи для цього спеціальних мовних конструктивів. Крім того, більшість транзакцій SQL-програм обробляється різними користувачами паралельно або з рознесенням часових інтервалів їхнього виконання. SQL-інструкції можуть використовуватись безпосередньо (інтерактивним чином) у базі даних або інтегруватися в одну з паралельних мов програмування у вигляді вбудованого блоку SQL.

Основними особливостями розглянутого підходу є те, що обробкою даних керує одна програма:

- простір імен є глобальним, тобто для програміста існує одна єдина пам'ять, деталі структури даних, доступу до пам'яті й межпроцесорного обміну даними від нього сховані;

- слабка синхронізація обчислень на паралельних процесорах, тобто виконання команд на різних процесорах відбувається, як правило, незалежно й тільки іноді виробляється узгодження виконання циклів або інших програмних конструкцій – їхня синхронізація. Кожен процесор виконує той же фрагмент програми, але немає гарантії, що в заданий момент часу на всіх процесорах виконується та сама машинна команда;
- паралельні операції над елементами масиву виконуються одночасно на всіх доступних даній програмі процесорах.

Підхід, оснований на паралелізмі даних, базується на використанні в програмах базового набору операцій:

- операції керування даними – у певних ситуаціях виникає необхідність у керуванні розподілом даних між процесорами, це може знадобитися, наприклад, для забезпечення рівномірного завантаження процесорів – чим більш рівномірно завантажені роботою процесори, тим ефективнішою буде робота комп'ютера;
- операції над масивами в цілому, а також над їхніми фрагментами – аргументами цих операцій є масиви в цілому або їхні фрагменти (перетини), при цьому одна операція застосовується одночасно до всіх елементів масиву або до його частини, наприклад, операціями такого типу є операції множення елементів масиву на скалярний або векторний множник, можливі здійснення й складніших дій – обчислення функції від масиву;
- умовні операції – вони застосовуються лише до тих елементів масиву, які задовольняють певну умову, наприклад, у сіткових методах це можуть бути елементи, що відповідають парним/непарним номерам рядків або стовпців сітки;
- операції приведення – застосовуються до всіх елементів масиву (або до його частки), а результатом є одне-єдине значення, прикладами операції приведення є операції обчислення суми елементів масиву або максимального значення його елементів;

– операції зсуву – операції зсуву масивів потрібні для ефективної реалізації деяких паралельних алгоритмів, наприклад, зсуви часто використовуються в алгоритмах обробки зображень, у кінцево-різницевих алгоритмах і т.ін.;
– операції сканування – такі операції також називаються префіксними/суфіксними операціями, наприклад, префіксна операція підсумовування виконується в такий спосіб – елементи масиву підсумовуються послідовно, а результат чергового підсумовування заноситься в чергову комірку нового, результуючого масиву, причому номер цієї комірки збігається із числом підсумованих елементів вихідного масиву;
– операції, пов'язані з пересиланням даних – операції можуть здійснюватися, наприклад, між масивами різної форми, які мають різну розмірність, довжину по кожному виміру та ін.

Реалізація моделі паралелізму даних потребує підтримки паралелізму на рівні транслятора. Таку підтримку можуть забезпечувати:

– препроцесори, що використовують існуючі послідовні транслятори й спеціалізовані бібліотеки, з реалізаціями паралельних алгоритмічних конструкцій;
– передтранслятори, які виконують попередній аналіз логічної структури програми, перевірку залежностей і обмежену паралельну оптимізацію;
– транслятори з розпаралелюванням, які виявляють паралелізм у вихідному коді програми й виконують його перетворення в паралельні конструкції. Для спрощення перетворення у вихідний текст програми можуть додаватися спеціальні директиви трансляції.

Навпаки, стиль програмування, оснований на паралелізмі задач, передбачає, що *обчислювальна задача розбивається на декілька незалежних одна від одної підзадач, і кожен процесор завантажується своєю власною підзадачею*. При цьому одночасно виконується множина різних операцій над множиною, загалом кажучи, різних і різнотипних даних. Для кожної підзадачі пишеться своя власна програма звичайною мовою програмування. Важливо те, що всі ці

програми обмінюються результатами своєї роботи й такий обмін здійснюється викликом процедур спеціалізованої бібліотеки. Програміст контролює розподіл даних між процесорами і обмін даними. У порівнянні з підходом, основаним на паралелізмі даних, такий підхід більш трудомісткий, з ним зв'язана низка таких проблем, як: підвищена трудомісткість налагодження програми, необхідність забезпечення рівномірного завантаження процесорів, мінімізація обміну даними між задачами, можливість виникнення тупикових ситуацій, коли відправлена одною програмою посилка з даними не доходить до місця призначення.

Привабливими особливостями цього підходу є більша гнучкість і більша свобода, яка надається програмістові в розробці програми, яка ефективно використовує ресурси паралельного комп'ютера й, як наслідок, є можливість досягнення максимальної швидкодії.

В обох підходах, при складанні паралельних алгоритмів роботи, необхідно вирішити дві основні проблеми:

- забезпечення рівномірного завантаження процесорів;
- забезпечення деякої мінімально необхідної швидкості обміну інформацією.

Перша проблема пов'язана з тим, що чим більш нерівномірно виконано розподіл задачі по процесорах/потоках, тим далі ми відходимо від паралельних обчислень. Так, якщо з десяти процесорів завантажений тільки один, то утворюється ефект звичайних послідовних обчислень. Таким чином, передбачається, що всі процесори повинні виконувати приблизно однакову кількість обчислень над однаковими обсягами даних. Неоднорідність процесорів в обчислювальній системі додає труднощів при розпаралелюванні задачі, тому що процесори працюють із різною швидкістю й швидкісні процесори будуть простоювати, очікуючи поки найповільніший з процесорів виконає свою частку обчислень.

Друга не менш важлива проблема, полягає в тому, що при низькій швидкості обміну між процесорами основна частина часу буде витрачатися на очікування процесорами необхідної інформації. Звідси витікає той висновок, що в розподілених системах дуже велике значення має пропускна здатність мережі

міжз'єднань і швидкодія окремих процесорів у таких системах зумовлюється швидкістю передачі даних.

Паралельна обробка даних, втілюючи ідею одночасного виконання декількох дій, має два різновиди: ***конвеєрність*** і власне ***паралельність***. Обидва види паралельної обробки, загалом кажучи, не потребують яких-небудь особливих пояснень, але коротко про них можна сказати таке.

Паралельна обробка. Якщо якийсь пристрій виконує одну операцію за одиницю часу, то тисячу операцій він виконає за тисячу одиниць. Якщо припустити, що є п'ять саме таких незалежних пристроїв, здатних працювати одночасно й незалежно, то ту ж тисячу операцій система з п'яти пристроїв може виконати вже не за тисячу, а за двісті одиниць часу. У загальному випадку, якщо однопроцесорний пристрій виконує деяку роботу за час $t_{оп}$, то система з n пристроїв, найімовірніше, ту ж роботу виконає приблизно за $t_{мп} = t_{оп}/n$ одиниць часу, тобто буде отримане прискорення роботи в n раз. Пізніше буде показано, що це співвідношення найчастіше не виконується, але в цілому ряді випадків цілком є справедливим для систем паралельної обробки даних.

Конвеєрна обробка. Для додавання двох чисел дійсного типу, представлених у формі із плаваючою комою, необхідно виконати велику кількість дрібних операцій, таких як порівняння порядків, вирівнювання порядків, додавання мантис, нормалізація й т.ін. (пізніше це питання буде розглянуте більш докладно). Процесори перших комп'ютерів виконували всі ці «мікрооперації» для кожної пари аргументів послідовно одна за одною доти, поки не доходили до остаточного результату, і лише після цього переходили до обробки такої пари доданків.

Ідея конвеєрної обробки полягає у виділенні окремих етапів виконання загальної операції, причому кожен етап, виконавши свою частку роботи, передає результат такому й одночасно приймає нову порцію вхідних даних. Таким чином одержується очевидний виграш у швидкості обробки за рахунок суміщення раніше рознесених у часі операцій. Припустимо, що в операції можна виділити п'ять мікрооперацій, кожна з яких виконується за одиницю часу. Якщо є один неподільний послідовний пристрій, то 100 пар аргументів він обро-

бить за 500 одиниць часу. Якщо ж кожну мікрооперацію виділити в окремий етап (або інакше кажучи – ступінь) конвеєрного пристрою, то на п’ятій одиниці часу на різній стадії обробки такого пристрою будуть перебувати перші п’ять пар аргументів. Перший результат, таким чином, визначиться через 5 одиниць часу, кожен такий – через одну одиницю після попереднього. Весь же набір зі ста пар буде оброблений за 5 + 99 = 104 одиниці часу, тобто буде отримане прискорення в порівнянні з послідовним пристроєм майже в п’ять разів (за кількістю ступенів конвеєра). Те ж саме можна сказати і у загальному випадку.

Якщо деяка операція містить n незалежних ступенів, які виконуються послідовно один за одним, кожен ступінь спрацьовує за одиницю часу й потрібно виконати m (m набагато більше за n) таких складених операцій, то час роботи однопроцесорного пристрою складе:

$$t_{оп} = m \times n.$$

Конвеєрний же пристрій для виконання тієї ж кількості операцій витратить час, який дорівнює:

$$t_{кп} = m + n - 1,$$

а прискорення роботи вже буде дорівнюватись:

$$\frac{t_{оп}}{t_{кп}} = \frac{m \times n}{m + n - 1} \approx n,$$

тобто у результаті маємо прискорення майже в n раз за рахунок використання конвеєрної обробки даних.

Здавалося б, конвеєрну обробку можна з успіхом замінити звичайним паралелізмом, для чого достатньо просто скопіювати основний пристрій стільки разів, скільки ступенів конвеєра передбачається виділити. Однак вартість і складність отриманої системи буде непорівнянна з вартістю й складністю конвеєрного варіанта, а продуктивність буде майже такою.

Зі сказаного вище можна зробити висновок, що розпаралелювання програм є доволі складним процесом. Тому перш ніж приступити безпосередньо до складання й опису деяких паралельних обчислювальних алгоритмів, необхідно розглянути загальні принципи розв'язання задачі розпаралелювання.

1.1 Організація паралельних обчислень з використанням наявних технологій (PVM, MPI)

Розробка алгоритмів (а особливо методів паралельних обчислень) для розв'язання складних науково-технічних задач часто є доволі значною проблемою. Розгляд математичних аспектів розробки й доказ збіжності алгоритмів не є метою цього конспекту лекцій. Ці питання повинні вивчатися в низці «класичних» математичних навчальних курсів, наприклад у курсі чисельних методів в інформатиці. Тому надалі передбачається, що обчислювальні схеми розв'язання задач, розглянуті у якості прикладів програм, уже відомі. З урахуванням висловленого, дії для визначення ефективних способів організації паралельних обчислень можуть полягати в такому (рис. 1.1):

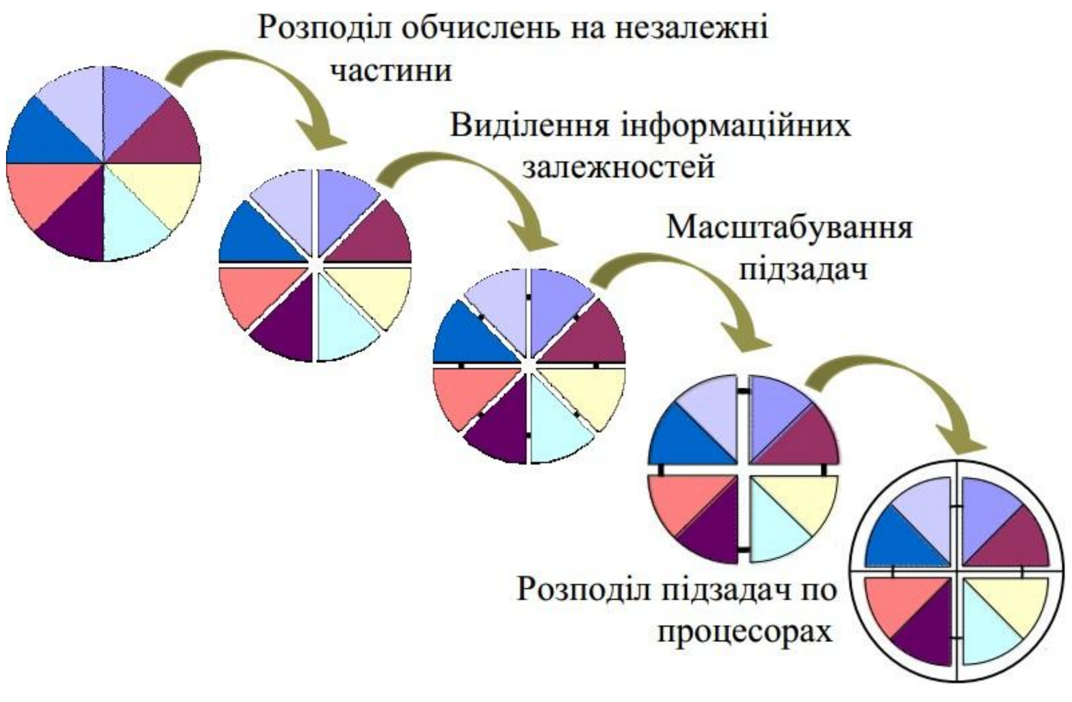


Рисунок 1.1 − Загальна схема розробки паралельних алгоритмів

– виконати аналіз наявних обчислювальних схем і здійснити їхній поділ (декомпозицію) на частині (підзадачі), які можна реалізувати значною мірою незалежно одна від одної;
– виділити для сформованого набору підзадач інформаційні взаємодії, які повинні здійснюватися в ході розв'язання вихідної поставленої задачі;
– визначити необхідну (або доступну) для розв'язання задачі обчислювальну систему й виконати розподіл отриманого набору підзадач між процесорами системи.

При найбільш загальному розгляді є цілком очевидним, що обсяг обчислень для кожного використаного в системі процесора повинен бути приблизно однаковим – це дозволить забезпечити рівномірне обчислювальне завантаження (балансування) процесорів. Також зрозуміло те, що розподіл підзадач між процесорами повинен виконуватись таким чином, щоб кількість інформаційних зв'язків (комунікаційних взаємодій) між підзадачами була мінімальною.

Після виконання всіх перерахованих етапів проектування можна оцінити ефективність розроблювальних паралельних методів: для цього за звичаєм обумовлюються значення показників якості породжуваних паралельних обчислень (прискорення, ефективність, масштабованість). За результатами проведеного аналізу може виявитися необхідним повторення окремих (у крайньому випадку всіх) етапів розробки. Слід зауважити, що повернення до попередніх кроків розробки може відбуватися на будь-якій стадії проектування паралельних обчислювальних схем.

Тому часто додатковою дією, яка виконується в наведеній вище схемі проектування, є коректування складу сформованої множини підзадач після визначення наявної кількості процесорів – підзадачі можуть бути укрупнені (агреговані) при наявності малого числа процесорів або навпаки деталізовані у протилежному випадку. В цілому, ці дії можуть визначатись як ***масштабування розроблювального алгоритму*** й виділятись як окремий етап проектування паралельних обчислень.

Щоб використати одержуваний в остаточному підсумку паралельний ме-

тод, необхідно виконати розробку програм для розв'язання сформованого набору підзадач і розмістити розроблені програми по процесорах відповідно до обраної схеми розподілу підзадач. Для проведення обчислень програми запускаються на виконання (програми на стадії виконання зазвичай йменуються процесами). Для реалізації інформаційних взаємодій програми повинні мати у своєму розпорядженні засоби обміну даними – ***канали передачі повідомлень***.

Слід зазначити, що найчастіше кожен процесор виділяється для вирішення єдиної підзадачі, однак, при наявності великої кількості підзадач або використанні обмеженого числа процесорів це правило може не дотримуватися й, у наслідку, на процесорах може виконуватись одночасно кілька програм (процесів). Зокрема, при розробці й початковій перевірці паралельної програми для виконання всіх процесів може використовуватися лише один процесор (при розташуванні на одному процесорі процеси виконуються в режимі поділу часу).

Уважно розглядаючи розроблену схему проектування й реалізації паралельних обчислень, можна відзначити, що запропонований підхід у значній мірі орієнтований на обчислювальні системи з розподіленою пам'яттю, коли необхідні інформаційні взаємодії реалізуються за допомогою передачі повідомлень по каналах зв'язку між процесорами. Проте, ця схема може застосовуватись без втрати ефективності паралельних обчислень і для розробки паралельних методів для систем з поділюваною пам'яттю – у цьому випадку механізми передачі повідомлень для забезпечення інформаційних взаємодій повинні замінюватись операціями доступу до спільних (поділюваних) змінних.

Станом на сьогодні розроблена безліч мов і систем програмування, як для паралельних, так і для розподілених обчислювальних систем. Прикладами таких систем є OpenMP, [p]threads, shmem (для систем з поділюваною пам'яттю), Occam, Concurrent C, HPF (високопродуктивний ФОРТРАН), HPC++, Distributed Java (для систем з розподіленою пам'яттю), Parlog, Concurrent Prolog (паралельні версії мови ПРОЛОГ), Multilisp (паралельна версія мови ЛІСП).

Дуже часто для паралельного програмування використовуються дві такі системи бібліотечних функцій: Message Passing Interface (MPI) – ***Інтерфейс***

Передачі Повідомлень і Parallel Virtual Machine (PVM) – ***Паралельна Віртуальна Машина***. Ці системи програмування призначені для роботи на розподілених системах, але вони можуть використовуватися й для систем з поділюваною пам’яттю.

Обидві системи реалізують модель передачі повідомлень і містять бібліотеки функцій і підпрограм для стандартних мов програмування C, C++, Fortran забезпечують взаємодії типу «точка-точка» і групові. У той же час системи мають і істотні відмінності.

Центральним елементом системи розробки програм у проекті PVM є «віртуальна машина» – набір різнорідних обчислювальних вузлів, який з точки зору користувача є одним великим паралельним комп’ютером. Одним з нюансів реалізації цієї паралельної машини є збереження простоти способу обміну даними між задачами й універсальність інтерфейсу у подальшому розвитку проекту.

На відміну від PVM, який був зароджений й по сьогодні розвивається в рамках дослідницького середовища, API MPI-1 був створений провідними експертами в індустрії комп’ютерів і програмного забезпечення в 1993-1994 роках.

Необхідність розробки MPI обумовлена потребою стандартизувати й уніфікувати API передачі повідомлень тому, що раніш кожен виробник багатопроцесорних обчислювальних комплексів пропонував свій API для обміну повідомленнями. Таким чином, MPI був прийнятий як стандарт API для обміну повідомленнями.

І PVM, і MPI надають можливість логічного з’єднання машин у єдину обчислювальну систему. І PVM, і MPI від самого народження містять велику кількість бібліотек для розробки паралельних програм мовами Fortran і C. На сьогодні ж розроблені бібліотеки класів для мов Java (кілька різних бібліотек від різних розроблювачів), Python, C# й для багатьох інших. Обидві системи реалізують ***модель передачі повідомлень***.

Якщо застосування, яке розробляється для багатопроцесорного обчислювального комплексу, буде запускатися лише на одній архітектурі й не поребує

значної переносимості, то використання MPI може принести значний виграш у швидкодії. Система містить значно багатіші засоби комунікації, тому її застосування є кращим в обчислювальних системах, де використовуються спеціальні режими комунікації, недоступні в PVM. Але з метою забезпечення високої швидкості взаємодії на MPI були накладені обмеження. Найбільш значимими з них є відсутність сумісності – різні реалізації MPI не можуть взаємодіяти між собою і відсутність можливості писати стійкі до відмови застосування за допомогою MPI. Правда стандарт MPI-2 і всі пізніші дозволяють створювати стійкі до відмови застосування, але, на жаль, поки не всі розроблювачі підтримують цю властивість другого стандарту MPI.

Через те, що PVM реалізує поняття «віртуальної машини» , ця система має переваги при виконанні обчислень у системі, яка складається з неоднорідних вузлів. PVM містить системи керування ресурсами й процесами, які важливі для написання застосувань, що запускаються як на кластерах робочих станцій, так і на багатопроцесорних обчислювальних комплексах. Чим більше вузлів у кластері, тим більш важливим стає стійкість PVM до відмови. Можливість написання програм, які успішно працюють навіть під час відмови обчислювальних вузлів, дуже важлива для розподілених обчислень у неоднорідному середовищі.

Вибір найбільш зручного API (MPI або PVM) залежить від структури обчислювальної системи й задач, які потрібно вирішити.

Паралельна Віртуальна Машина – PVM просто є пакетом програм, який дозволяє використовувати зв’язаний у локальну мережу набір різнорідних комп’ютерів, працюючих під керуванням Unix-сумісної операційної системи (існують системи й для Windows, але вони мають дуже обмежене застосування), як один великий паралельний комп’ютер. Таким чином, проблема великих обчислень досить ефективно вирішується за рахунок використання сукупної потужності й пам’яті великої кількості комп’ютерів. Пакет програм PVM легко переноситься на будь-яку Unix-платформу.

Проект PVM був створений в 1989 році Вайді Сандерманом (Vaidy

Sunderman) – професором університету Еморі м. Атланта і лабораторією Oak Ridge National Laboratory. Перший реліз був запущений у березні 1991. Комплекс PVM забезпечує роботу паралельних програм у гетерогенних обчислювальних середовищах. Машина PVM поєднує безліч різнорідних обчислювальних вузлів в один величезний ресурс. У системі можливе створення застосувань як типу ***SPMD*** (одна паралельна програма, багато потоків даних), так і типу ***MPMD*** (багато паралельних програм, багато потоків даних). Вона сумісна із застосуваннями, написаними на різних мовах програмування (наприклад, C, C++, Fortran) і коректно перетворює дані при передачі їх між вузлами різної архітектури. PVM від створення є відкритою й вільно розповсюджуваною системою.

Паралельну віртуальну машину можна визначити як частину засобів реального обчислювального комплексу (процесори, пам'ять, периферійні пристрої й т.ін.), призначену для виконання множини задач, які беруть участь в одержанні загального результату обчислень. У загальному випадку число задач може перевершувати число процесорів, включених в PVM. Крім того, до складу PVM можна включати досить різнорідні обчислювальні машини, несумісні по системах команд і форматах даних. Інакше кажучи, Паралельною Віртуальною Машиною може стати як окремо взятий ПК, так і локальна мережа, що включає в себе суперкомп'ютери з паралельною архітектурою, універсальні ЕОМ, графічні робочі станції й ті ж самі малопотужні ПК. Важливо лише, щоб про обчислювальні засоби, які включаються в PVM, була інформація у використовуваному програмному забезпеченні PVM. Завдяки цьому програмному забезпеченню користувач може вважати, що він спілкується з одною обчислювальною машиною, у якій можливо паралельне виконання множини задач.

PVM дозволяє користувачам використовувати існуючі апаратні засоби, для розв'язання досить складних задач при мінімальних витратах. Сотні дослідницьких груп в усьому світі використовують PVM, щоб вирішити важливі наукові, технічні, і медичні проблеми, а так само використовують PVM як освітній інструмент, для викладання паралельного програмування.

Задачі PVM можуть містити структури для забезпечення необхідних рів-

нів контролю й залежності. Інакше кажучи, у будь-якій «точці» виконання взаємозалежних застосувань будь-яка існуюча задача може запускати або зупиняти інші задачі, додавати або видаляти комп'ютери з віртуальної машини. Кожен процес може взаємодіяти та/або синхронізуватися з будь-яким іншим. Кожна специфічна структура для контролю і залежності може бути реалізована в системі PVM адекватним використанням конструкцій PVM і керуючих конструкцій головної (хост-) мови системи.

Маючи таку всеосяжну природу (специфічно для концепції віртуальної машини), а також через свій простий, але функціонально повний програмний інтерфейс, система PVM набула широкого розповсюдження, у тому числі й у науковому співтоваристві, пов'язаному з високошвидкісними обчисленнями.

Система PVM складається із двох частин. Перша частина – це демон, що поміщається на всі комп'ютери, які входять до складу віртуальної машини. (Прикладом програми-демона може бути поштова програма, що виконується у фоновому режимі й обробляє всю вхідну й вихідну електронну пошту комп'ютера). Демон розроблюється таким чином, щоб будь-який користувач із достовірним логином міг інсталювати його на машину. Коли користувач бажає запустити розподілене застосування PVM, він, насамперед, створює віртуальну машину, запускаючи PVM. Після цього застосування PVM можна запустити з будь-якого Unix-термінала на кожному з хостів. Декілька користувачів можуть конфігурувати віртуальні машини, які перекриваються, кожен користувач може послідовно запустити кілька застосувань PVM.

Друга частина системи – це бібліотека підпрограм інтерфейсу PVM. Вона містить функціонально повний набір примітивів, необхідних для взаємодії між задачами застосування. Ця бібліотека містить викликувані користувачем підпрограми для обміну повідомленнями, породження процесів, координування задач і модифікації віртуальної машини.

Функціонування PVM ґрунтується на механізмах обміну інформацією між задачами, виконуваними в її середовищі. Щодо цього, найбільше зручно реалізовувати PVM у рамках багатопроцесорного обчислювального комплексу,

виділивши віртуальній машині кілька процесорів і індивідуальні або загальне ОЗП (залежно від умов). Використання PVM припустиме як на багатопроцесорних комп'ютерах (SMP) так і на обчислювальних комплексах, побудованих за кластерною технологією. При використанні PVM, як правило, значно спрощуються проблеми швидкого інформаційного обміну між задачами, а також проблеми узгодження форматів представлення даних між задачами, виконуваними на різних процесорах

Ефективне програмування для PVM починається з того, що алгоритм обчислень доцільно адаптувати до складу PVM і до її характеристик. Це досить творча задача, яка у багатьох випадках повинна вирішуватися програмістом. Крім задачі розпаралелювання обчислень виникає необхідність вирішення задачі керування обчислювальним процесом, координації дій задач-учасників цього процесу. Іноді для керування доводиться створювати спеціальну задачу, яка сама не бере участь в обчисленнях, але забезпечує погоджену роботу інших задач-обчислювачів.

В PVM існує два варіанти організації обчислень.

При першому варіанті всі задачі запускаються одною командою, у якій вказується ім'я виконуваного файла, кількість задач, що запускаються, а також кількість й тип використовуваних процесорів. За цією командою на зазначених процесорах запускається необхідна кількість копій зазначеного виконуваного файла. Отже, програмні коди всіх запущених в PVM задач у цьому випадку однакові. Для того щоб ці задачі могли виконувати різні дії, їм повинен бути відомий деякий атрибут, який відрізняє кожну задачу від інших. Тоді, використовуючи цей атрибут в умовних операторах, легко запрограмувати виконання задачами різних дій. Наявність такої ознаки передбачено в будь-який багатозадачній операційній системі. Це так званий ***ідентифікатор задачі*** (TID) – ціле число, яке привласнюється задачі при її запуску. Слід зазначити, що при запуску задача одержує ідентифікатор, відмінний від ідентифікаторів, виконуваних у цей час задач. Це гарантує, що ідентифікатори всіх запущених задач в PVM будуть різними. Якщо тепер забезпечити задачі можливістю визначати власний

ідентифікатор і обмінюватися інформацією з іншими задачами, то зрозуміло, що їм легко розподілити між собою обчислювальну роботу, у залежності, наприклад, від займаного місця в упорядкованому наборі ідентифікаторів задач.

Другий варіант на практиці використовується набагато частіше. У ньому спочатку запускається одна задача (master), що у множині задач буде відігравати функції координатора робіт. Ця задача виконує деякі підготовчі дії, після чого запускає інші задачі (slaves), яким може відповідати той самий виконуваний файл або різні виконувані файли. Такому варіанту організації паралельних обчислень віддають перевагу при ускладненні логіки керування обчислювальним процесом, а також в тому випадку, коли алгоритми, реалізовані в різних задачах, істотно різняться або є великий обсяг операцій, які обслуговують обчислювальний процес у цілому (наприклад, операції введення/виведення).

Як вже відзначалось, у системі PVM кожна задача (якій відповідає виконуваний файл), будучи запущеною на деякому процесорі, ідентифікується цілим числом – ідентифікатором задачі (TID). За змістом TID схожий на ідентифікатор процесу в операційній системі Unix. Конкретні значення TID несуттєві, важливо лише, щоб всі задачі, запущені в PVM, мали різні TID. Окремо слід зазначити, що навіть копії того самого виконуваного файла, будучи запущені паралельно на N процесорах PVM, створюють N задач із різними TID.

Модель передачі повідомлень набула практичного втілення й подальшого розвитку в специфікації, яка дістала назву ***Інтерфейс Передачі Повідомлень*** – MPI. Ця специфікація була розроблена в 1993-1994 роках групою MPI Forum, до складу якої входили представники академічних і промислових кіл. Вона стала *першим стандартом* систем передачі повідомлень. В MPI були враховані досягнення інших проектів зі створення систем передачі повідомлень: NX/2, Express, nCUBE, Vertex, p4, PARMACS, PVM, Chameleon, Zipcode, Chimp і т.ін. Різні реалізації MPI розробляються як бібліотеки підпрограм, які можуть використовуватися в програмах мовами C/C++ і Fortran. На момент написання конспекту прийнята і вже діє версія специфікації MPI-3.

У моделі програмування, яка підтримується MPI, програма породжує кі-

лька процесів, взаємодіючих між собою за допомогою звертань до підпрограм передачі й прийому повідомлень. За звичаєм, при ініціалізації MPI-програми створюється фіксований набір процесів, причому кожен процес виконується на своєму процесорі. У цих процесах можуть виконуватись різні програми, тому модель програмування MPI іноді називають ***MPMD***-моделлю (Multiple Program Multiple Data – багато програм, багато даних), на відміну від ***SPMD***-моделі, коли на кожному процесорі виконуються тільки однакові задачі.

Для організації локальних і неструктурованих комунікацій використовуються, так звані, двоточкові обміни даними. При виконанні глобальних операцій застосовуються колективні обміни. Асинхронні комунікації реалізуються за допомогою запитів про одержання повідомлень. Механізм, який має назву ***комунікатор***, приховує від програміста внутрішні комунікаційні структури.

Алгоритми, у яких є фіксована кількість підзадач, допускають пряму реалізацію за допомогою двоточкових і групових обмінів. Алгоритми, у яких кількість процесів змінюється в процесі виконання програми, не можуть бути реалізовані в MPI безпосередньо (але починаючи з специфікації MPI-2 така можливість вже реалізована). У цьому випадку доводиться відразу, у момент запуску застосування, створювати множину процесів зі структурою, у яку вписуються всі можливі конфігурації підзадач, що виникають у процесі виконання програми. Динаміка буде в цьому випадку підтримуватися змінюваною структурою комунікацій. Часто це є можливим.

Специфікація MPI забезпечує переносимість програм на рівні вихідних кодів і велику функціональність. Підтримується робота на гетерогенних кластерах і симетричних багатопроцесорних системах: Не підтримується, як вже відзначалось, запуск процесів під час виконання MPI-програми. У специфікації відсутні описи паралельного введення/виведення й налагодження паралельних програм (починаючи з стандарту MPI-2 уже передбачені й такі операції). Інколи такі можливості включаються до складу конкретної реалізації MPI у вигляді додаткових пакетів і утиліт. Стандартом не гарантується сумісність різних реалізацій MPI.

Важливою властивістю паралельної програми, так само як і послідовної, є її ***детермінізм*** – це означає, що програма повинна завжди давати той самий результат для того самого набору вхідних даних. Модель паралельного програмування з передачею повідомлень, загалом кажучи, цієї властивості не має, оскільки порядок одержання повідомлень від двох процесів третім не визначений. Якщо ж один процес послідовно посилає кілька повідомлень іншому процесу, MPI гарантує, що другий процес одержить їх саме в тім порядку, у якому вони були відправлені. Відповідальність за забезпечення детермінованого виконання програми повністю лягає на програміста.

Однією з найпоширеніших реалізацій специфікації MPI є бібліотека процедур MPICH (MPI CHameleon), яка підтримує роботу на великій кількості платформ і з різними комунікаційними інтерфейсами, у тому числі TCP/IP, і є вільно розповсюджуваним програмним забезпеченням.

Різні версії MPICH (наприклад, така версія як MPICH 1.2.2) характеризуються такими основними особливостями:

- повна сумісність зі специфікацією MPI-1;
- наявність інтерфейсу в стилі MPI-2 з функціями для мови C++ зі специфікації MPI-1;
- наявність інтерфейсу із процедурами мови Fortran-77/90;
- існує реалізація для Windows NT, яка розповсюджується у вихідних текстах. Її встановлення й використання відрізняються від відповідних дій у системах стандарту POSIX;
- підтримка великої кількості архітектур, у тому числі кластерів робочих станцій, симетричних багатопроцесорних систем й т.ін.;
- часткова підтримка специфікації MPI-2;
- часткова підтримка паралельних введення/виведення (ROMIO);
- наявність засобів трасування й протоколювання (на основі масштабованого формату log-файлів SLOG);
- наявність засобів візуалізації продуктивності паралельних програм (upshot і jumpshot);

− наявність у складі MPICH тестів продуктивності й перевірки функціонування системи.

До числа недоліків MPICH можна віднести неможливість запуску процесів під час виконання програми й відсутність засобів моніторингу за поточним станом обчислювальної системи. У результаті цього, якщо відбувається, наприклад, апаратний збій на одному із процесорів, які беруть участь у виконанні паралельної MPI-програми, її робота завершується аварійно. Неможливість динамічної зміни набору процесів не дозволяє використовувати ресурси обчислювальної системи з максимальною ефективністю.

Якщо при установленні бібліотека MPICH була сконфігурована для роботи на кластері, то навіть при пересиланнях повідомлень на одному комп’ютері буде використовуватися мережний протокол TCP/IP. Це не найефективніше рішення, але воно працює. Якщо ж зборка виконувалась для системи з поділюваною пам’яттю, MPICH-програма не зможе працювати на кластері.

До складу MPICH входять бібліотечні й заголовні файли. MPICH містить більше сотні підпрограм. З пакетом MPICH поставляються засоби візуального налагодження й профілювання паралельних програм. Це jumpshot або її більш стара версія upshot. Ці засоби написані мовою Java і працюють із файлами-протоколами подій (tracefiles) CLOG (jumpshot 2) і SLOG (jumpshot 3). Розмір файла-протоколу в третій версії jumpshot може бути дуже великим, до гібібайта. Існує документація до MPICH у форматах HTML і PostScript.

До складу MPICH включені також приклади програм, які (наприклад, для системи Linux) розташовуються в каталогах:

− mpich/examples/basic/ – демонстрація основних можливостей MPICH;
− mpich/examples/test/ – тестові програми;
− mpich/examples/perftest/ – тестові програми для визначення продуктивності.

Набір прикладів може бути різним у різних версіях MPICH.

Є й комерційні варіанти MPI, які випускаються, як правило, з оптимізацією для конкретної платформи й можуть містити додаткові функції й утиліти. Є

варіанти для інших операційних систем. Як приклад можна навести WMPI – реалізацію MPI для Microsoft Windows.

У лабораторних роботах курсу використовується реалізація бібліотеки MPI для мови Java – MPJ Express. Докладний опис, у межах стандарту MPI-1, методів бібліотеки класів цього пакета є предметом окремого навчального посібника[1]. У пакет MPJ Express також входить документація у файлах форматів HTML і PDF.

1.2 Паралельні перетворення арифметичних виразів

У програмуванні арифметичні вирази є окремим випадком операторів програми. Тому є сенс, спочатку розглянути технологію розпаралелювання програми в цілому і окремих її операторів, а вже потім повернутися до питання паралельного перетворення арифметичних виразів.

Розглянемо доволі простий фрагмент програми, який зустрічається досить часто. Приклад написаний мовою Fortran, але він цілком може бути написаний і будь-якою іншою мовою, від цього властивості такого фрагмента програми не зміняться (фрагмент 1.1).

Фрагмент коду 1.1

```
DO k = 1, 1000
   DO j = 1, 40
      DO i = 1, 40
         A(i, j, k)=A(i-1, j, k) + B(j, k) + B(j, k)
      END  DO
   END DO
END DO
```

Програма, яка містила цей фрагмент, була відкомпільована за допомогою спеціального компілятора з можливістю автоматичного розпаралелювання й запущена на виконання на суперкомп’ютері CRAY Y-MP C90, який відповідно

[1] Рольщиков В.Б. Застосування засобів інтерфейсу передачі повідомлень при програмуванні розподілених систем мовою Java: навчальний посібник / Одеса, 2018. 209 с.

до паспортних технічних характеристик зобов'язаний забезпечувати пікову продуктивність 960 MFLOPS. Але коли експериментатори зробили вимірювання продуктивності комп'ютера при виконанні фрагмента, вони, на свій подив, виявили, що комп'ютер працює з продуктивністю лише 20 MFLOPS. Тобто ефективна продуктивність склала ледве більше за 2% від максимально можливої. Наведений приклад наочно показує, що застосування суперкомп'ютера для проведення обчислень ще не гарантує значного прискорення процесу і є цілком очевидним, що зменшення продуктивності комп'ютера пов'язано саме із властивостями програми і можливостями використаного компілятора.

Коли програмісти проаналізували отриманий результат, то виявилося, що компілятор не дуже добре використовує структуру кешпам'яті комп'ютера. Але змінити структуру кешпам'яті не можливо, тому було вирішено трохи перетворити фрагмент програми в такий спосіб (фрагмент 1.2):

Фрагмент коду 1.2

```
DO i = 1, 40, 2
  DO j = 1, 40
    DO k = 1, 1000
      A(i, j, k) = A(i-1, j, k) + 2*B(j, k)
      A(i+1, j, k) = A(i, j, k) + 2*B(j, k)
    END DO
  END DO
END DO
```

Незважаючи на те, що одна з операцій додавання була замінена на дві більш повільні операції множення, за рахунок еквівалентної перестановки параметрів циклів і еквівалентної розбивки оператора присвоювання на два послідовно виконуваних оператора, була отримана продуктивність 700 MFLOPS тобто вже більше за 70% від пікової.

Наведений приклад наочно демонструє залежність ефективності обчислень на суперкомп'ютері від структури програми. Тому дуже важливо навчитися робити розпаралелювання програм і алгоритмів.

Як вказувалося вище, для написання паралельної програми, у ній *необхід-*

но виділити групи операцій, які можуть обчислюватись одночасно й незалежно різними процесорами, функціональними пристроями або різними ступенями конвеєра.

При цьому власне задача розпаралелювання зводиться, по-перше, до *знаходження* в програмі достатньої кількості незалежних одна від одної груп операцій і, по-друге, *до рівномірного розподілу* їх між обчислювальними пристроями.

Для формалізації задачі й полегшення аналізу з метою її розпаралелювання, уводиться поняття ***графа програми***. Але граф програми поняття не досить чітке тому, що він у великому ступені залежить від мови програмування, на якій складається програма. Тому частіше говорять про більш абстрактне представлення програми й широко використовують модель у вигляді ациклічного орієнтованого графа, який має назву ***граф алгоритму***.

У цій моделі множина операцій алгоритму й існуючі між операціями залежності описуються парою: $G = (V, E)$, де $V = \{1,...,|V|\}$ – множина вершин графа, яка представляє операції алгоритму, а E – множина дуг графа, які встановлюють частковий порядок операцій. Дуга $E_{i,j} = (i, j)$ належить графу тільки в тому випадку, коли операція j виконується після операції i або використовує результат її виконання. Властивість ациклічності графа алгоритму полягає в тому, що ніяка величина не може визначатися через саму себе. Описана вище модель є ***спрямованим графом***.

У такій моделі алгоритму вершинами графа можуть бути:

- окремі оператори;
- цілі процедури;
- лінійні фрагменти, які містять декілька операторів;
- цикли;
- ітерації циклів;
- окремі спрацьовування операторів...

А для дуг, які пов’язують вершини й відбивають відношення між ними, виділяють два типи відносин:

– ***операційне*** *відношення*;

– ***інформаційне*** *відношення*.

Серед типів вершин є сенс окремо розглянути такі поняття як ***ітерації циклів*** і ***спрацьовування операторів***, оскільки інші типи вершин досить добре відомі з попередніх дисциплін й цілком зрозумілі. Розгляд і визначення цих понять найпростіше зробити за допомогою якого-небудь елементарного циклу й графів алгоритму, які йому відповідають.

Як приклад можна розглянути такий цикл (фрагмент 1.3):

Фрагмент коду 1.3

```
for(i = 0; i < n; ++i)
  {
    A[i] = A[i-1] + 2;
    B[i] = B[i] + A[i];
  }
```

Граф алгоритму ***ітерацій*** для цього циклу має такий вигляд (рис. 1.2).

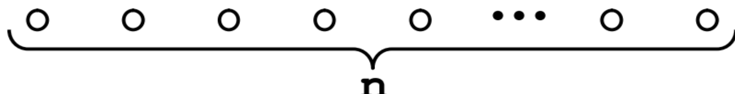


Рисунок 1.2 – Граф алгоритму ітерацій циклу

Тобто, у цьому випадку граф алгоритму має просто n вершин за кількістю ітерацій циклу. Відсутність дуг, які з’єднують вершини, пояснюється тим фактом, що граф побудований винятково для ілюстрації поняття ***ітерація циклу*** й абсолютно не враховує існування яких-небудь відносин між вершинами. Іншими словами, кожна вершина в цьому випадку відповідає відразу двом операторам – *тілу циклу*, виконаним на *одній і тій же ітерації циклу*.

У випадку ж, коли вершини представляють собою *спрацьовування операторів*, граф алгоритму того ж самого фрагмента 1.3 програми виглядає так, як це представлено на рис. 1.3.

Як видно з рисунка, кожна вершина тепер уже відповідає *одному з двох операторів тіла* даного циклу, виконаному на деякій ітерації. Дуги в графі від-

сутні з причини, яка повністю аналогічна описаній вище, тобто метою побудови графа є лише пояснення саме поняття ***спрацьовування операторів***.

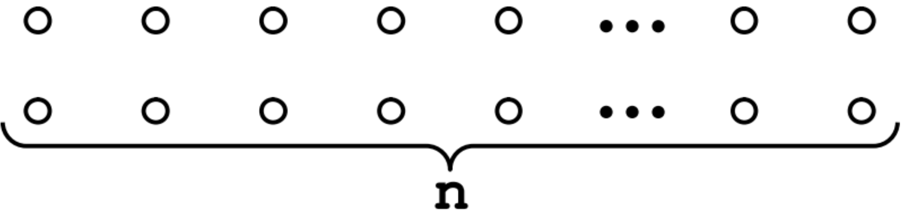


Рисунок 1.3 – Граф алгоритму для випадку спрацьовування операторів

На рис. 1.4 показані дві вершини A і B, пов'язані між собою спрямованою дугою.

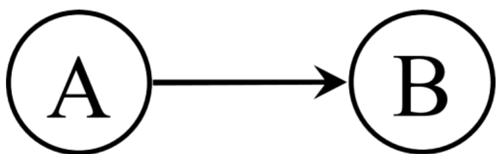


Рисунок 1.4 – Вершини графа зв'язані відношенням

Як сказано вище, відношення між вершинами може бути двох різних типів.

За визначенням дві вершини A і B з'єднуються спрямованою дугою, яка має тип ***операційне відношення*** (відношення з передачі керування), *тоді й тільки тоді*, коли операція вершини B може бути виконана відразу після виконання оператора вершини A.

Прикладом графа, між вершинами якого встановлюються такі відносини, може бути граф (рис. 1.5) такого фрагмента коду програми (фрагмент 1.4).

Фрагмент коду 1.4

```
x(i) = a + b(i)          // (1)
y(i) = 2 * x(i) - 3      // (2)
t1 = y(i) * y(i) + 1     // (3)
t2 = b(i) - y(i) * a     // (4)
```

Якщо розглядати виконання такого фрагмента коду програми з погляду черговості виконання операторів, як це робиться в парадигмі послідовного програмування, то цілком очевидно, що операції присвоювання повинні виконуватися в такому порядку: спочатку 1 потім 2 потім 3, нарешті, 4 і ніяк не інакше. Це й відображено в наведеному графі.

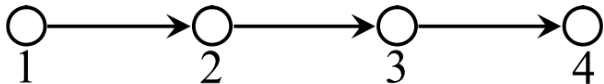


Рисунок 1.5 − Граф алгоритму з операційними відносинами для фрагмента 1.4

Можливість виділення в програмі груп операцій, які можуть обчислюватися одночасно й незалежно різними паралельними пристроями, обумовлюється наявністю або відсутністю в програмі ***інформаційних залежностей***. Дві операції програми (причому під операцією розуміють як окреме спрацьовування деякого оператора, так і більші фрагменти вихідного коду програми) називаються ***інформаційно залежними***, якщо результат виконання однієї операції використовується як аргумент в такій. Очевидно, що якщо операція B інформаційно залежить від операції A (тобто, використовує якісь результати операції A як свої аргументи), то операція B може бути виконана тільки по завершенні операції A. З іншого боку, якщо операції A і B не є інформаційно залежними, то алгоритмом не накладається ніяких обмежень на порядок виконання операцій, зокрема, вони можуть бути виконані одночасно. Таким чином, задача розпаралелювання програми за звичаєм зводиться до знаходження в ній достатньої кількості інформаційно незалежних операцій.

Іншими словами, дві вершини A і B з’єднуються спрямованою дугою, що має тип ***інформаційне відношення*** (відношення з передачі даних), *тоді й тільки тоді*, коли операція у вершині B використовує як аргумент деяке значення, отримане при виконанні операції у вершині A.

Уведене поняття інформаційної залежності є досить простим, але дослі-

дження всього набору інформаційних залежностей, що існують у реальній програмі, є дуже складною задачею. Досить лише уявити собі програму, що складається з десятків тисяч операторів, і врахувати, що кожен цикл може складатися з величезної кількості ітерацій, щоб приблизно визнати для себе складність виникаючих проблем. Для того, щоб формалізувати задачу й полегшити аналіз, уводиться поняття ***графів інформаційних залежностей***.

Вершинами в таких графах зазвичай є деякі операції програми, а у випадку, якщо між двома операціями існує інформаційна залежність, то відповідно цим операціям вершини з'єднуються спрямованою дугою, початком якої є вершина-постачальник інформації, а кінцем – вершина-споживач інформації. Для спрощення дослідження графа залежностей у ньому виділяється основний підграф, який має мінімальну кількість дуг при виконанні такої умови: якщо дві вершини вихідного графа залежностей зв'язані дугою, то вони ж повинні бути зв'язані дугою і у виділеному підграфі, який називається ***мінімальним графом залежностей***. За допомогою мінімальних графів залежностей можна вирішувати всі ті ж задачі, які можна вирішувати й за допомогою звичайних графів залежностей, однак вони набагато простіші й у більшості випадків дозволяють проводити не менш ефективний аналіз структури залежностей програми.

Граф інформаційних залежностей для того ж самого коду (фрагмент 1.4) виглядає вже так, як це представлено на рис. 1.6 а).

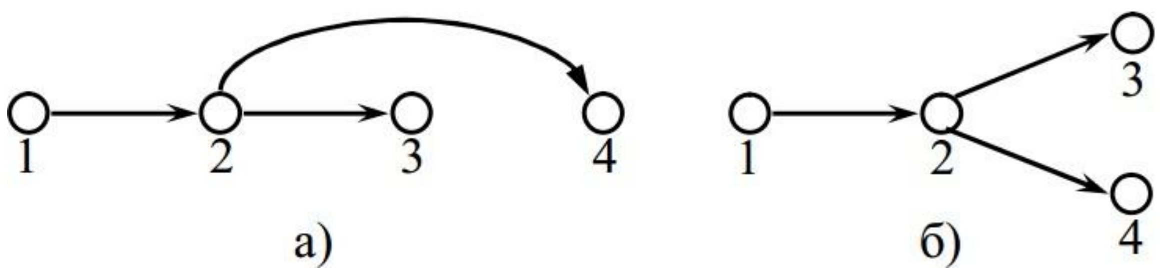


Рисунок 1.6 − Граф інформаційних залежностей для фрагмента коду 1.4

а) граф залежностей при послідовному проходженні операторів,

б) еквівалентно перетворений граф залежностей

Отриманий граф відображає той факт, що операція у вершині 2 інформаційно залежить від операції вершини 1, тобто y(i) може бути обчислено тільки тоді, коли закінчилося обчислення x(i). Крім того з наведеного графа видно, що операції у вершинах 3 і 4 інформаційно залежать тільки від операції у вершині 2, тобто від значення y(i) і безпосередньо ніяк не залежать від операції у вершині 1 (x(i)). Слід також зазначити, що відсутня інформаційна взаємозалежність між операціями у вершинах 3 і 4 – змінна t2 не залежить від змінної t1, яка обчислюється оператором у другій вершині графа. Це найбільш чітко видно на рис. 1.6 б) на якому зображений той же граф, що й на рис. 1.6 а), але еквівалентно перетворений до більш зручного вигляду. При уважному розгляді цього графа напрошується один досить очевидний висновок, який докладніше буде розглянутий далі на стор. 51.

Ґрунтуючись на тім, що є два типи вершин – *оператори* й *спрацьовування операторів*, а так само два типи дуг – *операційне відношення* й *інформаційне відношення*, можна говорити про існування чотирьох різних основних моделей представлення програм у вигляді графів. Ці моделі й типи вершин і дуг, які входять до їхнього складу, наведені в табл. 1.1

Якщо вершинам графа відповідають окремі спрацьовування операторів програми, то такий граф називається ***операційною*** або ***інформаційною історією*** виконання програми відповідно до типу дуги, яка їх з'єднує.

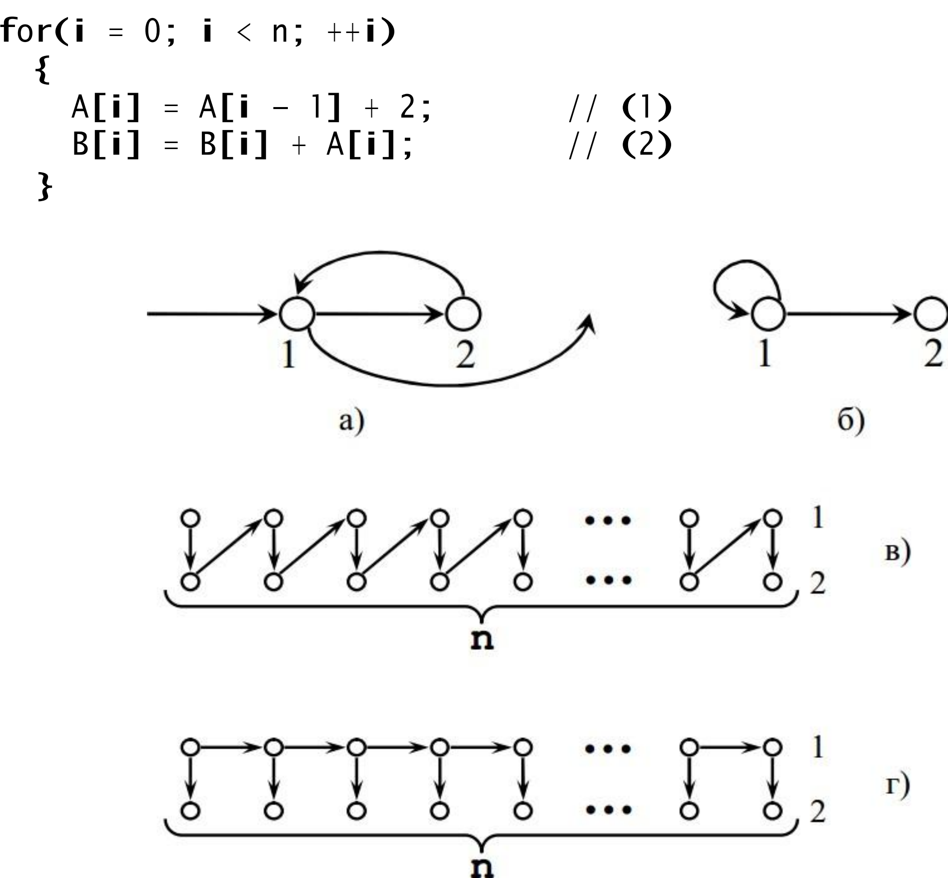
Таблиця 1.1 − Основні моделі графів алгоритмів

№	Модель програми	Вершини	Дуги
1	Граф керування	Оператори	Операційне відношення
2	Інформаційний граф	Оператори	Інформаційне відношення
3	Операційна історія	Спрацьовування операторів	Операційне відношення
4	Інформаційна історія	Спрацьовування операторів	Інформаційне відношення

На рис. 1.7 представлені графи основних моделей алгоритмів. Графи від-

повідають фрагменту коду 1.5, який власне повністю повторює фрагмент коду 1.3, лише для зручності кожен оператор циклу постачений коментарем, який вказує його номер усередині циклу. Відповідними номерами для визначеності позначені вершини в графах основних моделей.

Фрагмент коду 1.5



```
for(i = 0; i < n; ++i)
  {
    A[i] = A[i - 1] + 2;         // (1)
    B[i] = B[i] + A[i];          // (2)
  }
```


Рисунок 1.7 − Графи основних моделей алгоритмів

а) граф керування програми (вершини – оператори, дуги – операційні відношення), б) інформаційний граф програми (вершини – оператори, дуги – інформаційні відношення), в) операційна історія програми (вершини – спрацьовування операторів, дуги – операційні відношення), г) інформаційна історія програми (вершини – спрацьовування операторів, дуги – інформаційні відношення)

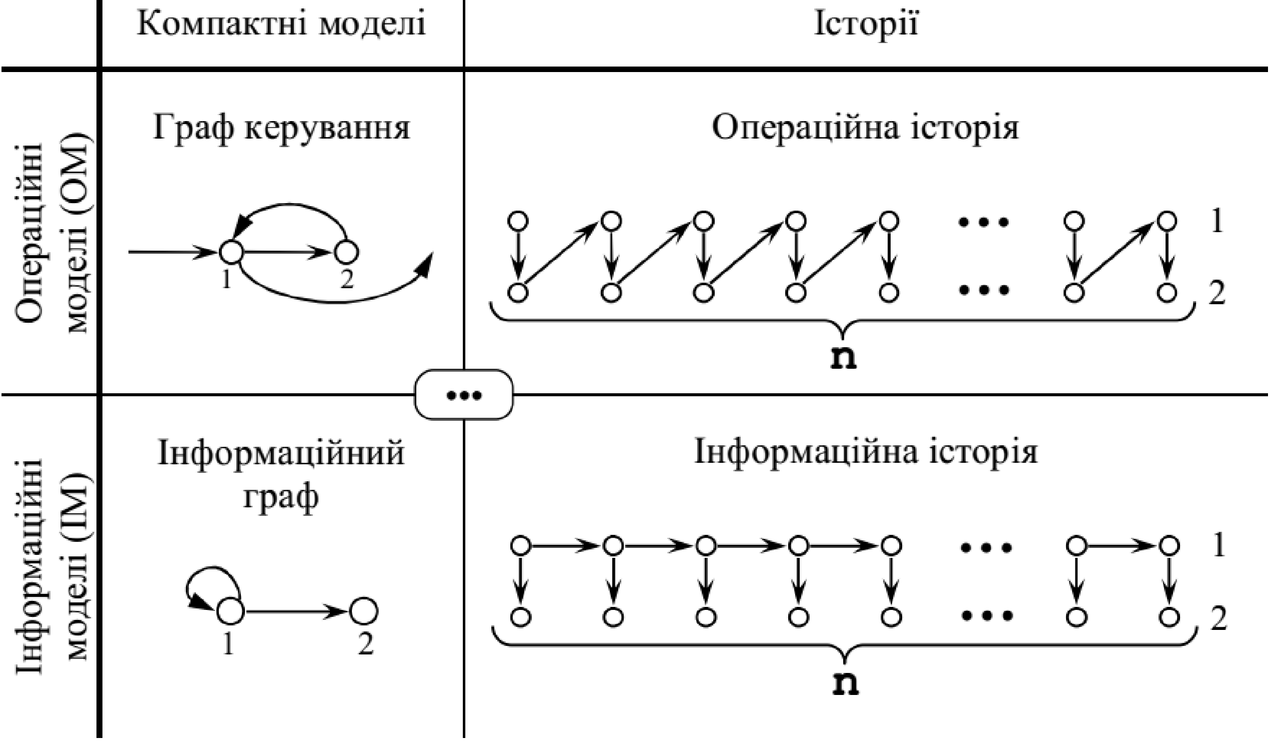
Рис. 1.7 а) ілюструє першу з моделей табл. 1.1 – ***граф керування програ-***

ми, у якого вершинами слугують *оператори*, а дуги відповідають *операційним відносинам*. На графі видно всі елементи циклу – вхід у цикл (дуга до вершини 1), послідовне виконання операторів тіла циклу (дуги між вершинами 1 і 2) і вихід із циклу (дуга від першої вершини). Граф другої моделі з табл. 1.1 – ***інформаційний граф програми*** наведений на рис. 1.7 б), тут, як і в першому випадку, вершинами слугують *оператори*, але дуги вже визначають інформаційні відносини. Петля в першій вершині графа показує інформаційну залежність цього оператора від значення, обчисленого на попередній ітерації (`A[i] = A[i - 1]...`), другий же оператор інформаційно залежить тільки від першого – дуга між 1 і 2. Графи двох типів історій – ***операційної історії програми*** й ***інформаційної історії програми*** наведені на рис. 1.7 в) і г) відповідно. За визначенням поняття *історії програми* у вершини графів відображаються *спрацьовування операторів*. Для *операційної історії програми* (рис. 1.7 в) дугами є *операційні відносини* і вони наочно демонструють передачу керування від оператора 1 до оператора 2 (дуги від 1 до 2) і зворотно (дуги від 2 до 1) протягом всіх ітерацій циклу. У графі *інформаційної історії програми* (рис. 1.7 г) дуги – *інформаційні відносини* протягом всіх ітерацій циклу демонструють інформаційну залежність першого оператора від значення, обчисленого на попередній ітерації (дуги від 1 до 1 – горизонтальні) і інформаційну залежність другого оператора від першого всередині однієї ітерації циклу (дуги від 1 до 2 – вертикальні).

Чотири описані моделі графів програм, як уже відзначалось, є основними й, так би мовити, опорними точками у всьому різноманітті моделей. Як для сімейства операційних моделей, так і для сімейства інформаційних існує цілий спектр моделей від компактних до розгорнутих. Умовно весь спектр моделей показаний у табл. 1.2. В таблицю поміщені початкові і кінцеві точки переходів. Це різноманіття моделей обумовлюється різноманіттям структурних елементів програми, різноманіттям і властивостями апаратних засобів на яких буде виконуватися програма, а так само вимогами до наочності й повноти представлення моделі програми у вигляді графа. Тому при аналізі структури програми важли-

во визначитися з тим, яку з моделей програми вибрати для найбільш ефективного проведення її аналізу і розв’язання задачі розпаралелювання програми.

Таблиця 1.2 − Демонстрація переходу від компактних моделей до розгорнутих



Наприклад, для представлення фрагмента коду 1.4 можна скористатися як моделлю операційного відношення (рис. 1.5), так і моделлю з використанням інформаційної залежності (1.6 б). Але, як видно з рис. 1.8, інформаційна модель має значну перевагу перед операційною при розробці паралельної програми.

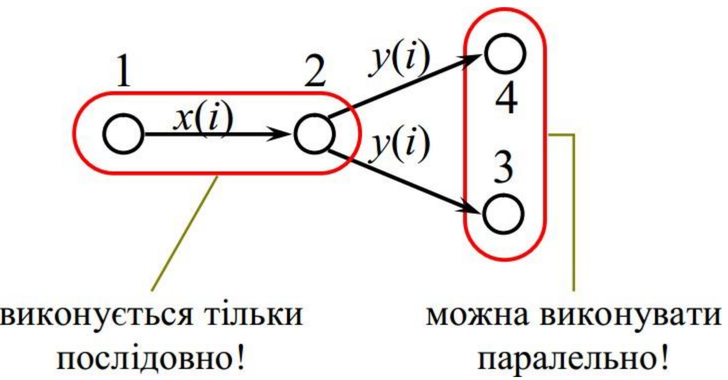


Рисунок 1.8 − Граф інформаційної історії фрагмента коду 1.4
з поділом на послідовну й паралельну частини

На цьому рисунку чітко видно, що оператори у вершинах 1 і 2 зв'язані інформаційною залежністю і повинні виконуватися тільки послідовно один за одним. Навпаки, оператори у вершинах 3 і 4 не мають інформаційної залежності між собою і *можуть виконуватись паралельно* відразу після виконання оператора у вершині 2.

Із сказаного випливає очевидний висновок: *інформаційна структура* – це *основа аналізу* властивостей програм і алгоритмів. При цьому *інформаційна залежність обумовлює критерій еквівалентності перетворень* програм. А от *інформаційна незалежність обумовлює ресурс паралелізму* програми.

За звичаєм при виборі моделі для опису властивостей програми користуються трьома критеріями вибору:

- компактність моделі;
- інформативність моделі;
- складність побудови моделі.

Якщо критерієм побудови графа моделі програми виступає компактність моделі опису програми або ж критерієм є полегшення процесу побудови моделі, то перевага віддається компактним моделям. У разі коли на перший план виходить одержання максимальної інформативності моделі, то використання компактних моделей стає проблематичним. Наприклад, компактній моделі на рис. 1.9 а) може відповідати як розгорнута модель у вигляді історії рис. 1.9 б), так і історія, наведена на рис. 1.9 в), а ще й безліч інших історій. Тобто компактна модель з погляду інформативності є неоднозначною.

На противагу компактній моделі, *інформаційна історія містить максимально докладну інформацію* про структуру інформаційних залежностей аналізованої програми. Тому практично у більшості випадків саме така модель використовується при аналізі програм з метою розпаралелювання. Однак найчастіше тут можуть виникнути проблеми, які складаються з такого. Складність аналізу графів інформаційних залежностей така, що якщо в простих випадках можна вручну побудувати й проаналізувати одержаний граф, то для великих реальних програм необхідне застосування спеціальних досить складних і дуже ко-

штовних інструментальних програмних засобів аналізу.

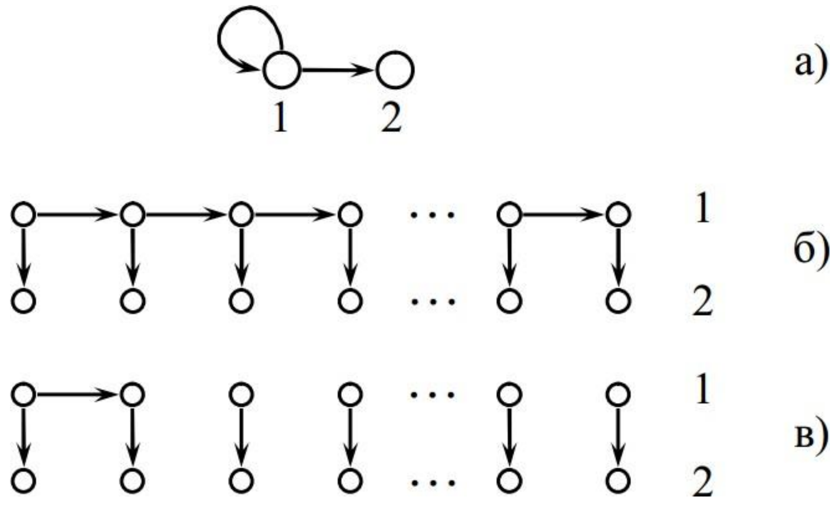


Рисунок 1.9 − Відповідність компактної й розгорнутої моделей
а) – компактна модель опису програми, б) і в) – дві різні історії,
які еквівалентні компактній моделі

Тому, як відзначалося раніше, замість моделі програми при аналізі за звичаєм переходять до побудови моделі алгоритму. Такий перехід виправдується тим, що граф алгоритму, фактично будучи *параметризованою інформаційною історією* програми, забезпечує:

− компактність опису за рахунок параметризації;
− повну інформативність історії;
− існують методики побудови графа алгоритму за вихідним текстом програм.

Представлення алгоритму програми у вигляді набору ансамблів операцій, причому в кожному ансамблі всі операції не зв'язані одна з одною, не є остаточним етапом зі створення паралельної програми. Необхідно ще вирішити питання розподілу цих ансамблів між процесорами або іншими обробними пристроями. Якщо архітектура паралельної системи дозволяє реалізовувати одночасно всі операції кожного ансамблю, то без урахування часу на передачі даних час виконання алгоритму буде майже прямо пропорційним числу ансамблів. Число ансамблів при такому представленні називається ***висотою алгоритму***. А

алгоритми, у яких висота менша від загального числа операцій називаються ***паралельними***. Представлення алгоритму через виконання послідовності ансамблів з незалежних операцій і обумовлює ***паралельну форму алгоритму***.

Найбільш очевидним способом розподілу завдань по процесорах є такий. В інформаційній історії позначаються ті операції, які залежать тільки від зовнішніх даних програми й говорять, що такі операції належать до першого ярусу інформаційної історії. На другий ярус поміщаються операції, які залежать тільки від операцій першого ярусу й зовнішніх даних. Третій ярус містить операції інформаційно залежні від результатів виконання операцій другого й першого ярусів і вихідних даних і так далі. При розподілених у такий спосіб операціях інформаційної історії, утворюється так звана ***ярусно-паралельна форма*** (***ЯПФ***) програми. На рис. 1.10 наведений приклад такої структури.

Аналіз вихідного графа (рис.1.10 а) трохи скрутний через нерегулярне розташування його вершин. Незначні перестановки вершин графа зі збереженням його топології приводять до більш зручної форми (рис.1.10 б). На цьому графі вже можна виділити 9 ярусів ЯПФ (рис. 1.10 в). Таким чином, видно, що вихідний граф алгоритму зводиться до паралельної форми з висотою, яка дорівнює числу ярусів – 9 при загальному числі операцій одинадцять. При уважному розгляді отриманої ЯПФ можна помітити, що вершина 4, знаходячись на п’ятому ярусі, інформаційно залежить лише від вершин 2 і 3, які належать другому ярусу, тому її можна перенести з п’ятого ярусу на третій. Таким кроком буде перенесення вершини 6 із шостого ярусу на четвертий, тому що вона інформаційно залежить від операцій на третьому ярусі (вершина 4). І, на завершення, вершини 7, 9, 11 можна перенести на яруси п’ять, шість і сім відповідно. Тим самим висота отриманої ярусно-паралельної форми програми буде дорівнювати семи.

У будь-якому орієнтованому ациклічному графі завжди можна виділити так званий ***критичний шлях*** – шлях максимальної довжини від початкової вершини до кінцевої. У розглянутому прикладі такий шлях може, наприклад, пролягати через вершини 1→3→4→6→7→9→11.

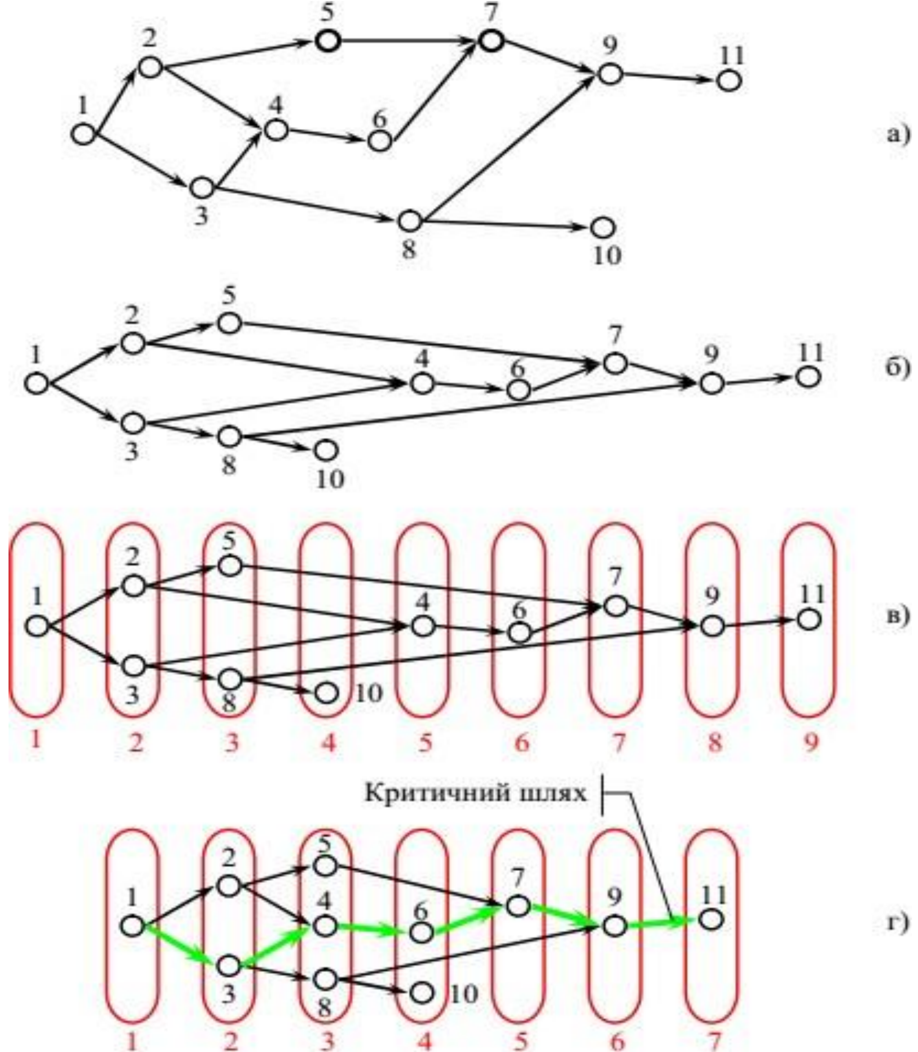


Рисунок 1.10 − Приклад переходу від графа програми до її канонічної ярусно-паралельної формі

а) вихідний граф програми, б) еквівалентно перебудований граф,

в) яруси ярусно-паралельної форми програми,

г) канонічна ярусно-паралельна форма програми

Довжина критичного шляху обумовлює *мінімальну висоту* із всіх можливих висот ярусно-паралельної форми даного ациклічного графа. ЯПФ, у якій кожна операція з ярусу з номером k більшим за одиницю, як один з аргументів одержує результат виконання деякої операції з попереднього (k – 1)-го ярусу і має мінімальну висоту, називається ***канонічною ярусно-паралельною формою*** програми. Для будь-якого алгоритму при заданих вхідних даних канонічна форма існує завжди і є єдиною. Висота канонічної ЯПФ на одиницю більша від

довжини критичного шляху. Таким чином, на рис. 1.10 г) представлена канонічна ЯПФ із висотою, яка дорівнює семи. Відповідно, довжина критичного шляху цієї ЯПФ дорівнює шести.

Кількість операцій на одному ярусі визначає ***ширину ярусу***, а максимальна ширина ярусів у канонічної ЯПФ називається ***шириною ЯПФ***. Для наведеного прикладу ширина ЯПФ дорівнює трьом – це ширина третього ярусу (рис. 1.10 г). У канонічній паралельній формі, як і в будь-якій іншій формі мінімальної висоти, яруси в середньому мають максимально можливу ширину.

Побудова ярусів канонічної ЯПФ виконувалась таким чином, щоб на один ярус потрапляли операції, які не перебувають в інформаційній залежності відносно одна до одної, а тому вони можуть виконуватись одночасно.

Однак подібне побудування канонічної ЯПФ програми можна зробити лише для дуже простих графів з достатньо структурованим розташуванням вершин і дуг. Але такі випадки зустрічаються дуже і дуже рідко, тому було б бажаним мати інший більш строгий метод створення ЯПФ програми. Такий метод існує і ґрунтується він на такому.

У курсі дискретної математики було показано, що будь-який граф може описуватись за допомогою так званої ***матриці суміжності*** A, у який кількість стовпців і рядків дорівнює кількості вершин графа, а значення елементів a_{ij} встановлюються у одиницю, якщо відповідні вершини зв'язані між собою, і дорівнюють нулю, якщо між вершинами дуги немає.

У свою чергу матриця суміжності дає можливість шляхом її перетворення безпосередньо одержати канонічну ярусно-паралельну форму графа програми. Алгоритм одержання канонічної ЯПФ програми потребує виконання таких кроків:

1) складається матриця суміжності графа;
2) в одержаній матриці суміжності виконується пошук нульових стовпців;
3) вершини, яким відповідають нульові стовпці, поміщаються в поточний ярус ЯПФ;
4) з матриці суміжності викреслюються стовпці й рядки, які відповідають

вершинам поточного ярусу;

5) пункти 2, 3 і 4 даного алгоритму повторюються доти, поки не будуть охоплені всі вершини;

6) за вихідною матрицею суміжності відновлюються дуги між вершинами.

Так, наприклад, для графа, зображеного на рис. 1.10 а), матриця суміжності має вигляд:

	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1	0	0	0	0	0	0	0	0
2	0	0	0	1	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	1	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0 .

З матриці цілком очевидно, що на першому ярусі ЯПФ повинна розташовуватись вершина за номером 1 оскільки стовпчик, який відповідає цій вершині, має лише нульові значення.

Після викреслювання першого стовпця і першого рядка перетворена матриця має вигляд:

	2	3	4	5	6	7	8	9	10	11
2	0	0	1	1	0	0	0	0	0	0
3	0	0	1	0	0	0	1	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0 .

В цій матриці вже потрібно викреслити стовпці і рядки з номерами 2 і 3, а відповідні вершини занести до другого ярусу ЯПФ. Знову перетворена матриця набере вже такого вигляду:

	4	5	6	7	8	9	10	11
4	0	0	1	0	0	0	0	0
5	0	0	0	1	0	0	0	0
6	0	0	0	1	0	0	0	0
7	0	0	0	0	0	1	0	0
8	0	0	0	0	0	1	1	0
9	0	0	0	0	0	0	0	1
10	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0 .

З неї вже потрібно викреслити 4, 5 і 8 стопці й рядки. Це говорить про те, що до третього ярусу ЯПФ належать три відповідні вершини. Зі знов одержаної матриці

	6	7	9	10	11
6	0	1	0	0	0
7	0	0	1	0	0
9	0	0	0	0	1
10	0	0	0	0	0
11	0	0	0	0	0 ,

з викреслюванням двох стовпців і двох рядків, відповідні їм шоста і десята вершини заносяться до четвертого ярусу. До останніх ярусів ЯПФ: п’ятого, шостого і сьомого, згідно з такими перетвореними матрицями

	7	9	11
7	0	1	0
9	0	0	1
11	0	0	0 ,

	9	11
9	0	1
11	0	0

і

	11
11	0

належать сьома, дев’ята і одинадцята вершини графа відповідно.

Як видно, в результаті послідовного виконання кроків, які пропонуються

алгоритмом, одержана канонічна ЯПФ, зображена на рис. 1.10 г).

Саме процес одержання паралельної програми виконується так: спочатку між процесорами розподіляються операції першого ярусу, після їхнього завершення – операції другого ярусу, потім третього й так далі. Оскільки будь-яка операція n-го ярусу залежить хоча б від однієї операції попереднього (n - 1)-го ярусу, то її не можна почати виконувати раніше, ніж завершиться виконання операцій цього ярусу. Тому канонічна ярусно-паралельна форма, в певному розумінні, обумовлює максимально паралельну реалізацію програми. При цьому довжина критичного шляху характеризує кількість паралельних кроків, необхідних для її виконання.

Однак у дійсності ярусно-паралельна форма майже ніколи не використовується для практичного розпаралелювання програм. Причиною цього є те, що описаний спосіб розпаралелювання погано погоджується як з конструкціями реальних мов програмування, так і з архітектурними особливостями різних систем сучасних комп’ютерів.

У реальності набагато частіше використовуються інші способи розпаралелювання, які ґрунтуються на застосуванні характерних рис найпоширеніших мов програмування. Ці способи основані на існуванні двох типів паралелізму – ***кінцевого*** і ***масового***, масовий паралелізм, у свою чергу, підрозділяється на ***координатний*** і ***скошений*** (рис. 1.11).

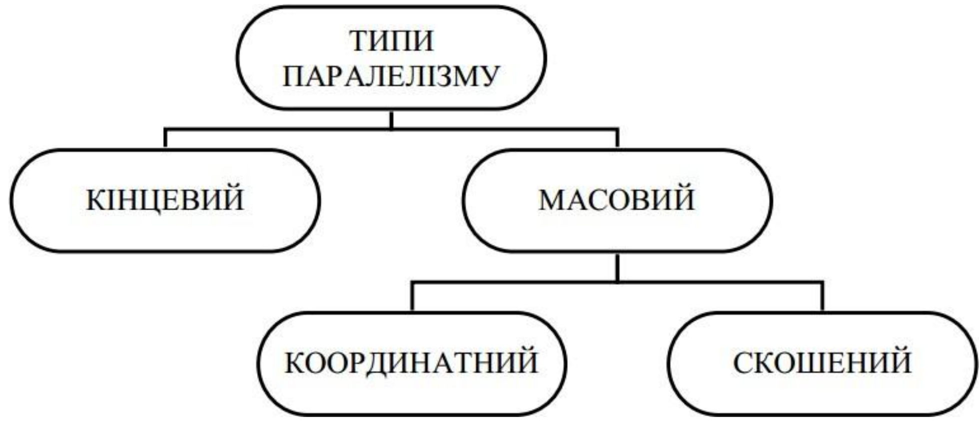


Рисунок 1.11 − Ієрархія типів паралелізму

Кінцевий паралелізм обумовлюється наявністю *інформаційної незалежності* між деякими досить *великими фрагментами* в тексті програми. ***Масовий паралелізм*** обумовлює більш низькорівневе розпаралелювання й обумовлюється *інформаційною незалежністю ітерацій циклів* програми.

Можливо найпростішим варіантом розподілу робіт між процесорами із застосуванням кінцевого паралелізму є спосіб, так званого, ***великоблочного розпаралелювання***.

Цей процес розпаралелювання програми можна описати таким псевдокодом:

```
if(Myproc = 0) {/* операції, виконувані 0-им процесором */}
...
if(Myproc = k) {/* операції, виконувані k-им процесором */}
```

При цьому передбачається, що кожен процесор якимсь чином може одержати свій унікальний номер, привласнити його змінній `MyProc` і використовувати її надалі для одержання фрагмента коду для незалежного виконання. Таким чином, у наведеному прикладі операції в перших фігурних дужках будуть виконані тільки процесором з номером 0, операції в других фігурних дужках – процесором з номером k і т.ін. При цьому, природно, необхідно, щоб одночасно різні процесори могли виконувати тільки блоки інформаційно незалежних операцій. Така вимога може зажадати виконання операцій синхронізації процесорів через неоднаковість їхніх фрагментів коду і різної скорості роботи процесорів. Крім того, при необхідності потрібно забезпечувати обмін даними між процесорами.

Нижче наведений фрагмент повністю умовного коду (фрагмент 1.6), який служить тільки лише для демонстрації технології великоблочного розпаралелювання й не несе ніякого іншого значеннєвого навантаження. На рисунку ж (рис. 1.12) зображена ЯПФ цього фрагмента після розбивки його на інформаційно незалежні частини.

Фрагмент коду 1.6

```
//...
cout << "N=" << N << endl;
//----------------------------------------
func1_KI_WithUnroll ("i");
func1_KI_WithUnroll ("k");
func2_KI_WithUnroll ("i");
func2_KI_WithUnroll ("k");
//---------------------------------------- (1)
func3_KJI("j, i, k");
func3_KJI("i, k, j");
func3_KJI("k, j, i");
//---------------------------------------- (2)
float ***a12=new float**[N];
for (i=0;i<N;i++) {
  a12[i]=new float*[N];
  for (j=0;j<N;j++) {
    a12[i][j]=new float[N];
    for (k=0;k<N;k++)
      a12[i][j][k]=(float)1/(i+j+k+1);
    }
  }

//---------------------------------------- (3)
for (i=1;i<N;i++)
  for (j=1;j<N;j++)
    for (k=1;k<N;k++) {
      testee[i][k] = testee[i][k] +
        S[k]*A[k][j][i] + P[i][j]*A[k][j][i-1] +
        P[i][k]*A[k][j-1][i] + P[j][k]*
        A[k-1][j][i];
    }
//...
```

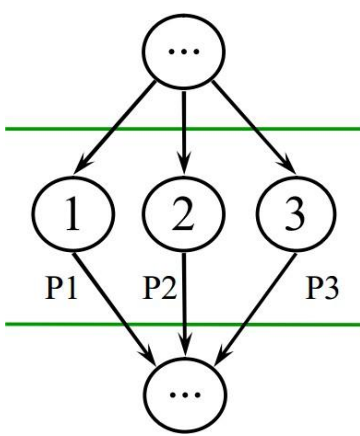


Рисунок 1.12 − ЯПФ фрагмента коду 1.6

Частини коду, позначені коментарями 1, 2 і 3, і відповідні їм вершини можуть розташовуватись на одному ярусі ЯПФ і виконуватися паралельно, кожна на своєму окремому процесорі P1, P2 і P3.

Однак далеко не завжди вдається виділити в програмі досить велику кількість блоків незалежних операцій, а значить, при наявності великої кількості процесорів у використовуваному комп'ютері, частина з них буде простоювати. Тому, як було сказано вище, поряд з попереднім способом використовують також більш низькорівневе розпаралелювання.

Як показує практика, найбільший ресурс паралелізму в програмах зосереджений у циклах. Тому найпоширенішим способом розпаралелювання є той або інший спосіб ***розподілу ітерацій циклів***. Якщо між ітераціями деякого циклу немає інформаційних залежностей, то їх можна тим або іншим способом роздати різним процесорам для одночасного виконання. Умовно це може бути виражено приблизно такою конструкцією псевдокоду:

```
for (i=0; i < k; i++)
  if (i ~ MyProc) {/* операції i-ої ітерації для виконання
                        процесором MyProc */}
```

Тут конструкція `i ~ MyProc` застосована для того, щоб указати, що номер ітерації `i` якимсь чином співвідноситься з номером процесора `MyProc`. Конкретний спосіб завдання цього співвідношення обумовлює те, які ітерації циклу на які процесори будуть розподілені. У принципі, ніщо не заважає, наприклад, роздати всім процесорам по одній ітерації, а всі інші ітерації виконати якимсь одним процесором. Однак очевидно, що у дуже великій кількості випадків такий розподіл виявляється неефективним, оскільки всі процесори, крім одного, виконавши свою ітерацію, будуть скоріш за все простоювати. Таким чином, однією з вимог до розподілу ітерацій (як, втім, і до процесу розпаралелювання взагалі) є по можливості *рівномірне завантаження процесорів*. Найпоширеніші способи розподілу ітерацій циклів тією чи іншою мірою задовольняють цю вимогу. Серед цих способів розподілу виділяють:

− блокове розподілення;

– блочно-циклічне розподілення;

– циклічне розподілення.

Блокове розподілення ітерацій припускає, що розподіл ітерацій циклу по процесорах ведеться блоками по кілька послідовних ітерацій. У найпростішому випадку для кількості ітерацій у циклі, яка дорівнює n і кількості процесорів у системі p, функція ***стеля*** обумовлює кількість ітерацій у блоці $k = \lceil n / p \rceil$, тобто кількість ітерацій ділиться на число процесорів і результат округляється до найближчого цілого зверху. Програмно це можна визначити, наприклад таким чином:

```
n % p ? n / p + 1 : n / p.
```

При цьому майже всі процесори одержують однакову кількість ітерацій, однак один або кілька останніх процесорів можуть простоювати.

Вибір меншого розміру блоку може зменшити цей дисбаланс, однак при цьому частина ітерацій залишиться нерозподіленою. Якщо нерозподілені ітерації, після завершення виконання першого блоку ітерацій, знову почати розподіляти такими ж блоками, починаючи з першого процесора, то утворюється так званий ***блочно-циклічний розподіл*** ітерацій.

Якщо й надалі зменшувати кількість ітерацій, то дійдемо до розподілу по одній ітерації. Такий розподіл ітерацій має назву ***циклічний розподіл ітерацій***. Циклічний розподіл дозволяє мінімізувати дисбаланс у завантаженні процесорів, який найчастіше виникає при блокових розподілах.

Наприклад, блоковий розподіл ітерацій для звичайного циклу підсумовування

```
for (i = 0; i < n; i++) a[i] = a[i] + b[i];,
```

у випадку якщо в цільовому комп’ютері є p процесорів з номерами від 0 до p-1 для процесора з номером MyProc можна записати в такий спосіб (фрагмент 1.7):

Фрагмент коду 1.7

```
k = (n - 1) / p + 1;                // обчислення розміру блоку
                                    // ітерацій
ibeg = MyProc * k;                  // індекс початку блоку для
                                    // процесора MyProc
iend = (MyProc + 1) * k - 1;        // індекс кінця блоку для
                                    // процесора MyProc
if(ibeg >= n) iend = ibeg - 1;      // якщо процесору не
                                    // дісталося ітерацій
else if(iend >= n) iend = n - 1;    // якщо дісталося менше
                                    // ітерацій
// нарешті цикл для процесора з номером MyProc
for(i = ibeg; i <= iend; i++) a[i] = a[i] + b[i];
```

Циклічний же розподіл ітерацій для того ж вихідного циклу можна записати так:

```
for (i = MyProc; i < n; i += p) a[i] = a[i] + b[i];
```

На рис. 1.13 зображені графи алгоритмів для обох розглянутих способів розподілу ітерацій циклів для паралельного виконання одинадцяти ітерацій на системі, що складається з трьох процесорів.

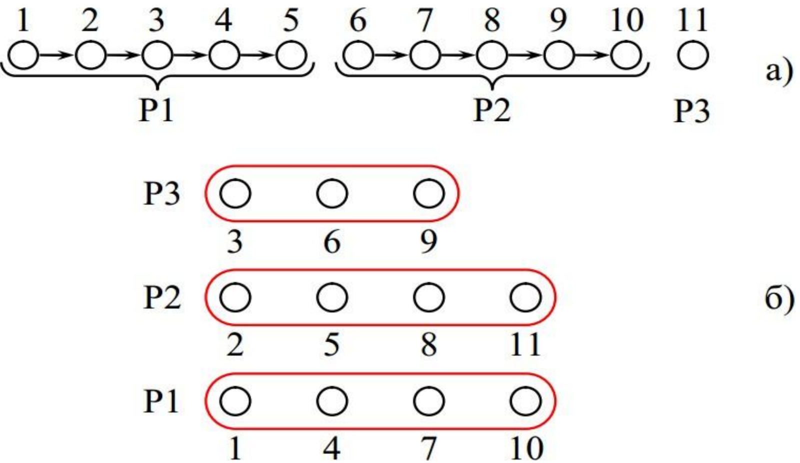


Рисунок 1.13 – Графи алгоритмів розподілу ітерацій циклів

а) операційна історія для блокового розподілу,

б) ЯПФ для циклічного розподілу ітерацій

Рис. (1.13 а) представляє операційну історію блокового розподілу ітерацій

циклу. Для зазначених параметрів – кількості ітерацій циклу $n = 11$ і кількості процесорів $p = 3$ при блоковому розподілі ітерацій, функція стеля дає значення кількості ітерацій на процесор k, яке дорівнює п'яти. Тому першому й другому процесорам виділяється 5 ітерацій, а на долю третього залишається одна ітерація, тобто спостерігається явний дисбаланс завантаження процесорів. ЯПФ для циклічного розподілу ітерацій (рис. 1.13 б), коли на кожному ярусі, кожен з процесорів виконує тільки одну ітерацію, наочно демонструє більш рівномірний розподіл ітерацій (ширина перших трьох ярусів дорівнює трьом, а останнього двом).

Слід відмітити, що всі міркування про розподіл ітерацій циклів з метою досягнення рівномірності завантаження процесорів мають рацію тільки в припущенні, що розподіляються ітерації, які приблизно рівноцінні за часом виконання. Але у багатьох реальних випадках (наприклад, при розв'язанні матричних задач із трикутною матрицею) таке припущення може виявитись практично не правільним, а це означає, що можуть знадобитися цілком інші способи розподілу циклів.

Однак, тільки рівномірного завантаження процесорів у більшості практичних випадків ще недостатньо для одержання ефективної паралельної програми. Тільки у вкрай рідких випадках програма не містить інформаційних залежностей взагалі. Якщо ж є інформаційна залежність між операціями, які, при обраній схемі розподілу, потрапляють на різні процесори, то в цьому разі буде потрібним виконувати пересилання даних між процесорами. На практиці пересилання даних потребують досить великого часу для свого здійснення. Тому іншим важливим завданням при розпаралелюванні коду є мінімізація необхідної *кількості* й *обсягу* пересилань даних. Так, наприклад, при наявності інформаційних залежностей між i-ою і $(i + 1)$-ою ітераціями деякого циклу, блоковий розподіл ітерацій може виявитися ефективнішим за циклічний, тому що при блоковому розподілі сусідні ітерації попадають на один процесор, а відповідно, буде потрібна менша кількість пересилань, ніж при циклічному розподілі. Це твердження для коду з наявністю вказаних інформаційних зв'язків

```
for(i=MyProc; i<n; i += p) a[i] = a[i-1] + b[i];
```

наочно ілюструється рис. 1.14.

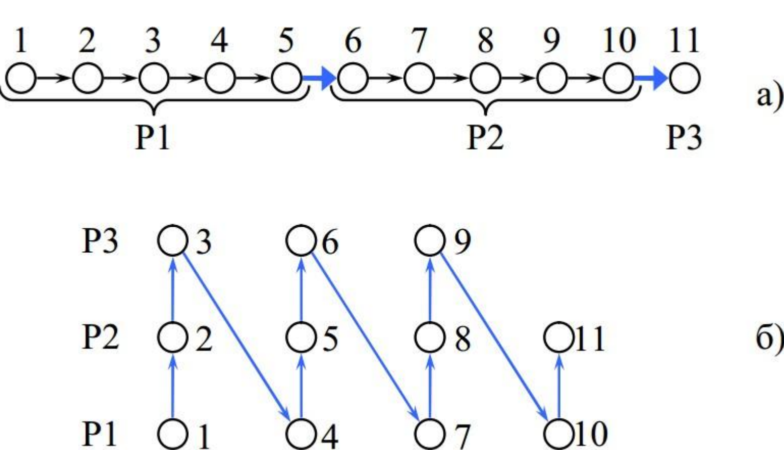


Рисунок 1.14 − Графи алгоритмів розподілу ітерацій циклів

а) інформаційна історія для блокового розподілу,

б) ЯПФ для циклічного розподілу ітерацій

На рис. 1.14 а) видно, що пересилання даних між процесорами у випадку блокового розподілу й наявності інформаційних зв’язків у циклі виконується всього лише два рази (грубі стрілки). У той же самий час у випадку циклічного розподілу ітерацій (рис. 1.14 б) створення ярусів ЯПФ неможливе, програма повністю послідовна і виконується десять операцій пересилання даних.

Дотепер мова йшла про розподіл ітерацій одновимірних циклів, однак у програмах часто зустрічаються багатовимірні циклічні гнізда, причому кожен цикл такого гнізда може містити деякий ресурс паралелізму. Для використання такого ресурсу паралелізму вводиться поняття *простору ітерацій,* і виконують аналіз і розбивку цього простору для досліджуваного фрагмента.

Простором ітерацій гнізда тісновкладених циклів називають кінцеву множину $N:=\{n_1, n_2, \dots n_n\}$ цілочисельних векторів n_i, координати яких задаються значеннями параметрів циклів даного гнізда.

Задача розпаралелювання в цьому випадку зводиться до розбивки множини векторів N на підмножини N_k (N_k □ N), які виконуються послідовно одна

за одною, але в рамках кожної такої підмножини ітерації можуть бути виконані одночасно й незалежно.

Серед методів аналізу простору ітерацій можна виділити декілька найбільш відомих – це методи *координат*, *паралелепіпедів* (різновиди координатного паралелізму), *гіперплощин* і *пірамід* (скошений паралелізм).

Метод гіперплощин полягає в тому, що простір ітерацій розмірності n розбивається на гіперплощини розмірності n–1 таким чином, що всі операції, які відповідають точкам однієї гіперплощини, можуть виконуватися одночасно й асинхронно.

Метод координат полягає в тому, що простір ітерацій фрагмента розбивається на гіперплощини, ортогональні однієї з координатних осей.

Метод паралелепіпедів є логічним розвитком двох попередніх методів і полягає в розбивці простору ітерацій на n-вимірні паралелепіпеди, об’єм яких обумовлює результуюче прискорення програми.

У ***методі пірамід*** вибираються ітерації такі, що операції на цих ітераціях виробляють значення, які далі в тілі гнізда циклів не використовуються. Кожна така ітерація слугує основою окремої паралельної гілки, у яку також входять всі ітерації, що інформаційно впливають на обрану. При цьому найчастіше спостерігається неприємне явище, коли інформація в різних гілках дублюється, а це може спричинити втрату ефективності.

За допомогою прикладу простого циклічного гнізда (фрагмент 1.8) можна продемонструвати використання поняття *простору ітерацій*.

Фрагмент коду 1.8

```
for(i = 1; i <= n; ++i)
  for(j = 1; j <= m ; ++j)
    a[i][j] = a[i][j - 1] + c[i][j] * x;
```

Простір ітерацій даного фрагмента можна зобразити в такий спосіб (рис. 1.15):

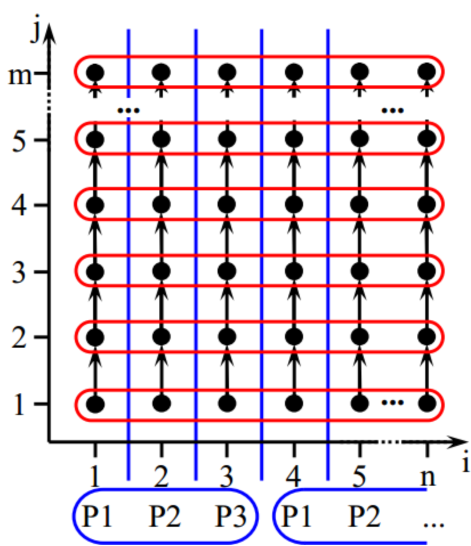


Рисунок 1.15 – Простір ітерацій подвійного циклу 1.8

На цьому рисунку чорні кружечки відповідають окремим спрацьовуванням оператора присвоювання у внутрішньому циклі, а стрілки показують інформаційні залежності значення змінної a від її значення на попередній ітерації по змінній j. Відразу видно, що розбивка простору ітерацій по виміру **j** призведе до розриву інформаційних залежностей, а це викличе необхідність виконувати багату кількість пересилань даних. Однак інформаційних залежностей по виміру **i** не існує, тому в цьому випадку можливо застосування методу координат з розбивкою простору ітерацій гіперплощинами, ортогональними осі **i**, наприклад, як це зображено на рисунку вертикальними лініями. Таким чином, утворюється ЯПФ висота якої дорівнює m, а ширина – n. Тепер ітерації зовнішнього циклу можна розподілити по процесорах цільового комп'ютера, наприклад, блоковим або блочно-циклічним способом.

У розглянутому прикладі спостерігалися інформаційні залежності лише по одному з вимірів – j. Якщо ж додати ще й інформаційну залежність по виміру **i** (фрагмент 1.9), то картина значно зміниться.

Фрагмент коду 1.9

```
for(i = 1; i <= n; ++i)
  for(j = 1; j <= m; ++j)
    a[i][j] = a[i][j-1] + a[i-1][j]*x;
```

Простір ітерацій даного фрагмента зображений на рис. 1.16.

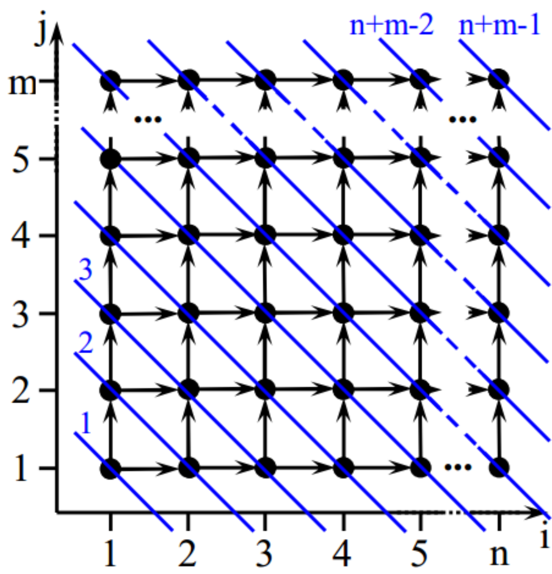


Рисунок 1.16 – Простір ітерацій подвійного циклу з інформаційними залежностями по обох координатах (скошений паралелізм)

Очевидно, що метод координат у цьому випадку вже не може застосовуватись, тому що будь-яка розбивка циклів як по виміру i, так і по виміру j призведе до розриву інформаційних залежностей. Однак на рисунку похилими лініями показані проекції гіперплощин, які проведені за законом

$$i + j = \text{const},$$

і містять вершини, між якими немає інформаційних залежностей. Це означає, що можливо створити ЯПФ програми, яруси якої збігаються з обраними гіперплощинами. Висота такої ЯПФ буде дорівнювати $n + m - 1$, а ширина дорівнює $\max(n, m)$ – максимальному значенню з вимірів двох координат. Здійснюючи тепер перебір гіперплощин $i + j = \text{const}$ для кожного з ярусів, відповідні операції розподіляються між процесорами цільового комп’ютера.

Простір ітерацій для методу паралелепіпедів можна продемонструвати для коду (фрагмент 1.10), характерного для задачі перемножування матриць. Такий простір ітерацій представлений на рис. 1.17.

Фрагмент коду 1.10

```
      DO 20 I = 1, N
      DO 20 J = 1, N
      A(I, J) = 0
      DO 20 K = 1, N
20    A(I, J) = A(I, J) + B(I, K) * C(K, J)
```

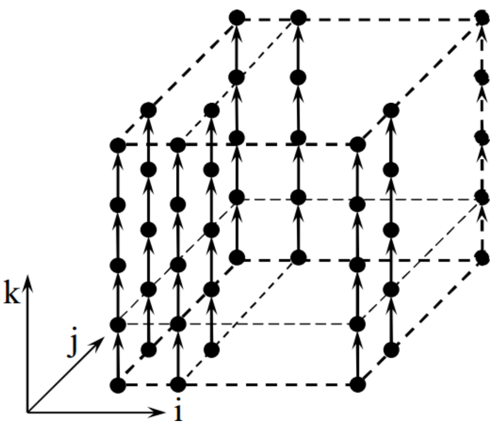


Рисунок 1.17 − Простір ітерацій для методу паралелепіпедів

Якщо переходити від розпаралелювання окремих циклічних конструкцій до розпаралелювання цілої програми, то може виявитися невигідним використовувати весь знайдений ресурс паралелізму (особливо, на комп'ютерах з розподіленою пам'яттю), оскільки буде потрібна велика кількість перерозподілів даних між виконанням циклів.

У деяких випадках можна домогтися більш ефективного розпаралелювання програми за допомогою ***еквівалентних перетворень*** – таких перетворень коду програми, при яких повністю зберігається результат її виконання. Існує досить велика кількість подібних перетворень, корисних у різних випадках, наприклад: ***розподіл циклів***, ***схлопування циклів***, ***розщеплення циклів*** і т.ін.

Еквівалентні перетворення від вихідної програми до перетвореної здійснюються через етапи побудови алгоритму програми, його дослідження й подальшого перетворення. Уже за перетвореним алгоритмом вносяться зміни в програму.

Яскравим прикладом *розподілу циклів* може слугувати еквівалентне пере-

творення такого циклу (фрагмент 1.11).

Фрагмент коду 1.11

```
for(i= i < n; ++i)
  {
    a[i] = a[i - 1]*p + q;          // 1
    c[i] = (a[i] + b[i - 1])*s;     // 2
    b[i] = (a[i] - b[i])*t;         // 3
  }
```

На рис. 1.18 представлені етапи перетворення алгоритму для цього циклу.

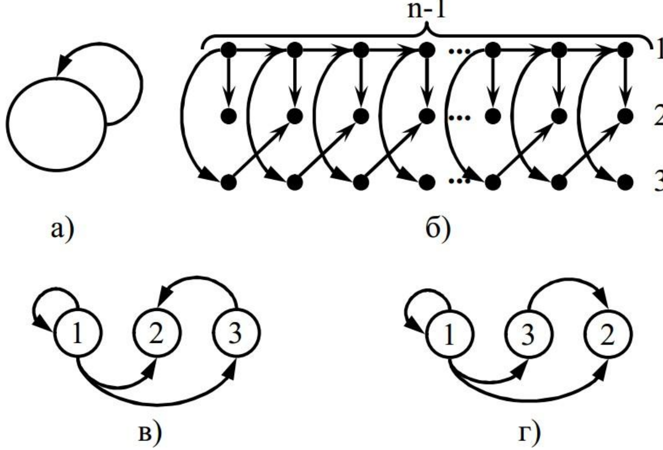


Рисунок 1.18 − Процес перетворення й дослідження алгоритму для циклу 1.11 а) компактний граф керування програми, б) інформаційна історія, в) компактний варіант інформаційної історії, г) перетворений алгоритм

На рис. 1.18 а) наведена компактна модель графа керування програмою для вихідного фрагмента програми. Петля зв’язку оператора самого на себе вказує на те, що фрагмент є звичайним одиночним циклом. Але, як було зазначено вище, найбільший обсяг корисної інформації про алгоритм несе в собі інформаційна історія, граф якої наведений на рис. 1.18 б). З інформаційної історії видно, що в даному циклі є інформаційні залежності між спрацьовуваннями першого оператора з тіла циклу на ітерації з номером **i** від спрацьовування його на попередній ітерації. Крім того, спостерігається така ж залежність другого

оператора від значення, яке отримано в третьому операторі після попередньої операції. Тому не можна просто роздати ітерації вихідного циклу для виконання різним процесорам, а необхідно якимсь чином перетворити алгоритм. Отримана інформаційна історія досить складна для дослідження, тому для спрощення в ній виділений мінімальний граф, який, не дивлячись на свої невеликі розміри, є повністю еквівалентним вихідному інформаційному графу (рис. 1.18 в). Аналіз цього графа дозволяє зробити висновок, що для розпаралелювання програми досить в алгоритмі зробити перестановку операторів програми 2 і 3 місцями (1.18 г) і розподілити вихідний цикл програми на три окремих цикли (фрагмент 1.12).

Фрагмент коду 1.12

```
for(i= i < n; ++i) A[i] = A[i - 1]*p + q;
for(i = 1; i < n; ++i) B[i] = (A[i] - B[i])*t;
for(i = 1; i < n; ++i) C[i] = (A[i] + B[i - 1])*s;
```

На перший погляд програма виглядає дуже дивно – замість одного циклу – цілих три. Але якщо вихідний цикл в принципі не може бути розпаралелений, то в перетвореній програмі другий і третій цикли, хоча й виконуються послідовно один за одним, але кожен з них може виконуватись у паралельному режимі через відсутність усередині них інформаційних залежностей. Перший же цикл, маючи у своїй структурі інформаційну залежність, все-таки теж може бути розпаралелений одним зі способів розподілу ітерацій циклів, описаних вище. На додаток до сказаного, в перетвореному фрагменті мінімізована кількість пересилань даних – замість пересилань на кожній ітерації виконуються тільки два пересилання в проміжках між циклами.

Розщеплення циклів можна проілюструвати на такому прикладі, у якому передбачається, що в програмі присутній цикл, представлений фрагментом 1.13:

Фрагмент коду 1.13

```
for(i=50 i <= 2000; ++i) a[i] = a[i] + a[i - 500];
```

Процес аналізу й перетворення алгоритму представлений на рис. 1.19. У цьому випадку, як і в попередньому, аналіз виробляється, починаючи з побудови компактної моделі графа керування програми (рис. 1.19 а). Топологія інформаційної історії (рис. 1.19 б) цілком підтверджує той факт, що цикл на кожній ітерації має інформаційну залежність від ітерації, виконаній на 500 кроків раніше. Але уважне вивчення й аналіз графа дозволяє перетворити цей послідовний алгоритм у ЯПФ, шириною 500 і висотою всього 3 послідовно виконуваних один за одним ярусів (рис. 1.19 в) лише із двома операціями пересилання даних. Для цього цілком достатньо зробити розщеплення вихідного циклу на три окремих цикли, як це показано вертикальними лініями на рис. 1.19 б). При цьому кожен такий окремий цикл утворює один ярус ЯПФ.

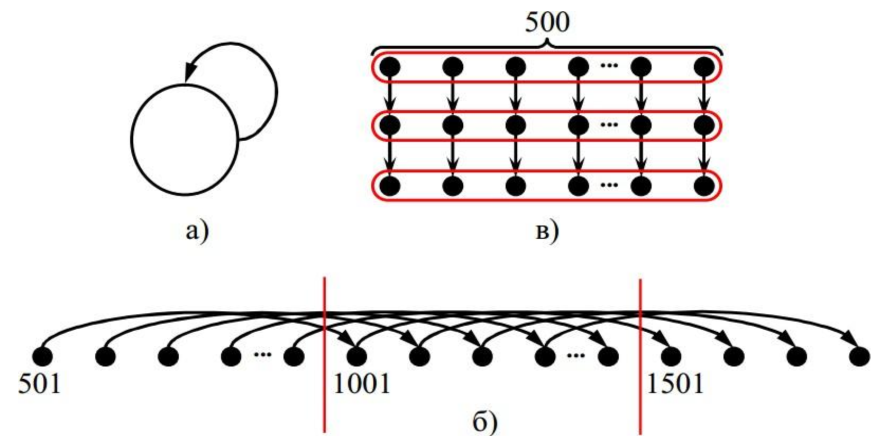


Рисунок 1.19 − Перетворення графа програми циклу 1.13
а) компактний граф керування програми, б) інформаційна історія,
в) канонічна ЯПФ алгоритму після розщеплення циклу

У книзі «Паралельні обчислення» авторів Воєводіна Вл.В і Воєводіна В.В.[2] наведений досить цікавий приклад перетворення програми. Автори жартуючи на-

[2] Воеводин Вл.В, Воеводин В.В. Параллельные вычисления / СПб, БХВ-Питербург, 2002. 608 с.

звали цей приклад «Дуже “простий” приклад». Код приклада наведений на фрагменті 1.14.

Фрагмент коду 1.14

```
DO I = 1, N
       DO 10 J = 1, N
10     U(I + J) = U(2 * N - I - J + 1) * Q + P
```

За твердженням авторів цей приклад не був узятий ні з теорії, ні з практики. Він спеціально придуманий для того, щоб показати, наскільки складними можуть бути інформаційні зв’язки навіть у начебто найпростіших алгоритмах. Формально цей приклад реалізує деяке сортування з n чисел. Незважаючи на уявлювану простоту коду, граф розглянутого алгоритму у дійсності влаштований дуже складно. Це видно хоча б з рис. 1.20, на якому зображені інформаційні структури коду для випадку n = 10.

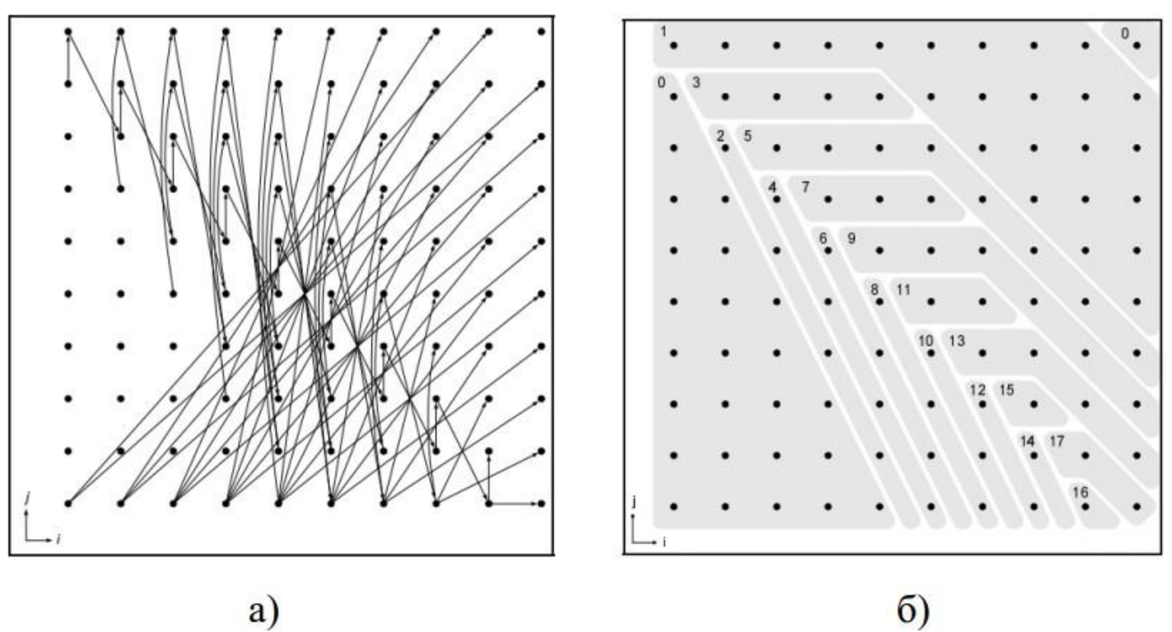


Рисунок 1.20 − Інформаційні структури уривка коду 1.14

а) інформаційна історія, б) яруси канонічної ЯПФ

Дивлячись на інформаційну історію (рис. 1.20 а), не можна відразу визначити чи є деякий паралелізм фрагмента або по координаті i, або по координаті j.

Не має особливого сенсу наводити всі етапи дослідження інформаційної історії, оскільки такий аналіз досить складний. Але авторами, на основі отриманої ними канонічної ЯПФ коду (рис. 1.20 б), наводиться перетворена програма (фрагмент 1.15).

Фрагмент коду 1.15

```
      DO I = 1, N
      DO 20 J = 1, N – I
20    U(I + J) = U(2 * N – I – J + 1) * Q + P
      DO 30 J = N – I + 1, N
30    U(I + J) = U(2 * N – I – J + 1) * Q + P
10    CONTINUE
```

У цьому перетвореному фрагменті при будь-якому фіксованому значенні **i** операції по координаті **j** у кожному із двох циклів по мітках 20 і 30 уже можна виконувати паралельно. Самі ж цикли виконуються послідовно один за одним.

Цікаво, що автори запропонували використовувати цей фрагмент як хороший тест для перевірки того, наскільки ефективно та або інша технологія визначає паралелізм у записах алгоритмів. З'ясувалося, що всі доступні авторам на сьогодні компілятори й автономні системи програмування, які здатні проводити розпаралелювання коду, як вітчизняні, так і закордонні не змогли виявити ніякого паралелізму у фрагменті.

Досить нетривіальним прикладом перетворення коду може слугувати перетворення в паралельну форму алгоритму (фрагмент 1.16) розв'язання системи лінійних алгебраїчних рівнянь (СЛАР) методом Гаусса.

Фрагмент коду 1.16

```
DO I = N, 1, -1
      S = 0
      DO 20 J = I + 1, N
20    S = S + A(I, J) * X(J)
10    X(I) = (B(I) - S) / A(I,I)
```

Графи інформаційних залежностей у процесі аналізу й перетворення алгоритму наведені на рис. 1.21.

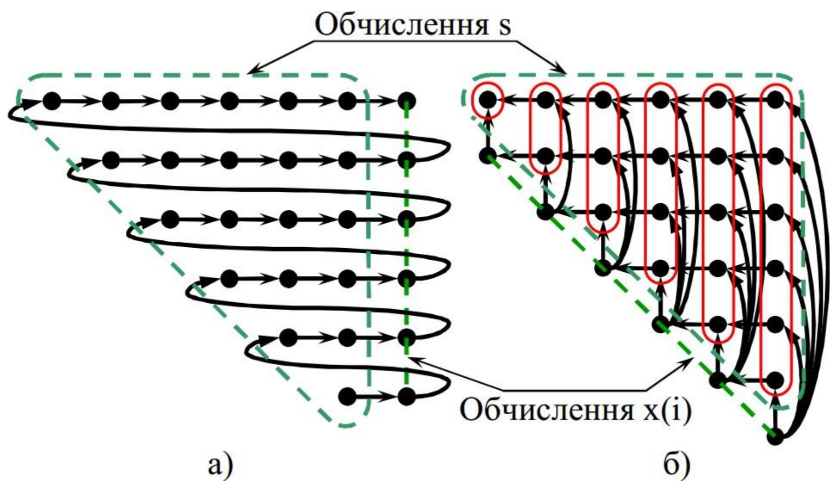


Рисунок 1.21 − Перетворення графа алгоритму розв’язання СЛАР методом Гаусса
а) інформаційна історія вихідного алгоритму,
б) ЯПФ перетвореного алгоритму

Граф інформаційної історії вихідного алгоритму (рис. 1.21 а) не такий складний, як у попередньому прикладі, але його топологія вказує на абсолютно послідовний характер алгоритму. У графі штриховими лініями виділені дві групи операцій – обчислення суми s і обчислення значень вектора рішень СЛАР – x(i).

Але якщо трохи змінити взаєморозташування цих груп, залишивши незмінними інформаційні залежності між операціями в групах, то граф перетвориться до вигляду, показаному на рис. 1.21 б). Такий граф уже є канонічною ЯПФ і операції обчислення сум уже можуть виконуватися паралельно відразу на декількох процесорах.

У коді, отриманому на основі перетвореного графа (фрагмент 1.17),

Фрагмент коду 1.17

```
DO I = N, 1, -1
       S = 0
```

```
      DO 20 J = N, I + 1, -1
20    S = S + A(I, J) * X(J)
10    X(I) = (B(I) - S) / A(I,I)
```

уже немає необхідності виконувати розподіл циклів, досить лише внутрішній цикл (по мітці 20) виконувати не в межах від i + 1 до n, як у вихідному циклі, а в протилежному напрямку від n до i + 1 з кроком –1.

Зі сказаного вище можна зробити цілком очевидний висновок – *у залежності від структури зв'язків між операціями той самий алгоритм може бути представлений різними способами у вигляді сукупності ансамблів*. Зокрема, звичайна послідовна реалізація означає, що в кожному ансамблі міститься тільки одна операція. Для більшості алгоритмів навіть таких представлень сукупностей ансамблів може існувати дуже багато. Ясно, що для кожної задачі особливий інтерес становить знаходження алгоритмів мінімальної висоти. Відповідно до теорії послідовних алгоритмів представлення того самого алгоритму різними ансамблями *необхідно розглядати як різні алгоритми*, тому що, як мінімум, змінюється порядок виконання операцій. Отже, деякі характеристики цих різних алгоритмів опиняються свідомо різними, але якісь напевно збережуться.

Різні програми, для створення яких авторами часто використовується той самий алгоритм, насправді майже завжди описують різні алгоритми, хоча й математично еквівалентні. А на практиці це призводить до різних результатів.

Задача побудови швидких паралельних алгоритмів повинна бути математично коректною. Для задоволення цієї вимоги зазвичай робляться деякі припущення, наслідком яких стає спрощення властивостей паралельної обчислювальної системи. Це такі припущення:

– система має нескінченно багато паралельно працюючих процесорів;
– всі вони працюють синхронно під загальним керуванням і виконують будь-яку операцію точно й за однаковий час;
– система має нескінченно велику пам'ять;
– всі обміни інформацією між процесорами й пам'яттю, а також між самими процесорами здійснюються миттєво й без конфліктів.

Концепція побудови алгоритмів для подібних паралельних систем одер-

жала назву концепції ***необмеженого паралелізму***. Звичайно, така концепція повністю ідеалізована, але дозволяє одержати цікаві результати.

Як приклад можна розглянути процес підсумовування n чисел, який доволі часто зустрічається у програмах, коли на кожному кроці до часткової суми s додається черговий доданок a_i:

```
for(i =0; i < n; ++i) s = s + a[i];.
```

Цей алгоритм має тільки одну паралельну форму, у кожному ансамблі якої є лише одна операція (див. рис. 1.22 а). Отже, ніякої можливості використати паралелізм у ньому немає.

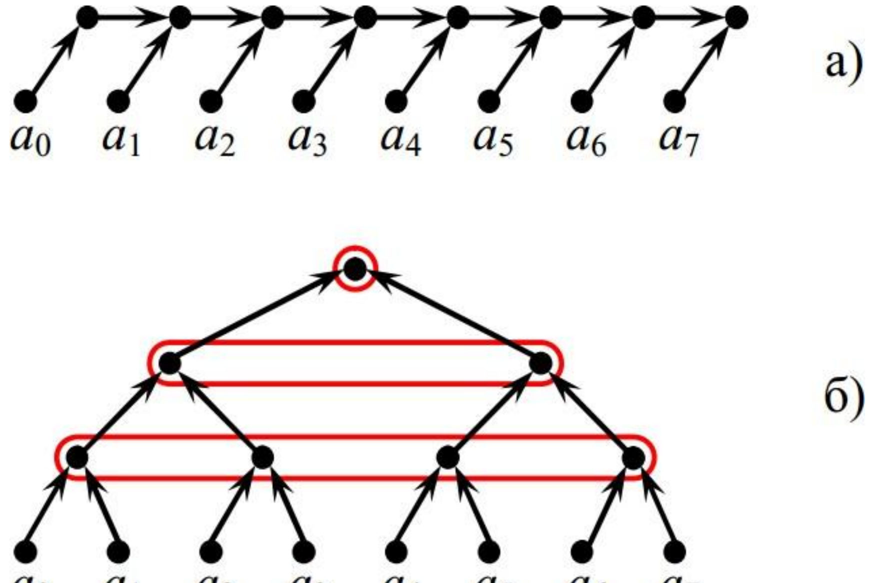


Рисунок 1.22 − Графи алгоритмів додавання послідовності чисел
а) інформаційна історія, б) алгоритм здвоювання (ЯПФ)

Оскільки операція підсумовування великої кількості доданків є дуже розповсюдженою, то в припущенні, що припустимо знехтувати помилками округлення, спричиненими різним порядком підсумовування елементів масиву a, був придуманий інший спосіб підсумовування, який має набагато кращий паралелізм – це широко відома ***схема здвоювання***. Всі доданки розбиваються на пари, і спочатку на декількох процесорах в паралельному режимі здійснюється під-

сумовування таких пар чисел (рис. 1.22 б). Всі ці операції незалежні. Операція розбивки й підсумовування пар знову повторюється, тепер уже для отриманих часткових сум. І знову всі операції незалежні одна від одної. Вся сума буде отримана через $\log_2 n$ кроків. Це й буде висота нового алгоритму. У ньому вже є значний ресурс паралелізму, хоча він не рівномірний. На першому ярусі паралельної форми програми можуть бути використані $n/2$ процесорів, на другому $n/4$ і т.д. Останнє підсумовування виконує лише один процесор.

Цілком очевидно, що обидва алгоритми основані на реалізації *математично еквівалентних виразів* підсумовування чисел, але вони мають різні властивості, принаймні, з погляду паралельних обчислень. Насправді, у них багато й інших розходжень: вони по-різному реагують на помилки округлення, по-різному використовують пам’ять і т.п. Тому ці алгоритми доцільно вважати *принципово різними*, незважаючи на те, що *вони математично еквівалентні*!

Розглянутий приклад демонструє ні що інше, як алгоритмічне представлення звичайного виразу:

$$s = a_0 + a_1 + a_2 + \dots + a_n ,$$

але цей вираз є просто окремим випадком виразів присвоювання, які у величезній кількості зустрічаються в будь-якій програмі й у будь-якому алгоритмі й також мають достатній ресурс паралелізму.

Основою побудови паралельного алгоритму для арифметичних виразів може слугувати відповідний послідовний алгоритм і сама по собі розв’язувана задача. Найбільш розумний підхід – це виявлення в послідовних алгоритмах елементів, які часто зустрічаються, і перетворення їх на паралельну форму. Очевидно, що хоча б частина цих перетворень може виконуватись автоматично. Це дуже важливо, тому що дозволяє не використовувати на мультипроцесорних машинах дороге програмне забезпечення.

Перш ніж розглядати основні підходи для розпаралелювання арифметичних виразів, необхідно навести кілька визначень.

1. Арифметичний вираз називається ***альтерніровани***м, якщо він має ви-

гляд:

$$E_n(a_1,\ldots,a_n) = E_n(\ldots(a_1\ q_1\ a_2)\ q_2\ a_3)\ldots a_{n-1}\ q_{n-1}\ a_n),$$

де q_i позначає одну з операцій {+, -, *, /, \}. Операція {\} визначає операцію зворотного ділення, тобто $a \setminus b = b / a$. При цьому, якщо у виразі E_n q_1 – операція додавання або вирахування, то альтернірований вираз називається ***адитивним***, у противному випадку – ***мультиплікативним***.

2. Вираз

$$E = (\ldots(a_1 a_2 + a_3)a_4 + a_5)a_6 + \ldots)a_{2n} + a_{2n+1}$$

називається ***узагальненою схемою Горнера***.

3. Два арифметичних вирази $\tilde{E}$ й E, називаються ЕКВІВАЛЕНТНИМИ, якщо $\tilde{E}$ можна одержати з E використовуючи кінцеве число раз закони асоціативності, комутативності й дистрибутивності.

Всі алгоритми паралельного обчислення арифметичних виразів базуються на побудові еквівалентних виразів, і ґрунтуються на теоремі, яка стверджує, що будь-який альтернірований вираз E можна розкласти на арифметичні вирази A, B і C такі, що:

$$E = A * a_1 + B \text{ або } E = B / (a_1 + A) + C$$

Математичним апаратом для формування паралельного способу обчислень арифметичних виразів, так само як і для фрагментів програм, є побудова орієнтованих ациклічних графів, які описують спосіб обчислення виразів. Відповідно до теореми для перетворення таких графів можуть бути використані необхідне число раз закони асоціативності, комутативності й дистрибутивності.

Таким чином, побудувавши граф для послідовного обчислення арифметичного виразу, можна за допомогою формальних правил побудувати граф для

паралельного обчислення цього виразу.

Наприклад, для виразу:

$$E = (a_1 * a_2 * a_4 * a_6 + a_3 * a_4 * a_6) + (a_5 * a_6 * a_7). \quad (1.1)$$

Ациклічний граф алгоритму для обчислення цього виразу на однопроцесорному комп’ютері має висоту вісім і ширину ярусу – один. В той самий час при обчисленні на багатопроцесорному комп’ютері граф має висоту чотири, а ширину канонічної ЯПФ – три. Обидва ці графи представлені на рис. 1.23 а), і рис. 1.23 б) відповідно.

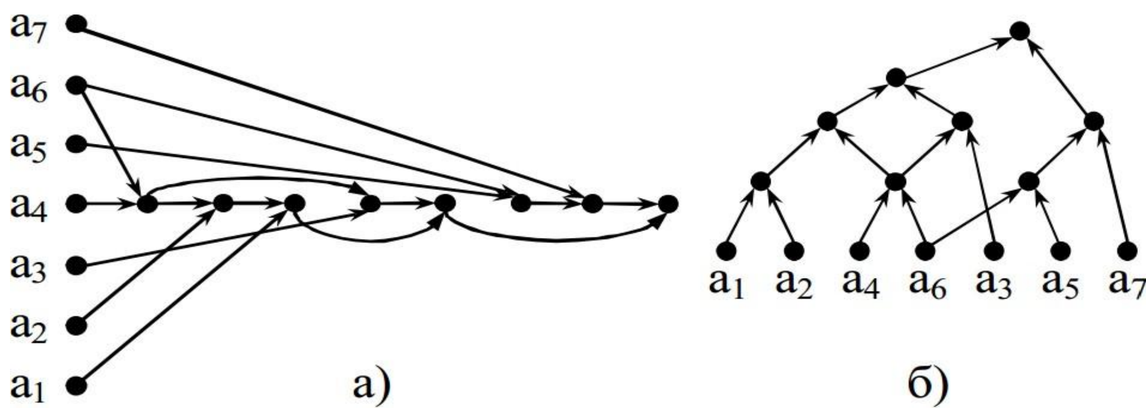


Рисунок 1.23 − Ациклічні графи для розрахунку арифметичного виразу (1.1)
а) інформаційна історія, б) канонічна ЯПФ

Існує багато алгоритмів, які дозволяють виконувати ці перетворення автоматично. Найбільш відомим з них є ***алгоритм Винограду***. Суть його полягає в такому.

Нехай E – арифметичний вираз, до якого кожна змінна входить тільки один раз. На першому кроці алгоритму будується адитивний або мультиплікативний альтернірований вираз E' еквівалентний E:

$$E' = (\dots(F_1\ q_1\ F_2)\ q_2\dots)\ q_{k-1}\ F_k,$$

де, F_i – підвираз E.

Потім виконується вибір змінних, які входять до підвиразу F_i, і його роз-

кладання ще на два підвирази G і H такі, що $F_i = G\ q_i\ H$, $q_i \in \{+, -, *, /, \backslash\}$. Розкладання здійснюється залежно від значення індексу k. При цьому розрізняються два випадки:

- при $k = 1$ (тобто $E' = F_1 = G\ q_1\ H$), рекурсивно із застосуванням цієї процедури обчислюється G і H й у результаті знаходиться остаточне представлення для $E' = G\ q_1\ H$;
- при $k > 1$, обчислюється $G, H, F_1, \ldots, F_k$ і застосовується таж сама процедура до F_2 і т.д.

Однак іноді можуть виникнути труднощі, пов’язані з діленням на нуль.

За допомогою таких перетворень множина арифметичних виразів зводиться до узагальненої схеми Горнера. Обчислення таких виразів – це сугубо послідовний процес, який досить легко розпаралелюється. Приклад розпаралелювання виразу

$$E = ((a_1 * a_2 + a_3) * a_4 + a_5) * a_6 + a_7 \qquad (1.2)$$

наведений на рис. 1.24.

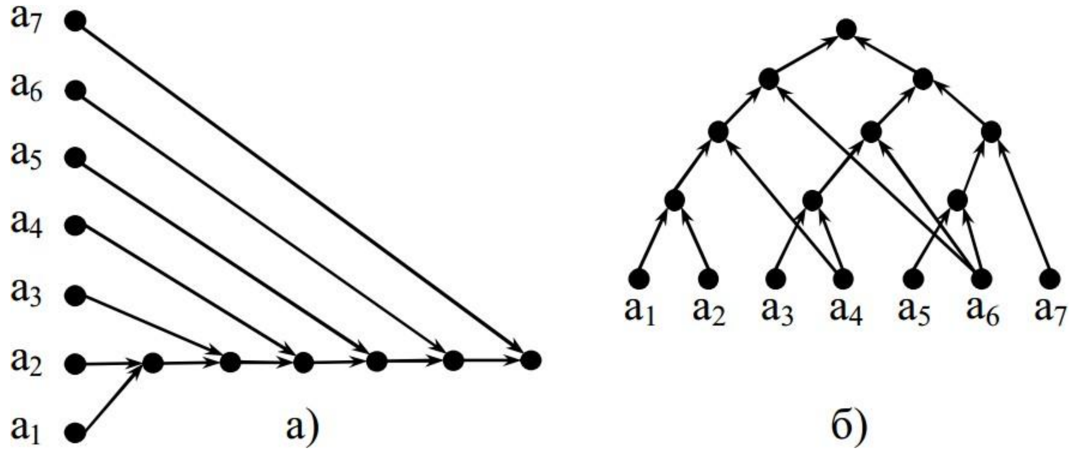


Рисунок 1.24 − Ациклічні графи для розрахунку арифметичного виразу (1.2) за схемою Горнера а) інформаційна історія, б) канонічна ЯПФ

Більшість алгоритмів даного типу функціонують у припущенні, що число процесорів необмежено (***необмежений паралелізм***). *Лема Брента* дозволяє ві-

дійти від математичної абстракції необмеженого паралелізму. Лема зв'язує час t виконання деяких обчислень на необмеженому числі процесорів з часом t' виконання тих самих обчислень на числі процесорів, яке дорівнює p. Відповідно до цієї леми, якщо на необмеженій кількості процесорів W операцій можуть бути виконані за t одиниць часу, то при наявності p процесорів час розрахунків t' буде дорівнювати:

$$t' = t + (W - t) / p.$$

1.3 Питання і завдання для самоперевірки

1) Чим відрізняються паралельні і розподілені обчислення, що в них є спільного?
2) Які два типи паралелізму Ви знаєте? Назвіть і охарактеризуйте кожен з них.
3) Якими технологіями реалізується паралелізм? У чому полягає відмінність між двома технологіями паралелізму?
4) Які основні особливості притаманні підходу основаному на паралелізмі даних?
5) На використанні якого базового набору операцій базується паралелізм даних в програмах?
6) Якими механізмами забезпечується підтримка реалізації моделі паралелізму даних на рівні транслятора?
7) Яка головна властивість притаманна і є дуже характерною для стиля програмування, основаного на паралелізмі задач?
8) Дійсно або ні, те, що парадигма паралелізму задач є більш трудомісткою для програмістів у порівнянні з підходом, основаним на паралелізмі даних? Якщо так, то докладніше обґрунтуйте власну відповідь.
9) Які дві основні проблеми повинні бути вирішені при складанні паралельних алгоритмів роботи програми, як при застосуванні паралелізму даних, так і при використанні паралелізму задач?

10) Опишіть такі два різновиди розпаралелювання обчислювальних процесів, як конвеєрність і паралельність. Чим вони проміж собою розрізняються і що в них є спільного?
11) Опишіть і поясніть загальну схему розробки паралельних програм (алгоритмів). Які дії можуть бути визначені як масштабування розроблювального алгоритму і чим вони викликаються?
12) Назвіть деякі приклади з безлічі сучасних мов і систем програмування, розроблених для створення програм, як для паралельних, так і розподілених обчислювальних систем?
13) Які дві системи бібліотечних функцій найчастіше використовуються для паралельного програмування, як розподілених, так і паралельних обчислювальних систем? Надайте ним коротку порівняльну характеристику.
14) Докладно опишіть структуру і властивості пакета програмного забезпечення Паралельної віртуальної машини: яка парадигма взаємодії процесів підтримується бібліотекою, які моделі програм можуть створюватись на базі бібліотеки, як запускаються і управляються процеси в програмі і т.д.
15) Докладно охарактеризуйте бібліотеку підпрограм, яка дістала назву «Інтерфейс передачі повідомлень». Розкажіть, які моделі програмування підтримуються в бібліотеці, поясніть поняття «комунікатор», які особливості притаманні різним стандартам бібліотеки, які реалізації MPI бібліотек Вам відомі.
16) Що означає така важлива властивість паралельних програми, як її детермінізм? Чому ні в якому разі не можна відмовитися від цього?
17) Яке поняття уводиться для формалізації задачі знаходження в програмі достатньої кількості незалежних одна від одної груп операцій, рівномірного розподілу їх між обчислювальними пристроями і полегшення її аналізу з метою розпаралелювання?
18) Чому поняття графа програми не досить чітке і чому частіше говорять про більш абстрактне представлення програми й широко використовують модель у вигляді графа алгоритму?

19) Що таке спрямований граф алгоритму, як він визначається, в чому полягає його ациклічність?

20) Назвіть структури, які можуть виступати в якості вершин і дуг ациклічного спрямованого графа?

21) Що означають поняття «ітерації циклів» і «спрацьовування операторів»? Поясніть їх відповідними прикладами графів і фрагментами кодів.

22) Наведіть визначення таких дуг графів, як «операційне відношення». Як визначається «інформаційне відношення»?

23) Які операції програми називаються інформаційно залежними? До чого на практиці зводиться задача розпаралелювання програми?

24) Як введення поняття «графів інформаційних залежностей» і їхнє застосування у практиці розпаралелювання програм дозволяє формалізувати цю задачу і полегшити аналіз програм?

25) Який граф називається мінімальним графом залежностей?

26) Назвіть чотири різних основних моделі представлення програм у вигляді графів. Чим моделі типу історії принципово відрізняються від інших моделей? Проілюструйте моделі відповідними графами і фрагментом коду.

27) Побудуйте всі чотири моделі представлення програм для фрагменту програми, наведеного нижче:

```
a[0] = 2 * x[8] - 3;
c = a[0] * a[0] + 1;
b[0] = c - a[0] * D;
for(i = 1; i < n; ++i)
  {
    a[i] = a[i - 1] + 2;
    b[i] = b[i] + a[i];
  }
```

28) Які дії є найбільш важливими при аналізі структури програми з метою ефективнішого розв’язання задачі її розпаралелювання?

29) Які з моделей програм інформаційні або операційні є основою аналізу властивостей програм і алгоритмів? Поясніть, чому це так.

30) Що на Ваш погляд обумовлюють інформаційні залежності в графах програм? А що, в свою чергу, обумовлюють інформаційні незалежності в

програмах? Поясніть і проілюструйте свої відповіді графами.

31) Якими трьома критеріями користуються розробники при виборі моделі для опису властивостей програми? Пояснить, у яких випадках і чому доцільно застосовувати той чи інший критерій. Наведіть графи, які ілюструють Ваші відповіді.

32) Може або ні інформаційна історія деякого фрагмента програми мати 100 вершин і жодної дуги? Якщо так, то нарисуйте її граф і проілюструйте такий фрагмент відповідним кодом.

33) Чи може інформаційна історія деякого фрагмента програми мати 67 вершин и 3 дуги? Проілюструйте власну відповідь відповідним графом історії і кодом такого фрагмента.

34) Чи можете Ви уявити собі фрагмент програми, інформаційна історія якого має 20 вершин і 200 дуг? Якщо такий фрагмент неможливий, то поясніть чому, в іншому разі проілюструйте відповідь графом історії і кодом.

35) Якій моделі, або яким моделям програми: графу керування, інформаційному графу, операційній історії, інформаційній історії, може відповідати такий граф:

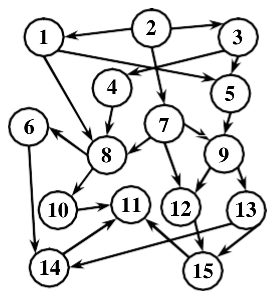
36) Сформулюйте, чим фактично є граф алгоритму програми. Що і чому він забезпечує при аналізі властивостей програм з метою їхнього розпаралелювання?

37) Що таке висота алгоритму, які алгоритми називаються паралельними і що, в решті решт, обумовлює паралельну форму алгоритму?

38) У який спосіб з операцій інформаційної історії, утворюється так звана ярусно-паралельна форма (ЯПФ) програми, яка основна властивість операцій історії, що розташовуються на однім ярусі ЯПФ?

39) Що таке канонічна ЯПФ, як визначається і що обумовлює її критичний шлях? Надайте визначення ширини і висоти алгоритму.

40) Нарисуйте канонічну ярусно-паралельну форму графа, зображеного нижче, визначте всі основні характеристики одержаного графа (ширини ярусів, ширину і висоту графа, критичний шлях).



41) Які типи паралелізму Ви знаєте? Коротко охарактеризуйте кожен з них. Наведіть приклади коду або графів, які їх ілюструють.

42) Відрізками коду і відповідними графами інформаційних або операційних історій проілюструйте такі розподілення ітерацій циклів: великоблочне, блочно-циклічне, циклічне.

43) Яке припущення робиться при розподілі ітерацій циклів з метою досягнення рівномірності завантаження процесорів? Чи є це припущення справедливим в реальних випадках? Свою відповідь «так» або «ні» поясніть докладніше.

44) Чому при розпаралелюванні коду найважливішим завданням є мінімізація необхідної кількості й обсягу пересилань даних? Відповідь повинна бути проілюстрована відповідними графами і кодом.

45) Чому вводиться поняття простору ітерацій? Надайте визначення цього поняття. Які різновиди просторів ітерацій Ви знаєте? Охарактеризуйте кожен з них.

46) Для чого іноді потребується виконання еквівалентних перетворень програми, що це саме таке і які види перетворень коду програми Ви знаєте? Наведіть приклади графів і коду таких перетворень. У разі перетворення програми одержуються однакові чи різні алгоритми?

47) Які припущення робляться відносно паралельних обчислювальних систем в межах концепції необмеженого паралелізму, для чого це потрібно і який виграш дає?

48) Надайте визначення для альтернірованого арифметичного виразу, адитивного і мультиплікативного альтернірованих арифметичних виразів, виразу узагальненої схеми Горнера і еквівалентних арифметичних виразів.

49) Сформулюйте теорему, на якій будуються всі алгоритми паралельного обчислення еквівалентних арифметичних виразів. Наведіть приклади застосування цієї теореми.
50) Поясніть, в чому полягає суть алгоритму Винограду для автоматичного еквівалентного перетворення арифметичних виразів. Проілюструйте цей алгоритм на прикладі деякої узагальненої схеми Горнера.
51) Що стверджує лема Брента і як за її допомогою можна зв’язати час виконання програми на необмеженому числі процесорів з часом виконання обчислень на обмеженому числі?

2 БАЗОВІ АЛГОРИТМИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ

Багатократно виконуючи побудову графів для великої кількості конкретних алгоритмів, фахівці виявили дивну закономірність: *велика розмаїтість існуючих методів не призводить до такої ж розмаїтості їхніх інформаційних структур*. Точніше кажучи, багато графів формально зовсім різних алгоритмів виявилися ізоморфними, відрізняючись один від одного тільки змістом вершин і дуг. Тому була висунута гіпотеза про те, що в конкретних обчислювальних галузях існує дуже небагато типових інформаційних структур. Не дивлячись на доволі великій вік цієї гіпотези, практика поки що повністю підтверджує її. Наприклад, на всій множині алгоритмів лінійної алгебри типових інформаційних структур виявилося всього лише близько десятка. Далі будуть розглянуті деякі з них.

2.1 Алгоритми паралельного обчислення рекурентних співвідношень

Багато чисельних алгоритмів являють собою обчислення по ***рекурентним формулам***, при цьому обчислюється послідовність змінних $x_1, \dots x_n$, у яких x_i можуть залежати від всіх x_j ($j < i$). Співвідношення, що описують ці послідовності, називають ***рекурентними***. Рекурентне співвідношення має *порядок* k, якщо воно дозволяє виразити чергове x_{i+k} через попередні x_i, x_{i+1}, ..., x_{i+k-1}... Якщо при цьому для x_k задане початкове значення x_0, то говорять про ***рекурентну задачу***. При цьому якщо x_n є скаляром, то рекурентна задача – це ***задача зі скалярним результатом***, якщо ж x є вектором, то – це ***задача з векторним результатом***. ***Лінійною рекурентною задачею*** називають задачу, яка описується таким чином:

$$x_k = \begin{cases} 0, & k \le 0 \\ c_k + \sum_{j=1}^{k-1} a_{kj} x_j, & k = 1 \dots n \end{cases} \qquad (2.1)$$

Наприклад, відшукання скалярного добутку двох векторів (матриць-рядків): $X = [x_1, \ldots, x_n]$, $Y = [y_1, \ldots, y_n]$ зводиться до лінійної рекурентної задачі першого порядку зі скалярним результатом. При цьому початкове значення добутку z_0 покладається так, щоб воно дорівнювалось 0, а кожне таке значення обчислюється як $z_k = z_{k-1} + x_k y_k$, де $k = 1, \ldots, n, \ldots$

Цілком очевидно, що обчислення багаточлена порядку n: $P_n(x) = \sum_{i=1}^{n} a_i x^i$ за розглянутою віще схемою Горнера при $x = x_0$ також призводить до лінійної рекурентної задачі зі скалярним результатом вигляду: $P_k = a_{n-k} + x_0 P_{k-1}$, де початкове наближення P_0 дорівнює a_n, а k змінюється від 1 до n.

Якщо у виразі (2.1) покласти $A=\{a_{ik}\}$, i, $k = 1,\ldots, n$ – строго нижньотрикутна матриця, а x і c – це вектори $(x_1, \ldots, x_n)$ і $(c_1, \ldots, c_n)$ відповідно, то для паралельного розв'язання таких задач застосовуються такі три алгоритми.

2.1.1 Алгоритм викреслювання стовпців

Нехай необхідно вирішити таку задачу:

$$\begin{aligned} x_1 &= c_1 \\ x_2 &= c_2 + a_{21}x_1 \\ x_3 &= c_3 + a_{31}x_1 + a_{32}x_2 \\ x_4 &= c_4 + a_{41}x_1 + a_{42}x_2 + a_{43}x_3 \end{aligned} \quad (2.2)$$

З системи (2.2) неважко помітити, що якщо відомо x_1, то всі вирази

$$c_i^1 = a_{i1}x_1 + c_i, \qquad i = 2,\ldots,n, \quad (2.3)$$

де, n – вимірність вектора x, можуть бути обчислені паралельно, тобто незалежно один від одного.

В свою чергу, якщо відомо x_1 і x_2, то вирази

$$c_i^2 = a_{i2}x_2 + c_i^1, \qquad i = 3,\ldots,n \quad (2.4)$$

також можуть обчислюватись паралельно.

Вважаючи, що одна одиниця часу еквівалентна часу виконання однієї операції незалежно від її типу, можна визначити коефіцієнт прискорення $K_{\text{пр}}$ для цього алгоритму. Зрозуміло, що для оцінки $K_{\text{пр}}$ досить лише оцінити кількість неодночасно виконуваних операцій у послідовному й у паралельному варіантах алгоритму викреслювання стовпців. При такому оцінюванні можна не враховувати час, необхідний для встановлення зв'язків і обміну даними між процесорами під час розрахунків.

Для послідовного алгоритму загальне число операцій дорівнює:

$$N_1 = \sum_{j=1}^{n-1} 2j = 2(0.5(n-1)n) = n^2 - n. \qquad (2.5)$$

Для паралельного варіанта алгоритму викреслювання стовпців, кожному з p процесорів системи найпростіше надати масив даних довжиною $q = \lceil n / p \rceil$ (блоковий розподіл даних), і вважати, що кожен процесор виконує обчислення тільки в межах цих даних. Неважко помітити, що при такій організації обчислень, кількість неодночасно виконуваних операцій для оцінки величин (2.3) або (2.4) максимум буде дорівнювати $2q$. Після того як всіма процесорами були зроблені всі необхідні операції, процесори обмінюються один з одним отриманими результатами. Таким чином, для такого варіанта алгоритму викреслювання стовпців загальна кількість неодночасно виконуваних операцій, яка необхідна для розв'язання поставленої задачі, обумовлюється співвідношенням:

$$N_p = 2(n-1)n / p = \frac{2}{p}(n^2 - n).$$

Остаточно звідси витікає:

$$K_{\text{пр}} = N_1 / N_p = p / 2.$$

Легко помітити, що отримане значення $K_{\text{пр}}$ у два рази менше за ідеальне.

Це відбувається внаслідок так званого ***ефекту Гайдна***. Суть його полягає в тому, що після виконання перших q кроків яку-небудь корисну роботу перестає виконувати перший процесор, після виконання $2q$ кроків зупиняється другий процесор, і, нарешті, після виконання $(q - 1)$-го кроку в роботі залишається лише один процесор. Зменшення $K_{пр}$, що є наслідком ефекту Гайдна, часто оцінюють ***коефіцієнтом Гайдна***. Таким чином у розглянутому випадку коефіцієнт Гайдна дорівнює $1/2$.

А чи можна так побудувати алгоритм викреслювання стовпців, щоб $K_{пр}$ був близький до p? Так, цього можна домогтися, якщо організувати процес обчислень так, як показано на рис. 2.1.

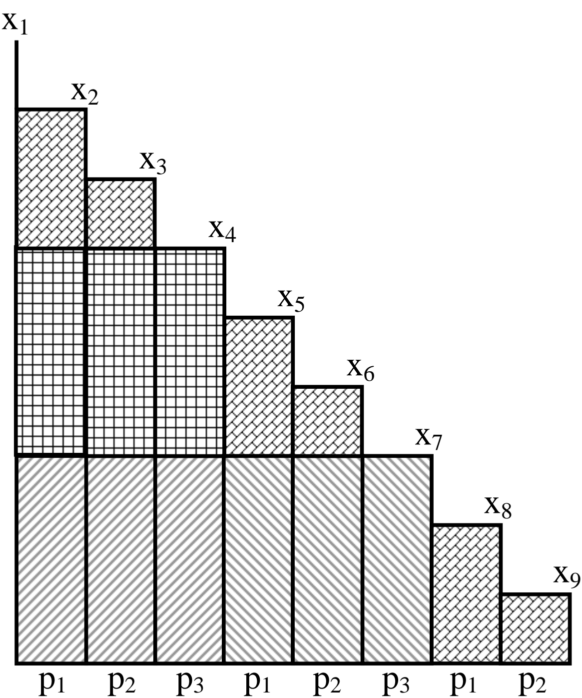


Рисунок 2.1 − Схема організації обчислень при нейтралізації ефекту Гайдна для алгоритму викреслювання стовпців при розв’язанні рекурентної задачі (2.2)

В цьому разі всі дані вже блочно-циклічним способом розподілу розрізаються на $h = \lceil n / p \rceil$ смуг, кожна з яких шириною p.

Елементи, які позначені прямокутниками з таким типом заливки, можуть бути обчислені лише послідовно, а елементи з заливкою, і

обчислюються на p процесорах у паралельному режимі (на рисунку p = 3). Тоді алгоритм можна побудувати у такий спосіб.

Для всіх $i=1(1)h$ виконується така послідовність кроків.

1) Послідовний крок:

$$m = (i-1)p+1,$$
$$x_j = c_j + \sum_{k=m}^{j-1} a_{jk} x_k, \qquad ,$$
$$j = m(1),\ m+p-1 \le n$$

де $m = ip + 1$.

2) Паралельний крок:

$$c_j = c_j + \sum_{k=1}^{(i+1)p-1} a_{jk} x_k,$$

де $j = m\ (1),\ m+p-1$.

Якщо ж провести підрахунок кількості неодночасно виконуваних операцій, необхідних для роботи алгоритму, то для послідовних кроків їх кількість $S_{пос}$ дорівнює:

$$S_{пос} = h(1+2+K + (p-1))2 = h\frac{p(p-1)}{2}2 = \frac{np(p-1)}{p} = n(p-1).$$

Відповідно для паралельних кроків ця кількість буде:

$$S_{пар} = ((p-1) + 2(p-1) + K + (h-1)(p-1))2 = \frac{(p-1)h(h-1)}{2}2 \approx \left(\frac{n^2}{p} - n\right)$$

Таким чином сумарна кількість операцій:

$$N_p = \frac{n^2}{p} - n + n(p-1) = \frac{n^2}{p} + n(p-2).$$

Тоді з урахуванням (2.5), а також того факту, що n ? p (у противному

випадку немає сенсу робити паралельні обчислення), можна одержати коефіцієнт прискорення, який буде:

$$K_{ПР} = \frac{N_1}{N_p} = \frac{n^2 - n}{\frac{n^2}{p} + n(p-2)} \approx p .$$

2.1.2 Алгоритм логарифмічного підсумовування (алгоритм здвоювання)

Цей алгоритм дозволяє ефективно вирішувати асоціативні рекурентні задачі. Розглянути ***алгоритм логарифмічного підсумовування*** можна на прикладі простої лінійної рекурентної задачі, яка вже власне розглядалась віще (див. стор. 44) – задача визначення суми n чисел.

Задача описується простим рекурентним виразом:

$$\begin{aligned} x_0 &= 0, \\ x_k &= a_k + x_{k-1}, \quad k = \overline{1,n}. \end{aligned} \qquad (2.6)$$

Ациклічні графи інформаційної історії послідовного алгоритму підсумовування послідовності чисел і ярусно-паралельна форма алгоритму логарифмічного підсумовування (здвоювання) наведені вище на рис. 1.22 а) і б) відповідно.

При оцінці найбільшої швидкодії алгоритму логарифмічного підсумовування слід вважати, що на кожному кроці в обчислювальних діях бере участь максимально можлива кількість процесорів.

Нехай розглянута задача призводить до результату в деякій момент часу і для її розв’язання при цьому був потрібен $r + 1$ крок. Тоді максимально можлива кількість корисних операцій m, яку можна виконати на цій кількості кроків, буде дорівнювати:

$$m = 1 + 2^1 + 2^2 + \ldots + 2^r = 2^{(r+1)-1}.$$

Звідки, кількість необхідних для розрахунків одиниць часу, за умови, що для виконання однієї операції потребується одиничний проміжок часу, дорівнює:

$$t_p = r + 1 = \lceil \log_2(m + 1) \rceil .$$

Якщо врахувати, що реально має місце ситуація, коли $n - 1 >= m$ і $p >= 2^r$, то не важко бачити, що $t_p = r + 1 = \lceil \log_2(n) \rceil$.

У випадку ж, коли існують такі співвідношення $2^r <= p < 2^{(r+1)}$ і

$$m > 2^{(r+1)} + 1, \tag{2.7}$$

то на кожному з k перших кроків аж до деякого кроку $R - (r + 1)$, де R – повна потрібна кількість кроків, можуть працювати тільки p процесорів. Тоді для кількості корисних операцій m буде отримана нерівність $m \geq 2r + 1 - 1 + pk$, звідки $k \geq \lceil (m + 1 - 2^{(r+1)}) / p \rceil$.

Тоді повний час на виконання підсумовування виразиться як:

$$t_p = r + 1 + k \geq r + 1 + \lceil (m + 1 - 2^{(r+1)})/p \rceil .$$

Якщо врахувати нерівність, яка витікає з (2.7), тобто $r + 1 \geq \lceil \log_2 p \rceil$, то остаточно для t_p виходить:

$$t_p \geq \lceil \log_2 p \rceil + \lceil (m + 1 - 2^{(r+1)}) / p \rceil$$

Якщо тепер покласти, що кількість процесів є деяким ступенем двійки, тобто $p = 2^v$, тоді $2^{\log_2 p} = p$ і $\lceil \log_2 p \rceil = \log_2 p$, то t_p буде:

$$t_p \geq \begin{cases} \log_2 p + (m + 1 - p)/p, & \text{при } m \geq p, \\ \lceil \log_2 (m + 1) \rceil, & \text{при } m < p. \end{cases}$$

І насамкінець, з урахуванням того, що $n - 1 = m$, для коефіцієнта прискорення виходить:

$$K_{ПР} \le \begin{cases} \dfrac{(n-1)p}{p\log_2 p + (n-p)}, & \text{якщо} \quad n-1 \ge p, \\ \dfrac{n-1}{\log_2 n}, & \text{якщо} \quad n-1 > p. \end{cases}$$

2.1.3 Алгоритм рекурентного добутку

У цьому алгоритмі використовується той факт, що лінійну рекурентну задачу можна звести до розв’язання системи рівнянь вигляду:

$$x = Ax + c, \tag{2.8}$$

де x і c – вектори вимірністю n, а A – строго нижньотрикутна матриця. Тоді з (2.8) витікає, що:

$$x = (E - A) - Ec = L - Ec. \tag{2.9}$$

У свою чергу в (2.9) $L - E = M_n * M_{n-1} * \ldots * M_1$, де:

$$M_i = \begin{bmatrix} 1 & & & & & \\ & \Lambda & & & & \\ & & 1 & 0 & & \\ & & -a_{i+1,i} & \Lambda & & \\ & & & & \Lambda & \\ & & & & -a_{n,i} & 1 \end{bmatrix}.$$

Безсумнівно, що добуток $M_i c$ можна виконати за допомогою процедури логарифмічного підсумовування. А це означає, що для алгоритму рекурентного добутку коефіцієнт прискорення виражений через ***оцінку складності алгоритму***, буде:

$$K_{ПР} = O(\log_2 n),$$

де запис $y = O(x)$, означає буквально таке: існує така позитивна константа α, що для всіх x, більших від деякого доволі великого значення x_0, виконується нерівність $y \le \alpha x$.

2.1.4 Блоковий алгоритм

У багатьох чисельних процедурах часто зустрічається необхідність оцінювання на k-му кроці ітерації деякого функціонала F:

$$F_k(x_1, \ldots, x_n) = (F_k^{(1)}, \ldots, F_k^{(m)}),$$

де всі $F_k^{(1)}, \ldots, F_k^{(m)}$ оцінюються на основі інформації про F_i, $i = 1, \ldots, k$.

Наприклад оцінка подібного функціонала необхідна при розв'язанні системи рівнянь $F(x) = 0$ методом Ньютона:

$$x_k = x_{k-1} + \alpha_k J_{k-1}^{-1},$$

де:

$$\alpha_k = \arg\min_{\alpha}\left(F\left(x_{k-1} + \alpha J_{k-1}^{-1}\right)\right) \qquad (2.10)$$

Ітерації за формулою (2.10) припиняються, коли справедливою стає така умова:

$$\frac{\left|\left(F\left(x_{k-1} + \alpha^{(m)} J_{k-1}^{-1}\right)\right) + \left(F\left(x_{k-1} + \alpha^{(m-1)} J_{k-1}^{-1}\right)\right)\right|}{2\left|\left(F\left(x_{k-1} + \alpha^{(m)} J_{k-1}^{-1}\right)\right)\right|} \le \epsilon,$$

де ϵ – деяка наперед визначена точність.

Ще одним прикладом проведення подібного оцінювання може бути розв'язання задачі пошуку екстремуму деякої функції, тобто:

$$\min_x \Phi(x),$$

яка, наприклад для методів 1-го й 2-го порядків, зводиться до пошуку точок:

$$x^{(k)} = x^{(k-1)} + \alpha^{(k)} S^{(k)}, \qquad (2.11)$$

де $S(k)$ – напрямок пошуку, який для градієнтних методів визначається як:

$$S^{(k)} = -\Phi'^{(k)},$$

а для квазіньютонівських методів як:

$$S^{(k)} = -H^{-1(k)} \Phi'^{(k)}.$$

В цьому разі для одержання виразу (2.11) необхідно вирішити задачу:

$$\alpha^{(k)} = \arg \min_{\alpha} \left(\Phi \left(x^{(k-1)} + \alpha S^{(k)} \right) \right) \qquad (2.12)$$

Більшість відомих методів розв’язання задач (2.10), (2.12) зводяться до оцінки значень $\Phi\left(x^{(k-1)} + \alpha S^{(k)}\right)$ при різних значеннях α, як правило, з інтервалу [0, 1]. Зрозуміло, що оцінку цих значень можна робити паралельно.

Ще наочніше виграшність такого підходу ілюструється на задачах статистики, більшість яких зводиться до обчислення послідовності з n значень деякої функції $\Phi(x)$, при значеннях x, обираних з діапазону $[x_{min}, x_{max}]$ випадковим чином.

Далі в цьому розділі будуть розглянуті декілька прикладів того, як можна застосувати прийоми, досліджені вище, для розпаралелювання відомих і таких, що вже добре зарекомендували себе, чисельних алгоритмів обчислювальної математики. Програмні приклади паралельних алгоритмів будуть виконуватись із застосуванням функції бібліотеки MPI.

2.2 Паралельні методи розв’язання СЛАР

При моделюванні різноманітних фізичних процесів, технічних розробках, суспільних моделях дуже часто використовуються методи обчислювальної ал-

гебри і однією з основних задач таких обчислень є розв’язання систем лінійних алгебраїчних рівнянь (СЛАР):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{aligned}$$

Для розв’язання системи з матрицями високої розмірності використовуються ітераційні чисельні методи, які потребують меншого обсягу обчислень і пам’яті у порівнянні з прямими методами. Як завжди при використовуванні ітераційних методів задається деяке довільне початкове наближення розв’язку системи x^0 і будується послідовність $\{x^k\} \to x^*$ при $k \to \infty$, де x^* – точний розв’язок системи, k – номер ітерації. У дійсності ітераційний процес припиняється, як тільки x^k стає досить близьким до x^*.

Нижче розглянуті два методи розв’язання СЛАР – Якобі й Гаусса-Зейделя.

2.2.1 Метод простої ітерації розв’язання СЛАР

Метод простої ітерації – метод Якобі розглянутий тому, що обчислення компонентів вектора усередині однієї ітерації повністю незалежні одне від одного й паралелізм такого методу очевидний, тому легко написати MPI програму.

У матричній формі СЛАР представляється у вигляді:

$$Ax = b; \quad x, b \in \mathbb{R}^n; \quad A \in \mathbb{R}^{n \times n}. \qquad (2.13)$$

Матрицю коефіцієнтів A можна представити як $A = A^+ + D + A^-$,

де D – діагональна матриця з діагональними членами матриці A;

A^- – частина матриці A, що лежить нижче від центральної діагоналі;

A^+ – частина матриці A, що лежить вище від центральної діагоналі.

Тоді матрична форма СЛАР набуває вигляду:

$$(A^{+} + D + A^{-})\, x = b, \qquad (2.14)$$

або $D\,x = -(A^{-} + A^{+})\,x + b$, а ітераційний процес записується як:

$$D\,x^{k+1} = -(A^{-} + A^{+})\,x^{k} + b; \qquad k = 0, 1, 2 \ldots . \qquad (2.15)$$

Алгоритм методу Якобі утворюється при поверненні до координатної форми й одержання x^{k+1} з останнього рівняння (2.15):

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^k - \sum_{j=i+1}^{n} a_{ij}x_j^k\right); \quad i = \overline{1,n}; \quad k = 1,2,\ldots \qquad (2.16)$$

У координатній формі метод Якобі – це розв'язання кожного рівняння системи відносно одного з компонентів вектора. З першого рівняння системи виражається x_1 і його значення береться за значення на такій ітерації x_1^{k+1}. Із другого рівняння визначається x_2 і його значення береться за x_2^{k+1} й так далі. Тобто змінні в правій частині цих співвідношень покладаються рівними їхнім значенням на попередній ітерації.

Ґрунтуючись на цьому алгоритмі, легко написати функцію, яка реалізує послідовний алгоритм методу простої ітерації. Для цієї функції на вході повинні бути задані:

- матриця коефіцієнтів A;
- вектор вільних членів B;
- початкове наближення вектора X;
- точність обчислень ϵ.

Критерій завершення процесу обчислень визначається як:

$$\left\|x^{k+1} - x^{k}\right\| = \max\left|x_i^{k+1} - x_i^{k}\right| < \varepsilon, \qquad 1 \le i \le n ,$$

де x^k – наближене значення розв'язку на k-му кроці чисельного методу.

Таким чином послідовна процедура обчислення нових значень компонент вектора X може виглядати, наприклад таким чином:

```
/* допоміжна функція звертання до елемента матриці A */
int ind(int i, int j, int SIZE)
  { return (i * (SIZE + 1) + j); }

/* Основна функція послідовного методу Якобі.
   На вході задається матриця A, початкове наближення
   вектора X_old, вимірність матриці size.
   Обчислюється нове значення вектора X */
void Iter_Jacoby(double *A, double *X, double *X_old,
                 int size) {
  unsigned int i, j;
  double Sum;
  for (i = 0; i < size; ++i) {
    Sum = 0;
    for (j = 0; j < i; ++j)
      Sum += A[ind(i, j, size)] * X_old[j];
    for (j = i + 1; j < size; ++j)
      Sum += A[ind(i, j, size)] * X_old[j];
    X[i] = (A[ind(i, size, size)] - Sum) /
            A[ind(i, i, size)];
  }
}
```

Таким етапом розв'язання СЛАР методом простої ітерації може бути процедура `SolveSLAE`. Ця функція у циклі, поки не досягнута необхідна точність обчислень, зберігає значення вектора, обчислене на попередній ітерації циклу, потім у процедурі `Iter_Jacoby` обчислює нове значення вектора. Остаточно визначається норма похибки. Наводити вихідний код процедури не має особливого сенсу, оскільки дії, виконувані в функції, очевидні.

Набагато цікавіше виконати розгляд паралельного алгоритму методу простої ітерації й коду програми, що його здійснює.

Зі сказаного вище очевидно, що метод простої ітерації описується такою системою рівнянь:

$$\begin{aligned} x_1^{k+1} &= f_1\left(x_1^k, x_2^k, \ldots, x_n^k\right), \\ x_2^{k+1} &= f_2\left(x_1^k, x_2^k, \ldots, x_n^k\right), \\ &\ldots \\ x_n^{k+1} &= f_n\left(x_1^k, x_2^k, \ldots, x_n^k\right). \end{aligned} \tag{2.17}$$

З наведеної системи (2.17) видно, що обчислення кожної координати век-

тора залежать лише від значень вектора на попередній ітерації й не мають інформаційних залежностей між собою. Простір ітерацій, що обумовлює ярусно-паралельну форму для розв'язання цієї системи, представлений на рис. 2.2

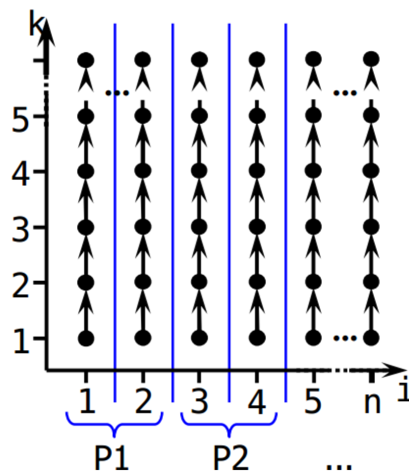


Рисунок 2.2 – Простір ітерацій для розв'язання СЛАР методом Якобі

Для паралельної реалізації методу можна запропонувати, наприклад великоблочний розподіл ітерацій циклу. Кількість координат вектора, які обчислюються в кожному процесі, в цьому разі можна визначити, наприклад, в такий спосіб:

```
size = (MATR_SIZE / numprocs) + ((MATR_SIZE % numprocs) > myid ? 1 : 0);
```

де size – кількість координат вектора, що обчислюються в даному процесі,
myid – власний номер процесу,
numprocs – кількість запущених процесів застосування,
MATR_SIZE – вимірність вектора.

Відповідно до обраного розподілу ітерацій, процедура Iter_Jacoby у паралельній версії повинна в кожному процесі обчислювати size значень вектора X, починаючи з номера first. Код процедури представлений нижче:

```
/* Основна функція паралельного методу Якобі.
   На вході задається матриця A, початкове наближення
   вектора X_old, вимірність матриці MATR_SIZE,
   кількість елементів size вектора, які підлягають обчисленню
```

```
   в даному процессе. Обчислюється нові значення вектора X
   починаючи з номера first */
void Iter_Jacoby(double *X_old, int size, int MATR_SIZE,
                int first) {
  int i, j;
  double Sum;
  for (i = 0; i < size; ++i) {
    Sum = 0;
    for (j = 0; j < i + first; ++j)
      Sum += A[ind(i, j, MATR_SIZE)] * X_old[j];
    for (j = i + 1 + first; j < MATR_SIZE; ++j)
      Sum += A[ind(i, j, MATR_SIZE)] * X_old[j];
    X[i + first] = (A[ind(i, MATR_SIZE, MATR_SIZE)] - Sum)
                  / A[ind(i, i + first, MATR_SIZE)];
  }
}
```

А код процедури SolveSLAE, що реалізує паралельний алгоритм методу простої ітерації, може виглядати наприклад так:

```
/* На вході задається матриця A, її вимірність MATR_SIZE,
   кількість елементів size, які обчислюються в даному процесі,
   похибка обчислень Error */
void SolveSLAE(int MATR_SIZE, int size, double Error) {
  double *X_old;
  int Iter = 0, i, Result, first;
  double dNorm = 0, dVal;
/* визначення номера элемента first вектора X, з якого
  будуть обчислюватися нові значення в даному процесі */
  MPI_Scan(&size, &first, 1, MPI_INT, MPI_SUM,
          MPI_COMM_WORLD);                              // 1
  first -= size;                                        // 2
/* заповнення масиву sendcounts значеннями size від кожного
   процесу*/
MPI_Allgather(&size, 1, MPI_INT, sendcounts, 1, MPI_INT,
             MPI_COMM_WORLD);
/* заповнення масиву displs - відстаней між елементами
   вектора, які розподілені по процесах */
  displs[0] = 0;
  for (i = 0; i < size; ++i)
    displs[i] = displs[i - 1] + sendcounts[i - 1];
/* виділення пам'яті для X_old */
  X_old = (double *) malloc(sizeof(double) * MATR_SIZE);
  do {
    ++Iter;
/* збереження попереднього значення вектора X */
    memcpy(X_old, X, MATR_SIZE);
/* обчислення нового значення вектора X*/
    Iter_Jacoby(X_old, size, MATR_SIZE, first);
/* розсилання вектора X всім процесам */
```

```
    MPI_Allgatherv(&X[first], size, MPI_DOUBLE, X,
                   sendcounts, displs, MPI_DOUBLE,
                   MPI_COMM_WORLD);                      // 3
/* розрахунок норми */
    if (myid == Root) {
      for (i = 1; i <= MATR_SIZE; ++i) {
        dVal = fabs(X[i] - X_old[i]);
        if (dNorm < dVal) dNorm = dVal; }
      Result = Error < dNorm; }
/* розсилання результату всім процесам */
    MPI_Bcast(&Result, 1, MPI_INT, Root,
              MPI_COMM_WORLD);                           //4
  } while(Result);
  free(X_old);
  return;
}
```

В тілі цієї функції в операторах 1 і 2 за допомогою виклику колективної функції MPI_Scan визначається номер **first** – номер елемента вектора, з якого обчислюються нові значення в кожному процесі. Потім у циклі до досягнення заданої точності обчислення виконуються такі дії. Спочатку виконується збереження значення вектора, обчисленого на попередній ітерації, а потім викликом процедури **Iter_Jacoby** обчислюються нові значення вектора. В операторі 3 колективною функцією MPI_Allgatherv здійснюється розсилання обчислених значень вектора від усіх до усіх. Для правильної роботи функції MPI_Allgatherv попередньо виконується заповнення масивів sendcounts і **displs**. Після обміну кожен процес одержує новий обчислений вектор X. У кореневому процесі з рангом Root обчислюється норма похибки й порівнюється із наперед заданою. Умовою продовження циклу – є те, що Result ≠ 0, тому кореневий процес розсилає значення Result всім процесам за допомогою колективної функції MPI_Bcast (оператор 4).

Ну й нарешті, нижче наведений можливий вихідний код головної програми паралельного алгоритму методу простої ітерації. Тут виконується ініціалізація паралельних процесів, одержання й розсилання вихідної матриці AB значень коефіцієнтів системи (це матриця A, доповнена стовпцем вільних членів B). Також розсилаються вимірність матриці MATR_SIZE, початкове наближення вектора розв’язку X, точність обчислень Error. Ця головна функція може мати, на-

приклад, такий вигляд:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <memory.h>
int myid, numprocs, Root = 0, *sendcounts, *displs;
/* myid - ранг поточного процесу, numprocs - кількість процесів,
   Root - номер кореневого процесу, sendcounts и displs -
   допоміжні масиви для організації обмінів даними */
double *AB, *A,* X;
/* AB - вихідна матриця методу, може бути задана будь-яким
   способом X - вектор, містить початкове наближення*/
int main(int argc,char *argv[]) {
  int i, size, MATR_SIZE, SIZE;
/* MATR_SIZE - вимірність системи, size - кількість рядків
   матриці в кожному процесі, SIZE - кількість елементів
   матриці в кожному процесі*/
  double Error;                          // точність обчислень
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == Root) {
// тут програма одержує вимірність системи MATR_SIZE
// виділяється пам'ять для матриці AB
  AB = (double *)malloc(sizeof(double) * MATR_SIZE *
                        (MATR_SIZE + 1));
/* тут повинна уводитися матриця AB вимірності MATR_SIZE + 1
   (матриця + стовпець вільних членів) і задаватися точність
   обчислень Error */
  }
// розсилання необхідних даних всім процесам одразу
  MPI_Bcast(&MATR_SIZE, 1, MPI_INT, Root, MPI_COMM_WORLD);
  MPI_Bcast(&Error, 1, MPI_DOUBLE, Root, MPI_COMM_WORLD);
// виділення пам'яті для вектора X
  X = (double *)malloc(sizeof(double) * MATR_SIZE);
  if (myid == Root) {
// тут задається початкове значення для вектора X
  }
// розсилання вектора X всім процесам
  MPI_Bcast(X, MATR_SIZE, MPI_DOUBLE, Root, MPI_COMM_WORLD);
/* визначення кількості елементів вектора X,
   обчислюються паралельно в кожному процесі */
  size = (MATR_SIZE / numprocs) + ((MATR_SIZE % numprocs) >myid
         ? 1 : 0 );                                        // 1
// у кожному процесі виділяється пам'ять для матриці A
  A = (double *)malloc(sizeof(double) *
                       (MATR_SIZE + 1) * size);
  displs = (int *)malloc(numprocs * sizeof(int));
  sendcounts = (int *)malloc(numprocs * sizeof(int));
```

```
  SIZE = (MATR_SIZE + 1) * size;
// розсилання частин матриці по процесах
  MPI_Gather(&SIZE, 1, MPI_INT, endcounts, 1, MPI_INT,
             Root, MPI_COMM_WORLD);                          // 3
  displs[0] = 0;
  for (i = 1; i < numprocs; ++i)
    displs[i] = displs[i - 1] + sendcounts[i - 1];
  MPI_Scatterv(AB, sendcounts, displs, MPI_DOUBLE, A,
               (MATR_SIZE + 1) * size, MPI_DOUBLE, Root,
               MPI_COMM_WORLD);                              // 2
// розв'язання СЛАР методом простої ітерації
  SolveSLAE(MATR_SIZE, size, Error);
// звільнення пам'яті
  free(sendcounts); free(displs);
  free(AB); free(A); free(X);
  MPI_Finalize();
  return 0;
}
```

Після всіх підготовчих операцій кожен процес визначає кількість координат `size` вектора, з якими він буде працювати (оператор 1). Розподіл матриці `AB` по процесах зручно виконати функцією `MPI_Scatterv` (оператор 2). Функція `MPI_Gather` (оператор 3) використана для заповнення допоміжного масиву `sendcounts`, необхідного для виконання `MPI_Scatterv`. Інший допоміжний масив `displs` заповнюється просто в циклі.

2.2.2 Метод Гаусса-Зейделя розв'язання СЛАР

Метод Якобі, як уже вказувалося, легко піддається розпаралелюванню, але метод Гаусса-Зейделя більш ефективний, оскільки потребує помітно менше ітерацій, маючи кращу збіжність до розв'язку. Однак у методі Гаусса-Зейделя обчислення кожного компонента вектора залежить від компонентів вектора, обчислених на цій же ітерації. Тому на перший погляд метод носить суто послідовний характер. Нижче розглянута паралельна модифікація алгоритму методу Гаусса-Зейделя.

Виходячи з матричного представлення системи рівнянь (2.13), можна одержати ітераційну схему Гаусса-Зейделя:

$$(A^- + D)\, x^{k+1} = b - A^+ x^k; \qquad k = 0, 1, 2, \dots,$$

або

$$Dx^{k+1} = -A^- x^{k+1} - A^+ x^k + b; \qquad k = 0, 1, 2, \dots,$$

Звідси координатна форма методу для системи загального вигляду:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k\right); \quad i = \overline{1,n}; \quad k = 1, 2, \dots . \qquad (2.18)$$

Таким чином, відмінність методу Гаусса-Зейделя від методу простої ітерації полягає в тім, що нові значення вектора обчислюються не тільки на основі значень попередньої ітерації x^k, але й з використанням значень уже обчислених на даній ітерації x^{k+1}. А система рівнянь, що описує метод Гаусса-Зейделя, має вигляд:

$$\begin{aligned} x_1^{k+1} &= f_1\left(x_1^k, x_2^k, \dots, x_n^k\right), \\ x_2^{k+1} &= f_2\left(x_1^{k+1}, x_2^k, \dots, x_n^k\right), \\ &\dots \\ x_n^{k+1} &= f_n\left(x_1^{k+1}, x_2^{k+1}, \dots, x_{n-1}^{k+1}, x_n^k\right). \end{aligned} \qquad (2.19)$$

З системи рівнянь (2.19) є цілком очевидним, що через наявність інформаційних зв’язків не можна безпосередньо реалізовувати паралельний алгоритм, аналогічний методу простої ітерації, тому для паралельної реалізації використовується модифікований метод Гаусса-Зейделя.

Метод полягає в тім, що обчислення координат вектора розподіляються по процесах аналогічно методу простої ітерації. *У кожному процесі обчислюється своя кількість координат вектора за методом Гаусса-Зейделя, використовуючи тільки обчислені значення вектора даного процесу*. Розходження в паралельній реалізації методу Гаусса-Зейделя в порівнянні з методом простої ітерації полягає тільки в процедурі обчислення значень вектора, коли замість про-

цедури Iter_Jacoby використовується паралельна процедура GaussZeidel, яка наведена нижче.

```
/* На вході задається матриця A, її вимірність MATR_SIZE,
   кількість елементів size вектора, які обчислюються, у
   даному процесі. Обчислюються нові значення вектора X с
   номера first, використовуючи старі значення вектора X */
void GaussZeidel(int size, int MATR_SIZE, int first) {
  int i, j;
  double Sum;
  for (i = 0; i < size; ++i) {
    Sum = 0;
    for (j = 0; j < i + first; ++j)
      Sum += A[ind(i, j, MATR_SIZE)] * X[j];
    for (j = i + 1 + first; j < MATR_SIZE; ++j)
      Sum += A[ind(i, j, MATR_SIZE)] * X[j];
    X[i + first] = (A[ind(i, MATR_SIZE, MATR_SIZE)] -
                    Sum) / A[ind(i, i + first, MATR_SIZE)];
  }
}
```

Дуже цікавим є паралельний алгоритм Гаусса-Зейделя, який можна застосувати для розв'язання СЛАР великої вимірності, коли кількість процесорів обчислювальної системи менша або дорівнює вимірності задачі.

В цьому випадку процес ітерацій для СЛАР (2.13) замість координатної форми методу (2.18) може представлятись у дещо іншому вигляді:

$$x_i^{k+1} = \sum_{j=1}^{i-1} c_{ij} x_j^{k+1} + \sum_{j=i}^{n} c_{ij} x_j^k + g_i \qquad i = \overline{1,n},\ k = \overline{0,\infty}, \qquad (2.20)$$

де, як і раніш $x^k = \left(x_1^k, \ldots, x_n^k\right)^T \in \Upsilon^n$ – вектор наближення, що обчислюється на k-му кроці ітерації (вектор k-го наближення). В свою чергу $C = \left\| c_{ij} \right\| \in \Upsilon^{nn}$ – ***матриця ітерації***; $g = (g_1, \ldots, g_n)^T \in \Upsilon^n$ – ***вектор ітерації***. Співвідношення між матрицею і вектором ітерації обумовлюються такими виразами $C = E - \alpha A$, $g = \alpha b$, де α – ***параметр ітерації***. Матриця A и вектор b, які входять в ці вирази, є відповідно матрицею коефіцієнтів і вектором вільних членів в системі рівнянь (2.13), E – одинична матриця. Зрозуміло, що процес ітерацій починається з деякого вектора початкового наближення –

$x^0 = x^{0}_{0} \in \Upsilon^n$, а як оцінка близькості вектора k-го наближення до вектора-розв'язку може використовуватися оцінка відносної похибки обчислення: $\epsilon = \|x^{k+1} - x^k\| / \|x^{k+1}\|$, $k = \overline{1,\infty}$.

Характер задачі повністю обумовлюється виразом (2.20), тому метод паралельного обчислення повинен засновуватися на ньому і при цьому природно припустити, що кожен крок ітерації алгоритму може виконуватись паралельно, а також вважати, що для проведення обчислень є p процесорів, тобто p гілок алгоритму, які здатні проводити обчислення незалежно одна від одної.

Є сенс кожній такій гілці привласнити її порядковий номер, причому нумерацію слід починати з нуля. При цьому також припускається, що всі гілки рівноцінні в обчислювальному відношенні. Це означає, що час обчислень не залежить від порядкового номера гілки і обчислення, що виконуються всіма гілками над заданим набором даних потребують того ж самого часу. Порядок проведення обчислень кожною гілкою, а також і процедури взаємодії, які здійснюються при цьому між ними, бажано описувати за допомогою паралельного алгоритму.

Цілком очевидно, що в (2.20) для фіксованого i ($i = \overline{1,n}$) всі добутки можуть виконуватися паралельно. Результатом підсумовування множень є i-та компонента вектора нового наближення. Як вже відмічалось раніш, указане паралельне обчислення буде мати сенс тільки у випадку, коли кількість гілок алгоритму менша або дорівнює вимірності задачі. При рівності зазначених параметрів утворюється випадок, коли кожна гілка робить одне множення при обчисленні чергового компонента вектора. Тому було б цілком правильним кожен стовпець матриці C, а також відповідні компоненти векторів наближень, обробляти в окремій гілці алгоритму. Якщо стовпці цієї матриці пронумерувати, починаючи ліворуч з нуля й до n праворуч, то m-й ($m = \overline{1,n}$) стовпець буде використовуватися в гілці з номером m. Компоненти вектора ітерації також повинні бути розподілені по гілках так, що компонента з номером m ($m = \overline{1,n}$) дістається ($m - 1$)-й гілці. Описана декомпозиція наведена на рис. 2.3.

$$\begin{bmatrix} x_1^k & x_2^k & \ldots & \boxed{x_m^k} & \ldots & x_n^k \end{bmatrix}^T$$

$$\begin{bmatrix} c_{1,1} & c_{1,2} & \ldots & \boxed{c_{1,m}} & \ldots & c_{1,n} \\ c_{2,1} & c_{2,2} & \ldots & c_{2,m} & \ldots & c_{2,n} \\ \vdots & \vdots & \ldots & \vdots & \ldots & \vdots \\ c_{m,1} & c_{m,2} & \ldots & c_{m,m} & \ldots & c_{m,n} \\ \vdots & \vdots & \ldots & \vdots & \ldots & \vdots \\ c_{n,1} & c_{n,2} & \ldots & c_{n,m} & \ldots & c_{n,n} \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ \boxed{g_m} \\ \vdots \\ g_n \end{bmatrix}$$

$$\begin{bmatrix} x_1^k & x_2^{k+1} & \ldots & \boxed{x_m^{k+1}} & \ldots & x_n^{k+1} \end{bmatrix}^T$$

Рисунок 2.3 − Декомпозиція даних для випадку, коли n дорівнює p

Прямокутниками на цьому рисунку позначені дані, які в результаті указаної декомпозиції одержуються m-ю гілкою.

Паралельне обчислення вектора k-го наближення складається з послідовного визначення його компонентів за виразом (2.20). Причому не викликає сумніву те, що при зроблених припущеннях необхідні операції множення виконуються гілками паралельно. Після одержання відповідних добутків повинна бути знайдена їхня сума. Таке підсумовування здійснюється шляхом взаємодії гілок алгоритму. І знову ж ці дії можуть виконуватись паралельно. Максимальна ефективність при підсумовуванні досягається у випадку, коли гілки організують звичайне двійкове ***дерево комунікацій***. На рис. 2.4 зображено саме таке дерево комунікацій для семи гілок алгоритму (рис. 2.4 а) і, як видно з рисунка, воно повністю подібне до двійкового дерева підсумовування методом здвоювання, яке наведено на рис. 1.22 б).

Кожному вузлу дерева відповідає своя гілка алгоритму. Стрілками показаний напрямок передачі результатів підсумовування (часткових сум).

Процедура одержання суми $\sum_{i=0}^{p-1} \sigma(i)$, де через $\sigma(i)$ позначений доданок, який міститься в i-у вузлі дерева, називається ***здвоюванням щодо операції суми***. Вона виконується знизу-вверх шляхом послідовного застосування елемен-

тарних здвоювань, які виробляються трійкою вузлів. Зазначена трійка являє собою піддерево вихідного дерева (рис. 2.4 б) і є 3-х вузловим ***шаблоном взаємодії*** – $\Psi(p_i, p_j, p_k)$.

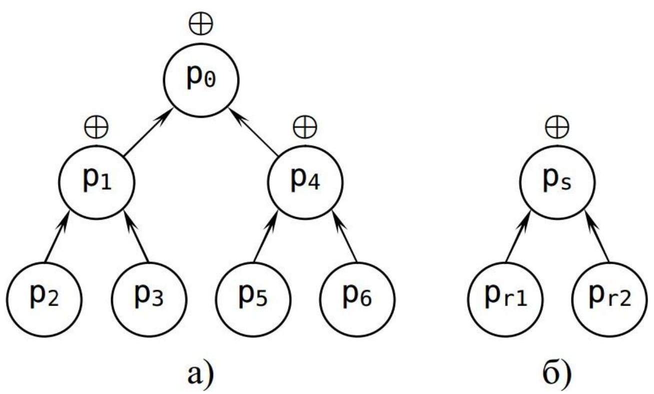


Рисунок 2.4 − Дерева комунікацій: а) – дерево комунікацій семи гілок алгоритму, б) – трьох вузлове піддерево здвоювання

Безпосередньо одержання суми здійснюється в такій послідовності. Вузол p_s одержує від вузла величину $\sigma(p_{r1})$ й збільшує власний доданок $\sigma(p_s)$ на одержану величину: $\sigma(p_s) = \sigma(p_s) + \sigma(p_{r1})$. Потім виконується аналогічна взаємодія з вузлом p_{r2} і виконується нове перерахування значення $\sigma(p_s)$, тобто $\sigma(p_s) = \sigma(p_s) + \sigma(p_{r2})$. У підсумку вузол p_s містить суму величин, що зберігаються в розглянутому піддереві. При цьому говорять, що до вузла p_s застосована операція здвоювання на шаблоні $\Psi(p_s, p_{r1}, p_{r2})$. Отримана в результаті сума потім виступає як елементарний доданок у піддереві, для якого p_s є дочірнім вузлом, і т.д. Процедура одержання суми всіх елементів дерева завершується після застосування операції здвоювання у кореневому вузлі. Нескладно помітити, що при такому підсумовуванні існують здвоювання, які можна виконувати паралельно. Відповідні трійки вузлів лежать на одному рівні дерева, а кількість цих рівнів дорівнює: $n_p = [\log 2(p)]$, де операція [] (квадратні дужки) означає одержання цілої частини числа.

Вертаючись до формули (2.20), можна помітити, що i-а компонента вектора нового наближення, одержувана при здвоюванні, призначена для подаль-

шого використання в гілці з номером і. Тому доцільно саме цю гілку співвіднести з кореневим вузлом дерева здвоювання. Таке співвіднесення може реалізовуватись за допомогою так званого ***динамічного дерева***, у якому по черзі кореню відповідають всі гілки алгоритму.

Тепер можна описати порядок виконання паралельних обчислень, тобто алгоритм, для випадку, коли $n = p$. При цьому, попереднє значення компонента вектора наближення, яке використовується i-ю гілкою алгоритму, буде позначене як $(x_i)_{попер}$, а через $(x_i)_{поточ}$ – позначене значення, використовуване гілкою при обчисленнях за формулою (2.20).

Таким чином послідовність обчислень, яка виконується i-ю гілкою алгоритму ($i = \overline{0,p-1}$), являє собою періодичне виконання таких дій.

1) Участь у паралельному знаходженні j-ї ($j = \overline{1,n}$) компоненти нового вектора. В цьому разі спостерігаються два випадки:
   а) при $j \neq i$ гілка спочатку обчислює $\sigma_j(i) = p_{ji}(x_i)_{поточ}$, а потім використовує знайдену величину в процедурі здвоювання на дереві комунікацій;
   б) при $j = i$, гілка просто бере участь в обчисленні відповідної їй компоненти вектора нового наближення; для цього обчислюється величина поточної суми $\sigma_i(i) = p_{ii}(x_i)_{поточ} + g_i$, а потім виконується здвоювання величин $\sigma_i(k)$ ($k = \overline{0,p-1}$), і у кожній гілці перераховується $(x_i)_{попер}$, й $(x_i)_{поточ}$. Перераховування виконується таким чином: $(x_i)_{попер} = (x_i)_{поточ}$; $(x_i)_{поточ} = \sum_{k=0}^{p-1} \sigma_i(k)$ (результат здвоювання).
2) Перевірка умови завершення процесу обчислення після обчислення всіх компонентів нового вектора.

Взаємодія при перевірці також здійснюється на дереві комунікацій. Норма вектора, необхідна для оцінки близькості знайденого вектора до точного розв'язку може обчислюватись різними способами, кожен з яких обумовлює свій порядок взаємодії гілок. В кожному разі, у якості вихідних даних у вузлах дерева виступають величини $\mu(i) = (x_i)_{поточ} - (x_i)_{попер}$ й $\eta(i) = (x_i)_{поточ}$. У випа-

дку невиконання умови закінчення роботи $\epsilon \geq \epsilon_0$, де ϵ_0 – наперед задана відносна похибка обчислення, приймається рішення про продовження обчислень, починаючи з п. 1.

Зрозуміло, що результат перевірки повинен бути отриманий у всіх гілках алгоритму. Це означає, що після здвоювання необхідно розподілити прапор закінчення роботи по всіх гілках алгоритму. Зазначений розподіл можна виконати в напрямку, зворотному напрямку здвоювання, тобто зверху-вниз. При цьому здійснюється взаємодія вузлів у протилежному напрямку (передача інформації від вузла-предка до вузла-нащадка). Так само як і для здвоювання, для розподілу можна виділити трійки вузлів, які взаємодіють між собою паралельно. Вони лежать на одному рівні дерева комунікацій, а кількість цих рівнів дорівнює n_p.

На завершення розглядання описаного паралельного алгоритму, є сенс з'ясувати доцільність його використання. Для цього можна спробувати оцінити і порівняти час, потрібний для проведення обчислень за послідовним й паралельним алгоритмами.

Базові операції, яких потребує обчислення за формулою (2.20), а також й перевірка умови закінчення обчислень потребують певних тимчасових витрат. Якщо для відповідних дій ввести такі позначення:

1) $t_{дод}$ – час, необхідний для виконання операції додавання;

2) $t_{множ}$ – час, витрачуваний на виконання операції множення;

3) $t_{пор}$ – час порівняння двох операндів;

4) k – кількість ітерацій послідовного й паралельного алгоритмів;

5) $t_{ком}(d)$ – час пересилання d елементів від однієї гілки до іншої.

При перевірці умови закінчення роботи вважати, що обчислюється кубічна норма $\|x\|_3 = \max\limits_{1 \leq i \leq n} |x_i|$, а також позначити час виконання послідовного алгоритму через T_1, а час виконання паралельного алгоритму через T_p, то можна показати, що часи обчислень за послідовним й паралельним алгоритмами будуть оцінюватися відповідно до таких виразів:

$$T_1 = k((n^2 + 1)t_{множ} + n(n + 1)t_{дод} + (2n + 1)t_{пор}),$$

$$T_p = k((n+1)t_{множ} + (n+2n_p+1)t_{дод} + 2n_p((n+1)t_{ком}(1) + t_{ком}(2)) + 2(2n_p+1)t_{пор})$$

Маючи ці два часи, можна приблизно оцінити прискорення наведеного алгоритму. Як завжди формула для обчислення прискорення має вигляд $S_p = T_1/T_p$. Якщо у нульовому наближенні вважати, що $t_{дод} \approx t_{множ} \approx t_{пор}$, час пересилання одиниці даних $\mathbf{t}_{ком}(1) \sim 5 \cdot 10^2 \mathbf{t}_{множ}$, а, в свою чергу, $\mathbf{t}_{ком}(d) \sim d \cdot \mathbf{t}_{ком}(1)$, то залежність прискорення алгоритму від кількості процесорів (гілок) обчислювальної системи має вигляд приблизно зображений на рис. 2.5.

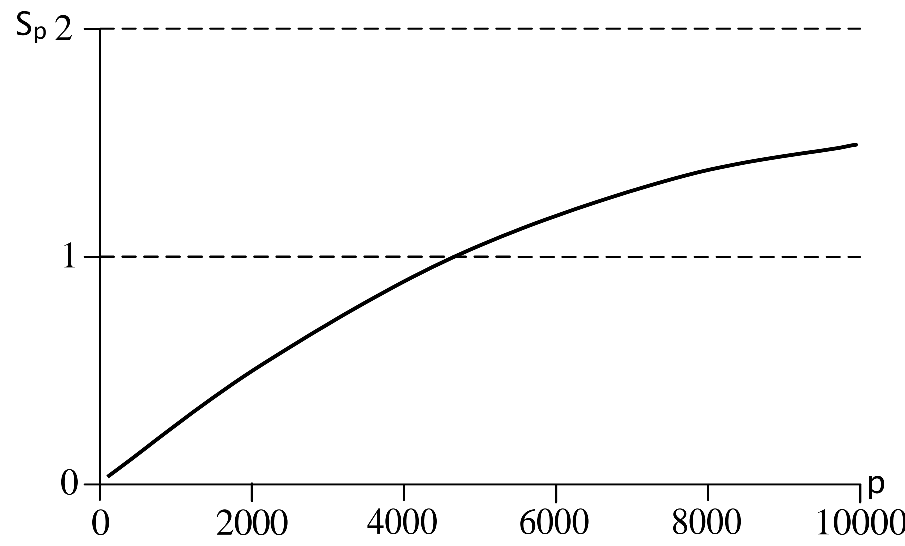


Рисунок 2.5 − Графік залежності прискорення алгоритму Гаусса-Зейделя від кількості паралельних гілок

З рисунка видно, що прискорення зростає з ростом розмірності задачі. Однак воно зростає дуже повільно і асимптотично наближається до значення 2. Така поведінка прискорення пояснюється наявністю частих і тривалих взаємодій між гілками алгоритму. Незважаючи на це, для розв'язання СЛАР великої вимірності існує можливість прискорити процес обчислення й одержати виграш у часі при паралельних обчисленнях методом Гаусса-Зейделя в разі, коли кількість гілок алгоритму дорівнює вимірності задачі. Взагалі кажучи, використання паралельних обчислювальних систем в таких випадках є цілком доцільним.

2.3 Паралельні алгоритми роботи з матрицями

Поряд з базовими алгоритмами розв'язання СЛАР розташовуються й тісно пов'язані з ними базові матричні задачі. Це пояснюється тим, що при моделюванні різноманітних явищ як матеріального світу, так і суспільного життя дуже часто зустрічаються операції з матрицями і векторами. Таким чином є сенс розглянути й деякі алгоритми паралельного виконання матричних обчислень, наприклад, операцій множення матриць.

2.3.1 Обчислення добутку матриці на вектор

Найбільш простим і зрозумілим є ***самоплануючий алгоритм*** множення матриць, зокрема алгоритм обчислення добутку матриці на вектор. Цей алгоритм у подальшому може бути доволі просто узагальнений і на задачу множення матриць. У самоплануючому алгоритмі один з процесів, який призначається головним (master), відповідає за роботу підлеглих (slave) йому процесів, розсилаючи їм вихідні дані й збираючи результати їхньої роботи.

Відповідна програма з використанням функцій бібліотеки MPI представлена нижче.

У загальній частині цієї програми описуються основні об'єкти задачі: матриця A, вектор-множник b, вектор результатів обчислення – c. Визначається, також, число процесів у комунікаторі. Код двох альтернативних гілок програми описує дії, які з одного боку виконуються головним процесом і з іншого – його підлеглими процесами.

У задачі множення матриці на вектор одиниця роботи, яку потрібно роздати процесам, складається зі скалярного добутку рядка матриці A на вектор b.

```
      program main
      use mpi
      integer MAX_ROWS, MAX_COLS, rows, cols
      parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
C      Матриця A, вектор-множник b, результуючий вектор c
     double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS)
     &c(MAX_ROWS)
```

```
C      Низка деяких додаткових, допоміжних змінних
       double precision buffer (MAX_COLS), ans
       integer myid, master, numprocs, i, j, numsent, sender
       integer status (MPI_STATUS_SIZE), anstype, row, ierr
       call MPI_INIT(ierr)
       call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
       call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
C      Визначення рангу головного процесу
       master = 0
C      Кількість рядків і стовпців матриці A
       rows = 10
       cols =100
       if (myid .eq. master) then
C      Код головного процесу
C      Ініціалізація A и b
       do 20 j = 1, cols
       b(j) = j
       do 10 i = 1, rows
   10  a(i, j) = i
   20  continue
       numsent = 0
C      Пересилання вектора b всім підлеглим процесам
       call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
      &MPI_COMM_WORLD, ierr)
C      Пересилання відповідного рядка кожному підлеглому процесу;
       do 40 i = 1, min(numprocs - 1, rows)
       do 30 j = 1, cols
   30  buffer(j) = a(i, j)
C      У параметрі TAG передається номер рядка - i
       call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i, i,
      &MPI_COMM_WORLD, ierr)
C      Підрахування кількості відправлених рядків
       numsent = numsent + 1
   40  continue
C      Прийом результатів від підлеглих процесів
       do 70 i = 1, rows
C      MPI_ANY_TAG - указує на те, що приймаються дані від
C      будь-якого з підлеглих процесів
       call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
      &MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
C      Визначення рангу відправника та номера обробленого рядка
       sender = status(MPI_SOURCE)
       anstype = status(MPI_TAG)
C      Заповнення відповідного елемента результуючого масиву
C      значенням одержаним від підлеглого процесу
       c(anstype) = ans
       if (numsent .lt. rows) then
C      Пересилання такого необробленого рядка матриці
       do 50 j = 1, cols
   50  buffer(j) = a(numsent + 1, j)
       call MPI_SEND (buffer, cols, MPI_DOUBLE_PRECISION, sender,
      &numsent + 1, MPI_COMM_WORLD, ierr)
```

```
        numsent = numsent + 1
        else
C       Якщо всі рядки оброблені - пересилання ознаки кінця роботи
        call MPI_SEND(MPI_BOTTQM, 0, MPI_DOUBLE_PRECISION, sender,
     &0, MPI_COMM_WORLD, ierr)
        endif
   70 continue
        else
C       Код підлеглих процесів
C       Одержання вектора b всіма підлеглими процесами
        call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     &MPI_COMM_WORLD, ierr)
C       Вихід, якщо процесів більше кількості рядків матриці
        if (numprocs .gt. rows) goto 200
C       Одержання підлеглим процесом рядка матриці
   90 call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
     &MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
C       Якщо одержано повідомлення про оброблення всіх рядків
        if (status(MPI_TAG) .eq. MPI_BOTTQM) then goto 200
        else
C       Визначення номера отриманого рядка
        row = status(MPI_TAG)
C       Обчислення скалярного добутку векторів
        ans = 0.0
        do 100 i = 1, cols
  100 ans = ans + buffer(i) * b(i)
C       Повернення результату обчислень головному процесу
        call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, row,
     &MPI_COMM_WORLD, ierr)
C       Продовження циклу роботи для прийому такого рядка
        goto 90
        endif
C       Закінчення роботи програми
  200 continue
        endif
        call MPI_FINALIZE(ierr)
        stop
        end
```

У головному процесі спочатку за допомогою виклику MPI_BCAST у кожен підлеглий процес пересилається вектор b. Потім за допомогою синхронної операції MPI_SEND у кожен підлеглий процес пересилається один рядок матриці A. Після цього головний процес переходить у режим очікування відповіді від будь-якого підлеглого процесу з будь-яким тегом повідомлення (функція MPI_RECV з параметрами MPI_ANY_SOURCE і MPI_ANY_TAG). Одержавши результат від підлеглого процесу, який завершив обчислення скалярного добутку ряд-

ка матриці на вектор b, головний процес поміщає отриманий скаляр у відповідний елемент результуючого масиву c. Індекс цього елемента масиву c визначається за вмістом службового масиву status. Після цього перевіряється факт розсилання всіх рядків матриці A, і при необхідності підлеглому процесу посилає нова порція роботи. Цикл закінчується, коли всі рядки будуть роздані й від кожного підлеглого процесу буде отриманий результат. Посилка повідомлення з параметром MPI_BOTTQM означає наказ про закінчення роботи підлеглим процесом.

У альтернативній гілці програми підлеглі процеси спочатку викликом функції MPI_BCAST одержують вектор b. Потім виконується цикл, у тілі якого процеси переходять до очікування одержання повідомлення від головного процесу (функція MPI_RECV). Одержання повідомлення, зміст якого не дорівнює MPI_BOTTQM, а являє собою черговий рядок матриці A, служить сигналом до обчислення скалярного добутку рядка й вектора b. Обчислене значення відправляється головному процесу й здійснюється перехід на початок циклу.

Таким чином, якщо припустити відсутність втрат часу, потрібного для виконання пересилання даних між процесами, одержується прискорення роботи паралельної програми відносно послідовної, яке дорівнює кількості процесорів у обчислювальній системі.

2.3.2 Самоплануючий алгоритм множення матриць

Описаний вище самоплануючий алгоритм добутку матриці на вектор легко узагальнюється на задачу множення матриць. Програма такого самоплануючого алгоритму представлена нижче.

```
      program main
      use mpi
      integer MAX_AROWS, MAX_ACOLS, MAX_BCOLS
      parameter (MAX_AROWS = 20, MAX_ACOLS = 1000, MAX_BCOLS = 20)
C     Оголошення матриць A,B,C
      double precision a(MAX_AROWS,MAX_ACOLS),
     &b(MAX_ACOLS,MAX_BCOLS), c(MAX_AROWS, MAX_BCOLS)
C     Оголошення додаткових допоміжних змінних
```

```
      double precision buffer(MAX_ACOLS), ans(MAX_ACOLS)
      double precision starttime, stoptime
      integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
     integer i, j, numsent, sender, anstype, row, arows, acols,
     &brows, bcols, crows, ccols
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
C     Завдання кількості рядків і стовпців для матриць А, В й С
      arows = 10
      acols = 20
      brows = 20
      bcols = 10
      crows =arows
      ccols = bcols
C     Призначення рангу головному процесу
      master = 0
      if (myid .eq. master) then
C     Код головного процесу
C     Виконання ініціалізації матриць А и В
      do 10 j = 1, acols
      do 10 i = 1, arows
   10 a(i, j) = i
      do 20 j = 1, bcols
      do 20 i = 1, brows
   20 b(i, j) = i
C     Посилання матриці В всім підлеглим процесам одразу
      do 25 i = 1, bcols
   25 call MPI_BCAST(b(1, i), brows, MPI_DOUBLE_PRECISION, master,
     &MPI_COMM_WORLD, ierr)
      numsent = 0
C     Посилання одного рядка матриці А кожному підлеглому процесу
C     У параметр TAG заноситься номер рядка = i
C     Для простоти покладається, що arows >= numprocs - 1
      do 40 i = 1, numprocs - 1
      do 30 j = 1, acols
   30 buffer(j) = a(i, j)
      call MPI_SEND (buffer, acols, MPI_DOUBLE_PRECISION, i, i,
     &MPI_COMM_WORLD, ierr)
C     Підраховується кількість відправлених рядків
   40 numsent = numsent+1
C     Цікл очикування результатів обчислення від підлеглих процесів
      do 70 i = 1, crows
      call MPI_RECV(ans, ccols, MPI_DOUBLE_PRECISION,
     &MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
     sender = status(MPI_SOURCE)
      anstype =status(MPI_TAG)
C     Розміщення одержаного вектора у відповідне місце матриці С
      do 45 j =1, ccols
   45 c(anstype, j) = ans(j)
C     Якщо оброблені не всі рядки матриці А
      if (numsent .lt. arows) then
```

```
C      Відправлення вільному процесору чергоаого рядка матриці A
       do 50 j = 1, acols
    50 buffer(j) = a(numsent + 1, j)
       call MPI_SEND (buffer, acols, MPI_DOUBLE_PRECISION, sender,
      &numsent + 1, MPI_COMM_WORLD, ierr)
       numsent = numsent + 1
       else
C      Якщо оброблені всі рядки - відправлення ознаки кінця роботи
       call MPI_SEND(MPI_BOTTQM, 1, MPI_DOUBLE_PRECISION, sender, 0,
      &MPI_COMM_WORLD, ierr)
       endif
    70 continue
       else
C      Код підлеглих процесів
C      Одержання матриці B всіма підлеглими процесами
       do 85 i = 1, bcols
    85 call MPI_BCAST(b(1, i), brows, MPI_DOUBLE_PRECISION, master,
      &MPI_COMM_WORLD, ierr)
C      Одержання рядка матриці A кожним підлеглим процесом
    90 call MPI_RECV (buffer, acols, MPI_DOUBLE_PRECISION, master,
      &MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
       if (status (MPI_TAG) .eq. 0) then goto 200
       else
C      Обчислення результату множення рядка матриці A на матрицю B
       row = status(MPI_TAG)
       do 100 i = 1, bcols
       ans(i) = 0.0
       do 95 j = 1, acols
    95 ans(i) = ans(i) + buffer(j) * b(j, i)
   100 continue
C      Відправлення одержаного вектора результату головному процесу
      call MPI_SEND(ans, bcols, MPI_DOUBLE_PRECISION, master, row,
      &MPI_COMM_WORLD, ierr)
       goto 90
       endif
   200 endif
       call MPI_FINALIZE(ierr)
       stop
       end
```

Наведена програма не потребує особливих пояснень, оскільки алгоритм її роботи практично ні чим не відрізняється від програми множення матриці на стовпець. Основні дії програми пояснені у відповідних коментарях.

2.3.3 Клітинний алгоритм множення квадратних матриць

Для множення квадратних матриць $A = (a_{ij})$ і $B = (b_{ij})$ однакового поряд-

ку n часто використовується інший алгоритм, який одержав назву ***клітинного алгоритму***. Цей алгоритм можна продемонструвати на основі використання деяких основних особливостей бібліотеки MPI, які полягають у наявності такої структури як комунікатор і в можливості створення віртуальних топологій процесів.

У клітинному алгоритмі число процесів p повинне бути повним квадратом деякого числа q, тобто $p = q^2$, а порядок матриці n повинен націло ділитися на це ж число q так, що $n' = n/q$. Обидві матриці A і B розподіляються по процесах таким чином, що кожен з них одержує невелику підматрицю порядку n'. Тоді процеси можна розглядати як віртуальну двовимірну сітку розміром $q \times q$, і кожен процес пов’язаний з підматрицею $n' \times n'$. Для випадку, коли $p = 9$ і $n = 6$, розбивка матриці A на відповідні підматриці показана на рис. 2.6.

Процес 0 $A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$	Процес 1 $A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$	Процес 2 $A_{02} = \begin{pmatrix} a_{04} & a_{05} \\ a_{14} & a_{15} \end{pmatrix}$
Процес 4 $A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$	Процес 4 $A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$	Процес 5 $A_{12} = \begin{pmatrix} a_{24} & a_{25} \\ a_{34} & a_{35} \end{pmatrix}$
Процес 6 $A_{20} = \begin{pmatrix} a_{40} & a_{41} \\ a_{50} & a_{51} \end{pmatrix}$	Процес 7 $A_{21} = \begin{pmatrix} a_{42} & a_{43} \\ a_{52} & a_{53} \end{pmatrix}$	Процес 8 $A_{22} = \begin{pmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{pmatrix}$

Рисунок 2.6 − Розбивка матриці A на підматриці порядку $n' = 2$ при $p = 9$ і $n = 6$

У загальному випадку кожному **i**-му процесу передаються підматриці A_{st} індекси елементів, яких визначаються в такий спосіб:

$$A_{st} = \begin{pmatrix} a_{s \times n', t \times n'} & \Lambda & a_{(s+1) \times n' - 1, t \times n'} \\ M & O & M \\ a_{s \times n', (t+1) \times n' - 1} & \Lambda & a_{(s+1) \times n' - 1, (t+1) \times n' - 1} \end{pmatrix}$$

Алгоритм клітинного множення матриць можна описати так.

У циклі від 0 до q – 1 з кроком, який дорівнює одиниці, виконується:

1) у кожному рядку процесів з матриці A вибирається підматриця A_{ru} так, що в r-му рядку u = (r + k) mod q, де k – поточне значення змінної циклу;
2) в кожному рядку процесів обрана підматриця пересилається всім іншим процесам того ж рядка;
3) кожен з процесів множить отриману підматрицю A_{ru} на підматрицю B_{st}, яка зараз належить цьому процесу;
4) всі процеси пересилають підматрицю B_{st} процесу, що перебуває вище (з першого рядка пересилання цієї підматриці виконується по кільцю в останній рядок);
5) здійснюється перехід до пункту 1.

У бібліотеці MPI всі взаємодії процесів здійснюються усередині структури, що називається ***комунікатором***. Безперечно, що в клітинному алгоритмі має сенс кожен рядок і кожен стовпець процесів трактувати як новий комунікатор. Створення декількох комунікаторів з вихідного комунікатора MPI_COMM_WORLD у бібліотеці MPI можна зробити декількома різними способами.

Наприклад, комунікатор для першого рядка, у який входять процесори з номерами 0, 1, ... , q – 1, можна зробити, використовуючи приблизно такий код:

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;
/* Створення списку процесів у новому комунікаторі */
process_ranks = (int*)malloc(q * sizeof(int));
for (proc = 0; proc < q; proc++) process_ranks[proc] = proc;
/* Одержання групи процесів у комунікаторі MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
/* Створення нової групи */
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks,
               &first_row_group);
/* Створення нового комуникатора зі створеної групи */
```

```
MPI_Comm_create(MPI_COMM_WORLD, first_row_group,
                &first_row_comm);
```

У цьому фрагменті в операторі циклу спочатку створюється список процесів для нового комунікатора. Потім процеси включаються в групу, яка в свою чергу останнім оператором фрагмента перетворюється на новий комунікатор з ідентифікатором first_row_comm. Аналогічно створюються комунікатори для інших рядків і стовпців. Таким чином, процеси в першому рядку одержують можливість виконувати колективні операції комунікації повністю незалежно від інших процесів. В цьому разі код обміну інформацією може виглядати, наприклад, так:

```
int my_rank_in_first_row;
float* A_00;
// Тут my_rank - номер процесу в MPI_GROUP_WORLD
if (my_rank < q) {
  MPI_Comm_rank(first_row_comm, &my_rank_in_first_row);
// Виділяється пам'ять для A_00, n_bar - порядок підматриці
  A_00 = (float*) malloc (n_bar * n_bar * sizeof(float));
  if(my_rank_in_first_row == 0) {/* В цьому циклі виконується
    ініціалізація підматриці A_00 */}
  MPI_Bcast(A_00, n_bar*n_bar, MPI_FLOAT, 0, first_row_comm);
}
```

Навіть для простого прикладу при p = 9 і n = 6, при такому підході до створення комунікаторів, необхідно повторити код створення комунікаторів шість разів (по три рази для рядків, а також стовпців). Для того, щоб уникнути подібного повторення рутинних дій, в бібліотеці MPI передбачена спеціальна функція MPI_Comm_split, що дозволяє створювати декілька окремих комунікаторів одночасно. У такому фрагменті коду єдиний виклик MPI_Comm_split створює q нових комунікаторів. Недоліком такого способу є те, що всі комунікатори мають те саме однакове ім'я my_row_comm.

```
MPI_Comm my_row_comm;
int my_row;
/* my_rank - номер процесу в MPI_COMM_WORLD. q * q = p */
my_row = my_rank / q;
MPI_Comm_split(MPI_COMM_WORLD, my_row, my_rank,
               &my_row_comm);
```

Бібліотека MPI надає ще один дуже ефективний засіб для створення комунікаторів, при якому застосовується можливість створення і використання віртуальної топології процесів. Такий підхід дозволяє ідентифікувати процеси в глобальному комунікаторі `MPI_COMM_WORLD` не по їхніх рангах, а по координатах квадратної сітки й відповідно сформувати окремі комунікатори для кожного рядка й стовпця.

Для зв'язування віртуальної декартової топології із глобальним комунікатором необхідно викликати функцію `MPI_Cart_create` передавши їй відповідні параметри:

1) вимірність декартової системи (для клітинного алгоритму – два);
2) розмір кожного виміру (для алгоритму – `q` рядків і `q` стовпців);
3) періодичність кожного виміру – параметри, які визначають необхідність згортання декартової системи в циліндр по тій або іншій координаті (у клітинному алгоритмі це необхідно зробити по другій координаті – по стовпцях);
4) параметри, які вказують на можливість переупорядкування процесів у групі, що дозволяє провести оптимізацію накладення декартової топології на дійсну топологію міжз'єднань фізичних процесорів у системі.

Після виклику `MPI_Cart_create` створюється новий комунікатор, що містить всі процеси із глобального комунікатора, і за допомогою функцій `MPI_Cart_rank` і `MPI_Cart_coords` вже усередині цього комунікатора можна визначити нові ранги процесів і їхні координати в декартовій мережі. Розбивка отриманого комунікатора на комунікатори для рядків і стовпців виконується за допомогою функції `MPI_Cart_sub`. Як параметри в цю функцію передається масив булевих елементів, які визначають приналежність кожного виміру до нового комунікатора. Функція `MPI_Cart_sub` як і `MPI_Comm_split` створює відразу `q` нових комунікаторів, але `MPI_Cart_sub` використовується тільки з комунікатором, який попередньо має зв'язану декартову топологію.

Нижче наведений приклад функції `Setup_grid`, яка створює ціле сімейство комунікаторів для клітинного алгоритму перемножування матриць:

```
void Setup_grid(GRID_INFO_TYPE* grid) {
  int old_rank, dimensions[2], periods[2], coordinates[2],
      varying_coords[2];
/* Визначення параметрів координатної сітки */
  MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
  MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
  grid->q = (int) sqrt((double) grid->p);
  dimensions[0] = dimensions[1] = grid->q;
  periods[0] = periods[1] = 1;
/* Створення комунікатора декартової топології вимірності 2 */
  MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods, 1,
                  &(grid->comm));
  MPI_Comm_rank(grid->comm, &(grid->my_rank));
  MPI_Cart_coords(grid->comm, grid->my_rank, 2, coordinates);
  grid->my_row = coordinates[0];
  grid->my_col = coordinates[1];
/* Створення комунікатора рядка */
  varying_coords[0] = 0;
  varying_coords[1] = 1;
  MPI_Cart_sub(grid->comm, varying_coords, &(grid->row_comm));
/* Створення комунікатора стовпця*/
  varying_coords[0] = 1;
  varying_coords[1] = 0;
  MPI_Cart_sub(grid->comm, varying_coords, &(grid->col_comm));
}
```

У цьому прикладі через велику кількість змінних, необхідних для роботи функції, вони об'єднані в спеціальну структуру GRID_INFO_TYPE, яка має такий вигляд:

```
typedef struct {
  int p;                // загальна кількість процесів
  MPI_Comm comm;        // комунікатор сітки процесів
  MPI_Comm row_comm;    // комунікатор рядка
  MPI_Comm col_comm;    // комунікатор стовпця
  int q;                // порядок сітки
  int my_row;           // номер рядка
  int my_col;           // номер стовпця
  int my_rank;          // номер у комунікаторі сітки
} GRID_INFO_TYPE;
```

Тепер, на основі отриманого сімейства комунікаторів, можна виконати множення матриць за допомогою, наприклад, функції Mult_Matr код якої наведений нижче.

```
#define LOCAL_MATRIX_TYPE double*
#define DERIVED_LOCAL_MATRIX MPI_Aint
void Mult_Matr (int n, GRID_INFO_TYPE* grid,
                LOCAL_MATRIX_TYPE* local_A, local_B, local_C) {
  LOCAL_MATRIX_TYPE* temp_A;
  int step, bcast_root, n_bar, source, dest, tag = 43;
  MPI_Status status;
/* Розраховується порядок підматриць = n/q */
  n_bar = n/grid->q;
/* Попереднє обнулiння елементiв результуючої матрицi */
  Set_to_zero(local_C);
/* Обчислення адрес для циклiчного зсуву пiдматриць матрицi B */
  source = (grid->my_row + 1) % grid->q;
  dest = (grid->my_row + grid->q - 1) % grid->q;
/* Видiлення пам’ятi для пересилання пiдматриць матрицi A */
  temp_A = Local_matrix_allocate(n_bar);
  for (step = 0; step < grid->q; step++) {
    bcast_root = (grid->my_row + step) % grid->q;
    if (bcast_root == grid->my_col) {
      MPI_Bcast(local_A, 1, DERIVED_LOCAL_MATRIX, bcast_root,
                grid->row_comm);
/* Множення вiдповiдних пiдматриць у кожному процесi */
      Local_matrix_multiply(local_A, local_B, local_C); }
    else {
/* Пересилання пiдматрицi A */
      MPI_Bcast(temp_A, 1, DERIVED_LOCAL_MATRIX, bcast_root,
                grid->row_comm);
/* Знов таки множення пiдматриць*/
      Local_matrix_multiply(temp_A, local_B, local_C); }
/* Пересилання пiдматриць B */
    MPI_Send(local_B, 1, DERIVED_LOCAL_MATRIX, dest, tag,
             grid->col_comm);
    MPI_Recv(local_B, 1, DERIVED_LOCAL_MATRIX, source, tag,
             grid->col_comm, &status); }
}
```

Для зручності й більшої наочності програми за допомогою директив `#define` визначені два нових типи даних:

– `LOCAL_MATRIX_TYPE` – тип елементів множених матриць;

– `DERIVED_LOCAL_MATRIX` – тип матриць, що перемножуються, (відповідає простому скалярному типу бібліотеки MPI для мови C – `MPI_Aint`, змінна якого має розмір, що дорівнює розміру покажчика).

2.4 Питання і завдання для самоперевірки

1) Які співвідношення називають рекурентними порядку k? Що таке рекурентна задача і її різновиди: рекурентна задача зі скалярним результатом, рекурентна задача з векторним результатом? Як визначається лінійна рекурентна задача? Наведіть приклади.
2) Які три алгоритми застосовуються для паралельного розв'язання лінійної рекурентної задачі у разі коли ця задача розв'язується для строго нижньотрикутної матриці A і двох векторів x і c?
3) Розкажіть, у чому полягає суть алгоритму викреслювання стовпців, як виконується оцінка коефіцієнта прискорення цього алгоритму у разі блокового розподілу даних. Спробуйте оцінити коефіцієнт прискорення у разі блочно-циклічного розподілу даних в припущенні відсутності часу на обмін даними між процесами.
4) В чому полягає ефект Гайдна, що визначає коефіцієнт Гайдна?. Чому дорівнює коефіцієнт Гайдна при блоковому розподілі даних? Оцініть коефіцієнт Гайдна для випадку блочно-циклічного розподілу даних.
5) Опишіть модифікацію алгоритму викреслювання стовпців, яка дозволяє практично уникнути впливу ефекту Гайдна на прискорення паралельних обчислень.
6) Опишіть алгоритм логарифмічного підсумовування для розв'язання лінійної рекурентної задачі підсумовування n чисел. Як визначається коефіцієнт прискорення для цього алгоритму?
7) На чому оснований алгоритм рекурентного добутку? Що таке оцінка складності алгоритму і як через неї виражається коефіцієнт прискорення?
8) У яких випадках доцільно застосовувати так званий блоковий алгоритм? Наведіть відповідні приклади. Спробуйте самостійно розглянути цей підхід для статистичної задачі одержання послідовності з 10^6 чисел, ймовірність появи яких підлягає розподіленню Пуассона $p(k) = \frac{\lambda^k}{k\,!} e^{-\lambda}$,

при $\lambda = 10$ і значеннях k випадковим чином рівномірно розподілених у діапазоні $[0, 20]$.

9) Виведіть ітераційну формулу для розв'язання СЛАР – методом Якобі. Зобразіть простір ітерацій, який обумовлює ярусно-паралельну форму вирішення цієї задачі. Надайте словесне описання паралельного алгоритму розв'язання задачі.
10) Доповніть програму на стор. 105 вводом вихідних даних, перекладіть її мовою Java з використанням методів бібліотеки MPJ Express і спробуйте запустити на виконання.
11) Чому алгоритм Гаусса-Зейделя розв'язання СЛАР гірше піддається розпаралелюванню аніж алгоритм Якобі? Які способи обходу цього ускладнення Ви знаєте? Докладно опишіть паралельний алгоритм Гаусса-Зейделя, який застосовується для розв'язання СЛАР великої вимірності при кількості процесорів системи меншій за вимірність задачі.
12) Поясніть, у чому полягає самоплануючий алгоритм множення матриць, зокрема алгоритм обчислення добутку матриці на вектор? Перекладіть програму на стор. 115 мовою Java, відкомпілюйте і виконайте множення матриці на вектор.
13) На стор. 118 наведена програма мовою Fortran множення двох матриць. Перекладіть програму мовою Java, відкомпілюйте і виконайте тестове множення двох матриць.
14) Поясніть роботу клітинного алгоритму множення квадратних матриць. Яким повинно бути співвідношення між кількістю паралельних процесів і порядком матриць при застосуванні цього алгоритму? Надайте словесний опис алгоритму (послідовність операцій для досягнення поставленої мети).
15) Розкажіть, як за допомогою методів (функцій) бібліотеки MPI можна спростити клітинний алгоритм множення квадратних матриць за рахунок створення окремих комунікаторів. Які методи створення комунікаторів

Ви знаєте? Як у бібліотеці MPI створюється віртуальна декартова топологія?

16) Спробуйте мовою Java написати, відкомпілювати і запустити на виконання власну тестову програму множення матриць з використанням клітинного алгоритму

3 ПАРАЛЕЛЬНІ МЕТОДИ РОЗВ'ЯЗАННЯ СИСТЕМ НЕЛІНІЙНИХ РІВНЯНЬ

На практиці дуже часто, наприклад при моделюванні різноманітних електронних схем, доводиться розв'язувати як одиночні нелінійні рівняння, так і великі системи нелінійних рівнянь. Цей розділ конспекту як раз і присвячений питанням паралельного розв'язання подібних рівнянь.

3.1 Паралельне розв'язання нелінійних рівнянь

Вирішити питання паралельного розв'язання нелінійного рівняння доволі просто.

З курсу чисельних методів добре відомі послідовні методи розв'язання нелінійних рівнянь. Наприклад метод хорд для знаходження кореня нелінійного рівняння $f(x) = 0$ на відрізку $[a; b]$ за умов $m < f'(x) < M$ і $f''(x) > 0$ полягає у такому. Крива $y = f(x)$ заміняється хордою $P(a)Q(b)$ (рис. 3.1 а).

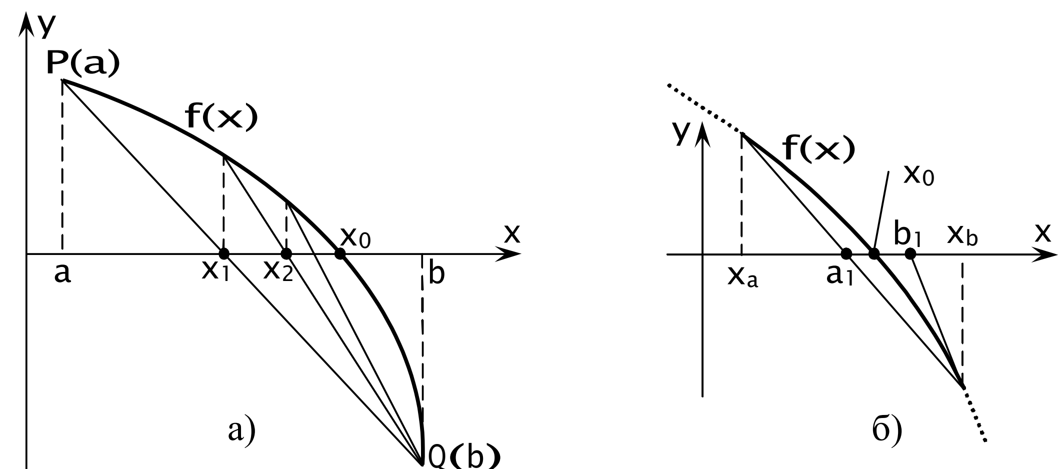


Рисунок 3.1 − Знаходження кореня нелінійного рівняння методом хорд
а) послідовний алгоритм; б) паралельний алгоритм

Після цього з розв'язку звичайного лінійного рівняння визначається точка $x_1 = a - \dfrac{f(a)(b-a)}{f(b) - f(a)}$, після чого проводиться хорда $P(x_1)Q(b)$, знаходиться точка x_2 й так далі, поки не буде досягнута відповідна, наперед задана точність

$\epsilon \leq |f(x_k) - f(x_{k-1})|$. Швидкість збіжності методу сильно залежить від вигляду функції $f(x)$ і ширини інтервалу $[a;\ b]$. Тому в ряді випадків є сенс використання найпростішого паралельного алгоритму розв'язання нелінійного рівняння.

Для реалізації задачі на n процесорах, тобто для побудови паралельного алгоритму з p гілок, відрізок $[a;\ b]$ розбивається на $n - 1$ підвідрізків на границях яких обчислюється $f(x)$. Це дає змогу визначити новий підвідрізок $[x_a;\ x_b]$ довжиною $\frac{b - a}{n}$, який містить корінь x_0. При цьому точки x_a й x_b є суміжними таким чином, що значення функції у них мають різні знаки (рис. 3.1 б).

Після відрізка $[x_a;\ x_b]$ розглядається відрізок $[a_1;\ b_1]$, де

$$a_1 = x_a - \frac{f(x_a)(x_b - x_a)}{f(x_b) - f(x_a)}; \qquad b_1 = x_b - \frac{f(x_b)}{M} \tag{3.1}$$

Відрізок $[a_1;\ b_1]$ знову розбивається на n частин, повторюється описана процедура, одержується відрізок $[a_2;\ b_2]$ і т.д., поки довжина відрізка не стане менше заданого значення. Паралельні методи обчислення кореня нелінійного рівняння виправдані тільки тоді, коли обсяг обчислень значення $f(x)$ набагато перевершує обсяг обчислень границь відрізка $[a_i;\ b_i]$ за формулою (3.1). Але при виконанні цієї умови вони, без урахування часу на пересилання даних, дають прискорення обчислень майже в n раз.

Не важко показати, що аналогічні міркування можна провести і при розв'язанні нелінійного рівняння з використанням методу Ньютона, одержавши той самий результат.

3.2 Вирішення питання паралельного розв'язання системи нелінійних рівнянь

Питання паралельного розв'язання системи нелінійних рівнянь виникає при вирішенні великого кола прикладних задач, наприклад при пошуку безумовного екстремуму функцій багатьох змінних при задоволенні необхідних умов

пошуку, або при застосуванні неявних методів інтегрування звичайних диференціальних рівнянь і т.і. Перелік можна продовжувати нескінченно довго. Сама ж проблема розв'язання системи n нелінійних рівнянь з n невідомими є більш складною ніж проблема розв'язання одиночного нелінійного рівняння, або проблема розв'язання системи n лінійних рівнянь.

Система n нелінійних рівнянь з n невідомими має вигляд:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0, \\ f_2(x_1, x_2, \ldots, x_n) = 0, \\ \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0, \end{cases} \tag{3.2}$$

де $f_i(x_1, x_2, \ldots, x_n) = 0$, $i = 1, \ldots, n$ – нелінійні функції, визначені й безперервні в деякій області.

На відміну від систем лінійних рівнянь не існує прямих методів розв'язання нелінійних систем загального вигляду. Тому для розв'язання систем нелінійних рівнянь за звичаєм використовуються ітераційні методи. При цьому система представляється у вигляді:

$$\begin{cases} x_1 = \varphi_1(x_1, x_2, \ldots, x_n), \\ x_2 = \varphi_2(x_1, x_2, \ldots, x_n), \\ \vdots \\ x_n = \varphi_n(x_1, x_2, \ldots, x_n). \end{cases}$$

Або переходячи до ітераційної форми можна записати:

$$x_i^{k+1} = \varphi_i(x_1^k, x_2^k, \ldots, x_n^k), \; i = 1, 2, \ldots, n. \tag{3.3}$$

Як видно з отриманого виразу (3.3), оскільки відсутні інформаційні зв'язки між різними компонентами вектора рішення x_i, задача може бути вирішена паралельно, наприклад, *методом простих ітерацій*, який є аналогічним відповідному методу простих ітерацій для СЛАР. Значення невідомих на $(k+1)$-

й ітерації обчислюється із використанням їхніх значень на попередній ітерації. Можливо й застосування *методу Зейделя*, що практично збігається з методом Гаусса-Зейделя для розв'язання систем лінійних рівнянь, коли значення x_i^{k+1} обчислюється для i-го рівняння системи з використанням уже обчислених на поточній ітерації значень невідомих.

Але обидва ці методи, і простої ітерації, і метод Зейделя мають досить погану збіжність, тому що успіх багато в чому залежить від удалого вибору початкових наближень невідомих. Початкове наближення повинне бути досить близькими до істинного рішення. У противному випадку ітераційний процес може навіть розходитися.

Набагато більшу збіжність має ***метод Ньютона*** розв'язання системи нелінійних рівнянь. Він оснований на таких міркуваннях. Якщо задано вектор початкових наближень $\left(x_1^0, x_2^0, \ldots, x_n^0\right)$, то ітераційний процес знаходження рішення системи можна представити у вигляді:

$$\begin{cases} x_1^{k+1} = x_1^k + \Delta x_1^k, \\ x_2^{k+1} = x_2^k + \Delta x_2^k, \\ \ldots \\ x_n^{k+1} = x_n^k + \Delta x_n^k. \end{cases} \quad k=0, 1, 2, \ldots \qquad (3.4)$$

де значення деяких приростів Δx_1^k, Δx_2^k, …, Δx_n^k визначаються з розв'язання нової системи лінійних алгебраїчних рівнянь, всі коефіцієнти якої виражаються через вже відоме попереднє наближення. Ця нова система алгебраїчних рівнянь будується в такий спосіб.

Якщо позначити функції $f_i(x_1, x_2, \ldots, x_n)$ в деякій точці, яка відповідає попередньому наближенню $\left(x_1^k, x_2^k, \ldots, x_n^k\right)$ через F_i і розкласти кожну з них у ряд Тейлора в околиці цієї точки, обмежившись тільки похідними першого порядку малості, то утворюється система таких лінійних алгебраїчних рівнянь для приростів Δx_i^k.

$$
\begin{cases}
F_1 + \dfrac{\partial F_1}{\partial x_1}\Delta x_1^k + \dfrac{\partial F_1}{\partial x_2}\Delta x_2^k + \Lambda + \dfrac{\partial F_1}{\partial x_n}\Delta x_n^k = 0, \\
F_2 + \dfrac{\partial F_2}{\partial x_1}\Delta x_1^k + \dfrac{\partial F_2}{\partial x_2}\Delta x_2^k + \Lambda + \dfrac{\partial F_2}{\partial x_n}\Delta x_n^k = 0, \\
\Lambda \\
F_n + \dfrac{\partial F_n}{\partial x_1}\Delta x_1^k + \dfrac{\partial F_n}{\partial x_2}\Delta x_2^k + \Lambda + \dfrac{\partial F_n}{\partial x_n}\Delta x_n^k = 0.
\end{cases}
\qquad (3.5)
$$

Таким чином, ітераційний процес розв’язання вихідної системи рівнянь (3.2) методом Ньютона складається з визначення приростів Δx_1^k, Δx_2^k, …, Δx_n^k до відповідних значень невідомих вихідної системи на кожній k-й ітерації за допомогою розв’язання отриманої СЛАР (3.5) і, тим самим, одержання наближення на $(k+1)$-й ітерації.

Для розв’язання таких лінійних систем можна застосовувати самі різні методи, як прямі, так і ітераційні з урахуванням вимірності n розв’язуваної задачі, причому, як видно з отриманих рівнянь, розв’язання легко розпаралелюються через відсутність інформаційних залежностей між F_i.

Використання методу Ньютона передбачає диференційованість функцій F_i і невиродженість, так званої, ***матриці Якобі***:

$$
J = \begin{bmatrix}
\dfrac{\partial F_1}{\partial x_1} & \dfrac{\partial F_1}{\partial x_2} & \Lambda & \dfrac{\partial F_1}{\partial x_n} \\
\dfrac{\partial F_2}{\partial x_1} & \dfrac{\partial F_2}{\partial x_2} & \Lambda & \dfrac{\partial F_2}{\partial x_n} \\
 & \Lambda & & \\
\dfrac{\partial F_n}{\partial x_1} & \dfrac{\partial F_n}{\partial x_2} & \Lambda & \dfrac{\partial F_n}{\partial x_n}
\end{bmatrix},
$$

тобто не рівність нулю $\det J(x^k)$ при всіх наближеннях вектора вирішень x^k .

При використанні метода Ньютона удалий вибір початкового наближення також залишається доволі важливим. Якщо початкове наближення обране в досить малій околиці шуканого розв’язку системи, то ітерації сходяться до точного розв’язку, причому збіжність в цьому випадку квадратична. Зі збільшенням кількості рівнянь системи збіжність дещо погіршується.

У практичних обчисленнях критерієм закінчення ітерацій звичайно використовується нерівність $\left\| x^{k+1} - x^k \right\| \le \epsilon$ або, що те ж саме, $\max\limits_{1 \le i \le n} \left| \Delta x_i^k \right| \le \epsilon$, де ϵ – це задана точність обчислень.

При знаходженні розв'язання системи нелінійних рівнянь методом Ньютона доволі часто трапляється те, що найбільш трудомісткою частиною алгоритму виявляється знаходження елементів матриці Якобі. Але знаходження частинних похідних стандартними чисельними методами з огляду на інформаційну незалежність похідних однієї від одної дуже просто розпаралелюється й не є доволі складною задачею для багатопроцесорних систем. Але необхідність виконання операції диференціювання на кожній ітерації призводить до того, що частіше використовують ***модифікований метод Ньютона***. Суть цього методу полягає в тому, що коефіцієнти матриці Якобі обчислюються тільки один раз при завданні вектора початкової ітерації й потім ці значення використовується при всіх таких ітераціях. Такий підхід власне призводить до трохи меншої збіжності розв'язання. Але за рахунок лише одноразового обчислення коефіцієнтів матриці Якобі вдається значно заощадити загальний час необхідний для здійснення розрахунку.

3.3 Питання і завдання для самоперевірки

1) У чому полягає процес перетворення послідовного алгоритму розв'язання нелінійного рівняння у його паралельну форму? Мовою Java із застосуванням методів бібліотеки MPJ Express напишіть паралельний варіант програми розв'язання нелінійного рівняння методом хорд.
2) У чому полягає проблема розв'язання системи нелінійних рівнянь у порівнянні з розв'язанням одиночного нелінійного рівняння, або системи лінійних рівнянь? Як виглядає представлення системи нелінійних рівнянь для її розв'язання ітераційними методами, як в цьому разі строїться ітераційна процедура?

3) Що Ви можете сказати про збіжність методів простих ітерацій і Зейделя розв'язання системи нелінійних рівнянь? Який метод має кращу збіжність? Поясніть його алгоритм.
4) Які обмеження накладаються на функції, що входять у систему рівнянь? У чому полягає суть модифікованого методу Ньютона розв'язання системи нелінійних рівнянь?
5) Із застосуванням методів бібліотеки MPJ Express спробуйте написати паралельну Java-програму розв'язання системи нелінійних рівнянь модифікованим методом Ньютона.

4 ЕФЕКТИВНІСТЬ ПАРАЛЕЛЬНИХ ОБЧИСЛЮВАЛЬНИХ МЕТОДІВ ПІД ЧАС РОЗВ'ЯЗАННЯ НЕЛІНІЙНОЇ ЗАДАЧІ КОШІ ДЛЯ ЗДР

Для розгляду питання ефективності паралельних методів розв'язання ЗДР необхідно визначити коефіцієнт прискорення чисельного рішення диференціальних рівнянь яким-небудь методом на багатопроцесорному комп'ютері відносно до однопроцесорного.

Наприклад, для паралельного розв'язання ***задачі Коші*** для одного звичайного диференціального рівняння першого порядку

$$x' = f(x), \tag{4.1}$$

при початковій умові $x(t_0) = x_0$, можна скористатися алгоритмами так званих ***блокових методів***. За аналогією з послідовними методами чисельного розв'язання задачі Коші для звичайних диференціальних рівнянь розрізняють два типи ***паралельних блокових методів*** – однокроковий і багатокроковий. Так само, як і в послідовних методах, ***однокрокові блокові методи*** дозволяють одержати розв'язок, використовуючи початкові умови задачі. Для розв'язання багатокроковим блоковим методом спочатку необхідно яким-небудь однокроковим методом знайти значення у відповідному числі точок початкового відрізка.

При використанні обчислювальної схеми паралельного однокрокового блокового методу множина точок рівномірної сітки із кроком τ, тобто $\Omega_\tau = \{t_l = l\tau,\ l = 0, 1, 2, \ldots\}$, розбивається на N блоків, які містять k точок кожен (рис. 4.1). На рисунку для кожного блоку уведений номер точки i, який лежить у діапазоні від 1 до k, i-та точка блоку з номером n позначена як $t_{n,i} = t_{0,1} + [(n-1)k + i]\tau$, а для всієї множини k точок $t_{n,i}$, які належать блоку, можна ввести позначення $t_{n,i} \in T_n^{(k)}$, де i змінюється в діапазоні від 1 до k, а $n = 1, 2, \ldots$ Початкова точка $t_{0,1}$ включена до деякого блоку 0 (на рисунку непозначений).

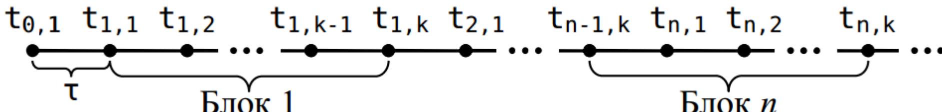


Рисунок 4.1 − Схема розбивки на блоки для однокрокового k-точкового методу

При чисельному розв’язанні задачі Коші однокроковим блоковим методом для кожного такого блоку нові k значень приблизного розв’язку обчислюються одночасно з використанням значення тільки в останній точці попереднього блоку.

Якщо для приблизного значення розв’язання задачі Коші (4.1) у точці $t_{n,i}$, оброблюваного блоку, увести позначення $u_{n,i}$, то різницеві рівняння для однокрокових блокових методів приймуть вигляд

$$u_{n,i} = u_{n,0} + i\tau\left[b_i f_{n,0} + \sum_{j=1}^{k} a_{i,j} f_{n,j} \right], \; i = 1 \ldots k, \; n = 1, 2, \ldots, \tag{4.2}$$

де $f_{n,j} = f(t_n + j\tau, u_{n,j})$, граф розрахунку цих рівнянь наведений на рис. 4.2.

$t_{n-1,k} = t_{n,0}$, $t_{n,1}$, $t_{n,2}$ … $t_{n,k}$, $T_n^{(k)}$

Рисунок 4.2 − Граф розрахунку однокрокової k-точкової різницевої схеми

У випадку блокового багатокрокового методу початковий блок уже повинний містити кілька точок сітки, у яких задаються необхідні початкові значення приблизного розв’язку для продовження розрахунків (рис 4.3).

$t_{0,1}$ $t_{0,2}$ … $t_{0,m}$ $t_{1,1}$ $t_{1,2}$ … $t_{1,k-1}$ $t_{1,k}$ $t_{2,1}$ … $t_{n-1,k}$ $t_{n,1}$ $t_{n,2}$ … $t_{n,k}$

Блок 0 τ Блок 1 Блок n

Рисунок 4.3 − Схема розбивки на блоки для m-крокового k-точкового методу

У загальному випадку рівняння багатокрокових різницевих методів для блоку з k точок при використанні обчислених значень приблизного розв'язку в m вузлах, які передують блоку і з урахуванням уведених вище позначень можна записати у вигляді

$$u_{n,i} = u_{n,0} + i\tau\left[\sum_{j=1}^{m} b_{i,j} f_{n,j-m} + \sum_{j=1}^{k} a_{i,j} f_{n,j}\right], \quad i = 1 \ldots k, \; n = 1, 2, \ldots \tag{4.3}$$

Формули (4.3) визначають m-кроковий k-точковий різницевий метод, граф розрахунку якого приведений на рис. 4.4. У ньому $T_n^{(k)} = \{t_{n,1}, t_{n,2}, \ldots, t_{n,k}\}$ – множина точок, у яких по формулах (4.3) визначаються приблизні значення розв'язку. Множина $T_{n-1}^{(m)} = \{t_{n,1-m}, t_{n,2-m}, \ldots, t_{n,0}\}$ містить точки, приблизне значення розв'язку в яких було обчислено на попередньому етапі.

$$\underbrace{t_{n,1-m}\; t_{n,-m} \cdots t_{n,-1}\; t_{n,0}}_{T_{n-1}^{(m)}} \;\; \underbrace{t_{n,1}\; t_{n,2} \cdots t_{n,k}}_{T_n^{(k)}}$$

Рисунок 4.4 – Граф розрахунку m-крокової k-точкової різницевої схеми

Одним зі способів визначення коефіцієнтів $a_{i,j}$ і $b_{i,j}$ у формулах (4.2) і (4.3) є, так званий, ***інтегроінтерполяційний метод***. Для виконання розрахунків цим методом будується інтерполяційний багаточлен $L_{m+k-1}(t)$ з вузлами інтерполяції в точках $t_{n,j-m}$ і з відповідними значеннями правої частини рівняння (4.1) тобто $f_{n,j-m} = f(t_{n,j-m}, u_{n,j-m})$ для значень j в інтервалі від 1 до $m + k$. Потім виконується інтегрування отриманого полінома в на відрізку ($t_{n,0}$, $t_{n,i}$), де i змінюється в діапазоні $1 \ldots k$

$$u_{n,i} = u_{n,0} + \int_{t_{n,0}}^{t_{n,i}} L_{m+k-1}(t)\,dt,$$

у результаті одержуються рівняння (4.3) для обраних m і k.

Так для однокрокового чотирьохточкового блокового методу обчислення вказаним методом з інтерполяційним багаточленом $L_4(t)$ у вузлах інтерполяції $t_{n,j-m}$, при $m = 1$, $k = 4$ та j у діапазоні від $1-m$ до k призводять до таких формул:

$$u_{n,1} = \frac{1}{720}\tau\left(251f_{n,0} + 646f_{n,1} - 264f_{n,2} + 106f_{n,3} - 19f_{n,4}\right) + u_{n,0},$$

$$u_{n,2} = \frac{1}{90}\tau\left(29f_{n,0} + 124f_{n,1} + 24f_{n,2} + 4f_{n,3} - f_{n,4}\right) + u_{n,0},$$

$$u_{n,3} = \frac{3}{80}\tau\left(9f_{n,0} + 34f_{n,1} + 24f_{n,2} + 14f_{n,3} - f_{n,4}\right) + u_{n,0},$$

$$u_{n,4} = \frac{2}{45}\tau\left(7f_{n,0} + 32f_{n,1} + 12f_{n,2} + 32f_{n,3} + 7f_{n,4}\right) + u_{n,0}.$$

Отримані формули неявно визначають значення приблизного розв'язку, тобто необхідно розв'язати отриману нелінійну систему алгебраїчних рівнянь відносно $u_{n,i}$ при i від 1 до 4. Розв'язання необхідно починати зі значення $n = 1$.

Опускаючи досить громіздкий доказ, можна стверджувати, що якщо різницеве рівняння (4.2) апроксимує вихідне рівняння (4.1) й норма погрішності приблизного розв'язку, отриманого яким-небудь способом на початковій ділянці інтегрування для перших *m* вузлів $\|Z_0\| = 0$, то розв'язання різницевої задачі (4.2) сходиться при $\tau \to 0$ до розв'язання вихідної задачі (4.1), причому порядок точності збігається з порядком апроксимації $O(\tau^{k+m})$.

Безпосередньо ітераційні формули паралельного розв'язання системи різницевих рівнянь багатокрокового багатоточкового методу (4.2) для довільного блоку n можна одержати в такий спосіб.

Якщо для графа розрахунку (див. рис. 4.4) увести позначення:

– $U_n^{(k)} = \left\{u_{n,i},\ i = 1,2,\ldots,k\right\}$ – вектор значень приблизного розв'язку в точках блоку $T_n^{(k)}$;

– $F_n^{(k)} = \left\{f\left(t_{n,i}, u_{n,i}\right),\ i = 1,2,\ldots,k\right\}$ – вектор, компонент якого дорівнює значенню правої частини рівняння (4.1) в точці блоку $T_n^{(k)}$ для відповідного значення наближеного розв'язку;

– $U_{n-1}^{(m)} = \left\{u_{n,j-m},\ j = 1,2,\ldots,m\right\}$ – вектор значень приблизного розв'язку в точ-

ках блоку $T_{n-1}^{(m)}$;

– $F_{n-1}^{(m)} = \{f(t_{n,j-m}, u_{n,j-m}),\ j = 1, 2, \ldots, m\}$ – вектор, компонент j якого дорівнює значенню правої частини рівняння (4.1) у відповідній точці блоку $T_{n-1}^{(m)}$ для відповідного їй значення приблизного розв'язку,

то систему (4.2) можна записати у векторній формі в такий спосіб:

$$D^{-1}\frac{\left(U_n^{(k)} - u_{n,0}e\right)}{\tau} = BF_{n-1}^{m} + AF_n^{k}, \qquad (4.4)$$

де $D = (d_{ii})$ – діагональна матриця з елементами $d_{ii} = i$, при $i = 1, 2, \ldots, k$;

B – матриця з елементами b_{ij}, індекси i змінюються у діапазоні від 1 до k, а j від 1 до m;

A – матриця з елементами a_{ij}, індекси i та j лежать у діапазоні $1\ldots k$;

e – одиничний вектор-стовпець.

Вираз (4.4) вже можна перетворити в ітераційні формули паралельного розв'язання системи:

$$u_n^{s+1} = u_n e + \tau DBv'_{n-1} + \tau DA\left(u'_n\right)^s, \qquad (4.5)$$

де s – номер ітерації. По формулі (4.5) перераховуються компоненти вектора приблизного розв'язку u_n^{s+1}, які належать блоку n. При цьому компоненти вектора $v'_{n-1,i} = F_{n-1,i} = f\left(t_{n-1,i}, u_{n-1,i}\right)$ для i від 1 до m, які вже визначені у вузлах попереднього блоку, зберігають свої значення при цих обчисленнях, а компоненти вектора $\left(u'_{n,i}\right)^s = \left(F_{n,i}\right)^s = f\left(t_{n,i}, u_{n,i}^s\right)$ при i від 1 до k обчислюються знову на кожній ітерації. Щоб розпочати розв'язання m-кроковим різницевим методом, попередньо необхідно яким-небудь однокроковим різницевим методом визначити значення приблизного розв'язку $u_{0,i}$ у перших $m-1$ точках, що примикають до початкової точки t_0 відрізка інтегрування. Таким чином, будуть визначені значення компонентів $v'_{0,i} = F_{0,i} = f\left(t + i\tau, u_{0,i}\right)$ вектора v'_0. Після чого треба задати

вихідне наближення для компонентів $u^0_{1,i}$ вектора u^0_1 при i від 1 до k, значення яких можуть бути, наприклад, знайдені за допомогою m-крокових формул ***Адамса-Башфорта***. У загальному випадку m-крокові формули Адамса-Башфорта для значень приблизного розв'язку у вузлах k-точкового блоку на першій ітерації у векторній формі можна записати у вигляді

$$u^1_n = u_n \mathbf{e} + \tau_1 \mathbf{B}_n y'_{n-1} .$$

Нарешті, можна показати, що виконання k кроків обчислень за формулою (4.5) забезпечує одержання приблизного розв'язку з локальною помилкою порядку $O(\tau^{k+m+1})$.

І на закінчення, ефективність чисельного розв'язання задачі Коші паралельними методами можна оцінити на основі таких міркувань.

У випадку довільної правої частини рівняння (4.1) трудомісткість методу обумовлюється числом звертань для обчислення значень правої частини рівняння на кожен вузол сітки. Оцінку ефективності однокрокових блокових методів можна зробити, розрахувавши відношення часу виконання алгоритму ***Рунге-Кутти*** на однопроцесорному комп'ютері до часу виконання однокрокового блокового алгоритму відповідного порядку на паралельній обчислювальній системі. Час послідовного обчислення T_s приблизних значень розв'язку з точністю $O(\tau^{k+1})$ у всіх k вузлах блоку при виконанні алгоритму Рунге-Кутти $(k+1)$-го порядку точності на одному процесорі можна визначити за формулою

$$T_s = (k + 1)^2 \mathbf{t}_f + k^2 (\mathbf{t}_{ad} + \mathbf{t}_{mul}),$$

де T_s – час виконання алгоритму на одному процесорі,

$\mathbf{t}_f$ – час обчислення значення функції $f(t, x)$,

$\mathbf{t}_{ad}$, $\mathbf{t}_{mul}$ – час виконання операцій додавання й множення відповідно.

Паралельні обчислення за формулами (4.3) можна виконувати, наприклад, виділивши кожному вузлу блоку окремий процесор. У такий спосіб при наявності k процесорів можна одночасно робити обчислення значень $F_{n,i,s}$, після

чого також одночасно за формулами (4.3) розраховувати значення $u_{n,i,s}$ для кожного фіксованого s. При цьому буде потрібно одночасно здійснювати пересилання даних сусіднім процесорам, ці дії можна виконати навіть у комутаційній мережі найпростішої топології, наприклад, у кільці. Урахування часу передачі даних сусідньому процесору можна виконати ввівши деякий додатковий параметр t_{ta}, який характеризує часові властивості мережі міжз'єднань.

Тоді повний час T_p паралельного обчислення приблизних значень розв'язку з тією же точністю, що й при послідовному алгоритмі, для всіх вузлів блоку складе

$$T_p = k\mathbf{t}_f + 2k(\mathbf{t}_{ad} + \mathbf{t}_{mul}) + k(k-1)\mathbf{t}_{ta}.$$

Звідси прискорення паралельного однокрокового k-точкового алгоритму, яке визначається як відношення часу виконання послідовного алгоритму до часу виконання паралельного, обчислюється за формулою

$$W(k) = \frac{T_s}{T_p} = \frac{(k+1)^2 \mathbf{t}_f + k^2(\mathbf{t}_{ad} + \mathbf{t}_{mul})}{k\mathbf{t}_f + 2k(\mathbf{t}_{ad} + \mathbf{t}_{mul}) + k(k-1)\mathbf{t}_{ta}}.$$

Оскільки у реальних системах час виконання операцій додавання й множення $\mathbf{t}_{ad}$ та $\mathbf{t}_{mul}$ і навіть час передачі даних сусідньому процесору $\mathbf{t}_{ta}$ значно менший від часу обчислення значення правої частини виразу (4.1) – $\mathbf{t}_f$, то, з урахуванням цього фактора, прискорення k-точкового паралельного алгоритму можна вважати таким, що приблизно дорівнює

$$W(k) \approx (k+1)^2 / k$$

і практично лінійно зростає зі збільшенням кількості процесорів у паралельній системі.

У випадку оцінки прискорення m-крокового k-точкового блокового методу правільніше буде виконувати порівняння часу його виконання на мультипроцесорній системі з часом виконання не алгоритму Рунге-Кутти, а з часом

виконання на однопроцесорному комп’ютері алгоритму m-крокового методу Адамса-Башфорта, який можна розглядати як багатокроковий одноточковий блоковий метод. При такому підході для обчислення приблизного розв’язку в k вузлах блоку буде потрібно k-кратне послідовне застосування формул Адамса-Башфорта. У тих же самих k вузлах блоку за k ітерацій паралельно можна обчислити приблизний розв’язок m-кроковим k-точковим блоковим методом. На перший погляд, у цьому випадку, створюється враження, що ніякого виграшу при застосуванні паралельного алгоритму немає, оскільки час обчислення буде приблизно однаковим для обох алгоритмів. Але тут потрібно врахувати той факт, що точність приблизного розв’язку, отриманого m-кроковим k-точковим блоковим методом, має порядок $O(\tau^{m+k})$, а точність приблизного розв’язку, отриманого за m-кроковою формулою Адамса-Башфорта, має порядок лише $O(\tau^{m+1})$. Таке розходження у точностях методів призводить до того, що для одержання розв’язку з однаковою точністю для методу Адамса-Башфорта крок сітки треба вибрати в $M^{(k-1)/(m+k)}$ разів менший, ніж крок для m-крокового k-точкового методу. У цій формулі параметр M – число вузлів сітки на відрізку розв’язання задачі методом Адамса-Башфорта.

Таким чином, прискорення паралельного m-крокового k-точкового алгоритму дорівнює

$$W(m,k) \approx M^{(k-1)/(m+k)}.$$

За аналогічними міркуваннями можна одержати оцінку ефективності розв’язання задачі Коші й для системи звичайних диференціальних рівнянь паралельними блоковими методами. Так, наприклад, для найбільш уживаної формули Адамса-Башфорта з $m = 4$ виходить таке значення прискорення

$$W(4,4) \approx M^{3/8}.$$

І тоді якщо на відрізку інтегрування міститься, наприклад, сто вузлів ($M = 100$), то коефіцієнт прискорення $W(4,4)=5.62$.

4.1 Питання і завдання для самоперевірки

1) Якими алгоритмами користуються для паралельного розв'язання задачі Коші для одиночного звичайного диференціального рівняння першого порядку $x' = f(x)$ при початковій умові $x(t_0) = x_0$? Які два типи паралельних методів для розв'язання звичайних диференціальних Ви знаєте? Коротко охарактеризуйте ці методи.
2) Розкажіть, які кроки роботи алгоритму виконуються при використанні обчислювальної схеми паралельного однокрокового блокового методу. Як виглядають різницеві рівняння для однокрокових блокових методів? Напишіть їх і ілюструйте відповідним графом розрахунку.
3) Чим багатокрокові блокові методи відрізняються від однокрокових блокових методів і як вони працюють (розкажіть алгоритм їхньої роботи)? Як в цьому разі виглядають різницеві рівняння багатокрокових різницевих методів для блоку з k точок? Напишіть їх і ілюструйте відповідним графом розрахунку.
4) Як за допомогою інтегроінтерполяційного методу визначаються коефіцієнти $a_{i,j}$ і $b_{i,j}$ у формулах різницевих рівнянь для обох типів різницевих методів?
5) Поясніть спосіб, за допомогою якого можна безпосередньо одержати ітераційні формули паралельного розв'язання системи різницевих рівнянь багатокрокового багатоточкового методу для довільного блоку n.
6) Зробить оцінку ефективності однокрокових блокових методів відповідного порядку на паралельній обчислювальній системі у порівнянні з часом виконання алгоритму Рунге-Кутти на однопроцесорному комп'ютері.
7) Як можна зробити оцінку прискорення m-крокового k-точкового блокового методу на мультипроцесорній системі з часом виконання на однопроцесорному комп'ютері алгоритму m-крокового методу Адамса-Башфорта? Чому дорівнює це прискорення?

5 ПАРАЛЕЛЬНІ МЕТОДИ ЧИСЕЛЬНОГО РОЗВ'ЯЗАННЯ ЖОРСТКИХ ЗДР ТА ЇХ РЕАЛІЗАЦІЯ В БАГАТОПРОЦЕСОРНИХ СТРУКТУРАХ

Останнім часом особлива увага при моделюванні різноманітних явищ навколишнього світу приділяється вирішенню важливого класу так званих ***жорстких систем диференціальних рівнянь***. Якщо представити систему диференціальних рівнянь у матричному вигляді $y = Ax$, то така система належить до жорстких при виконанні двох таких умов:

1) дійсні частини всіх власних значень матриці A негативні, тобто $Re(\lambda_k) < 0$, $(k = 0, 1, \ldots, n-1)$;
2) величина $s = \frac{\max|Re(\lambda_k)|}{\min|Re(\lambda_k)|}$, $(k = 0, 1, \ldots, n-1)$, яка зветься ***жорсткістю системи***, повинна бути великою.

Жорсткі системи вперше з'явилися при розв'язанні систем диференціальних рівнянь хімічної кінетики. Розв'язання таких систем представляється фрагментами, які дуже сильно відрізняються крутістю залежностей. Нерідко це трапляється й при аналізі електричних кіл з різко відмінними постійними часу.

Якщо крок розв'язання h є рівним або більшим від найменшої постійної часу розв'язку, то застосування стандартних методів (наприклад, Рунге-Кутти) з незмінним кроком призводить до великих похибок обчислень і навіть до розбіжності обчислювального процесу, у ході якого розв'язок грубо відмінний від існуючого.

Взагалі, розв'язання жорстких систем диференціальних рівнянь звичайними засобами не викликає особливих труднощів і може бути здійснено навіть при виборі не досить вдалого методу. Однак такий підхід до розв'язання системи рівнянь передбачає дуже значне зменшення кроку інтегрування, а це у свою чергу може призвести до виникнення таких неприємних ситуацій:

- різке зростання часу обчислень через надмірного зменшення кроку розв'язання;
- у ході дроблення кроку може виявитися перевищеним число ітерацій;

– для «особливо жорстких» систем адаптивний вибір кроку може не допомогти й похибка розв'язку буде великою.

Взагалі розв'язання систем звичайних диференціальних рівнянь великої розмірності, які використовуються при моделюванні різних динамічних процесів із зосередженими параметрами, належить до задач, для розв'язання яких можливостей існуючих засобів однопроцесорної обчислювальної техніки недостатньо.

Тому особливий інтерес викликають паралельні алгоритми чисельного розв'язання задачі Коші, які можуть застосовуватися для розв'язання, як звичайних, так і жорстких задач.

Методи, описані в попередньому розділі, можна використовувати й для розв'язання системи з m диференціальних рівнянь, яка має такий вигляд:

$$\begin{cases} y' = F(x, y); \\ y(x_0) = y_0 \end{cases}, \tag{5.1}$$

де $\bar{y} = (y_1, y_2, \ldots, y_m)^T$ – вектор розв'язку, а права частина рівнянь $F = \bar{f} = (f_1, f_2, \ldots, f_m)^T$ у загальному випадку – нелінійна функція, яка задає відображення $F: R \times R^m \to R^m$.

У цьому випадку однокроковий багатостадійний метод розв'язання нелінійної задачі Коші, для СЗДР (5.1), має вигляд:

$$\begin{aligned} \bar{y}_{n+1} &= \bar{y}_n + h\sum_{i=1}^{s} b_i \bar{k}_i \\ \bar{k}_i &= F\left(x_n + c_i h,\ \bar{y}_n + h\sum_{j=1}^{s} a_{ij}\bar{k}_j\right), \quad i = 1, \ldots, s \end{aligned},$$

де s-вимірні вектора $c = (c_1, c_2, \ldots, c_s)^T$, $b = (b_1, b_2, \ldots, b_s)^T$ і $s \times s$-вимірна матриця $A = (a_{ij})$, $i, j = 1, \ldots, s$, описують унікальний варіант методу й вибираються з міркувань точності. До того ж вигляд матриці A задає тип чисельної схеми, що покладається в основу методу. Якщо матриця A є трикутною так, що $a_{i,j} = 0$ при $i \le j$, то метод є явним і при наявності малого ступеня паралелізму, має

умовну стійкість. При повністю заповненій матриці утворюється цілком неявна схема, що застосовується для розв'язання жорстких задач.

Серед неявних чисельних схем можна, наприклад, розглянути два такі типи методів: ***блокові однокрокові методи*** й ***цілком неявні методи типу Рунге-Кутти*** (ЦНМРК). Достоїнствами цілком неявних, однокрокових методів загального вигляду (звичайно на основі квадратурних формул Радо й Лобатто) є добрі характеристики стійкості й точності, які достатні для розв'язання навіть жорстких задач. Недоліками цих методів є висока обчислювальна складність, яка обумовлюється ітераційним процесом визначення стадійних коефіцієнтів $\bar{k}_i$ ($i = 1, \ldots, s$)... Ця властивість методів не дозволяє ефективно застосовувати їх на послідовних машинах.

На відміну від явних методів, застосування ***цілком неявних багатостадійних методів*** для визначення коефіцієнтів $\bar{k}_i$ потребує розв'язання системи вимірністю $s \times m$, у загальному випадку нелінійних, рівнянь вигляду:

$$\begin{cases} \bar{k}_1 = \bar{f}\left(x_n + c_1 h, \bar{y}_n + h\left(a_{11}\bar{k}_1 + a_{12}\bar{k}_2 + \ldots + a_{1s}\bar{k}_s\right)\right) \\ \bar{k}_2 = \bar{f}\left(x_n + c_2 h, \bar{y}_n + h\left(a_{21}\bar{k}_1 + a_{22}\bar{k}_2 + \ldots + a_{2s}\bar{k}_s\right)\right) \\ \ldots \\ \bar{k}_s = \bar{f}\left(x_n + c_s h, \bar{y}_n + h\left(a_{s1}\bar{k}_1 + a_{s2}\bar{k}_2 + \ldots + a_{ss}\bar{k}_s\right)\right) \end{cases}.$$

Розв'язання цієї системи можна здійснити з використанням такої ітераційної схеми:

$$\begin{cases} \bar{k}_i^0 = \bar{f}\left(x_n + c_i h, \bar{g}_i^0\right), \\ \bar{k}_i^{l+1} = \bar{f}\left(x_n + c_i h, \bar{g}_i^l\right) \end{cases},$$

де $\bar{g}_i^0 = \bar{y}_n$, $\bar{g}_i^l = \bar{y}_n + h\sum_{j=1}^{s} a_{ij}\bar{k}_j^l$, при $i = 1, \ldots, s$ та $l = 1, \ldots, N$...

У цій схемі ітераційний процес, повторений l раз, представляє $\bar{k}_i^l$ при $i = 1, \ldots, s$, як l-ту апроксимацію для крокового коефіцієнта $\bar{k}_i$.

Швидкість збіжності такого ітераційного процесу залежить від порядку цілком неявного багатостадійного методу, який використовується, і визначається як: $r^{*} = \min(r, N+1)$, де r – порядок багатостадійного методу.

Є цілком очевидним, що блокові або багатоточкові неявні методи розв'язання задачі Коші особливо актуальні, оскільки добре узгоджуються з архітектурою паралельних обчислювальних систем і не потребують обчислення значень у проміжних точках, що значно підвищує ефективність розрахунків. Ці методи мають високу стійкість і, що є найбільш істотним, є первісно паралельними, тому що дозволяють одержувати розв'язок одночасно в декількох точках сітки інтегрування одночасно.

Як і раніш безліч точок рівномірної сітки $\Omega_h:\{x_j\}$ при $j = 1, \ldots, M$ розбивається на N блоків, кожен з яких містить k точок. Рівняння однокрокових блокових різницевих методів у застосуванні до ЗДР для блоку з k точок можна записати у вигляді:

$$y_{n,i} = y_{n,0} + ih\left(b_i F_{n,0} + \sum_{j=1}^{k} a_{i,j} F_{n,j}\right);\ i = 1, \ldots, k;\ n = 1, \ldots, N.$$

Нескладними математичними перетвореннями можна показати, що однокроковий k-точковий блоковий метод має найвищий порядок апроксимації який дорівнює $k+1$, з цього випливає, що локальна помилка у вузлах блоку має порядок $O(h^{k+2})$. Як було сказано вище, блокові паралельні методи належать до класу неявних, тому для обчислення приблизних значень розв'язання задачі Коші необхідно розв'язати систему нелінійних рівнянь.

Одним зі способів одержання розв'язку є ***метод простої функціональної ітерації***:

$$\begin{cases} y_{n,i,0} = y_{n,0} + ihF_{n,0}, & i = 1, \ldots, k,\ n = 1, 2, \ldots, N, \\ y_{n,i,l+1} = y_{n,0} + ih\left(b_i F_{n,0} + \sum_{j=1}^{k} a_{i,j} F_{n,j,l}\right), & l = 0, \ldots, L-1 \end{cases},$$

де $F_{n,i} = f(x_{n,i}, y_{n,i})$;

n – номер блоку $n = 1, 2, \ldots, N$;

i – номер точки блоку, $i = 1, \ldots, k$;

l – номер поточної ітерації $l = 0, \ldots, L - 1$;

L – максимальне число ненульових ітерацій.

Для мінімізації обчислювальних витрат і досягнення деякої заданої точності наближеного розв’язку, чисельний алгоритм повинен мати механізм управління кроком інтегрування. Це можливо на основі інформації про так звану ***апостеріорну локальну похибку***.

Але, на відміну від явних методів розв’язання СЗДР, реалізація альтернативних способів оцінки апостеріорної локальної похибки на основі блокових методів пов’язана з рядом особливостей:

- немає відповідних послідовних аналогів;
- зміна кроку інтегрування можлива тільки після обчислення всіх значень в k вузлах поточного n-го блоку;
- за умови незадовільної оцінки локальної похибки й необхідності зміни кроку інтегрування практично всі обчислення для точок блоку виявляться даремними (хоча деякі звертання до правої частини можуть бути використані знову).

Тому для реалізації поставленої мети рекомендується використовувати альтернативні способи оцінки похибки на кроці:

- дублювання кроку за правилом Рунге;
- вкладені пари схем;
- технологію локальної екстраполяції Річардсона.

Для оцінки ефективності розпаралелювання (абсолютного коефіцієнта прискорення) у порівнянні з відповідними послідовними алгоритмами можна провести порівняльний аналіз двох класів неявних методів при послідовній і паралельній реалізаціях на основі найбільш ефективного способу оцінки локальної похибки – вкладених формул.

При такому аналізі основними параметрами, які характеризують цілком

неявні методи Рунге-Кутти, є: порядок методу r і число стадій s, причому ці величини є взаємозалежними. Крім того, при розв'язанні систем нелінійних алгебраїчних рівнянь для визначення стадійних коефіцієнтів з'являється такий параметр, як необхідне число ітерацій: L. Відповідно, обчислювальна складність блокових методів залежить від порядку методу r, числа точок у блоці k і також числа ітерацій для одержання розв'язку СНАР необхідної точності: $\hat{L}$.
Для коректного порівняння чисельних методів на базі неявних однокрокових схем необхідно, щоб виконувалися такі умови:

- забезпечення того самого порядку точності, r;
- одержання розв'язку в однаковому числі нових точок, k;
- породжувані ітераційні процеси повинні реалізовувати граничне число ітерацій, які передбачені кожним з розглянутих методів: $L = 2s$ і $\hat{L} = k$.

Теореми Батчера, для широко застосовуваних цілком неявних методів Рунге-Кутти, зв'язують число стадій з порядком методу одним з таких співвідношень: для методу Гаусса $r = 2s$; для методів Радо $r = 2s - 1$; і для методів Лобатто $r = 2s - 2$. Цілком очевидно, що для ЦНМРК найкращим варіантом буде значення $r = 2s$. У той же час для блокових однокрокових методів порядок можна визначити через число точок одного блоку: $r = k + 1$. Тоді, число стадій ЦНМРК і число точок у блоці багатоточкового методу пов'язані між собою таким співвідношенням:

$$r = 2s = k + 1 \Rightarrow k = 2s - 1.$$

Порівняння найкраще здійснювати, ґрунтуючись на отриманих співвідношеннях, приводячи всі часові характеристики до одного параметра, наприклад, до числа стадій s і вибираючи для порівняння цілком неявні методи з усієї множини неявних методів типу Рунге-Кутти. Такий вибір найбільш виправданий тому, що ЦНМРК мають характеристики стійкості, ідентичні характеристикам стійкості блокових однокрокових методів. Крім того вони забезпечують оптимальне співвідношення між порядком і числом стадій методів.

Проведене з урахуванням сказаного порівняння часів виконання послідо-

вних алгоритмів методів, які описані вище, дозволяє зробити такі висновки. При досить складній правій частині рівнянь системи й великих витратах часу для обчислень при звертанні до правої частини ЗДУ, повний обсяг обчислень для ЦНМРК в s раз перевищує обсяг обчислень, які виконуються при використанні блокових алгоритмів. Порівняння виконання алгоритмів на паралельних системах при використанні ієрархічної декомпозиційної методики й програмної реалізації виконаної з використанням стандарту передачі повідомлень MPI так само показує більшу швидкодію блокових алгоритмів у порівнянні із ЦНМРК, але вже в $2s$ разів.

У випадку ж з відносно простою правою частиною ЗДУ основний обсяг обчислювальної роботи як при послідовній, так і при паралельній організації обумовлюється обчисленням розв’язку в нових k точках сітки інтегрування й потребує більших накладних витрат для неявних методів Рунге-Кутти в порівнянні із блоковими методами. Причому для паралельних алгоритмів ця різниця збільшується майже в 2 рази.

Зі сказаного вище можна зробити висновок, що для будь-яких способів оцінки локальної похибки блокові багатоточкові однокрокові методи є найбільш ефективними з погляду обчислювальних витрат у порівнянні з цілком неявними методами Рунге-Кутти як для розв’язання СЗДР, так і для розв’язання систем із жорсткими диференціальними рівняннями.

5.1 Питання і завдання для самоперевірки

1) Назвіть умови, при виконанні яких система диференціальних рівнянь вигляду $y = Ax$ належить до жорстких систем диференціальних рівнянь. Що таке жорсткість системи? Чому неможливе застосування стандартних методів Рунге-Кутти з незмінним кроком для розв’язання подібних систем?
2) Які неприємні ситуації виникають при розв’язанні жорстких систем диференціальних рівнянь звичайними засобами при значному зменшенні кроку інтегрування? Чому бажано розв’язувати такі системи на багатопроцесорних комплексах?

3) Чи існує можливість для розв'язання системи з m диференціальних рівнянь, яка має такий вигляд:

$$\begin{cases} \bar{y}' = F(x, \bar{y}); \\ \bar{y}(x_0) = \bar{y}_0 \end{cases},$$

де $\bar{y} = (y_1, y_2, \ldots, y_m)^T$ – вектор розв'язків, а права частина рівнянь $F = \bar{f} = (f_1, f_2, \ldots, f_m)^T$ у загальному випадку – нелінійна функція, яка задає відображення $F : R \times R^m \to R^m$, використовувати методи, описані в попередньому розділі (див. розділ 4)?

4) За якої умові для розв'язання жорстких задач утворюється і застосовується цілком неявна схема розв'язання? Які два типи методів для розв'язку неявних чисельних схем Ви знаєте? Що не дозволяє ефективно застосовувати ці методи на послідовних машинах?

5) Що відрізняє застосування цілком неявних багатостадійних методів для визначення коефіцієнтів $\bar{k}_i$ від відповідних явних методів? Наведіть і поясніть алгоритм ітераційної схеми розв'язання системи ЗДР при застосуванні таких методів. Як визначається і від чого залежить швидкість збіжності такого ітераційного процесу?

6) Чому для розв'язання задачі Коші особливо актуальними є блокові або багатоточкові неявні методи? Поясніть алгоритм методу простої функціональної ітерації для обчислення приблизних значень розв'язання задачі Коші. Чим відрізняється оцінка апостеріорної локальної похибки при застосуванні блокових методів від її оцінки при реалізації явних методів розв'язання СЗДР?

7) Виконайте оцінку ефективності розпаралелювання (абсолютного коефіцієнта прискорення) у порівнянні з відповідними послідовними алгоритмами. Який висновок можна зробити відносно ефективності блокових багатоточкових однокрокових методів у порівнянні з цілком неявними методами Рунге-Кутти для розв'язання систем із жорсткими диференціальними рівняннями з точки зору оцінки локальної похибки?

6 ПАРАЛЕЛЬНІ АЛГОРИТМИ ЧИСЕЛЬНОГО РОЗВ'ЯЗАННЯ ЗАДАЧІ ПУАССОНА

Моделювання значної кількості задач фізики й техніки призводить до диференціальних рівнянь у частинних похідних (рівняння математичної фізики).

Характерним (типовим) прикладом складної обчислювальної задачі є задача про комп'ютерне моделювання клімату (зокрема, задача метеорологічного прогнозування). В основі кліматичної моделі лежать рівняння суцільного середовища й рівняння рівноважної термодинаміки.

Найпростіша модель, яка моделює погоду в приземному сферичному шарі Σ, представляє з себе систему нелінійних рівнянь у частинних похідних у тривимірному просторі й складається з таких рівнянь:

– рівняння кількості руху $\frac{D\bar{v}}{Dt} = -\frac{1}{\rho}\nabla p + g - 2\bar{\Omega} \times \bar{v}$, де p і ρ – відповідно тиск і густина повітря, g – прискорення вільного падіння, $\bar{\Omega}$ – вектор кутової швидкості обертання Землі, $\bar{v}$ – вектор швидкості вітру;

– рівняння збереження енергії $c_p \frac{DT}{Dt} = \frac{1}{\rho}\frac{Dp}{Dt}$, де c_p – питома теплоємність;

– рівняння нерозривності середовища $\frac{D\rho}{Dt} + \mathrm{div}(\rho\bar{v}) = 0$;

– рівняння стану середовища $p = \rho RT$, де R – газова стала.

Тут уведене позначення $\frac{D}{Dt} = \frac{\partial}{\partial t} + v_x \frac{\partial}{\partial t} + v_y \frac{\partial}{\partial t} + v_z \frac{\partial}{\partial t}$.

Її розв'язання досить приблизно відтворює всі головні характеристики ансамблю станів кліматичної системи. Фактично, це система шести нелінійних скалярних рівнянь щодо шести невідомих функцій (які залежать від трьох координат $(x, y, z) \in \Sigma$ і часу t), а саме, щодо компонентів v_x, v_y, v_z вектора швидкості $\bar{v}$ й функцій p, ρ, T. До цих рівнянь приєднуються початкові й граничні умови.

Обчислювальну складність такої моделі стану атмосфери можна якісно уявити собі хоча б з того факту, що час виконання одного чисельного експерименту на комп'ютері із продуктивністю 10^{12} флопсів оцінюється величиною по-

рядку 30 годин. На практиці, через відсутність точної інформації про початкові й про крайові умови, потрібно прорахувати сотні, а то й тисячі подібних варіантів. При цьому необхідно відзначити, що повна кліматична модель у дійсності набагато складніша й розрахунок за нею займе ще на порядок більше часу.

Зі сказаного зрозуміло, що для подібних експериментів потрібні досить потужні паралельні обчислювальні системи і відповідні паралельні алгоритми. Взагалі кажучи, розв'язання диференціальних рівнянь у частинних похідних є ядром багатьох додатків тому, що моделі переважної кількості природних і соціальних явищ зводяться до розв'язання такої задачі.

Взагалі сталі процеси різної фізичної природи описуються рівняннями еліптичного типу. Точні рішення крайових задач для еліптичних рівнянь вдається одержати лише в окремих випадках. Тому ці задачі вирішують в основному приблизно чисельними методами. Одним з найбільш універсальних і ефективних методів, що набули в цей час значного поширення для приблизного розв'язання рівнянь математичної фізики, є метод кінцевих різниць або метод сіток.

Суть методу полягає в такому – область безперервної зміни аргументів заміняється сіткою або ґратами тобто дискретною множиною точок (вузлів). Замість функції безперервного аргументу розглядаються функції дискретного аргументу, визначені у вузлах цієї сітки. Такі функції ще називають ***сітковими функціями***. Похідні, що входять у диференціальне рівняння й граничні умови, заміняються різницевими похідними, при цьому крайова задача для диференціального рівняння заміняється системою лінійних або нелінійних алгебраїчних рівнянь (сіткових або різницевих рівнянь). Такі системи часто називають різницевими схемами. І ці схеми вирішуються щодо невідомої сіткової функції.

Існує велика кількість паралельних алгоритмів розв'язання задачі Пуассона, але як приклад доцільно розглянути паралельну адаптацію звичайного послідовного алгоритму на скінченно-різницевій обчислювальній сітці методом Якобі завдяки простоті його обчислювальної частини. Крім того такий підхід дає змогу продемонструвати можливості бібліотеки MPI у використанні віртуа-

льних топології для більш зручного розміщення процесів у обчислювальному середовищі, а також дозволяє дослідити ефективність різних способів комунікації для розв'язання однієї й тієї ж задачі.

Як відомо, задача Пуассона, тобто розв'язання диференціальних рівнянь частинних похідних, виражається такими рівняннями:

$$\Delta^2 u = f(x,y) \text{ – у середині області} \tag{6.1}$$

$$u(x, y) = k(x, y) \text{ – на границі} \tag{6.2}$$

Якщо обмежитися прямокутною областю вирішення, то для знаходження приблизного розв'язку рівняння (6.1) треба визначити прямокутну сітку, точки якої мають координати x_i, y_i і задаються такими рівняннями:

$$x_i = \frac{i}{n+1},\ i = 0, \ldots, n+1, \qquad y_j = \frac{j}{n+1},\ j = 0, \ldots, n+1.$$

Таким чином, уздовж кожного краю сітки є $n + 2$ точки. За результатами розв'язання необхідно у точках (x_i, y_j) обраної сітки знайти апроксимацію для функції $u(x, y)$. Якщо позначити значення u в (x_i, y_j) через $u_{i,j}$, а відстань між точками через $h = 1/(n + 1)$, тоді формула (6.1). для кожної із точок буде виглядати таким чином:

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} \quad 4u_{i,j}}{h^2} = f_{i,j}. \tag{6.3}$$

Переходячи до ітераційного процесу, поточні значення $u_{i,j}$ у кожній точці сітки обчислюються шляхом заміни попередніх за виразом:

$$u_{i,j}^{k+1} = \left(u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} + u_{i+1,j}^{k} - h^2 f_{i,j}\right)/4.$$

Це й є ітерації Якобі, які повторюються до одержання розв'язку. Фрагмент звичайної послідовної програми для цього випадку може мати такий вигляд:

```
 integer i, j, n
 double precision, u(0:n+1, 0:n+1), unew(0:n+1, 0:n+1)
 do 10 j = 1, n
 do 10 i = 1,n
 unew (i, j) = 0.25*(u(i-1, j) + u(i, j+1) + u(i, j-1) +
&u(i+1, j)) - h * h * f(i, j)
10 continue
```

Як відмічалось раніш, найбільший запас паралелізму міститься у циклах, тому, щоб перетворити послідовний алгоритм на паралельний, потрібно розподілити дані, тобто u, unew, f по процесах. Одна з найпростіших декомпозицій полягає в такому: фізична область поділяється на шари, кожен з них обробляється окремим процесом. На рис. 6.1 частково (для точок 1 і 2) наведений граф інформаційних залежностей, а також пунктирною лінією позначені шари для кожного з трьох процесорів. Програмно ж цю декомпозицію можна описати в такий спосіб:

```
 integer i, j, n, s, e
 double precision u(0:n+1, s:e), unew(0:n+1, s:e)
 do 10 j = s, e
 do 10 i = 1,n
 unew(i, j) = 0.25*(u(i-1, j)+ u(i, j+1)+ u(i, j-1) +
&u(i+1,j)) - h * h * f(i,j)
10 continue
```

Тут s, e указують початкове і кінцеве значення номерів рядків шару, які належать тому чи іншому процесу.

З рисунку добре видно, що для точок в рядках для яких $s < j < e$ (точка 1), кожен з процесів може вирахувати нове значення, ґрунтуючись на попередніх значеннях зі свого власного шару. Але для точки 2, при виконанні ітерації, процес повинен використовувати вже не тільки елементи, що належать його шару, але й елементи із суміжних процесів. Тобто вже необхідно виконувати пересилання даних між сусідніми процесами, від процесора 2 до процесора 1, так само як і в протилежному напрямку. Для одержання і зберігання пересланих даних треба зробити розширення робочих масивів (шарів сітки) кожного процесу так, як це зображено на рис. 6.1 контуром із суцільною лінією. Елементи середовища, які використовуються, щоб зберігати дані з інших процесів, називаються

тіньовими.

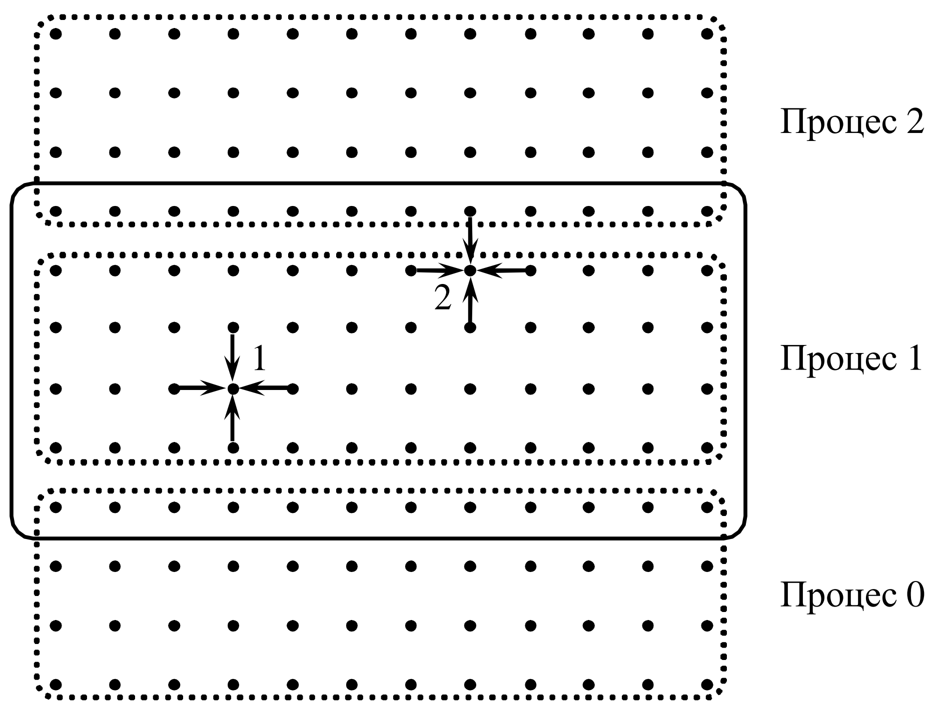


Рисунок 6.1 − Область із тіньовими точками для процесу 1

Для створення тіньових елементів треба просто змінити оголошення вимірності масиву *u* так, як це зроблено у такому коді:

```
double precision u(0:n+1, s-1: e+1).
```

Крім розподілу даних, треба кожному процесору правильно назначити його сусідів зверху і знизу. Можна було б спробувати назначити сусідів додавши або віднявши одиницю до власного номера процесу в комунікаторі. Але у цьому випадку вибір, скоріш за все, буде не найкращим, оскільки такі сусіди можуть не бути сусідами по апаратурі. Природно, що виникає бажання якимось чином упорядкувати процеси і так назначити їм номера, щоб вони дійсно були сусідами, як за номером, так і по апаратурі. Допомогти добитись цього може набір процедур бібліотеки MPI для створення, дослідження й маніпулювання віртуальними картезіанськими топологіями.

Наприклад, в описуваному випадку можна за допомогою функції `MPI_CART_CREATE` створити одновимірну (1D) віртуальну картезіанську тополо-

гію. Але й у цьому разі знаходження сусіда шляхом додавання або віднімання одиниці від власного номера процесу уже в новому комунікаторі також не є ефективним.

І знов тут може допомогти набір функцій бібліотеки MPI. Справа в тому, що у вибраному алгоритмі кожен процес є як приймачем, так і відправником даних. Оскільки копія верхнього рядка одного процесу є дном тіньового рядка процесу, розташованого вище, то фактично виконується зсув даних від одного процесу до іншого, а MPI має процедуру MPI_CART_SHIFT, яку можна використати, щоб знайти сусідів. Таким чином можна створити деяку власну процедуру MPE_DECOMP1D, яка обумовлює декомпозицію масивів і викликається функцією

```
call MPE_DECOMP1D(n, nprocs, myrank, s, e),
```

де nprocs – число процесів у картезіанській топології, myrank – координати процесу у новому комунікаторі, n – вимірність масиву.

MPE_DECOMP1D обчислює значення s і e, наприклад, у такий спосіб.

Якщо n ділиться на nprocs без залишку, то

```
s = 1 + myrank * (n/nprocs)
e = s + (n/nprocs) - 1,
```

у іншому випадку, коли залишок не дорівнює 0, виконуються такі дії:

```
s = 1 + myrank * floor(n/nprocs)
if (myrank .eq. nprocs - 1) then e = n
else e = s + floor(n/nprocs) - 1
endif,
```

де функція floor(x) – функція округлення з надлишком.

Для кожного процесу повинні бути одержані тіньові дані для рядка з номером s – 1 від процесу, розташованого нижче, й дані для рядка з номером e + 1 від процесу, розташованого вище. Черговим кроком є створення процедури обміну даними, до якої передаються такі параметри: a – масив значень у точках сітки області розв’язання, nx – кількість даних у рядку масиву a, який пе-

ресилається, s – номер першого рядка масиву для даного процесу, e – номер його останнього рядка для процесу, nbrbottom – номер процесу, розташованого нижче від даного, а nbrtop – номер процесу, розташованого вище, comm1d – ідентифікатор картезіанського комунікатора.

```
      subroutine EXCHG1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
      use mpi
      integer nx, s, e
      double precision a(0:nx+1, s-1:e+1)
      integer comm1d, nbrbottom, nbrtop, status(MPI_STATUS_SIZE),
     &ierr
      call MPI_SEND(a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
     &comm1d, ierr)
      call MPI_RECV(a(1,s-1), nx, MPI_DOUBLE_PRECISION,
     &nbrbottom, 0, comm1d, status, ierr)
      call MPI_SEND(a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom,
     &1,comm1d, ierr)
      call MPI_RECV(a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
     &comm1d, status, ierr)
      return
      end
```

У цій процедурі кожен процес за допомогою функції MPI_SEND посилає дані процесу nbrtop, розташованому вище за нього, і потім функцією MPI_RECV приймає дані від нижнього процесу nbrbottom. Потім порядок змінюється на зворотний – дані посилаються до нижчого процесу й приймаються від процесу, вищого від даного.

Підсумовуючи сказане вище, можна вже описати роботу основної частини головної програми для розв’язання задачі Пуассона. Приблизний код (без операцій ініціалізації і завершення паралельних процесів) цієї частині програми може виглядати таким чином.

```
C     визначення нового комунікатора картезіанської топології
C     вимірності 1 для декомпозиції процесів
      call MPI_CART_CREATE(MPI_COMM_WORLD, 1, numprocs, .false.,
     &.true., comm1d, ierr)
C     визначення позиції процесу в одержаному комунікаторі
      call MPI_COMM_RANK(comm1d, myid, ierr)
C     визначення номерів процесів для заповнення тіньових точок
      call MPI_CART_SHIFT(comm1d, 0, 1, nbrbottom, nbrtop, ierr)
```

```
C      визначення декомпозиції вихідного масиву
       call MPE_DECOMP1D(ny, numprocs, myid, s, e)
C      ініціалізація масивів f та a
       call ONEDINIT(a, b, f, nx, s, e)
C      цикл обчислення ітерацій Якобі, maxit - максимальна
C      кількість ітерацій
       do 10 it = 1, maxit
C      визначення тіньових точок
       call EXCHNG1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
C      тут кожен процес обчислює одну звичайну послідовну
C      ітерацію Jacobi для своєї малнької частини від загальної
C      області розв'язання
       call SWEEP1D(a, f, nx, s, e, b)
C      повторюється, для одержання розв'язку в a
       call EXCHNG1(b, nx, s, e, comm1d, nbrbottom, nbrtop)
C      знов паралельно всіма процесами виконується ітерація
C      Jacobi
       call SWEEP1D(b, f, nx, s, e, a)
C      перевірка точності обчислень
       diffw = DIFF(a, b, nx, s, e)
       call MPI_ALLREDUCE(diffw, diffnorm, 1, MPI_DOUBLE_PRECISION,
     &MPI_SUM, comm1d, ierr)
       if (diffnorm .lt. 1.0e-5) goto 20
       if (myid .eq. 0) print *, 2*it, ' Difference is ', diffnorm
   10 continue
       if (myid .eq. 0) print *, 'Failed to converge'
   20 continue
       if (myid .eq. 0) then print *, 'Converged after ', 2*it,
     &' Iterations'
       endif
```

Тут для призначення процесів розділеним областям масиву застосовується картезіанська топологія одиничної вимірності, для чого використовується процедура MPI_CART_CREATE. Потім кожен з процесів в одержаному комунікаторі виконує процедуру MPE_DECOMP1D визначення своєї частки масиву. Не описана раніш процедура ONEDINIT просто якимсь чином (обчислення або введення з файла) у кожному з процесів виконує ініціалізацію відповідної частки елементів масивів a та f. Безпосередньо розв'язання на такій ітерації обчислюється поперемінно в масивах a й b, тому що в циклі є два звертання до EXCHNG1 і SWEEP1D (це тривіальна процедура обмину значень між масивами a та b). Ітерації закінчуються, коли різниця між двома суміжними значеннями апроксимацій, яка обчислюється функцією DIFF стає меншою за 10^{-5}. Застосування бібліотечної процедури MPI_ALLREDUCE потрібне для гарантування, що у всіх проце-

сах досягнута необхідна точність обчислень. В процесі обчислень програма друкує поточні значення досягнутої точності на відповідній ітерації, а на закінчення повідомляє про завершення обчислень із зазначенням кількості виконаних ітерацій. Цикл DO з максимумом ітерацій maxit гарантує, що програма закінчиться, навіть якщо ітерації не сходяться.

У результаті вдалого вибору декомпозиції даних і знаходження сусідів для кожного процесу, в паралельному одновимірному алгоритмі розв'язання задачі Пуассона вдається значно прискорити досягнення розв'язання. Але в цьому разі кожному процесору доводиться виконувати доволі багато обчислень через велику кількість точок по одній з координат. Однак і тут бібліотека MPI надає можливість прискорити знаходження розв'язку, завдяки створенню двовимірної віртуальної топології в системі процесорів.

Для цього потрібна доволі невелика модифікація програми, яка перетворює її з одновимірної на двовимірну. 2D картезіанська топологія в цьому випадку також створюється за допомогою функції MPI_CART_CREATE у такій спосіб:

```
 isperiodic(1) = .false.
 isperiodic(2) = .false.
 reorder = .true.
 call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, isperiodic,
&reorder, comm2d, ierr)
```

Так само як і раніш при знаходженні процесів-сусідів зверху та знизу, процеси-сусіди ліворуч nbrleft і праворуч nbrright визначаються за допомогою функції MPI_CART_SHIFT таким чином:

```
call MPI_CART_SHIFT(comm2d, 0, 1, nbrleft, nbrright, ierr)
call MPI_CART_SHIFT(comm2d, 1, 1, nbrbottom, nbrtop, ierr)
```

Дещо змінюється і процедура SWEEP, яка обчислює нові значення масиву unew:

```
 integer i, j, n
 double precision u(sx-1:ex+1, sy-1:ey+1), unew(sx-1:ex+1, &sy-
1:ey+1)
 do 10 j = sy, ey
```

```
    do 10 i = sx, ex
    unew(i, j) = 0.25*(u(i - 1, j) + u(i, j + 1) + u(i, j - 1) +
   &u(i + 1, j)) - h * h * f(i, j)
 10 continue
```

Залишається лише переробити, модернізувати процедуру обміну даними, але зробити це так само легко, як в для двох попередніх процедур не вдасться. Проблема пов'язана з тим, що в даному разі дані, які підлягають пересиланню до процесів праворуч і ліворуч, не зберігаються в безперервних відрізках пам'яті, як це було для пересилань уверх і униз.

Справа в тому, що коли повідомлення займають безперервну область пам'яті, то для їх уміщування у буфер пересилання у відповідних функціях MPI достатньо вказати кількість даних, які підлягають пересиланню, та їх тривіальний тип, наприклад, MPI_INTEGER або MPI_DOUBLE_PRECISION. В інших випадках необхідно визначати новий тип даних, який описує групу елементів, адреси яких в пам'яті відстоять одна відносно іншої на деяку постійну величину (страйд). Для цього до бібліотеки MPI включені спеціальні функції, використання яких у даному випадку відбувається таким чином.

```
    call MPI_TYPE_VECTOR(ey - sy + 4, 1, ex - sx + 4,
   &MPI_DOUBLE_PRECISION, stridetype, ierr)
    call MPI_TYPE_COMMIT(stridetype, ierr)
```

В цьому фрагменті програми за допомогою функції MPI_TYPE_VECTOR створюється новий тип даних stridetype, який описує блок даних, що складається з декількох блоків копій вхідного типу даних. Перший аргумент процедури – це кількість таких часткових блоків; другий – кількість елементів старого типу в кожному блоці (часто це один елемент); третій аргумент є страйд (дистанція в термінах вхідного типу даних між послідовними елементами); четвертий аргумент – це оригінальний тип даних; п'ятий аргумент – ідентифікатор створюваного нового типу.

Створений новий тип даних реєструється у системі за допомогою процедури MPI_TYPE_COMMIT для можливості подальшого використання. Всі сконструйовані користувачем типи даних повинні бути зареєстровані у системі до їх-

нього використання в операціях обміну. Коли тип даних більше не потрібен, він повинен звільнятись процедурою MPI_TYPE_FREE. Після визначення нових типів даних програма для передачі рядка відрізняється від програми для передачі стовпця тільки в аргументах типу даних. Кінцева версія процедури обміну EXCHNG2 представлена нижче.

```
 subroutine EXCHNG2 (a, sx, ex, sy, ey, comm2d, stridetype,
&nbrleft, nbrright, nbrtop, nbrbottom)
 use mpi
 integer sx, ex, sy, ey, stridetype, nbrleft, nbrright,
&nbrtop, nbrbottom, comm2d
 double precision a(sx-1:ex+1, sy-1:ey+1)
 integer status(MPI_STATUS_SIZE), ierr, nx
 nx = ex - sx + 1
 call MPI_SENDRECV(a(sx,ey), nx, MPI_DOUBLE_PRECISION,
&nbrtop, 0, a(sx, sy-1), nx, MPI_DOUBLE_PRECISION, nbrbottom,
&0, comm2d, status, ierr)
 call MPI_SENDRECV(a(sx,sy), nx, MPI_DOUBLE_PRECISION,
&brbottom, 1, a(sx,ey+1), nx, MPI_DOUBLE_PRECISION, nbrtop,
&1, comm2d, status, ierr)
 call MPI_SENDRECV(a(ex,sy), 1, stridetype, nbrright, 0,
&a(sx-1,sy), 1, stridetype, nbrleft, 0,comm2d, status, ierr)
call MPI_SENDRECV(a(sx,sy), 1, stridetype, nbrieft, 1,
&a(ex+1,sy), 1, stridetype, nbrright, 1, comm2d, status, ierr)
return
 end
```

У ній визначені такі параметри: a – масив, nx – кількість даних у рядку, який пересилається, sx – номер першого рядка масиву в даному процесі, ex – номер останнього рядка масиву даного процесу, sy – номер першого стовпця масиву в даному процесі, ey – номер останнього стовпця масиву даного процесу, nbrbottom – номер процесу-сусіда нижчого, nbrtop – номер процесу-сусіда вищого, nbrleft – номер процесу-сусіда ліворуч, nbrright – номер процесу-сусіда праворуч, stridetype – новий тип даних для елементів стовпця.

На рис. 6.2 представлена UML-діаграма взаємодії процесів усередині процедури EXCHG1, яка застосовує найбільш часто використовуваний метод обміну даними – послідовне виконання процедур з відправлення і прийому повідомлень з блокуванням (MPI_SEND і MPI_RECV). При такім методі спілкування процесів паралелізм великою мірою залежить від розміру буфера передачі по-

відомлень, який повністю забезпечується системою. Коли програма запускається у системі з малим об'ємом буферного простору й з великим розміром повідомлень, то посилки не завершаються доти, поки приймаючі процеси не закінчать прийом. У процедурі EXCHG1 тільки один останній процес не посилає повідомлень на першому кроці, отже, він може одержати повідомлення від процесу вищого, потім відбувається прийом у попередньому процесі й так далі. З діаграми добре видно, що процесор P0 доволі тривалий час простоює, чекаючи поки він зможе отримати і обробити посилання, яке адресується йому. Цілком очевидно, що це явище обмежує можливість дійсно паралельного находження розв'язку завдання.

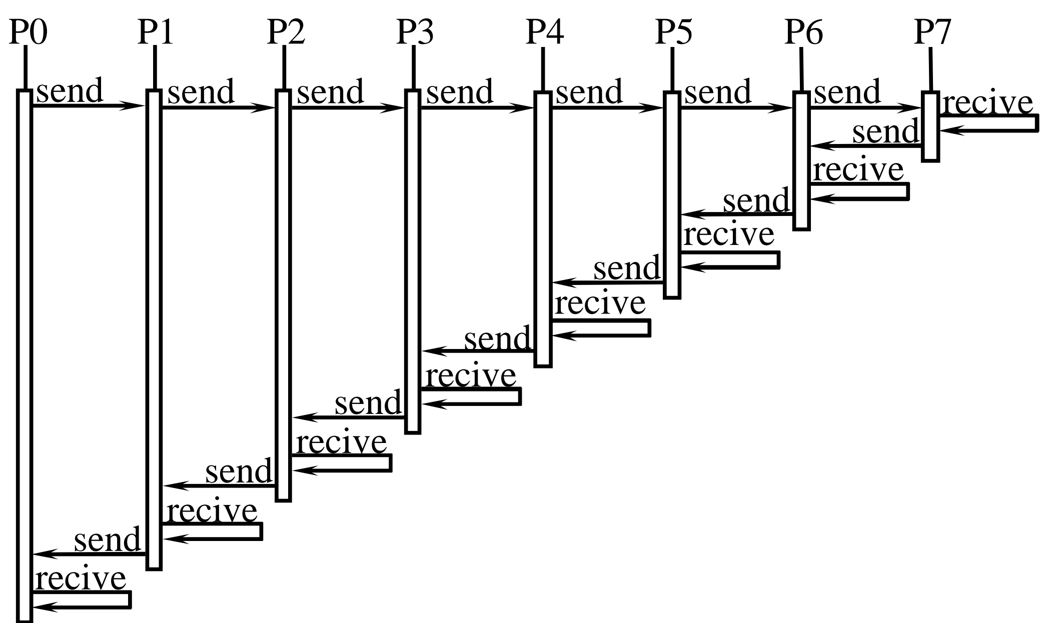


Рисунок 6.2 – UML-діаграма взаємодії процесів у процедурі EXCHG1

Задля усунення такого небажаного явища, у бібліотеці передбачена низка процедур, які тим або іншим способом зменшують додаткові витрати, пов'язані з наявністю подібних послідовних ланок паралельної програми. Тому має сенс розглянути застосування деяких з цих процедур у контексті поставленої задачі. Розгляд найдоцільніше виконати на прикладі процедури EXCHG1 завдяки її відносної простоти.

Найпростіший шлях скорегувати залежності,які виникають через буфері-

зацію, полягає в упорядкуванні посилок і прийомів таким чином, щоб приймаючий процес завжди спочатку виконував прийом, а тільки потім починав свою посилку, якщо вона в нього є. Це так званий ***упорядкований*** прийом/передача.

В цьому разі застосовуються ті ж самі процедури MPI_SEND і MPI_RECV, що і в попередньому випадку, але просто в іншій послідовності. Відповідна програма представлена нижче.

```
 subroutine EXCHNG1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
 use mpi
 integer nx, s, e, comm1d, nbrbottom, nbrtop, rank, coord
 double precision a(0:nx+1, s-1:e+1)
 integer status (MPI_STATUS_SIZE), ierr
 call MPI_COMM_RANK(comm1d, rank, ierr)
 call MPI_CART_COORDS(comm1d, rank, 1, coord, ierr)
 if (mod(coord, 2) .eq. 0) then
 call MPI_SEND(a(1,e), nx, MPI_DOUBLE_PRECISION,nbrtop, 0,
&comm1d, ierr)
 call MPI_RECV(a(1, s-1), nx, MPI_DOUBLE_PRECISION,
&nbrbottom, 0, comm1d, status, ierr)
 call MPI_SEND(a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom,
&1, comm1d, ierr)
 call MPI_RECV(a(1, e+1), nx, MPI_DOUBLE_PRECISION, nbrtop,
&1, comm1d, status, ierr)
 else
 call MPI_RECV(a(1, s-1), nx, MPI_DOUBLE_PRECISION,
&nbrbottom, 0, comm1d, status, ierr)
 call MPI_SEND(a(1, e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
&comm1d, ierr)
 call MPI_RECV(a(1, e+1), nx, MPI_DOUBLE_PRECISION, nbrtop,
&1, comm1d, status, ierr)
 call MPI_SEND(a(1, s), nx, MPI_DOUBLE_PRECISION, nbrbottom,
&1, comm1d, ierr)
 endif
 return
 end
```

Програма побудована так, що парні процеси посилають повідомлення першими, а непарні процеси першими виконують прийом. Тут слід відзначити, що подібне спарювання посилки й прийому є доволі ефективним, але його реалізація при складній взаємодії процесів може виявитись важкою, наприклад, на нерегулярній сітці.

Альтернативою є використання ***комбінованого*** прийому/передачі за до-

помогою спеціальної процедури бібліотеки – MPI_SENDRECV. Ця процедура дозволяє послати й прийняти дані, не турбуючись про те, що може мати місце тупикова ситуація (дедлок) через недостачу буферного простору. Така програма наочно демонструє, як у цьому випадку кожен процес посилає дані процесу, розташованому вище за нього, і одночасно приймає дані від процесу, який міститься нижче:

```
 subroutine EXCHNG1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
 use mpi
 integer nx, s, e, comm1d, nbrbottom, nbrtop
 double precision a(0:nx+1, s-1:e+1)
 integer status(MPI_STATUS_SIZE), ierr
 call MPI_SENDRECV(a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop,
&0, a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, comm1d,
&status, ierr)
 call MPI_SENDRECV(a(1,s), nx, MPI_DOUBLE_PRECISION,
&nbrbottom, 1, a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
&comm1d, status, ierr)
 return
 end
```

Оскільки тупикові ситуації частіш за все виникають через нестачу буферу, який виділяється системою для пересилань, у MPI передбачена можливість для програміста створювати і резервувати власний буфер, у якому дані можуть розміщуватись до їхньої доставки за призначенням. Повідомлення, які відправляються за допомогою процедури MPI_BSEND через такий буфер, мають назву пересилань з ***буферуванням***. Це знімає з програміста відповідальність за безпечне упорядкування операцій посилки й прийому даних. В програмах, які використовують пересилання із буферуванням, звичайні звернення до процедур пересилання повідомлень MPI_SEND просто замінюються на виклики процедур MPI_BSEND, як це зроблено у такому коді:

```
 subroutine EXCHNG1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
 use mpi
 integer nx, s, e, integer coimn1d, nbrbottom, nbrtop
 double precision a(0:nx+1, s-1:e+1)
 integer status(MPI_STATUS_SIZE), ierr
 call MPI_BSEND(a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop,
&0, comm1d, ierr)
```

```
 call MPI_RECV(a(1, s-1), nx, MPI_DOUBLE_PRECISION,
&nbrbottom, 0, comm1d, status, ierr)
 call MPI_BSEND(a(1,s), nx, MPI_DOUBLE_PRECISION,
&nbrbottom, 1, comm1d, ierr)
 call MPI_RECV(a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop,
&1, commid, status, ierr)
 return
 end
```

Звільняючись від упорядкування операцій посилки й прийому даних, програміст зобов'язаний власноручно зарезервувати пам'ять під буфер достатнього розміру, в якому можуть розміщуватись повідомлення будь-якої можливої у програмі довжини. Резервування буфера виконується за допомогою бібліотечної процедури MPI_BUFFER_ATTACH. У процедурі EXCHNG1, що розглядається, потрібен буфер розміром 2*nx значень подвійної точності. Це можна зробити за допомогою нижченаведеного відрізка програми (8 – число байтів у значеннях подвійної точності).

```
 double precision buffer(2*MAXNX + 2*MPI_BSEND_OVERHEAD)
call MPI_BUFFER_ATTACH(buffer, MAXNX * 8 + 2 *
&MPI_BSEND_OVERHEAD * 8, ierr)
```

З коду видно, що крім байтів безпосередньо під дані, розмір буфера повинен збільшуватись на деяку кількість байтів, яка описується бібліотечною константою MPI_BSEND_OVERHEAD. Застосування цієї константи, причому у вигляді саме її ідентифікатора, для створення буфера повідомлення є обов'язковим. Така вимога пов'язана з тим, що у різних реалізаціях бібліотеки значення цієї константи може відрізнятись, а додатковий простір у буфері потрібен, щоб управляти буферною областю й комунікаціями при зверненні і під час виконання процедури MPI_BSEND.

Коли програма більш не має потреби в буфері, область пам'яті, що йому належала, необхідно визволити шляхом звернення до відповідної бібліотечної процедуру MPI_BUFFER_DETACH.

На перший погляд здається, що використання пересилань із буферуванням завжди і повністю вирішує проблему з виникненням дедлок ситуацій у

програмах. Але насправді і в цьому випадку іноді можливе виникнення тупикової ситуації. Прикладом до сказаного може бути такій фрагмент програми:

```
      size = 100 + MPI_BSEND_OVERHEAD
      call MPI_BUFFER_ATTACH(buf, size, ierr)
      do 10 i = 1, n
      call MPI_BSEND(sbuf, 100, MPI_BYTE, 0, dest,
     &MPI_COMM_WORLD, ierr)
C     Виконання яких-небудь дій
   10 continue
      call MPI_BUFFER_DETACH(buf, size, ierr)
```

Тут виконується n ітерацій пересилання сотні байтів даних через буфер, розмір якого оголошений як 100 байтів плюс MPI_BSEND_OVERHEAD. Здається, що всі ітерації циклу будуть виконуватись без яких-небудь ускладнень (розмір буфера відповідає довжині повідомлення). Але проблема тут полягає в тому, що прийом повідомлення, яке відправлено на ітерації з номером i, може виявитись не закінченим при настанні наступної (i + 1)-ої ітерації циклу і відправленні чергового повідомлення. Щоб у цьому випадку не виникало подібних колізій, при використанні процедури MPI_BSEND, необхідно, щоб або буфер, описаний за допомогою MPI_BUFFER_ATTACH, був досить великий для зберігання всіх відправлених даних (ідеальним в даному випадку був би буфер, розмір якого дорівнює 100 * (n + MPI_BSEND_OVERHEAD) байтів), або операції приєднання й від'єднання буфера у обов'язковому порядку виконувались безпосередньо усередині циклу.

Такий спосіб зменшення додаткових витрат при пересиланні повідомлень полягає у застосуванні процедур відправки/прийому даних ***без блокування***. Справа в тому, що для більшості паралельних процесорів обмін даними між процесами потребує значно більше часу, чим усередині одиночного процесу. Щоб уникнути падіння ефективності, у багатьох паралельних процесорах використовується сполучення обчислень із одночасним обміном даними. У цьому випадку й використовуються операції прийому і передачі повідомлень без блокування процесів.

Процедура відправлення повідомлення без блокування процесів має назву

MPI_ISEND. Ця процедура аналогічна звичайній процедурі MPI_SEND за винятком того, що процедура MPI_ISEND закінчується одразу після її виклику, але буфер, що містить повідомлення, не може бути модифікований доти, поки повідомлення не буде доставлене повністю. Найлегший шлях перевірки завершення операції полягає у використанні операції MPI_TEST, наприклад, таким чином:

```
      call MPI_ISEND(buffer, count, datatype, dest, tag, cornm,
     &request, ierr)
C     Виконання яких-небудь дій
   10 call MPI_TEST (request, flag, status, ierr)
      if (.not. flag) goto 10
```

Часто бажано почекати до закінчення передачі. Тоді замість того, щоб писати цикл, як у попередньому прикладі, можна використовувати MPI_WAIT:

```
      call MPI_WAIT (request, status, ierr)
```

Коли операція без блокування процесів завершується (тобто MPI_WAIT або MPI_TEST повертають результат з flag = .true.), установлюється значення MPI_REQUEST_NULL.

За аналогією процедура MPI_IRECV починає операцію прийому без блокування процесів. Так само як і для MPI_ISEND, функція MPI_TEST може використовуватись для перевірки завершення операції прийому, початої MPI_IRECV, а функція MPI_WAIT може використовуватись для очікування завершення такого прийому. У бібліотеці MPI забезпечується спосіб очікування закінчення всіх або деякої комбінації операцій без блокування процесів (функції MPI_WAITALL і MPI_WAITANY). Процедура EXCHNG1 з використанням обміну без блокування, буде виглядати в такий спосіб:

```
      subroutine EXCHNG1(a, nx, s, e, comm1d, nbrbottom, nbrtop)
      use mpi
      integer nx, s, e, comm1d, nbrbottom, nbrtop
      double precision a(0:nx+1, s-1:e+1)
      integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
      call MPI_IRECV(a(1, s-1), nx, MPI_DOUBLE_PRECISION,
```

```
&nbrbottom,0, comm1d, req(1), ierr)
 call MPI_IRECV(a(1, e+1), nx, MPI_DOUBLE_PRECISION,
&nbrtop, 1, comm1d, req(2), ierr)
 call MPI_ISEND(a(1, e), nx, MPI_DOUBLE_PRECISION, nbrtop,
&0, comm1d, req(3), ierr)
 call MPI_ISEND(a(1, s), nx, MPI_DOUBLE_PRECISION,
&nbrbottom, comm1d, req(4), ierr)
 MPI_WAITALL(4, req, status_array, ierr)
 return
 end
```

Цей підхід дозволяє виконувати одночасно як передачу, так і прийом даних. При цьому можливо досягнути подвійного прискорення в порівнянні з вихідним варіантом процедури EXCHNG1, хоча небагато існуючих системи дозволяють зробити це. Щоб сполучати обчислення з обміном, необхідні деякі зміни і в інших вихідних програмах, розглянутих вище. Зокрема, необхідно змінити процедуру SWEEP так, щоб у ній можна було здійснювати деяку роботу, поки очікується прихід даних.

Осталось розглянути останню можливість модифікування процедури EXCHNG1 – повідомлення із ***синхронізацією***. У загальному випадку дуже важко передбачити, що треба зробити, щоб гарантувати незалежність програми від буферізації, але в багатьох спеціальних випадках можна показати, що, якщо програма успішно виконується без буферізації, вона буде виконуватися й з деяким розміром буферів. MPI забезпечує спосіб послати повідомлення, при якому відповідь від одержувача повідомлення не приходить доти, поки приймач дійсно не почне прийом повідомлення. Процедура називається MPI_SSEND. Програми, які використовують такі процедури не потребують буферізації, і тому називаються «ощадливими». В процедурі обміну EXCHNG1 виклик процедури MPI_SEND просто замінюється на звернення до MPI_SSEND.

6.1 Питання і завдання для самоперевірки

1) Який метод є одним з найбільш універсальних і ефективних методів, які на сьогодня набули значного поширення для приблизного розв'язання рівнянь математичної фізики? Як називаються функції дискретного аргу-

менту, визначені у вузлах деякої сітки замість функцій безперервного аргументу?

2) Опішіть алгоритм ітерацій Якобі, який використовується для розв'язання диференціальних рівнянь у частинних похідних? Зобразіть простір ітерацій циклу цього алгоритму і вкажіть інформаційні залежності між точками простору.
3) Як у процесі розпаралелювання розв'язання диференціального рівняння у частинних похідних виконується декомпозиція фізичної області існування розв'язку цього рівняння? Які спеціальні методи (функції) бібліотеки MPI допомагають вирішити проблему з тіньовими даними і взагалі дозволяють спростити процес розпаралелювання?
4) Застосовуючи фрагменти програми розв'язання задачі Пуассона, наведені в цьому розділі, спробуйте створити повністю працездатну програму мовою Java з використанням методів бібліотеки MPJ Express.

АЛФАВІТНИЙ ПОКАЖЧИК

Адамса-Башфорта
  метод 152, 153
  формули 150, 152, 153
Алгоритм
  блоковий 104
  викреслювання стовпців 96, 97, 98
  Гаусса-Зейделя паралельний 115, 122, 136
  здвоювання 84, 100
  клітинний 128, 129, 132, 136
  логарифмічного підсумовування 100
  рекурентного добутку 103
  розв'язання нелінійного рівняння паралельний найпростіший 139
  самоплануючий 123, 126, 136
Алгоритми паралельні 57
Багаточлен інтерполяційний 148
Батчера теореми 160
Брента лема 89
Векторизація 24
Великоблочне розпаралелювання 64, 65
Взаємодії комунікаційні 31
Винограду алгоритм 87
Вираз
  адитивний арифметичний 86
  альтернірований арифметичний 86
  мультиплікативний арифметичний 86
Вирази еквівалентні арифметичні 86
Висота алгоритму 57
Відношення
  інформаційне 46, 49, 51, 52
  операційне 46, 47, 51, 52
Віртуальна машина 33, 34, 37
Гайдна
  ефект 98, 99
Гаусса метод 81, 82, 160
Гаусса-Зейделя метод 106, 113, 114, 141
  модифікований 114
  паралельний 122
Горнера схема узагальнена 86, 88, 89, 96
Граф
  алгоритму 45, 46, 47, 56, 57
    ациклічний 87
  залежностей мінімальний 49
  інформаційний 51, 52

інформаційних залежностей 49, 50, 56, 167
керування 51, 52, 77
компактний 79
програми 45, 59, 60, 79
спрацьовування операторів 47
спрямований 45
Дерево
динамічне 119
комунікацій 117, 118, 119
Елементи тіньові 168, 169, 170
Жорсткі системи диференціальних рівнянь 155
Жорсткість системи 155
Задача
з векторним результатом 95
зі скалярним результатом 95
лінійна рекурентна 95
рекурентна 95
рекурентна лінійна 103
Задачі зі скалярним результатом 96
Залежність інформаційна 48, 49, 55, 66, 70, 73,
.... 74, 75, 78, 81, 109, 142
Збіжність методу 141, 143, 158
Збіжності методу 139
Здвоювання
схема 84, 100
щодо операції суми 118
Зейделя метод 141
Ідентифікатор задачі 7, 38
Інтерфейс передачі повідомлень 7, 15, 39
Історія
інформаційна 84
операційна 51
Історія інформаційна 51, 77, 87, 89
параметризована 56
Ітерації
вектор 116
матриця 115
параметр 116
Ітерації циклів 45, 46
Комунікатор 40, 128, 130, 131, 132, 169
глобальний 132
картезіанський 170
Комунікаторів сімейство 133
Коші задача 145, 146, 151, 153, 156, 158
Лобатто методи 160

Масштабування алгоритму 32
Матриця суміжності 60, 61
Метод
  m-кроковий k-точковий 147, 153
  багатокроковий багатоточковий 149
  блоковий 145
    однокроковий 157
    паралельний 145
  блоковий паралельний
    багатокроковий 145
    однокроковий 145
  гіперплощин 72, 74, 75
  здвоювання 118
  інтегроінтерполяційний 148
  кінцевих різниць 165
  координат 72
  однокроковий k-точковий 146, 158
  однокроковий чотирьохточковий блоковий 148
  паралелепіпедів 72, 75
  простої ітерації 106, 107, 108, 110, 111, 114, 141
  простої функціональної ітерації 159
  сіток 165
  хорд 138
Метод пірамід 72
Методи
  квазіньютонівські 105
  цілком неявні багатостадійні 157
Модель
  MPMD 35, 40
  SPMD 35, 40
  передачі повідомлень 33
Незалежність інформаційна 55
Необмеженого паралелізму концепція 83
Норма кубічна 121
Ньютона метод 104, 139, 141, 142
Ньютона метод модифікований 143
Обробка
  конвеєрна 28, 30
  паралельна 28
Обчислення
  паралельні 8, 9, 21, 23, 24, 27, 30, 32, 33, 39, 79, 85, 95, 119, 151
  розподілені 23, 33
Оцінка складності алгоритму 103
Паралелізм
  даних 23, 25, 26, 27

задач 23, 26, 28
кінцевий 63
координатний 63, 72
масовий 63
скошений 63, 72, 74
Паралельна віртуальна машина 33, 35, 36
Паралельні алгоритми 8, 9, 26, 27, 31, 83, 105, 161, 165
Перетворення еквівалентні 76
Погрішність апостеріорна локальна 159
Прискорення паралельного обчислення 121, 152, 160
Пристрій конвеєрний 29
Простір ітерацій 71, 72, 73, 74, 109
Пуассона задача 164, 165, 166, 171, 172
Радо й Лобатто квадратурні формули 157
Рівняння нелінійне 138, 139, 140, 144
Різницеві
похідні 165
рівняння 149, 165
схеми 165
Річардсона екстраполяція локальна 160
Розв'язання системи нелінійних рівнянь паралельне 140
розпаралелювання 86
Розпаралелювання 8, 23, 24, 30, 38, 43, 44, 45, 49, 53, 55,
.... 63, 64, 66, 67, 71, 76, 77, 81, 86, 88, 105, 160
Розподилення ітерацій
циклічне 67
Розподіл
ітерацій циклів 66, 67, 68, 69, 71, 78
циклів 70, 76, 83
Розподіл циклів 76
Розподілення ітерацій
блокове 67, 69, 71
блочно-циклічне 68
блочно-циклічне розподілення 67
великоблочне 109
циклічне 68, 69, 71
Розщеплення циклів 76, 78
Рунге правило 159
Рунге-Кутти
алгоритм 151, 152
цілком неявні методи 7, 157, 160, 161
Система
декартова віртуальна 132
лінійних алгебраїчних рівнянь 81
програмування 33

розподілена 24
Ситуація тупикова 6, 27, 177
Співвідношення рекурентні 95
Спрацьовування операторів 45, 46, 47, 50, 51, 52, 53
Спрацьовування операторів 52
Структура інформаційна 55
Схема
m-крокова k-точкова різницева 147
однокрокова k-точкова різницева 146
Схлопування циклів 76
Тейлора ряд 142
Топології віртуальні
картезіанські 169
Топологія віртуальна 128, 132, 166
декартова 132
Топологія картезіанська
одиничної вимірності 171
подвійної вимірності 172
Формула рекурентна 95
Функції сіткові 165
Функція стеля 67, 69
Шаблон взаємодії вузловий 118
Ширин
ярусу ЯПФ 87
Ширина
ЯПФ 60
ярусу 60
Шлях критичний 58
Якобі
ітерації 167
матриця 142, 143
метод 106, 113, 166
Ярусно-паралельна форма 109
програми
канонічна 59, 60, 63
програми (алгоритму) 7, 57, 59, 63, 101

ПЕРЕЛІК ІЛЮСТРАЦІЙ

Рисунок 1.1 − Загальна схема розробки паралельних алгоритмів 29
Рисунок 1.2 − Граф алгоритму ітерацій циклу 44
Рисунок 1.3 − Граф алгоритму для випадку спрацьовування операторів 45
Рисунок 1.4 − Вершини графа зв’язані відношенням 45
Рисунок 1.5 − Граф алгоритму з операційними відносинами для фрагмента 1.4 46
Рисунок 1.6 − Граф інформаційних залежностей для фрагмента коду 1.4 а) граф залежностей при послідовному проходженні операторів, б) еквівалентно перетворений граф залежностей 47
Рисунок 1.7 − Графи основних моделей алгоритмів а) граф керування програми (вершини – оператори, дуги – операційні відношення), б) інформаційний граф програми (вершини – оператори, дуги – інформаційні відношення), в) операційна історія програми (вершини – спрацьовування операторів, дуги – операційні відношення), г) інформаційна історія програми (вершини – спрацьовування операторів, дуги – інформаційні відношення) 49
Рисунок 1.8 − Граф інформаційної історії фрагмента коду 1.4 з поділом на послідовну й паралельну частини 51
Рисунок 1.9 − Відповідність компактної й розгорнутої моделей а) – компактна модель опису програми, б) і в) – дві різні історії, які еквівалентні компактній моделі 53
Рисунок 1.10 − Приклад переходу від графа програми до її канонічної ярусно-паралельної формі а) вихідний граф програми, б) еквівалентно перебудований граф, в) яруси ярусно-паралельної форми програми, г) канонічна ярусно-паралельна форма програми 55
Рисунок 1.11 − Ієрархія типів паралелізму 59
Рисунок 1.12 − ЯПФ фрагмента коду 1.6 61

Рисунок 1.13 − Графи алгоритмів розподілу ітерацій циклів а) операційна історія для блокового розподілу, б) ЯПФ для циклічного розподілу ітерацій 64
Рисунок 1.14 − Графи алгоритмів розподілу ітерацій циклів а) інформаційна історія для блокового розподілу, б) ЯПФ для циклічного розподілу ітерацій 66
Рисунок 1.15 − Простір ітерацій подвійного циклу 1.8 68
Рисунок 1.16 − Простір ітерацій подвійного циклу з інформаційними залежностями по обох координатах (скошений паралелізм) 69
Рисунок 1.17 − Простір ітерацій для методу паралелепіпедів 70
Рисунок 1.18 − Процес перетворення й дослідження алгоритму для циклу 1.11 а) компактний граф керування програми, б) інформаційна історія, в) компактний варіант інформаційної історії, г) перетворений алгоритм 71
Рисунок 1.19 − Перетворення графа програми циклу 1.13 а) компактний граф керування програми, б) інформаційна історія, в) канонічна ЯПФ алгоритму після розщеплення циклу 73
Рисунок 1.20 − Інформаційні структури уривка коду 1.14 а) інформаційна історія, б) яруси канонічної ЯПФ 74
Рисунок 1.21 − Перетворення графа алгоритму розв’язання СЛАР методом Гаусса а) інформаційна історія вихідного алгоритму, б) ЯПФперетвореного алгоритму 76
Рисунок 1.22 − Графи алгоритмів додавання послідовності чисел а) інформаційна історія, б) алгоритм здвоювання (ЯПФ)................ 78
Рисунок 1.23 − Ациклічні графи для розрахунку арифметичного виразу (1.1) а) інформаційна історія, б) канонічна ЯПФ 81
Рисунок 1.24 − Ациклічні графи для розрахунку арифметичного виразу (1.2) за схемою Горнера а) інформаційна історія, б) канонічна ЯПФ 82
Рисунок 2.1 − Схема організації обчислень при нейтралізації ефекту Гайдна для алгоритму викреслювання стовпців при розв’язанні рекурентної

задачі (2.2) 92
Рисунок 2.2 − Простір ітерацій для розв’язання СЛАР методом Якобі 102
Рисунок 2.3 − Декомпозиція даних для випадку, коли n дорівнює p............... 110
Рисунок 2.4 − Дерева комунікацій: а) – дерево комунікацій семи гілок алгоритму, б) – трьох вузлове піддерево здвоювання................ 111
Рисунок 2.5 − Графік залежності прискорення алгоритму Гаусса-Зейделя від кількості паралельних гілок 114
Рисунок 2.6 − Розбивка матриці A на підматриці порядку n’ = 2 при p = 9 і n = 6 121
Рисунок 3.1 − Знаходження кореня нелінійного рівняння методом хорд а) послідовний алгоритм; б) паралельний алгоритм 130
Рисунок 4.1 − Схема розбивки на блоки для однокрокового k-точкового методу 138
Рисунок 4.2 − Граф розрахунку однокрокової k-точкової різницевої схеми 138
Рисунок 4.3 − Схема розбивки на блоки для m-крокового k-точкового методу 138
Рисунок 4.4 − Граф розрахунку m-крокової k-точкової різницевої схеми......... 139
Рисунок 6.1 − Область із тіньовими точками для процесу 1 158
Рисунок 6.2 − UML-діаграма взаємодії процесів у процедурі EXCHG1 165

Навчальне електронне видання

РОЛЬЩИКОВ ВАДИМ БОРИСОВИЧ

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ (ЗМІСТОВНИЙ МОДУЛЬ №1)

Конспект лекцій

Видавець і виготовлювач
Одеський державний екологічний університет
вул. Львівська, 15, м. Одеса, 65016
тел./факс: (0482) 32-67-35
E-mail: info@odeku.edu.ua
Свідоцтво суб'єкта видавничої справи
ДК № 5242 від 08.11.2016