Лабораторні роботи

Паралельні та розподілені обчислення

Частина 1
Технологія OpenMP

Укладачі І. В. Лупан, С. Д. Паращук

Кропивницький
2024

Лабораторні роботи

Паралельні та розподілені обчислення

Частина 1
Технологія OpenMP

Укладачі І. В. Лупан, С. Д. Паращук

Кропивницький
2024

УДК 004.42 (076)
П 18

Паралельні та розподілені обчислення: лабораторні роботи. Частина 1: Технологія OpenMP / Укладачі І. В. Лупан, С. Д. Паращук. Кропивницький: ФОП Піскова М. А., 2024. 58 с.

У посібнику наведено завдання та методичні рекомендації до виконання лабораторних робіт з курсу «Паралельні та розподілені обчислення», метою яких є ознайомлення студентів з основами технології OpenMP, призначеної для паралельного програмування в середовищі зі спільною пам'яттю.

Методичні рекомендації призначені для здобувачів першого (бакалаврського) ступеня, що навчаються за спеціальністю 122 Комп'ютерні науки та 126 Інформаційні системи та технології.

Рецензенти:

Баранюк О.Ф. *– кандидат технічних наук, доцент кафедри інформатики, програмування, штучного інтелекту та технологічної освіти Центральноукраїнського державного університету імені Володимира Винниченка;*

Ізвалов О.В. *– кандидат технічних наук, доцент кафедри інформаційних технологій Економіко-технологічного інституту ім. Роберта Ельворті.*

Рекомендовано методичною радою Центральноукраїнського державного університету імені Володимира Винниченка (протокол № 1 від 28.08.2024 р.)

© І. В. Лупан, С. Д. Паращук, 2024

Зміст

Практична робота №1: Розпаралелювання арифметичних виразів 4
Теоретичні відомості 4
Порядок виконання роботи 8
Варіанти арифметичних виразів для виконання завдання: 8
Контрольні питання 9
Рекомендовані джерела 9
Лабораторна робота №2. Використання OpenMP 11
Теоретичні відомості 11
Завдання 18
Контрольні питання 18
Рекомендовані джерела 19
Лабораторна робота №3. Директива for 20
Теоретичні відомості 20
Завдання 28
Контрольні питання 29
Рекомендовані джерела 29
Лабораторна робота №4. Паралельне обчислення числа Пі. 30
Завдання 1 30
Завдання 2 30
Контрольні питання 31
Рекомендовані джерела 32
Завдання для індивідуального виконання 32
Лабораторна робота №5: Алгоритми планування паралельного виконання циклів в технології OpenMP 33
Теоретичні відомості 33
Завдання до виконання 38
Варіанти для індивідуального виконання 38
Контрольні запитання 40
Лабораторна робота №6. Множення матриці на вектор 42
Лабораторна робота №7. Множення матриць 44
Теоретичні відомості 44
Завдання 45
Рекомендовані джерела 52
Індивідуальне завдання №1 53
Варіанти завдань 53
Індивідуальне завдання №2. Множення матриць та векторів 56
Рекомендовані джерела 57

ПРАКТИЧНА РОБОТА №1: РОЗПАРАЛЕЛЮВАННЯ АРИФМЕТИЧНИХ ВИРАЗІВ

Теоретичні відомості

1. Основні підходи до розпаралелювання

Нехай E – простий арифметичний вираз, що задовольняє умові:

"*Кожна змінна входить в E рівно один раз*". (*)

Виконання цієї умови завжди можна досягти перейменуванням змінних.

Мета будь-якого алгоритму розпаралелювання – мінімізувати час, необхідний для паралельного обчислення E.

Найбільш відомі алгоритми розпаралелювання арифметичних виразів Баєра-Бовета, Брента і Винограда засновані на загальному принципі: *орієнтований ациклічний граф, що описує послідовне обчислення виразу E, представляє собою, з урахуванням властивості* (*), *бінарне дерево*.

До цього дерева застосовують перетворення у відповідності до законів комутативності, асоціативності і дистрибутивності. В результаті бінарне дерево трансформується в орієнтований ациклічний граф $\tilde{G}$, що відповідає виразу $\tilde{E}$, еквівалентному до E.

Отже, ***основна мета*** розпаралелювання арифметичних виразів – розробити дерево обчислень мінімальної висоти. Тобто, за умови реалізації на кожному ярусі максимальної кількості незалежних операцій, вираз має бути обчислений за мінімально можливий час або за мінімальну кількість кроків. Розпаралелювання АВ здійснюється в рамках моделі MIMD (Multiple Instruction Multiple Date – множинний потік команд і множинний потік даних) з необмеженим паралелізмом.

Порівняно з операторними схемами вирази мають принципову відмінність – в них не завжди вказано порядок застосування операцій. У тому випадку, коли у виразі є дужки, з'являється деяка свобода у застосуванні операцій. Навіть якщо опускати деякі дужки, то дерево виразу також будується неоднозначно. Наприклад, ***a*b*c*** може бути "відновлено" як ***a*(b*c)*** або ***(a*b)*c***.

Порядок обчислення може бути перерозподілений або довизначений таким чином, щоб деякі операції можна було виконувати паралельно. Тому говорять, що вирази мають *неявний паралелізм*.

Задача розпаралелювання виразів полягає у побудові такого алгоритма, який за кожним виразом E дає еквівалентний йому $\tilde{E}$ з мінімальною висотою дерева обчислень.

Два вирази E та $\tilde{E}$ називають *еквівалентними* ($E \sim \tilde{E}$) в тому і тільки в тому випадку, якщо вираз E перетворюється в $\tilde{E}$ і навпаки, шляхом використання скінчену кількість разів законів асоціативності, комутативності і дистрибутивності.

Розглянемо процес обчислення арифметичного виразу

$$E = (x + (a * ((b / c) * d))) - (y - z)$$

та еквівалентного йому:

$$\tilde{E} = (a * b) / (c / d) - ((y - z) - x).$$

На рис.1 і рис.2 наведено дерева обчислень для двох еквівалентних арифметичних виразів E та $\tilde{E}$.

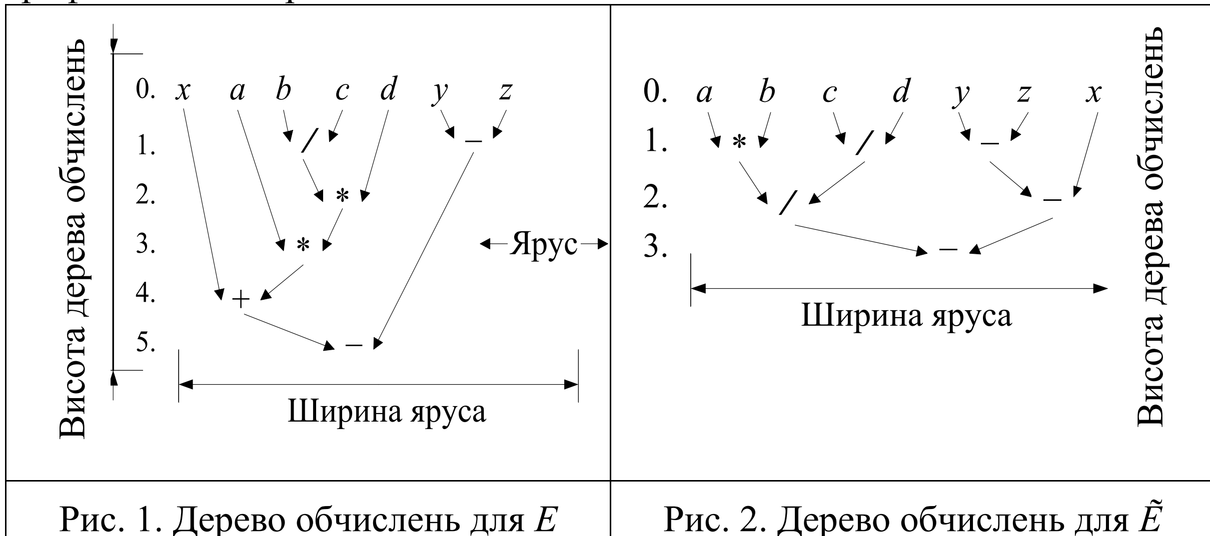


Рис. 1. Дерево обчислень для E	Рис. 2. Дерево обчислень для $\tilde{E}$

2. Характеристики складності і паралельності

Характеристиками складності обчислення арифметичного виразу є:

- час $\boldsymbol{t}$, що витрачається на обчислення арифметичного виразу;
- загальна кількість операцій $\boldsymbol{w}$, необхідна для обчислення виразу;
- кількість обчислювачів (або процесорів) $\boldsymbol{p}$, потрібна для реалізації обчислень.

Якщо вважати, що будь-яка операція виконується за одну одиницю часу, то час $\boldsymbol{t}$ обчислення арифметичного виразу дорівнює числу ярусів (висоті) дерева обчислень. Операції, що знаходяться на одному ярусі дерева, можуть виконуватися паралельно та визначають ширину даного яруса. Тому висота дерева відповідає числу кроків паралельного алгоритму для арифметичного виразу.

Потрібна кількість обчислювачів (або процесорів) $\boldsymbol{p}$ визначається як максимальна ширина яруса дерева обчислень.

Для виразів E та $\tilde{E}$ маємо такі характеристики складності обчислення:

Для виразу E	Для виразу $\tilde{E}$
$t = 5$	$t' = 3$
$p = 2$	$p' = 3$
$w = 6$	$w' = 6$

Варто відзначити, що застосування законів дистрибутивності при перетвореннях арифметичних виразів в еквівалентні може призвести до переповнення розрядної сітки. Наприклад, якщо у виразі $a*(b–c)$, a, b та c – великі додатні числа, то множення $a*(b–c)$ відбуватиметься без колізій, оскільки величини b та c "компенсують" одна одну, в той час як при обчисленні $a*b - a*c$ може виникнути переповнення при множеннях. Втім,

подібне може статися і при використанні інших законів, наприклад, асоціативності.

Наприклад, $(a*b)*c$ еквівалентне до $a*(b*c)$, але при великих a та b і $c<<1$ при цьому порядку множення може виникнути переповнення, якого б не було при множенні $a*(b*c)$. Навіть при перестановці аргументів ($a*b = b*a$) в деяких перемножувачах може виникнути переповнення або втрата точності. Тому в багатьох випадках процедура розпаралелювання поширюється лише на бездужкові фрагменти виразу, тобто на фрагменти, в яких черговість визначається тільки пріоритетністю операцій і ніяк не обмежена програмістом.

Для оцінки розпаралеленого виразу використовують такі характеристики паралельності, як ***прискорення*** та ***ефективність*** паралельних алгоритмів.

Нехай n – число параметрів задачі (для арифметичного виразу – число різних змінних, що входять в арифметичний вираз).

$T_p(n)$ – час виконання паралельного алгоритму на обчислювальній системі з числом процесорів p>1.

$T_1(n)$ – час виконання “найкращого” послідовного алгоритму.

Прискоренням $\boldsymbol{S_p}$ паралельного алгоритму називають відношення

$$S_p(n) = \frac{T_1(n)}{T_p(n)},$$

а ***ефективність*** $\boldsymbol{E_p}$ паралельного алгоритму визначається формулою

$$E_p(n) = \frac{S_p(n)}{p}.$$

Очевидно, чим ближче значення $\boldsymbol{S_p}(n)$ до p, а $\boldsymbol{E_p}(n)$ – до 1, тим “кращим” є побудований паралельний алгоритм.

Маємо такі характеристики паралельних обчислень:

Для виразу E	Для виразу $\tilde{E}$
$T_1(n) = n - 1 = 6$	$T_1(n) = n - 1 = 6$
$T_p(n) = t = 5$	$T_p(n) = t' = 3$
$\boldsymbol{S_p}(n) = 6/5 = 1.2$	$S'_p(n) = 6/3 = 2$
$E_p(n) = 1.2/2 = 0.6$	$E'_p(n) = 2/3 = 0.66$

Таким чином, вираз $\tilde{E}$ має кращі характеристики паралельності, ніж вираз E.

Проаналізуємо дерево обчислень на рис. 2 з погляду оптимізації завантаження процесорів і найкращої ефективності. Спробуємо зменшити число процесорів, але завантажимо їх більш рівномірно (див. рис. 3).

$T_1(n) = w = 6$ $S_p(n) = 6/4 = 1.5$

$T_p(n) = t = 4$ $\quad$ $E_p(n) = 1.5/2 = 0.75$

В результаті оптимізації завантаження процесорів отримано менше прискорення порівняно з деревом на рис.2, але більш високу ефективність.

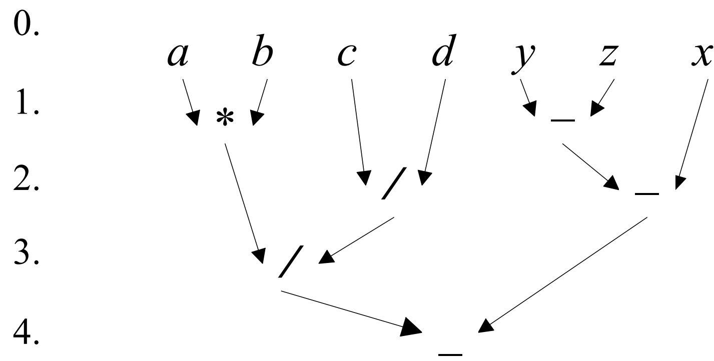


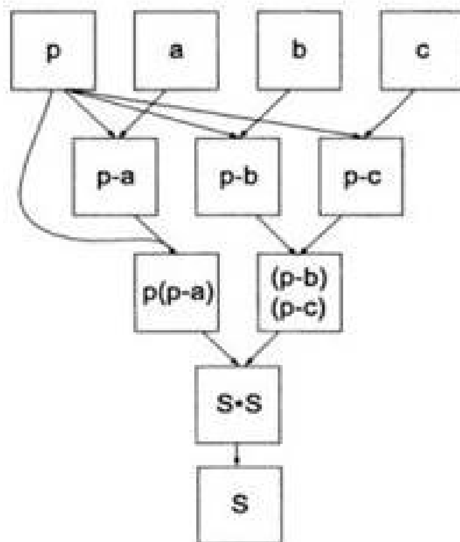
Рис. 3. Дерево обчислень для оптимізації завантаження процесорів

Розглянемо ще дві корисні оцінки для паралельних схем обчислень: ***ціну*** та ***цінність*** паралельного рішення.

Ціна паралельного рішення: $C_p = p*T_p(n)$.

Цінність паралельного рішення: $F_p = S_p/C_p = T_1(n)/(p*T_p^2(n))$.

Ще однією характеристикою алгоритму є *степінь паралелізму* – кількість операцій, які в принципі можна виконати паралельно. *Середній степінь паралелізму* визначається як відношення загальної кількості операцій, які потрібно виконати, до кількості паралельних кроків, за які ці операції виконуються, але на відміну від прискорення, ця характеристика ніяк не залежить від кількості ядер процесора і характеризує тільки алгоритм.



Для прикладу обчислення площі трикутника за формулою Герона загальна кількість операцій дорівнює 7, а кількість паралельних кроків — 4. Відповідно середній степінь паралелізму дорівнює 1.75.

Міркування з приводу часу виконання паралельного алгоритму при меншій кількості обчислювачів, ніж потрібно, сформульоване у вигляді леми Брента.

Лема Брента. Якщо при необмеженій кількості процесорів для обчислення виразу АВ, що містить ***w*** операцій, потрібен час ***t***, то при

наявності обмеженої кількості процесорів $\boldsymbol{p'}$ обчислення АВ може бути виконано не більш ніж за час $\boldsymbol{t'}$, що визначається формулою:

$$\boldsymbol{t' = t + (w - t) / p'.}$$

Дане твердження дозволяє за характеристиками АВ при необмеженій кількості обчислювачів оцінити час реалізації виразу на обмеженій кількості обчислювачів.

Перевіримо лему для дерева, отриманого на рис.3:

- число операцій $w = 6$,
- час виконання на необмеженій кількості процесорів дорівнює $t = 3$ (див. рис.2),
- при наявності $p' = 2$ час виконання, у відповідності з лемою, становитиме не більше $t' = 3 + (6 - 3)/2 = 4.5$.

Реально, виходячи з рис.3, $t = 4$, отже $t < t'$, тобто лема виконується.

Порядок виконання роботи

Ознайомитися з теоретичним матеріалом до роботи.

Для виразу зі свого варіанта

1) побудувати дерево паралельного обчислення арифметичного виразу мінімальної висоти (***перша схема***);

2) визначити характеристики складності та паралельності (*степінь паралелізма*, **прискорення $\boldsymbol{S}$ та ефективність $\boldsymbol{E}$, ціну $\boldsymbol{C_p}$ та цінність $\boldsymbol{F_p}$**) для побудованої паралельної схеми;

3) проаналізувати побудовану паралельну схему з погляду оптимізації завантаження процесорів та найкращої ефективності, якщо необхідно, то перебудувати дерево обчислень (***друга схема***);

4) визначити **прискорення та ефективність** паралельної схеми для випадка оптимального завантаження процесорів. Порівняти ці характеристики зі значеннями, отриманими в п.2;

5) графічно дослідити **залежності характеристик прискорення та ефективності від кількості процесорів**. Визначити **число процесорів**, що відповідає найкращому розпаралелюванню виразу як з погляду прискорення, так і ефективності;

6) перевірити лему Брента на застосовність до даного виразу, дослідивши декілька паралельних схем обчислень з різною кількістю процесорів (***третя схема***);

7) оформити результати виконання всіх пунктів у звіт по роботі;

8) при подготовці до захиту роботи відповісти на контрольні питання.

Варіанти арифметичних виразів для виконання завдання:

1. (r/t/y/u + i∗o + p – a + s)+(d – f)∗(g + h)∗(j + k);

2. (a∗(s – d) + f + g + h + j + k)∗(l + z – x/(c + v) + b∗n);
3. (z + x + c + v + b)/(n + m∗q + w/e + r/t∗y) – u + i;
4. (s – d)/(f + g)∗(h – j)∗(k + l) + (z + x + c + v)∗(b + n + m);
5. (x∗c∗v∗b + n/m/a/s + d)/(f∗(g + h) + (j – k)/ l);
6. (c + v + b + n/a)∗(s – d – f + r + t∗y + u∗k + p + z);
7. v – b – n + m∗a∗s∗d + (f∗g + h/j) + (k/l + q∗w);
8. (b + n + m + q)/(w + e – r∗t + y)∗(u + a – s – d) + f + g;
9. (s – d)∗(f + g)∗(h – j)/(k – l) + (z/x/c/v/b + n∗m);
10. (r∗t∗y∗u + i/o + p – a – s)+(d – f)∗(g – h + j + k);
11. (d – f∗g + h)∗(j + k)/(r + t/y∗u + i∗o + p – a/s);
12. (a/(s – d) + f – g∗h + j∗k)∗(l∗z + b/n – x/(c – v));
13. (m + n)∗(b – v) + c + x + z/(l + k) + j/(h – g) + f∗(d + s);
14. (l – k – j – h∗g + f/d/s)/(a∗p∗o + i + u – y + t);
15. (p – o)∗(i + u) + (y + t – r + e∗(w – q) + l)/(k/j + m∗n);
16. (m + n∗b/v) + c∗x + z/(l + k) + j/(h – g) + f∗(d – s);
17. (l – k – j – h + g∗f∗d/s)/(a∗p∗o + i∗(u – y + t));
18. e + r – t/y∗(u∗i + o/p) + a∗s + d∗(f – g)∗(h + j);
19. (u∗y∗t + r – e – w)∗(q∗l + k/j∗h + g – f + d – s);
20. (h – g – f + d)∗s + (a + q – w/e + r/t + y∗z)∗(x + c);
21. (u∗y/t + r – e∗w)∗(q∗l – k/j/h + g) – (f + d)∗s;
22. (f + d)/(s – a) + (e – r)∗(t + y)∗(u + p – m/n) + (b – v)∗c;
23. a+b+c+d*q*w*e–t/r+u*i*o*p/h/j+k+l/n/m;
24. (l – k –(j – h)∗g + f/d*s)*(a/p/o – i + u + y – t);
25. (s + d)/(f – g)/(h – j)+(k + l) + (z – x + c + v)∗(b + n)* m.

Контрольні питання

1. Який принцип розпаралелювання використовують при розпаралелюванні арифметичних виразів?
2. Чому арифметичні вирази відносять до класу задач з *неявним паралелізмом?*
3. В чому полягає основний сенс леми Брента?
4. Які характеристики паралельності Ви знаєте?
5. Чим характеризується ефективність схеми при оптимальному завантаженні процесорів?

Рекомендовані джерела

1. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. 188 с. – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf

2. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти / К.Т. Кузьма, О.В. Мельник. – Миколаїв: ФОП Швець В.М., 2020. – 172 с.
3. J.L. Baer and D.P. Bovet. *Compilation of arithmetic expressions for parallel computations*. In Proc. IFIP Congress, pp 340-346, Amsterdam, 1968. (*"Компіляція арифметичних виразів для паралельних обчислень").*
4. R. P. Brent. *The parallel evaluation of arithmetic expressions in logarithmic time In Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub, Ed, Academic Press, New York, pp. 83-102, 1973.
5. Richard P. Brent. *The Parallel Evaluation of General Arithmetic Expressions*. Journal of the ACM, v.21 n.2, pp. 201-206, 1974.
6. S. Winograd. *On the parallel evaluation of certain arithmetic expressions*. Journal of the ACM, v.22 n.4, pp. 477-492, 1975.

ЛАБОРАТОРНА РОБОТА №2. ВИКОРИСТАННЯ OPENMP

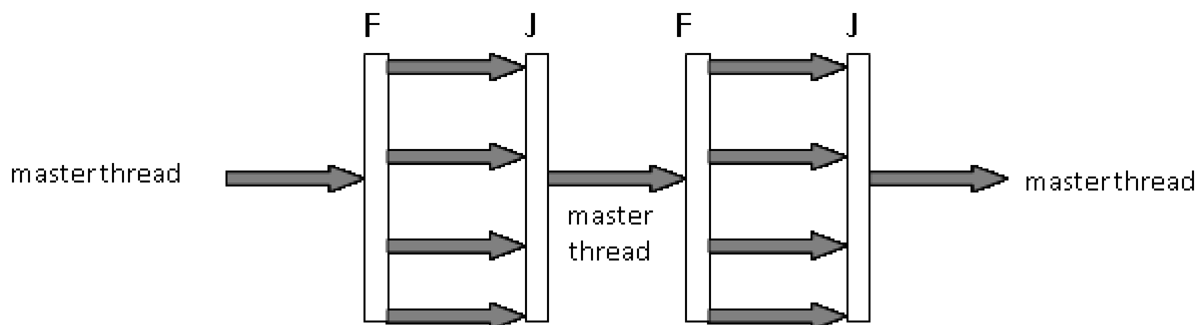
Теоретичні відомості

1. Методи розпаралелювання і моделі програм, які підтримує OpenMP

Базовий потоковий паралелізм: загальнодоступний процес пам'яті може складатися із декількох потоків, декількох «ниток» (threads) управління, що мають спільний адресний простір, але різні потоки команд і роздільні стеки. У найпростішому випадку процес складається з однієї нитки. Нитки також інколи називають потоками або легкими процесами LWP (light-weight processes). OpenMP заснований на існуванні множинних потоків у спільновикористовуваній (загальнодоступній) пам'яті, але пропонує програмісту засоби управління розпаралелюванням, тобто ***явний паралелізм***.

OpenMP використовує ***Fork-Join*** модель паралельного виконання: всі програми OpenMP починаються як один процес – головний потік, який виконується послідовно, поки не стикнеться з першою областю паралельної конструкції. Тоді відбувається процес розгалуження – ***Fork***, – створення групи паралельних потоків.

Коли потоки завершать виконання своїх задач, вони синхронізуються і закриваються. Залишається тільки головний. Таким чином здійснюється ***Join*** – об'єднання:



2. Система OpenMP

OpenMP (Open specifications for Multi-Processing) – це набір специфікацій для паралелізації програм у середовищі із загальною пам'яттю. Інтерфейс OpenMP задуманий як стандарт для програмування на масштабованих SMP-системах (SSMP, ccNUMA) в моделі загальної пам'яті (shared memory model). У стандарт OpenMP входять специфікації набору директив компілятора, процедур і змінних середовища.

OpenMP дозволяє легко і швидко створювати багатопоточні додатки на алгоритмічних мовах Fortran і C/C++. При цьому директиви OpenMP аналогічні до директив препроцесора для мови C/C++ і є аналогом коментарів в алгоритмічній мові Fortran. Це дозволяє в будь-який момент розробки паралельної реалізації програмного продукту при необхідності повернутися до послідовного варіанта програми.

3. Спільна пам'ять

OpenMP − це стандартна модель для паралельного програмування в середовищі зі спільною пам'яттю. У цій моделі всі процеси спільно використовують загальний адресний простір, до якого вони асинхронно звертаються із запитами на читання і запис. У таких моделях для управління доступом до загальної пам'яті використовують різні механізми синхронізації типу семафорів і блокування процесів. Перевага цієї моделі з точки зору програмування полягає в тому, що поняття монопольного володіння даними відсутнє, отже, не потрібно явно задавати обмін даними між потоками, що їх задають, та потоками, що їх використовують. Ця модель, з одного боку, спрощує розробку програми, але, з іншого боку, ускладнює розуміння та управління локальністю даних, написання детермінованих програм. В основному вона використовується при програмуванні для архітектур зі спільною пам'яттю.

Прикладами систем зі спільною пам'яттю, які мають велике число процесорів, можуть слугувати суперкомп'ютери Cray Origin2000 (до 128 процесорів), HP 9000 V-class (до 32 процесорів в одному вузлі, а в конфігурації з чотирьох вузлів − до 128 процесорів), Sun Starfire (до 64 процесорів).

4. Стандарт OpenMP

Розробкою стандарту займається організація OpenMP ARB (ARchitecrure Board), до якої увійшли представники найбільших компаній − розробників SMP-архітектур і програмного забезпечення. Перша версія специфікації OpenMP (www.openmp.org) з'явилася в 1997 р. і була призначена для мови програмування Fortran. Біля витоків OpenMP стояли такі відомі компанії, як IBM, Intel, Sun і Hewlett-Packard. У 1998 р. з'явилися варіанти OpenMP для мов C/C++, і на цей момент останньою є версія 5.2.

На сьогодні OpenMP підтримується більшістю розробників паралельних обчислювальних систем: компаніями Intel, Hewlett-Packard, Silicon Graphics, Sun, IBM, Fujitsu, Hitachi, Siemens, Bull і іншими. Багато відомих компаній в галузі розробки системного програмного забезпечення також приділяють значну увагу розробці системного програмного забезпечення з OpenMP, зокрема Intel, KAI, PGI, PSR, APR, Absoft і деякі інші. Значне число компаній і науково-дослідних організацій, які розробляють прикладне програмне забезпечення, сьогодні використовує OpenMP при розробці своїх програмних продуктів. Серед цих компаній і організацій відзначимо ANSYS, Fluent, Oxford Molecular, NAG, DOE ASCI, Dash, Livermore Software.

Компілятори gcc і gfortran мають вбудовану підтримку паралелізації OpenMP, починаючи з версії 4.2. Для використання цієї можливості слід додати ключ -fopenmp при компіляції. OpenMP версії 3.0 підтримується, починаючи з версії gcc 4.4.

OpenMP 2.5 підтримується також компілятором Microsoft Visual C++ 2005 (слід використовувати ключ − /openmp) і компіляторами Intel C або Intel Fortran починаючи з версії 10.1 (ключ -openmp).

5. Основи OpenMP

Будь-яка програма, послідовна або паралельна, складається з набору областей двох типів: послідовних областей і областей розпаралелювання. При виконанні послідовних областей породжується тільки один головний потік виконання (процес). У цьому потоці ініціюється виконання програми, а також відбувається її завершення. У послідовної програми цей потік є єдиним протягом виконання всієї програми. У паралельній програмі в областях розпаралелювання породжується ціла низка паралельних потоків. Породжені паралельні потоки можуть виконуватися як на різних процесорах, так і на одному процесорі обчислювальної системи. В останньому випадку паралельні процеси (потоки) конкурують між собою за доступ до процесора Управління конкуренцією здійснюється планувальником операційної системи за допомогою спеціальних алгоритмів. В операційній системі Linux планувальник завдань здійснює обробку процесів за допомогою стандартного карусельного (round-robin) алгоритму. При цьому тільки адміністратори системи мають можливість змінити або замінити цей алгоритм системними засобами. Таким чином, у паралельних програмах в областях розпаралелювання виконується низка паралельних потоків. Принципова схема паралельної програми зображена на рис. 1:

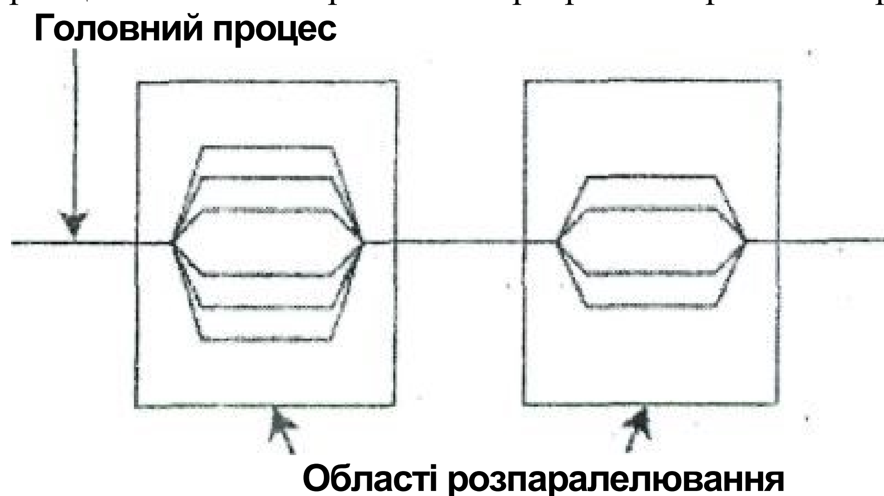


Рис. 1. Принципова схема паралельної програми

При виконанні паралельної програми робота починається з ініціалізації та виконання головного потоку (процесу), який у міру необхідності створює і виконує паралельні потоки виконання, передаючи їм необхідні дані. Паралельні потоки з однієї паралельної області програми можуть виконуватися як незалежно один від одного, так і з пересиланням та отриманням повідомлень від інших паралельних потоків. Остання обставина ускладнює розробку програми, оскільки в цьому випадку програмісту доводиться займатися плануванням, організацією і синхронізацією посилки повідомлень між паралельними потоками. Таким чином, при розробці паралельної програми бажано виділяти такі області розпаралелювання, в яких можна організувати виконання незалежних паралельних потоків. Для обміну даними між паралельними процесами (потоками) в OpenMP використовуються

загальні змінні. При зверненні до загальних змінних у різних паралельних потоках можливе виникнення конфліктних ситуацій при доступі до даних. Для запобігання конфліктам можна скористатися процедурою синхронізації. При цьому треба мати на увазі, що процедура синхронізації − дуже дорога операція за тимчасовими витратами і бажано по можливості уникати її або застосовувати якомога рідше. Для цього необхідно дуже ретельно продумувати структуру даних програми.

Виконання паралельних потоків в паралельній області програми починається з їхньої ініціалізації. Вона полягає у створенні дескрипторів породжуваних потоків і копіюванні всіх даних з області даних головного потоку в області даних створюваних паралельних потоків. Ця операція надзвичайно трудомістка − вона еквівалентна приблизно трудомісткості не менше 1000 машинних команд. Ця оцінка надзвичайно важлива при розробці паралельних програм з допомогою OpenMP, оскільки її ігнорування веде до створення неефективних паралельних програм, які часто виявляються повільнішими ніж їхні послідовні аналоги. Справді: для того щоб отримати виграш у швидкодії паралельної програми, необхідно, щоб трудомісткість паралельних процесів в областях розпаралелювання програми істотно перевершувала б трудомісткість породження паралельних потоків. В іншому випадку ніякого виграшу за швидкодією отримати не вдасться, а часто можна опинитися навіть і в програші.

Після завершення виконання паралельних потоків управління програмою знову передається головному потоку. При цьому виникає проблема коректної передачі даних від паралельних потоків до головного. Тут важливу роль відіграє синхронізація завершення роботи паралельних потоків, оскільки в силу цілої низки обставин час виконання навіть однакових за трудомісткістю паралельних потоків непередбачуваний (він визначається як історією конкуренції паралельних процесів, так і поточним станом обчислювальної системи). При виконанні операції синхронізації паралельні потоки, вже завершили своє виконання, простоюють і чекають завершення роботи останнього потоку. Природно, при цьому неминуча втрата ефективності роботи паралельної програми. Крім того, операція синхронізації має трудомісткість, порівнянну з трудомісткістю ініціалізації паралельних потоків.

6. Переваги, які OpenMP дає розробнику:

- завдяки ідеї *«інкрементального розпаралелювання»* OpenMP ідеально підходить для розробників, які хочуть швидко розпаралелити свої обчислювальні програми з великими циклами. Розробник не створює нову паралельну програму, а просто додає в текст послідовної програми OpenMP-директиви;
- при цьому, OpenMP − досить *гнучкий механізм,* що надає розробнику великі можливості контролю над поведінкою паралельної програми;
- передбачається, що OpenMP-програма на одноцроцесорній платформі може бути використана як *послідовна* програма, тобто немає необхідності

підтримувати послідовну і паралельну версії. Директиви OpenMP просто ігноруються послідовним компілятором, а для виклику процедур OpenMP можуть бути підставлені заглушки (stubs), текст яких наведений у специфікаціях.

7. Порядок створення паралельних програм

1. Написати та налагодити послідовну програму.
2. Доповнити програму директивами OpenMP.
3. Скомпілювати програму компілятором з підтримкою OpenMP.
4. Задати змінні оточення.
5. Запустити програму.

Приклад 1: Оголошення паралельної секції

```
#include <omp.h>
int main()
{
  // послідовний код

  #pragma omp parallel [опції]
  {

    // паралельний код

  }
  // послідовний код
  return 0;
}
```

Приклад 2: Hello, World!

```
#include <stdio.h>
#include <omp.h>

int main()
{
  printf("Hello, World!\n");
  #pragma omp parallel
  { int i,n;
    i = omp_get_thread_num();
    n = omp_get_num_threads();
    printf("I'm thread %d of %d\n", i, n);
  }
  return 0;
}
```

8. Задання кількості потоків

- **Змінна оточення OMP_NUM_THREADS**

```
OMP_NUM_THREADS=4
```

- **Функція omp_set_num_threads(int)**

```
omp_set_num_threads(4);
#pragma omp parallel
{ . . .
}
```

- **Параметр num_threads**

```
#pragma omp parallel num_threads(4)
    { . . .
    }
```

9. Опції для прагми parallel:

- ***if*** (умова) – виконання паралельної області по умові. Входження в паралельну область здійснюється лише при виконанні деякої умови.

Якщо умову не виконано, то директива не спрацьовує і обробка програми продовжується у звичайному режимі;

- ***num_threads*** (цілочисельний вираз) – явне задання кількості потоків (threads), які будуть виконувати паралельну область; за замовченням обирається останнє значення, встановлене функцією omp_set_num_threads(), або значення змінної OMP_NUM_THREADS;
- ***default*** (private|firstprivate|shared|none) – усім змінним в паралельній області, яким явно не призначено клас доступу, буде призачено класс private, firstprivate або shared відповідно; none означає, що усім змінним в паралельній області клас має бути призначений явно; в мові С задаються тільки варіанти shared або none;
- ***private*** (список) – задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
- ***firstprivate*** (список) – задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в головному потоці (master thread);
- ***shared*** (список) – задає список змінних, спільних для всіх потоків;
- ***copyin*** (список) – задає список змінних, оголошених як threadprivate, які при вході в паралельну область ініціалізуються значеннями відповідних змінних в головному процесі;
- ***reduction*** (оператор:список) – задає оператор і список спільних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно до типу оператора (для адитивних операцій – 0 або його аналоги, для мультиплікативних операцій – 1 або її аналоги); над локальними копіями змінних після виконання усіх операторів паралельної області виконується заданий оператор; для мови Сі операторами є +, *, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску.

Приклад 3. Кожен процес виводить на екран свій ідентифікаційний номер та кількість замовлених паралельних процесів:

```
#include<omp.h>
#include<stdio.h>

main ()
{
int size, rank;

/* створення множини паралельних процесів, у кажному з них задані */
/* свої приватні змінні size та rank */
omp_set_dynamic(0);
```

```
#pragma omp parallel private(size, rank)
  {
/* Кожен процес знаходить свій порядковий номер та виводить його на
екран */
  rank = omp_get_thread_num();
  printf("Hello World from thread = %d\n", rank);

/* Головний процес - master - виводить на екран кількість процесів */
  if (rank == 0)
    {
    size = omp_get_num_threads();
    printf("Number of threads = %d\n", size);
    }

  }  /* Завершення паралельної частини */
}
```

Розглянемо детальніше типи видимості даних shared та private. За замовченням змінні, видимі в області, що обіймає блок паралельного виконання, є спільними (shared). Змінні, оголошені всередині блока, за замовченням вважають закритими (private).

#pragma omp parallel private(j)		*#pragma omp parallel shared(j)*	
j	**private**	j	**shared**
j=1	потік 1	j=1, j=2, j=3	потік 1
j=2	потік 2	j=1, j=2, j=3	потік 2
j=3	потік 3	j=1, j=2, j=3	потік 3
j=4	потік 4	j=1, j=2, j=3	потік 4

Примітка: В системах з динамічною зміною кількості потоків значення за замовченням визначено **OMP_DYNAMIC=true**. Значення змінної **OMP_DYNAMIC** можна визначити за допомогою функції **omp_get_dynamic()**, або встановити за допомогою функції **omp_set_dynamic()**.

Щоб відключити динамічну зміну кількості потоків слід встановити **omp_set_dynamic(0)** або **OMP_DYNAMIC=false.**

10. Підключення бібліотеки omp.h до проекту

Для використання технології OpenMP в C/C++ необхідно підключити бібліотеку omp: #include та при компілюванні встановити необхідний параметр компілятора. Якщо параметр /openmp не вказано, компілятор

ігноруватиме прагми і директиви OpenMP. Параметр /openmp можна включити через інтерфейс проекта Visual C. Для цього у діалоговому вікні властивостей проекта слід відконфігурувати Мову, змінивши значення *Підтримка OpenMP*.

Завдання

1. Підключіть до середовища виконання програм (наприклад, CodeBlock) бібліотеку omp.h.
2. Відкомпілюйте та запустіть програми, наведені у Прикладі 2 та Прикладі 3.
3. У звіті покажіть результати запуску програм з різною кількістю потоків (одним, двома, чотирма, вісьмома). Використовуйте різні способи задання кількості потоків (за допомогою змінної OMP_NUM_THREADS або функції omp_set_num_threads(int), аргумента директиви parallel).
4. Дайте відповіді на контрольні запитання.

Контрольні питання

1. Які комп'ютерні платформи відносяться до обчислювальних систем зі спільною пам'яттю?
2. Які підходи використовують для розробки паралельних програм?
3. Що таке OpenMP? На якій моделі базується технологія OpenMP?
4. Які основні переваги технології OpenMP? З яких компонентів складається OpenMP?
5. Які проблеми виникають при використанні спільних даних в паралельно виконуваних потоках?
6. Які компілятори забезпечують підтримку технології OpenMP?
7. Що розуміють під паралельною програмою в рамках технології OpenMP? Яка структура OpenMP програми?
8. Який формат запису директив OpenMP?
9. Який мінімальний набір директив OpenMP дозволяє почати розробку паралельних програм? Яким чином здійснюється запуск OpenMP програми?
10. Як визначити час виконання OpenMP програми?
11. Що таке потік (thread)? Яким чином створюються потоки (threads) в OpenMP програмі?
12. Яким чином виконується розподіл обчислень в OpenMP?
13. Яким чином відбувається в OpenMP взаємодія між потоками?
14. Які класи даних фігурують в паралельній області?
15. Чим визначається кількість потоків у паралельній області?
16. Яким чином можна змінити кількість потоків у паралельній області?
17. Яким чином здійснюється синхронізація потоків?
18. Назвіть найважливіші змінні оточення OpenMP.
19. Яке призначення директиви parallel?

Рекомендовані джерела

1. *Коцовський В. М.* Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. 188 с. (Розділ 10) – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf
2. *Коцовський В. М.* Теорія паралельних обчислень: Конспект лекцій. Частина 1. – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/25826/1/Конспект%20лекцій.%20Частина%201.pdf
3. OpenMP API specification [Ел. ресурс, англ.]: специфікації OpenMP – URL: https://www.openmp.org/specifications/.
4. OpenMP [Ел. ресурс]: Матеріал з Вікіпедії – URL: https://en.wikipedia.org/wiki/OpenMP
5. Ясько, М.М. Навчальний посібник до вивчення курсів "Паралельна обробка даних" та "Мови обчислень та кластерні системи" [Текст] /М.М.Ясько. – Д.: РВВ ДНУ, 2010. – 76 с. – URL: http://repository.dnu.dp.ua:1100/upload/0437891d38e3501a2c067570a5fcea63PP6.pdf
6. Чернишенко С. В., Ясько М. М., Чернишенко В. С. Паралельні та розподілені обчислення: навч. посібник / С. В. Чернишенко, М. М. Ясько, В. С. Чернишенко. − Хмельницький: ХНУ, 2013. − 111 с.
7. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. 48 с. – URL: https://dspace.onu.edu.ua/server/api/core/bitstreams/de54e483-b53f-442b-a5ec-b3fd91eff98a/content
8. Мочурад Л. І., Бойко Н. І. Використання технології OpenMp для розрахунку електростатичного поля систем електронної оптики. Науковий вісник НЛТУ України. 2019, т. 29, № 3. С. 125-128. – URL: https://doi.org/10.15421/40290326

ЛАБОРАТОРНА РОБОТА №3. ДИРЕКТИВА FOR

Теоретичні відомості

Директиви **OpenMP** можна розділити на три категорії: визначення паралельної секції, розподіл роботи, синхронізація.

Кожна директива може мати кілька додаткових атрибутів-*клауз* (**clause**). Окремо специфікуються клаузи для призначення класів змінних, які можуть бути атрибутами різних директив.

Більшість директив **OpenMP** застосовується до структурних блоків. Структурні блоки − це послідовності операторів з однією точкою входу на початку блоку і однією точкою виходу в кінці блоку.

Директиви розподілу роботи **(work-sharing directives)** розподіляють виконання зазначеної області між потоками групи. Ці директиви не породжують нових потоків і при вході в гілки цієї директиви не використовується бар'єрна синхронізація.

OpenMP визначає три директиви розподілу роботи: **for**, **sections** і **single**.

1. Директива for має наступний вигляд:

#pragma omp for [clause [[,] clause] ...] new-line for-loop

Директива **for** визначає, що цикл **for**, наступний за директивою, повинен виконуватися паралельно групою потоків.

Опції для прагми for:

- ***private*** (список) – задає список змінних, для яких породжується локальна копія в кожному потоці; початкове значення локальних копій змінних зі списку не визначене;
- ***firstprivate*** (список) – задає список змінних, для яких породжується локальна копія в кожному потоці; локальні копії змінних ініціалізуються значеннями цих змінних в головному потоці (master thread);
- ***lastprivate*** (список) – змінним, переліченим у списку, присвоюють результат з останньої ітерації цикла;
- ***reduction*** (оператор:список) – задає оператор і список спільних змінних; для кожної змінної створюються локальні копії в кожному потоці; локальні копії ініціалізуються відповідно до типу оператора (для адитивних операцій – 0 або його аналоги, для мультиплікативних операцій – 1 або її аналоги); над локальними копіями змінних після виконання усіх операторів паралельної області виконується заданий оператор; для мови Сі операторами є +, *, -, &, |, ^, &&, ||; порядок виконання операторів не визначений, тому результат може відрізнятися від запуску до запуску.
- ***schedule*** (type[, chunk]) – опція задає, яким чином ітерації цикла будуть розподілені між потоками;

- ***collapse*** (n) — опція вказує, що n послідовних тісновкладених циклів асоціюється з даною директивою; для циклів створюється спільний простір ітерацій, який ділиться між потоками; якщо опція collapse не задана, то директива відноситься тільки до одного циклу, що безпосередньо за нею слідує;
- ***ordered*** – опція, яка вказує на те, що в циклі моуть зустрічатися директиви ordered; в такому випадку визначається блок всередині тіла цикла, який має виконуватися в тому порядку, в якому ітерації ідуть в последовному циклі;
- ***nowait*** – в кінці паралельного цикла відбувається неявна бар'єрна синхронізація паралельних потоків: подальше виконання цих потоків буде здійснюватися тільки тоді, коли всі вони досягнуть даної точки; якщо у такій затримці немає необхідності, то опція nowait дозволяє потокам, які вже досягли кінця цикла, продовжити виконання без синхронізації з рештою. Якщо директива end do в явному вигляді не вказана, то в кінці паралельного цикла синхронізація все одно буде виконана.

Приклад 1: Директива omp for

<table>
<tr>
<td>

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
 #pragma omp parallel
 {
  #pragma omp for
  for (i=0;i<1000;i++)
    printf("%d ",i);
 }
 return 0;
}
```

</td>
<td>

```
#include <stdio.h>
#include <omp.h>
int main()
{ int i;
 #pragma omp parallel for
  for (i=0;i<1000;i++)
    printf("%d ",i);
 return 0;
}
```

</td>
</tr>
</table>

Приклад 2: «Згладжування» масиву

Розглянемо приклад, який визначає середні значення двох сусідніх елементів масиву і записує результати в інший масив. Застосування директиви **#pragma omp for** означає, що при виконанні циклу **for** в паралельній області ітерації циклу повинні бути розподілені між потоками групи:

```
#pragma omp parallel
{
  #pragma omp for
     for (int i = 1; i <size; ++ i) x[i] = (y[i-1] + y[i+1])/2;
}
```

Якби цей код виконувався на чотирьохпроцесорному комп'ютері, а у змінної **size** було б значення 100, то виконання ітерацій 1-25 могло б бути

доручено першому процесору, 26-50 – другому, 51-75 – третьому, а 76-99 – четвертому. Це характерно для так званої статичної політики планування.

Аналогічний результат буде і при застосуанні скороченого способу запису комбінації директив **#pragma omp parallel** та **#pragma omp for**:

```
#pragma omp parallel for
  for (int i = 1; i <size; ++ i) x[i] = (y[i-1] + y[i+1])/2;
```

Слід зазначити, що в кінці паралельної секції виконується бар’єрна синхронізація (*barrier synchronization*). Інакше кажучи, досягнувши кінця секції, всі потоки блокуються доти, поки останній потік не завершить свою роботу. Якщо з щойно наведеного прикладу виключити директиву **#pragma omp for**, то кожен потік виконає повний цикл **for.**

При розпаралелюванні циклів слід переконатися, що ітерації циклу не мають залежностей. Якщо цикл не містить залежностей, компілятор може виконувати цикл в будь-якому порядку, в тому числі і паралельно. Дотримання цієї важливої вимоги компілятор не перевіряє – програміст має піклуватися про це сам. Якщо вказати компілятору розпаралелити цикл, що містить залежності, то спроби виконати вказівку призведуть до помилки.

Крім того, OpenMP накладає обмеження на цикли **for**, які можуть бути включені в блок **#pragma omp for** або **#pragma omp parallel for**. Цикли **for** повинні відповідати наступному формату:

```
for (
    [цілочисельний тип] i = інваріант циклу;
            i {<,>, =, <=,> =} інваріант циклу;
              i {+,-}= інваріант циклу
             )
```

Ці вимоги введені для того, щоб при вході в цикл OpenMP міг визначити число ітерацій.

Приклад 3. Паралельне обчислення скалярного добутку двох векторів $\boldsymbol{y = \sum_{i=1}^{n} a_i \cdot b_i}$. В програмі застосовано комбінацію паралельного цикла та операції редукції[1] по всіх процесах.

```
#include <omp.h>
#include<stdio.h>
main ()
{
int   i, n;
float a[100], b[100], sum;
```

[1] Операція редукції – операція, що збирає результати виконання задач усіма процесами та здійснює над ними деяку, вказану у параметрах, дію. Такими можуть бути додавання, визначення мінімального та інш.

```
/* Ініціалізація елементів векторів */
n = 100;
for (i=0; i < n; i++)
        a[i] = b[i] = i * 1.0;
sum = 0.0;

/* Створення множини паралельних процесів і розспаралелювання
 * цикла по ітераціях. При виході з цикла всі значення змінної sum
 * додаються по всіх процесах. */
 #pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);

/* Головний процес виводить на екран значення sum */
printf("   Sum = %f\n", sum);
}
```

Приклад 4. Приклад аналогічний до попереднього: паралельне обчислення скалярного добутку двох векторів, але додавання здійснюється в циклі в окремій підпрограмі. В програмі застосовується комбінація паралельного цикла та операції редукції по всіх процесах.

```
#include <omp.h>
#include<stdio.h>

#define VECLEN 100
float a[VECLEN], b[VECLEN], sum;

/* Підпрограма, в якій додаються елементи векторів */
float dotprod ()
{
int i,rank;

rank = omp_get_thread_num();
#pragma omp for reduction(+:sum)
  for (i=0; i < VECLEN; i++)
    {
    sum = sum + (a[i]*b[i]);
    printf(" rank = %d i=%d\n", rank, i);
    }
return(sum);
}
```

```
main ()
{
int i;

/* Ініціалізація елементів векторів */
for (i=0; i < VECLEN; i++)
  a[i] = b[i] = 1.0 * i;
sum = 0.0;

/* Створення множини паралельних процесів */
#pragma omp parallel
  sum = dotprod();

printf("Sum = %f\n",sum);

}
```

2. Функції роботи з таймерами

В OpenMP передбачені функції для роботи з системним таймером. Функція **omp_get_wtime()** повертає в потік виконання, який її викликає, астрономічний час, що минув з певного моменту в минулому в секундах (дійсне число подвійної точності). Формат функції:

double omp_get_wtime(void);

Якщо деяку ділянку програми оточити викликами даної функції, то різниця значень, що повертаються, покаже час роботи даної ділянки.

Гарантується, що момент часу, який використовується в якості точки відліку, не буде змінений за час існування процесу. Таймери різних ниток можуть бути не синхронізовані і видавати різні значення.

Функція **omp_get_wtick()** повертає в нитку виконання, що її викликає, роздільну здатність таймера в секундах. Цей час можна розглядати як міру точності таймера. Формат функції:

double omp_get_wtick(void);

Наступний приклад ілюструє застосування функцій **omp_get_wtime()** та **omp_get_wtick()** для роботи з таймерами в OpenMP:

Приклад 5. Використання таймера:

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char * argv [])
{
    #pragma omp parallel num_threads (3)
      {
```

```
        double start_time, end_time, tick;
        start_time = omp_get_wtime ();
        for (int i = 0; i <100000; i ++);
        end_time = omp_get_wtime ();
        tick = omp_get_wtick ();
        printf("Потік N% d Час виконання% lf \n", omp_get_thread_num (),
        end_time-start_time);
        printf("Потік N% d Точність таймера% lf \n", omp_get_thread_num (),
        tick);
        }
}
```

В даному прикладі здійснюється вимір початкового моменту часу, потім, після виконання циклу, вимір кінцевого моменту часу. Різниця значень дає час, витрачений на виконання програмного фрагменту. Крім того, вимірюється точність системного таймера.

3. Загальні та приватні змінні

Розробляючи паралельні програми, треба розуміти, які дані є загальними (**shared**), а які приватними (**private**), – від цього залежить не тільки продуктивність, але й коректна робота програми.

Загальні змінні доступні всім потокам з групи, тому змінення таких змінних в одному потоці видимі іншим потокам в паралельній області. Що стосується приватних змінних, то кожен потік з групи має в своєму розпорядженні їхні окремі екземпляри, тому змінення таких змінних в одному потоці ніяк не позначаються на їхніх примірниках, що належать іншим потокам.

За замовчуванням всі змінні в паралельній області – спільні, але з цього правила є три винятки:

1) приватними є індекси паралельних циклів **for**;
2) приватними є локальні змінні блоків паралельних областей;
3) приватними будуть будь-які змінні, зазначені в опціях **private**, **firstprivate**, **lastprivate** та **reduction**.

Розглянемо наступний фрагмент програми:

Приклад 6. Використання змінних:

```
float sum = 10.0f;
MatrixClass myMatrix;
int j = myMatrix.RowStart();
int i;
#pragma omp parallel
  {
    #pragma omp for firstprivate (j) lastprivate (i) reduction (+: sum)
       for (i = 0; i <count; ++ i)
```

```
        {
          int dI = 2 * i;
          for (; j <dI; ++j)
          {
              sum + = myMatrix.GetElement(i, j);
          }
        }
}
```

Змінна ***i*** є приватною, бо вона є індексом циклу що розпаралелюється. Змінна ***j*** за замовчуванням не є приватною, але явно зроблена такою через опцію **firstprivate.** Змінна **dI** є приватною, тому що вона оголошена в паралельній області. Будь-які нестатичні та ті, що не являються членами класу **MatrixClass** змінні, оголошені в методі **myMatrix::GetElement**, будуть приватними. Змінні **i, j** та **sum** зроблені приватними для кожного потоку з групи (кожен потік буде розпоряджатися своєю копією кожної з цих змінних), тому що вони вказані в опціях **firstprivate, lastprivate** та **reduction**.

Нагадаємо, що кожна з чотирьох опцій **private, firstprivate, lastprivate** і **reduction** визначає список приватних змінних, але семантика цих опцій різниться. Опція **private** означає, що для кожного потоку повинна бути створена приватна копія кожної змінної зі списку. Приватні копії будуть ініціалізуватися значенням за замовчуванням (із застосуванням конструктора за замовчуванням, якщо це доречно). Наприклад, змінні типу **int** мають за замовченням значення 0.

У опції **firstprivate** така ж семантика, але перед виконанням паралельної області вона вказує копіювати поточне значення змінної, яка буде приватною змінною, в кожен потік, використовуючи конструктор копій, якщо це доречно.

Семантика опції **lastprivate** теж збігається з семантикою опції **private**, але при виконанні останньої ітерації циклу або розділу конструкції розпаралелювання значення змінних, зазначених в опції **lastprivate**, будуть просвоєні змінним головного потоку (master). Якщо це доречно, для копіювання об’єктів застосовується оператор присвоювання копій **(copy assignment operator**).

Схожу семантику має й опція **reduction**, але її параметрами опції є змінна і оператор. Підтримувані цією опцією оператори перераховані в табл. 1. Змінна повинна бути скалярного типу (наприклад, **float, int** або **long**). Зміна розділу **reduction** ініціалізується в кожному потоці значенням, зазначеним у таблиці 1. У кінці блоку коду оператор **reduction** застосовується до кожної приватної копії змінної, а також до початкового значення змінної.

У фрагменті програми (**Приклад 6**) змінна **sum** неявно ініціалізується в кожному потоці значенням 0.0f (зауважте, що в таблиці вказано канонічне

значення 0, але в даному випадку воно приймає форму 0.0f, оскільки **sum** має тип **float**). Після виконання блоку **#pragma omp for** над усіма приватними значеннями та вихідним значенням **sum** (яке в нашому випадку дорівнює 10.0f) виконується операція +. Результат присвоюється вихідній загальній змінній **sum**.

Таблиця 1. Оператор reduction

Оператор **reduction**	Значення (канонічне), що використовується для ініціалізації
+	0
*	1
-	0
&	~0 (кожен біт встановлений)
|	0
^	0
&&	1
||	0

Передача даних за допомогою директиви threadprivate

Для передачі даних між паралельними потоками з одного паралельного структурного блоку програми до іншого, минаючи послідовний структурний блок використовується директива **threadprivate**.

Усі змінні, включені до списку директив **threadprivate**, повинні бути проініціалізовані до початку першого паралельного блоку, і їх ініціалізація можлива лише один раз протягом усієї програми. Крім того, змінні, включені до списку, не можуть фігурувати в інших висловах OpenMP, за винятком **copyin**, **schedule** або **if**. Категорично заборонено їх використання у директивах **private**, **firstprivate**, **lastprivate**, **shared** та **reduction**.

Не допускається застосування передачі значень змінних з одного паралельного структурного блоку в інший у динамічному (**dynamic**) режимі роботи паралельної програми. Тобто, протягом дії директиви **threadprivate** повинен бути встановлений статичний режим роботи паралельної програми.

Нижче приведено приклад використання директиви **threadprivate**.

Приклад 7. Використання директиви threadprivate:

```
#include <stdio.h>
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate (alpha)
void main ( )
{
```

```
/*Turn    off    dynamic    mode    */
   omp_set_dynamic (0);
/* first parallel region*/
#pragma omp parallel private (i, beta)
   for (i=0; i<10; i++)
                alpha[i] = beta[i] = omp_get_thread_num()+i;
/*second  parallel region*/
#pragma omp parallel
       printf  ("num_thread=%d  alpha[3]=%d  and  beta  [3]  =%d\n",
                omp_get_thread_num(), alpha[3], beta[3]);
  }
```

Завдання

1. Розробити послідовну програму обчислення скалярного добутку двох векторів $y = \sum_{i=1}^{n} a_i \cdot b_i$.
2. Виконати дослідження часу виконання послідовної програми для різних розмірів векторів. З'ясувати, вектори якого найбільшого розміру можна обробити.
3. Занесіть результати дослідження до таблиці.

Розмір вектора (n)**	Час виконання послідовної програми (1 потік)	Кількість потоків **p***	
		Час виконання програми 3	Час виконання програми 4
1000000			
2000000			
4000000			
5000000			
10000000			

* кількість потоків визначити відповідно до можливостей комп'ютера (мінімум p=2).

** Розмір вектора визначається відповідно до варіанту[2]:

1 варіант	2 варіант	3 варіант	4 варіант
1000000	1000000	1000000	1000000
2000000	3000000	4000000	1500000
3000000	5000000	7000000	2000000
4000000	7000000	10000000	2500000
5000000	9000000	13000000	3000000

4. Вивчіть програми прикладів 3-4. Скомпілюйте та запустіть всі програми.

[2] Запропоновані розміри векторів дозволено змінити.

5. Порівняйте час виконання програм та занесіть результати до таблиці. Побудуйте графіки часу виконання програм різного розміру для різної кількості процесорів.
6. Зробіть висновки. У звіті вкажіть характеристики процесора, на якому були виконані програми.

Контрольні питання

1. Що таке секція паралельної програми?
2. Як здійснюється розпаралелювання циклів в OpenMP? Які умови мають виконуватися, щоб цикли можна було розпаралелити?
3. Які засоби є в OpenMP для управління розподілом ітерацій циклів між потоками?
4. Як визначається порядок виконання ітерацій в розпаралелюваних циклах в OpenMP?
5. Які правила синхронізації обчислень в розпаралелюваних циклах в OpenMP?
6. Як можна обмежити роспаралелювання фрагментів програмного коду з невисокою обчислювальною складністю?
7. Як визначають спільні і локальні змінні потоків?
8. Які засоби є в OpenMP для управління кількістю створюваних потоків?
9. Що розуміють під динамічним режимом створення потоків?
10. Як здійснюється управління вкладенням паралельних фрагментів?
11. Як забезпечується єдиність програмного коду для послідовного та паралельного варіантів програми?

Рекомендовані джерела

1. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. (Розділ 10) – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf
2. OpenMP API specification [Ел. ресурс, англ.]: специфікації OpenMP – URL: http://openmp.org/wp/openmp-specifications/.
3. 3.OpenMP [Ел. ресурс, англ.]: Матеріал з Вікіпедії – URL: http://en.wikipedia.org/wiki/OpenMP
4. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. 48 с. – URL: https://dspace.onu.edu.ua/server/api/core/bitstreams/de54e483-b53f-442b-a5ec-b3fd91eff98a/content

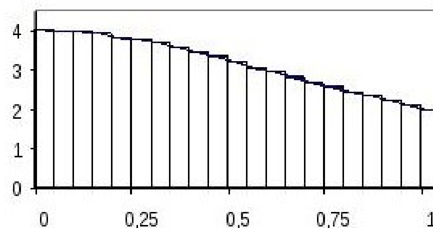
ЛАБОРАТОРНА РОБОТА №4. ПАРАЛЕЛЬНЕ ОБЧИСЛЕННЯ ЧИСЛА ПІ.

Завдання 1

1. Розробити послідовну програму обчислення числа π за формулою прямокутників (або обрати визначений інтеграл з варіантів завдань):

$$\pi = \int_0^1 \frac{4}{1 + x^2} dx$$

2. У програмі використати кількість прямокутників (n), на які розбивається криволінійна трапеція, як вхідний параметр.



3. Застосувати у програмі таймер для обліку часу виконання обчислень.
4. Виконати програму для різних значень n та зафіксувати час виконання програми для різних n на однопроцесорній системі.

Примітка: для кожного значення n отримати не менше трьох значень. Обчислити для них середні.

5. Занести отримані результати до відповідних комірок таблиці 1.

Завдання 2

1. Модифікуйте послідовну програму обчислення числа π (див. завдання 1) за формулою прямокутників $\pi = \int_0^1 \frac{4}{1+x^2} dx$ прагмами OpenMP (*застосуйте прагму for*).
2. Застосуйте у програмі таймер для обліку часу виконання обчислень (omp_get_wtime).

Виконайте програму для п’яти різних значень n та різної кількості потоків виконання (threads), принаймні для p=2.

Примітка: для кожного значення n та для кожного значення p отримати не менше трьох значень. Обчисліть для них середні значення.

3. Занесіть отримані результати до таблиці 1.
4. Обчисліть прискорення та ефективність програм, виконаних на двох та чотирьох потоках.
5. Побудуйте графіки залежності прискорення та ефективності від розміру задачі (n).
6. Зробіть висновки про те, 1) чи узгоджуються отримані результати з теоретичними; 2) чи є аномалії ефективності; якщо так, то чим, на Вашу

думку, їх можна пояснити; 3) для якої кількості потоків та якої кількості прямокутників параметри кращі…

Таблиця 1.

Кількість прямокутників (n)*	Час виконання послідовної програми (1 потік)	Час виконання програми (2 потоки)	Прискорення S_2	Ефективність E_2	Час виконання програми (4 потоки)	Прискоренн S_4	Ефективність E_4
10000000							
50000000							
100000000							
500000000							
1000000000							

* Кількість прямокутників визначається відповідно до варіанту:

1 варіант	2 варіант	3 варіант	4 варіант	5 варіант
10000000	100000000	200000000	600000000	800000000
50000000	300000000	400000000	700000000	850000000
100000000	500000000	800000000	800000000	900000000
500000000	700000000	800000000	900000000	950000000
1000000000	900000000	1000000000	1000000000	1000000000

Контрольні питання

1. Що таке секція паралельної програми?
2. Як здійснюється розпаралелювання циклів в OpenMP? Які умови мають виконуватися, щоб цикли можна було розпаралелити?
3. Які засоби є в OpenMP для управління розподілом ітерацій циклів між потоками?
4. Як визначається порядок виконання ітерацій в розпаралелюваних циклах в OpenMP?
5. Які правила синхронізації обчислень в розпаралелюваних циклах в OpenMP?
6. Як можна обмежити розпаралелювання фрагментів програмного коду з невисокою обчислювальною складністю?
7. Як визначають спільні і локальні змінні потоків?
8. Які засоби є в OpenMP для управління кількістю створюваних потоків?
9. Що розуміють під динамічним режимом створення потоків?
10. Як здійснюється управління вкладенням паралельних фрагментів?
11. Як забезпечується єдиність програмного коду для послідовного та паралельного варіантів програми?

Рекомендовані джерела

1. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. (Розділ 10) – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf
2. OpenMP API specification [Ел. ресурс, англ.]: специфікації OpenMP – URL: http://openmp.org/wp/openmp-specifications/.
3. OpenMP [Ел. ресурс, англ.]: Матеріал з Вікіпедії – URL: http://en.wikipedia.org/wiki/OpenMP
4. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. 48 с. – URL: https://dspace.onu.edu.ua/server/api/core/bitstreams/de54e483-b53f-442b-a5ec-b3fd91eff98a/content

Завдання для індивідуального виконання

1.	$\int_{-2}^{0}(x^2+5x+6)cos2xdx$	9.	$\int_{0}^{2x}(3x^2+5)cos2xdx$
2.	$\int_{-2}^{0}(x^2-4)cos3xdx$	10.	$\int_{0}^{2x}(2x^2-15)cos3xdx$
3.	$\int_{-1}^{0}(x^2+4x+3)cos\ x\ dx$	11.	$\int_{0}^{2x}(3-7x^2)cos2xdx$
4.	$\int_{-2}^{0}(x+2)^2cos3xdx$	12.	$\int_{0}^{2x}(1-8x^2)cos4xdx$
5.	$\int_{-4}^{0}(x^2+7x+12)cos\ x\ dx$	13.	$\int_{-1}^{0}(x^2+2x+1)sin\ 3x\ dx$
6.	$\int_{0}^{x}(2x^2+4x+7)cos\ 2x\ dx$	14.	$\int_{0}^{3}(x^2-2x)sin\ 2x\ dx$
7.	$\int_{0}^{x}(9x^2+9x+11)cos\ 3x\ dx$	15.	$\int_{0}^{x}(x^2-3x+2)sin\ x\ dx$
8.	$\int_{0}^{x}(8x^2+16x+17)cos\ 4x\ dx$	16.	$\int_{0}^{x/2}(x^2-5x+6)sin\ 3x\ dx$

ЛАБОРАТОРНА РОБОТА №5: АЛГОРИТМИ ПЛАНУВАННЯ ПАРАЛЕЛЬНОГО ВИКОНАННЯ ЦИКЛІВ В ТЕХНОЛОГІЇ OPENMP

Теоретичні відомості

1. Виконання критичних секцій: директива critical

1. За допомогою директив critical можна вказати ділянку коду, який буде виконуватися тільки одним потоком в один момент часу. Критичну секцію, якщо у програмі така не одна, можна іменувати.

```
#pragma omp critical [(ім’я)] новий рядок
структурований блок
```

Якщо один з потоків почав виконання критичної секції з даним іменем, то інші потоки, які почали виконання цієї ж секції, будуть заблоковані. Щойно перший потік завершить виконання секції, один із заблокованих потоків увійде в неї. Вибір чергового потоку здійснюється випадковим чином.

Приклад 1. Дві критичні секції:

```
#pragma omp critical (first)
{
    workA();
}
#pragma omp critical (second)
{
    workB();
}
//секції з workA() та workB() будуть виконані одночасно
```

Приклад 2. Одна критична секція:

```
#pragma omp critical
{
    workC();
}
#pragma omp critical
{
    workD();
}
//буде завершена секція з workC(), і тільки потім з workD()
```

2. Відключення синхронізації: директива nowait

Конструкція for за замовченням передбачає завершення бар’єрною `синхронізацією`, тобто паралельні потоки виходять з секції for тільки тоді,

коли всі вони досягнуть кінця цикла. Якщо такої необхідності немає, то вказують директиву nowait.

Приклад 3. Директива nowait:

```
#pragma omp parallel shared(n,a,b,c,d,sum) private(i)
schedule(static)
{
   #pragma omp for nowait
   for (i = 0; i < n; i++)
   a[i] += b[i];

   #pragma omp for nowait
   for (i = 0; i < n; i++)
   c[i] += d[i];

   #pragma omp for nowait reduction(+:sum)
   for (i = 0; i < n; i++)
   sum += a[i] + c[i];
}
```

Директива працює тільки з schedule(static).

Також nowait можна використовувати з директивою single.

Приклад 4. nowait та single:

```
#pragma omp parallel
{
    #pragma omp single
        printf_s("Beginning work1.\n");
    work1();
    #pragma omp single
        printf_s("Finishing work1.\n");
    #pragma omp single nowait
        printf_s("Finished work1 and beginning work2.\n");
    work2();
}
```

Фрагмент коду, відмічений директивою single, виконує один потік, а решта потоків за замовченням чекає його завершення, або, у випадку nowait, продовжують виконання.

3. Впорядкування результатів роботи: директива ordered

Директива **ordered** потрібна для послідовного упорядкування результатів роботи, що виконується паралельно.

Приклад 5. Виведення індексів послідовно:

```
#pragma omp for ordered schedule(dynamic)
    for (i=lb; i<ub; i+=st)
        work(i);
```

```
void work(int k)
{
    #pragma omp ordered
        printf_s(" %d", k);
}
```

4. Директива atomic

Директива **atomic** дозволяє уникнути «гонки даних», тобто одночасного оновлення значення змінної декількома потоками. У прикладі кожен потік буде оновлювати свої елементи, причому паралельно з іншими потоками, в той час як з директивою **critical** доступ до елементів масиву здійснювався б послідовно, причому дозвіл на виконання дії кожен потік отримував би у випадковому порядку. Директива застосовується тільки до оператора, що безпосередньо слідує за нею (у прикладі – дія директиви не поширюється на змінну ***y***)

Приклад 6. директива atomic:

```
#pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++)
    {
        #pragma omp atomic
            x[index[i]] += work1(i);
        y[i] += work2(i);
    }
```

5. Директива schedule

Директива ***schedule*** допомагає збалансувати навантаження потоків.

schedule (type[, chunk]) – опція задає, яким чином ітерації цикла будуть розподілені між потоками.

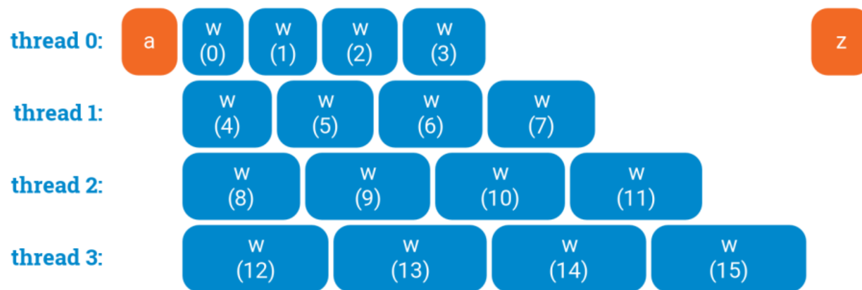
Розглянемо фрагмент коду:

Приклад 7[3].

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i)
{
    w(i);
}
z();
```

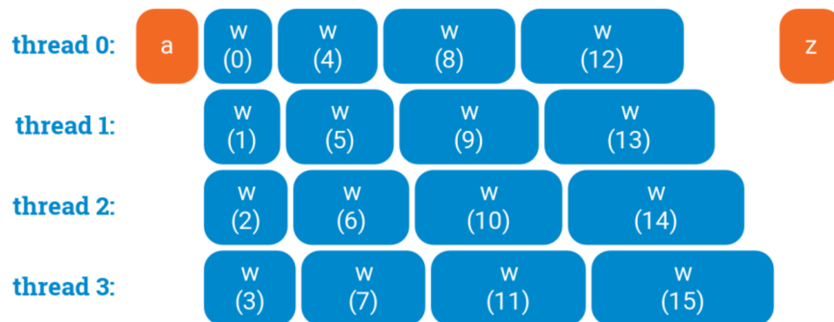
Нехай час виконання функції ***w(i)*** пропорційний до значення її аргумента. Тоді при виконанні наведеного фрагмента коду навантаження потоків за замовченням буде таким:

[3] https://ppc.cs.aalto.fi/ch3/schedule/



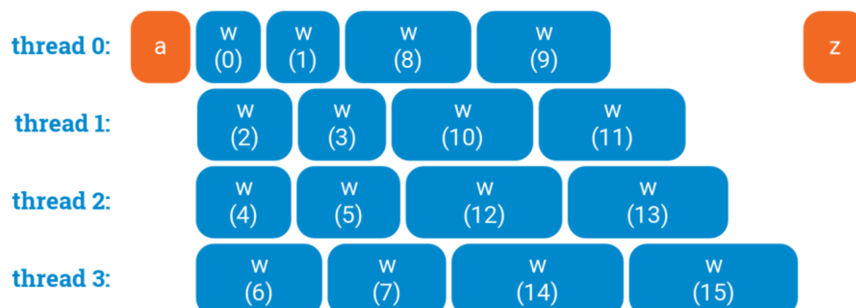
За допомогою директиви ***schedule(static, 1)*** буде створено розклад виконання ітерацій з розподілом по 1 завданню кожному потоку.

```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i)
{
    w(i);
}
z();
```



Або при ***schedule(static, 2)*** – по 2 завдання кожному потоку (що у даному прикладі не має сенсу):

```
a();
#pragma omp parallel for schedule(static,2)
for (int i = 0; i < 16; ++i)
{
    w(i);
}
z();
```



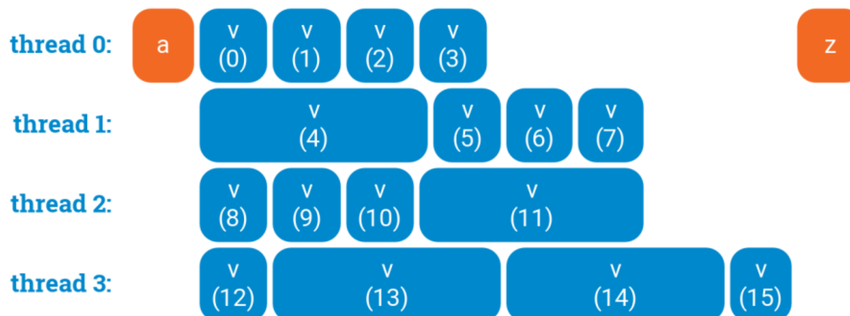
При статичному плануванні розподіл завдань між потоками здійснюється в момент запуску циклу.

У разі складаних робіт є сенс у динамічному плануванні, при якому кожен потік приймає і виконує спочатку одну ітерацію, а по завершенні її виконання приймає іншу, ще ніким не виконану.

Розглянемо наступний приклад:

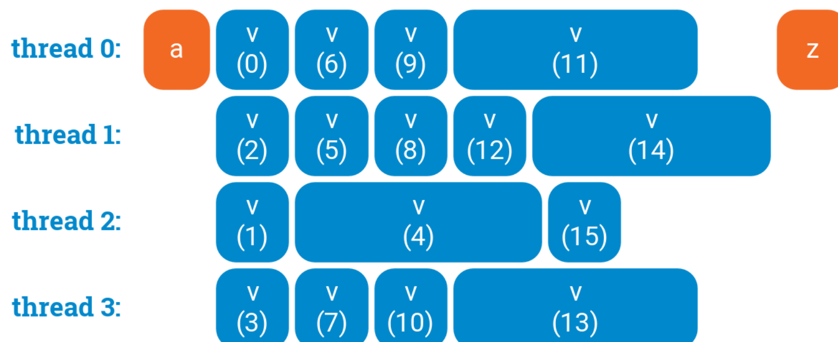
Приклад 8

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i)
{
    v(i);
}
z();
```



При використанні директиви ***schedule*** з динамічним плануванням навантаження потоків буде більш збалансованим:

```
a();
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < 16; ++i)
{
    v(i);
}
z();
```



Однак у підсумку слід зазначити, що розклад формується жадібним чином, тобто не завжди оптимально.

6. Директива sections

Приклад 9

```
#pragma omp parallel sections
{
#pragma omp section
{
printf ("id = %d, \n", omp_get_thread_num());
}

#pragma omp section
{
printf ("id = %d, \n", omp_get_thread_num());
}
}
```

Завдання до виконання

Порядок виконання роботи: Створити програму яка має реалізувати наступні дії:

1) створити квадратні матриці A та B розміром n×n, елементи яких визначені випадковим чином. n задає кількість рядків і стовпців. Також задати p – кількість потоків, які виконуватимуть паралельну область програми. Змінні можуть задаватися у програмному коді або вводитися з клавіатури;
2) у паралельній області за допомогою директиви single або master вивести такі дані: номер лабораторної роботи; назву лабораторної роботи; групу студента; ПІБ студента; номер варіанта і номер завдання;
3) у кожній матриці окремо обчислити паралельним способом вираз відповідно до свого варіанта. Розподіл ітерацій між потоками виконати за допомогою директиви for з використанням опції schedule. Для кожної з матриць використати різні значення параметра type та розміри блоку – chunk. Результати обробки записати в масиви C та D;
4) вивести результати обробки матриць паралельним способом, вказавши під час виводу розподіл ітерацій за потоками для різних значень опції schedule. Подати результати у звіті у вигляді таблиці.
5) Порівняти результати, отримані за різних значень параметра type та розміром блоку – chunk. Зробити висновки.

Варіанти для індивідуального виконання

Варіант 1. Знайти мінімальне значення кожного рядка матриці.
Варіант 2. Знайти максимальне значення кожного рядка матриці.
Варіант 3. Порахувати кількість додатних елементів у кожному рядку матриці.

Варіант 4. Порахувати кількість від'ємних елементів у кожному рядку матриці.

Варіант 5. Порахувати кількість нульових елементів у кожному рядку матриці.

Варіант 6. Порахувати суму елементів у кожному рядку матриці.

Варіант 7. Порахувати суму додатних елементів у кожному рядку матриці.

Варіант 8. Порахувати суму від'ємних елементів у кожному рядку матриці.

Варіант 9. Порахувати суму додатних елементів у кожному стовпці матриці.

Варіант 10. Порахувати суму від'ємних елементів у кожному стовпці матриці.

Варіант 11. Знайти мінімальне значення кожного рядка матриці.

Варіант 12. Знайти максимальне значення кожного рядка матриці.

Варіант 13. Порахувати кількість додатних елементів у кожному рядку матриці.

Варіант 14. Порахувати кількість від'ємних елементів у кожному рядку матриці.

Варіант 15. Порахувати кількість нульових елементів у кожному рядку матриці.

Варіант 16. Порахувати суму елементів у кожному рядку матриці.

Варіант 17. Порахувати суму додатних елементів у кожному рядку матриці.

Варіант 18. Порахувати суму від'ємних елементів у кожному рядку матриці.

Варіант 19. Порахувати суму додатних елементів у кожному стовпці матриці.

Варіант 20. Порахувати суму від'ємних елементів у кожному стовпці матриці.

Варіант 21. Знайти мінімальне значення кожного рядка матриці.

Варіант 22. Знайти максимальне значення кожного рядка матриці.

Варіант 23. Порахувати кількість додатних елементів у кожному рядку матриці.

Варіант 24. Порахувати кількість від'ємних елементів у кожному рядку матриці.

Варіант 25. Порахувати кількість нульових елементів у кожному рядку матриці.

Варіант 26. Порахувати суму елементів у кожному рядку матриці.

Варіант 27. Порахувати суму додатних елементів у кожному рядку матриці.

Варіант 28. Порахувати суму від'ємних елементів у кожному рядку матриці.

Варіант 29. Порахувати суму додатних елементів у кожному стовпці матриці.

Варіант 30. Порахувати суму від'ємних елементів у кожному стовпці матриці.

Контрольні запитання

1. Які комп’ютерні платформи відносяться до обчислювальних систем зі спільною пам’яттю?
2. Які підходи використовують для розробки паралельних програм?
3. Що таке OpenMP? На якій моделі базується технологія OpenMP?
4. Які основні переваги технології OpenMP? З яких компонентів складається OpenMP?
5. Які проблеми виникають при використанні спільних даних в паралельно виконуваних потоках?
6. Які компілятори забезпечують підтримку технології OpenMP?
7. Що розуміють під паралельною програмою в рамках технології OpenMP? Яка структура OpenMP програми?
8. Який формат запису директив OpenMP?
9. Який мінімальний набір директив OpenMP дозволяє почати розробку паралельних програм? Яким чином здійснюється запуск OpenMP програми?
10. Як визначити час виконання OpenMP програми?
11. Що таке потік (thread)? Яким чином створюються потоки (threads) в OpenMP програмі?
12. Яким чином виконується розподіл обчислень в OpenMP?
13. Яким чином відбувається в OpenMP взаємодія між потоками?
14. Які класи даних фігурують в паралельній області?
15. Чим визначається кількість потоків у паралельній області?
16. Яким чином можна змінити кількість потоків у паралельній області?
17. Яким чином здійснюється синхронізація потоків?
18. Назвіть найважливіші змінні оточення OpenMP.
19. Яке призначення директиви parallel?
20. Що таке секція паралельної програми?
21. Як здійснюється розпаралелювання циклів в OpenMP? Які умови мають виконуватися, щоб цикли можна було розпаралелити?
22. Які засоби є в OpenMP для управління розподілом ітерацій циклів між потоками?
23. Як визначається порядок виконання ітерацій в розпаралелюваних циклах в OpenMP?
24. Які правила синхронізації обчислень в розпаралелюваних циклах в OpenMP?
25. Як можна обмежити роспаралелювання фрагментів програмного коду з невисокою обчислювальною складністю?
26. Як визначають спільні і локальні змінні потоків?
27. Які способи організації взаємовиключень можуть бути використані в OpenMP?
28. Що розуміють під операцією редукції?
29. Що розуміють під атомарною (неподільною) операцією?
30. Як визначають критичну секцію?

31. Які операції є в OpenMP для змінних семафорного типу (замків)?
32. В яких ситуаціях слід застосовувати бар'єрну синхронізацію?
33. Як здійснюється в OpenMP розпаралелювання по задачах (директива sections)?
34. Як визачаються однопотокові ділянки паралельних фрагментів (директиви single та master)?
35. Як здійснюється синхронізація стану пам'яті (директива flush)?
36. Як використовують постійні локальні змінні потоків (директиви threadprivate та copyin)?
37. Які засоби є в OpenMP для управління кількістю створюваних потоків?
38. Що розуміють під динамічним режимом створення потоків?
39. Як здійснюється управління вкладенням паралельних фрагментів?
40. Як забезпечується єдиність програмного коду для послідовного та паралельного варіантів програми?

ЛАБОРАТОРНА РОБОТА №6. МНОЖЕННЯ МАТРИЦІ НА ВЕКТОР

Завдання 1. Пропрацювати приклад паралельного множення матриці на вектор. Відкомпілювати програму, запустити на двох потоках та зафіксувати час виконання програми для матриць різного розміру:

Розмір матриці та вектора (N)	Час виконання програми на одному процесорі	Час виконання програми на двох процесорах	Прискореня S_2	Ефективність E_2
10000				
15000				
20000				
25000				
30000				

Примітка: запустити програму послідовно для матриць: A = [10000 x 100000] [15000 x 15000] і так далі та відповідних векторів.

Розміри матриці та вектора можна змінити відповідно до можливостей комп'ютера. З'ясувати граничні розміри матриць, які можна розмістити у внутрішній пам'яті.

Приклад. Приклад паралельного множення матриці на вектор.

```
#include <omp.h>
#include<stdio.h>

#define M 10

main ()
{
float A[M][M], b[M], c[M];
int i, j, rank;

/* Ініціалізація даних */
for (i=0; i < M; i++)
  {
  for (j=0; j < M; j++)
    A[i][j] = (j+1) * 1.0;
  b[i] = 1.0 * (i+1);
  c[i] = 0.0;
  }
printf("\Виведення значень матриці A та вектора b на екран:\n");
for (i=0; i < M; i++)
  {
  printf("  A[%d]= ", i);
  for (j=0; j < M; j++)
    printf("%.1f ", A[i][j]);
  printf("  b[%d]= %.1f\n", i, b[i]);
  }
```

```
double start = omp_get_wtime();
omp_set_dynamic(0);
omp_set_num_threads(2);

/* Створення множини паралельних процесів і в кожному з них задаються
 * свої локальні змінні rank та i*/
#pragma omp parallel shared(A, b, c) private(rank, i)
  {
  rank = omp_get_thread_num();

/* Директива розпаралелювання цикла по ітераціях */
#pragma omp for private(j)
  for (i=0; i < M; i++)
    {
    for (j=0; j < M; j++)
      c[i] += (A[i][j] * b[j]);

/* Кожен процес виводить свій порядковий номер, значення ітерації цикла
 * та значення результуючого вектора на кожній ітерації цикла і в середині
 * критичної секції */
    #pragma omp critical
      {
      printf(" rank= %d i= %d c[%d]=%.2f\n", rank, i, c[i]);
      }
    }   /* Кінець паралельного цикла */
  } /* Завершення паралельної конструкції */
double end = omp_get_wtime();
        double time = end - start;
        printf(" %f sec.\n", time);
}
```

Завдання 2. Модифікувати програму множення матриці на вектор, застосувати поділ матриці на горизонтальні (варіанти для непарних номерів) або вертикальні смуги (варіанти для парних номерів).
Використати для розподілу робіт директиву **sections**.

Рекомендовані джерела

1. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. (Розділ 10) – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf
2. Офіційний сайт OpenMP – www.openmp.org.

ЛАБОРАТОРНА РОБОТА №7. МНОЖЕННЯ МАТРИЦЬ

Теоретичні відомості

На прикладі простої програми множення матриць можна побачити, як слід використовувати **OpenMP** для паралелізації програми. Розглянемо наступний невеликий фрагмент коду множення двох матриць. Це дуже простий приклад, і якщо дійсно необхідно написати хорошу підпрограму для перемноження матриць, слід взяти до уваги ефекти кешування або використовувати найкращий алгоритм (Штрассе, Копперсміта і Винограда або якийсь інший).

Приклад 0. Послідовне множення матриць

```
for (ii = 0; ii < dim; ii++)
  { for(jj = 0;jj<dim;jj++)
     { for (kk = 0; kk < dim; kk++)
         { array[ii][jj] += array1 [ii][kk] * array2[kk][jj]; }
     }
  }
```

Зверніть увагу, що тіла циклів на кожному рівні вкладеності можуть виконуватися незалежно один від одного. Одже паралелізувати наведений вище код дуже просто: вставте прагму **#pragma omp parallel for** перед самим зовнішнім циклом (циклом зі змінною ***ii***). Краще всього вставити прагму перед самим зовнішнім циклом, оскільки це дасть найбільший виграш в продуктивності. У паралелізованному циклі змінні **array**, **arrayl**, **array2** і **dim** є загальними для потоків, а змінні **ii**, **jj** і **kk** − приватними для кожного потоку. Наведений код тепер стає таким:

Приклад 1

```
#pragma omp parallel for shared(array,array1, array2, dim) privat (ii, jj, kk)
        for (ii = 0; ii < dim; ii++)
        {  for jj = 0; jj < dim; jj++)
             { for (kk = 0; kk < dim; kk++)
                   array[ii][jj] = array1 [ii][kk] * array2[kk][jj];
             }
        }
```

Як ще один приклад розглянемо наступний фрагмент коду, призначений для обчислення суми значень **f**(**x**) для цілочисельних аргументів з інтервалу **0 <= x < n**:

```
for (ii = 0; ii < n; ii++)
  { sum = sum + some_complex_long_fuction(a[ii]);
  }
```

Для паралелізації наведеного фрагмента як перший крок можна зробити наступне:

```
#pragma omp parallel for shared(sum, a, n) private(ii, value)
        for (ii = 0; ii < n; ii++)
          { value * some_complex_long_fuction(a[ii]);
           #pragma omp critical
                sum = sum + value;
          }
```

або, що краще, можна використовувати оператор редукції, в результаті чого вийде таке:

```
#pragma omp parallel for shared(a, n) private(ii) reduction(+: sum)
        for (ii = 0; ii < n; ii++)
              { sum = sum + some_complex_long_fuction(a[ii]);
              }
```

Інші варіанти розпаралелювання множення матриць наведено у прикладах (див. Приклад 2, Приклад 3, Приклад 4, Приклад 5).

Завдання

1. Опрацювати приклади «множення матриць» (тобто програму одного з прикладів 2, 3, 4, або 5 налагодити у середовищі виконання, з'ясувати, як саме здійснено розпаралелювання).
2. Ознайомитися з різними підходами до множення матриць (поділ на горизонтальні та вертикальні смуги, блоки).
3. * Розробити паралельну програму вирішення задачі відповідно до варіанта:
   номери 1, 4, 7, 10, 13, 16 – розбиття на горизональні смуги;
   номери 2, 5, 8, 11, 14, 17 – розбиття на вертикальні смуги;
   номери 3, 6, 8, 12, 15, 18 – розбиття на блоки.
4. Дослідити час виконання програми у послідовному і у паралельному варіантах. Прокоментувати отримані результати.

Приклад 2. Множення матриць (перетворення послідовного алгоритму).

#2.1. Послідовний варіант перемноження матриць мовою C

```
#define N 4096

int main()
{
   int i, j, k;
   double s;
/* Виділення пам’яті під матриці */
   double **a=(double**)malloc(sizeof(double*)*N);
   for(i=0; i<N; i++)
      a[i]=(double*)malloc(sizeof(double)*N);
   double **b=(double**)malloc(sizeof(double*)*N);
   for(i=0; i<N; i++)
      b[i]=(double*)malloc(sizeof(double)*N);
   double **c=(double**)malloc(sizeof(double*)*N);
   for(i=0; i<N; i++)
      c[i]=(double*)malloc(sizeof(double)*N);
// Ініціалізація матриць
   for(i=0; i<N; i++)
      for(j=0; j<N; j++){
         a[i][j]=(i+j)*0.0001e0;
         b[i][j]=(i-j)*0.0001e0;
         c[i][j]=0.0e0;
      }
// Основний обчислювальний блок
   for(i=0; i<N; i++)
      for(j=0; j<N; j++)
         for(k=0; k<N; k++) c[i][j]+=a[i][k]*b[k][j];



/* Обчислення контрольної суми та друк результату */
   for(i=0, s=0.0e0; i<N; i++)
      for(j=0; j<N; j++)
         s+=c[i][j];
   printf("N= %d, Sum=%lf\n", N, s);
}
```

Приклад 2. Множення матриць (перетворення послідовного алгоритму).

```
#2.2. Перемноження матриць мовою C з використанням технології OpenMP
#include <stdio.h>
#include <omp.h>
#define N 4096
int main()
{
   int i, j, k, size;
   double t1, t2, s;
/* Виділення пам'яті під матриці */
   double **a=(double**)malloc(sizeof(double*)*N);
   for(i=0; i<N; i++)
      a[i]=(double*)malloc(sizeof(double)*N);
   double **b=(double**)malloc(sizeof(double*)*N);
   for(i=0; i<N; i++)
      b[i]=(double*)malloc(sizeof(double)*N);
   double **c=(double**)malloc(sizeof(double*)*N);
   for(i=0; i<N; i++)
      c[i]=(double*)malloc(sizeof(double)*N);
// Ініціалізація матриць
   for(i=0; i<N; i++)
      for(j=0; j<N; j++){
         a[i][j]=(i+j)*0.0001e0;
         b[i][j]=(i-j)*0.0001e0;
         c[i][j]=0.0e0;
   }
   t1=omp_get_wtime();
// Основний обчислювальний блок
#pragma omp parallel shared(a, b, c) private(i, j, k)
   {
      size=omp_get_num_threads();
   #pragma omp for schedule(static)
      for(i=0; i<N; i++)
         for(j=0; j<N; j++)
            for(k=0; k<N; k++) c[i][j]+=a[i][k]*b[k][j];
   }
   t2=omp_get_wtime();
/* Обчислення контрольної суми та друк результату */
   for(i=0, s=0.0e0; i<N; i++)
      for(j=0; j<N; j++)
         s+=c[i][j];
   printf("N= %d, Nproc=%d, Sum=%lf, Time=%lf\n",
            N, size, s, t2-t1);
}
```

Приклад 3. Множення матриць (розпаралелювання з використанням секцій).

Розподілимо задачу на два потоки таким чином: для знаходження елемента результуючої матриці необхідно перемножити **N** пар чисел; кожен з двох потоків буде виконувати половину цих множень, а потім результати будуть додані та записані в підсумкову комірку.

```
#include <omp.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

/* Процедура перемножує матриці aMatrixA*aMatrixB узгоджених розмірів M*N*K
Результат передається в матрицю aMatrixC */
void MatrixMul(double aMatrixA[], double aMatrixB[], double aMatrixC[], unsigned M, unsigned N, unsigned K)
 {
  double cell=0; // Оголошено змінну для збору перемножених елементів
  #pragma omp parallel num_threads(2)
        //Ця прагма ініціалізує паралельні потоки
        //Конструкція під прагмою (в даному випадку цикл for(unsigned iRow=0; iRow<M; iRow++))
        //тепер виконається в кожному з потоків
        for(unsigned iRow=0; iRow<M; iRow++)
        {
         for(unsigned iCell= 0; iCell<K; iCell++)
                {
                cell= 0;
                #pragma omp sections reduction(+:cell)
         //Ця прагма ініціалізує паралельні секції.
         //Тепер кожен створений раніше потоків виконає тільки одну з вказаних секцій.
         //При цьому, по завершенні секцій в змінну cell буде занесена сума локальних змінних cell кожного з потоків
```

```
            {
            #pragma omp section
              {//Цей фрагмент буде виконаний у першому потоці.
               printf("%dth thread begin to summaries firts elements of %dth row\n", omp_get_thread_num(), iRow);
                  for(unsigned i= 0; i<N/2; i++)
                          cell+= aMatrixA[iRow*N + i] * aMatrixB[K*i+iCell];
              }
            #pragma omp section
            //Ця прагма ініціалізує нову секцію, яка повинна виконатися в одному з потоків
              {//Цей фрагмент буде виконаний у другому потоці
               printf("%dth thread begin to summaries last elements of %dth row\n", omp_get_thread_num(), iRow);
               for(unsigned i= N/2; i<N; i++)
                     cell+= aMatrixA[iRow*N + i] * aMatrixB[K*i +iCell];
              }
            }
        #pragma omp single
        //Ця прагма ініціалізує виконання операції тільки одним з потоків
        //Запис у пам'ять буде здійснена тільки одним з потоків,
        //не зважаючи на те, що команда знаходиться в паралельній частині програми
        aMatrixC[iRow*K + iCell]= cell;
        }
    }
}
```

Приклад 4

```
//Програма. Паралельне перемноження двох квадратних матриць
#include <stdio.h>
//#include "stdafx.h"
#include <omp.h>
#define N 200
int main()
{
double A[N][N], B[N][N], C[N][N];
int i, j, k,size;
double start, end;
for (i = 0; i < N; i++)
	for (j = 0; j < N; j++)
	{
		A[i][j] = i + j;
		B[i][j] = i * j;
	}
printf("Enter number of threads:  ");
scanf_s("%d", &size);
omp_set_dynamic(0);
omp_set_num_threads(size);
start = omp_get_wtime();
#pragma omp parallel for shared(A, B, C) private(i,  j,  k)
for (i = 0; i < N; i++) {
	for (j = 0; j < N; j++) {
		C[i][j] = 0;
		for (k = 0; k < N; k++)
			C[i][j] += A[i][k] * B[k][j];
	}
}
end = omp_get_wtime();
for (i = 0; i < N; i++)
	for (j = 0; j < N; j++)
	{
		printf("C[%d][%d] = %lf\n", i, j, C[i][j]);
	}
	printf("Time %lf\n", end - start);
return 0;
}
```

Приклад 5

```
#include <iostream>
#include <cstdlib>
#include <omp.h>
```

```
using namespace std;

void randomiseMatrix(int **matrix, int n, int m) {
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < m; j++) {
			matrix[i][j] = rand() % 11;
		}
	}

	return;
}

int main(int argc, char** argv) {
	//srand(time(NULL));
	double start, end;
	int threadsNum;
	//int n1 = 1000;
	//int m1 = 500;
	//int n2 = 500;
	//int m2 = 1200;
	int n1, m1, n2, m2;
	cout << "input n";
	cin >> n1;
	cout << endl;
	m1 = n1; n2 = n1; m2 = n1;

	//Матриця n1 x m1
	int **matrix1;
	//Матриця n2 x m2
	int **matrix2;

	matrix1 = (int**)malloc(sizeof(int*)*n1);
	for (int i = 0; i < n1; i++) {
		matrix1[i] = (int*)malloc(sizeof(int)*m1);
	}
	matrix2 = (int**)malloc(sizeof(int*)*n2);
	for (int i = 0; i < n2; i++) {
		matrix2[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Генеруємо випадкові матриці для множення
	randomiseMatrix(matrix1, n1, m1);
	randomiseMatrix(matrix2, n2, m2);

	int **result = (int**)malloc(sizeof(int*)*n1);;
```

```
	for (int i = 0; i < n1; i++) {
		result[i] = (int*)malloc(sizeof(int)*m2);
	}

	//Встановлюємо кількість потоків
	omp_set_dynamic(0);
	cout << "Enter number of threads:  ";
	cin >>  threadsNum;
	cout << endl;

	omp_set_num_threads(threadsNum);
	start = omp_get_wtime();

	int i, j, k;
#pragma omp parallel for shared(matrix1, matrix2, result) private(i, j, k)
	for (i = 0; i < n1; i++) {
		for (j = 0; j < m2; j++) {
			result[i][j] = 0;
			for (k = 0; k < m1; k++) {
				result[i][j] += (matrix1[i][k] * matrix2[k][j]);
			}
		}
	}
	end = omp_get_wtime();

	printf("Time %lf\n", end - start);

	return 0;
}
```

Рекомендовані джерела

1. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. (Розділ 10) – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf
2. OpenMP API specification [Ел. ресурс, англ.]: специфікації OpenMP – URL: http://openmp.org/wp/openmp-specifications/.
3. OpenMP [Ел. ресурс, англ.]: Матеріал з Вікіпедії – URL: http://en.wikipedia.org/wiki/OpenMP
4. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. 48 с. – URL: https://dspace.onu.edu.ua/server/api/core/bitstreams/de54e483-b53f-442b-a5ec-b3fd91eff98a/content

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ №1

1. Створити послідовну програму для виконання обчислень за умовою (по варіантах). Зафіксувати час виконання послідовної програми.
2. Модифікувати програму для паралельного виконання.
3. Порівняти час виконання послідовної та паралельної програми для різного розміру задачі та різної кількості потоків. Обчислити прискорення та ефективність паралельного алгоритму.
4. У звіті навести 1) опис послідовного алгоритму; 2) обгрунтувати можливості розпаралелювання алгоритму; 3) дослідити характеритики паралельного алгоритму; 4) зробити висновки.

Варіанти завдань

Варіант 1. Задача комівояжера.

Комівояжер розпочинає подорож з міста_1 та повинен відвідати усі n міст і повернутися в початкове. Відстані між містами відомі. В якому порядку слід відвідувати міста, щоб замкнений шлях комівояжера був найкоротшим?

А) Вивести 20 найкоротших маршрутів у порядку зростання відстані.

Б) Створити графічний інтерфейс для задання координат міст, надати користувачу можливість переглядати знайдений шлях на екрані.

В) Використати генетичний алгоритм.

Варіант 2. Задача про упаковку.

Є диск, на якому фрагментами записано дані. Потрібно записати файли різного розміру таким чином, щоб якомога раціональніше використати доступне на диску вільне місце.

Вивести 20 найбільш вдалих розв’язків.

Примітка: файл, що містить розміри фрагментів даних згенерувати випадковим чином або прочитати з текстового. Формат даних такий: перша послідовність визначає стан пам’яті – перше число визначає обсяг зайнятої, а друге – обсяг вільної; друга послідовність – розміри файлів, що потрібно записати (через ;). Послідовності розділено символом нового рядка.

Варіант 3. Пошук максимального прибутку.

Є набір можливих капіталовкладень, вартість кожного s_i. Вкладення мають принести прибуток w_i. Слід розподілити інвестиції вартістю К таким чином, щоб отримати максимальний прибуток.

Вивести 20 найбільш вдалих розв’язків.

Примітка: Вартість капіталовкладень та прибутки від них генеруються випадковим чином або отримуються з текстового файла Data.txt, розмір інвестицій задається користувачем.

Формат файла такий: дві послідовності чисел розділено символом нового рядка. Перша послідовність визначає розмір вкладення, друга – прибуток від них. Числа у послідовностях розділені символом «;».

Варіант 4: Задача про 8 ферзів.

Потрібно розмістити на шаховій дошці 8 ферзів таким чином, щоб вони не били один одного. Вивести усі можливі комбінації (за допомогою консольного додатка чи графічного інтерфейсу).

Варіант 5: Максимальна спільна послідовність.

Є два рядка довжиною n та m. Знайти найдовшу послідовність символів, що є спільною для обох рядків.

Варіант 6: Розфарбування графа.

Є неорієнтований граф. Знайти мінімальну кількість кольорів, якими можна розфарбувати вершини графа таким чином, щоб сусідні вершини, з'єднані ребром, мали різний колір.

Варіант 7: Сортування.

Є масив довжиною n. Необхідно відсортувати масив, використовуючи алгоритм злиття.

Алгоритм полягає у наступному: доки довжина масива не стане рівною 1, рекурсивно розбити масив навпіл (масив з одного елемента вважати відсортованим). На другому етапі зливати два відсортованих масиви в один (спочатку по одному елементу, потім по два, по чотири і т.д.). На виході отримати відсортований масив.

Варіант 8: Знаходження числа Пі.

Знайти число Пі, використовуючи метод Монте-Карло.

Варіант 9: Збільшення контрасту зображення.

Є картинка, для якої потрібно збільшити контраст. Процедура полягає у наступному: взяти вікно розміром 3×3 і визначити колір центрального пікселя P. Для цього знайти найбільший темний та найбільший світлий піксель серед сусідів та обчислити величину k=(P-мін)/(макс-мін). Якщо k>0.5, то збільшити значення P на 1, у протилежному випадку – зменшити на 1 (розрахунок проводити окремо для R, G, B). Після цього пересунути вікно і повторити дії для наступного пікселя. Зображення потрібно обробити визначену кількість разів – кожна ітерація підвищує якість.

Варіант 10: Надійність мережі.

Задано мережу у вигляді графа (множини вершин і ребер): G=(V, E), – та імовірність виходу з ладу кожного ребра мережі. Потрібно визначити

надійність мережі – імовірність, з якою між будь-якими вершинами графа існує зв'язок через справні ребра.

Варіант 11: Кросворд.

Є матриця A розміром $n \times m$, в якій містяться нулі та одиниці, та набір слів (файл Data.txt), розділених символом нового рядка. Використовуючи слова зі словника, потрібно створити кросворд для матриці A (розміщуючи слова у клітинках з нулями).

Варіант 12: Найдовший шлях.

Задано граф G=(V, E) та I(e) – вага кожного ребра. Знайти найдовший шлях у графі (довжина графа – сума ваг ребер, через які цей шлях проходить).

Варіант 13: Очікувана вартість пошуку.

У базі даних є множина записів R. Для кожного запису визначено імовірність звернення до нього – p(r), $r \in R$. Розбити множину R на m сегментів для того, щоб мінімізувати очікуваний час доступу до даних. Час переходу d(i, j) між сегментами i та j визначають як i-j-1, якщо i>j, та як m-i+j-1, якщо i<j. Очікувана вартість переходу визначається як сума по усіх можливих i та j величин p(Ri)*p(Rj)*d(I, j), де $p(R_i) = \sum_{r \in Ri} p(r)$.

Варіант 14: Розв'язання системи лінійних рівнянь методом Гауса.

Метод Гаусса полягає у наступному: нехай маємо систему рівнянь $\sum_{k=1}^{n} a_{ik} x_k = b_k, i = \overline{1, n}$.

Віднімемо від другого рівняння перше, помножене на таке число, що коефіцієнт при x_1 знищиться. На наступному кроці віднімемо таким самим чином від 3-го рівняння друге. Аналогічні дії слід продовжувати так далі, доки не виключимо усі коефіцієнти нижче головної діагоналі. Після цього знаходимо розв'язки рівняння за формулою:

$$x_k = \frac{1}{a_{kk}^{(k)}} \left(b_k^{(k)} - \sum_{i=k+1}^{n} a_{kl}^{(k)} x_l \right), \ k = n, n-1, \dots, 1$$

Вхідні дані згенерувати випадковим чином, розмір системи рівнянь (n) задається користувачем.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ №2. МНОЖЕННЯ МАТРИЦЬ ТА ВЕКТОРІВ

Завдання:

1. Реалізувати послідовне та паралельне виконання обчислень виразів (великими літерами позначено матриці, маленькими - вектори):

Варіант		***Варіант***	
1	$b \cdot (A \cdot b^T + d^T)$	***7***	$A \cdot b^T - d \cdot C$
2	$A \cdot b^T + (b \cdot C)^T$	***8***	$(A \cdot b^T - d^T) \cdot b^T$
3	$(d - b \cdot C) \cdot d^T$	***9***	$b \cdot (d - b \cdot C)$
4	$b \cdot (A - C) \cdot b^T$	***10***	$b \cdot (A - C) - d \cdot A$
5	$b \cdot (A - C) - d$	***11***	$(d \cdot A - b \cdot A) \cdot b^T$
6	$b \cdot A - d \cdot C$	***12***	$b \cdot (A - C) - d \cdot C$
13	Дано матриці **A** та **B** невід'ємних цілих чисел розміру **m**×**m**. Знайти найбільші спільні дільники елементів **a**(**i**,**j**) та **b**(**i**,**j**) (1≤**i**,**j**≤**m**). Результати записати в матрицю **C**. Найбільший спільний дільник двох чисел `v1 та v2` можна знайти за алгоритмом: `while (v1!=v2) if (v1>v2)v1-=v2;else v2-=v1;`		
14	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m**. Знайти найбільший спільний дільник її елементів. Найбільший спільний дільник двох чисел `v1 та v2` можна знайти за алгоритмом: `while (v1!=v2) if (v1>v2)v1-=v2;else v2-=v1;`		
15	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m**. Знайти кількість простих чисел серед її елементів.		

2. Порівняти час виконання обчислень та обчислити прискорення і ефективність паралельного алгоритму.

Альтернативні завдання

Варіант	Завдання
1	Протабулювати функцію на заданому відрізку із заданим кроком.
2	Виконати додавання двох матриць однакового розміру.
3	Обчислити суму максимальних елементів рядків матриці.
4	Знайти площу опуклого многокутника, заданого координатами вершин.
5	Знайти у даному тексті усі паліндроми.
6	Задано послідовність дійсних чисел. У кожному числі скоротити кількість знаків після коми до двох.
7	Знайти в тексті усі входження даного абзацу.

8	Дано послідовність арифметичних виразів, операндами яких є однозначні числа, а кількість операцій не більше двох. Знайти значення усіх виразів.
9	Дано матрицю дійсних чисел. Перетворити матрицю таким чином, щоб елементи її рядків були впорядковані за спаданням.
10*	Обчислити $\mathbf{A}^{\mathbf{m}}$, де **A** – дійсна матриця розміру **n**×**n**.
11*	Знайти **m**-е просте число.
12*	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m**. Побудувати матрицю **B** таким чином: **b**(**i**,**j**) – це **a**(**i**,**j**)-е просте число.
13*	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m** і просте число **p**. Побудувати матрицю **B** таким чином: **b**(**i**,**j**) =($\mathbf{log_2}$ **a**(**i**,**j**)) **mod p** (або 0).
14*	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m** та просте число **p**. Побудувати матрицю **B** таким чином: **b**(**i**,**j**) =($\mathbf{log_3}$ **a**(**i**,**j**)) **mod p** (або 0).
15*	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m** та просте число **p**. Побудувати матрицю **B** таким чином: **b**(**i**,**j**) =($\mathbf{log_4}$ **a**(**i**,**j**)) **mod p** (або 0).
16*	Дана матриця **A** невід'ємних цілих чисел розміру **m**×**m** та просте число **p**. Побудувати матрицю **B** таким чином: **b**(**i**,**j**) =($\mathbf{log_5}$ **a**(**i**,**j**)) **mod p** (або 0).

Рекомендовані джерела

1. Коцовський В. М. Теорія паралельних обчислень: навчальний посібник. Ужгород: ПП «АУТДОР-Шарк», 2021. (Розділ 10) – URL: https://dspace.uzhnu.edu.ua/jspui/bitstream/lib/38994/1/Навчальний%20посібник.pdf
2. OpenMP API specification [Ел. ресурс, англ.]: специфікації OpenMP – URL: http://openmp.org/wp/openmp-specifications/
3. OpenMP [Ел. ресурс, англ.]: Матеріал з Вікіпедії – URL: http://en.wikipedia.org/wiki/OpenMP
4. Вербіцький В. В. Паралельне програмування з використанням технології OpenMP : метод. вказівки / В. В. Вербіцький, А. Л. Максимов. Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2022. 48 с. – URL: http://dspace.onu.edu.ua:8080/bitstream/123456789/34441/1/Вербіцький%20ВВ_Парал%20програмування%20з%20OpenMP_Метод%20вказівки.pdf
5. Максимова Л. П. Методичні вказівки щодо виконання лабораторних робіт з навчальної дисципліни «Паралельне програмування та розподілені обчислення». Кременчук, 2019. 55 с.
6. Паралельні та розподілені обчислення [Текст] : навч. підруч. для студентів вищ. навч. закл. / А. Луцків, С. Луценко, В. Пасічник. Львів : Магнолія 2006, 2017. 565 с.

7. Жуков І., Корочкін О. Паралельні та розподілені обчислення. Навч. посібн. Київ: Корнійчук, 2014. 284 с. – URL: https://comsys.kpi.ua/katalog/files/paralelkorochkin.pdf
8. Кузьма К. Т., Мельник О. В. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти. Миколаїв: ФОП Швець В.М., 2020. 172 с.
9. Минайленко Р. М. Паралельні та розподілені обчислення : навч. посіб. Кропивницький: ЦНТУ, 2021. 153 с. – URL: https://dspace.kntu.kr.ua/server/api/core/bitstreams/396e02d2-725b-47b5-a1c0-ae07a9bec326/content
10. Жуковський В. В., Жуковська Н. А., Харів Н. О. Методичні вказівки до виконання лабораторних робіт з дисциплін «Оптимізація обчислень», «Паралельні та розподілені обчислення» для студентів спеціальностей 113 «Прикладна математика», 121 «Інженерія програмного забезпечення», 122 «Комп’ютерні науки». Рівне: НУВГП, 2017. 54 с. – URL: https://ep3.nuwm.edu.ua/7536/1/04-01-31.pdf
11. Наконечна О. А., Ярмоленко Т. А., Алексеєнко В. В., Якимчук Б. М. Інструктивно-методичні рекомендації з дисципліни «Технології розподілених систем та паралельних обчислень» / уклад.: Оксана Наконечна, Тетяна Ярмоленко, Вікторія Алексеєнко, Богданна Якимчук. Житомир: Житомир: Вид-во ЖДУ ім. Івана Франка, 2023. 74 с. – URL: http://eprints.zu.edu.ua/35948/1/%d1%96%d0%bd%d1%81%d1%82-%d0%bc%d0%b5%d1%82%d0%be%d0%b4_%d1%82%d0%b5%d1%85%d0%bd%d0%be%d0%bb%d0%be%d0%b3%d1%96%d1%97%20%d1%80%d0%be%d0%b7%d0%bf%d0%be%d0%b4%20%d1%81%d0%b8%d1%81%d1%82%d0%b5%d0%bc%20(4).pdf

Паралельні та розподілені обчислення:

Лабораторні роботи

Частина 1.
Технологія OpenMP

Укладачі
Ірина Володимирівна Лупан,
Степан Дмитрович Паращук

Підписано до друку *29.10.2024.* Зам. 672. Формат 60x84 $^{1}/_{16}$
Папір офсет. Цифровий друк. Ум. др. арк. 3,4. Обл. вид. арк. 1,91.
Віддруковано: ФОП Піскова М. А.
Свідоцтво про державну реєстрацію
№24440000000027816 від 18.08.2016