Міністерство освіти і науки України
Харківський національний університет імені В. Н. Каразіна

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Методичні вказівки
з виконання лабораторних робіт

Електронний ресурс

Харків – 2024

УДК 004.272.2
Т 58

Рецензенти:

С. В. Погорєлов – доктор фізико-математичних наук, професор, професор кафедри комп'ютерної математики і аналізу даних Національного технічного університету «Харківський політехнічний інститут»;

М. А. Мірошник – доктор технічних наук, професор, професор кафедри теоретичної та прикладної системотехніки Харківського національного університету імені В. Н. Каразіна.

Затверджено до розміщення в мережі Інтернет рішенням Науково-методичної ради Харківського національного університету імені В. Н. Каразіна (протокол № 9 від 18 червня 2024 року)

Толстолузька О. Г.

Т 58 Технології розподілених систем та паралельні обчислення : методичні вказівки з виконання лабораторних робіт [Електронний ресурс] / О. Г. Толстолузька, Д. П. Лабенко, Н. С. Бакуменко. – Харків : ХНУ імені В. Н. Каразіна, 2024. – (PDF 72 с.)

Методичні вказівки до виконання лабораторних робіт з дисципліни «Технології розподілених систем та паралельні обчислення» для студентів, які навчаються за освітніми програмами спеціальностей 123 «Комп'ютерна інженерія», 174 «Автоматизація, комп'ютерно-інтегровані технології та робототехніка», 122 «Комп'ютерні науки» усіх форм навчання.

Видання містить описи та завдання для проведення лабораторних робіт, контрольні питання для кожної лабораторної роботи, вимоги до оформлення звітів з лабораторних робіт, орієнтовні питання для контрольних робіт, орієнтовні завдання для курсового проектування.

УДК 004.272.2

Зміст

Вступ

Застосування комп'ютерної обробки інформації вимагає створення все більш продуктивних комп'ютерних систем. Однак можливості по нарощуванню потужності комп'ютерів мають очевидні фізичні обмеження. Одним з можливих роз в'язків цієї проблеми є збільшення продуктивності обчислювальних систем за рахунок використання паралельних та розподілених обчислень. Розпаралелювання процесів використовується в більшості комп'ютерних пристроїв від багатоядерних процесорів до суперкомп'ютерів. Тому для майбутніх IT-фахівців надзвичайно важливо володіти технологіями програмування для багатопроцесорних систем і саме на це спрямований курс «Технології розподілених систем та паралельних обчислень».

Метою викладання даної навчальної дисципліни є засвоєння студентами методів паралельної обробки даних, організації паралельних і розподілених обчислень, математичних моделей і методів розподілених і паралельних обчислень для багатопроцесорних систем.

При використанні багатопроцесорних пристроїв якість програмної реалізації паралельного алгоритму є надважливою. В якості показників оцінки ефективності паралельної програми зазвичай розглядають прискорення та ефективність.

Прискоренням паралельної програми S_p для системи з p процесорами називається відношення часу виконання програми з послідовним алгоритмом но одному процесорі T_1 і часу виконання задачі на p процесорах T_p:

$$S_p = \frac{T_1}{T_p}.$$

Ефективністю паралельного алгоритму E_p називається відношення прискорення до відповідної кількості процесорів:

$$E_p = \frac{S_p}{p}.$$

Найбільш розповсюдженою на теперішній час технологією програмування для систем з розподіленою пам'яттю є MPI (Massage Passive Interface). Способом взаємодії паралельних процесів в MPI є обмін повідомленнями. MPI дозволяє створювати паралельні програмні реалізації для широкого кола пристроїв, починаючи від настільних комп'ютерів із багатопроцесорною та/або багатоядерною архітектурою до кластерів робочих станцій або виділених обчислювальних вузлів і навіть високопродуктивних машин зі спільною пам'яттю. Тому в якості технології для практичної реалізації паралельних програм в цьому курсі був обраний саме MPI. В посібнику розглядається використання бібліотеки MPI для C/C++, однак навички роботи з цією бібліотекою можуть бути легко розповсюджені і використані для програм на інших мовах програмування (Fortran, Python тощо).

ЛАБОРАТОРНА РОБОТА № 1. Організація паралельної реалізації методів сортування

Мета:

Навчання практичним прийомам автоматичної оцінки і візуалізації процесу виконання експерименту і аналізу одержаних результатів, що дозволяють вивчити ефективність використання тих або інших алгоритмів на різних обчислювальних системах, зробити висновки про масштабованість алгоритмів і визначити можливе прискорення процесу паралельних обчислень.

1.1 Теоретичні відомості

Методи сортування

Сортування є однією з типових проблем обробки даних і звичайно розуміється як задача розміщення елементів неврегульованого набору значень

$$S = \{a_1, a_2, \dots, a_n\}$$

у порядку монотонного зростання або спадання

$$S \sim S' = \{(a'_1, a'_2, \dots, a'_n): a'_1 \le a'_2 \le \dots \le a'_n\}$$

(тут і далі всі пояснення скорочено даватимуться тільки на прикладі впорядковування даних за збільшенням).

1.1.1 Бульбашкове сортування

Послідовний алгоритм. Послідовний алгоритм бульбушкового сортування (bubble sort algorithm) порівнює і обмінює сусідні елементи в послідовності, яку потрібно відсортувати. Для послідовності $(a_1, a_2, \dots, a_n)$ алгоритм спочатку виконує $n - 1$ базових операцій "порівняння-обміну" для послідовних пар елементів $(a_1, a_2), (a_2, a_3), \dots, (a_{n-1}, a_n)$.

У результаті після першої ітерації алгоритму найбільший елемент переміщується ("спливає") в кінець послідовності. Далі останній елемент в перетвореній послідовності може бути виключений з розгляду, і описана вище процедура застосовується до частини послідовності, що залишилася $(a'_1, a'_2, \dots, a'_{n-1})$.

Таким чином, послідовність буде відсортована після $n - 1$ ітерації. Ефективність бульбашкового сортування може бути поліпшена, якщо припинити виконання алгоритму разі відсутності будь-яких змін сортованої послідовності даних в ході будь якої ітерації сортування.

Алгоритм пар-непарної перестановки

Алгоритм бульбашкового сортування в прямому вигляді достатньо складний для розпаралелювання – порівняння пар значень упорядкованого набору даних відбувається строго послідовно. У зв'язку з цим для паралельного виконання

використовується не сам цей алгоритм, а його модифікація, відома в літературі як метод пар-непарної перестановки (odd-even transposition method). Суть модифікації полягає в тому, що в алгоритм сортування вводяться два різні правила виконання ітерацій методу: залежно від парності або непарності номера ітерації сортування для обробки вибираються елементи з парними або непарними індексами відповідно, порівняння значень, що виділяються, завжди здійснюється з їх правими сусідніми елементами. Таким чином, на всіх непарних ітераціях порівнюються пари $(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n)$, а на парних ітераціях обробляються елементи $(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1})$.

Після n-кратного повторення ітерацій сортування початковий набір даних стає впорядкованим.

Визначення підзадач і виділення інформаційних залежностей

Отримання паралельного варіанту для методу пар-непарної перестановки вже не представляє будь-яких ускладнень - порівняння пар значень на ітераціях сортування є незалежними і можуть бути виконані паралельно. У разі $p < n$, коли кількість процесорів менше числа значень, які упорядковуються, процесори містять блоки даних розміру n/p і як базова підзадача може бути використана операція "порівняти і розділити".

Масштабування і розподіл підзадач по процесорах

Як наголошувалося раніше, кількість підзадач відповідає числу наявних процесорів, і тому необхідності в проведенні масштабування обчислень не виникає. Початковий розподіл блоків упорядковуваного набору даних по процесорах може бути вибраний довільним чином. Для ефективного виконання розглянутого паралельного алгоритму сортування потрібно, щоб процесори з сусідніми номерами мали прямі лінії зв'язку.

Аналіз ефективності

При аналізі ефективності проведемо загальну оцінку складності розглянутого паралельного алгоритму сортування. Визначимо спочатку трудомісткість послідовних обчислень. При розгляді даного питання алгоритм бульбашкового сортування дозволяє продемонструвати наступний важливий момент. Як вже наголошувалося, використаний для розпаралелювання послідовний метод впорядковування даних характеризується квадратичною залежністю складності від числа даних, що підлягають упорядкуванню, тобто $T_1 \sim n^2$. Проте вживання подібної оцінки складності послідовного алгоритму приведе до спотворення початкового цільового призначення критеріїв якості паралельних обчислень – показники ефективності в цьому випадку характеризуватимуть спосіб паралельного виконання даного конкретного методу сортування, а не результативність використовування паралелізму, що використовується, для задачі впорядкування даних в цілому як такої. Відмінність полягає в тому, що для сортування можуть бути застосовані більш ефективні послідовні алгоритми, трудомісткість яких має порядок

$$T_1 \sim n \log_2 n \qquad (1.1)$$

і щоб порівняти, наскільки швидше можуть бути впорядковані дані при використовуванні паралельних обчислень, в обов'язковому порядку повинна застосовуватися саме ця оцінка складності. Як основний результат приведених міркувань, можна сформулювати твердження про те, що **при визначенні показників прискорення і ефективності паралельних обчислень у якості оцінки складності послідовного способу розв'язання даної задачі слід використовувати трудомісткість найкращих послідовних алгоритмів**. Паралельні методи розв'язання задач повинні порівнюватися з найбільш швидкодійними послідовними способами обчислень!

Визначимо тепер складність розглянутого паралельного алгоритму впорядковування даних. Як наголошувалося раніше, на початковій стадії роботи методу кожний процесор проводить впорядковування своїх блоків даних (розмір блоків при рівномірному розподілі даних дорівнює n/p). Припустимо, що дане початкове сортування може бути виконано за допомогою швидкодійних алгоритмів впорядковування даних, тоді трудомісткість початкової стадії обчислень можна визначити виразом вигляду:

$$T_p^1 = (n/p)\, log_2(n/p) \tag{1.2}$$

Далі, на кожній ітерації, що виконується, паралельного сортування взаємодіючі пари процесорів здійснюють передачу блоків між собою, після чого одержувані на кожному процесорі пари блоків об'єднуються за допомогою процедури злиття. Загальна кількість ітерацій не перевищує величини p, і, як результат, загальна кількість операцій цієї частини паралельних обчислень буде дорівнювати

$$T_p^2 = 2p(n/p) = 2n \tag{1.3}$$

З урахуванням одержаних співвідношень показники ефективності і прискорення паралельного методу сортування мають вигляд:

$$S_p = \frac{nlog_2 n}{(n/p)log_2(n/p)+2n} = \frac{plog_2 n}{log_2(n/p)+2p} \quad E_p = \frac{nlog_2 n}{p((n/p)log_2(n/p)+2n)} = \frac{log_2 n}{log_2(n/p)+2p} \tag{1.4}$$

Розширимо наведені вирази – врахуємо тривалість обчислювальних операцій, що виконуються, і оцінимо трудомісткість операції передачі блоків між процесорами. При використанні моделі Хокні загальний час всіх операцій сортування, що виконуються в ході передачі блоків, можна оцінити за допомогою співвідношення:

$$T_p(comm) = p \cdot (\alpha + \omega \cdot (n/p)/\beta)$$

де α – латентність , β – пропускна здатність мережі передачі даних, а ω – розмір елемента даних, що упорядковуються, в байтах.

З урахуванням трудомісткості комунікаційних дій загальний час виконання паралельного алгоритму сортування визначається наступним виразом:

$$T_p = ((n/p)log_2(n/p) + 2n) \cdot \tau + p \cdot (\alpha + \omega \cdot (n/p)/\beta)$$

де τ - час виконання базової операції сортування.

1.1.2. Сортування Шелла

Послідовний алгоритм. Загальна ідея сортування Шелла (Shell sort) полягає в порівнянні на початкових стадіях сортування пар значень, що розташовуються достатньо далеко один від одного в упорядковуваному наборі даних. Така модифікація методу сортування дозволяє швидко переставляти далекі неврегульовані пари значень (сортування таких пар звичайно вимагає великої кількості перестановок, якщо використовується порівняння тільки сусідніх елементів).

Загальна схема методу полягає в наступному. На першому кроці алгоритму відбувається впорядковування елементів $\frac{n}{2}$ пар $(a_i, a_{n/2+i})$ для $1 \le i \le 2$. Далі, на другому кроці упорядковуються елементи в $\frac{n}{4}$ групах з чотирьох елементів $(a_i,\ a_{n/4+i}, a_{n/2+i}, a_{3n/4+i})$, для $1 \le i \le 4$. На третьому кроці упорядковуються елементи вже в $\frac{n}{4}$ групах з восьми елементів і т.д. На останньому кроці упорядковуються елементи відразу у всьому масиві $(a_1, a_2, \ldots, a_n)$. На кожному кроці для впорядковування елементів в групах застосовується метод сортування вставками. Як можна помітити, загальна кількість ітерацій алгоритму Шелла дорівнює $log_2 n$.

Організація паралельних обчислень

Для алгоритму Шелла може бути запропонований паралельний аналог методу, якщо топологія комунікаційної мережі може бути ефективно представлена у вигляді N-мірного гіперкуба (тобто кількість процесорів дорівнює $p = 2N$). Виконання сортування у такому разі може бути розділено на два послідовні етапи. На першому етапі (N ітерацій) здійснюється взаємодія процесорів, що є сусідніми в структурі гіперкуба. Формування пар процесорів, що взаємодіють між собою при виконанні операції "порівняти і розділити", може бути забезпечено за допомогою наступного простого правила - на кожній ітерації i, $0 \le i \le N$, парними стають процесори, у яких відмінність в бітових представленнях їх номерів є тільки у позиції $N - i$.

Другий етап полягає в реалізації звичних ітерацій паралельного алгоритму пар-непарної перестановки. Ітерації даного етапу виконуються до припинення фактичної зміни сортованого набору, і, тим самим, загальна кількість L таких ітерацій може бути різною – від 2 до p.

На рис.1.1 показаний приклад сортування масиву з 16 елементів за допомогою розглянутого способу (процесори показані кружками, номери процесорів дані в

бітовому представленні). Потрібно відзначити, що дані виявляються впорядкованими вже після першого етапу і немає необхідності виконувати пар-непарну перестановку.

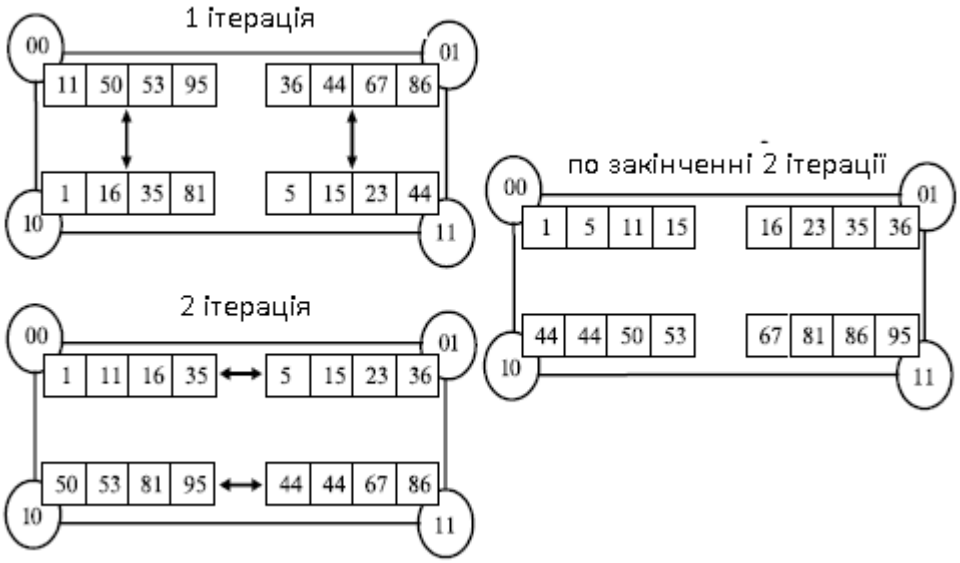


Рис. 1.1. Приклад роботи алгоритму Шелла для 4 процесорів

З урахуванням представленого опису паралельного варіанту алгоритму Шелла базова підзадача для організації паралельних обчислень, як і раніше (див. п. **Визначення підзадач і виділення інформаційних залежностей**), може бути визначена на основі операції "порівняти і розділити". Як результат, кількість підзадач завжди співпадає з числом наявних процесорів (розмір блоків даних в підзадачах дорівнює n/p) і не виникає проблеми масштабування. Розподіл блоків упорядковуваного набору даних по процесорах повинен бути вибраний з урахуванням можливості ефективного виконання операцій "порівняти і розділити" при представленні топології мережі передачі даних у вигляді гіперкуба.

1.1.3 Швидке сортування

Послідовний алгоритм. При загальному розгляді алгоритму швидкого сортування (quick sort algorithm), запропонованого Хоаром (C.A.R. Hoare), перш за все слід зазначити, що цей метод ґрунтується на послідовному діленні сортованого набору даних на блоки меншого розміру таким чином, щоб між значеннями різних блоків забезпечувалося відношення впорядкованості (для будь-якої пари блоків всі значення одного з цих блоків не перевищують значень іншого блоку). На першій ітерації методу здійснюється розподіл початкового набору даних на перші дві частини – для організації такого розподілу вибирається деякий ведучий елемент і всі значення набору, менші провідного елементу, переносяться в перший формований блок, вся решта значень утворює другий блок набору. На другій ітерації сортування описані правила застосовуються рекурсивно для обох сформованих блоків і т.д. При належному виборі провідних елементів після виконання $log_2 n$ ітерацій початковий масив даних виявляється впорядкованим.

Ефективність швидкого сортування в значній мірі визначається правильністю вибору провідних елементів при формуванні блоків. У гіршому разі трудомісткість методу має той же порядок складності, що і бульбашкове сортування (тобто $T_1 \sim n^2$).

При оптимальному виборі провідних елементів, коли ділення кожного блоку відбувається на рівні за розміром частини, трудомісткість алгоритму співпадає з швидкодією найефективніших способів сортування $T_1 = n\,log_2 n$. В середньому випадку кількість операцій, які виконуються алгоритмом швидкого сортування, визначається виразом: $T_1 = 1{,}4n\,log_2 n$.

Паралельний алгоритм швидкого сортування

Паралельне узагальнення алгоритму швидкого сортування найпростішим способом може бути одержано, якщо топологія комунікаційної мережі може бути ефективно представлена у вигляді N-мірного гіперкуба (тобто $p = 2N$). Нехай, як і раніше, початковий набір даних розподілений між процесорами блоками однакового розміру n/p; результуюче розташування блоків повинне відповідати нумерації процесорів гіперкуба. Можливий спосіб виконання першої ітерації паралельного методу за таких умов може полягати в наступному:

- вибрати яким-небудь чином ведучий елемент і розіслати його на всі процесори системи (наприклад, у якості провідного елементу можна взяти середнє арифметичне елементів, розташованих на вибраному ведучому процесорі);
- розділити на кожному процесорі наявний блок даних на дві частини з використанням одержаного ведучого елементу;
- -створити пари процесорів, для яких бітове представлення номерів відрізняється тільки у позиції *N*, і здійснити взаємообмін даними між цими процесорами.

У результаті виконання такої ітерації сортування початковий набір виявляється розділеним на дві частини, одна з яких (із значеннями меншими, ніж значення провідного елементу) розташовується на процесорах, в бітовому представленні номерів яких біт N дорівнює 0. Таких процесорів всього $p/2$, і, таким чином, початковий *N*-мірний гіперкуб також виявляється розділеним на два гіперкуби розмірності $N - 1$. До цих підгіперкубів, у свою чергу, може бути паралельно застосована описана вище процедура. Після *N*-кратного повторення подібних ітерацій для завершення сортування достатньо упорядкувати блоки даних, що отримані на кожному окремому процесорі обчислювальної системи.

Для пояснення на рис.1.2 представлений приклад впорядковування даних при $n = 16, p = 4$ (тобто блок кожного процесора містить 4 елементи). На рисунку процесори зображені у вигляді прямокутників, усередині яких показаний вміст упорядковуваних блоків даних; значення блоків приводяться на початку і при завершенні кожної ітерації сортування. Взаємодіючі пари процесорів сполучені двонаправленими стрілками. Для розділення даних вибиралися якнайкращі значення провідних елементів: на першій ітерації для всіх процесорів використовувалося значення 0, на другій ітерації для пари процесорів 0, 1 ведучий елемент дорівнює -5, для пари процесорів 2, 3 це значення було прийнято рівним 4.

Як і раніше, у якості базової підзадачі для організації паралельних обчислень може бути вибрана операція "порівняти і розділити", а кількість підзадач співпадає з числом процесорів, що використовуються. Розподіл підзадач по процесорах повинен

проводитися з урахуванням можливості ефективного виконання алгоритму при представленні топології мережі передачі даних у вигляді гіперкуба.

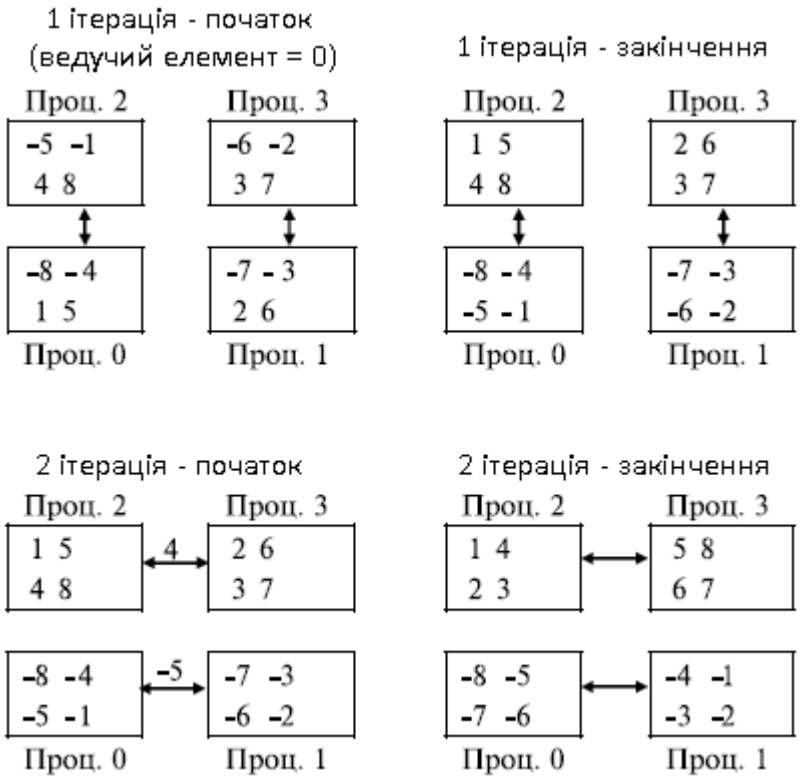


Рис.1.2. Приклад впорядковування даних паралельним методом швидкого сортування (без результатів локального сортування блоків)

Аналіз ефективності

Оцінимо трудомісткість розглянутого паралельного методу. Нехай у нас є N-мірний гіперкуб, що складається з $p = 2N$ процесорів, де $p < n$.

Ефективність паралельного методу швидкого сортування, як і в послідовному варіанті, багато в чому залежить від правильності вибору значень провідних елементів. Визначення загального правила для вибору цих значень є досить складним. Складність такого вибору може бути знижена, якщо виконати впорядкування локальних блоків процесорів перед початком сортування і забезпечити однорідний розподіл сортованих даних між процесорами обчислювальної системи.

Визначимо обчислювальну складність алгоритму сортування. На кожній з $log_2 n$ ітерацій сортування кожний процесор здійснює розподіл блоку щодо провідного елементу, складність цієї операції складає n/p операцій (припускатимемо, що на кожній ітерації сортування кожний блок ділиться на рівні за розміром частини). При завершенні обчислень процесор виконує сортування своїх блоків, що може бути виконано при використанні швидких алгоритмів за $\frac{n}{p} log_2 \frac{n}{p}$ операцій.

1.2 Завдання до виконання

1. Ознайомитися з теоретичними відомостями.
2. Для кожного методу виділити інформаційні залежності і можливості програми по розпаралелюванню. Для цього розібрати і взяти за основу методику розв’язання задачі розпаралелювання обчислення часткових сум послідовності числових значень, наведену у Додатку А.
3. Розробити послідовні Сі-програми розглянутих методів сортування. Передбачити вимір часу реалізації програм.
4. Провести декілька (не менше 5) обчислювальних експериментів, задаючи різні розміри масивів даних. Визначити залежність часу виконання алгоритму від об'єму початкових даних.
5. Побудувати графіки залежності часу виконання алгоритму від об'єму початкових даних.
6. Зробити висновки.

1.3 Зміст звіту

1. Титульний аркуш.
2. Постановка задачі. Короткий опис алгоритму.
3. Лістінг створених програм.
4. Скріншоти результатів виконання програм.
5. Оцінка показників прискорення й ефективності послідовних і паралельних обчислень (у вигляді графіків).
6. Відповіді на будь-які 5 запитань (питання переписувати обов’язково).
7. Висновки з виконаної роботи.

1.4 Контрольні питання

1. В чому полягає ідея сортування бульбушкою?
2. Ідея алгоритму пар-непарної перестановки.
3. Які параметри ефективності паралельного алгоритму сортування?
4. Ідея сортування Шела та особливості його розпаралелювання.
5. Ідея швидкого сортування та особливості організації паралельних обчислень.
6. Аналіз ефективності сортування Шела.
7. Аналіз ефективності швидкого сортування.

ЛАБОРАТОРНА РОБОТА № 2. Матричне множення. Організація паралельної реалізації методів матричного множення

Мета:

набуття практичних навичок та прийомів автоматизованої оцінки і візуалізації процесів матричного множення, виконання експерименту та аналізу одержаних результатів, що дозволяють вивчити ефективність використання тих чи інших алгоритмів на різних обчислювальних системах, зробити висновки про масштабованість алгоритмів і визначити можливе прискорення процесу паралельних обчислень.

2.1 Теоретичні відомості

Матричне множення

Задача множення матриці на матрицю визначається співвідношеннями:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}\,, \qquad 1 \le i,j \le n$$

(для простоти викладу матеріалу припускатимемо, що матриці A і B, які перемножуються, є квадратними і мають порядок $n \times n$). Як випливає з приведених співвідношень, обчислювальна складність задачі є достатньо високою (оцінка кількості виконуваних операцій має порядок n^3).

Основу можливості паралельних обчислень для матричного множення складає незалежність розрахунків для отримання елементів c_{ij} результуючої матриці C. Тим самим, всі елементи матриці C можуть бути обчислені паралельно за наявності n^2 процесорів, при цьому на кожному процесорі розташовуватиметься по одному рядку матриці A і одному стовпцю матриці B. При меншій кількості процесорів подібний підхід приводить до стрічкової схеми розбиття даних, коли на процесорах розташовуються по декілька рядків і стовпців початкових матриць.

Інший широко відомий підхід для побудови паралельних способів виконання матричного множення, що використовується, полягає у вживанні блокового представлення матриць, при якому початкові матриці A, B, і результуюча матриця C розглядаються у вигляді наборів блоків (як правило, квадратного виду деякого розміру $m \times m$). Тоді операцію матричного множення матриць A і B в блоковому вигляді можна представити таким чином:

$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1k} \\ & & \dots & \\ A_{k1} & A_{k2} & \dots & A_{kk} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} & \dots & B_{1k} \\ & & \dots & \\ B_{k1} & B_{k2} & \dots & B_{kk} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1k} \\ & & \dots & \\ C_{k1} & C_{k2} & \dots & C_{kk} \end{pmatrix}$$

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

$$C_{ij} = \sum_{l=1}^{k} A_{il} B_{lj}$$

Одержані блоки C_{ij} також є незалежними, і, як результат, можливий підхід для паралельного виконання обчислень може полягати в розрахунках, пов'язаних з отриманням окремих блоків C_{ij}, на різних процесорах. Вживання подібного підходу дозволяє одержати багато ефективних паралельних методів множення блочно-представлених матриць.

Стрічковий алгоритм

При стрічковій схемі розділення даних початкові матриці розбиваються на горизонтальні (для матриці A) і вертикальні (для матриці B) смуги. Одержані смуги розподіляються по процесорах, при цьому на кожному з наявного набору процесорів розташовується тільки по одній смузі матриць A і B. Перемноження смуг (а дана операція може бути виконана процесорами паралельно) приводить до отримання частини блоків результуючої матриці C. Для обчислення блоків матриці C, що залишилися, поєднання смуг матриць A і B на процесорах повинні бути змінені. В найпростішому вигляді це може бути забезпечено, наприклад, при **кільцевій топології обчислювальної мережі** (при числі процесорів, рівному кількості смуг) - в цьому випадку необхідна для матричного множення зміна положення даних може бути реалізована циклічним зсувом смуг матриці B по кільцю. Після багатократного виконання описаних дій (кількість необхідних повторень дорівнює числу процесорів) на кожному процесорі отримуємо набір блоків, що створює горизонтальну смугу матриці C.

Розглянута схема обчислень дозволяє визначити паралельний алгоритм матричного множення при стрічковій схемі розділення даних як ітераційну процедуру, на кожному кроці якої відбувається паралельне виконання операції перемноження смуг і подальшого циклічного зсуву смуг однієї з матриць по кільцю.

Блокові алгоритми Фокса і Кеннона

При блоковому представленні даних паралельна обчислювальна схема матричного множення в найпростішому вигляді може бути побудована, якщо топологія обчислювальної мережі має вид прямокутної решітки (якщо реальна топологія мережі має інший вигляд, представлення мережі у вигляді решітки можна забезпечити на логічному рівні). Основні положення паралельних методів для блочно- представлених матриць полягають в наступному:

1. кожний з процесорів решітки відповідає за обчислення одного блоку матриці C;
2. у ході обчислень на кожному з процесорів розташовується по одному блоку початкових матриць A і B;
3. при виконанні ітерацій алгоритмів блоки матриці A послідовно зсуваються уздовж рядків процесорної решітки, а блоки матриці B – уздовж стовпців решітки;

4. у результаті обчислень на кожному з процесорів отримуємо блок матриці C, при цьому загальна кількість ітерацій алгоритму дорівнює p (де p – число процесорів).

Алгоритм Фокса множення матриць при блоковому розділенні даних

Для більш простого пояснення наступного матеріалу припускатимемо далі, що всі матриці є квадратними розміром $n \times n$, кількість блоків по горизонталі і вертикалі однакова і дорівнює q (тобто розмір всіх блоків дорівнює $k \times k, k = {}^{n}/_{q}$). При такому представленні даних операція матричного множення матриць A і B в блочному вигляді може бути представлена так:

$$\begin{pmatrix} A_{00}A_{01} & \dots & A_{0q-1} \\ & \dots & \\ A_{q-10}A_{q-11} & \dots & A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00}B_{01} & \dots & B_{0q-1} \\ & \dots & \\ B_{q-10}B_{q-11} & \dots & B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00}C_{01} & \dots & C_{0q-1} \\ & \dots & \\ C_{q-10}C_{q-11} & \dots & C_{q-1q-1} \end{pmatrix}$$

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

$$C_{ij} = \sum_{s=0}^{kq-1} A_{is}B_{sj}$$

При блочному розбитті даних для визначення базових підзадач природним є узяти за основу обчислення, що виконуються над матричними блоками. З урахуванням сказаного **визначимо базову підзадачу як процедуру обчислення всіх елементів одного з блоків матриці** C.

Для виконання всіх необхідних обчислень базовим підзадачам повинні бути доступні відповідні набори рядків матриці A і стовпців матриці B. Розміщення всіх необхідних даних в кожній підзадачі неминуче приведе до дублювання і до значного зростання об'єму пам'яті, що використовується. В результаті, обчислення повинні бути організовані так, щоб в кожний поточний момент часу підзадачі містили лише частину необхідних для проведення розрахунків даних, а доступ до решти частини даних забезпечувався б за допомогою передачі даних між процесорами.

Отже, за основу паралельних обчислень для матричного множення при блочному розділенні даних прийнятий підхід, при якому базові підзадачі відповідають за обчислення окремих блоків матриці C і при цьому в підзадачах на кожній ітерації розрахунків розміщується тільки по одному блоку початкових матриць A і B. Для нумерації підзадач використаємо індекси блоків матриці C, які розміщуються в підзадачах, тобто підзадача (i, j) відповідає за обчислення блоку C_{ij} – тим самим, набір підзадач утворює квадратну решітку, що відповідає структурі блочного представлення матриці C.

Відповідно до алгоритму Фокса в ході обчислень на кожній базовій підзадачі (i, j) розташовується чотири матричні блоки:

- блок C_{ij} матриці C, що обчислюється підзадачею;

- блок A_{ij} матриці A, який розміщується в підзадачі перед початком обчислень;
- блоки A'_{ij}, B'_{ij} матриць A і B, отримані підзадачею в ході виконання обчислень.

Виконання паралельного методу включає:

- **етап ініціалізації**, на якому кожній підзадачі (i,j) передаються блоки A_{ij}, B_{ij} і обнуляються блоки C_{ij} на всіх підзадачах;
- **етап обчислень**, в рамках якого на кожній ітерації $l, 0 \le l < q$, здійснюються наступні операції:
  - для кожного рядка $i, 0 \le i < q$, блок A_{ij} підзадачі (i,j) пересилається на всі підзадачі того ж рядка і решітки; індекс j, що визначає положення підзадачі в рядку, обчислюється відповідно до виразу

$$j = (i + l)\ mod\ q,$$

де mod – операція отримання залишку від цілочисельного ділення;
  - отримані в результаті пересилок блоки A'_{ij}, B'_{ij} кожної підзадачі (i,j) перемножуються і додаються до блоку C_{ij};
  - блоки B'_{ij} кожної підзадачі (i,j) пересилаються підзадачам, що є сусідами зверху в стовпцях решітки підзадач (блоки підзадач з першого рядка решітки пересилаються підзадачам останнього рядка решітки).

Для пояснення цих правил паралельного методу на рис. 2.1 наведений стан блоків кожної підзадачі в ході виконання ітерацій етапу обчислень (для решітки підзадач 2×2).

Алгоритм Кеннона множення матриць при блочному розділенні даних

Як і для алгоритму Фокса, у якості базової підзадачі виберемо обчислення, пов'язані з визначенням одного з блоків результуючої матриці C. Як вже наголошувалося раніше, для обчислення елементів цього блоку підзадача повинна мати доступ до елементів горизонтальної смуги матриці A і елементам вертикальної смуги матриці B.

Відмінність алгоритму Кеннона від методу Фокса полягає в зміні схеми початкового розподілу блоків матриць, які перемножуються, між підзадачами обчислювальної системи. **Початкове розташування блоків в алгоритмі Кеннона підбирається так, щоб блоки в підзадачах могли б бути перемножені без будь-яких додаткових передач даних**. При цьому подібний розподіл блоків може бути організований таким чином, що переміщення блоків між підзадачами в ході обчислень може здійснюватися з використанням більш простих комунікаційних операцій.

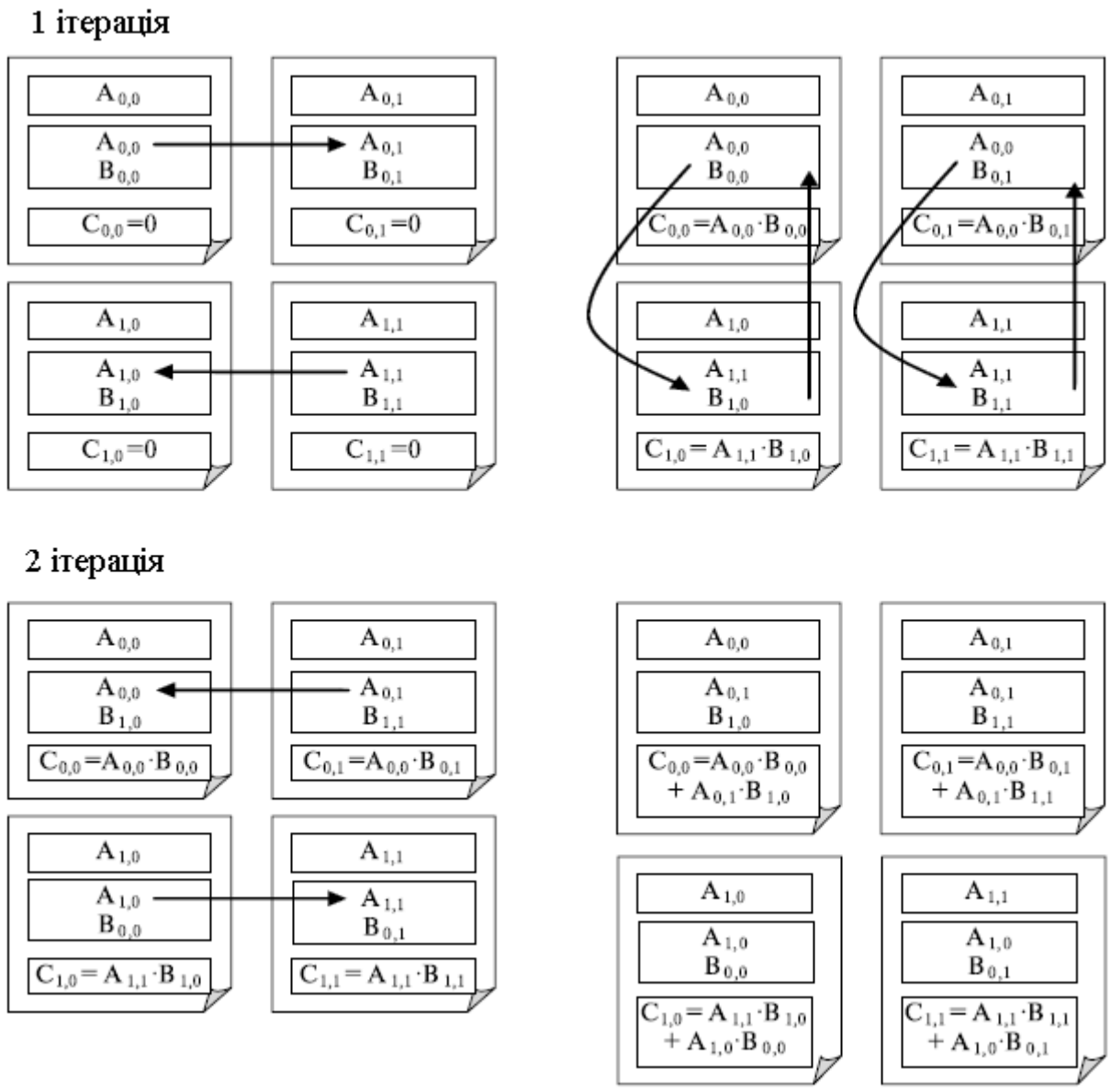


Рис. 2.1 Стан блоків в кожній підзадачі в ході виконання ітерацій алгоритму Фокса

З урахуванням вказаних зауважень етап ініціалізації алгоритму Кеннона включає виконання наступних операцій передач даних:

- в кожну підзадачу (i, j) передаються блоки A_{ij}, B_{ij};
- для кожного рядка i решітки підзадач блоки матриці A зсуваються на $(i - 1)$ позицій ліворуч;
- для кожного стовпця j решітки підзадач блоки матриці B здвигаються на $(j - 1)$ позицій вверх.

Для пояснення способу початкового розподілу даних, що використовується, на рис. 2 показаний приклад розташування блоків для решітки підзадач 3×3.

В результаті такого початкового розподілу в кожній базовій підзадачі розташовуватимуться блоки, які можуть бути перемножені без додаткових операцій передачі даних. Крім того, отримання всіх подальших блоків для підзадач може бути забезпечено за допомогою простих комунікаційних дій – **після виконання операції блокового множення кожний блок матриці A повинен бути переданий попередній підзадачі ліворуч по рядках решітки підзадач, а кожний блок матриці**

B – **попередній підзадачі вверх по стовпцях решітки**. Можна показати, що послідовність таких циклічних зсувів і множення блоків початкових матриць A і B, які приведе до отримання в базових підзадачах відповідних блоків результуючої матриці C.

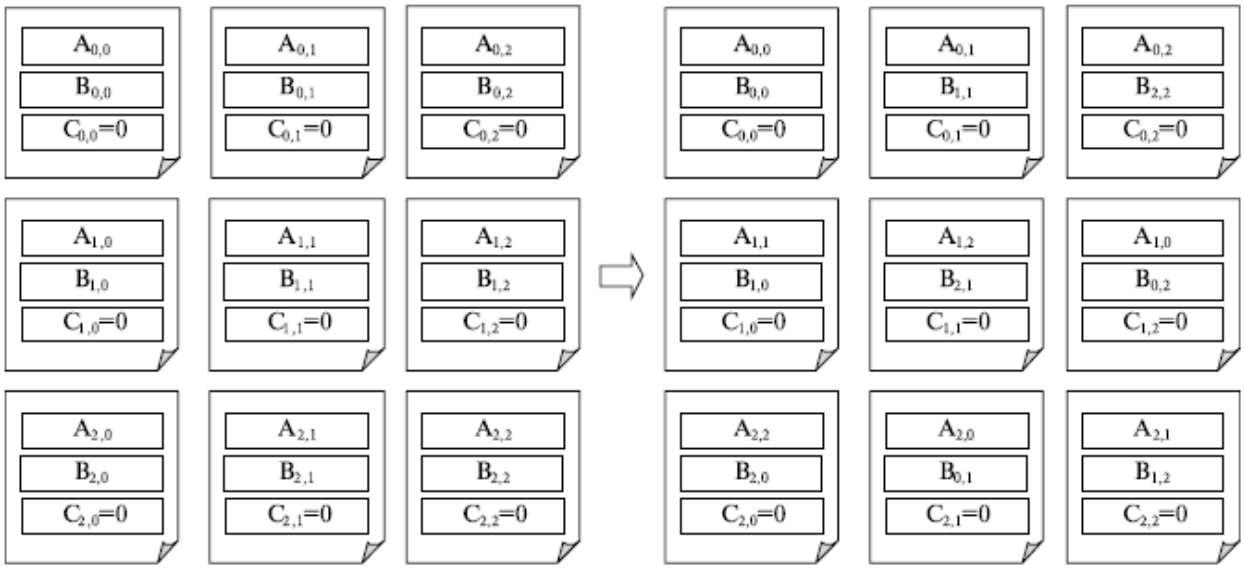


Рис. 2.2 Перерозподіл блоків початкових матриць між процесорами при виконанні алгоритму Кеннона

Як і раніше в методі Фокса, для алгоритму Кеннона розмір блоків може бути підібраний так, **щоб кількість базових підзадач співпадала з числом наявних процесорів**. Оскільки об'єми обчислень в кожній підзадачі одинакові, це забезпечує повне балансування обчислювального навантаження між процесорами.

Для розподілу підзадач між процесорами може бути застосований підхід, використаний в алгоритмі Фокса – множина наявних процесорів представляється у вигляді квадратної решітки і розміщення базових підзадач (i, j) здійснюється на процесорах P_{ij} відповідних вузлів процесорної решітки. Необхідна структура мережі передачі даних, як і раніше, може бути забезпечена на фізичному рівні при топології обчислювальної системи у вигляді решітки або повного графа.

2.2 Завдання до виконання

1. Розробити послідовну Сі-програму множення матриць стрічковим методом:
   a) виділити інформаційні залежності і можливості програми по розпаралелюванню;
   b) провести декілька (не менше 5) обчислювальних експериментів. Визначити залежність часу виконання алгоритму від об'єму початкових даних;
   c) побудувати графіки залежності часу виконання від об'єму початкових даних;
   d) зробити висновки.
2. Розробити послідовні Сі-програми блочного множення матриць методами Фокса і Кеннона:

a) виділити інформаційні залежності і можливості програми по розпаралелюванню;
b) вибрати метод Фокса множення матриць і провести декілька (не менше 5) обчислювальних експериментів для різних розмірів матриц, побудувати графіки;
c) вибрати алгоритм Кеннона матричного множення провести декілька (не менше 5) обчислювальних експериментів для різних розмірів матриць, побудувати графіки;
d) порівняти часові характеристики виконання всіх алгоритмів (розміри матриць повинні бути однакові) алгоритмів, побудувати графіки;
e) зробити висновки.

2.3 Зміст звіту

1. Титульний аркуш.
2. Назва та мета виконання лабораторної роботи.
3. Теоретичний опис, лістинг розроблених мовою С++ програм заданого методу.
4. Результати виконання за декількома варіантами даних та висновки для кожного методу.
5. Вихідні дані та відповідні графіки.
6. Зроблені висновки.
7. Відповіді на 5 будь-яких контрольних запитань.

2.4 Контрольні питання

1. У чому полягає постановка задачі множення матриць?
2. Привести приклади задач, в яких використовується операція множення матриць. Чи відрізняється їх обчислювальна трудомісткість?
3. Які способи розділення даних використовуються при розробці паралельних алгоритмів матричного множення?
4. Розробити загальні схеми розглянутих паралельних алгоритмів множення матриць.
5. Які інформаційні взаємодії виконуються для алгоритмів при стрічковій схемі розділення даних?
6. Які інформаційні взаємодії виконуються для блочних алгоритмів множення матриць?
7. Який з розглянутих алгоритмів характеризується найменшими і найбільшими вимогами до об'єму необхідної пам'яті?
8. Оцініть можливість виконання матричного множення як послідовності операцій множення матриці на вектор.
9. Дайте загальну характеристику програмної реалізації алгоритмів Фокса і Кеннона. В чому полягають відмінності в програмній реалізації розглянутих алгоритмів?

ЛАБОРАТОРНА РОБОТА № 3. Показники ефективності паралельної реалізації алгоритмів та їх зв`язок з вимогами практики

Мета:

а) ознайомлення з основами розробки паралельних програм з використанням можливостей MPI у мові програмування C++;

б) набуття навичок розробки програм проведення паралельних обчислень, визначення часу виконання, оцінка показників прискорення та ефективності виконання розроблених програм.

3.1 Теоретичні відомості

Основи MPI

MPI надає програмісту єдиний механізм взаємодії гілок всередині паралельного додатка незалежно від машинної архітектури (однопроцесорні / багатопроцесорні із загальною/роздільною пам'яттю), взаємного розташування гілок (на одному процесорі/на різних) і API операційної системи.

(API = «applications programmers interface» = «інтерфейс розробника додатків»)

Приведемо мінімально необхідний набір функцій MPI, достатній для розробки порівняльно простих паралельних програм.

Формат MPI-функций

Мова C (case sensitive): **`error = MPI_Xxxxx(parameter,...);`**

`MPI_Xxxxx(parameter,...);`

Мова C++ (case sensitive): **`error = MPI::Xxxxx(parameter,...);`**

`MPI::Xxxxx(parameter,...);`

Ініціалізація і завершення MPI-програм

Першою функцією MPI, що викликається, повинна бути функція

В мові C: **`int MPI_Init (int *argc, char ***argv)`**

В мові C++: **`void MPI::Init (int& argc, char**& argv),`**

де

– `argc` – покажчик на кількість параметрів командного рядка;

– `argv` – параметри командного рядка.

Ця функція застосовується для ініціалізації середовища виконання MPI-програми. Параметрами функції є кількість аргументів в командному рядку і адреса покажчика на масив символів тексту самого командного рядка.

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

`int MPI_Finalize(void).`

Як результат, можна відзначити, що структура паралельної програми, яка розроблена з використанням MPI, повинна мати такий вигляд:

```
#include «mpi.h»
int main(int argc, char *argv[]){
```

```
<програмний код без використання функцій MPI>
 MPI_Init(&argc &argv);
<програмний код з використанням функцій MPI>
 MPI_Finalize();
<програмний код без використання функцій MPI>
 return 0;
}
```

Слід зазначити:

– файл `mpi.h` містить визначення іменованих констант, прототипів функцій і типів даних бібліотеки MPI;

– функції `MPI_Init` і `MPI_Finalize` є обов'язковими і повинні бути виконані один (і лише один раз) кожним процесом паралельної програми. Якщо який-небудь із процесорів не виконує `MPI_Finalize`, то програма зависає;

– перед викликом `MPI_Init` може бути використана функція `MPI_Initialized` для визначення того, чи був раніше виконаний виклик `MPI_Init`, а після виклику `MPI_Finalize` - `MPI_Finalized1`) аналогічного призначення.

Розглянуті приклади функцій дають представлення синтаксису найменування функцій в MPI. Імені функції передує префікс **MPI**, далі слідує одне або декілька слів назви, перше слово в імені функції починається із заголовного символу, слова розділяються знаком підкреслення. Назви функцій MPI, як правило, пояснюють призначення виконуваних функцією дій.

Визначення кількості і рангу процесів

Визначення кількості процесів у виконуваній паралельній програмі здійснюється за допомогою функції:

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

де

- `comm` – комунікатор, розмір якого визначається;
- `size` – кількість процесів в комунікаторі, що визначається.

Для визначення рангу процесу використовується функція:

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

де

- `comm` – комунікатор, в якому визначається ранг процесу;
- `rank` – ранг процесу в комунікаторі.

Як правило, виклик функцій `MPI_Comm_size` і`MPI_Comm_rank` виконується відразу після `MPI_Init` для отримання загальної кількості процесів і рангу поточного процесу:

```
#include «mpi.h»
int main(int argc, char *argv[])
{
   int ProcNum, ProcRank;
<програмний код без використання функцій MPI>
```

```
   MPI_Init(&argc &argv);
   MPI_Comm_size(MPI_COMM_WORLD &ProcNum);
   MPI_Comm_rank(MPI_COMM_WORLD &ProcRank);
<програмний код з використанням функцій MPI>
   MPI_Finalize();
<програмний код без використання функцій MPI>
   return 0;
}
```

Слід зазначити:

– комунікатор `MPI_COMM_WORLD`, як наголошувалося раніше, створюється за замовчуванням і представляє всі процеси паралельної програми, яка виконується;

– ранг, що одержується за допомогою функції `MPI_Comm_rank`, є рангом процесу, що виконав виклик цієї функції, тобто змінна `ProcRank` набуває різних значень у різних процесах.

Передача повідомлень

Для передачі повідомлення процес-відправник повинен виконати функцію:

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

де

– `buf` – адреса буфера пам'яті, в якому розташовуються дані повідомлення, що відправляється;

– `count` – кількість елементів даних в повідомленні;

– `type` – тип елементів даних повідомлення, що пересилається;

– `dest` – ранг процесу, якому відправляється повідомлення;

– `tag` – значення-тег, що використовується для ідентифікації повідомлення;

– `comm` – комунікатор, в рамках якого виконується передача даних.

Для вказівки типу даних, що пересилаються, в MPI є низка базових типів, повний список яких приведений в табл. 3.1.

Таблиця 3.1

Базові (зарезервовані) типи даних MPI для алгоритмічної мови С

Тип даних MPI	Тип даних С
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Слід зазначити:

– повідомлення, що відправляється, визначається через вказівку блоку пам'яті (буфера), в якому це повідомлення розташовується. Тріада (`buf`, `count`, `type`), що використовується для вказівки буфера, входить до складу параметрів практично всіх функцій передачі даних;

– процеси, між якими виконується передача даних, в обов'язковому порядку повинні належати комунікатору, що вказується у функції `MPI_Send`;

– параметр `tag` використовується тільки за необхідності розрізняти повідомлення, які передаються, інакше як значення параметра може бути використано довільне позитивне ціле число[1] (див. також опис функції `MPI_Recv`).

Відразу ж після завершення функції `MPI_Send` процес-відправник може почати повторно використовувати буфер пам'яті, в якому розташовувалося повідомлення, що відправляється. Також слід розуміти, що у момент завершення функції `MPI_Send` стан повідомлення, що пересилається, може бути абсолютно різним: повідомлення може розташовуватися в процесі-відправнику, може знаходитися в стані передачі, може зберігатися в процесі-одержувачі або ж може бути прийнято процесом-одержувачем за допомогою функції `MPI_Recv`. Тому завершення функції `MPI_Send` означає лише те, що операція передачі почала виконуватися і пересилка повідомлення рано чи пізно буде виконана.

Приклад використання функції буде представлений далі після опису функції `MPI_Recv`.

Прийом повідомлень

Для прийому повідомлення процес-одержувач повинен виконати функцію:

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

де

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

– `source` – ранг процесу, від якого повинен бути виконаний прийом повідомлення;

– `tag` – тег повідомлення, яке повинне бути прийняте для процесу;

– `comm` – комунікатор, в рамках якого виконується передача даних;

– `status` - покажчик на структуру даних з інформацією про результат виконання операції прийому даних.

Слід зазначити, що:

– буфер пам'яті повинен бути достатнім для прийому повідомлення. При браку пам'яті частина повідомлення буде втрачена, і в коді завершення функції буде зафіксована помилка переповнення; з іншого боку, повідомлення, що приймається,

[1]Максимально можливе ціле число, однак, не може бути більше, ніж обумовлена реалізацією константа `MPI_TAG_UB`.

може бути і коротше, ніж розмір приймального буфера, у такому разі зміняться тільки ділянки буфера, зайняті прийнятим повідомленням;

– типи елементів повідомлення, яке передається і приймається повинні співпадати;

– за необхідності прийому повідомлення від будь-якого процесу-відправника для параметра `source` може бути вказано значення `MPI_ANY_SOURCE` (на відміну від функції передачі `MPI_Send`, яка посилає повідомлення строго визначеному адресату);

– за необхідності прийому повідомлення з будь-яким тегом для параметра **tag** може бути вказано значення `MPI_ANY_TAG` (знову-таки при використанні функції `MPI_Send` повинне бути вказано конкретне значення тегу);

– на відміну від параметрів «процес-одержувач» і «тег», параметр «комунікатор» не має значення, що означає «будь-який комунікатор»;

– параметр `status` дозволяє визначити ряд характеристик прийнятого повідомлення:

- `status.MPI_SOURCE` – ранг процесу-відправника прийнятого повідомлення;
- `status.MPI_TAG` – тег прийнятого повідомлення.

Приведені значення `MPI_ANY_SOURCE` і `MPI_ANY_TAG` іноді називають *джокерами*.

Значення змінної **status** дозволяє визначити кількість елементів даних в прийнятому повідомленні за допомогою функції:

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

де

– `status` – статус операції `MPI_Recv`;

– `type` – тип прийнятих даних;

– `count` – кількість елементів даних у повідомленні.

Виклик функції `MPI_Recv` не зобов'язаний бути узгодженим з часом виклику відповідної функції передачі повідомлення `MPI_Send` – прийом повідомлення може бути ініційований до моменту, в момент або після моменту початку відправки повідомлення.

Після закінчення функції `MPI_Recv` в заданому буфері пам'яті розташовуватиметься прийняте повідомлення. Принциповий момент тут полягає в тому, що ***функція `MPI_Recv` є блокуючою для процесу-одержувача***, тобто його виконання припиняється до завершення роботи функції. Таким чином, якщо з якихось причин очікуване для прийому повідомлення буде відсутнє, виконання паралельної програми буде блоковано.

Приклад програми з використанням MPI

Розглянутий набір функцій виявляється достатнім для розробки паралельних програм[2]. Програма, що приводиться нижче, є стандартним початковим прикладом для алгоритмічної мови С.

Програма 1. Перша паралельна програма з використанням MPI

```
 #include <stdio.h>
 #include «mpi.h»
 int main(int argc, char* argv[])
{
   int ProcNum, ProcRank, RecvRank;
   MPI_Status Status;
   MPI_Init(&argc &argv);
   MPI_Comm_size(MPI_COMM_WORLD &ProcNum);
   MPI_Comm_rank(MPI_COMM_WORLD &ProcRank);
   if ( ProcRank == 0 )
   {
   // Дії, виконувані тільки процесом з рангом 0
     printf(«\n Hello from process %3d», ProcRank);
     for (int i = 1; i < ProcNum; i++ )
     {
         MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
         MPI_ANY_TAG, MPI_COMM_WORLD &Status);
         printf(«\n Hello from process %3d», RecvRank);
      }
   }
   else // Повідомлення, що відправляється всіма процесами
        // окрім процесу з рангом 0
      MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
   MPI_Finalize();
   return 0;
 }
```

Як випливає з тексту програми, кожний процес визначає свій ранг, після чого дії в програмі розділяються. Всі процеси, окрім процесу з рангом 0, передають значення свого рангу нульовому процесу. Процес з рангом 0 спочатку друкує значення свого рангу, а далі послідовно приймає повідомлення з рангами процесів і також друкує їх значення. При цьому важливо відзначити, що порядок прийому повідомлень наперед не визначений і залежить від умов виконання паралельної програми (більш того, *цей порядок може змінюватися від запуску до запуску*). Так, можливий варіант результатів друку процесу 0 може бути таким (для паралельної програми з чотирьох процесів):

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

[2]Як було обіцяно раніше, кількість функцій MPI, необхідних для початку розробки паралельних програм, дорівнює шести.

Такий «плаваючий» вид одержуваних результатів істотним чином ускладнює розробку, тестування і налагодження паралельних програм, оскільки в цьому випадку зникає один з основних принципів програмування – повторюваність обчислювальних експериментів, що виконуються. Як правило, якщо це не приводить до втрати ефективності, слід забезпечувати однозначність розрахунків і при використанні паралельних обчислень. Для даного простого прикладу можна відновити постійність результатів, що отримуються за допомогою задання рангу процесу-відправника в операції прийому повідомлення:

MPI_Recv(&RecvRank, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD,&Status).

Вказівка рангу процесу-відправника регламентує порядок прийому повідомлень, і, як результат, рядки друку з'являтимуться строго в порядку зростання рангів процесів (нагадуємо, що така регламентація в окремих ситуаціях може приводити до уповільнення паралельних обчислень, які виконуються).

Слід зазначити ще один важливий момент: програма, що розробляється із застосуванням MPI, як в даному частковому варіанті, так і в найзагальнішому випадку, використовується для породження всіх процесів паралельної програми а значить, повинна визначати обчислення, які виконуються всіма цими процесами. Можна сказати, що MPI-програма є деякою «макропрограмою», різні частини якої використовуються різними процесами. Так, наприклад, у наведеному прикладі програми виділені рамкою ділянки програмного коду не виконуються одночасно жодним з процесів. Перша виділена ділянка з функцією прийому `MPI_Recv` виконується тільки процесом з рангом 0, друга ділянка з функцією передачі `MPI_Send` радіюється всіма процесами, за винятком нульового процесу.

Для розділення фрагментів коду між процесами звичайно використовується підхід, застосований в щойно що розглянутій програмі, – за допомогою функції `MPI_Comm_rank` визначається ранг процесу, а потім відповідно до рангу виділяються необхідні для процесу ділянки програмного коду. Наявність в тій же програмі фрагментів коду різних процесів також значно ускладнює розуміння і, в цілому, розробку MPI-програми. Як результат, можна рекомендувати при збільшенні об'єму програм, що розробляються, виносити програмний код різних процесів в окремі програмні модулі (функції). Загальна схема MPI-програми в цьому випадку матиме вигляд:

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

У багатьох випадках, як і в розглянутому прикладі, виконувані дії є відмінними тільки для процесу з рангом 0. У цьому випадку загальна схема MPI-програми приймає більш простий вигляд:

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

На завершення обговорення прикладу пояснимо застосований в MPI підхід для контролю правильності виконання функцій. Всі функції MPI (окрім MPI_Wtime і MPI_Wtick) повертають як своє значення код завершення. При успішному виконанні функції код, що повертається, дорівнює MPI_SUCCESS. Інші значення коду завершення свідчать про виявлення тих чи інших помилкових ситуацій під час виконання функцій. Для з'ясування типу знайденої помилки використовуються зарезервовані іменовані константи, серед яких:

– `MPI_ERR_BUFFER` – неправильний покажчик на буфер;

– `MPI_ERR_TRUNCATE` – повідомлення перевищує розмір приймального буфера;

– `MPI_ERR_COMM` – неправильний комунікатор;

– `MPI_ERR_RANK` – неправильний ранг процесу й ін.

Повний список констант для перевірки коду завершення міститься у файлі ***mpi.h***. Проте, за замовчуванням, виникнення будь-якої помилки під час виконання функції MPI приводить до негайного завершення паралельної програми. Для того щоб мати нагоду проаналізувати код завершення, що повертається, необхідно скористатися MPI функціями, які надаються, зі створення обробників помилок і управління ними, розгляд яких не входить в матеріалу даного посібника.

3.2 Завдання до виконання

1. Ознайомитися з теоретичними відомостями.
2. Виконати налаштування проєкту для програми з прикладу 1.
3. Виконати завдання:
   a) У початковому тексті програми на мові C++ пропущені виклики процедур підключення до MPI, визначення кількості процесів і рангу процесу. Додати ці виклики, відкомпілювати і запустити програму.

```
#include «mpi.h»
#include <stdio.h>
int main(int argc,char *argv[])
{
 int myid, numprocs;
 fprintf(stdout,»Process %d  %d\n», myid, numprocs);
 MPI_Finalize();
 return 0;
}
```

   b) У початковому тексті програми на мові C++ пропущені виклики процедур стандартного блокуючого двоточкового обміну. Передбачається, що при запуску двох процесів один з них відправляє повідомлення іншому. Додати ці виклики, відкомпілювати і запустити програму.

```
#include «mpi.h»
#include <stdio.h>
int main(int argc,char *argv[])
```

```
{
  int myid, numprocs;
  char message[20];
  int myrank; MPI_Status status;
  int TAG = 0; MPI_Init(&argc &argv);
  MPI_Comm_rank(MPI_COMM_WORLD &myrank);
  if (myrank == 0)
   {
    strcpy(message, «Hi, Second Processor!»);
    MPI_Send(...);
   }
  else
   {
    MPI_Recv(...);
    printf(«received: %s\n», message);
   }
  MPI_Finalize();
  return 0;
}
```

c) У початковому тексті програми на мові C++ пропущені виклики процедур стандартного блокуючого двохточкового обміну. Передбачається, що при запуску парного числа процесів, ті з них, які мають парний ранг, відправляють повідомлення наступним по величині рангу процесам. Додати ці виклики, відкомпілювати і запустити програму.

```
#include «mpi.h»
#include <stdio.h>
int main(int argc,char *argv[])
{
  int myrank, size, message;
  int TAG = 0;
  MPI_Status status; MPI_Init(&argc &argv);
  MPI_Comm_rank(MPI_COMM_WORLD &myrank);
  MPI_Comm_size(MPI_COMM_WORLD &size);
  message = myrank;
  if((myrank % 2) == 0)
   {
    if((myrank + 1) != size) MPI_Send(...);
   }
  else
   {
    if(myrank != 0)
    MPI_Recv(...);
    printf(«received :%i\n», message);
   }
  MPI_Finalize();
  return 0;
}
```

3.3 Зміст звіту

1. Титульний аркуш.
2. Постановка задачі. Короткий опис алгоритму.

3. Лістінг MPI-програм.
4. Скріншоти результатів виконання програм.
5. Оцінка показників прискорення й ефективності послідовних і паралельних обчислень (у вигляді графіків).
6. Відповіді на будь-які 6 запитань (питання переписувати обов'язково).
7. Висновки з виконаної роботи.

3.4 Контрольні питання

3 Дайте визначення комунікатора.
4 Що таке повідомлення?
5 Що потрібно зробити, щоб передати повідомлення?
6 Для чого потрібна процедура MPI_INIT?
7 Перелічте мінімальний набір команд для найпростішої MPI-програми.
8 Що повертає MPI у разі успішного завершення виклику?
9 Назвіть стандартні комунікатори MPI.
10 Яке призначення аргументу RANK у процедурі MPI_COMM_RANK(COMM, RANK, IERR)?
11 Як визначити, скільки процесів виконують програму до виклику процедури MPI_INIT та після візова процедури MPI_FINALIZE?
12 Яка процедура використовується для визначення рангу процесу?

ЛАБОРАТОРНА РОБОТА №4. Тема: Структура MPI-програми і процедури блокуючого двоточкового обміну MPI

Мета: *набуття навичок розробки програм проведення паралельних обчислень з використанням процедури блокуючого двоточкового обміну MPI, визначення часу виконання, оцінка показників прискорення та ефективності виконання розроблених програм.*

4.1 Теоретичні відомості

Модель паралельної програми в MPI

У моделі програмування MPI паралельна програма при запуску породжує декілька процесів, що взаємодіють між собою за допомогою повідомлень.

Сукупність всіх процесів, що становлять паралельний додаток, або їх частини, описується спеціальною структурою, яка називається ***комунікатором*** (областю взаємодії).

Кожному процесу в області взаємодії призначається унікальний числовий ідентифікатор – ранг, значення якого від 0 до $n_p - 1$ (n_p – число процесів). Ранги, що призначаються тому ж процесу в різних комунікаторах загалом різні.

Структура програми, яка написана за схемою «господар/працівник», наведена нижче.

```
program para
   if (ранг процесу = рангу майстер-процесу) then
       код майстер-процесу
     else
       код підлеглого процесу (підлеглих процесів)
   endif
end
```

Кожний екземпляр програми вже в процесі свого виконання визначає, чи є він майстер-процесом. Потім, залежно від результату цієї перевірки, виконується одна із гілок умовного оператора. Перша гілка відповідає майстер-задачі а друга – підлеглій задачі. Способи взаємодії між підзадачами визначаються програмістом. Перед використовуванням процедур передачі повідомлень програма повинна підключитися до системи обміну повідомленнями.

Повідомлення

Повідомлення містить дані і службову інформацію, що пересилаються. Для того щоб передати повідомлення, необхідно вказати:

- ранг процесу-відправника повідомлення;
- адресу, за якою розміщуються дані процесу-відправника, що пересилаються;
- тип даних, що пересилаються;
- кількість даних;
- ранг процесу, який повинен одержати повідомлення;
- адресу, за якою повинні бути розміщені дані процесом-одержувачем;
- тег повідомлення;
- ідентифікатор комунікатора, що описує область взаємодії, всередині якої відбувається обмін.

Тег – це ціле число, що задається користувачем, від **0** до **32767**, яке відіграє роль ідентифікатора повідомлення і дозволяє розрізняти повідомлення, що приходять від одного процесу. Теги можуть використовуватися і для дотримання певного порядку прийому повідомлень.

Прийом повідомлення починається з підготовки буфера достатнього розміру. В цей буфер записуються дані, що приймаються.

Операція відправки або прийому повідомлення вважається завершеною, якщо програма може знову використовувати буфери повідомлень.

Різновиди обмінів повідомленнями

У MPI реалізовані різні види обмінів.

Перш за все, це ***двохточкові*** (задіяно тільки два процеси) і ***колективні*** (задіяно більше двох процесів).

Двохточкові обміни використовуються для організації локальних і неструктурованих комунікацій.

При виконанні глобальних операцій використовуються колективні обміни. Асинхронні комунікації реалізуються за допомогою запитів про отримання повідомлень.

Є ***декілька різновидів двохточкового обміну***.

Блокуючі прийом/передачу – припиняють виконання процесу на час прийому повідомлення.

Неблокуючі прийом/передачу – виконання процесу продовжується у фоновому режимі, а програма в потрібний момент може запитати підтвердження завершення прийому повідомлення.

Синхронний обмін – супроводжується повідомленням про закінчення прийому повідомлення.

Асинхронний обмін – повідомленням не супроводжується.

Двохточкові блокуючі обміни

Учасниками двохточкового обміну є два процеси: процес-відправник і процес-отримувач (рис. 4.1).

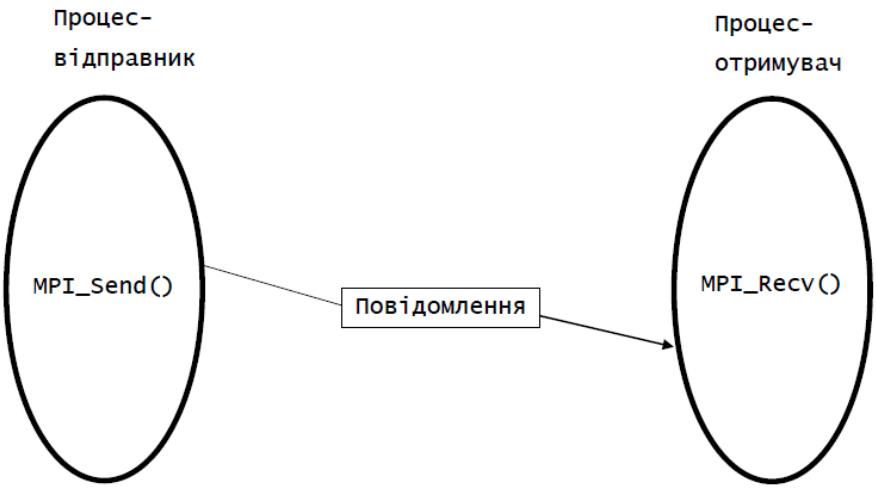


Рис. 4.1 Учасники двоточкового обміну

Блокуючі операції двохточкового обміну припиняють виконання процесу, який викликається. Він переходить е стан очікування завершення передачі даних. Блокування гарантує виконання дій в заданому порядку, забезпечуючи передбачуваність поведінки програми. З іншого боку, воно створює умови для виникнення тупикових ситуацій, коли обидва процеси-учасники обміну блокуються одночасно.

4.2. Завдання для виконання

Пропонується написати коди програм на мові C++ з використанням процедур MPICH, у тому числі, процедур блокуючого двоточкового обміну.

Завдання. Розробити паралельну програму, зробити вимір послідовного і паралельного часу виконання програми та оцінити показники прискорення і

ефективності паралельних обчислень. Початкові дані задати самостійно. Провести низку експериментів.

1. Для задачі скалярного добутку двох векторів

$$y = \sum_{i=0}^{N} a_i b_i$$

2. Для задачі добутку матриці на вектор.
3. Для задачі пошуку максимального і мінімального значень для заданого набору числових даних

$$y_{min} = \min_{1 \le i \le N} a_i, y_{max} = \max_{1 \le i \le N} a_i$$

4. Для задачі знаходження середнього арифметичного значення для заданого набору числових даних

$$y = \frac{1}{N} \sum_{i=0}^{N} a_i$$

Таблиця 4.1

Варіанти завдань

Задача	Виконують студенти за номерами прізвищ у журналі
1, 3	1, 6, 11,16, 21
1, 4	2, 7, 12, 17, 22
2, 3	3, 8, 13, 18, 23
2, 4	4, 9, 14, 19, 24
3. 4	5,10, 15, 20, 25

4.3 Зміст звіту

1. Титульний аркуш.
2. Постановка задачі. Короткий опис алгоритму.
3. Листінги MPI-програм.
4. Скріншоти результатів виконання програм.
5. Оцінка показників прискорення й ефективності послідовних і паралельних обчислень (у вигляді графіків).
6. Відповіді на будь-які 6 запитань (питання переписувати обов’язково).
7. Висновки з виконаної роботи.

4.4 Контрольні питання

1. Що таке комунікатор і чого він використовується?
2. Для пересилки повідомлення необхідно вказати (що?).
3. Які види повідомлень реалізовані в MPI? Їх характеристика.

4. Відповідність типів даних MPI стандартам мови C++.
5. Які стандартні домовленості про коди завершення виклику підпрограм (функцій) прийняті в MPI.
6. Структура і виклик функцій ініціалізації і завершення роботи MPI у мові C++.
7. Структура і виклик функцій розміру області взаємодії MPI у мові C++.
8. Як визначити ранг процесора?
9. Як визначити ім'я вузла, на якому виконується даний процес?
10. Двоточковий обмін. Що це?
11. Структура та формат виклику функції передачі повідомлень.
12. Структура та формат виклику функції прийому повідомлень.
13. Наведіть типову структуру MPI програми.
14. Що називають тегом?

ЛАБОРАТОРНА РОБОТА № 5. Тема: Вступ до паралельного програмування з використанням стандарту MPI для паралельної реалізації методів матричного множення

Мета: *закріплення знань і навичок розробки програм* ***паралельної обробки*** *за стандартом MPI і дослідження паралельних алгоритмів розв'язання складних обчислювальних задач (задач матричного множення)*.

5.1. Теоретичні відомості

Операція множення матриць є однією з основних задач матричних обчислень. В цій лабораторній роботі розглядаються кілька різних паралельних алгоритмів для виконання цієї операції. Два з них засновані на стрічковій схемі поділу даних. Інші два методи засновані на блоковій схемі поділу – це широко відомі алгоритми Фокса і Кенона.

Постановка задачі матричного множення

Множення матриці $\boldsymbol{A}$ розміру $\boldsymbol{m} \times \boldsymbol{n}$ і матриці $\boldsymbol{B}$ розміру $\boldsymbol{n} \times \boldsymbol{l}$ призводить до отримання матриці $\boldsymbol{C}$ розміру $\boldsymbol{m} \times \boldsymbol{l}$, кожен елемент якої визначається відповідно до виразу:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, \qquad 0 \le i \le m, 0 \le j \le l \tag{3.1}$$

Як випливає із (3.1), кожен елемент результуючої матриці $\boldsymbol{C}$ є скалярним добутком відповідних рядка матриці $\boldsymbol{A}$ і стовпця матриці $\boldsymbol{B}$:

$$c_{ij} = \left(a_i, {b_j}^T\right), a_i = \left(a_{i0,} a_{i1,} \dots, a_{in-1}\right), {b_j}^T = \left(b_{0j,} b_{1j,} \dots, b_{n-1,j}\right)^T \tag{3.2}$$

Цей алгоритм передбачає виконання $m \cdot n \cdot l$ операцій множення і стільки ж операцій додавання елементів вихідних матриць. При множенні квадратних матриць розміру $n \times n$ кількість виконаних операцій має порядок $O(n^3)$. Відомі послідовні алгоритми множення матриць, що володіють меншою обчислювальною складністю (наприклад, алгоритм Страсена), але ці алгоритми вимагають певних зусиль для їх освоєння. Для простоти викладу матеріалу припускатимемо, що матриці A і B, які перемножуються, є *квадратними* і мають порядок $n \times n$.

Послідовний алгоритм множення матриць задається трьома вкладеними циклами:

```
// Послідовний алгоритм множення матриць
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++)
{
  for (j=0; j<Size; j++)
  {
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++)
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
  }
}
```

Цей алгоритм є ітеративним і орієнтований на послідовне обчислення рядків матриці C. Дійсно, при виконанні однієї ітерації зовнішнього циклу (циклу по змінній i) обчислюється один рядок результуючої матриці (рис. 5.1).

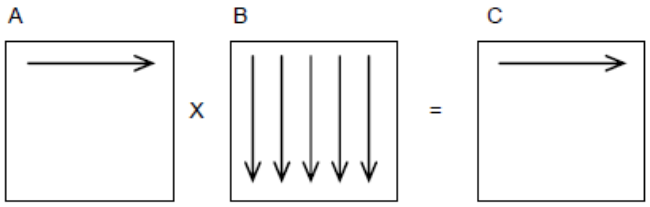


Рис. 5.1 На першій ітерації циклу по змінній *i* використовується перший рядок матриці *A* і всі стовпці матриці *B* для того, щоб обчислити елементи першого рядка результуючої матриці *C*

Оскільки кожен елемент результуючої матриці є скалярним добутком рядка і стовпця вихідних матриць, то для обчислення всіх елементів результуючої матриці C розміром $n \times n$ необхідно виконати $n^2 \cdot (2n-1)$ скалярних операцій і затратити час

$$T_1 = n^2(2n-1)\tau \qquad (3.3)$$

де $\boldsymbol{\tau}$ – час виконання однієї елементарної скалярної операції.

Основу можливості паралельних обчислень для матричного множення складає незалежність розрахунків для отримання елементів $\boldsymbol{c_{ij}}$ результуючої матриці $\boldsymbol{C}$. Отже, всі елементи матриці $\boldsymbol{C}$ можуть бути обчислені паралельно за наявності $\boldsymbol{n^2}$ процесорів, при цьому на кожному процесорі розташовуватиметься по одному рядку матриці $\boldsymbol{A}$ і одному стовпцю матриці $\boldsymbol{B}$. При меншій кількості процесорів подібний підхід приводить до стрічкової схеми розбиття даних, коли на процесорах розташовуються по декілька рядків і стовпців початкових матриць.

При побудові паралельних способів виконання матричного множення поряд з розглядом матриць у вигляді наборів рядків і стовпців широко використовується блокове представлення матриць. Більш докладно розглянемо цей спосіб організації обчислень.

При такому способі поділу даних вихідні матриці $\boldsymbol{A}$, $\boldsymbol{B}$ і результуюча матриця $\boldsymbol{C}$ представляються у вигляді наборів блоків. Для більш простого викладу наступного матеріалу будемо припускати далі, що всі матриці є *квадратними* розміру $\boldsymbol{n \times n}$, кількість блоків по горизонталі і вертикалі є однаковим і дорівнює $\boldsymbol{q}$ (тобто розмір всіх блоків дорівнює $\boldsymbol{k \times k}$, $\boldsymbol{k = n / q}$). При такому поданні даних операція матричного множення матриць $\boldsymbol{A}$ і $\boldsymbol{B}$ в блоковому вигляді може бути представлена у вигляді:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0,q-1} \\ & & \dots & \\ A_{q-1,0} & A_{q-1,1} & \dots & A_{q-1,q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0,q-1} \\ & & \dots & \\ B_{q-1,0} & B_{q-1,1} & \dots & B_{q-1,q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0,q-1} \\ & & \dots & \\ C_{q-1,0} & C_{q-1,1} & \dots & C_{q-1,q-1} \end{pmatrix}$$

де кожний блок $\boldsymbol{C_{ij}}$ матриці $\boldsymbol{C}$ визначається відповідно до виразу:

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} \cdot B_{sj},$$

При блочному розбитті даних для визначення базових підзадач природним представляється узяти за основу обчислення, що виконуються над матричними блоками. З урахуванням сказаного визначимо базову підзадачу як процедуру обчислення всіх елементів одного з блоків матриці $\boldsymbol{C}$.

Отримані блоки $\boldsymbol{C_{ij}}$ також є незалежними, і, як результат, можливий підхід для паралельного виконання обчислень може полягати в розрахунках, пов'язаних з отриманням окремих блоків $\boldsymbol{C_{ij}}$, на різних процесорах. Використання подібного підходу дозволяє одержати багато ефективних паралельних методів множення блочно представлених матриць.

Для виконання всіх необхідних обчислень базовим підзадачам повинні бути доступні відповідні набори рядків матриці $\boldsymbol{A}$ і стовпців матриці $\boldsymbol{B}$. Розміщення всіх необхідних даних у кожній підзадачі неминуче призведе до дублювання і до значного зростання обсягу використовуваної пам'яті. В результаті обчислення повинні бути організовані таким чином, щоб у кожний поточний момент часу підзадачі містили лише частину необхідних для проведення розрахунків даних, а доступ до решти даних

забезпечувався б за допомогою передачі повідомлень. Один із можливих підходів – алгоритм Фокса. другий спосіб – алгоритм Кеннона.

Стрічковий алгоритм

При стрічковій схемі розділення даних початкові матриці розбиваються на горизонтальні (для матриці A) і вертикальні (для матриці B) смуги. Одержувані смуги розподіляються по процесорах, при цьому **на кожному з наявного набору процесорів розташовується тільки по одній смузі матриць A і B**. Перемноження смуг (дана операція може бути виконана процесорами паралельно) приводить до отримання частини блоків результуючої матриці C. Для обчислення блоків матриці C, що залишилися, співвідношення смуг матриць A і B на процесорах повинні бути змінені. В найпростішому вигляді це може бути забезпечено, наприклад, при **кільцевій топології обчислювальної мережі** (при кількості процесорів, що дорівнює кількості смуг) – в цьому випадку необхідна для матричного множення зміна положення даних може бути реалізована циклічним зсувом смуг матриці B по кільцю. Після багатократного виконання описаних дій (кількість необхідних повторень дорівнює числу процесорів) на кожному процесорі отримуємо набір блоків, що створює горизонтальну смугу матриці C.

Розглянута схема обчислень дозволяє визначити паралельний алгоритм матричного множення при стрічковій схемі розділення даних як ітераційну процедуру, на кожному кроці якої відбувається паралельне виконання операції перемноження смуг і подальшого циклічного зсуву смуг однієї з матриць по кільцю.

Аналіз ефективності

Виконаємо аналіз ефективності першого паралельного алгоритму множення матриць.

Загальна трудомісткість послідовного алгоритму, як уже зазначалося раніше, є пропорційною n^3. Для паралельного алгоритму на кожній ітерації кожен процесор виконує множення наявних на процесорі смуг матриці A і матриці B (розмір смуг дорівнює $\frac{n}{p}$ і, як результат, загальна кількість виконуваних при цьому множенні операцій дорівнює $\frac{n^3}{p^2}$). Оскільки число ітерацій алгоритму збігається з кількістю процесорів, складність паралельного алгоритму без урахування витрат на передачу даних може бути визначена за допомогою виразу

$$T_p = \frac{n^3}{p^2} \cdot p = \frac{n^3}{p}.$$

З урахуванням цієї оцінки показники прискорення і ефективності даного паралельного алгоритму матричного множення приймають вид:

$$S_p = \frac{n^3}{\frac{n^3}{p}} = p,\ E_p = \frac{n^3}{p \cdot \frac{n^3}{p}} = 1.$$

Таким чином, загальний аналіз складності дає ідеальні показники ефективності паралельних обчислень.

Блокові алгоритми Фокса і Кенона

При блоковому представленні даних паралельна обчислювальна схема матричного множення в найпростішому вигляді може бути побудована, якщо *топологія обчислювальної мережі має вигляд прямокутної решітки* (якщо реальна топологія мережі має інший вигляд, представлення мережі у вигляді решітки можна забезпечити на логічному рівні). Основні положення паралельних методів для блочно представлених матриць такі:

1. Кожний із процесорів решітки відповідає за обчислення одного блоку матриці ***C***;
2. У ході обчислень на кожному з процесорів розташовується по одному блоку початкових матриць ***A*** і ***B***;
3. При виконанні ітерацій алгоритмів блоки матриці ***A*** послідовно зсуваються уздовж рядків процесорної решітки, а блоки матриці ***B*** - уздовж стовпців решітки;
4. У результаті обчислень на кожному з процесорів отримуємо блок матриці ***C***, при цьому загальна кількість ітерацій алгоритму дорівнює ***p*** (де ***p*** – число процесорів).

Алгоритм Фокса множення матриць при блоковому розділенні даних

Для більш простого пояснення наступного матеріалу припускатимемо далі, що всі матриці є квадратними розміром ***n×n***, кількість блоків по горизонталі і вертикалі однакова і дорівнює ***q*** (тобто розмір всіх блоків дорівнює ***k×k***, ***k=n/q***). При такому представленні даних операція матричного множення матриць ***A*** і ***B*** в блочному вигляді може бути представлена так:

$$\begin{pmatrix} A_{00} & A_{01} & \dots & A_{0,q-1} \\ & & \dots & \\ A_{q-1,0} & A_{q-1,1} & \dots & A_{q-1,q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} & \dots & B_{0,q-1} \\ & & \dots & \\ B_{q-1,0} & B_{q-1,1} & \dots & B_{q-1,q-1} \end{pmatrix} = \begin{pmatrix} C_{00} & C_{01} & \dots & C_{0,q-1} \\ & & \dots & \\ C_{q-1,0} & C_{q-1,1} & \dots & C_{q-1,q-1} \end{pmatrix},$$

де кожний блок C_{ij} матриці ***C*** визначається відповідно до виразу

$$C_{ij} = \sum_{s=0}^{qk_q-1} A_{is} \cdot B_{sj}.$$

При блочному розбитті даних для визначення базових підзадач природним представляється узяти за основу обчислення, що виконуються над матричними блоками. З урахуванням сказаного визначимо базову підзадачу як процедуру обчислення всіх елементів одного з блоків матриці ***C***.

Для виконання всіх необхідних обчислень базовим підзадачам повинні бути доступні відповідні набори рядків матриці ***A*** і стовпців матриці ***B***. Розміщення всіх необхідних даних в кожній підзадачі неминуче приведе до дублювання і до значного зростання об'єму пам'яті, що використовується. В результаті, обчислення повинні бути організовані так, щоб у кожний поточний момент часу підзадачі містили лише

частину необхідних для проведення розрахунків даних, а доступ до решти частини даних забезпечувався б за допомогою передачі даних між процесорами.

Отже, за основу паралельних обчислень для матричного множення при блочному розділенні даних прийнятий підхід, при якому базові підзадачі відповідають за обчислення окремих блоків матриці ***C*** і при цьому в підзадачах на кожній ітерації розрахунків розміщується тільки по одному блоку початкових матриць ***A*** і ***B***. Для нумерації підзадач використаємо індекси блоків матриці ***C***, які розміщуються в підзадачах, тобто підзадача (***i,j***) відповідає за обчислення блоку C_{ij} – тим самим, набір підзадач утворює квадратну решітку, що відповідає структурі блочного представлення матриці ***C***.

Відповідно до алгоритму Фокса під час обчислень на кожній базовій підзадачі (***i,j***) розташовується чотири матричні блоки:

– блок C_{ij} матриці ***C***, що обчислюється підзадачею;

– блок A_{ij} матриці ***A***, який розміщується в підзадачі перед початком обчислень;

– блоки A'_{ij}, B'_{ij} матриць ***A*** і ***B***, які отримуються підзадачею під час виконання обчислень.

Виконання паралельного методу включає:

– ***етап ініціалізації***, на якому кожній підзадачі (***i, j***) передаються блоки A_{ij}, B_{ij} і обнуляються блоки C_{ij} на всіх підзадачах;

– ***етап обчислень***, в рамках якого на кожній ітерації l, $0 \le l < q$, здійснюються наступні операції:

- для кожного рядка i, $0 \le i < q$, блок A_{ij} підзадачі (***i, j***) пересилається на всі підзадачі того ж рядка і решітки; індекс ***j***, що визначає положення підзадачі в рядку, обчислюється відповідно до виразу

$$j = (i + l) \bmod q,$$

де mod – операція отримання залишку від цілочисельного ділення;

- отримані в результаті пересилок блоки A'_{ij}, B'_{ij} кожної підзадачі (***i, j***) перемножуються і додаються до блоку C_{ij}:

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

- блоки B'_{ij} кожної підзадачі (***i,j***) пересилаються підзадачам, що є сусідами зверху в стовпцях решітки підзадач (блоки підзадач з першого рядка решітки пересилаються підзадачам останнього рядка решітки).

Для пояснення цих правил паралельного методу на рис. 3 наведений стан блоків кожної підзадачі під час ході виконання ітерацій етапу обчислень (для решітки підзадач 2×2).

Масштабування і розподіл підзадач по процесорах

У розглянутій схемі паралельних обчислень кількість блоків може варіюватися залежно від вибору розміру блоків – ці розміри можуть бути підібрані таким чином,

щоб загальна кількість базових підзадач збігалася із числом процесорів p. Так, наприклад, в найбільш простому випадку, коли число процесорів представимо у вигляді $p = \delta^2$, тобто є повним квадратом, можна вибрати кількість блоків у матрицях по вертикалі і горизонталі рівним δ (тобто $q = \delta$). Такий спосіб визначення кількості блоків призводить до того, що обсяг обчислень в кожній підзадачі є однаковим і, тим самим, досягається повне балансування обчислювального навантаження між процесорами.

Для ефективного виконання алгоритму Фокса, в якому базові підзадачі представлені у вигляді квадратної решітки і під час обчислень виконуються операції передачі блоків по рядках і стовпцях решітки підзадач, найбільш адекватним рішенням є організація безлічі наявних процесорів також у вигляді квадратної решітки. В цьому випадку можна здійснити безпосереднє відображення набору підзадач на множину процесорів – базову підзадачу (i, j) слід розташовувати на процесорі P_{ij}. Необхідна структура мережі передачі даних може бути забезпечена на фізичному рівні, якщо топологія обчислювальної системи має вигляд решітки або повного графа.

Аналіз ефективності

Визначимо обчислювальну складність даного алгоритму Фокса. Побудова оцінок відбуватиметься за умови виконання всіх раніше висунутих припущень – всі *матриці є квадратними* розміру $n \times n$, кількість блоків по горизонталі і вертикалі є однаковим і дорівнює q (тобто розмір всіх блоків дорівнює $k \times k$, $k = n / q$), *процесори утворюють квадратну решітку* і їх кількість дорівнює $p = q^2$.

Як уже зазначалося, алгоритм Фокса вимагає для свого виконання q ітерацій, під час яких кожен процесор перемножує свої поточні блоки матриць A і B і додає результати множення до поточного значення блоку матриці C. З урахуванням висунутих припущень загальна кількість операцій, які виконуються при цьому, матиме порядок n^3/p. В результаті, показники прискорення й ефективності алгоритму мають вигляд:

$$S_p = \frac{n^3}{\frac{n^3}{p}} = p,\ E_p = \frac{n^3}{p \cdot \frac{n^3}{p}} = 1.$$

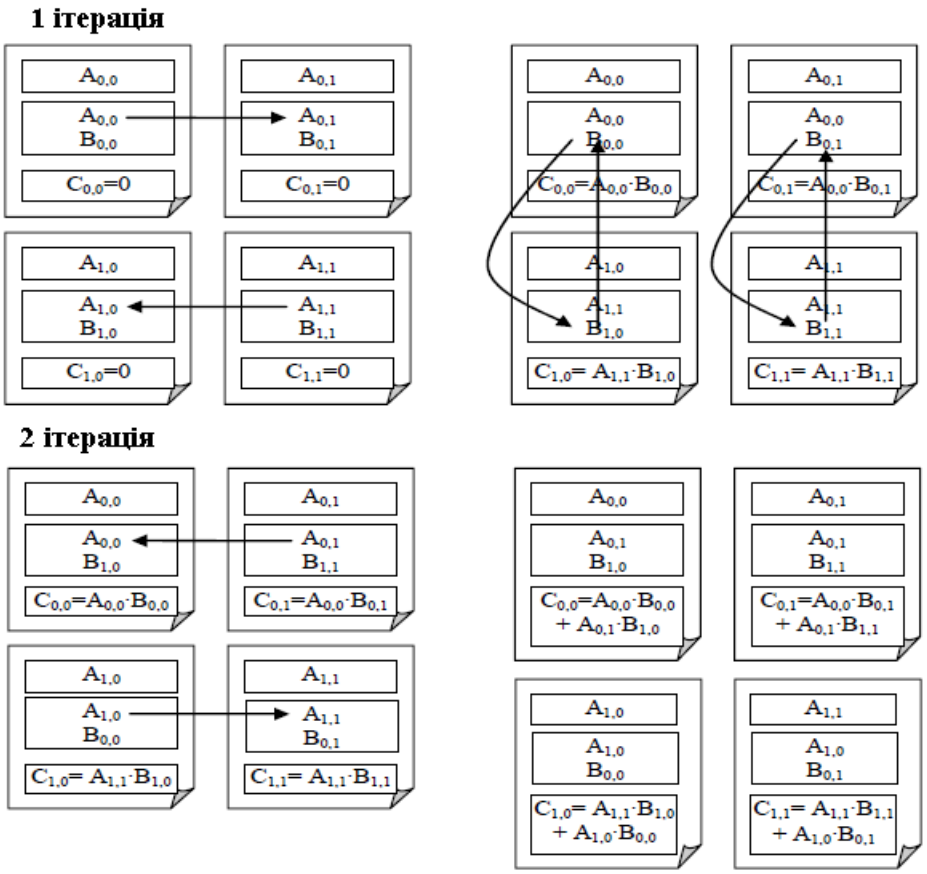


Рис. 5.2 Стан блоків у кожній підзадачі під час виконання ітерацій алгоритму Фокса

Алгоритм Кенона множення матриць при блочному розділенні даних

Як і при розгляді алгоритму Фокса, як базову підзадачу виберемо обчислення, пов'язані з визначенням одного з блоків результуючої матриці ***C***. Як вже наголошувалося раніше, для обчислення елементів цього блоку підзадача повинна мати доступ до елементів горизонтальної смуги матриці ***A*** і елементам вертикальної смуги матриці ***B***.

Відмінність алгоритму Кенона від методу Фокса полягає в зміні схеми початкового розподілу блоків матриць, які перемножуються, між підзадачами обчислювальної системи. Початкове розташування блоків в алгоритмі Кенона підбирається так, щоб блоки в підзадачах могли б бути перемножені без будь-яких додаткових передач даних. При цьому подібний розподіл блоків може бути організований таким чином, що переміщення блоків між підзадачами під час обчислень може здійснюватися з використанням більш простих комунікаційних операцій.

З урахуванням вказаних зауважень етап ініціалізації алгоритму Кенона включає виконання наступних операцій передач даних:

– в кожну підзадачу (***i, j***) передаються блоки A_{ij}, B_{ij};

– для кожного рядка ***i*** решітки підзадач блоки матриці ***A*** здвигаються на (***i-1***) позицій ліворуч;

– для кожного стовпця ***j*** решітки підзадач блоки матриці ***B*** здвигаються на (***j- 1***) позицій вверх.

Процедури передачі даних, що виконуються при перерозподілі матричних блоків, є прикладом операції *циклічного зсуву*. Для пояснення способу початкового розподілу даних, що використовується, на рис. 4 показаний приклад розташування блоків для решітки підзадач 3x3.

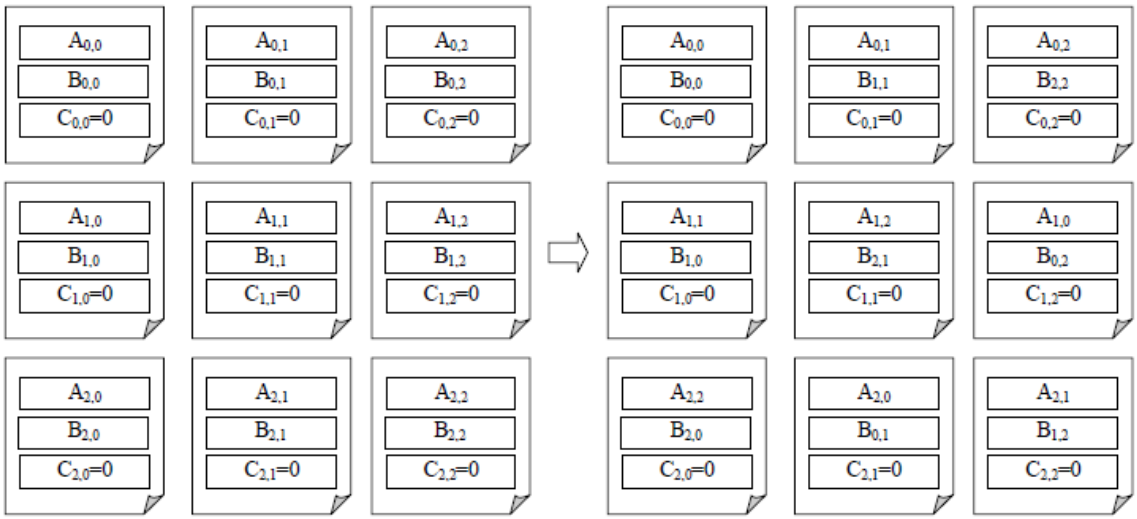


Рис. 5.3. Перерозподіл блоків початкових матриць між процесорами при виконанні алгоритму Кенона

В результаті такого початкового розподілу в кожній базовій підзадачі розташовуватимуться блоки, які можуть бути перемножені без додаткових операцій передачі даних.

Крім того, отримання всіх подальших блоків для підзадач може бути забезпечено за допомогою простих комунікаційних дій – після виконання операції блокового множення кожний блок матриці ***A*** повинен бути переданий попередній підзадачі ліворуч по рядках решітки підзадач, а кожний блок матриці ***B*** – попередній підзадачі вверх по стовпцях решітки. Можна показати, що послідовність таких циклічних зсувів і множення блоків початкових матриць ***A*** і ***B*** приведе до отримання в базових підзадачах відповідних блоків результуючої матриці ***C***.

Масштабування і розподіл підзадач по процесорах

Як і раніше в методі Фокса, для алгоритму Кенона розмір блоків може бути підібраний так, **щоб кількість базових підзадач співпадала з числом наявних процесорів**. Оскільки об'єми обчислень у кожній підзадачі одинакові, це забезпечує повне балансування обчислювального навантаження між процесорами.

Для розподілу підзадач між процесорами може бути застосований підхід, використаний в алгоритмі Фокса – множина наявних процесорів представляється у вигляді квадратної решітки і розміщення базових підзадач (***i,j***) здійснюється на процесорах $P_{i,j}$ відповідних вузлів процесорної решітки. Необхідна структура мережі передачі даних, як і раніше, може бути забезпечена на фізичному рівні при топології обчислювальної системи у вигляді решітки або повного графа.

Аналіз ефективності

Слід зазначити, що алгоритм Кеннона відрізняється від методу Фокса тільки видом виконуваних під час обчислень комунікаційних операцій. Таким чином, показники прискорення й ефективності алгоритму мають вигляд:

$$S_p = \frac{n^3}{\frac{n^3}{p}} = p,\ E_p = \frac{n^3}{p \cdot \frac{n^3}{p}} = 1.$$

5.2 Завдання для виконання

Стрічковий алгоритм.

1. Розробіть послідовну Сі-програму множення матриць стрічковим методом.
2. Виділіть інформаційні залежності і можливості програми по розпаралелюванню.
3. Розробіть паралельну MPI-програму множення матриць стрічковим методом.
4. Проведіть серію обчислювальних експериментів. Вивчіть залежність часу виконання алгоритму від об'єму початкових даних і від кількості процесорів. Побудуйте графіки залежності часу виконання алгоритму (прискорення і ефективності) від об'єму початкових даних і від кількості процесорів.
5. Зробіть висновки.

Блоковий алгоритм Фокса

6. Розробіть послідовну Сі-програму множення матриць алгоритмом Фокса.
7. Виділіть інформаційні залежності і можливості програми по розпаралелюванню.
8. Розробіть паралельну MPI-програму множення матриць даним алгоритмом.
9. Проведіть серію обчислювальних експериментів. Вивчіть залежність часу виконання алгоритму від об'єму початкових даних і від кількості процесорів. Побудуйте графіки залежності часу виконання алгоритму (прискорення і ефективності) від об'єму початкових даних і від кількості процесорів.
10. Зробіть висновки.

Блоковий алгоритм Кенона

11. Розробіть послідовну Сі-програму множення матриць алгоритмом Кенона.
12. Виділіть інформаційні залежності і можливості програми по розпаралелюванню.
13. Розробіть паралельну MPI-програму множення матриць даним алгоритмом.
14. Проведіть серію обчислювальних експериментів. Вивчіть залежність часу виконання алгоритму від об'єму початкових даних і від кількості процесорів. Побудуйте графіки залежності часу виконання алгоритму (прискорення і ефективності) від об'єму початкових даних і від кількості процесорів.
15. Зробіть висновки.
16. Зробіть загальні висновки з лабораторної роботи.
17. Оформіть звіт з лабораторної роботи такої структури:
    1) титульний аркуш;
    2) назва та мета виконання лабораторної роботи;

3) теоретичний опис, лістинг розроблених мовою C++ паралельної MPI-програми стрічкового алгоритму;
4) проведіть серію обчислювальних експериментів, побудуйте графіки залежності часу виконання алгоритму (прискорення і ефективності) від об'єму початкових даних і від кількості процесорів, зробіть висновки;
5) пункти 3, 4 для блокових алгоритмів Фокса та Кеннона;
6) дайте відповіді на будь-які 6 контрольних питань.

5.3. Контрольні питання

1. У чому полягає постановка задачі матрично-векторного множення?

2. Які є загальні способи розподілу даних, що представлені у вигляді матриць (масивів)?

3. Опишіть стрічкову схему розподілу матриці по рядкам для задачі матрично-векторного множення.

4. Опишіть стрічкову схему розподілу матриці по стовпцям для задачі матрично-векторного множення.

5. Опишіть блочну схему задачі матрично-векторного множення.

6. Яка топологія комунікаційної мережі є доцільною для кожного з розглянутих алгоритмів?

7. Який з розглянутих алгоритмів характеризується найменшими і найбільшими вимогами до обсягу необхідної пам'яті?

8. Який з розглянутих алгоритмів володіє найкращими показниками прискорення й ефективності?

9. Оцініть можливість виконання матричного множення як послідовності операцій множення матриці на вектор.

10. Які функції бібліотеки MPI виявилися необхідними при програмній реалізації алгоритмів?

ЛАБОРАТОРНА РОБОТА №6. Тема: Обмін даними в MPI. Розпаралелювання програм розв’язання систем лінійних рівнянь методом Гауса з використанням процедур колективного обміну.

Мета: *закріплення знань і навиків розробки програм* ***паралельної обробки*** *за стандартом MPI і дослідження паралельних алгоритмів розв’язання лінійних рівнянь.*

6.1 Теоретичні відомості

Системи лінійних рівнянь виникають при розв’язанні низки прикладних задач, що описуються диференціальними, інтегральними або системами нелінійних (трансцендентних) рівнянь. Вони можуть з'являтися також у задачах математичного програмування, статистичної обробки даних, апроксимації функцій, при дискретизації крайових диференціальних завдань методом кінцевих різниць або методом кінцевих елементів та ін.

Постановка задачі

Лінійне рівняння з n невідомими $x_0, x_1, \dots, x_{n-1}$ може бути визначене за допомогою виразу

$$a_0x_0 + a_1x_1 + \cdots + a_{n-1}x_{n-1} = b \qquad (3.1)$$

де $a_0, a_1, \dots a_{n-1}, b$ – сталі величини.

Множина ***n*** лінійних рівнянь називається **системою лінійних рівнянь** або **лінійної системою**. У більш короткому (матричному) вигляді система може представлена як

$$Ax = b,$$

де $\boldsymbol{A = (a_{i,j})}$ – дійсна матриця розміру $\boldsymbol{n \times n}$, а вектори ***b*** і ***x*** складаються з ***n*** елементів.

Під задачею розв’язання системи лінійних рівнянь для заданих матриці A і вектора b зазвичай розуміється знаходження значення вектора невідомих ***x***, при якому виконуються всі рівняння системи.

Алгоритм Гауса

Метод Гауса є широко відомим *прямим* алгоритмом розв’язання систем лінійних рівнянь, для яких матриці коефіцієнтів є *щільними*. Якщо система лінійних рівнянь є невиродженою, то метод Гауса гарантує знаходження розв’язку з похибкою, яка визначається точністю машинних обчислень. Основна ідея методу полягає в приведенні матриці ***A*** за допомогою еквівалентних перетворень (які не змінюють рішення системи (3.1)) до трикутного вигляду, після чого значення шуканих невідомих може бути отримано безпосередньо в явному вигляді.

Ми дамо загальну характеристику методу Гауса, яка достатня для початкового розуміння алгоритму і дозволяє розглянути можливі способи паралельних обчислень при розв’язанні систем лінійних рівнянь.

Послідовний алгоритм

Метод Гауса ґрунтується на можливості виконання перетворень лінійних рівнянь, які не змінюють при цьому розв’язання даної системи (такі перетворення називаються *еквівалентними*). До таких перетворень належать:

- множення будь-якого з рівнянь на ненульову константу;
- перестановка рівнянь;
- додавання до рівняння будь-якого іншого рівняння системи.

Метод Гауса включає послідовне виконання двох етапів.

– *На першому етапі* – ***прямий хід методу Гауса*** – початкова система лінійних рівнянь за допомогою послідовного виключення невідомих приводиться до верхнього трикутного вигляду

$$Ux = c,$$

де матриця коефіцієнтів системи має вигляд

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & \cdots & u_{0,n-1} \\ 0 & u_{1,1} & \cdots & u_{1,n-1} \\ & & \cdots & \\ 0 & 0 & \cdots & u_{n-1,n-1} \end{pmatrix}.$$

– На зворотному ході методу Гауса (*другий етап алгоритму*) здійснюється визначення значень невідомих. З останнього рівняння перетвореної системи може бути обчислено значення змінної x_{n-1}, після цього з передостаннього рівняння стає можливим визначення змінної x_{n-2} і т. д.

Прямий хід алгоритму Гауса

Прямий хід методу Гауса полягає в послідовному виключенні невідомих в рівняннях системи лінійних рівнянь, яка розв'язується. На ітерації ***i***, ***0≤i <n-1***, методу проводиться виключення невідомої ***i*** для всіх рівнянь з номерами ***k***, більшими за ***i*** (тобто ***i< k ≤ n-1***). Для цього з цих рівнянь здійснюється віднімання рядка ***i***, помноженого на константу $\left({}^{a_{ki}}/_{a_{ii}}\right)$, щоб результуючий коефіцієнт при невідомій x_i в рядках виявився нульовим – всі необхідні обчислення можуть бути визначені за допомогою співвідношень:

$$a'_{kj} = a_{kj} - \left({}^{a_{kj}}/_{a_{ii}}\right) \cdot a_{ij},$$

$$b'_k = b_k - \left({}^{a_{kj}}/_{a_{ii}}\right) \cdot b_i, i \le j \le n-1, i < k \le n-1.$$

Пояснимо виконання прямого ходу методу Гауса на прикладі системи лінійних рівнянь вигляду:

$$\begin{array}{llll} x_0 & + 3x_1 & + x_2 & = 1 \\ 2x_0 & + 7x_1 & + 5x_2 & = 18 \\ x_0 & + 4x_1 & + 6x_2 & = 26 \end{array}$$

На першій ітерації проводиться виключення невідомої ***x_0*** з другого і третього рядка. Для цього із цих рядків потрібно відняти перший рядок, помножений відповідно на ***2*** і ***1***. Після цих перетворень система рівнянь приймає вигляд:

$$\begin{array}{llll} x_0 & + 3x_1 & + 2x_2 & = 1 \end{array}$$

$$x_1 + x_2 = 16$$
$$x_1 + 4x_2 = 25$$

.

В результаті залишається виконати останню ітерацію і виключити невідому x_1 з третього рівняння. Для цього необхідно відняти другий рядок, і в остаточній формі система виглядає так:

$$x_0 + 3x_1 + 2x_2 = 1$$
$$x_1 + x_2 = 16$$
$$3x_2 = 9$$

.

На рис. 5 представлена загальна схема стану даних на ***i***-й ітерації прямого ходу алгоритму Гауса. Всі коефіцієнти при невідомих, розташовані нижче за головну діагональ і ліворуч стовпця ***i***, вже є нульовими. На ***i***-й ітерації прямого ходу методу Гауса здійснюється обнуління коефіцієнтів стовпця ***i***, розташованих нижче головної діагоналі, шляхом віднімання рядка ***i***, помноженого на потрібну ненульову константу. Після проведення (***n-1***) подібної ітерації матриця, що визначає систему лінійних рівнянь, стає приведеною до верхнього трикутного вигляду.

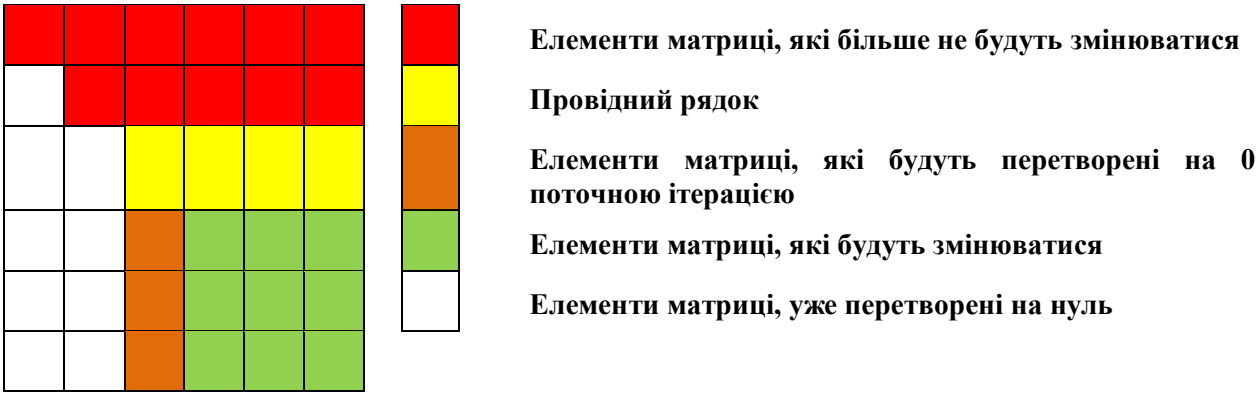


Рис. 6.1 Ітерація прямого ходу алгоритму Гауса

При виконанні прямого ходу методу Гауса рядок, який використовується для виключення невідомих, називається *провідним елементом*. Як бачимо, виконання обчислень є можливим тільки, якщо провідний елемент має ненульове значення. Більш того, якщо провідний елемент a_{ii} має мале значення, то ділення і множення рядків на цей елемент може приводити до накопичення обчислювальної погрішності та обчислювальної нестійкості алгоритму.

Можливий спосіб уникнути цієї проблеми може бути таким: при виконанні кожної чергової ітерації прямого ходу методу Гауса слід визначити коефіцієнт із максимальним значенням по абсолютній величині в стовпці, в якому знаходиться невідома, що виключається, тобто

$$y = \max_{i \le k \le n-1} |a_{ki}|,$$

і вибрати як провідний рядок, в якому цей коефіцієнт розташовується (така схема вибору провідного значення називається *методом головних елементів*).

Обчислювальна складність прямого ходу алгоритму Гауса з вибором провідного рядка має порядок $O(n^3)$.

Зворотний хід алгоритму Гауса

Після приведення матриці коефіцієнтів до верхнього трикутного вигляду стає можливим визначення значень невідомих. З останнього рівняння перетвореної системи може бути обчислено значення змінної x_{n-1}, після цього з передостаннього рівняння стає можливим визначення змінної x_{n-2} і т. д. В загальному вигляді обчислення, які виконуються при зворотному ході методу Гауса можуть бути представлені за допомогою співвідношень:

$$x_{n-1} = \frac{b_{n-1}}{a_{n-1,n-1}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j}{a_{ii}}, i = n\text{-}2, n\text{-}3, \ldots, 0 .$$

Пояснимо виконання зворотного ходу методу Гауса на прикладі системи лінійних рівнянь, розглянутій раніше:

$$\begin{aligned} x_0 &+ 3x_1 + 2x_2 = 1 \\ &\quad x_1 + x_2 = 16 \\ &\quad\quad 3\,x_2 = 9 \end{aligned}$$

Із останнього рівняння системи можна визначити, що невідома $x_2 = 3$. В результаті стає можливим розв'язок другого рівняння і визначення значення невідомої $x_1 = 13$, тобто

$$\begin{aligned} x_0 &+ 3x_1 + 2x_2 = 1 \\ &\quad x_1 = 13 \\ &\quad\quad x_2 = 3 \end{aligned}$$

На останній ітерації зворотного ходу методу Гауса визначається значення невідомої $x_0 = -44$.

З урахуванням подальшого паралельного виконання можна відзначити, що обчислення значень невідомих, що отримуються, може виконуватися відразу у всіх рівняннях системи (і ці дії можуть виконуватися в рівняннях одночасно і незалежно один від одного). Так, в даному прикладі після визначення значення невідомої x_2 система рівнянь може бути приведена до вигляду

$$\begin{aligned} x_0 &+ 3x_1 = -5 \\ &\quad x_1 = 13 \end{aligned}$$

$x_2 = 3$

Обчислювальна складність зворотного ходу алгоритму Гауса складає $O(n^2)$.

Визначення підзадач

При уважному розгляді методу Гауса можна помітити, що всі обчислення зводяться до однотипних обчислювальних операцій над рядками матриці коефіцієнтів системи лінійних рівнянь. Тому в основу паралельної реалізації алгоритму Гауса може бути покладений принцип розпаралелювання за даними. Як *базову підзадачу* можна прийняти тоді всі обчислення, пов'язані з обробкою одного рядка матриці A і відповідного елементу вектора b.

Виділення інформаційних залежностей

Розглянемо загальну схему паралельних обчислень і виникаючі при цьому інформаційні залежності між базовими підзадачами.

Для виконання прямого ходу методу Гауса необхідно здійснити ($n-1$) ітерацію по виключенню невідомих для перетворення матриці коефіцієнтів A до верхнього трикутного вигляду.

Виконання ітерації i, $0 \le i < n-1$, прямого ходу методу Гауса включає низку послідовних дій. Перш за все, на самому початку ітерації необхідно вибрати провідний рядок, який при використанні методу головних елементів визначається пошуком рядка з найбільшим по абсолютній величині значенням серед елементів стовпця i, відповідного змінній x_i, що виключається. Оскільки рядки матриці A розподілені по підзадачах, для пошуку максимального значення підзадачі з номерами k, $k<i$, повинні обмінятися своїми елементами при змінній x_i, що виключається. Після збору всіх необхідних даних в кожній підзадачі може бути визначено, яка з підзадач містить провідний рядок і яке значення є провідним елементом.

Далі для продовження обчислень провідна підзадача повинна розіслати свій рядок матриці A і відповідний елемент вектора b решті підзадач з номерами k, $k<i$. Одержавши провідний рядок, підзадачі виконують віднімання рядків, забезпечуючи тим самим виключення відповідної невідомої x_i.

При виконанні **зворотного ходу** методу Гауса підзадачі виконують необхідні обчислення для знаходження значення невідомих. Як тільки яка-небудь підзадача i, $0 \le i < n-1$, визначає значення своєї змінної x_i, це значення повинне бути розіслано всім підзадачам з номерами k, $k<i$. Далі підзадачі підставляють набуте значення нової невідомої і виконують коректування значень для елементів вектора b.

Масштабування і розподіл підзадач по процесорах

Виділені базові підзадачі характеризуються однаковою обчислювальною трудомісткістю і збалансованим об'ємом даних, що передаються. У разі, коли розмір матриці, що описує систему лінійних рівнянь, виявляється більшим, ніж число доступних процесорів (тобто $p<n$), базові підзадачі можна збільшити, об'єднавши в рамках однієї підзадачі декілька рядків матриці. Проте вживання послідовної схеми

розділення даних для паралельного розв'язання систем лінійних рівнянь приведе до нерівномірного обчислювального навантаження процесорів: на етапі виключення (на прямому ходу) або визначення (на зворотному ходу) невідомих в методі Гауса для дедалі все більшої частини процесорів всі необхідні обчислення будуть завершені і процесори будуть простоювати. Можливе розв'язання проблеми балансування обчислень може полягати у використанні стрічкової циклічної схеми для розподілу даних між збільшеними підзадачами. В цьому випадку матриця ***A*** ділиться на набори (смуги) рядків вигляду (рис. 6):

$$A = \left(A_0, A_1, \dots, A_{p-1}\right)^T, \qquad A_i = \left(a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}}\right), \qquad i_1 = i + jp, 0 \le j < k, k = \frac{n}{p}$$

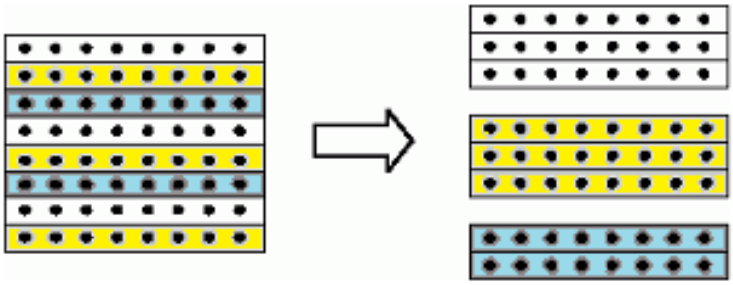


Рис. 6.2. Приклад використання стрічкової циклічної схеми розділення рядків матриці між трьома процесорами

Зіставивши схему розділення даних і порядок виконання обчислень у методі Гауса, можна відзначити, що використання циклічного способу формування смуг дозволяє забезпечити краще балансування обчислювального навантаження між підзадачами.

Розподіл підзадач між процесорами повинен враховувати характер комунікаційних операцій, які виконуютьсяв методі Гауса. *Основним видом інформаційної взаємодії підзадач є операція передачі даних від одного процесора всім процесорам обчислювальної системи*. Як результат, для ефективної реалізації необхідних інформаційних взаємодій між базовими підзадачами *топологія мережі* передачі даних повинні мати структуру *гіперкуба* або *повного графа*.

6.2 Завдання для виконання

Розробити послідовні і паралельні MPI-програми розв'язання системи лінійних рівнянь методом Гауса (алгоритм прямого ходу і зворотного ходу), зробити вимір послідовного і паралельного часу виконання програми й оцінити показники часу, прискорення і ефективності паралельних обчислень. Початкові дані задати самому. Провести низку експериментів.

Оформити звіт, до складу якого повинні бути включені:

а) титульний аркуш;
б) лістинги відкомпільованих MPI-програм;
в) результати виконання програм (скріншоти результатів);

г) наведена оцінка часу виконання паралельних обчислень та побудовані графіки залежностей;

д) відповіді на будь-які 4 контрольні питання у форматі: питання – відповідь;

д) висновки з роботи.

6.3. Контрольні питання

1. Що являє собою система лінійних рівнянь? Які типи систем вам відомі? Які методи можуть бути використані для розв’язання систем різних типів?
2. У чому полягає постановка задачі розв'язання системи лінійних рівнянь?
3. У чому ідея паралельної реалізації методу Гауса?
4. Які інформаційні взаємодії є між базовими підзадачами для паралельного варіанту методу Гауса?
5. Які показники ефективності для паралельного варіанта методу Гауса?
6. У чому полягає схема програмної реалізації паралельного варіанта методу Гауса?

ЛАБОРАТОРНА РОБОТА № 7. Обмін даними в MPI. Розпаралелювання програми обчислення визначеного інтегралу з використанням процедур колективного обміну.

Мета: *продовжити вивчення функцій обміну бібліотеки MPI, зокрема, функцій колективного обміну, засвоїти деякі прийоми їх використання для розподілу даних та обчислень між паралельними процесорами.*

7.1 Теоретичні відомості

Наближене обчислення визначеного інтегралу

Нехай безперервна і невід'ємна функція $f(x)$ визначена на відрізку $[a;b]$. Розділимо відрізок $[a;b]$ на n частин точками $a = x_0 < x_1 < x_2 < \cdots < x_n = b$ і виберемо на кожному відрізку $[x_{i-1}; x_i]$ довільну точку ξ_i.

Інтегральною сумою σ для функції *f(x)* на відрізку $[a;b]$ називається сума виду

$$\sigma = \sum_{i=1}^{n-1} f(\xi_i)\Delta x_i,$$

де $\Delta x_i = x_i - x_{i-1}$.

Визначеним інтегралом від функції $f(x)$ на відрізку $[a;b]$ називається границя інтегральної суми за умови, що довжина найбільшого з відрізків прагне до нуля $\max_i \Delta x_i \to 0$:

$$\int_a^b f(x)dx = \lim_{\Delta x \to 0} \sigma = \lim_{\Delta x \to 0} \sum_{i=1}^{n-1} f(\xi_i)\Delta x_i.$$

Метод трапецій

$$\int_a^b f(x)dx \qquad (5.1)$$

В основу ідеї **методу трапецій** покладено заміну кривої підінтегральної функції (5.1) на ламану. Цього можна досягнути таким чином. Розділимо відрізок $[a;b]$ на n рівних частин (довжина кожної частинки дорівнює $h=\frac{b-a}{n}$, і сполучимо прямими лініями значення функцій на кінцях відрізків, тобто **площу криволінійної трапеції** наближено замінюємо на суму площин ***n*** трапецій (рис.6.1).

Площу однієї такої трапеції можна обчислити за формулою

$$S_i = \frac{1}{2}\cdot h\cdot (f(x_i)+f(x_{i+1}))$$

.

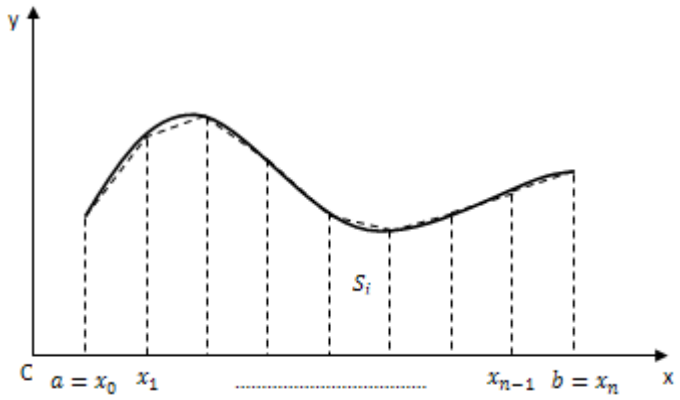


Рис. 7.1. Метод трапецій

А загальна площа *S* усіх *n* трапецій і, відповідно, *наближене значення інтеграла* дорівнює*:*

$$S=\sum_{i=0}^{n-1} S_i = \sum_{i=0}^{n-1}\frac{h}{2}\cdot\left(f(x_i)+f(x_{i+1})\right)=\frac{h}{2}\cdot\left(f(x_0)+f(x_n)+2\sum_{i=0}^{n-1} f(x_i)\right).$$

Якщо підставити граничні значення відрізка обчислення інтеграла, то формула набуде такого вигляду:

$$S=\frac{h}{2}\cdot\left(f(a)+f(b)+2\sum_{i=0}^{n-1} f(x_i)\right)$$

Метод Сімпсона

Розглянемо визначений інтеграл (6.1), де функція $f(x)$ визначена на відрізку $[a;b]$.

Якщо замінити функцію $f(x)$ квадратичним поліномом $P(x)$що набуває тих же значень що й $f(x)$ у точках a, b, $m = \frac{a+b}{2}$, використавши інтерполяційну формулу Лагранжа, то одержимо формулу:

$$P(x) = f(a) \cdot \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \cdot \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \cdot \frac{(x-a)(x-m)}{(b-a)(b-m)}.$$

Після необхідних обчислень отримуємо:

$$\int_a^b P(x)dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

Для точнішого обчислення інтеграла відрізок $[a;b]$ озбивають на *N* відрізків однакової довжини і застосовують формулу Сімпсона на кожному з них. Значення інтеграла є сумою для всіх відрізків.

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(\frac{1}{2}f(x_0) + 2\sum_{k=1}^{N-1} f(x_k) + 2\sum_{k=1}^{N} f\left(\frac{x_{k-1}+x_k}{2}\right) + \frac{1}{2}f(x_N)\right),$$

де $h = \frac{b-a}{2}$ – величина кроку, а x_{k-1}, x_k – межі відрізків.

Загальну похибку при інтегруванні на відрізку $[a;b]$ з кроком визначають за формулою:

$$|E(f)| \leq \frac{b-a}{2880} h^4 \max_{x\in[a;b]} \left|f^{(4)}(x)\right|.$$

За неможливості оцінити похибку за допомогою четвертої похідної можна використати слабшу оцінку:

$$|E(f)| \leq \frac{b-a}{2880} h^3 \max_{x\in[a;b]} \left|f^{(3)}(x)\right|.$$

Приклад

Розглянемо як приклад задачу обчислення визначеного інтегралу від функції $f(x)$ на відрізку $[a;b]$. Для прискорення роботи програми на обчислювальній установці з *p* процесорами ми скористаємося адитивністю інтегралу:

$$\int_a^b f(x)dx = \sum_{i=0}^{p-1} \int_{a_1}^{b_1} f(x)dx,$$

де $a_i = a + il,\ b_i = a_i + l, l = \frac{b-a}{p}$.

Використавши для наближеного обчислення кожного з доданків $\int_{a_1}^{b_1} f(x)dx$ цієї суми складену формулу трапецій, і проводячи ці обчислення на своєму процесорі, ми отримаємо *p*-кратне прискорення роботи. Найпростіша реалізація цієї задачі наведена нижче.

Кожен із процесів ініціалізує підсистему MPI за допомогою виклику `MPI_Init`, отримує свій номер і загальну кількість процесів за допомогою **MPI_Comm_rank** і `MPI_Comm_size`. Основна робота проводиться в функції `integrate`, після закінчення якої процес закінчує роботу з MPI за допомогою `MPI_Finalize` і завершується. Функція `integrate` обчислює свою частину інтегралу і додає його до відповіді `total` в процесі з номером 0 за допомогою `MPI_Reduce`. Процес з номером 0 перед закінченням своєї роботи виводить змінну `total` на екран.

Вхідні дані (межі відрізку і кількість точок розбиття інтервала) приймаються від користувача в процесі з номером 0. Потім ці дані розсилаються всім іншим процесам за допомогою `MPI_Bcast`.

Текст програми (мовою С)**:**

```
/* MPI_integral.c */
#include<stdio.h>
#include «mpi.h

/* Функція, яка інтегрується */

double f(double x)
{
     return x;
}

/* Обчислити інтеграл на відрізку [a, b] з числом точок розбиття n за
формулою трапецій */

double integrate(double a, double b, int n)
{
     double res;   /* результат */
     double h;      /* крок інтегрування */
     double x;
     int i;
     h = (b-a)/n;
     res = 0.5*(f(a)+f(b))*h;
     x = a;
     for (i=1 ; i<n; i++)
     {
          x += h;
          res += f(x)*h;
     }
     return res ;
}
int main(int argc, char *argv[])
{
     Int my_rank;           /* ранг поточного процесу */
     Int numprocs;         /* загальне число процесів */
     Double a;              /* ліва межа інтервала */
     Double b;              права межа інтервала*/
     Int n;                    /* кількість точок розбиття */
```

```
  doublele n;/* довжина відрізку інтегрування для поточного процесу*/
  double local_a;     /* ліва межа інтервала для поточного процесу */
  doublelocal_b;    /* права межа інтервала для поточного процесу */
  int local_n;      /* кількість точок розбиття поточного процесу */
  double local_res; /* значення інтеграла в поточному процесі */
  double result;      /* результат інтегрування */
  double wtime;     /* час роботи програми */
/* Почати роботу з MPI */
  MPI_Init(&argc, &argv);
/* Отримати номер поточного процесу в групі усіх процесів */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/* Отримати загальну кількість запущених процесів */
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
/* Отримати дані */
  if (my_rank= = 0)
  {
       printf(«Inputa: «);
       scanf(«%lf», &a);
       printf(«Inputb: «);
       scanf(«%lf», &b);
       printf(«Inputn: «);
       scanf(«%d», &n);
  }
/* Розсилаємо дані із процесу 0 решті процесів */
  MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
/* Синхронізація процесів */
  MPI_Barrier(MPI_C0MM_W0RLD);
/* Запускаємо таймер */
  wtime= MPI_Wtime();
/* Обчислюємо відрізок інтегрування для поточного процесу */
  len = (b-a)/numprocs;
  local_n = n/numprocs;
  local_a = a + my_rank*len;
  local_b = local_a + len;
/* Обчислити інтеграл на кожному із процесів */
  local_res = integrate(local_a, local_b, local_n);
/* Підсумувати усі відповіді і передати процесу 0 */
  MPI_Reduce(&local_res,&result,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
/* Синхронізація процесів */
  MPI_Barrier(MPI_COMM_WORLD);
/* Обчислення часу роботи */
  wtime= MPI_Wtime() - wtime;
/* Вивести на екран відповідь */
  if (my_rank== 0)
  {
       printf(«Integral from %.21f to %.21f = %.81f\n»,a, b, result);
       printf(«Working time: %.21f seconds\n», wtime);
  }
/* Закінчуємо роботу з MPI */
  MPI_Finalize();
  return 0;
```

```
}
```

Даний файл потрібно відкомпілювати, виправити помилки, якщо є, і запустити на виконання стандартним способом для MPI-програм.

Має сенс проекспериментувати з кількістю працюючих паралельно процесів, щоб з'ясувати, як змінюється час роботи програми.

7.2 Завдання до виконання

1. Уважно ознайомитись, вивчити та розібрати наведений приклад і виконати його на комп'ютері, зробивши необхідні зміни, характерні для мови C++.

2. Створити послідовні програми обчислення визначеного інтеграла (методами трапецій і Сімпсона) відповідно до варіанту (табл. 5.1).

3. Виконати розроблені програми, визначити час виконання програм. Початкові дані задати самому. Провести низку експериментів. Побудувати графіки, зробити висновки.

4. Розпаралелити розроблені програми за допомогою MPI. Визначити час, витрачений процесором на виконання розрахунків. Порівняти із результатами, одержаними в завданні 3.

5. Провести низку експериментів та визначити ефективність алгоритмів.

6. На підставі результатів, одержаних при виконанні завдань даної лабораторної роботи, написати звіт, в якому зробити висновки про ефективність різних способів розпаралелювання вихідного послідовного коду і трудомісткості реалізації цих способів на практиці. Побудувати графіки.

7. Оформити звіт, до складу якого повинно бути включено:

а) титульний аркуш;

б) лістинги відкомпільованих послідовних та паралельних MPI-програм;

в) результати виконання програм (скріншоти екранів);

г) оцінка часу виконання паралельних обчислень та побудовані графіки залежностей;

д) відповіді на 6 контрольних запитань у форматі «питання – відповідь»;

є) висновки за роботою.

Таблиця 7.1

Варіанти завдань

Варіант	Завдання	Варіант	Завдання
1	$\int_{-2}^{0}(x^2+5x+6)\cos 2x\ dx$	9	$\int_{0}^{2\pi}(3x^2+5)\cos 2x\,dx$
2	$\int_{-2}^{0}(x^2-4)\cos 3x\ dx$	10	$\int_{0}^{2\pi}(2x^2-15)\cos 3x\ dx$
3	$\int_{-1}^{0}(x^2+4x+3)\cos x\ dx$	11	$\int_{0}^{2\pi}(3-7x^2)\ \cos 2x\ dx$

4	$\int_{-2}^{0}(x+2)^2\cos 3x\ dx$	12	$\int_{0}^{2\pi}(1-8x^2)\cos 4x\ dx$
5	$\int_{-4}^{0}(x^2+7x+12)\cos x\ dx$	13	$\int_{-1}^{0}(x^2+2x+1)\cos 3x\ dx$
6	$\int_{0}^{\pi}(2x^2+4x+7)\cos 2x\ dx$	14	$\int_{0}^{3}(x^2-2x)\cos 2x\ dx$
7	$\int_{0}^{\pi}(9x^2+9x+11)\cos 3x\ dx$	15	$\int_{0}^{\pi}(x^2-3x+2)\cos x\ dx$
8	$\int_{0}^{\pi}(8x^2+16x+17)\cos 4x\ dx$	16	$\int_{0}^{\frac{\pi}{2}}(x^2-5x+6)\cos 3x\ dx$

7.3. Контрольні питання

1. Що покладено в основу метода трапецій?
2. Що покладено в основу метода Сімпсона?
3. У чому ідея паралельної реалізації методу трапецій?
4. У чому ідея паралельної реалізації методу Сімпсона?
5. Які існують види обмінів повідомленнями?
6. Які основні складові повідомлення?
7. Яка функція надає можливість визначити ранг (кількість) процесів? Її структура.
8. Яка функція надає можливість передати повідомлення від одного процесу до іншого? Її структура.
9. Яка функція надає можливість отримати повідомлення від одного процесу до іншого. Її структура.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна література

1. Синтез та аналіз паралельних процесів в адаптивних часопараметризованих обчислювальних системах: монографія / Г. А. Поляков, С. І. Шматков, Є. Г. Толстолузька, Д. А. Толстолузький. - Х: ХНУ імені В. Н. Каразіна, 2012, 672 с.

2. Van Steen M. Distributed Systems / M. van Steen, A. Tanenbaum. – New York: Pearson Education, Inc., 2017. – 596 с.

3. Andrews G. Foundations of Multithreading, Parallel and Distributed Programming. Addison-Wesley / Gregory R. Andrews., 2000. – 667 с.

4. Hughes C. Parallel and Distributed Programming Using C++ / C. Hughes, T. Hughes., 2003. – 720 с.

5. Аксак Н.Г. Паралельні та розподілені обчислення : пірдуч./ Н. Г. Аксак, О. Г. Руденко, А. М. Гуржій. – Х. :Компанія СМІТ, 2009. – 480с.

6.

Допоміжна література

1. Ortega J. lntroduction to Parallel and Vector Solutiori of Linear Systems / James M. Ortega. – New York: Springer, 1988. – 305 с.

2. Жуков І.А., Корочкін О.В. Паралельні та розподілені обчислення. Навч. посіб. – К.: «Корнійчук» , 2005. – 226 с.

3. Кузьма К. Т. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти [Електронний ресурс] / К. Т. Кузьма, О. В. Мельник // Миколаїв: ФОП Швець В.М.. – 2020. – Режим доступу до ресурсу: http://dspace.mdu.edu.ua/jspui/bitstream/123456789/860/1/%D0%9A%D1%83%D0%B7%D1%8C%D0%BC%D0%B0%2C%20%D0%9C%D0%B5%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA_%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%96%20%D1%82%D0%B0%20%D1%80%D0%BE%D0%B7%D0%BF%D0%BE%D0%B4%D1%96%D0%BB%D0%B5%D0%BD%D1%96%20%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F.pdf.

4. Навчальний посібник до вивченння курсу \"Паралельні та розподілені обчислення\" [Електронний ресурс] // РВВ ДНУ. – 2011. – Режим доступу до ресурсу: http://repository.dnu.dp.ua:1100/upload/0437891d38e3501a2c067570a5fcea63PP6.pdf.

5. Поляков Г. А. Проблеми створення систем спільного автоматичного проектування апаратно-програмних засобів для мультипаралельної цифрової обробки даних // Зб. наук. тр. 1-й Міжнародний радіоелектронний форум «Прикладна радіоелектроніка. Стан та перспективи розвитку» МРФ-2002. - Харків: АНПРЕ, ХНУРЕ, Ч.2, 2002. С.241 - 244.

Посилання на інформаційні ресурси в Інтернеті, відео-лекції, інше методичне забезпечення

1. TOP500 Supercomputing sites : Project ranks and details the 500 most powerful computer systems in the world. URL: http://www.top500.org .
2. http://www. cs. wisc. edu/condor/
3. http://setiathome. ssl. berkeley. edu/
4. http://www. Distributed. net/
5. http://mersenne. org/ 6. http://www. globus. org/

ДОДАТКИ

Додаток А. Приклади розв'язання задач

Обчислення часткових сум послідовності числових значень

Розглянемо низку проблем, що виникають при розробці паралельних методів обчислень, порівняно просту задачу знаходження часткових сум послідовності числових значень

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

де n – кількість значень, які підсумовуються (ця задача відома також під назвою prefix sum problem).

Вивчення можливих паралельних методів розв'язання даної задачі почнемо із ще більш простого варіанту її постановки – із завдання обчислення загальної суми наявного набору значень (в такому вигляді завдання підсумовування є окремим випадком загальної задачі редукції)

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

1. Послідовний алгоритм підсумовування.

Традиційний алгоритм для розв'язання даної задачі полягає в послідовному підсумовуванні елементів числового набору

S=0,
S=S+x_1,...

Обчислювальна схема даного алгоритму може бути представлена таким чином(рис.13):

$G_1=(V_1,R_1)$,

де $V_1=\{v_{01},...,v_{0n}, v_{11},...,v_{1n}\}$ – множина операцій (вершини $v_{01},...,v_{0n}$ позначають операції введення, кожна вершина v_{1i}, $1\le i\le n$, відповідає збільшенню значення x_i накопичуваної суми S);

$R_1=\{(v_{0i},v_{1i}),(v_{1i},v_{1i+1}),\ 1\le i\le n-1\}$ – множина дуг, що визначають інформаційні залежності операцій.

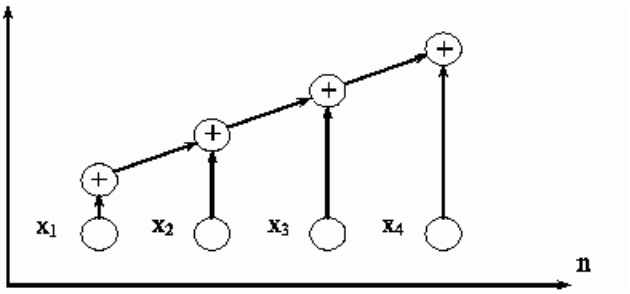


Рис. 13. Послідовна обчислювальна схема алгоритму підсумовування

Як можна помітити, даний «стандартний» алгоритм підсумовування передбачає лише послідовне виконання і не може бути розпаралеленим.

2. Каскадна схема підсумовування.

Паралелізм алгоритму підсумовування стає можливим тільки за іншого способу побудови процесу обчислень, заснованому на використанні асоціативності операції додавання. Варіант підсумовування, який отримується (відомий в літературі як каскадна схема) такий (рис. 14):

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

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

Дана обчислювальна схема може бути визначена як граф $G_2(V_2, R_2)$ (нехай n=2k)

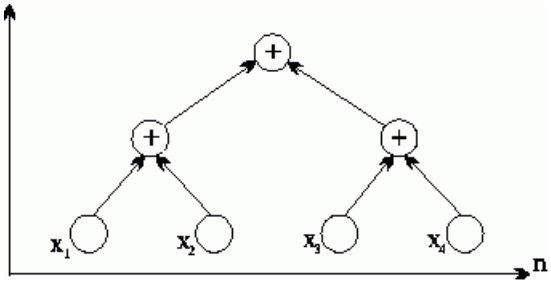


Рис. 14. Каскадна схема алгоритму підсумовування

де $V_2 = \{(v_{i1}, \dots, v_{1i}, 0 \le i \le k, 1 \le l_1 \le 2 - ln)\}$ – вершини графу ($(v_{01}, \dots, v_{0n})$ – операції введення, $(v_{1l}, \dots, v_{1n/2})$ – операції підсумовування першої ітерації і т. д.), а множина дуг графа визначається співвідношеннями:

$$R_2 = \{(v_{i-1,2j-1}v_{ij}), (v_{i-1,2j}v_{ij}), 1 \le i \le k, 1 \le j \le 2 - in)\}$$

Як неважко оцінити, кількість ітерацій каскадної схеми виявляється таким, що дорівнює величині

$$\boldsymbol{k = log_2 n,}$$

а загальна кількість операцій підсумовування

$$\boldsymbol{K_{\text{посл}} = \frac{n}{2} + \frac{n}{4} + \cdots + 1 = n - 1}$$

збігається з кількістю операцій послідовного варіанту алгоритму підсумовування. При паралельному виконанні окремих ітерацій каскадної схеми загальна кількість паралельних операцій підсумовування дорівнює:

$$\boldsymbol{K_{\text{пар}} = log_2 n.}$$

Оскільки вважається, що час виконання будь-яких обчислювальних операцій є однаковим і одиничним, то $\boldsymbol{T_1 = K_{\text{посл}}, T_p = K_{\text{пар}}}$, тому показники прискорення й ефективності каскадної схеми алгоритму підсумовування можна оцінити як

$$\boldsymbol{S_p = \frac{T_1}{T_p} = \frac{n - 1}{log_2 n}}$$

$$\boldsymbol{E_p = \frac{T_1}{pT_p} = \frac{n - 1}{p log_2 n} = \frac{n - 1}{\frac{n}{2} log_2 n}}$$

де $p = \frac{n}{2}$ – необхідна для виконання каскадної схеми кількість процесорів.

Додаток В. Бібліотека окремих функцій MPI

Існує декілька функцій, які використовуються в будь-якому додатку MPI та призначені для ініціалізації, коректного завершення та виконання необхідних дій після виявлення помилки за логікою роботи програми.

1. Ініціалізація бібліотеки та MPI-середовища.

Одна з перших функцій в тілі головної програми `main()` – це функція ініціализації MPI-середовища:

```
MPI_Init(&argc,&argv);
```

До неї передаються адреси аргументів, які стандартно отримуються самою `main()` від операційної системи та зберігають параметри командного рядка. В кінець командного рядка програми MPI-завантажувач `mpirun` додає ряд інформаційних параметрів, які необхідні `MPI_Init()`.

2. Аварійне завершення роботи MPI-середовища.

Викликається, якщо програма користувача повинна завершитися через помилки часу виконання, що пов'язані з MPI:

```
MPI_Abort(опис_області_зв'язку, код_помилки_MPI);
```

Виклик `MPI_Abort()` з будь-якої задачі примусово завершує роботу всіх задач, що під'єднані до заданої області зв'язку. Якщо вказано опис `MPI_COMM_WORLD`, то буде завершено весь додаток. Використовуйте код помилки `MPI_ERR_OTHER`, якщо не знаєте, як охарактеризувати помилку в класифікації MPI.

3. Нормальне закриття бібліотеки.

Останньою функцією будь-якого MPI-додатку є функція:

```
MPI_Finalize();
```

Дану функцію потрібно викликати перед поверненням з програми, тобто перед кожним оператором `return` в функції `main()`, який може бути викликаний після `MPI_Init()`.

4. Інформаційні функції.

Повідомляють розмір групи (тобто загальну кількість задач, під'єднаних до її області зв'язку) та порядковий номер задачі (процесу), що її викликає:

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

5. Функції пересилки даних.

Для організації простої пересилки даних між процесами використовуються функції:

```
int MPI_Send(buf,count,datatype,dest,tag,comm);
int MPI_Recv(buf,count,datatype,source,tag,comm,status);
```

де `void *buf` – адреса буфера, тобто початкова адреса буфера прийому (передачі). Кожний процес має власні набори даних та власний буфер прийому (передачі), тому адреси буферів кожного з процесів відрізняються одна від одної;

`int count` – розмір буфера в кількості комірок (не в байтах) типу `datatype`. Для функції передачі `MPI_Send()` вказується, скільки комірок потрібно передати, а для функції прийому `MPI_Recv()` – максимальна ємність приймального буфера. Якщо фактична довжина повідомлення, що надійшло, є меншою – останні комірки буфера не заповнюються, якщо більшою – виникне помилка часу виконання;

`MPI_Datatype datatype` – тип комірок буфера. Функції `MPI_Send()` та `MPI_Recv()` оперують масивами однотипних даних. Для опису базових типів мови С в MPI визначені константи `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE` та інші, які мають тип `MPI_Datatype`. Їхні назви утворюються префіксом *MPI_* та ім'ям відповідного типу (`int`, `char`, `double`, ...), що записуються великими літерами. Користувач може зареєструвати (визначити) в MPI-додатку свої власні типи даних, наприклад структури, після чого функції MPI зможуть обробляти їх таким же чином, як і базові типи;

`int dest(source)` – номер процесу призначення (приймача), з яким відбувається обмін даними;

`int tag` – ідентифікатор повідомлення, за допомогою якого одне повідомлення відрізняється від іншого. Ідентифікатор повідомлення – ціле число від 0 до 32767, яке призначається користувачем. Важливо, щоб відправлене повідомлення з призначеним номером, було прийнято з таким же номером;

`MPI_Comm comm` – опис області зв'язку (комунікатор);

`MPI_Status *status` – статус завершення прийому. За адресою *status* міститься інформація про прийнятеповідомлення, зокрема його ідентифікатор, номер процесу-передавача, код завершення та кількість фактично прийнятих даних.

5. Функції обміну в MPI.

Для обміну даними між процесами всередині заданої області взаємодії можуть використовуватись функції колективного обміну. Ці функції повинні викликатись у всіх процесах області взаємодії. Для розсилання одних й тих же даних від одного процесу до всіх інших використовується функція широкомовного розсилання:

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

де `buf` –адреса буфера;

`count` – кількість елементів даних у повідомленні;

`type` – тип даних;

`root` – ранг процесу, який виконує розсилання;

`comm` – комунікатор.

Для розподілу та збору даних використовуються, відповідно, такі функції, вони мають однакові аргументи:

```
int MPI_Scatter(void *sendbuf,int sendcount,
    MPI_Datatype sendtype,void *rcvbuf,int rcvcount,
    MPI_Datatype rcvtype,int root,MPI_Comm comm);

int MPI_Gather(void *sendbuf,int sendcount,
    MPI_Datatype sendtype,void *rcvbuf,int rcvcount,
    MPI_Datatype rcvtype,int root,MPI_Comm comm);
```

Функція розподілу **`MPI_Scatter()`** розсилає рівні частини буфера `sendbuf` процесу `root` всім процесам. При цьому зміст буфера процесу `root` розбивається на однакові частини за кількістю процесів, які беруть участь в обміні, кожна з яких складається з `sendcount` елементів. Перша частина поміщається до буфера `rcvbuf` процесу, ранг якого дорівнює нулю, друга – до буфера `rcvbuf` процесу, ранг якого дорівнює одиниці і т. д. Аргументи, що належать до тієї частини списку аргументів функції, яка передається, мають силу тільки для процесу `root`.

Функція **`MPI_Gather()`** має зворотню дію порівняно з функцією `MPI_Scatter()`, тобто вона приймає та розташовує за порядком прийняті дані з процесів, які передають. При цьому параметри прийому дійсні тільки для процесу, що приймає.

Наступна функція є векторною версією функції `MPI_Scatter()` та призначена для розсилання різним процесам різної кількості елементів даних.

```
int MPI_Scatterv(void *sendbuf, int *sendcounts,
int *displs, MPI_Datatype sendtype, void *rcvbuf,
int rcvcount, MPI_Datatype rcvtype, int root,
MPI_Comm comm);
```

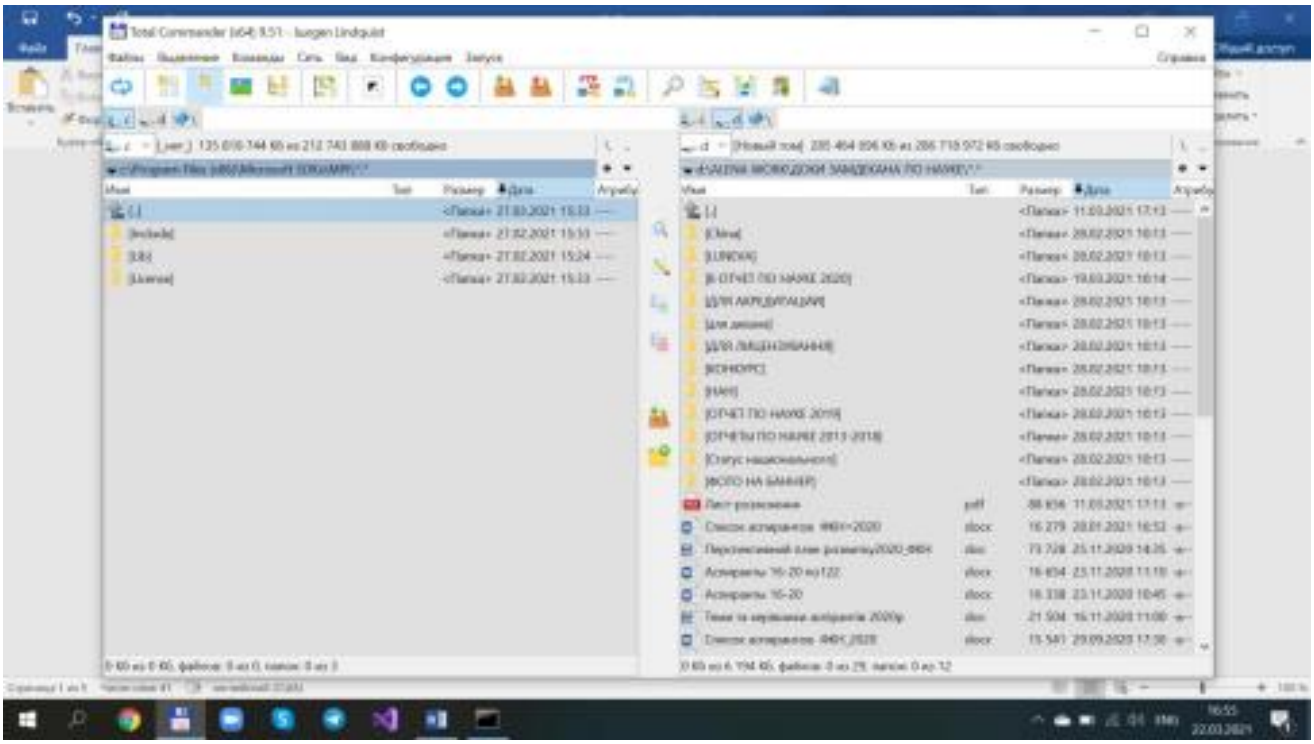
В **`MPI_Scatter()`** функції параметр `sendcounts` – масив цілих чисел, який містить кількість елементів, що передаються кожному процесу (індекс дорівнює рангу процесу). Параметр `displs` – масив цілих чисел, кожний з елементів якого задає зсув відносно початку буфера передачі. Таким чином, `displs[i]` – номер елемента буфера передачі, починаючи з якого будуть передані дані в *i*-й процес в кількості `sendcounts[i]` елементів.

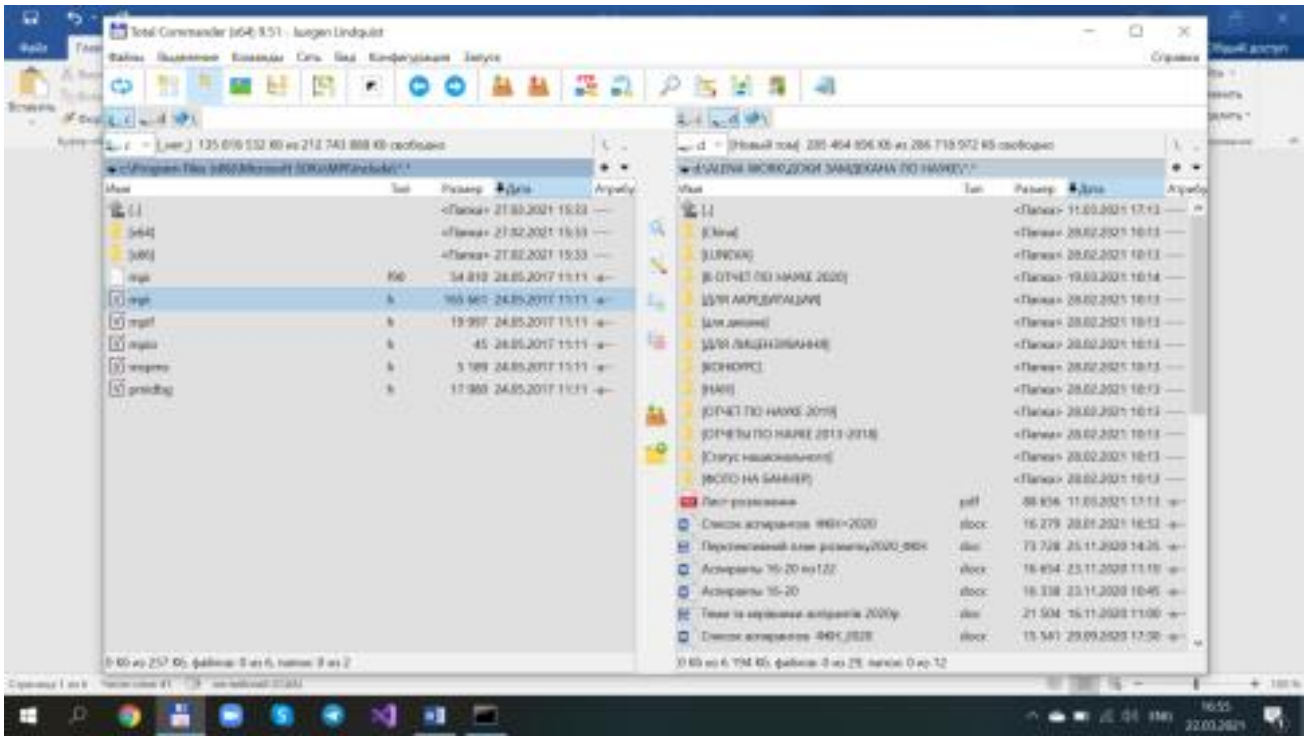
Функція зведення **`MPI_Reduce()`** оброблює елементи масиву даних таким чином. Функція бере один елемент від кожного процесу, виконує над ними задану операцію і розміщує результат у вказаному процесі. Синтаксис цієї функції:

```
MPI_Reduce(void *buf, void *result, int count,
MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm);
```

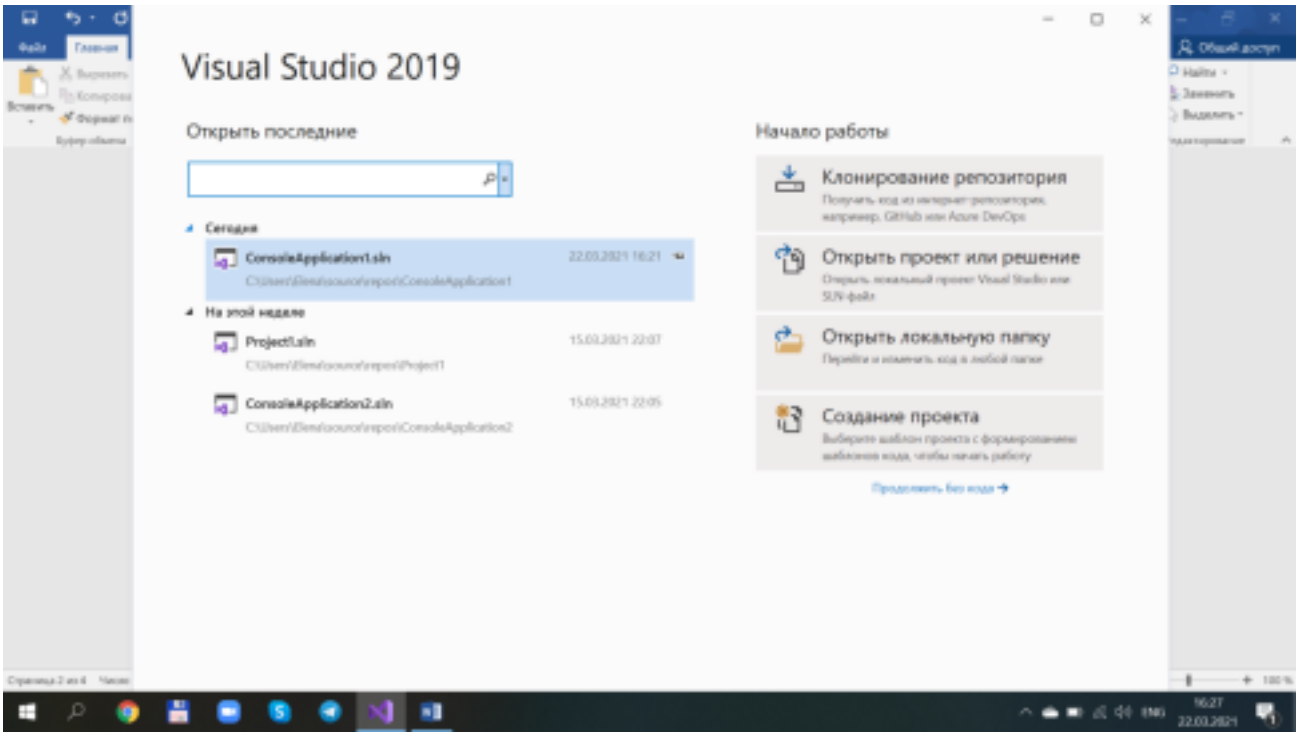
де `op` – операція зведення, яка може мати значення, що визначені попередньо, такі як `MPI_SUM`, `MPI_PROD`, `MPI_LAND`, `MPI_BAND`, `MPI_MAX` і т. д. Всього 12 операцій. Крім того, можна визначити свої власні операції зведення за допомогою функції **`MPI_Op_create()`**.

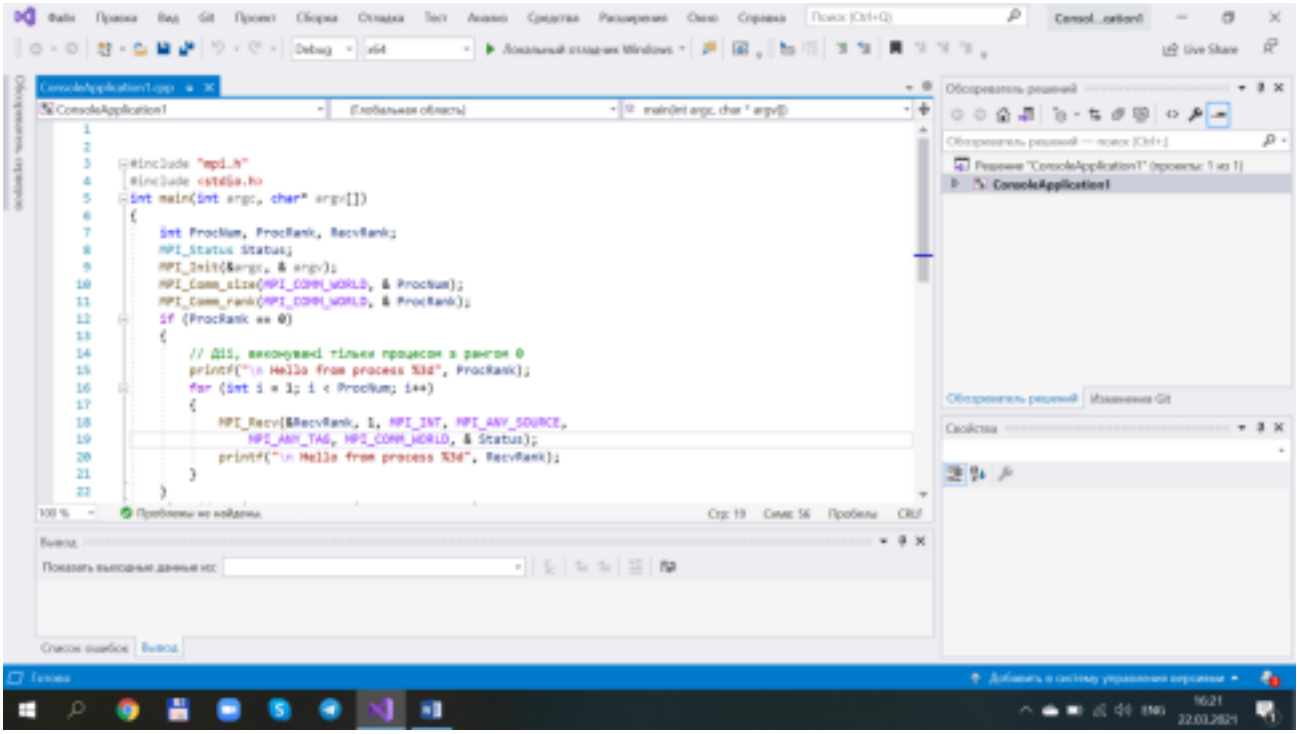
Додаток С. Налаштування проєкту в середовищі MS Visual Studio 2019

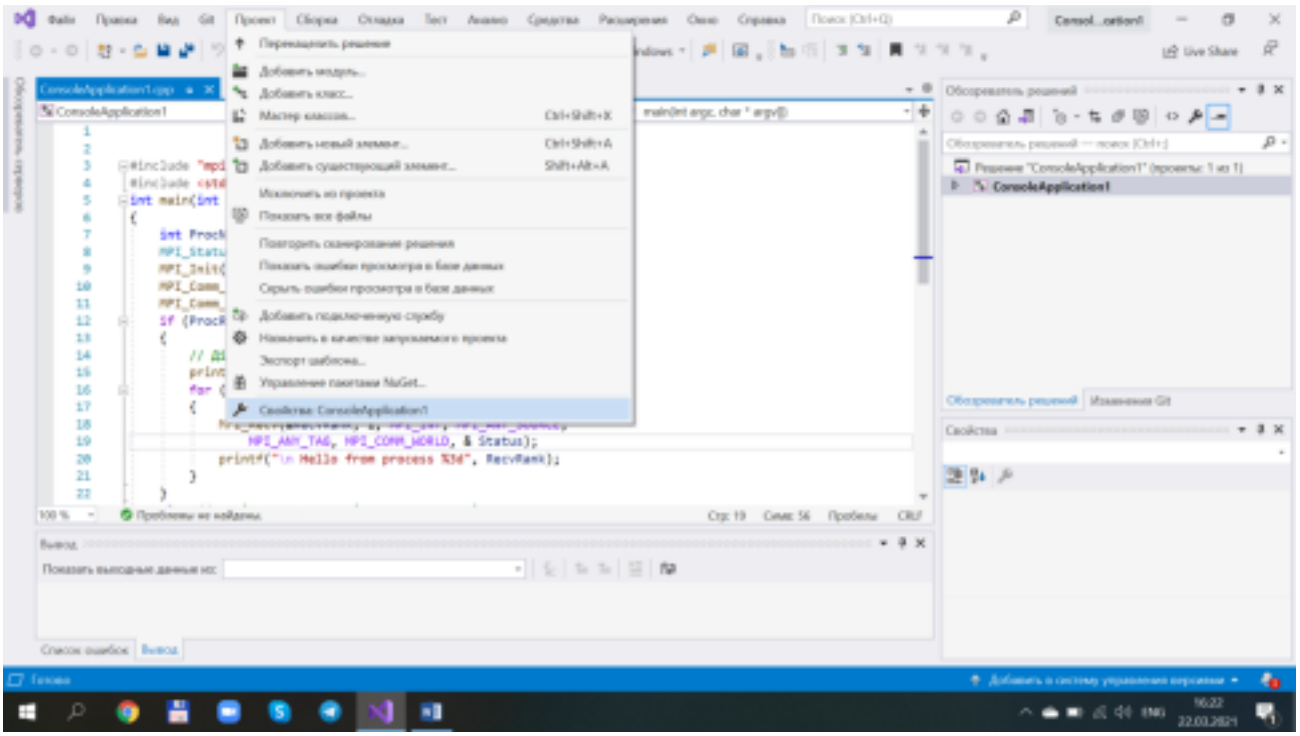


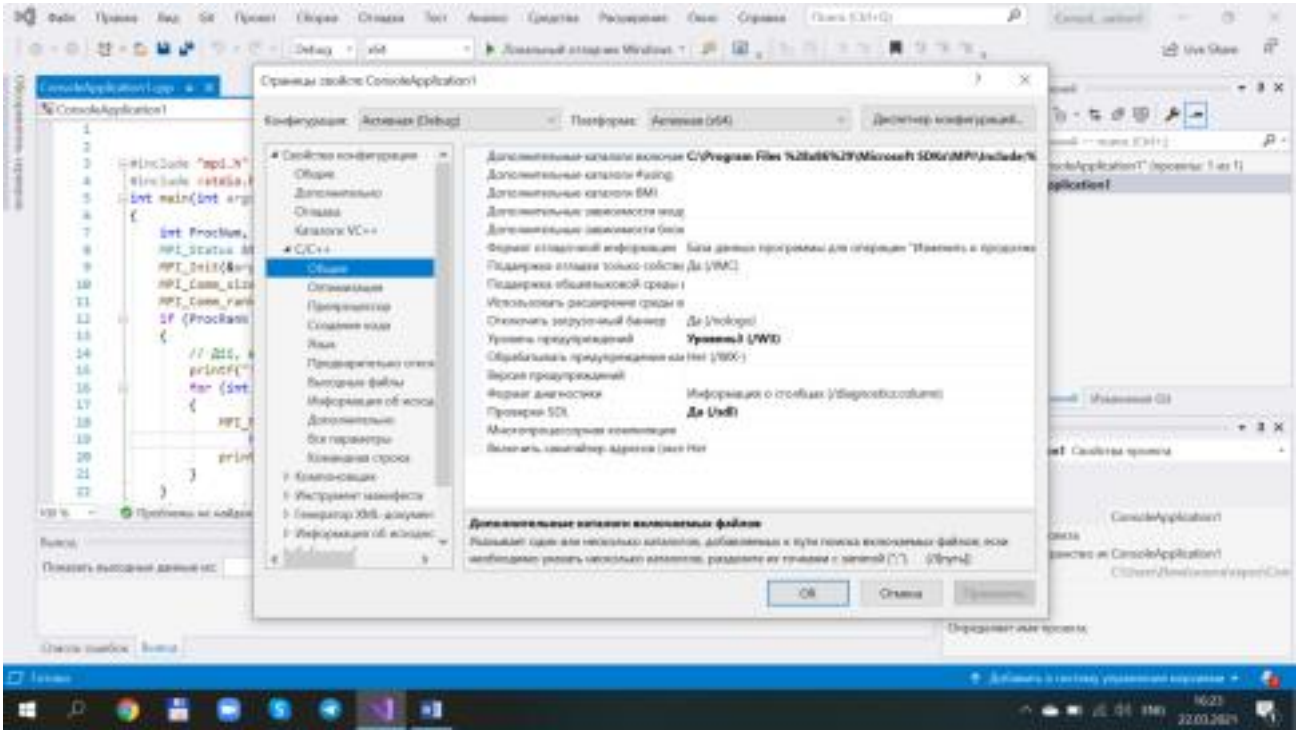


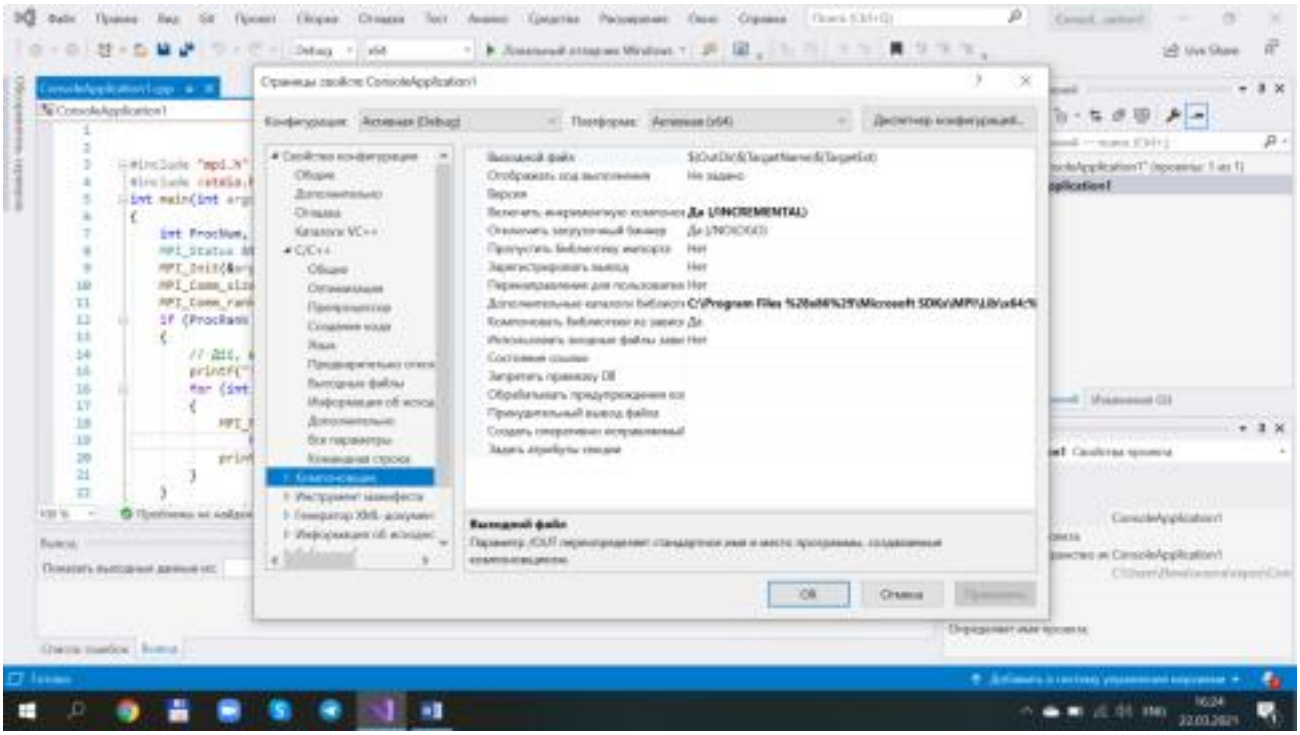

Visual Studio
2019

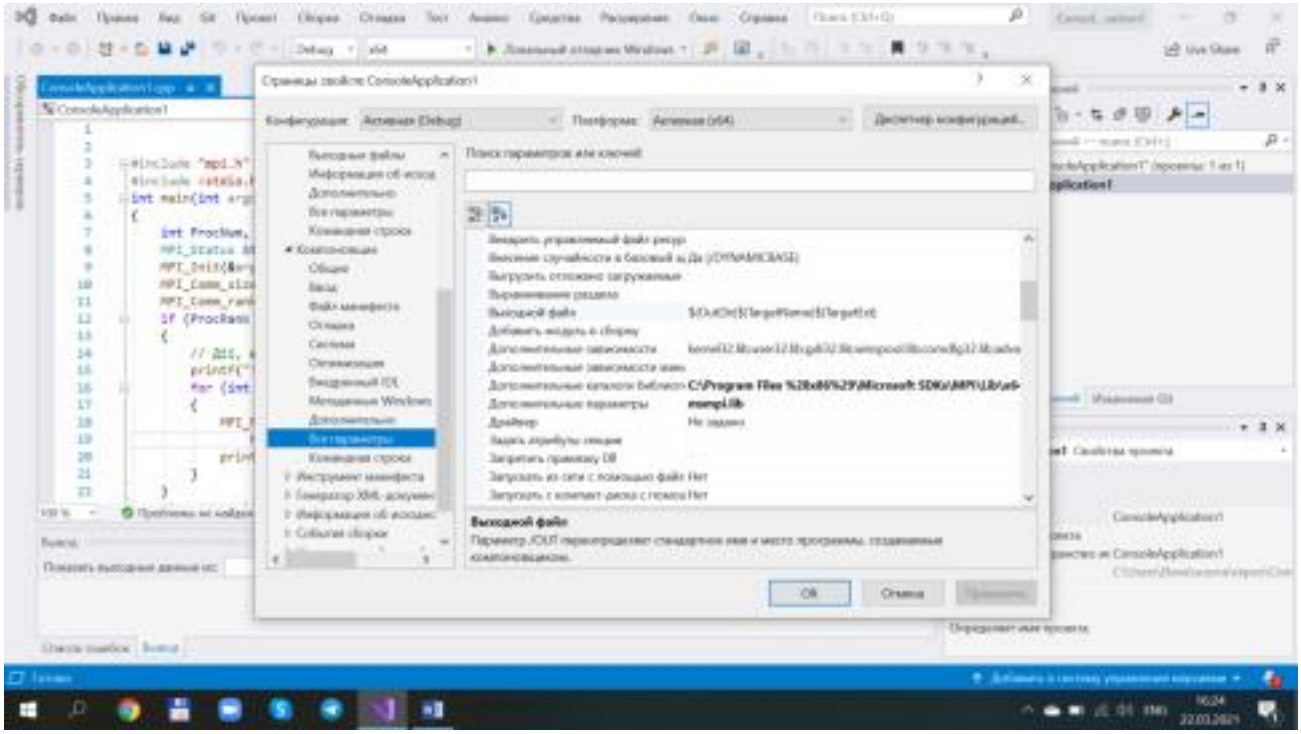

Visual Studio 2019
Открыть последние
Сегодня
ConsoleApplication1.sln
На этой неделе
Project1.sln
ConsoleApplication2.sln
Начало работы
Клонирование репозитория
Открыть проект или решение
Открыть локальную папку
Создание проекта

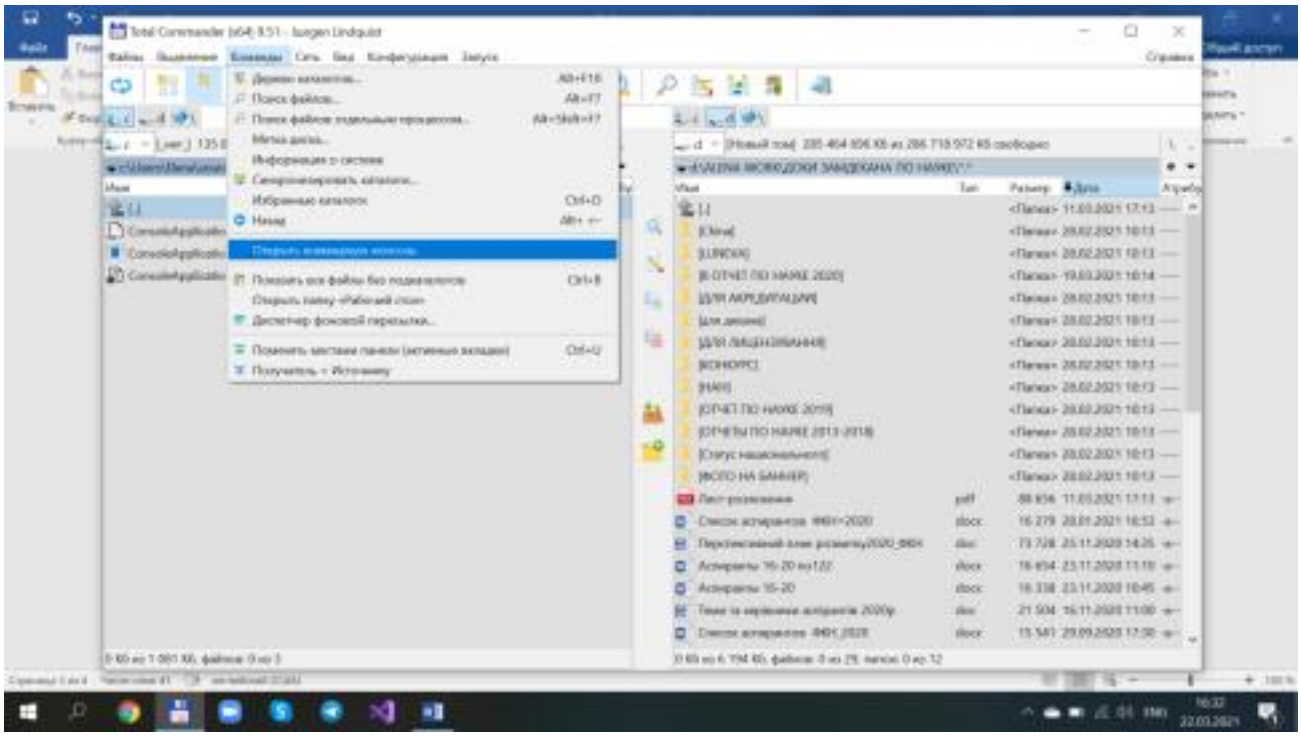

#include "mpi.h"
#include <stdio.h>
int main(int argc, char* argv[])
{
int ProcNum, ProcRank, RecvRank;
MPI_Status Status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if (ProcRank == 0)
{
// Дії, виконувані тільки процесом з рангом 0
printf("\n Hello from process %3d", ProcRank);
for (int i = 1; i < ProcNum; i++)
{
MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
printf("\n Hello from process %3d", RecvRank);
}
}

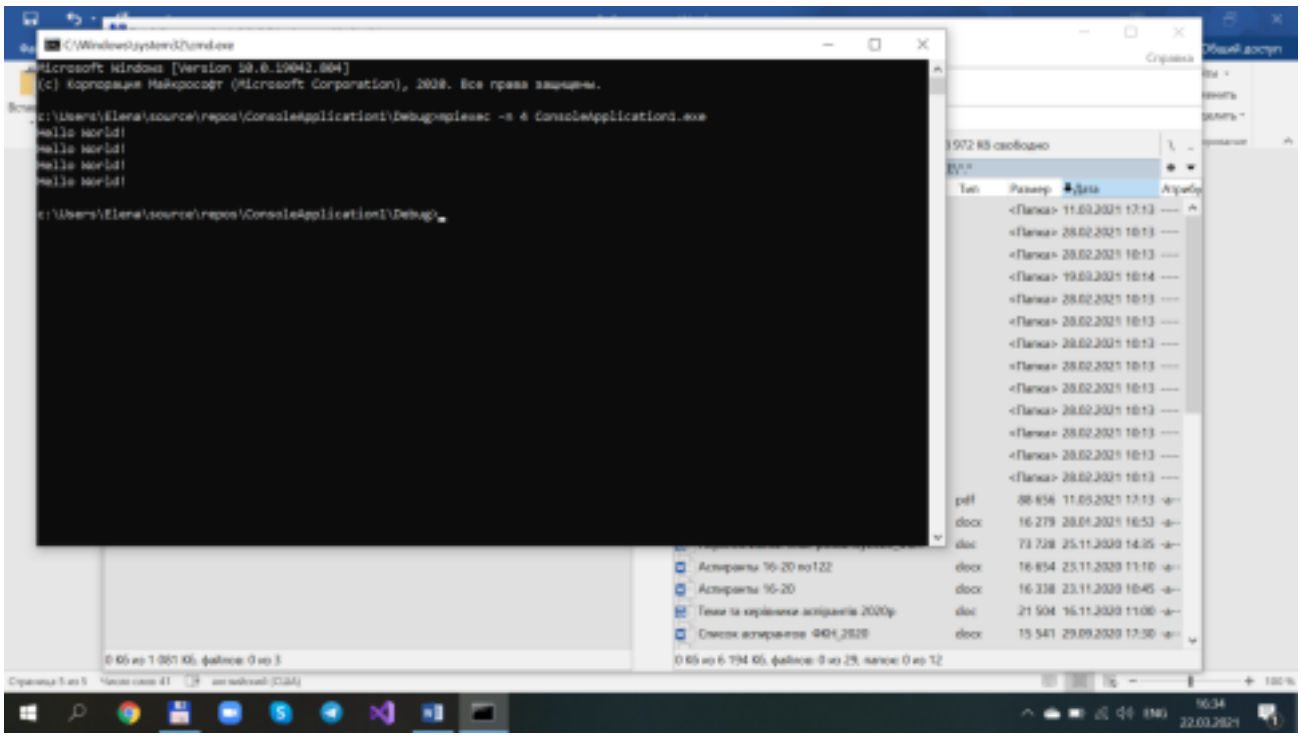

Перенацелить решение
Добавить модель...
Добавить класс...
Мастер классов... Ctrl+Shift+X
Добавить новый элемент... Ctrl+Shift+A
Добавить существующий элемент... Shift+Alt+A
Исключить из проекта
Показать все файлы
Повторить сканирование решения
Показать ошибки просмотра в базе данных
Скрыть ошибки просмотра в базе данных
Добавить подключенную службу
Назначить в качестве запускаемого проекта
Экспорт шаблона...
Управление пакетами NuGet...
Свойства: ConsoleApplication1










C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19042.804]
(c) Корпорация Майкрософт (Microsoft Corporation), 2020. Все права защищены.
c:\Users\Elena\source\repos\ConsoleApplication1\Debug>mpiexec -n 4 ConsoleApplication1.exe
Hello World!
Hello World!
Hello World!
Hello World!
c:\Users\Elena\source\repos\ConsoleApplication1\Debug>

Додаток D. Орієнтовні питання до контрольної роботи

Варіант 1

1. Схарактеризуйте два основні підходи до досягнення паралельності.
2. Назвіть основні характеристики архітектури фон Неймана. Які основні відмінності архітектури фон Неймана від гарвардської архітектури.
3. Розробіть модель і виконайте оцінку показників прискорення й ефективності паралельних обчислень для задачі скалярного добутку двох векторів.

Варіант 2

1. Схарактеризуйте найпростішу модель паралельного програмування.
2. На яких ознаках заснована класифікація Фліна? Наведіть фрагмент класифікації Фліна.
3. Розробіть модель і виконайте оцінку показників прискорення й ефективності паралельних обчислень для задачі пошуку максимального значення для заданого набору числових даних.

Варіант 3

1. Опишіть переваги розподіленого програмування. У яких процесорах використовується конвеєрна обробка даних?
2. Основне призначення й різновиди комутаторів у паралельній обчислювальній системі.
3. Розробіть модель і виконайте оцінку показників прискорення й ефективності паралельних обчислень для задачі пошуку мінімального значення для заданого набору числових даних.

Варіант 4

1. Які найпростіші моделі розподіленого програмування Ви знаєте?
2. Як розрізняються обчислювальні системи по ступені зв’язності?
3. Розробіть модель і виконайте оцінку показників прискорення й ефективності паралельних обчислень для задачі знаходження середнього значення для заданого набору числових даних.

Варіант 5

1. Дайте визначення поняття «топологія системи». Які види топології системи Ви знаєте?
2. Наведіть основні моделі розподіленого і паралельного програмування.
3. Розробіть модель і виконайте оцінку показників прискорення й ефективності паралельних обчислень для задачі множення матриці на вектор.

Електронне навчальне видання комбінованого використання
Можна використовувати в локальному та мережному режимі

Толстолузька Олена Геннадіївна
Лабенко Дмитро Петрович
Бакуменко Ніна Станіславівна

ТЕХНОЛОГІЇ РОЗПОДІЛЕНИХ СИСТЕМ ТА ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Методичні вказівки
з виконання лабораторних робіт

В авторській редакції

Підписано до розміщення 18.06.2024. Гарнітура Times New Roman.
Ум. друк. арк. 4,57. Обсяг. 2,407. Мб. Зам. № 193/24.

Харківський національний університет імені В. Н. Каразіна,
61022, м. Харків, майдан Свободи, 4.
Свідоцтво суб'єкта видавничої справи ДК № 3367 від 13.01.2009
Видавництво ХНУ імені В. Н. Каразіна