МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання та оформлення розрахунково-графічних завдань
з навчальної дисципліни
«Паралельні та розподілені обчислення»

для студентів денної та заочної форм навчання за спеціальностями
123 «Комп’ютерна інженерія»
125 «Кібербезпека»

Харків
НТУ «ХПІ»
2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання та оформлення розрахунково-графічних завдань
з навчальної дисципліни
«Паралельні та розподілені обчислення»

для студентів денної та заочної форм навчання за спеціальностями
123 «Комп’ютерна інженерія»
125 «Кібербезпека»

Затверджено редакційно-видавничою радою університету протокол №2 від 25.06.2020 р.

Харків
НТУ «ХПІ»
2021

Методичні вказівки до виконання та оформлення розрахунково-графічних завдань з навчальної дисципліни «Паралельні та розподілені обчислення» для студентів денної та заочної форм навчання за спеціальностями 123 «Комп'ютерна інженерія» і 125 «Кібербезпека» / уклад., О. П. Черних, В. І. Калашніков, С. С. Бульба, О.А. Горносталь. – Харків: НТУ "ХПІ", 2021. – 63 с.

Укладачі: О.П. Черних, В.І. Калашніков, С.С. Бульба, О.А. Горносталь

Рецензент Г.А. Кучук

Кафедра обчислювальної техніки та програмування

Викладена методика розрахована на застосування комп'ютерного моделювання, за допомогою якого знайомлять студентів з методологією побудови паралельних алгоритмів, конфігуруванням обчислювального середовища, інтерпретацією і дослідженням результатів роботи паралельної програми на безлічі вихідних даних, виконання та оформлення розрахунково-графічних завдань.

Методичні вказівки підготовлені на кафедрі «Обчислювальна техніка та програмування» і можуть бути використані для підготовки дипломованих фахівців за спеціальностями 123 «Комп'ютерна інженерія» і 125 «Кібербезпека».

Черних О.П., Калашніков В.І., Бульба С.С., Горносталь О.А.©

НТУ «ХПІ», 2021®

ЗМІСТ

ВСТУП.. 5
1 ІНДИВІДУАЛЬНЕ ЗАВДАННЯ 6
1.1 Формулювання завдання 6
1.2 Розробка опису і тексту програми 7
1.3 Вимоги до структури вершинного програмного коду 8
1.4 Рекомендації щодо оформлення завдання 9
СПИСОК ЛІТЕРАТУРИ 9
2 ТЕОРЕТИЧНІ ВІДОМОСТІ 10
2.1 Графова модель паралельних процесів 10
2.2 Опис графів інформаційної залежності 10
2.3 Властивості матриці суміжності орграфа 13
2.4 Лексикографічне впорядкування вершин по ярусах 13
2.5 Облік часу протікання верхових процесів 17
2.6 Пошук границь припустимих переміщень процесів 19
2.8 Організація взаємного обміну даними 24
2.9 Прискорення при паралельних обчисленнях 26
Додаток А 33
3 РОЗРОБКА ОПИСУ І ТЕКСТУ ПРОГРАМИ 50
3.1 Розробка алгоритму функціонування програми 50
3.2 Опис програми 50
3.3 Текст програми 51
3.4 Файл вхідних даних 59
3.5 Результати роботи програми 59
3 ВИСНОВКИ 62

ВСТУП

З метою досягнення максимальної швидкості розв'язування громіздких обчислювальних задач у багатомашинному обчислювальному середовищі було розроблено комплекс підготовчих дій, що одержав назву «паралельні та розподілені обчислення».

Основна мета інженерів і програмістів полягає в тому, щоб зменшити час обчислень або час обробки даних, підвищити ефективність завантаження обчислювального устаткування і скоротити накладні витрати на синхронізацію процесів і обмін даними. Очевидний шлях прискорення будь-якого виду взаємопов'язаних робіт в рамках однієї складної задачі полягає в розподілі окремих частин цієї роботи по декількох взаємодіючим паралельним потокам обробки.

Необхідно розуміти і враховувати множину нюансів, що виникають у паралельних обчислювальних системах при виконанні операцій обміну даними, як усередині локального процесора, так і між різними процесорами. Дуже важливими стають питання побудови паралельного алгоритму, який би виконувався за мінімально можливий час, використовував при цьому найменше число процесорів і не мав би тупикових зупинок.

Однією з основних систем паралельного програмування у теперішній час вважають MPI (система передачі повідомлень, яка реалізує стандарт інтерфейсу передачі повідомлень) з основними мовами програмування С і С++.

У даних методичних вказівках послідовно розглядаються усі названі питання та даються відповідні рекомендації з їхнього обліку і дозволу. По ходу викладення наводяться приклади.

1 ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

1.1 Формулювання завдання

Зважений граф інформаційної залежності щодо побудування паралельного алгоритму обчислювальної задачі задається базовою матрицею інциденцій і базовою таблицею ваг вершин:

Базова матриця інциденцій вершин:

2	3	2	2	3	7	5	5	8	1	6	– вершина початку ребра
6	6	7	5	1	4	8	1	4	4	4	– вершина кінця ребра

Таблиця ваг вершин:

2	3	7	5	6	8	1	4	– номер вершини
3	4	1	3	5	4	3	2	– вага вершини

Для отримання кожним студентом свого варіанту матриці інциденцій та таблиці ваг вершин необхідно знати свій номер N у журналі групи, або номер, заданий викладачем. Усі номери вершин у матриці інциденцій перетворити на інші номери по формулі:

$$z' = (z + N)\mathrm{mod}(8) + 1, \tag{1.1}$$

де z', z – відповідно новий і старий номери вершин, наприклад:

"(z+N)"	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
"(z+N)mod(8)+1"	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6

У таблиці ваг вершин базові значення ваг перетворити у ваги свого варіанту за аналогічною формулою, але суму в неї виконувати по модулю 5:

$$t' = (t + N)\mathrm{mod}(5) + 1, \tag{1.2}$$

де $t', \ t -$ відповідно нова і стара вага вершини, наприклад:

"(t+N)"	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
"(t+N)mod(5)+1"	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2

Виконати формальний аналіз інформаційної незалежності операторів:

1) Виконати лексикографічне упорядкування наданого графа.

2) Обчислити час раннього терміну закінчення виконання операторів.

3) Знайти критичний час виконання алгоритму $T_{кр}$ за означеним графом.

4) Приймаючи час виконання усього набору операторів найменшим за $T \le T_{кр} + 5$ одиниць, обчислити час пізнього терміну закінчення виконання операторів.

5) Відзначити на часових діаграмах інтервали можливого запуску і часу виконання кожного з операторів.

6) Побудувати графіки щільності навантаження процесорів для моменту раннього і пізнього термінів.

7) Побудувати діаграму розміщення інтервалів допустимого закінчення виконання операторів по паралельних гілках, розташованих у двох процесорах.

8) Обчислити степінь паралелізму, прискорення процесу, завантаження обладнання.

1.2 Розробка опису і тексту програми

Отримавши діаграму розпаралелення з аналітичної частини, описати конфігурацію її виконання бібліотечними процедурами розширеного інтерфейсу передачі повідомлень MPI, застосовуючи мову C++.

Створити логічну структуру і схему паралельного алгоритму. Сформувати групи процесорних об’єктів щодо об’яви і означення у тексті програми інтра- і інтеркомунікаторів, які забезпечать взаємодію між процесорними об’єктами у межах кожного процесора і між паралельно функціонуючими процесорами. Назвати прості чи створені типи даних щодо взаємодії процесів.

У тексті програми використати необхідні заголовочні файли, оператори об’яви та опису ідентифікаторів змінних і особисто ідентифікаторів груп і комунікаторів, процедур процесів, що виконують взаємодію процесів і імітують їх час роботи. У програмі використовувати підпрограми MPI, що задають виконання паралельної програми і взаємодії її процесів у середовищі NT-MPICH, а також оператори звичайної мови програмування С++, які б приймали та формували щодо передачі деякі умовні дані між паралельними процесорними об’єктами. Вставити у тексти процесорних об’єктів оператори контролю часу запуску та завершення виконання процесів, значення яких разом із умовними даними, що передавались, відобразити у файлі друку.

Приклади запису фрагментів програм, застосування підпрограм MPI розташовані у [**1**: гл. 3,5,6], а також у методичних матеріалах, що надаються викладачем.

1.3 Вимоги до структури вершинного програмного коду

Кожен програмний код, що пов'язаний з кожною вершиною графа інформаційної залежності (ГІЗ), повинен передбачати, як мінімум, слідуючи фрагменти:

1) підпрограми MPI, що забезпечують прийом і фіксацію даних від попередніх процесів згідно ГІЗ;

2) фіксацію системного часу, що відповідає прийому всіх даних, що визначають запуск на виконання даного процесорного об'єкта;

3) зібрати в зручну для запису форму прийняті дані разом з рангами процесів, що послали, час їхнього прийому й записати все у файл друку;

4) імітувати заданий відносний час виконання вершинного процесу, у рамках якого (або незалежно від цього часу) підготувати дані для пересилання згідно ГІЗ іншим вершинним процесам;

5) зібрати в зручну для запису форму підготовлені дані разом з рангами процесів-одержувачів і час відправлення, які записати у файл друку;

6) підпрограмами MPI розіслати підготовлені дані відповідно до ГІЗ наступним процесам.

Програмний код кінцевого вершинного процесу, крім перерахованого вище, повинен забезпечити роздруківку загального файлу друку в зручну щодо аналізу форму.

1.4 Рекомендації щодо оформлення завдання

Розрахунково-графічні завдання оформлювати на аркушах формату А4 з титульним листом заданої форми (див. нижче). Оформлення по ЕСПД. На одному листі 40-43 стрічки тексту з розміром шрифту Times New Roman 14. Зразок титульного листа показано у Додатку А.

Хід рішення повинен супроводжуватись поясненнями та перевірками.

Виконані завдання скріпляються з титульним листом і передаються на перевірку викладачу не пізніше ніж за тиждень до залікового тижню.

Залічені роботи служать допуском до складання іспиту за курсом «Паралельні та розподіленні обчислення».

Оформлення текстових документів, к яким відносяться і розрахунково-графічні роботи студентів, виконуються за державними стандартами України.

СПИСОК ЛІТЕРАТУРИ

1. Калашников В.И. Параллельное программирование: Уч. Пособие. – Харьков: НТУ “ХПИ”, 2005. – 320 с.
2. ДСТУ 3008-95. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення. “Київ”, Держстандарт України, 1995.
3. ГОСТ 2.105-95. ЕСКД. Загальні вимоги до текстових документів. “Київ”, Держстандарт України, 1995.

2 ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1 Графова модель паралельних процесів

Для побудови абстрактної моделі паралельного процесу у вигляді графа досить у якості множини вершин графа обрати множину елементів (зерен) паралельних форм, які являють собою окремі оператори, функції, процедури або інші відособлені програмні коди, що мають крапку входу (початок деякого процесу) і крапку виходу (завершення цього процесу). Абстрактне поняття вершини сполучає в одну крапку крапки входу і виходу та має оригінальне ім'я, що дозволяє відрізняти одну вершину множини від іншої. За вершиною може бути закріплена, наприклад, деяка числова характеристика.

Абстрактне поняття орієнтованого по напрямку ребра або дуги однозначно може бути закріплене за напрямком передачі вироблених даних (інформації) від однієї вершини до іншої. Число вхідних і вихідних з вершини ребер є важливою характеристикою вершини.

Характер взаємозв'язку абстрактних вершин і абстрактних орієнтованих ребер є відображенням однієї множини на іншу, семантикою якої для паралельного програмування може служити яка-небудь паралельна форма алгоритму розв'язку задачі.

Стосовно до задач паралельного програмування описану абстрактну модель паралельного обчислювального процесу називають *графом інформаційної залежності* (ГІЗ).

2.2 Опис графів інформаційної залежності

Для візуального зображення орієнтованого графа (орграфа) G можна використовувати в якості вершин кружечки, у якості спрямованих ребер – стрілки, а відображення множини дуг на множину вершин – з'єднанням конкретних вершин відповідною дугою.

Аналітичний опис графа G включає непусту множину вершин $\mathbf{V}=\{v_i | i\in N\}$, множину дуг $\mathbf{A}=\{a_k | k\in N\}$, що не перетинається з $\mathbf{V}$, і відображення Δ множини $\mathbf{A}$ на $\mathbf{V}\times\mathbf{V}$. Δ називається орієнтованим відображенням інцидентних вершин орграфа. Якщо $a\in\mathbf{A}$, то $\Delta(a)=(v,w)$

означає, що дуга a має початкову вершину v та кінцеву вершину w. Таким чином, орграф вводиться трійкою параметрів і позначається як $\mathbf{G}(\mathbf{V},\mathbf{A},\Delta)$.

Для визначення множини вершин орграфа за паралельною формою алгоритму досить включити в цю множину імена всіх зерен форми. Аналогічно в множину орієнтованих дуг потрібно включити імена всіх стрілок, які закріплені за орієнтованими дугами. Формальне відображення множини дуг на множину вершин можна задати декількома способами. Наприклад, множиною з елементами у вигляді інцидентних пар вершин, зв'язаних однією дугою, або множиною з елементами у вигляді інцидентних пар вершина-дуга. Використання інцидентних пар як аналітичне визначення орграфа для наступної машинної обробки найбільше популярно у вигляді матриці суміжності, матриці інцидентностей або розгорнутої трійки позначеного орграфа. Вони є впорядкованими множинами відносин.

Матриця суміжності $S=[s_{ij}]$ позначеного орграфа $\mathbf{G}$ з p вершинами є ($p\times p$) – матриця, у якій

$$s_{ij}=1\text{, якщо дуга } (v_i,v_j)\in\mathbf{A}\text{,}$$
$$s_{ij}=0\text{, якщо дуга } (v_i,v_j)\notin\mathbf{A}\text{,}$$

де i – перераховує вершини, з яких спрямована дуга виходить, а
j – перераховує вершини, у які дуга входить, наприклад:

$$S=\begin{array}{c} \begin{array}{ccccc} v_1 & \dots & v_j & \dots & v_p \end{array} \\ \left[\begin{array}{c:c:c:c:c} 0 & 1 & 1 & 0 & 1 \\ \hdashline 0 & 0 & 1 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 1 & 1 \\ \hdashline 0 & 0 & 0 & 0 & 1 \\ \hdashline 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array} \begin{array}{c} v_1 \\ \cdots \\ v_i \\ \cdots \\ v_p \end{array} \qquad (2.1)$$

Матриця інцидентностей $B=[b_{ij}]$ позначеного орграфа з p вершинами та q ребрами є $(p\times q)$ – матриця з елементами $b_{ij}\in\{0,1,-1\}$.

$$b_{ij} = \begin{cases} +1, \text{ коли дуга } a_j, \text{інцидентна вершині } v_i, \text{ виходить із вершини}, \\ 0, \text{ коли вершина } v_i \text{ не інцидентна дузі}, \\ -1, \text{ коли дуга } a_j, \text{інцидентна вершині } v_i, \text{ входить у вершину}. \end{cases}$$

Дуги $a_j \in \mathbf{A}$ перелічуються в матриці по стовпцях, а вершини v_i перелічуються по рядках, наприклад:

$$B = \begin{array}{c} \begin{matrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \end{matrix} \\ \left[\begin{array}{c|c|c|c|c|c|c} +1 & +1 & 1 & 0 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & +1 & 0 & 0 & 0 \\ \hline 0 & -1 & 0 & -1 & +1 & +1 & 0 \\ \hline 0 & 0 & 0 & 0 & -1 & 0 & +1 \\ \hline 0 & 0 & -1 & 0 & 0 & -1 & -1 \end{array}\right] \end{array} \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{matrix} \qquad (2.2)$$

Граф $\mathbf{G}(\mathbf{V}, \mathbf{A}, \Delta)$, заданий трьома множинами, наприклад, множиною інцидентних вершин $\mathbf{V} = \{v_i \mid i = \overline{1,5}\}$, множиною ребер $\mathbf{A} = \{a_k \mid k = \overline{1,7}\}$ і множиною відображень орієнтованих дуг на інцидентні вершини.

$$\mathbf{G} = \left\{ \begin{array}{c} \{a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6 \quad a_7\}, \\ \downarrow \left\{ \begin{array}{c|c|c|c|c|c|c} v_1 & v_1 & v_1 & v_2 & v_3 & v_3 & v_4 \\ v_2 & v_3 & v_5 & v_3 & v_4 & v_5 & v_5 \end{array} \right\} \end{array} \right\}. \qquad (2.3)$$

Число ребер, інцидентних вершині v, називають ***ступенем вершини*** v та позначають як $\delta(v)$. Для ізольованої вершини v ступінь вершини $\delta(v) = 0$, а для вершини орграфа $\delta(v) = \delta^+(v) + \delta^-(v)$, де доданки з верхніми індексами позначають відповідно позитивну та негативну ступені вершини v, тобто число дуг вихідних з вершини і число вхідних відповідно. Загальне число дуг $\delta_{\mathbf{A}}$ в орграфі дорівнює

$$\sum_{v \in \mathbf{V}} \delta^+(v) = \sum_{v \in \mathbf{V}} \delta^-(v) = \delta_{\mathbf{A}}. \qquad (2.4)$$

Відповідно розглянутим паралельним формам усі оператори розміщені так, що послідовність їх включення додержується строгого порядку, тобто, дивлячись на неї, можемо впевнено сказати, який з операторів повинен увімкнутися на виконання раніше іншого, що лежить на одному шляху. Отже, і в упорядкованій послідовності вершин орграфа, що представляє паралельну форму, початкова вершина будь-якого шляху повинна перебувати раніше кінцевої. Тому первісною задачею теоретичного аналізу та перетворення орграфа є така його розмітка, яка забезпечить цю строгу впорядкованість.

Шлях від початкової вершини орграфа до кінцевої вершини орграфа, що містить найменше число дуг, називають ***критичним шляхом***.

2.3 Властивості матриці суміжності орграфа

Початковими вершинами є ті вершини, стовпці матриці суміжності якої є нульовими.

Кінцевими вершинами є ті вершини, рядки матриці суміжності якої є нульовими.

Сума елементів в i-му рядку матриці сміжностей дорівнює числу вихідних дуг з вершини v_i, тобто позитивний ступінь вершини рівний $\delta^{+}(v)$ (підлоги ступінь результату), а сума елементів в i-тому стовпці матриці сміжностей дорівнює числу вхідних у v_i вершину дуг, тобто негативному ступеню вершини $\delta^{-}(v)$ (підлоги ступінь заходу).

У результаті піднесення матриці суміжності $\mathbf{S}$ в n-*ту* ступінь із логічними операціями (i, j)-й елемент матриці $\mathbf{S}^n$ буде дорівнювати:

$$s_{ij}^{(n)} = \begin{cases} 1, \text{коли від вершини } i \text{ до вершини } j \text{ є шлях з } n \text{ дугами,} \\ 0, \text{коли такого шляху немає.} \end{cases}$$

2.4 Лексикографічне впорядкування вершин по ярусах

У довільно складеній матриці суміжності орграфа множині імен вершин можна додати транзитивну властивість. Для цього досить помістити кожне ім'я вершини в інформаційну частину елемента індексного списку. У цьому випадку до імені вершини можна звертатися по індексу,

представленому числом натурального ряду. Упорядкований по індексу списку вершин стане множиною нових імен вершин орграфа. Однак для паралельних форм алгоритмів важливо, щоб транзитивна властивість зерен-вершин, яка обумовлена зростанням їх індексу, була погоджена з напрямком переходу від вершини до вершини по дугах. Це дозволить вибудувати всю множину вершин орграфа в такій послідовності, коли всі зерна-вершини паралельної форми алгоритму будуть передавати підготовлені дані тільки вершинам з більшим індексом. Натуральні числа індексів у першому наближенні будуть характеризувати тимчасову послідовність включення чергового оператора, закріпленого за іменем вершини. Саме цю задачу ми вирішували евристично, створюючи ярусні паралельні форми для регулярних фрагментів. Тепер необхідно побудувати ярусну форму по орграфу задачі з нерегулярною структурою взаємодіючих один з одним обчислювальних фрагментів. Для досягнення цієї мети треба провести сортування (переіндексацію) імен вершин у списку так, щоб їх упорядковане розташування відповідало напрямкам дуг.

Зв'яжемо з кожною вершиною v_i в списку дві підмножини вершин (при необхідності у вигляді двох списків). Одна підмножина з потужністю, рівною негативному напівступеню $\delta^-(v_i)$, включає імена вершин, від яких дуги приходять до v_i, і друга підмножина з потужністю, рівною позитивному напівступеню $\delta^+(v_i)$, включає імена вершин, на які дуги спрямовані від v_i .

Як приклад такого списку, може служити наведена нижче табл. 2.1 з декількома стовпцями, складена по графу інформаційної залежності, наведеному на рис. 2.1а.

Для лексикографічного впорядкування орграфа зручно використовувати стовпчик $\delta^-(v_i)$ табл. 2.1. У таблиці введено два додаткових стовпця: один призначений для закріплення за вершиною v_i порядкового номера ярусу *j*, а другий – для запису нового індексу $k \in N$, що впорядковує.

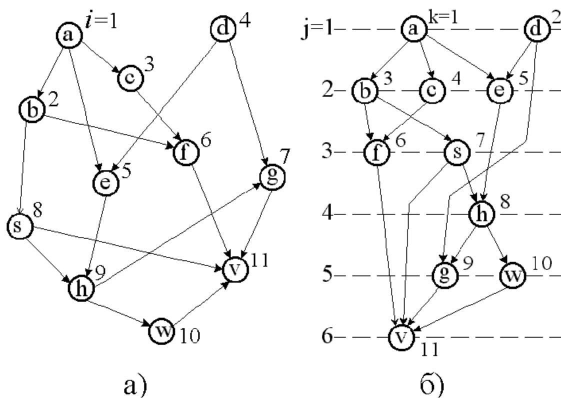

Рисунок 2.1 – Граф процесів

Таблиця 2.1 – Список вершин із множинами дуг (вихідних/вхідних)

Індекс імені i	Ім'я вершини	$\delta^{+}(v_i)$	$\delta^{-}(v_i)$	Ярус j	Новий індекс k
T(1)	T(2)	T(3)	T(4)	T(5)	T(6)
1	a	{b, e, c} = {2, 3, 5}	{∅}	1	1
2	b	{f, s} = {6, 8}	{a} = {1}	2	3
3	c	{f} = {6}	{a} = {1}	2	4
4	d	{e, g} = {5, 7}	{∅}	1	2
5	e	{h} = {9}	{a, d} = {1, 4}	2	5
6	f	{v} = {11}	{b, c} = {2, 3}	3	6
7	g	{v} = {11}	{d, h} = {4, 9}	5	9
8	s	{h, v} = {9, 11}	{b} = {2}	3	7
9	h	{g, w} = {7, 10}	{e, s} = {5,8}	4	8
10	w	{v} = {11}	{h} = {9}	5	10
11	v	{∅}	{f, g, s, w} = {6, 7, 8, 10}	6	11

Матриця суміжності для графа представленого рис. 2.1, до і після лексикографічного впорядкування наведена на рис. 2.2.

$S =$

	a	b	c	d	e	f	g	s	h	w	v
a	0	1	1	0	1	0	0	0	0	0	0
b	0	0	0	0	0	1	0	1	0	0	0
c	0	0	0	0	0	1	0	0	0	0	0
d	0	0	0	0	1	0	1	0	0	0	0
e	0	0	0	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0	0	0	1
g	0	0	0	0	0	0	0	0	0	0	1
s	0	0	0	0	0	0	0	0	1	0	1
h	0	0	0	0	0	0	1	0	0	1	0
w	0	0	0	0	0	0	0	0	0	0	1
v	0	0	0	0	0	0	0	0	0	0	0

а)

$S^{y} =$

	a	d	b	c	e	f	s	h	g	w	v
a	0	0	1	1	1	0	0	0	0	0	0
d	0	0	0	0	1	0	0	0	1	0	0
b	0	0	0	0	0	1	1	0	0	0	0
c	0	0	0	0	0	1	0	0	0	0	0
e	0	0	0	0	0	0	0	1	0	0	0
f	0	0	0	0	0	0	0	0	0	0	1
s	0	0	0	0	0	0	0	1	0	0	1
h	0	0	0	0	0	0	0	0	1	1	0
g	0	0	0	0	0	0	0	0	0	0	1
w	0	0	0	0	0	0	0	0	0	0	1
v	0	0	0	0	0	0	0	0	0	0	0

б)

Рисунок 2.2 – Матриця суміжності

Зміст лексикографічного впорядкування зручно описати, використовуючи візуальне або табличне подання орграфа в такий спосіб.

По іменах {*a, b, c,...*} або по їх первісних індексах {1, 2, 3,...} у списку (стовпчика Т(2), Т(1) у табл. 2.1) послідовно проглядаються негативні напівступені вершин $\{\delta^{-}(i)\}$ (стовпчик Т(4)). Вершини, що задовольняють умовам $K(i) = 0$ (новий індекс імені ще не привласнений) і $\{\delta^{-}(i)\} = \varnothing$ (немає вхідних дуг), містяться в робочу множину *M* с присвоєнням їм того самого номера ярусу ($j = 1, 2, \ldots$) і чергового індексу вершини ($k = 1, 2, \ldots, p$). Наприкінці чергового циклу індексна змінна $i = 0, 1, \ldots, p$ приймає значення $i = 0$, яке запускає процес виключення із усіх множин $\{\delta^{-}(i)\}$ тих вершин, які були занесені в робочу множину (*M*). При цьому виключенні з'являться нові вершини, у яких відсутні вхідні дуги. Вершини, яким раніше вже був привласнений новий індекс і для них $K(i) \neq 0$, у черговому циклі пропускаються, а вершини з негативним напівступенем, рівним нулю, розміщуються в робочу множину вершин нового ярусу. Цикли перегляду продовжують доти, поки всім вершинам не привласнять нові індекси.

Розташувавши вершини по своїх ярусах по зростанню нового індексу ліворуч праворуч і зверху вниз, одержимо граф, показаний на рис. 2.1б, і його верхньотрикутну впорядковану матрицю суміжності (рис. 2.2б).

Рисунок 2.1б для розглянутої задачі з 11 взаємодіючими блоками дає ідеальну картину розміщення паралельних процесів у часі. Тут передбачається, що між'ярусний часовий інтервал обраний таким чином, щоб усі верхові процеси одного ярусу до обміну даними були закінчені. Паралелізм на тому або іншому ярусі в деяких випадках може бути змінений. Наприклад, вершину 5 (e) на рис. 2.1б можна перемістити із другого ярусу на третій, а вершину 6 (f) – з третього на четвертий. Цими переміщеннями ми можемо добитися того, що на всіх ярусах ступінь паралелізму не перевищить два.

2.5 Облік часу протікання верхових процесів

Блоки задачі, що сформовані для їхнього паралельного виконання, повинні представляти по можливості закінчені алгоритми, які використовують для свого перетворення тільки апріорно задані значення та значення, передані в момент запуску блоку. Якщо алгоритми блоків представлені готовими текстами програми, то в процесі компіляції і тестування їх контрольними прикладами можна в зручних одиницях оцінити дійсний час протікання обчислювального процесу кожного виділеного блоку. Бажано, щоб похибка оцінок часу становила не більш 10%.

Тимчасові оцінки тривалості верхових процесів не повинні перевищувати по тривалості міжярусний проміжок. Адже кожний ярус після впорядкування графа визначає момент запуску (по готовності даних) процесів у вершинах, розташованих на більш пізніх ярусах далі, і збільшення тривалості верхового процесу може привести до переорієнтації деяких дуг графа й у першу чергу тієї, яка виходить від вершини, що затягла процес. Тому умовою вибору тривалості інтервалу між ярусами (і не обов'язково однакової з усіма) повинне бути перевищення цієї тривалості над тривалістю найбільш довгого процесу на ярусі:

$$\Delta t_j = t_{j+1} - t_j \geq \max_{k \in [j, j+1]} [t_{j,k}], \qquad (2.5)$$

де $t_{j,k}$, $k \in [j, j+1]$ – час виконання k-*того* процесу на j-*тому* ярусі.

Як приклад на рис. 2.3 зображений упорядкований по ярусах граф з рис. 2.1, у якому на місці вершин зображені відрізки з довжинами, пропорційними тривалості процесу у вершині.

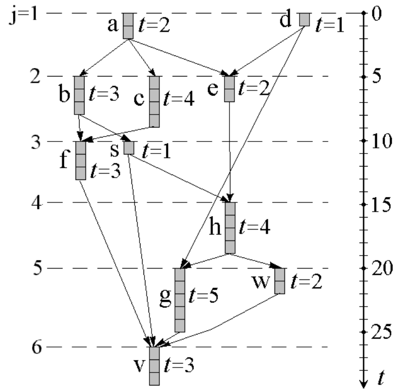

Рисунок 2.3 – Упорядкований граф

Праворуч показана шкала відносного часу, до якої прив'язані яруси. Вона дає подання про такі переміщення ярусів, які не змінили б на зворотний напрямок проекцій дуг на тимчасову вісь.

Однак такий очевидний розподіл у часі із синхронним запуском верхових процесів, що перебувають на одному ярусі, не вирішує, принаймні, двох суперечливих проблем: мінімізації числа процесорів, необхідного для розв'язку задачі за мінімальний час, і мінімізації часу розв'язку задачі на заданому числі процесорів. Перша проблема – це необхідність переміщення вершин з ярусу на ярус вверх та низ так, щоб, не міняючи напрямку проекцій дуг на тимчасову вісь, максимально зблизити послідовно виконувані процеси, після чого вибрати число процесорів по максимальному ступеню паралелізму отриманого графа інформаційної залежності. Друга проблема пов'язана із примусовим контрольованим переміщенням окремих процесів по осі часу (по ярусах) так, щоб ступінь паралелізму не перевищила числа заданої кількості процесорів.

Щоб розв'язати названі проблеми, необхідно ввести аналітичні залежності, що погоджують воєдино результати обробки, що впорядковує, графа інформаційної залежності, часи виконання верхових процесів, ступінь паралелізму і загальний час розв'язку задачі.

Перше, що необхідно зробити, знайти границі припустимого переміщення кожного верхового процесу по осі часу, тобто, маючи часи

виконання кожного елементарного процесу, потрібно знайти для них відносно початку моменти самого раннього та самого пізнього закінчення, які виражені в деяких умовних одиницях часу.

2.6 Пошук границь припустимих переміщень процесів

Для зручності у табл. 2.2 вносимо відсортовані по нових індексах вершини, стовпець ваги вершин t у відносних одиницях часу, стовпець для занесення ранніх строків закінчення процесів і стовпець для занесення пізніх строків закінчення верхових процесів.

Ранні строки закінчення кожного верхового процесу будуть заноситися в стовпчик табл. 2.2, представленої в алгоритмі попередньо обнульованим одномірним масивом $\tau_1[1..p]$. Індекс j перераховує стовпці впорядкованої матриці суміжності $S_{i,j}$ графа інформаційної залежності (ГІЗ), а в таблиці 2 (стовпчик негативних напівступенів $\delta^-(v_i)$) – указує на підмножину вершин, що перебувають на i -их рядках j -го стовпця матриці $S_{i,j}$. Значення елемента $S_{i,j}$ дорівнює або 1, або 0. Алгоритм працює доти, поки на всіх позиціях стовпчика ранніх строків закінчення верхових процесів не зникнуть усі попередньо записані нульові значення. Обчислення конкретного значення раннього строку закінчення j -го верхового процесу здійснюється по формулі:

$$\tau_1(j) := \max_{\mu \in \{\overline{1,p} \mid S_{\mu,j}=1 \ \& \ \tau_{1\mu} \neq 0\}} [\tau_{1\mu}] + t_j \,, \tag{2.6}$$

де $\tau_{1\mu}$ – ранні строки закінчення процесів μ -тих вершин, для яких одночасно виконана умова, що на μ -тому рядку j -го стовпця матриці суміжності S перебуває одиниця і для μ -того рядка стовпчика τ_1 табл. 2.2 ранній строк закінчення вже знайдений. До найбільшого з $\tau_{1\mu}$ -тих додається час виконання j -го процесу.

Значення ранніх строків закінчення процесів визначають мінімально можливий час готовності даних для продовження обчислень суміжними процесами.

Таблиця 2.2 – Ранні і пізні строки закінчення виконання операторів

Індекс k	Вихідне ім'я	$\delta^{+}(v_i)$	$\delta^{-}(v_i)$	Вага t	Ярус j	$\tau_1(k)$	$\tau_2^T(k)$
1	a	{b,з,e}	{∅}	2	1	2	2
2	d	{e,g}	{∅}	1	1	1	4
3	b	{f,s}	{a}	3	2	5	5
4	c	{f}	{a}	4	2	6	12
5	e	{h}	{a,d}	2	2	4	6
6	f	{v}	{b,c}	3	3	9	15
7	s	{h,v}	{b}	1	3	6	6
8	h	{g, w}	{e,s}	4	4	10	10
9	g	{v}	{d,h}	5	5	15	15
10	w	{v}	{h}	2	5	12	15
11	v	{∅}	{f,g,s,w}	3	6	18	18

Граничне значення пізніх строків пов'язане з послідовним розташуванням у часі всіх верхових процесів один за одним і застосуванням для обчислень одного процесора. Це уведе нас від проблеми розпаралелення.

Отже, задачу пошуку пізніх строків необхідно вирішувати, дотримуючись умови реалізації всього обчислювального процесу за мінімально можливий час, не обертаючи поки уваги на реалізованість максимального ступеня паралелізму.

Ранній строк закінчення самого останнього верхового процесу на самому нижньому ярусі визначає мінімальний час виконання всього паралельного процесу. Цей час називають критичним часом $T = T_{\text{кр}}$, тому що він складається із суми часів виконання верхових процесів, прив'язаних до вершин лежачих на самому довгому за часом критичному шляху зваженого графа інформаційної залежності. При усьому бажанні виконати паралельний алгоритм за менший час це принципово неможливо.

У межах критичного часу можна тепер знайти й найбільш пізні строки закінчення верхових процесів шляхом підтягування їх якнайближче до $T = T_{\text{кр}}$, не порушуючи спрямованості проекцій дуг критичних шляхів графа на тимчасову вісь. Відмінність полягає в організації перегляду вершин від

самої пізньої по виконанню до самої ранньої в ГІЗ і використання іншої формули обчислення пізнього строку закінчення верхового процесу:

$$\tau_2^T(i) = \min_{\mu \in \{\overline{1,p} \mid S_{i,\mu}=1 \ \& \ \tau_{2\mu}^T \neq 0\}} [\tau_{2\mu}^T - t_\mu], \qquad (2.7)$$

де $\tau_{2\mu}^T$ – пізні строки закінчення процесів μ-тих вершин для заданого загального часу обчислень $T \geq T_{кр}$. Мінімум вибирається з μ-тих різниць, для яких одночасно виконана умова, що на μ-тому стовпці i-го рядка матриці суміжності S перебуває одиниця та для μ-того рядка стовпчика τ_2^T табл. 2.2 пізній строк закінчення вже знайдений.

Перегляд рядків у матриці суміжності та табл. 2.2 починається з останнього рядка $i = p$. Якщо для вершини, яка лежить на i-тому рядку, пізній строк закінчення ще не обчислювався, тобто $\tau_2^T(i) = 0$, то перевіряється наявність вихідних дуг: чи порожній позитивний напівступінь вершини $\delta^+(i) \in \{\varnothing\}$. Якщо вихідних дуг не має, то пізній час закінчення процесу прирівнюється максимальному часу T виконання паралельного алгоритму, а якщо ні, то цей час обчислюється по (2.7).

У табл. 2.2 у двох крайніх колонках праворуч вписані ранні й пізні строки закінчення верхових процесів. Для кожної вершини ці значення є границями, у яких можна переміщати в часі моменти закінчення верхового процесу або, віднявши його тривалість, – моменти запуску процесу. Якщо пізні строки закінчення верхових процесів обчислювалися при $T = T_{кр}$, то на і-тих рядках у табл. 2.2 значення $\tau_2^T(i)$ й $\tau_1(i)$ виявляться рівними. Ці верхові процеси не можуть бути переміщені в часі. Неважко помітити, що при $T > T_{кр}$ різниця строків закінчення процесів буде рівна $\tau_2^T(i) - \tau_1(i) = T - T_{кр}$.

Цю залежність можна використовувати для контрольованого зниження середнього ступеня паралелізму шляхом збільшення часу виконання всього паралельного алгоритму.

Для задачі оптимального розміщення необхідно одержати аналітичне вираження функціонала, мінімізація якого привела б до потрібного результату.

2.7 Оптимізація розміщення процесів в алгоритмі

Представимо довжину кожного і-*того* верхового процесу на тимчасовій осі у вигляді ***відрізка одиничної функції***

$$f(\tau_i, t) = \begin{cases} 1 \text{ при } t \in [\tau_i - t_i, \tau_i], \\ 0 \text{ у противному разі} \end{cases} \quad (2.8)$$

де t – поточне час виконання алгоритму в інтервалі від 0 до T ;

t_i – тривалість виконання і-*того* процесу;

τ_i – час закінчення виконання і-*того* процесу.

Просумував відрізки одиничних функцій верхових процесів, одержимо аналітичне вираження ***функції щільності завантаження*** паралельно працюючих процесорів у будь-який момент часу $0 \le t \le T$:

$$F(\tau_1, \dots, \tau_i, \dots, \tau_p, t) = \sum_{i=1}^{p} f(\tau_i, t) . \quad (2.9)$$

На рис. 2.4а по параметрах графа інформаційної залежності, занесеним у табл. 2.2, наведені відрізки одиничних функцій для кожної вершини. Відрізки над віссю часу наближені до початку координат до границь ранніх строків закінчення верхових процесів $\tau_1(i)$, а під віссю – до границь пізніх строків закінчення $\tau_2^T(i)$. Пунктирні відрізки одиничних функцій показують можливість приміщення відрізка між лівої і правої границями.

Суми всіх одиничних відрізків функцій, наведені на рисунку 1.7б і 1.7г, відповідають функціям щільності завантаження паралельної обчислювальної системи, якщо керуватися при запуску обчислень відповідно тільки лівими або тільки правими границями закінчення верхових процесів. На цих функціях добре видне, що обчислювальна система в окремі моменти часу змушено включати в роботу те один, те два, те три процесорні обладнання.

Переміщаючи тим або іншим способом рухливі верхові процеси, наприклад, показані пунктирно на рис.2.4 а, і обчислюючи функцію

щільності завантаження, можна добитися використання в будь-який момент меншого числа процесорних обладнань. Один з можливих варіантів розміщення цих процесів, наведений на рис. 2.4а, дає функцію щільності завантаження, показану на рис. 2.4в, де використовуються для обчислень усього два процесори.

Об'єм обчислювальних робіт, зроблений системою протягом деякого відрізка часу $[\vartheta_k, \vartheta_l] \subset [0, T]$, можна обчислити шляхом інтегрування за часом функції щільності завантаження:

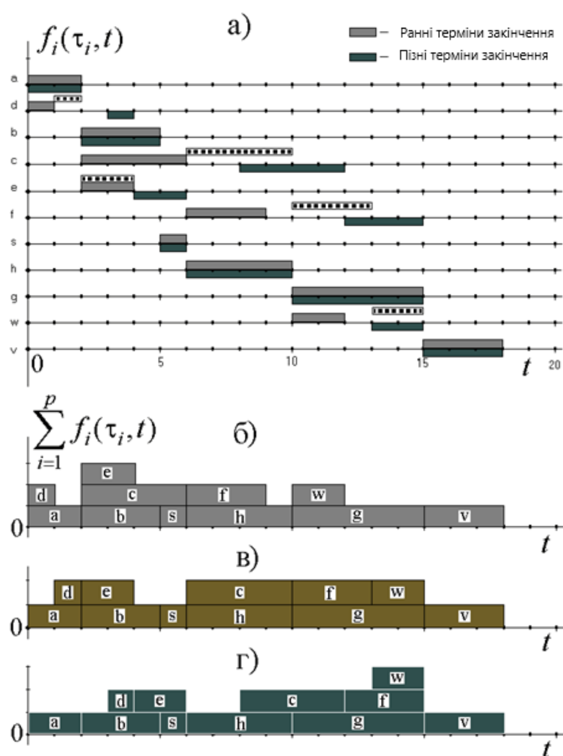

Рисунок 2.4 – Розподіл процесів за часом

$$\Phi(\tau_1,...,\tau_p,\vartheta_k,\vartheta_l) = \int_{\vartheta_k}^{\vartheta_l} F(\tau_1,...,\tau_p,t)\,dt \,. \tag{2.10}$$

Функція завантаження інтервалу $[\vartheta_k, \vartheta_l] \subset [0,T]$ є функцією значень $\vartheta_l > \vartheta_k$, $\vartheta_l, \vartheta_k \in \{t_j \mid j = 1,2,\ldots;\ t_j \subset [0,T]\}$ і може ухвалювати множину значень, обумовлених видом функції щільності завантаження в обраному інтервалі. Із множини значень функції завантаження інтервалу $[\vartheta_k, \vartheta_l] \subset [0,T]$ виберемо найменше значення, назвавши його ***функцією мінімального завантаження:***

$$\varphi^{(T)}(\vartheta_k, \vartheta_l) = \min_{\{\tau_1,\ldots,\tau_p\}} \Phi(\tau_1,\ldots,\tau_i,\ldots,\tau_p,\vartheta_k,\vartheta_l), \qquad (2.11)$$

де τ_i, $\tau_i \in \{t_j \mid j = 1,2,\ldots;\ t_j \subset [0,T]\}$ – момент закінчення i-*того* процесу.

При даному T та припустимих значеннях множини часів закінчення процесів $\{\tau_1,\ldots,\tau_p\}$ функція мінімального завантаження визначає мінімально можливий обсяг робіт, який може бути виконаний протягом інтервалу $[\vartheta_k, \vartheta_l]$.

Переміщувані процеси можуть змінювати своє положення лише в границях раннього τ_{1i} та пізнього τ_{2i}^T строків свого закінчення. При цьому можуть виникнути ситуації, у яких для обраного інтервалу $[\vartheta_k, \vartheta_l]$ неможливо хоча б частково виконати зсув відрізка одиничної функції вершини або їх переміщення нічого не змінює. До них можна віднести, наприклад, процеси початку і кінця паралельного алгоритму та частково процеси, що визначають критичний час алгоритму.

2.8 Організація взаємного обміну даними

Розглянуті формальні методи розпаралелення процесів і оптимального їхнього розміщення по процесорах і в часі дозволяють увесь обчислювальний процес сконфігурувати так, що готовність усіх даних, які визначають запуск будь-якого процесу, завжди відбудеться. Однак деякі особливості взаємного розташування взаємодіючих процесів і їх вплив на час виконання всього алгоритму, наприклад, можна обговорити, якщо на рис. 2.4в нанести всі інформаційні зв'язки. Ці зв'язки на рис. 2.5 показані в

умовно розширених проміжках між окремими процесами зі збереженням одночасності почав процесів, що перебувають на різних машинах.

По-перше, необхідно відзначити не одиничність існування декількох оптимальних у часі розміщень процесів і їх початків. По-друге, виділяються своєю нерухомістю по відношенню друг до друга процеси, що утворюють шлях із критичним часом. І, по-третє, не однозначність закріплення окремих процесів або груп процесів за конкретними процесорами.

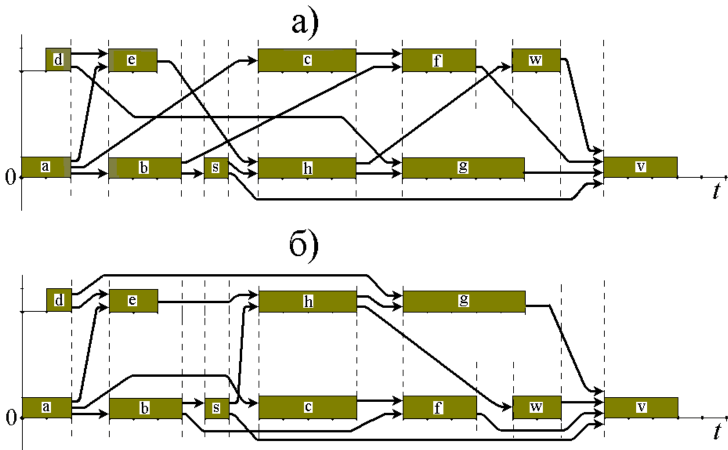

Рисунок 2.5 – Організація взаємного обміну даними

Переміщення процесів з однієї машини на іншу може змінити, і іноді досить суттєво, кількість передач даних з одного процесора на інший. Це безпосередньо вплине на включення існуючих у системі засобів передачі даних, які можуть виявитися досить повільними каналами міжпроцесорного зв'язку, а значить, може суттєво змінитися час рішення. Наприклад, на рис. 2.5 проміжки передач даних виділені вертикальними пунктирними лініями. Серед розподілу процесів на рис. 2.5а можна вказати на один інтервал передачі даних, який збільшить загальний час розв'язку, це зв'язок $w \rightarrow v$. На рис. 2.5б таких інтервала два: $s \rightarrow h$ і $g \rightarrow v$. Однак кількість міжпроцесорних передач на рис. 2.5а в багато разів більше, чим у варіанті розміщення по рис. 2.5б.

2.9 Прискорення при паралельних обчисленнях

Витрати часу на розв'язок задачі при програмуванні її для однопроцесорних обчислювальних машин, у головному, визначаються кваліфікацією програміста, який вибирає найбільш удалий і швидкий з відомих алгоритмів і раціонально зв'язує оператори мови в тексті програми.

Сумарний час розв'язку конкретної задачі більш точно можна одержати при її пробному прогоні на машині. Як правило інших непередбачених витрат часу тут не спостерігається.

Витрати часу на розв'язок розпаралелених задач містять у собі, крім названих вище, часи, затрачувані на взаємодію за допомогою загальних даних, і часи, витрачені процесами на очікування готовності даних.

Тому що основною метою застосування методів паралельного програмування для багатопроцесорних систем є прискорення обчислень у порівнянні з обчисленнями на однопроцесорній обчислювальній машині, то виникає природнє бажання оцінити, у скільки раз поменшається час розв'язку конкретної задачі, якщо буде використано p процесорів.

Прискоренням паралельного алгоритму називають відношення:

$$S_p = \frac{\text{час виконання алгоритма на одному процесорі}}{\text{час виконання алгоритма на } p \text{ процесорах}} = \frac{T_1}{T_p} = \frac{T_1}{T_{\text{кр}} \cdot (1+\alpha)}. \quad (2.12)$$

де $T_{\text{кр}} \geq T_1/p$ – мінімально можливий час виконання самого тривалого процесу розпаралелюваної задачі, який в сіткових графіках паралельно виконуваних робіт називають критичним шляхом;

α – коефіцієнт, що показує відносну частку часу, затрачуваного на обмін даними по каналах зв'язку між процесами, і вимушені простої процесів на критичному шляху через очікування готовності даних.

Ідеальним прискоренням при використанні p процесорів, яке реально недосяжне навіть для спеціально підібраних задачах із взаємодіючими процесами, можна вважати значення $S_p = p$.

Ступенем паралелізму обчислювального алгоритму називають число операцій, які можна виконувати паралельно.

У кожний момент розгортання паралельного обчислювального процесу ступінь паралелізму може змінюватися від одиниці (виконується

один процес) до деякого числа $n \le p$, обумовленого конкретною задачею, що й обмежується числом наявних процесорів.

Середнім ступенем паралелізму називають відношення загального числа операцій (точніше – процесів) на всіх дискретних відрізках часу до загального числа дискретних інтервалів, у сумі, що представляють повний час виконання паралельних процесів.

$$\overline{n} = \frac{\sum_i n_i \cdot \tau_i}{\sum_i \tau_i} = \frac{\sum_i n_i \cdot \tau_i}{T}, \qquad (2.13)$$

де n_i – число процесів (або операцій) на i-тому відрізку часу;

τ_i – тривалість інтервалу, на якому $n_i = const$;

T – час виконання паралельного алгоритму.

Якщо протягом усього алгоритму $n_i = n$, то $\overline{n} = n$.

2.10 Операції обміну даними

Найпростішою формою обміну повідомленнями є обмін між двома точками (point-to-point). У ньому беруть участь тільки два процеси: процес-відправник і процес-одержувач.

Є кілька різновидів такого обміну:

– синхронний обмін, який супроводжується повідомленням про закінчення прийому повідомлення;

– асинхронний обмін, який повідомленням не супроводжується;

– блокують прийом/передача, які припиняють виконання процесу на час прийому повідомлення

– прийом / передача без блокування, в якому виконання процесу триває в фоновому режимі, але в потрібний момент програма передавач може запросити підтвердження закінчення прийому повідомлення.

Більш різноманітною формою обміну є операція колективного обміну. У неї залучаються не два, а велика кількість процесів. Різновидами колективного обміну є:

– широкомовна передача – один процес передає повідомлення всім іншим процесам;

– обмін з бар'єром – це форма синхронізації роботи процесів, коли обмін повідомленнями відбувається тільки після того, як до відповідної процедури звернулося певна кількість процесів;

– операції приведення – вхідними даними виступають декількох процесів, а результат – одне значення, яке стає доступним всім процесам, які беруть участь в обміні.

2.11 Типи даних та коди завершення підпрограм

У MPI прийнята своя система позначення простих типів даних, яка в багатьох випадках відповідає типам даних в мовах C / C ++.

Таблиця 2.3 – Типи даних MPI для мови C/C++

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

У MPI повинні дотримуватися правила сумісності типів. Відповідність типів, як правило, повинно мати місце в процедурах відправки і процедурах прийому повідомлень.

Кодами завершення підпрограм MPI, в разі функцій C / C ++, є повернені значення функцій. Успішне завершення представляється константою MPI_SUCCESS, а невдале – MPI_ERR_OTHER.

Системою розпізнаються і інші помилки. Замість числових кодів в програмах зазвичай використовують спеціальні іменовані константи. Серед них:

– MPI_ERR_BUFFER – неправильний покажчик на буфер;
– MPI_ERR_COMM – неправильний комунікатор;
– MPI_ERR_RANK– неправильний ранг;

– MPI_ERR_OP – неправильна операція;
– MPI_ERR_ARG – неправильний аргумент;
– MPI_ERR_UNKNOWN – невідома помилка;
– MPI_ERR_TRUNCATE – повідомлення обрізане при прийомі;
– MPI_ERR_INTERN – внутрішня помилка. Зазвичай виникає, якщо системі не вистачає пам'яті.

2.12 Структура програми в середовищі MPI

На початку програми, відразу після її заголовка, необхідно підключити відповідний заголовки. У програмі на мові C це mpi.h а для C++ це mpi++.h:

#include "mpi.h"
#include "mpi++.h"

Підпрограма складається з:
– MPI_Init – підпрограми ініціалізації;
– MPI_Initialized – перевірка ініціалізації програми MPI;
– MPI_COMM_WORLD – комунікатор;
– MPI_Finalize – видалення структури даних MPI;
– MPI_Abort – переривання виконання процесів на комунікаторі;
– MPI_Comm_size – повертає кількість процесів;
– MPI_Comm_rank – визначення рангу процесу.

2.13 Підпрограми обміну даними

Передбачено чотири різновиди двухточкового обміну: синхронний, асинхронний, блокуючий і неблокуючий. У MPI є також чотири режими обміну, що розрізняються умовами ініціалізації і завершення передачі повідомлення:

– стандартна передача вважається виконаною і завершується, як тільки повідомлення надіслано, незалежно від того, дійшло воно до адресата чи ні. У стандартному режимі передача повідомлення може починатися, навіть якщо ще не розпочато його прийом;

– синхронна передача відрізняється від стандартної тим, що вона не завершується доти, поки не буде завершено прийом повідомлення; адресат, отримавши повідомлення, посилає процесу, який відправив його, повідомлення, яке повинно бути отримано відправником для того, щоб обмін вважався виконаним;

– буферизована передача завершується відразу ж, так як повідомлення копіюється в системний буфер, де і очікує своєї черги на пересилку; завершується буферизована передача незалежно від того, виконаний прийом повідомлення чи ні;

– передача "по готовності" починається тільки в тому випадку, коли адресат ініціалізує прийом повідомлення, а завершується відразу ж, незалежно від того, прийнято повідомлення чи ні.

Кожен з цих чотирьох режимів передбачений як в блокуючої, так і в неблокуючої формах.

У MPI прийняті – такі угоди про імена підпрограм двухточкового обміну: MPI_[I][R, S, B]Send

де MPI_ – префікс, що приписується всім підпрограм MPI;

[I] – (від Immediate – негайний) позначає режим блокування;

[R | S | B] – ці префікси позначає режим обміну, відповідно: по готовності, синхронний і буферізоване.

Відсутність префікса позначає підпрограму стандартного обміну. Таким чином, всього 8 різновидів операції передачі повідомлень.

2.14 Обмін повідомлень

Основними підпрограмами двухточкового обміну повідомленнями є MPI_Send і MPI_Recv. MPI_Send – це підпрограма стандартної блокуючої передачі. Вона блокує виконання передаючого процесу до завершення передачі повідомлення (що, однак, не гарантує завершення його прийому на стороні що приймає). В нотації мови С ++ оголошення цієї підпрограми має наступний вигляд:

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

Вхідні параметри підпрограми MPI_Send:

– buf – адреса першого елемента в буфері передачі з ім'ям buf;

– count – кількість елементів в буфері передачі;

– datatype – тип MPI кожного елемента що пересилається;

– dest – ранг процесу-одержувача повідомлення, що представляється цілим числом від 0 до n-1, де n - число процесів в області взаємодії, який визначається комунікатором з ім'ям comm;

– tag – тег (бирка, ярлик, мітка) повідомлення, що представляється цілим числом, нумеруються черговість відсилання повідомлень від одного і того ж джерела до одного і того ж приймачу, що зберігає задану черговість прийому незалежно від маршруту;

– comm – ідентифікатор приватного (локального) комунікатора;

Підпрограма стандартного блокуючого прийому оголошується так:

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

Вхідними параметрами цієї підпрограми є:

– count – максимальна кількість елементів в буфері прийому, яку можна визначити за допомогою підпрограми MPI_Get_count;

– datatype – тип даних, що приймаються;

– source – ранг джерела, в якості якого можна використовувати і спеціальну константу MPI_ANY_SOURCE, задану довільне значення рангу, процес якого може виконувати роль ведучого процесу ("джокера");

– tag – мітка повідомлення або константа MPI_ANY_TAG (joker – джокер), відповідний безпідставного значенням тега;

– comm – ідентифікатор конкретного комунікатора.

При синхронному обміні адресат посилає джерела повідомлення про завершення прийому. Тільки після отримання цього повідомлення обмін вважається завершеним і джерело "знає", що його повідомлення отримано. Синхронна передача виконується за допомогою підпрограми MPI_Ssend:

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

Параметри у неї такі ж, як і у підпрограми MPI_Send.

Буферизована передача завершується відразу, оскільки повідомлення негайно копіюється в буфер для подальшої передачі. На відміну від стандартного обміну, в цьому випадку робота джерела і адресата не синхронізовано. Параметри підпрограми буферизованого обміну MPI_Bsend:

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

При виконанні буферизованого обміну програміст повинен заздалегідь створити буфер достатнього розміру за допомогою виклику підпрограми:

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

В результаті виклику створюється буфер з ім'ям buf і розміром size в байтах, який можна використовувати тільки один раз, після чого його потрібно відключити шляхом виклику підпрограми відключення:

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

Передачу по готовності виконують за допомогою підпрограми MPI_Rsend, параметри якої збігаються з MPI_Send:

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

Додаток А

Зразок виконання розрахункового завдання

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Кафедра «ОБЧИСЛЮВАЛЬНА ТЕХНІКА І ПРОГРАМУВАННЯ»

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ
Розрахунково-графічна робота

Виконав:
студент групи КІТ -117а
Горносталь О.А.

Викладач: проф. Черних О.П.
*Прийнято:*_________________
«___ »_________2020 р.

Харків – 2020

Продовження Додатка А

ЗМІСТ

Вступ
1 Аналітична частина
1.1 Лексикографічне упорядкування
1.2 Пошук термінів закінчення процесів
1.3 Побудова часових діаграм
1.4 Побудова графіків щільності завантаження процесорів
1.5 Побудова діаграми розміщення процесів на двох процесорах
1.6 Розрахунок характеристик паралельного алгоритму
2 Розробка опису і тексту програми
2.1 Розробка алгоритму функціонування програми
2.2 Опис програми
2.3 Текст програми
2.4 Файл вхідних даних
2.5 Результат роботи
3 Висновки

Продовження Додатка А

ВСТУП

Ідея розпаралелювання обчислень заснована на тому, що більшість задач може бути розділено на набір менших задач, що можуть виконуватися одночасно. Паралельні обчислення використовуються багато років в основному у високопродуктивних обчисленнях, але в останній час до них виріс інтерес внаслідок існування фізичних обмежень на ріст тактової частоти процесорів. Паралельні та розподіленні обчислення стали домінуючою парадигмою в архітектурі комп'ютерів, в основному у формі багатоядерних процесорів.

Але навіть при наявності багатьох ядер вже навіть в одному процесорі, паралелізація програми – це все одна задача для розробника програми. І задача ця зовсім не легка, та за складністю значно перевершує програмування класичне. Для початку програміст має вирішити, які частини програми можуть виконуватися одночасно, яку схему паралелізму обрати, як окремі процеси, на які поділяється програма, будуть взаємодіяти між собою.

Продовження Додатка А

1 АНАЛІТИЧНА ЧАСТИНА

1.1 Лексикографічне упорядкування

Зважений граф інформаційної залежності між процесами паралельної програми задається двома матрицями:

– матрицею інциденцій вершин:

2	3	2	2	3	7	5	5	8	1	6
6	6	7	5	1	4	8	1	4	4	4

де перший рядок задає вершини початку ребра, а другий – кінець відповідного ребра;

– матрицею ваг вершин:

2	3	7	5	6	8	1	4
1	2	4	1	3	2	1	5

де перший рядок задає ім’я вершини, а другий – відповідну вагу.

На основі вказаної матриці інциденцій можна побудувати невпорядкований граф інформаційної залежності. Даний граф зображений на рис. 1.

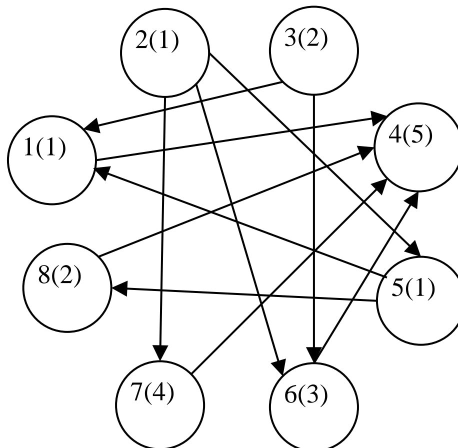

Рисунок 1 – Граф інформаційної залежності паралельної програми

Продовження Додатка А

Сутність лексикографічного упорядкування полягає в тому, щоб кожному процесу присвоїти індекс та розмістити вершини орграфа таким чином, щоб передача даних в паралельному процесі відбувалася тільки в сторону збільшення індексів.

При лексикографічному упорядкуванні першими обираються такі процеси, що не потребують вхідних даних для початку роботи. Такі елементи займають перший ярус в паралельній програмі. На наступному ярусі розміщуються процеси, що потребують вхідних даних лише від процесів попереднього ярусу.

$$S_{ij} = \begin{array}{c|cccccccc} & 1 & \mathbf{2} & \mathbf{3} & 4 & 5 & 6 & 7 & 8 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \mathbf{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

Після лексикографічного упорядкування граф інформаційної залежності має наступний вигляд (див. рис. 2)

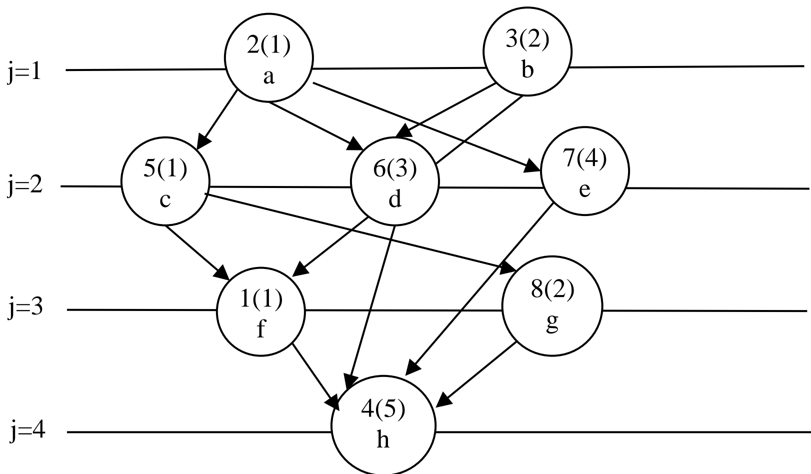

Рисунок 2 – Упорядкований граф інформаційної залежності

Продовження Додатка А

На рис. 2 величина j задає номер ярусу паралельного алгоритму.

1.2 Пошук термінів закінчення процесів

На рис. 3 відображено лексикографічне упорядкування графу із урахуванням ваги процесів.

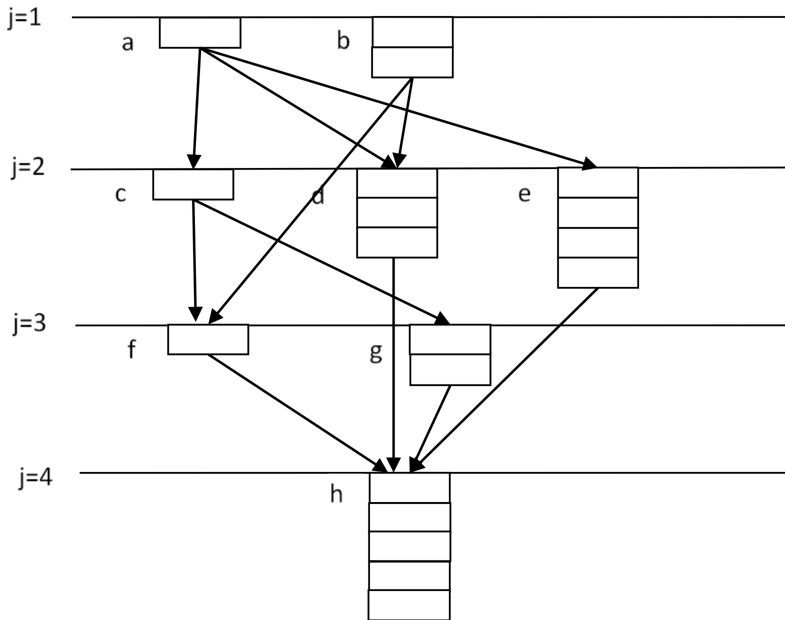

Рисунок 3 – Лексикографічне впорядкування графу із урахуванням ваги процесів

Можна визначити мінімальний час паралельного алгоритму $T_{кр}$ (критичний час), за який можливо виконати якомога швидше поставлене завдання. У нашому випадку $T_{кр}$=10 (див. рис. 3).

Продовження Додатка А

Кожний процес має такі характеристики як ранній та пізній терміни закінчення процесів. Якщо скористатися графом, що зображений на рис. 3, то ранні терміни закінчення кожного процесу можна знайти шляхом підняття кожного процесу максимально вгору, так, щоб не було змінене направлення зв'язків, що вказують на передачу даних. Тобто процес може починатися, одразу, з того моменту як для нього вже підготовлені дані, тобто закінчився процес, що підготовлює дані для поточного процесу.

Обчислення конкретного значення раннього строку закінчення j-го верхового процесу здійснюється по формулі:

$$\tau_1(j) := \max_{\mu \in \{\overline{1,p} \mid S_{\mu,j}=1 \ \& \ \tau_{1\mu} \neq 0\}} [\tau_{1\mu}] + t_j , \qquad (1)$$

де $\tau_{1\mu}$ – ранні строки закінчення процесів μ-тих вершин, для яких одночасно виконана умова, що на μ-тому рядку j-го стовпця матриці суміжності S перебуває одиниця і для μ-того рядка стовпчика τ_1 табл. 1 ранній строк закінчення вже знайдений. До найбільшого з $\tau_{1\mu}$-тих додається час виконання j-го процесу.

Розрахуємо за формулою (1) значення ранніх строків закінчення для восьмі процесів:

$\tau_1(a) = t(a) = 1$
$\tau_1(b) = t(b) = 2$
$\tau_1(c) = \tau_1(a) + t(c) = 1 + 1 = 2$
$\tau_1(d) = \max [\tau_1(a), \tau_1(b)] + t(c) = \max [1, 2] + 1 = 2 + 1 = 3$
$\tau_1(e) = \tau_1(a) + t(e) = 1 + 4 = 5$
$\tau_1(f) = \max [\tau_1(b), \tau_1(c)] + t(f) = \max [2, 2] + 1 = 2 + 1 = 3$
$\tau_1(g) = \tau_1(c) + t(g) = 2 + 2 = 4$
$\tau_1(h) = \max [\tau_1(d), \tau_1(e), \tau_1(f), \tau_1(g)] + t(h) = \max [3, 5, 3, 4] + 5 = 5 + 5 = 10$

Продовження Додатка А

Ранній строк закінчення самого останнього верхнього процесу на самому нижньому ярусі визначає мінімальний час виконання всього паралельного процесу. Цей час називають критичним часом $T = T_{\text{кр}}$. У нашому випадку $T_{кр}=10$. При усьому бажанні виконати паралельний алгоритм за менший час це принципово неможливо.

У межах критичного часу можна тепер знайти й найбільш пізні строки закінчення верхових процесів шляхом підтягування їх якнайближче до $T = T_{\text{кр}}$, не порушуючи спрямованості проекцій дуг критичних шляхів графа на тимчасову вісь. Відмінність полягає в організації перегляду вершин від самої пізньої по виконанню до самої ранньої в ГІЗ і використання іншої формули обчислення пізнього строку закінчення верхового процесу:

$$\tau_2^T(i) = \min_{\mu \in \{\overline{1,p} \mid S_{i,\mu}=1 \;\&\; \tau_{2\mu}^T \neq 0\}} [\tau_{2\mu}^T - t_\mu], \qquad (2)$$

де $\tau_{2\mu}^T$ – пізні строки закінчення процесів μ-тих вершин для заданого загального часу обчислень $T \geq T_{\text{кр}}$. Мінімум вибирається з μ-тих різниць, для яких одночасно виконана умова, що на μ-тому стовпці i-го рядка матриці суміжності S перебуває одиниця та для μ-того рядка стовпчика τ_2^T табл. 2.2 пізній строк закінчення вже знайдений.

Перегляд рядків у матриці суміжності та табл. 1 починається з останнього рядка $i = p = h$.

Розрахуємо за формулою (2) значення пізніх строків закінчення для восьмі процесів. Починаємо з останнього процесу.

$\tau_2(h) = \tau_1(h) = 10$
$\tau_2(g) = \tau_1(h) – t(h)= 10 – 5 = 5$
$\tau_2(f) = \tau_1(h) – t(h)= 10 – 5 = 5$
$\tau_2(e) = \tau_1(h) – t(h)= 10 – 5 = 5$
$\tau_2(d) = \tau_1(h) – t(h)= 10 – 5 = 5$
$\tau_2(c) = \min [(\tau_1(f) – t(f)), (\tau_1(g) – t(g))] = \min [(5 – 1), (5 – 2)] = \min [4, 3] = 3$
$\tau_2(b) = \min [(\tau_1(d) – t(d)), (\tau_1(f) – t(f))] = \min [(5 – 3), (5 – 1)] = \min [2, 4] = 2$
$\tau_2(a) = \min [(\tau_1(c) – t(c)), [(\tau_1(d) – t(d)), (\tau_1(e) – t(e))] =$
$= \min [(3 – 1), (5 – 3), (5 – 4)] = \min [2, 2, 1] = 1$

Продовження Додатка А

В табл. 1 вказані розраховані ранні та пізні терміни закінчення процесів.

Таблиця 1 – Ранні та пізні терміни закінчення процесів програми

Початкове ім'я	**$\delta^{+}(v_i)$**	**$\delta^{-}(v_i)$**	**Вага t**	**Ярус j**	**$\tau_1(k)$**	**$\tau_2(k)$**
a	{c, d, e}	{∅}	1	1	1	1
b	{d, f}	{∅}	2	1	2	2
c	{f, g}	{a}	1	2	2	3
d	{h}	{a, b}	3	2	3	5
e	{h}	{a}	4	2	5	5
f	{h}	{b, c}	1	3	3	5
g	{h}	{c}	2	3	4	5
h	{∅}	{d, e, f, g}	5	4	10	10

1.3 Побудова часових діаграм

На основі значень термінів закінчення процесів, отриманих у попередньому пункті, можна побудувати часову діаграму.

Представимо довжину кожного i-того верхового процесу на тимчасовій осі у вигляді відрізка одиничної функції

$$f(\tau_i, t) = \begin{cases} 1 \text{ при } t \in [\tau_i - t_i, \tau_i], \\ 0 \text{ у противному разі} \end{cases}, \qquad (3)$$

де t – поточне час виконання алгоритму в інтервалі від 0 до T;

t_i – тривалість виконання i-того процесу;

τ_i – час закінчення виконання i-того процесу.

Продовження Додатка А

Розрахуємо довжину кожного і-того верхового процесу для ранніх терміні закінчення на тимчасовій вісі за формулою (3):

$f(\tau_1(a), t(a)) = [\tau_1(a) - t(a)], \tau_1(a)] = [(1 - 1), 1] = [0, 1]$
$f(\tau_1(b), t(b)) = [\tau_1(b) - t(b)], \tau_1(b)] = [(2 - 2), 2] = [0, 2]$
$f(\tau_1(c), t(c)) = [\tau_1(c) - t(c)], \tau_1(c)] = [(2 - 1), 2] = [1, 2]$
$f(\tau_1(d), t(d)) = [\tau_1(d) - t(d)], \tau_1(d)] = [(3 - 3), 3] = [0, 3]$
$f(\tau_1(e), t(e)) = [\tau_1(e) - t(e)], \tau_1(e)] = [(5 - 4), 5] = [1, 5]$
$f(\tau_1(f), t(f)) = [\tau_1(f) - t(f)], \tau_1(f)] = [(3 - 1), 3] = [2, 3]$
$f(\tau_1(g), t(g)) = [\tau_1(g) - t(g)], \tau_1(g)] = [(4 - 2), 4] = [2, 4]$
$f(\tau_1(h), t(h)) = [\tau_1(h) - t(h)], \tau_1(h)] = [(10 - 5), 10] =[5, 10]$

Розрахуємо довжину кожного і-того верхового процесу для пізніх терміні закінчення на тимчасовій вісі за формулою (3):

$f(\tau_2(a), t(a)) = [\tau 2_1(a) - t(a)], \tau_2(a)] = [(1 - 1), 1] = [0, 1]$
$f(\tau_2(b), t(b)) = [\tau_2(b) - t(b)], \tau_2(b)] = [(2 - 2), 2] = [0, 2]$
$f(\tau_2(c), t(c)) = [\tau_2(c) - t(c)], \tau_2(c)] = [(3 - 1), 3] = [2, 3]$
$f(\tau_2(d), t(d)) = [\tau_2(d) - t(d)], \tau_2(d)] = [(5 - 3), 5] = [2, 5]$
$f(\tau_2(e), t(e)) = [\tau_2(e) - t(e)], \tau_2(e)] = [(5 - 4), 5] = [1, 5]$
$f(\tau_2(f), t(f)) = [\tau_2(f) - t(f)], \tau_2(f)] = [(5 - 1), 5] = [4, 5]$
$f(\tau_2(g), t(g)) = [\tau_2(g) - t(g)], \tau_2(g)] = [(5 - 2), 5] = [3, 5]$
$f(\tau_2(h), t(h)) = [\tau_2(h) - t(h)], \tau_2(h)] = [(10 - 5), 10] =[5, 10]$

Побудована часова діаграма зображена на рис. 4. Ліворуч на вертикальній вісі вказані номера відповідних процесів, по горизонтальній вісі задається часова вісь із відносними одиницями часу.

Продовження Додатка А

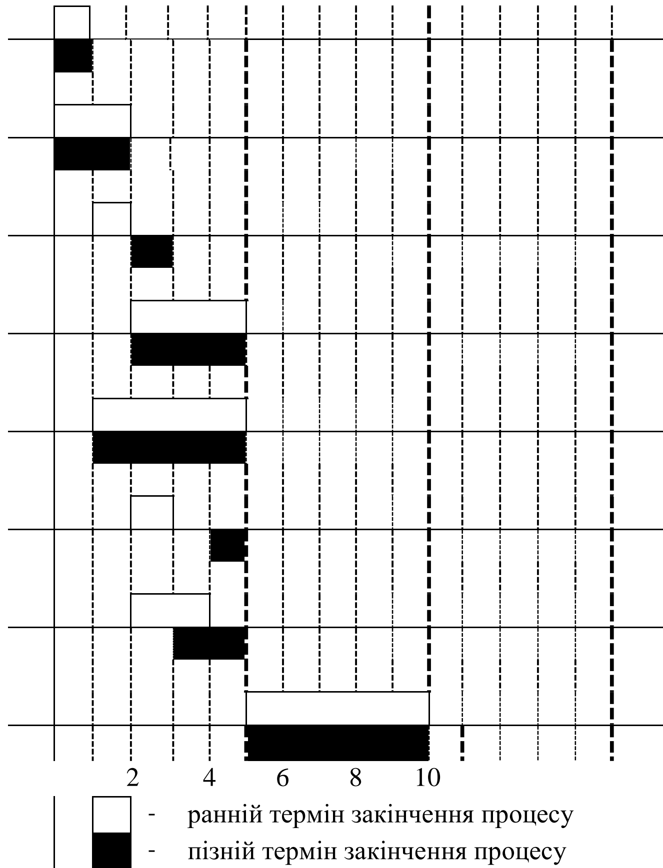

Рисунок 4 – Часова діаграма ранніх та пізніх термінів закінчення процесів

1.4 Побудова графіків щільності завантаження процесорів

Спираючись на часову діаграму, що зображена на рис. 4, можна побудувати графіки щільності завантаження процесорів.

Продовження Додатка А

Графіки щільності для граничних ситуацій (для випадків, коли всі процеси мають ранні терміни закінчення, та коли всі процеси мають пізні терміни закінчення) зображені на рис. 5, 6.

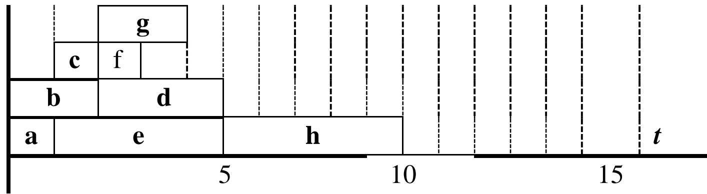

Рисунок 5 – Графік щільності завантаження процесорів для випадку закінчення всіх процесів у ранній термін

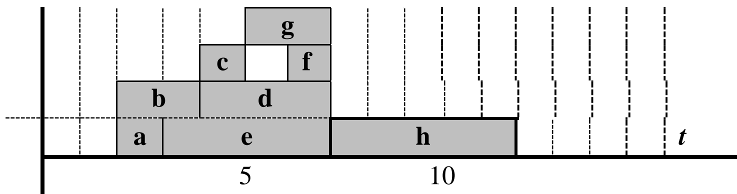

Рисунок 6 – Графік щільності завантаження процесорів для випадку закінчення всіх процесів у пізній термін

Із побудованих графіків щільності видно, що ступінь паралелізму, для випадку із ранніми термінами та для випадку із пізніми термінами закінчення процесів досягає 4.

Якщо попереставляти процеси в установлених для них межах ранніх та пізніх термінів закінчення, то можна досягти того, щоб ступінь паралелізму буде досягати максимум 2. Графік щільності тоді може мати вигляд (рис. 7).

Продовження Додатка А

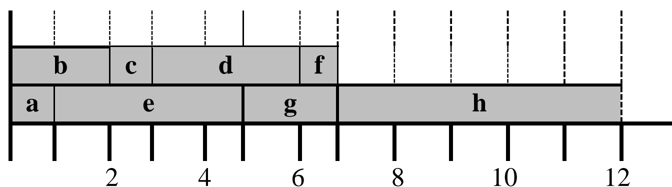

Рисунок 7 – Графік щільності завантаження процесорів із максимальним ступенем паралелізму рівним 2

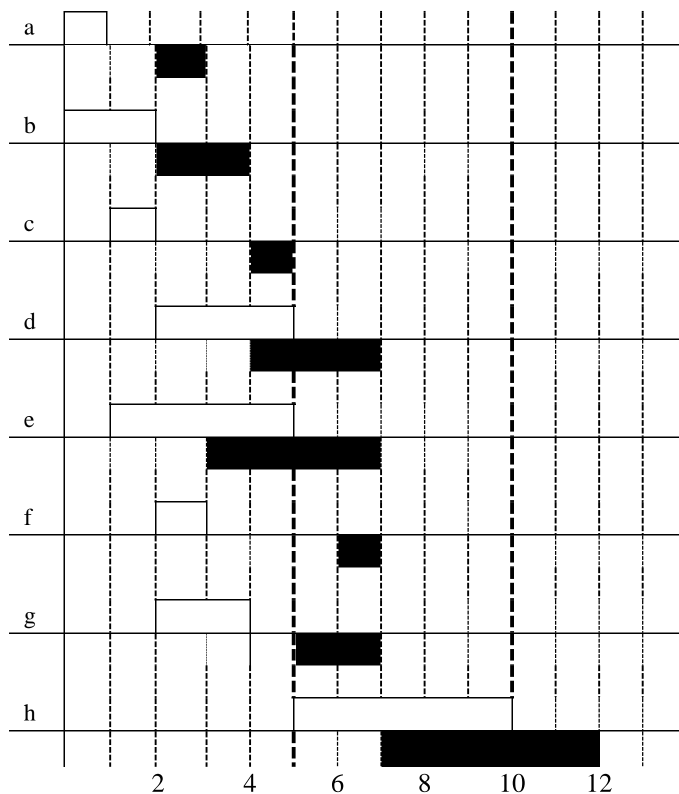

Рисунок 8 – Часова діаграма ранніх та пізніх термінів закінчення процесів із максимальним ступенем паралелізму рівним 2

Продовження Додатка А

1.5 Побудова діаграми розміщення процесів на двох процесорах

Якщо на графіку щільності відобразити зв'язки для передачі даних між процесами, то буде отримана діаграма розміщення процесів на процесорах. Із графіка щільності (рис. 7) можна побудувати наступну діаграму (рис. 8).

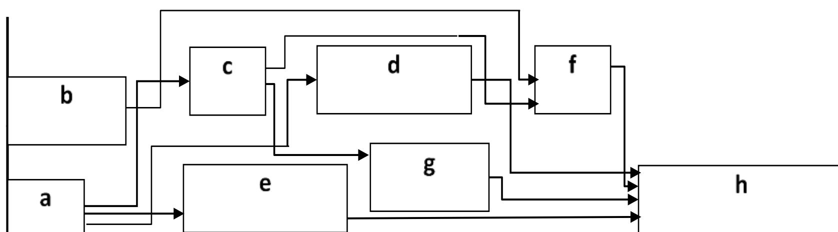

Рисунок 8 – Діаграма розміщення процесів на двох процесорах (не оптимізована)

На цій діаграмі повністю видно розміщення процесів на процесорах та зв'язки між ними. Видно, також, що міжпроцесорна передача здійснюється 5 разів, що може уповільнити роботу програми. В такому випадку необхідно спробувати розмістити процеси таким чином, щоб максимально скоротити кількість міжпроцесорних передач, та не порушити направлення передачі даних між процесами.

Після деяких перетворень можна отримати діаграму, що зображена на рис. 9.

Продовження Додатка А

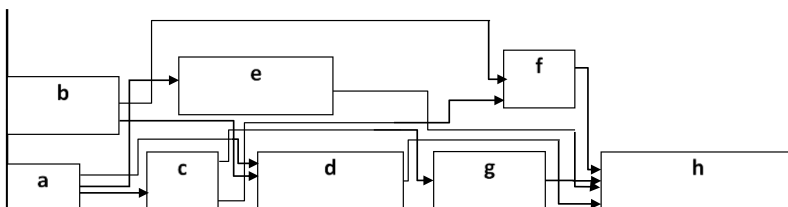

Рисунок 10 – Діаграма розміщення процесів на двох процесорах

На діаграмі, що зображена на рис. 9, видно, що навіть після допустимих переміщень додаткової оптимізації досягти неможливо (залишається 5 міжпроцесорних передач).

1.6 Розрахунок характеристик паралельного алгоритму

Ступінь паралелізму – це величина, що характеризує кількість процесорів, на яких одночасно виконуються процеси програми.

Якщо процесори не повністю завантажені, то розраховується середній ступінь паралелізму. Розрахунки виконуються за формулою:

$$\overline{n} = \frac{\sum_i n_i \cdot \tau_i}{\sum_i \tau_i} = \frac{\sum_i n_i \cdot \tau_i}{T}, \qquad (4)$$

де n_i – число процесів на i-ому відрізку часу;

τ_i – довжина інтервалу на якому n_i = const;

T – час виконання паралельного алгоритму.

Відповідно до формули (4) середній ступінь паралелізму для заданого паралельного алгоритму дорівнює:

$$\overline{n} = \frac{19}{14} = 1{,}357$$

Продовження Додатка А

Прискорення алгоритму розраховується за наступною формулою:

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_{kp} \cdot (1+\alpha)}, \quad (5)$$

де T_1 – час виконання алгоритму на одному комп’ютері;

T_p – час виконання алгоритму на *p* процесорах.

Відповідно до формули (5) прискорення алгоритму, що розглядається в даній роботі дорівнює:

$$S_p(\alpha = 0) = \frac{19}{14} = 1{,}357;$$

$$S_p(\alpha = 0.5) = \frac{19}{21} = 0{,}905;$$

Ефективність алгоритму розраховується шляхом ділення отриманого прискорення алгоритму на кількість процесорів, на яких дана програма виконується. Тобто формула ефективності має вигляд:

$$E_p = \frac{S_p}{p}; \quad (6)$$

Тоді із формули (6) можна отримати:

$$E_p(0) = \frac{1{,}357}{2} * 100\% = 67{,}85\%;$$

$$E_p(0.5) = \frac{0{,}905}{2} * 100\% = 45{,}25\%.$$

Продовження Додатка А

В результаті виконання зазначених операцій у першому розділі отримуємо таку відповідність номерів та імен процесів:

a	b	c	d	e	f	g	h
2	3	5	6	7	1	8	4

Продовження Додатка А

3 РОЗРОБКА ОПИСУ І ТЕКСТУ ПРОГРАМИ

3.1 Розробка алгоритму функціонування програми

При розробці алгоритму було розглянуто взаємодію процесів за схемою згідно з завданням. Процеси було впорядковано для організації оптимального взаємного обміну між двома процесорами, адже обмін даними це дуже ємка за часом процедура. В результаті розміщення процесів на двох процесорних об'єктах отримали 4 зв'язки між об'єктами, які можна розбити на 2 канали – перший відповідає за передачу даних з першого процесорного об'єкту до другого, а другий – за передачу даних у зворотному напрямку.

3.2 Опис програми

Підпрограми, що забезпечують обмін даними, є об'єктами зв'язку, за яких в MPI закріплені два поняття: інтракомунікатор і інтеркомунікатор (intracommunicator і intercommunicator). Ці об'єкти зв'язку покликані визначати ті розмежовувати область взаємодії (область зв'язку) усередині груп і між групами взаємодіючих процесів, розташованих на одному процесорі, і між групами процесів, розташованих на різних процесорах. Всі процеси, що належать одній області взаємодії, можуть обмінюватися повідомленнями тільки через свій комунікатор.

Всім процесам у будь-якій конкретній області взаємодії привласнюються цілі додатні номери від 0 до деякого максимального. Із процесів, що входять в існуючу область взаємодії, можуть створюватися нові області взаємодії. Нумерація процесів у різних областях взаємодії незалежна.

Серед безліч підпрограм MPI можна виділити шість, які використовуються в програмі:

1) **MPI_Init(int *argc, char **argv)** – підключення до MPI. Аргументи argc і argv потрібні тільки в програмах на C, де вони при запуску програми задають кількість аргументів у командному рядку та вектор цих аргументів. Даний виклик передує усім іншим викликам підпрограм MPI.

Продовження Додатка А

2) **MPI_Finalize()** – завершення роботи з MPI. Після виклику даної підпрограми не можна викликати підпрограми MPI. **MPI_Finalize** повинні викликати всі процеси перед завершенням своєї роботи.

3) **MPI_Comm_size(comm, size)** – визначення розміру області взаємодії. Тут **comm** - вхідний параметр, що задає ім'я комунікатору, а вихідним є параметр цілого типу **size**, що визначає кількість процесів в області взаємодії.

4) **MPI_Comm_rank(comm, pid)** – визначення номера (рангу) процесу. Тут **pid** - ідентифікатор процесу, що зв'язується з областю взаємодії, що належить комунікатору **comm**.

5) **MPI_Send(buf, count, datatype, dest, tag, comm)** – відправлення повідомлення. Всі параметри є вхідними: **buf** - адресу буфера відправлення, **count** - кількість елементів, що пересилаються, даних (ненегативне ціле значення), **datatype** - тип пересилання даних, **tag** - циклодром повідомлення (ціле значення), **comm** - комунікатор.

6) **MPI_Recv(buf, count, datatype, source, tag, comm, status)** – прийом повідомлення. Вихідні параметри: **buf** - адреса буфера, що одержує, і **status** - статус завершення операції; вхідні параметри: **source** - ідентифікатор процесу, від якого одержують повідомлення, а всі інші мають призначення, аналогічне параметрам **MPI_Send(...)**.

Програма повинна ініціалізувати MPI-систему, створювати комунікатори для передачі даних, а також проводити передачу, прийом та виведення даних від усіх паралельних процесів.

3.3 Текст програми

```
#include "mpi.h"
#include <algorithm>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <windows.h>
using namespace std;
```

Продовження Додатка А

```
char hostname[MPI_MAX_PROCESSOR_NAME];
MPI_Comm my_comm, inter_comm;
ofstream fileStream;

// Запис до файлу мітки поточного часу
void printTime() {
	char buf[18];
	_SYSTEMTIME timeNow;
	ZeroMemory(&timeNow, sizeof(timeNow));
	GetLocalTime(&timeNow);
	sprintf(buf, "[%02d:%02d:%02d:%03d] ", timeNow.wHour,
timeNow.wMinute, timeNow.wSecond, timeNow.wMilliseconds);
	fileStream << buf;
}
// Очікування, реалізовано на базі MPI викликів
void waiting(double sec) {
	double beg = MPI_Wtime();
	while (MPI_Wtime() - beg < sec);
}
// Клас, який містить дані процесу і його поведінку
class Proc {
private:
// Дані процесу
	struct procData {
		int	rank;
		int	weight;
		vector<int> sendTo;
		vector<int> recvFrom;
	};

	vector<procData> procTable; // Таблиця процесів, що була прочитана з
файлу конфігурації
	static const int bufSize = 1024; // Буфер для передачі між процесами
	char buffer[bufSize];
// Визначає, знаходиться два процеси в інтер чи интракомунікаторі
	MPI_Comm get_comm_(int rangA, int rangB) {
		if ((find(r0.begin(), r0.end(), rangA) != r0.end() &&
			find(r0.begin(), r0.end(), rangB) != r0.end()) ||
			(find(r1.begin(), r1.end(), rangA) != r1.end() &&
```

Продовження Додатка Б

```
                find(r1.begin(), r1.end(), rangB) != r1.end()))
            return my_comm;
        else
            return inter_comm;
    }

    // Повертає локальний ранг в групі джерела повідомлення
    int getLocalRank(int rank) {
        unsigned int i;
        for (i = 0; i < r0.size(); i++)
            if (r0[i] == rank) return i;
        for (i = 0; i < r1.size(); i++)
            if (r1[i] == rank) return i;

        return 0xFFFF;
    }
public:
    vector<int>  r0, r1;          // Ранги процесів двох груп

    Proc(void) {
        memset(buffer, 0, bufSize);
    }
    virtual ~Proc() {}

    // Зчитування налаштування процесу
    void read(ifstream &inf) {
        procData tab;
        char buf[1024];

        while (1) {
            stringstream line;
            if (inf.eof()) break;

            // Читаємо всю строку
            inf.getline(buf, 1024);
            if (buf[0] == '#' || buf[0] == 0) continue;

            // Ранги груп
            if (!strncmp(buf, "r0", 2) || !strncmp(buf, "r1", 2)) {
```

```
                    line << buf;
                    line >> buf;

                    vector<int> *v = new vector<int>();
                    if (!strcmp(buf, "r0")) v = &r0;
                    else if (!strcmp(buf, "r1")) v = &r1;
                    line >> buf;

                    while (buf[0]) {
                        v->push_back(atoi(buf) - 1);
                        line >> buf;
                    }
                    continue;
                }

                // Інформація про вузли
                line << buf;
                line >> tab.rank >> tab.weight;
                line >> buf;

                while (buf[0]) {
                    if (buf[0] == '>') tab.sendTo.push_back(atoi(buf + 1) -
1);

                    else if (buf[0] == '<') tab.recvFrom.push_back(atoi(buf
+ 1) - 1);

                    line >> buf;
                }

                tab.rank--;
                procTable.push_back(tab);
                tab.sendTo.clear();
                tab.recvFrom.clear();
            }
        }

        // Запуск процесу с заданим рангом на виконання
        void run(int rank) {
            vector<int>::iterator           i;
            vector<procData>::iterator      proc;
            MPI_Status                          st;
```

```
		// Знаходимо процес з рангом rank в таблиці процесів, що був
зчитан з файлу конфігурації
		bool wasIt = false;
	for (proc = procTable.begin(); proc < procTable.end(); proc++)
			if (rank == proc->rank) {
				wasIt = true;
				break;
			}

		if (!wasIt) {
		printTime();
		fileStream << "Rank " << rank << " in the config file is not
described\n";
			return;
		}

		// Час, витрачений на прийом і віддачу всіх даних
		double t = MPI_Wtime();

		// Отримуємо повідомлення
		i = proc->recvFrom.begin();
		while (i != proc->recvFrom.end()) {
			// Отримання повідомлення
			MPI_Comm c = get_comm_(*i, rank);		// В одном
коммуникаторе чи разных
			int src_rank = getLocalRank(*i);		// Ранг процесу в групі
яка є джерелом повідомлення
			MPI_Recv(buffer, bufSize, MPI_CHAR, src_rank,
MPI_ANY_TAG, c, &st);

			printTime();
			fileStream << "Recv: " << rank + 1 << " <- " << *i + 1 << "
(buf = '" << buffer << "')\n";
			i++;
		}

		// Виконуємо умовно-корисну роботу
		waiting((double)proc->weight);
```

Продовження Додатка Б

Продовження Додатка А

```
		// Відсилаємо повідомлення
		i = proc->sendTo.begin();
		while (i != proc->sendTo.end()) {
			// Формуємо буфер для відправки
			sprintf(buffer, "#%d -> #%d", rank + 1, *i + 1);
			MPI_Comm c = get_comm_(*i, rank);
			int destRank = getLocalRank(*i);

			// Надсилаємо повідомлення
			MPI_Send(buffer, bufSize, MPI_CHAR, destRank, 0, c);
			printTime();
			fileStream << "Send: " << rank + 1 << " -> " << *i + 1 << "
(buf = '" << buffer << "')\n";
			i++;
		}
		printTime();
		fileStream << "time: " << MPI_Wtime() - t << " sec\n\n";
	}
};

int main(int argc, char *argv[]) {
	int	my_rank, numb_procs, namelen;
	MPI_Group world_gr, gr_0, gr_1;
	MPI_Comm comm_0, comm_1, remote_comm;


	std::ifstream inf("in.txt");
	if (!inf) {
		printTime();
		fileStream << "Can't open for reading log-file\n";
		return EXIT_FAILURE;
	}

	Proc proc;
	proc.read(inf);
	inf.close();

	// Ініціалізація
	if (MPI_Init(&argc, &argv) != MPI_SUCCESS) {
		printTime();
```

Продовження Додатка А

```
            fileStream << "Can't init MPICH\n";
            return EXIT_FAILURE;
    }

    // Розмір групи пов'язаної з комутатором (по суті, кількість процесів)
    MPI_Comm_size(MPI_COMM_WORLD, &numb_procs);
    if (numb_procs != 8) {
            printTime();
            fileStream << "The model can work only with 8 processes\n";
            MPI_Finalize();
            return 1;
    }

    // Ранг поточного процесу
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    std::stringstream ss;
    ss << "log" << (my_rank + 1) << ".txt";
    // Відкриваємо файл для перенаправлення потоку виведення
інформації
    fileStream.open(ss.str(), ios::app);

    MPI_Get_processor_name(hostname, &namelen);

    printTime();
    fileStream << "Process #" << my_rank + 1 << " running on [" <<
hostname << "]\n";

    // Створюємо загальну групу з існуючого комунікатора
    MPI_Comm_group(MPI_COMM_WORLD, &world_gr);

    // Створюємо 2 групи із заданою кількістю процесів
    MPI_Group_incl(world_gr, (int)proc.r0.size(), &proc.r0[0], &gr_0);
    MPI_Group_incl(world_gr, (int)proc.r1.size(), &proc.r1[0], &gr_1);

    // Створення інтракоммунікаторов на основі вже існуючих двох груп
процесів
    MPI_Comm_create(MPI_COMM_WORLD, gr_0, &comm_0);
    MPI_Comm_create(MPI_COMM_WORLD, gr_1, &comm_1);
```

Продовження Додатка А

```
	// Блокування поточного процесу, поки всі процеси не викличуть цю
функцію	MPI_Barrier(MPI_COMM_WORLD);

	// Визначаємо комунікатор для поточного процесу
	int my_leader, remote_leader;
	if (find(proc.r0.begin(), proc.r0.end(), my_rank) != proc.r0.end()) {
		my_comm = comm_0;
		remote_leader = proc.r1[0];
	}
	else {
		my_comm = comm_1;
		remote_leader = proc.r0[0];
	}

	remote_comm = MPI_COMM_WORLD;
	my_leader = 0;

	// створюємо інтеркоммунікатор
	MPI_Intercomm_create(my_comm, my_leader, remote_comm,
remote_leader, 2809, &inter_comm);

	MPI_Barrier(MPI_COMM_WORLD);

	// запуск процесу
	proc.run(my_rank);

	MPI_Barrier(MPI_COMM_WORLD);
	MPI_Comm_free(&inter_comm);
	MPI_Comm_free(&my_comm);
	MPI_Group_free(&gr_1);
	MPI_Group_free(&gr_0);
	MPI_Finalize();

	return 0;
}
```

Продовження Додатка А

3.4 Файл вхідних даних

```
# вузли для першої і другої групи процесів
r0      3 5 6 1
r1      2 7 8 4

# номер_вузла вага [>передача_вузлу] [<прийом_від_вузла]
1 1     >4 <3 <5
2 1     >5 >6 >7
3 2     >1 >6
4 5     <1 <6 <7 <8
5 1     >1 >8 <2
6 3     >4 <2 <3
7 4     >4 <2
8 2     >4 <5
```

3.5 Результати роботи програми

В програмі відбувається розпаралелювання 8 процесів на 2-х процесорах. Вхідні дані задані у файлі. У ньому зберігається інформація кому і від кого кожен процес отримує дані, їх ваги.

В програмі спочатку відбувається розподіл процесів на 2 групи, що будуть виконуватись на різних процесорах. В ході виконання програми відбувається обмін даними у відповідності до індивідуального завдання. Кожен процес спочатку отримує дані, що були йому надіслані, а потім пересилає їх іншому процесу. Результат записується у вихідний файл. Вміст вихідних файлів (для кожного процесу свій) зображено нижче:

```
[23:25:57:410] Process #1 running on [HornostlPC]
[23:25:59:516] Recv: 1 <- 3 (buf = '#3 -> #1')
[23:25:59:573] Recv: 1 <- 5 (buf = '#5 -> #1')
[23:26:00:707] Send: 1 -> 4 (buf = '#1 -> #4')
[23:26:00:707] time: 3.29184 sec

[23:25:57:413] Process #2 running on [HornostlPC]
[23:25:58:500] Send: 2 -> 5 (buf = '#2 -> #5')
```

Продовження Додатка А

[23:25:58:500] Send: 2 -> 6 (buf = '#2 -> #6')
[23:25:58:500] Send: 2 -> 7 (buf = '#2 -> #7')
[23:25:58:500] time: 1.08517 sec

[23:25:57:412] Process #3 running on [HornostlPC]
[23:25:59:516] Send: 3 -> 1 (buf = '#3 -> #1')
[23:25:59:541] Send: 3 -> 6 (buf = '#3 -> #6')
[23:25:59:541] time: 2.12639 sec

[23:25:57:412] Process #4 running on [HornostlPC]
[23:26:00:885] Recv: 4 <- 1 (buf = '#1 -> #4')
[23:26:02:545] Recv: 4 <- 6 (buf = '#6 -> #4')
[23:26:02:548] Recv: 4 <- 7 (buf = '#7 -> #4')
[23:26:02:548] Recv: 4 <- 8 (buf = '#8 -> #4')
[23:26:07:549] time: 10.1344 sec

[23:25:57:408] Process #5 running on [HornostlPC]
[23:25:58:572] Recv: 5 <- 2 (buf = '#2 -> #5')
[23:25:59:573] Send: 5 -> 1 (buf = '#5 -> #1')
[23:25:59:739] Send: 5 -> 8 (buf = '#5 -> #8')
[23:25:59:739] time: 2.32406 sec

[23:25:57:411] Process #6 running on [HornostlPC]
[23:25:58:500] Recv: 6 <- 2 (buf = '#2 -> #6')
[23:25:59:516] Recv: 6 <- 3 (buf = '#3 -> #6')
[23:26:02:530] Send: 6 -> 4 (buf = '#6 -> #4')
[23:26:02:530] time: 5.11538 sec

[23:25:57:410] Process #7 running on [HornostlPC]
[23:25:58:500] Recv: 7 <- 2 (buf = '#2 -> #7')
[23:26:02:531] Send: 7 -> 4 (buf = '#7 -> #4')
[23:26:02:531] time: 5.11656 sec

[23:25:57:409] Process #8 running on [HornostlPC]
[23:25:59:868] Recv: 8 <- 5 (buf = '#5 -> #8')

Продовження Додатка А

[23:26:01:868] Send: 8 -> 4 (buf = '#8 -> #4')
[23:26:01:868] time: 4.45313 sec

Зверху зображено логування виконання восьми процесів, що виконуються на одному ПК з іменем [HornostlPC] у системі MS MPI. Також зазначено час виконання кожного процесу. Час виконання програми дещо (на 0.1344 секунди) перевищує критичний час (10 секунд).

Продовження Додатка А

3 ВИСНОВКИ

У ході виконання розрахунково-графічного завдання було отримано результати, представлені на рис. 11 та у табл. 2.

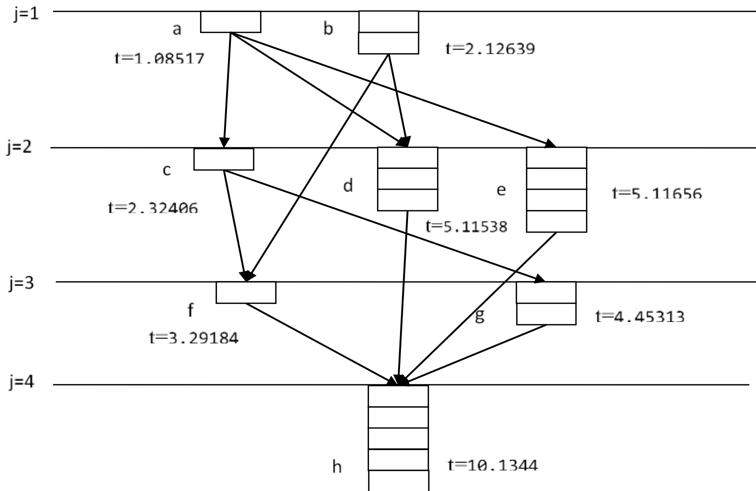

Рисунок 11 – Часова діаграма виконання процесів

Таблиця .2 – Порівняння результатів з аналітичними розрахунками ранніх та пізніх термінів закінчення процесів

Початкове ім'я	Результат	$\tau_1(k)$	$\tau_2(k)$
a	1.08517	1	3
b	2.12639	2	4
c	2.32406	2	5
d	5.11538	5	7
e	5.11656	5	7
f	3.29184	3	7
g	4.45313	4	7
h	10.1344	10	10+2 = 12

Продовження Додатка А

Планувалося тестування програми на двох процесорах (ядрах) за допомогою системи MS MPI, проте в процесі перевірки виявилися обмеження: система завжди запускає програми на максимальній кількості процесорів (ядер), якщо кількість процесів перевищує задану. У даному випадку кількість процесів – 8, тому програма починає виконуватися на 3 або 4х ядрах. Завдяки цьому час виконання необхідно звіряти за ранніми термінами закінчення. Виходячи з цього можна зробити висновок, що програма працює коректно, проте існують деякі розбіжності за часом, що зумовлені від 0,08517 до 0, 45313 секунд. Загальний час роботи відрізняється від раннього терміну закінчення на 0,1344 секунд. Це зумовлено тим, що виконання на одному процесорі з декількома ядрами (як фізичними, так і віртуальними) передбачає можливість затримки виконання команд. Це пов’язано і з безпосереднім спілкуванням між ядрами, і з тим, що процесор може бути зайнятим у певний момент часу.

Навчальне видання

МЕТОДИЧНІ ВКАЗІВКИ
до виконання та оформлення розрахунково-графічних завдань з дисципліни «Паралельні та розподілені обчислення» для студентів спеціальності 123 «Комп'ютерна інженерія» і 125 «Кібербезпека»

Укладачі:
ЧЕРНИХ Олена Петрівна
КАЛАШНІКОВ Володимир Іванович
БУЛЬБА Сергій Сергійович
ГОРНОСТАЛЬ Олексій Андрійович

Відповідальний за випуск проф. С. Г. Семенов

Роботу до видання рекомендував В.Д. Дмитрієнко

В авторській редакції

План 2020 р., поз. 263
Підписано до друку 27.01.21. Формат 60x84 1/16.
Папір офсетний. Друк ризографія. Ум. друк. арк. 3,5.
Наклад 150 прим. Замовлення № 72.

Видавець:
Видавничий центр НТУ "ХПІ".
Свідоцтво про державну реєстрацію ДК №116 від 10.07.2000 р.
61002, Харків, вул. Фрунзе, 21.

Надруковано з готового оригінал-макету у друкарні ФОП ФОП В.В. Петров.
Єдиний державний реєстр юридичних осіб та фізичних осіб-підприємців.
Запис №24800000000106167 від 08.01.2009 р.
61144, м. Харків, вул. Гв. Широнінців, 79в, к. 137,
тел. (057) 78–17–137. e-mail: bookfabrik@mail.ua