

# Обрахунок інтегралу заданої функції

Олег Фаренюк



7 березня 2023 р.

Creative Commons - Attribution Share Alike (CC BY-SA)



# Зміст

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Огляд задачі інтегрування</b>                     | <b>5</b>  |
| 1.1      | Вступ . . . . .                                      | 5         |
| 1.1.1    | Умовні позначення . . . . .                          | 5         |
| <b>2</b> | <b>Послідовне інтегрування</b>                       | <b>7</b>  |
| 2.1      | Ціль роботи . . . . .                                | 7         |
| 2.2      | Завдання . . . . .                                   | 7         |
| 2.2.1    | Інтерфейс . . . . .                                  | 7         |
| 2.2.2    | Скрипт Python для автоматизованих запусків . . . . . | 9         |
| 2.2.3    | Формат файлу конфігурації . . . . .                  | 10        |
| 2.2.4    | Функції для інтегрування . . . . .                   | 11        |
| 2.2.5    | Додаткові завдання . . . . .                         | 14        |
| 2.2.6    | Що потрібно здавати . . . . .                        | 14        |
| 2.2.7    | Методичні рекомендації . . . . .                     | 15        |
| 2.3      | Метод інтегрування . . . . .                         | 20        |
| 2.3.1    | Абсолютна та відносна похибка . . . . .              | 21        |
| 2.3.2    | Очевидна оптимізація . . . . .                       | 22        |
| 2.3.3    | Посилання . . . . .                                  | 23        |
| <b>3</b> | <b>Багатопоточне паралельне інтегрування</b>         | <b>25</b> |
| 3.1      | Ціль роботи . . . . .                                | 25        |
| 3.2      | Завдання . . . . .                                   | 25        |
| 3.2.1    | Інтерфейс . . . . .                                  | 26        |
| 3.2.2    | Скрипт Python для автоматизованих запусків . . . . . | 27        |
| 3.2.3    | Аналіз результатів запусків . . . . .                | 27        |
| 3.2.4    | Додаткові завдання . . . . .                         | 28        |
| 3.2.5    | Що потрібно здавати . . . . .                        | 28        |
| 3.2.6    | Методичні рекомендації . . . . .                     | 28        |
| <b>4</b> | <b>Потокобезпечна черга</b>                          | <b>33</b> |
| 4.1      | Ціль роботи . . . . .                                | 33        |

|       |  |    |
|-------|--|----|
| 4.1.1 | Аналіз проблеми . . . . .                            | 33 |
| 4.1.2 | Використання потокобезпечної черги . . . . .         | 35 |
| 4.1.3 | Розробка потокобезпечної черги . . . . .             | 36 |
| 4.2   | Завдання . . . . .                                   | 37 |
| 4.2.1 | Інтерфейс . . . . .                                  | 37 |
| 4.2.2 | Скрипт Python для автоматизованих запусків . . . . . | 38 |
| 4.2.3 | Аналіз результатів запусків . . . . .                | 38 |
| 4.2.4 | Що потрібно здавати . . . . .                        | 39 |
| 4.2.5 | Методичні рекомендації . . . . .                     | 39 |
| 4.2.6 | Додаткові завдання . . . . .                         | 40 |
| 4.2.7 | Що потрібно здавати . . . . .                        | 41 |

# Розділ 1





## Огляд задачі інтегрування

### 1.1 Вступ

Ця робота має двоякі цілі. Перша – базове практичне знайомство із чисельними розрахунками, роботою із числами з рухомою крапкою<sup>1</sup>, на прикладі чисельного інтегрування функції двох змінних.

Друга – оскільки розрахунок інтегралу ідеально паралелізується, перше знайомство із паралельними обчисленням.

#### 1.1.1 Умовні позначення

-  – порушення означає, що робота не зараховується як здана і може навіть не перевірятися.
-  – порушення може означати велику втрату балів (2-4).
-  – поодинокі порушення приводитиме до невеликої втрати балів (0.25-0.5). За систематичного повторення покарання в наступних роботах зростатиме.
-  – все решта.

---

<sup>1</sup>Floating point, строго кажучи, українською – з рухомою комою, але в коді ми використовуємо крапку, тому в тексті дотримуватимемося цієї традиції, забігаючи наперед, можна сказати – користуємося локаллю C.



## Розділ 2

# Послідовне інтегрування

### 2.1 Ціль роботи

Ціллю роботи є написати послідовний код інтегрування заданих функцій та виміряти час його роботи.

### 2.2 Завдання

Написати програму обчислення інтегралу функції двох змінних на вказаному інтервалі. Використати її обчислення інтегралу трьох переданих функцій. Виміряти час інтегрування кожної із функцій, як основу для дослідження паралельного прискорення у наступній роботі. Проаналізувати отримані результати.

Функції для інтегрування наведені в розділі 2.2.4, методи інтегрування – в 2.3. Інтерфейс – у наступному розділі, 2.2.1

#### 2.2.1 Інтерфейс



Виконавчий файл має називатися `integrate_serial`<sup>1</sup>.



Функція отримує з командного рядка номер функції, для якої потрібно виконати інтегрування **та** файл із конфігурацією. Формат файлу конфігурації описано в розділі 2.2.3.



Програма повинна вивести на консоль, у такому порядку, кожне число з нового рядка і без додаткових надписів (значення окремих еле-

---


<sup>1</sup>Якщо ваша система – Windows, буде автоматично додаватися розширення `.exe` – це ОК, не потрібно якось турбуватися про це.


ментів пояснено в розділі 2.3):


- Результат обчислення інтегралу.
- Досягнуту абсолютну похибку.
- Досягнуту відносну похибку.
- Час обчислення, в мілісекундах, цілим числом.

Приклад виводу, для (відсутнього) варіанту 7:

```
$ integrate_serial 7 data.cfg
2998.59
1.40493
0.00046853
25000
```

 Інший вивід – заборонено. Якщо у вас використовується вивід для відлагодження – передбачте механізм його вимкнення за допомогою `#define`.

 Інтерактивний ввід – з клавіатури абощо, заборонений, такі роботи автоматично не зараховуються.

 Якщо сталася помилка, необхідно вивести повідомлення із поясненням проблеми на консоль: "Wrong number of arguments", "Wrong function index", "Error opening configuration file" тощо та завершити програму із кодом завершення<sup>2</sup>, що описує проблему. Потрібно реалізувати такі коди завершення:

- 0 – якщо програма відпрацювала успішно,
- 1 – неправильна кількість аргументів командного рядка,
- 2 – неправильний номер функції,
- 3 – не вдалося відкрити файл із конфігурацією,
- 5 – помилка читання файлу конфігурації<sup>3</sup>, або помилка його вмісту – відсутні аргументи (при тому, у повідомленні про помилку слід вказати рядок, де вона сталася або інший контекст – ім'я відсутнього аргументу, вміст рядка, що створив проблему тощо),

---


<sup>2</sup>Те, що передається `return` із `main` чи функції `exit()`.

<sup>3</sup>Як ви думаєте, чому пропущено номер 4?




16 – не вдалося досягнути бажаної точності,

- Для інших проблем у виконанні вашої програми, обираєте коди повернення на ваш розсуд, починаючи з 64.


 Якщо бажаної точності не вдалося досягнути – спочатку вивести поточний результат, як описано вище, а потім повідомити про цю помилку.

### 2.2.2 Скрипт Python для автоматизованих запусків

 Для роботи із програмою слід написати скрипт мовою Python, який отримує з командного рядка:


- скільки раз потрібно розрахувати інтеграл кожної із функцій,

запускає програму вказану кількість разів для кожного із методів, використовуючи наперед створені файли конфігурації `funcX.cfg`, де  $X$  – номер функції, та аналізує вивід.

 Першим, критичним<sup>4</sup>, етапом аналізу є порівняння – чи всі спроби проінтегрувати кожну із функцій, співпадають. Однак, порівнювати floating point числа операцією `==` не можна – вона часто даватиме некоректний результат, тому слід перевірити чи значення співпадають в межах похибки:

$$|R_1 - R_2| < \epsilon,$$

де  $R_i$  – результати різних запусків,  $\epsilon$  – точність порівняння. Для цього завдання розумним є скористатися  $\epsilon = 10^{-7}$ . Більш коректним буде скористатися значенням, рівним абсолютній похибці з файлу конфігурації, але це складніше і для такої простої задачі не мало б бути необхідним – можна просто всюди скористатися узгодженими значеннями.

 Якщо результати всіх запусків співпадають в межах похибки – виводить на консоль, для кожної функції, розділяючи різні функції порожнім рядком:

1. Результат обчислення інтегралу.
2. Досягнуту абсолютну похибку.
3. Досягнуту відносну похибку.

---

<sup>4</sup>Особливо для наступної роботи

4. Мінімальний час виконання.
5. Середній час виконання.
6. Кореговане стандартне відхилення для вибірки (Corrected sample standard deviation) для часу виконання.

### 2.2.3 Формат файлу конфігурації



Файл конфігурації – текстовий файл, який містить:

- Бажана абсолютна похибка.
- Бажана відносна похибка. Похибка повинна бути меншою за 0.001 – 0.1%.
- Інтервали інтегрування по  $x$  та  $y$ .
- Кількість точок у початковому розбитті інтервалу інтегрування по осі  $x$  та по осі  $y$ .
- Максимальна кількість ітерацій – яка обмежуватиме час виконання програми<sup>5</sup>.

Значення елементів конфігурації пояснено в розділі 2.3.

Для конфігурації повинні підтримуватися коментарі – всі символи від '#' і до кінця рядка мають ігноруватися.

Порожні рядки слід ігнорувати.



Його формат наступний (конкретні числа наведені для прикладу, коментарі не обов'язкові):

```
abs_err = 0.000005      # Бажана абсолютна похибка
rel_err = 0.0002        # Бажана відносна похибка


x_start=-50             # Початок інтервалу інтегрування по x
x_end  = 50             # Кінець інтервалу інтегрування по x
y_start=-50             # Початок інтервалу інтегрування по y
y_end  = 50             # Кінець інтервалу інтегрування по y
```


---


<sup>5</sup>Зауважте, що за рекомендованого тут алгоритму інтегрування, кількість точок ростиме як  $N^2$ , де  $N$  – кількість ітерацій, тому максимальна кількість ітерацій не може бути особливо великою.


```
init_steps_x = 100      # Кількість точок у початковому розбитті  
                        # інтервалу по осі x  
init_steps_y = 100      # Кількість точок у початковому розбитті  
                        # інтервалу по осі y  
max_iter = 20           # Максимальна кількість ітерацій  
                        # поділу інтервалів інтегрування
```


 Зауважте, навколо '=' можуть бути пробіли, але їх може і не бути.

 Порядок аргументів може бути довільним! Ваш код повинен це враховувати. Однак, повтори не допускаються.

 Цей формат – підмножина TOML: <https://toml.io/en/>. Він достатньо зручний як людині, так і для автоматичного тестування. В майбутньому ми будемо використовувати більше його можливостей.


 Якщо для парсингу використовуєте сторонню бібліотеку – додайте її як директорію-підпроект, із своїм CMakeLists.txt, на який посилається ваш CMakeLists.txt. Це відрізняється від типової практики, однак – відповідні бібліотеки прості і не дуже поширені, тому навряд чи CMake матиме модулі для їх пошуку.

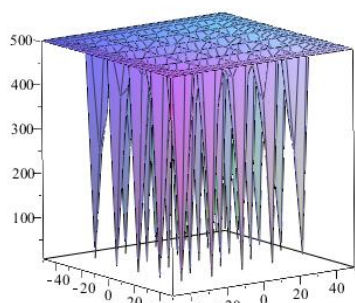
 Виняток: ця вимога не стосується бібліотек boost, зокрема – класу boost::program\_options::parse\_config\_file.

 Рекомендоване ім'я та розширення файлу конфігурації – funcX.cfg, де X – номер функції.

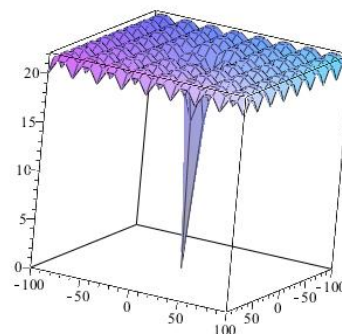
## 2.2.4 Функції для інтегрування

Всі ці функції використовуються для тестування різноманітних методів оптимізації, зокрема – генетичних алгоритмів. Вони мають достатньо складну поверхню, тому – помірно складні для запропонованого у цій роботі простого методу інтегрування – особливо грубих помилок у методиці вже не пробачать, але й особливо не капризують. Тому, перш ніж перейти до інтегрування, варто побудувати їх графік, приклад див. Рис. 2.1.

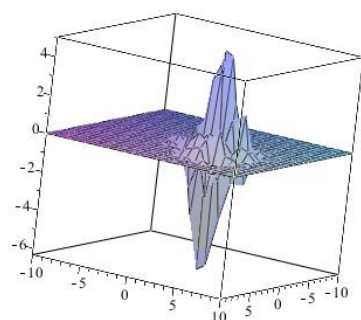
 **УВАГА!** Результати інтегрування наведені лише для довідки – щоб ви могли себе перевірити. Ці числа не повинні жодним чином використовуватися у вашій програмі!



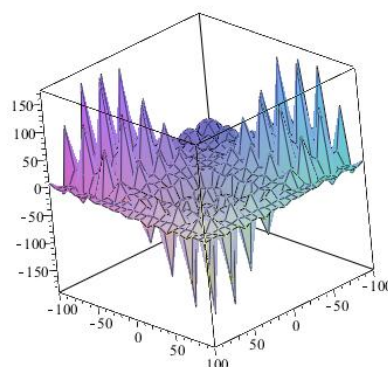
(а) Функція-1



(б) Функція-2



(в) Функція-3



(г) Додаткова функція

Рис. 2.1: Графіки досліджуваних функцій. Для тих, що мають доступ до системи комп'ютерної алгебри Maple, долучаю файл із кодом для побудови графіків цих функцій та їх інтегрування – там графіки можна буде покрутити.

### Функція – 1

$$f(x_1, x_2) = \left( 0.002 + \sum_{i=-2}^2 \sum_{j=-2}^2 \frac{1}{(5(i+2) + j + 3 + (x_1 - 16j)^6 + (x_2 - 16i)^6)} \right)^{-1}$$

Інтервал інтегрування:  $x = -50..50$ ;  $y = -50..50$ .

Ця функція відома як функція де Джонга.

Результат:  $4.545447652 * 10^6$ ;

### Функція – 2

$$f(x_1, x_2) = -a \cdot \exp \left( -b \sqrt{\frac{1}{2}(x_1^2 + x_2^2)} \right) - \exp \left( \frac{1}{2}(\cos(cx_1) + \cos(cx_2)) \right) + a + \exp(1)$$

Де  $a$ ,  $b$ ,  $c$  – параметри:

1.  $a = 20$ ,
2.  $b = 0.2$ ,
3.  $c = 2\pi$ .

Інтервал інтегрування:  $x = -100..100$ ;  $y = -100..100$ .

Відома як Ackley's function.

Результат:  $8.572082414 * 10^5$ .

### Функція – 3

$$f(x_1, x_2) = - \sum_{i=1}^m c_i \exp \left( -\frac{1}{\pi} ((x_1 - a_{1i})^2 + (x_2 - a_{2i})^2) \right) \cos \left( \pi ((x_1 - a_{1i})^2 + (x_2 - a_{2i})^2) \right)$$

де  $m$ ,  $c_i$ ,  $a_{1i}$ ,  $a_{2i}$ ,  $i = 1..m$  – параметри:

- $m = 5$ ,
- $a1 = [1, 2, 1, 1, 5]$ ,
- $a2 = [4, 5, 1, 2, 4]$ ,
- $c = [2, 1, 4, 7, 2]$ .

Інтервал інтегрування:  $x = -10..10$ ;  $y = -10..10$ .

Відома як Langermann function.

Результат:  $-1.604646665$ .

### Додаткова функція

Функція для додаткового завдання – вона помітно важча в інтегруванні.

$$f(x_1, x_2) = - \sum_{i=1}^m i \cos((i+1)x_1 + 1) \sum_{i=1}^m i \cos((i+1)x_2 + 1)$$

Де параметр  $m = 5$ .

Інтервал інтегрування:  $x = -100..100$ ;  $y = -100..100$ .

Результат:  $-5.045849736$

### 2.2.5 Додаткові завдання

- Додаткове завдання: Реалізувати цю задачу з використанням методу Монте-Карло, без використання сторонніх бібліотек.
- Додаткове завдання: Реалізувати цю задачу з використанням GSL (GNU Scientific Library): <https://www.gnu.org/software/gsl/doc/html/integration.html>. Оскільки безпосередньо інтегрування функцій двох змінних вона не підтримує, потрібно подумати – як реалізувати.
- Додаткове завдання: дослідити залежність часу виконання від особливостей процесора, для цього запустити програму на кількох різних комп'ютерах.
- Додаткове завдання: проінтегрувати додаткову функцію із розділу 2.2.4 із точністю, кращою за 0.1%. Безпосереднє застосування методу, запропонованого в цій роботі потребуватиме на такі розрахунки багато годин, якщо взагалі він збіжиться.
- Додаткове завдання: реалізувати обчислення інтегралу та функції, що інтегрується, з використанням методів підвищення точності, описаних в посиланнях із розділу 2.3.3.



Всі додаткові завдання потребують проаналізувати ваші результати та враження від різних методів інтегрування тих самих функцій. Наприклад, якщо ви проінтегрували функції методом із основного завдання, за допомогою адаптивних методів з GSL, за допомогою Монте-Карло з GSL і самостійно реалізованого Монте-Карло – опишіть, яку різницю ви побачили між цими методами – з точки зору швидкодії, точності, зручності.

### 2.2.6 Що потрібно здавати

Роботу можна виконувати втрьох, однак, наступну роботу потрібно виконувати у тому ж складі команди.

На git слід здати

-  Код програм, разом із CMakeLists.txt для їх компіляції.
-  Скрипт для їх запуску – вимоги до нього описані в розділі 2.2.2.




-  Супровідний документ із результатами запусків та вашим аналізом.
- ◇  Обов'язково в ньому вкажіть склад команди та додаткові завдання, якщо такі виконувалися!
- ◇ Також, в ньому повинні бути таблиці із мінімальним часом інтегрування, середнім часом та стандартним відхиленням.
- ◇ Бажаним є аналіз мікроархітектурних характеристик інтегрування цих функцій – кількості cache miss, branch predictor fails тощо, отриманий за допомогою VTune/perf/xperf, і т.д. Формат – вільний, але зрозумілість та адекватність оцінюватиметься, можливо – додатковими балами.






### На CMS слід здати

- Склад команди: [КОМАНДА]
- Репозиторій: [посилання на репозиторій]
- Додаткові завдання: [список].



## 2.2.7 Методичні рекомендації

### Загальні

-  Робота із пам'яттю – дорога. Тому у цій роботі категорично заборонено створювати масив із всіма можливими точками  $(x, y)$ , щоб потім для них обчислювати функцію! Координати точок розраховуєте за потребою.
- ◇ На жаль, це дуже типова помилка. Але ж розраховувати точки на люту зовсім просто:  $(x_0 + i\Delta x, y_0 + j\Delta y)$ .
-  Для адекватності вимірювання часу, обирайте таку точність, щоб розрахунки тривали не менше кількох десятків секунд або й хвилини.
-  За відсутності виміру часу практична не зараховуватиметься.

-  Потрібно вимірювати лише час інтегрування – не включайте вимірювання читання конфігурації тощо.
-  Використання глобальних шляхів до файлу конфігурації є неприйнятним, практична не перевірятиметься.
-  Відсутність належної обробки помилок<sup>6</sup>, в залежності від впливу на "юзабельність", може привести як до того, що практична не перевірятиметься, так і до великої втрати балів.
-  Пам'ятайте, що ввід-вивід повільний! Обчислювальний код не повинен його містити. Звичайно, наявність такого коду для відлагодження із можливістю його викинути за допомогою умовної компіляції є прийнятною.
- Компілювати задачі для заміру часу слід із оптимізацією.
- ◇ Однак, не ламайте CMakeLists.txt заради цього – скористайтесь `"cmake -DCMAKE_BUILD_TYPE=Release <...>"` чи `"-DCMAKE_BUILD_TYPE=RelWithDebInfo"` – для профайлінгу `perf` чи `VTune`.
- ◇ В більшості IDE це відповідає target Release.
-  Використовувати наведені вище очікувані правильні результати в обчисленнях заборонено.
- ◇ Код для відлагодження може ним користуватися, якщо його можна викинути за допомогою умовної компіляції і CMakeLists.txt по замовчуванню його вимикає.


### Чисельні методи

-  Щодо очікуваного результату, якщо має бути, скажімо,  $-1.604646665$ , а у вас вийшло  $-1.609$  – це нормально для студентської роботи,  $-1.62$  – терпимо. А от якщо  $+35$  чи навіть  $-1.005$  – результат неправильний.
-  Реалізуйте оптимізацію, описану в розділі 2.3.2.

---

<sup>6</sup>Це дуже ускладнює перевірку.



-  Дуже поширеною помилкою перших років було інтегрування "по діагоналі": прямокутник інтегрування розбивався на блоки, скажімо  $N$  на  $N$ , але інтегрували лише блоки, для яких обидва індекси були рівні між собою, як на Рис. 2.2. Добре подумайте, перевірте, чи ви так не робите!

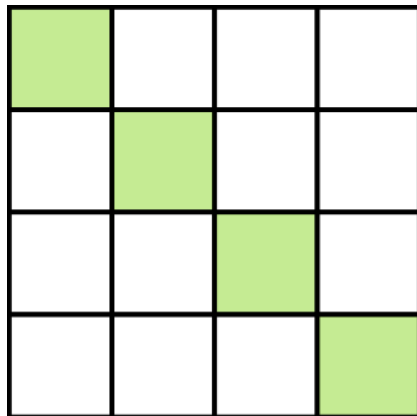




Рис. 2.2: Приклад типової помилки. Не робіть так!

-  В обчисленнях користуйтеся типом `double`! Типу `float` (4-байтовому у наших реалізаціях `C++`) буде явно недостатньо. Точніше, його використання потребуватиме помітно вищого професіоналізму.
  -  Хоча оптимізатор компілятора має шанс розібратися, якщо кожен доданок множиться на константу, це варто робити за межами циклу.
- ◇ Причина: компілятори побоюються виконувати навіть прості маніпуляції над `floating point`, типу заміни  $a \cdot b + a \cdot c$  на  $a \cdot (b + c)$ , оскільки, для важливих розрахунків, зміни, викликані такими перестановками, можуть бути неприйнятними.

### Засоби `C++`

- Обчислювальна програма повинна бути написана мовою `C++`<sup>7</sup>.
- Якщо ви хочете скористатися `GCC` чи `Clang`, оптимальним для наших потреб є використати `MSYS2` і їх пакетний менеджер, `pacman`, відомий по `Arch Linux`.

<sup>7</sup>Мову `C` тут можна вважати підмножиною `C++`, однак – ніяких `Julia`, `rust` тощо.

- Не соромтеся використовувати `std::array`, `std::vector` та інші засоби стандартної бібліотеки C++.
- Для збереження параметрів із командного рядка чи файлу конфігурації, зручним може бути `std::map`.
- Зоопарк із десятка аргументів варто зібрати у якусь структуру і передавати як єдину сутність.
- Не використовуйте низькорівневі конструкції (зокрема, запозичені з C – сирі вказівники, ручне керування пам'яттю, `#define`) без крайньої потреби!
- ◇ В більшості випадків `std::vector`, максимум – із викликом `reserve()`, чудово підійде замість `new`. У рідких винятках допоможе.
- ◇ А `malloc` у C++ взагалі не бажана<sup>8</sup> – на відміну від `new`, вона не викликає конструкторів.
- ◇ Замість "сирих" вказівників, краще скористатися `std::unique_ptr` чи `std::shared_ptr`.
- ◇ Ремарка: хоча `std::unique_ptr` не можна копіювати, не поспішайте використовувати `std::shared_ptr` – можливо, достатньо "протягувати" один об'єкт `std::unique_ptr` – переміщаючи в аргумент функції і повертаючи з неї.
- ◇ `#define` в C++14 і новіших взагалі ніде не потрібен, крім умовної компіляції – конструкцій типу `#ifdef/#ifndef`.
- Тип виключення – тип проблеми, про яку код хоче повідомити. Якщо ви робите `throw "my error";` у вас тип помилки `std::string`. Геніально, правда? Стрічка – то взагалі жахливий різновид помилки! Гірший – хіба тип помилки "ціле число".
- ◇ Створіть клас `parsing_error_t`, а як ще зробите його нащадком `std::runtime_error` чи його предка `std::exception`, ви зможете, додавши пару рядків тривіального коду на створення нащадка, писати значно більш виразний код:

```
throw parsing_error_t("My_error_description");
```

---

<sup>8</sup>Виняток – коли потрібна динамічна пам'ять, яку буде звільнятися кодом на C. Або ще щось таке екзотичне.

Детальніше див. у коді прикладу одномірного інтегрування: [https://github.com/indrekis/integrate1d\\_sample](https://github.com/indrekis/integrate1d_sample).



- Така річ безглузда: :

```
threads.emplace_back( std::thread(<args>) );
```

Слід писати:

```
threads.emplace_back(<args>);
```

`emplace_back()` конструює об'єкт зразу у призначеній для нього пам'яті контейнера. На противагу йому, `push_back()` просто копіює об'єкт, що вже існує. Тому перший, невірний, варіант використання `emplace_back()` створює (явно) тимчасовий об'єкт, який потім копіює чи переміщає на його місце – заперечуючи саму ідею використання `emplace`<sup>9</sup>.

-  Для виводу помилок варто використовувати `std::cerr`.
-  Перевірити, чи файл відкрився успішно, можна двома способами:

- ◇ Файловий потік перетворюється в `true`, під час приведення до `bool`, якщо пов'язаний із файлом і придатний до спроби читання<sup>10</sup>, і в `false`, якщо не пов'язаний, або перебуває в стані `fail` чи `bad`. Завдяки цьому працює ідіома `while( file >> word )`, але можна робити і так:

```
if (!config_stream) {
    . . . . . report error . . . . .
}
```

- ◇◇ Якщо під час читання чи запису сталася помилка, файловий потік переходить в стан `fail` (а іноді ще й `bad` або `eof`), і тоді його приведення до `bool` даватиме `false`.
- ◇◇ Іноді, хоч і не часто, `config_stream.is_open()` може бути корисним для детальнішої диференціації.

---

<sup>9</sup>Сучасні компілятори мають шанс розібратися і викинути зайві операції – але навіщо ускладнювати їм життя, перевіряти – зможуть, не зможуть.

<sup>10</sup>Ця спроба може провалитися з різних причин, наприклад – не той формат даних, але на цей момент із файловим потоком все ОК.

- ◇ Інший спосіб – дозволити файловим потокам кидати виключення у випадку переходу в стан помилки. Тут цей спосіб не розглядаємо.
- ◇ Приклад обробки файлових помилок див. тут: [https://github.com/indreki/integrate1d\\_sample](https://github.com/indreki/integrate1d_sample), а більше подробиць – книгу Nicolai M. Josuttis, "The C++ Standard Library", друге видання (або новіші, якщо будуть).
- Іноді кількість цифр у floating point, що виводяться за замовчуванням, буває недостатньою (рідше – надмірною). Керувати цим можна за допомогою маніпулятора `std::setprecision(p)` із `<iomanip>`:

```
std::cout << std::setprecision(16) << val << '\n';
```

- ◇ Уникаючи складної дискусії заради спрощення, для `double` немає сенсу вказувати виводити більше 18 цифр.
- Маніпулятори `std::fixed`, `std::scientific`, `std::defaultfloat` вказують виводити числа завжди із фіксованою крапкою, завжди в науковому форматі та за замовчуванням, відповідно.

## 2.3 Метод інтегрування

Для інтегрування можна використати звичайний метод клітин – узагальнення методу прямокутників на двовимірний випадок. Слід розбити прямокутник, на якому інтегрується функція, на прямокутники, та вважати значення інтегралу сумою значень функції у таких точках на площі прямокутників: [https://en.wikipedia.org/wiki/Riemann\\_sum](https://en.wikipedia.org/wiki/Riemann_sum), <http://www.mathros.net.ua/obchyslennja-podvijnyh-integraliv-metodom-klityn.html>. За бажанням дозволяється використовувати інші алгоритми, але забезпечення їх коректної роботи у багатопоточному середовищі – на вашій совісті.

Однак, сам по собі такий метод не дозволяє контролювати точність результатів. Для цього можна скористатися простим адаптивним підходом<sup>11</sup>, описаним у наступному розділі, 2.3.1 – ділити розбиття на прямокутники доти, поки результат не перестане змінюватися (в межах похибки). Можна довести, що для менш-більш гладких функцій такий підхід дає правильне значення інтегралу.

Ремарка: цьому методу присвячена стаття на Wiki: <https://en.wikipedia.org>.

<sup>11</sup> Існують більш складні адаптивні методи, де сітка є нерівномірною. Зокрема, такі використовує бібліотека GSL, згадана в додаткових завданнях.

[org/wiki/Adaptive\\_quadrature](https://www.gnu.org/software/gsl/doc/html/integration.html#qag-adaptive-integration). Ремарка: як приклад більш вишуканого підходу, див, скажімо: <https://www.gnu.org/software/gsl/doc/html/integration.html#qag-adaptive-integration>.

### 2.3.1 Абсолютна та відносна похибка

Нехай вам відомо точний результат обчислення інтегралу –  $F$ . Ваша програма розрахувала значення  $\text{Res}$ . Тоді абсолютна похибка:

$$\text{Err}_{abs} = |F - \text{Res}|,$$

відносна:

$$\text{Err}_{rel} = \left| \frac{F - \text{Res}}{F} \right|.$$

Зрозуміло, що якби ми знали  $F$ , нам би не потрібно було його розраховувати. Тому взнати поточну похибку ми не можемо. Проте, **є спосіб її оцінити**. Тоді, задавши бажані абсолютну та відносну похибку, обчислення можна виконати згідно такого адаптивного алгоритму:

1. Розбиваємо інтервал інтегрування по кожній із осей на якусь початкову кількість кроків (`init_steps_x`, `init_steps_y`).
  - ◇ Розумним початковим значенням буде 200-1000.
2. Розраховуємо інтеграл із таким розбиттям.
3. Збільшуємо кількість кроків розбиття вдвічі по кожній із осей.
4. Розраховуємо нове значення інтегралу.
5. Порівнюємо результати інтегрування, отримані з цих розбиттів.
  - ◇ Якщо вони відрізняються менш ніж на абсолютну похибку, а їх зміна – менша відносної, можна вважати, що обчислення завершилися успішно<sup>12</sup>.
  - ◇ Інакше переходимо до кроку 3.

---

<sup>12</sup>Такий ідеал – коли виконується і та умова і та, досяжний не завжди. На практиці тоді доводиться дивитися – для чого ми рахуємо інтеграл. Іноді важливіша абсолютна похибка – наприклад, коли це вимоги до обробки поверхонь частин якихось приладів, іноді – відносна, коли мова про дослідження якихось явищ. Для студентів, що виконують це завдання, тут прийнятно вважати, що достатньо досягнути бажаної абсолютної **або** відносної похибки.

Важливим є і порівняння різниці результатів ітерацій – абсолютна похибка і їх відносна зміна – відносна похибка. Якщо значення інтегралу великі за модулем, порядку  $10^7$ , відносна похибка, яка дорівнює 0.0001 (дуже хороша!) буде відповідати дуже великим значенням абсолютної похибки, і навпаки. Також, обережно із нульовим значенням результату – використовувати поняття відносної похибки тут важко, для простоти можна обмежитися абсолютною.

Зауважте, що навіть якщо цей алгоритм інтегрування не збігся до правильного значення<sup>13</sup>, якщо обрані похибки достатньо малі, зупинка означає, що метод не здатен далі покращити результат – відповідь перестала змінюватися, або змінюється дуже повільно.

### 2.3.2 Очевидна оптимізація

Зазвичай, найбільше часу під час інтегрування забирає обчислення підінтегральної функції. За примітивної реалізації методу, описаного вище, для випадку функції однієї змінної, вона буде на кожній ітерації *в половині точок* обчислюватися двічі – див. Рис. 2.3. Справа в тому, що, якщо границі інтегрування фіксовані, при збільшенні кількості точок розбиття вдвічі, посередині між кожними двома початковими точками з'явилася нова. Але обчислювати функцію у точках першого розбиття дуже неефективно і позбавлено сенсу – достатньо обчислити лише у нових точках.

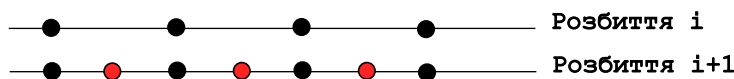


Рис. 2.3: Два послідовних розбиття. Зауважте – перераховувати потрібно лише червоні точки.

Нехай  $k$  – номер ітерації.  $\Delta x_k$  – відповідний крок розбиття, а оскільки крок ми зменшуємо вдвічі кожної ітерації,  $\Delta x_{k+1} = \Delta x_k/2$ , тоді обчислення значення інтегралу на ітерації<sup>14</sup>  $k$ :

$$F_k = \Delta x_k \sum_{i=0}^{N-1} f(x_0 + i\Delta x_k)$$

<sup>13</sup>Для цього може бути багато причин, наприклад – функція занадто швидкозмінна для обраного початкового розбиття, або містить сингулярності.

<sup>14</sup>Константу, на яку множимо всі доданки, виносимо за дужки – а значить, і з циклу, яким відбуватиметься обчислення. Для цілих чисел сучасні компілятори й самі роблять це добре, оптимізуючи, але для floating point часто не наважуються, тому краще вручну.

$$\begin{aligned}
F_{k+1} &= \frac{\Delta x_k}{2} \sum_{i=0}^{2N-1} f\left(x_0 + i \frac{\Delta x_k}{2}\right) = \\
&= \frac{\Delta x_k}{2} \left[ \sum_{i=0}^{N-1} f\left(x_0 + 2i \frac{\Delta x_k}{2}\right) + \sum_{i=0}^{N-1} f\left(x_0 + 2i \frac{\Delta x_k}{2} + \frac{\Delta x_k}{2}\right) \right]
\end{aligned}$$

Ремарка: квадратні дужки тут означають те ж, що й круглі, але виконані візуально іншими – для кращої читабельності.

Скорочуємо в аргументі на 2, порівнюємо перший доданок із виразом для  $F_k$  і отримуємо:

$$F_{k+1} = \frac{F_k}{2} + \frac{\Delta x_k}{2} \sum_{i=0}^{N-2} f\left(x_0 + i \Delta x_k + \frac{\Delta x_k}{2}\right)$$

Як бачимо, в ітерації  $k + 1$  доводиться розраховувати значення функції лише в нових точках:  $f(x_0 + i \Delta x_k + \frac{\Delta x_k}{2})$ . За всієї позірної складності, вираз очевидний.

Узагальнення на функцію двох вимірів залишаю студентам на самостійну роботу. Там виграш менший, але, все ж, значущий!

### 2.3.3 Посилання

- Прямолінійне підсумовування дає помітну похибку. Як з нею боротися див, наприклад: <https://habr.com/en/post/523654/> – "Сложение двух чисел с плавающей запятой без потери точности", <https://habr.com/en/post/525090/> – "Можно ли сложить N чисел типа double наиболее точно?".
- ◇ Там описано алгоритми: "Kahan summation algorithm" – [https://en.wikipedia.org/wiki/Kahan\\_summation\\_algorithm](https://en.wikipedia.org/wiki/Kahan_summation_algorithm),
- ◇ та алгоритм Rump–Ogita–Oishi – [https://www.researchgate.net/profile/Takeshi\\_Ogita/publication/220412129\\_Accurate\\_Floating-Point\\_Summation\\_Part\\_II\\_Sign\\_K\\_-Fold\\_Faithful\\_and\\_Rounding\\_to\\_Nearest/links/0fcfd50d7dc46843b6000000/Accurate-Floating-Point-Summation-Part-II-Sign-pdf](https://www.researchgate.net/profile/Takeshi_Ogita/publication/220412129_Accurate_Floating-Point_Summation_Part_II_Sign_K_-Fold_Faithful_and_Rounding_to_Nearest/links/0fcfd50d7dc46843b6000000/Accurate-Floating-Point-Summation-Part-II-Sign-pdf).
- ◇ Див. також "Accurate Matrix Multiplication with Multiple Floating-point Numbers" – [https://www.researchgate.net/publication/228966273\\_Accurate\\_Matrix\\_Multiplication\\_with\\_Multiple\\_Floating-point\\_Numbers](https://www.researchgate.net/publication/228966273_Accurate_Matrix_Multiplication_with_Multiple_Floating-point_Numbers).

- Jean-Michel Muller, "Handbook of floating-point arithmetic", 2nd ed, 2018 – <http://libgen.rs/book/index.php?md5=4230206901BF25FFEE04F8766FD3C00>
- Jean-Michel Muller, "Elementary Functions: Algorithms and Implementation", 2016 – <http://libgen.rs/book/index.php?md5=4230206901BF25FFEE04F8766FD3C00>



## Розділ 3

# Багатопоточне паралельне інтегрування

### 3.1 Ціль роботи

Чисельне інтегрування – задача із приголомшливим паралелізмом<sup>1</sup>, тобто – паралелізується практично ідеально. Досягається це розбиттям області інтегрування на частини, кожна із яких інтегрується в окремому потоці.

Функції для інтегрування – ті ж, що й в розділі 2, метод інтегрування в межах одного потоку – описано в розділі 2.3.

### 3.2 Завдання

- Розпаралелити розрахунок інтегралу з використанням багатопоточності.
- Реалізація повинна бути ефективною та коректною – не допускати гонитв даних (data races).
- Для роботи із засобами багатопоточності<sup>2</sup> потрібно використати стандартну бібліотеку багатопоточності C++, зокрема – клас `std::thread`.

---

<sup>1</sup>Детальніше види паралелізму вивчаються на лекціях.

<sup>2</sup>Як підтримку потоків, так і підтримку їх синхронізації, надає операційна система. Для роботи із ними мови програмування та різноманітні бібліотеки, включаючи SDK від OS, надають відповідні програмні конструкції. Тобто, потік, створений класом `std::thread` з C++, `threading.Thread` з Python чи `_beginthreadex()` з WinAPI або `pthread_create` з POSIX створюють ту ж саму сутність, керувати якою буде OS. Хоча, можливі нюанси, оскільки OS надають багато схожих та тісно переплетених засобів

- Кількість потоків для інтегрування передається з командного рядка.
- Програма повинна рівномірно поділити роботу між всіма потоками. Детальніше – див. розділ 3.2.6.
- Засоби синхронізації, які потрібно використовувати в цій роботі – в розділі 3.2.6.
- За допомогою скрипта (розділ 3.2.2) виконати запуски для кількості потоків<sup>3</sup>: 1, 2, ..., 16, 100, 1000.
- Проаналізувати, як змінюється час виконання із зміною кількості потоків, для діапазону 1-16 потоків. Пояснити отримані результати. Детальніше вимоги до аналізу описані в розділі 3.2.3.
- Проаналізувати, як відрізняється час виконання для потоків 1, 4, 16, 100, 1000 потоків. Пояснити, чому ви спостерігаєте різницю, або – чому ні.

### 3.2.1 Інтерфейс

Окрім керування кількістю потоків, інтерфейс – той же, що і в попередній роботі, див. розділ 2.2.1.



Виконавчий файл має називатися `integrate_parallel`<sup>4</sup>.



Кількість потоків передається третім аргументом командного рядка.



Вивід не відрізняється від попереднього варіанту.

Приклад виводу, для (відсутнього) варіанту 7, із 4-ма потоками:

```
$ integrate_parallel 7 data.cfg 4
2998.59
1.40493
0.00046853
13500
```

---

– потоки, `fiber`, `thread pool` тощо. Також, можуть бути нюанси із налаштуванням потоків, наприклад, під WinAPI, функція `CreateThread()` створює потік, однак, не ініціалізує `runtime` стандартної бібліотеки C для роботи у ньому – на відміну від `_beginthreadex()`, тому навіть виклик `malloc()` може стати проблемою.

<sup>3</sup>Число 16 обране як вдвічі більше за очікувану максимальну кількість фізичних ядер на ноутбуках студентів.

<sup>4</sup>Якщо ваша система – Windows, буде автоматично додаватися розширення `.exe` –

### 3.2.2 Скрипт Python для автоматизованих запусків

В цілому, скрипт той же, що і в попередньому завданні, див. розділ 2.2.2, єдина відмінність – ще одним аргументом передається кількість потоків, з якою він буде викликати програму. Тому список аргументів такий:

- скільки раз потрібно розрахувати інтеграл кожної із функцій,
- яку кількість потоків для цього використовувати.



Для цього завдання особливо критичним є перевірка чи результати різних запусків співпадають. При тому, саме – в межах похибки, оскільки за різних розбиттів на потоки результати будуть трохи відрізнятися. Якщо результати відрізняються суттєво – це типова ознака наявності гонитви даних (race condition), фатальної помилки для таких паралельних програм.

### 3.2.3 Аналіз результатів запусків



- Користуючись описаним вище скриптом, побудуйте графік залежності часу виконання задачі від кількості потоків, для вашого комп'ютера.



- Якщо паралельна програма буде повільнішою за послідовну, практична не зараховується – потрібно буде доопрацювати. Зокрема – зверніться до методичних вказівок, розділ 3.2.6, за ідеями.
- **(Додано після дедлайну, не впливає на перевірку)** Побудувати графік коефіцієнту ефективності паралелізації<sup>5</sup>  $E(n)$ :

$$E(n) = \frac{S(n)}{n},$$

де  $S(n)$  – коефіцієнт прискорення:

$$S(n) = \frac{L(1)}{L(n)},$$

де, у свою чергу,  $L(1)$  – час послідовного виконання задачі,  $L(n)$  – час паралельного виконання із використанням  $n$  потоків.

---

це ОК, не потрібно якось турбуватися про це.

<sup>5</sup>Подробиці розглядалися на лекціях.

### 3.2.4 Додаткові завдання

- Дослідити залежність часу виконання від кількості потоків на різних комп'ютерах і з різними наборами параметрів. Порівняти результати та проаналізувати їх.
- Провести порівняльний аналіз використання різних підходів до синхронізації – `std::mutex`, `std::atomic` тощо.
- Оцінити втрати продуктивності, від захоплення м'ютекса на кожній ітерації внутрішніх циклів<sup>6</sup>, та від накопичення у атомарній змінній, видимій всім потокам (C++20 або новіший).

### 3.2.5 Що потрібно здавати

Залишаються вимоги, описані в розділі 2.2.6.

### 3.2.6 Методичні рекомендації

Релевантними є всі рекомендації щодо однопоточної реалізації, в розділі 2.2.7!

#### Загальні

- Для покращення відтворюваності – максимально залиште комп'ютер в спокої на час запусків.
- Врахуйте граничні випадки – наприклад, коли точок стільки ж, скільки є потоків, або більше. Їх не обов'язково реалізовувати, але потрібно опрацювати та повідомити про помилку. Користуйтеся здоровим глуздом!




#### C++

- Кількість ітерацій цілком може перевищувати максимальне значення 32-бітного `int`.
- ◇ А знакове переповнення – невизначена поведінка.

---

<sup>6</sup>Так робити принципово невірно, суть цього завдання – показати, чому.

**Чисельні методи**

- Обережно на краях інтервалів, що передаються потокам на опрацювання.
- ◇  Стежте, щоб не враховувати межу двічі – один раз потоком "ліворуч" від межі, інший раз – потоком "праворуч"; та щоб інтегрувати до самого кінця інтервалу – особливо це буде помітно для великої кількості потоків.
- ◇ Симптом такої помилки: помітна залежність результатів від кількості потоків.
- ◇ Невелика зміна результатів, (порядку величини похибки, скажімо,  $10^{-8}$ ), може бути викликана особливостями комп'ютерного представлення double, але помітно більша – ймовірно, наслідок тих чи інших алгоритмічних помилок.
- Результат може, в межах похибки змінюватися для запусків із різною кількістю потоків.
-  Однак, якщо різні запуски із тією ж конфігурацією, дають різні результати – це завжди помилка, і очевидна ознака race condition!
- Для адаптивного знаходження інтегралу ми, на кожній ітерації, збільшуємо кількість точок розбиття, по кожній із осей, вдвічі. Точність визначається величиною кроку по кожній із осей –  $dx$  і  $dy$ <sup>7</sup>.
- ◇ Якщо ви зафіксували кінцеву точність, то існують найбільші  $dx$  та  $dy$ , для яких вона досягається (для достатньо гладких функцій і нехтуючи похибкою заокруглення) – далі ділити немає сенсу.
- ◇ Тому, для багатопоточного, коду потрібно контролювати саме **повну кількість точок**.
- ◇  **Помилкою** є робити так: 1 потік – 1000 точок, 2 потоки – по 1000 на потік, всього 2000 і т.д.
- ◇◇ Якщо ви задаєте кількість точок на потік – то із двома потоками буде вдвічі більше точок, із 4-ма – в 4 рази більше і т.д.
- ◇◇ Прискорення не буде – буде потенційно вища точність за той самий


---

<sup>7</sup>Якщо не очевидно – намалюйте одномірний варіант і ще раз подумайте.

час. Але ми якраз фіксуємо бажану кінцеву точність.


- ◇ Це ще й ілюстрація різниці між законами Амдала і Густафсона.
- ◇ При тому, через похибки заокруглення, немає гарантії, що точність буде більшою.
- ◇ **Правильно:** 1 потік – 1000 точок, 2 потоки – 500 точок на потік, всього 1000, 3 потоки – 333 точки на потік, 999 всього (одна загублена – не проблема для таких обчислень, ну, або – врахуйте її в останньому потоці), 4 потоки – по 250 на потік і т.д.

### Багатопоточність

- Робота із багатопоточністю в C++ описана в презентації `cpp1_app_parallel_threads.pdf` з архіву лекції курсу C++: <https://drive.google.com/open?id=17ezFglCDm1djYIVX0LiKEkfJ2qcTyCIz>.
- Компілюючи багатопоточний код із використанням GCC, включіть підтримку C++14, а краще C++17<sup>8</sup>.
- ◇ Для вказання стандарту, потрібно в командному рядку передати `-std=c++14` або `-std=c++17`. GCC 11 по замовчуванню використовує C++17, GCC між 6.1 та 11 – C++14.
- ◇  Тому, варто CMake "попросити" безпосередньо обрати стандарт<sup>9</sup>, так:

```
set (CMAKE_CXX_STANDARD 17)
set (CMAKE_CXX_STANDARD 14) # Чи так.
```

І ніколи для цього не робити `SET(CMAKE_CXX_FLAGS...)`!

- ◇ Якщо в процесі розробки ви користуєтеся середовищем, що не підтримує CMake безпосередньо, зазвичай, вибрати стандарт можна десь в опціях.
-  Під час лінування багатопоточного коду в GCC та Clang слід додати опцію `-pthread` – інакше може бути ціла купа "undefined

<sup>8</sup>Мінімум – C++11, але давно це було, не вартує ним обмежуватися.


<sup>9</sup>Нагадуємо – явної маніпуляції прапорцями компілятора в `CMakeLists.txt` потрібно уникати.

reference” помилок під час лінкування<sup>10</sup>.

- ◇ CMake про потребу компілювати із підтримкою потоків повідомляємо так:


```
find_package(Threads REQUIRED)
target_link_libraries(my_app Threads::Threads)
```

- ◇ УВАГА! В Інтернеті є багато застарілих варіантів<sup>11</sup> – не користуйтеся ними!

-  Вартує ще раз повторити: якщо результат змінюється за різних запусків того ж виконавчого файлу, із тією ж конфігурацією, це помилка – вказівка на data race!
- До C++20, atomic для double<sup>12</sup> не підтримує операції +=. А операція + не буде атомарною:

```
std::atomic<double> shared_data;
shared_data += val; // Не компілюється (до C++20)
shared_data = shared_data + val; // Компілюється,
                                // але не є атомарним!
```


Тому, синхронізувати доступ – щоб не було data race, **потрібно іншими способами**.

- ◇ Для такої простої задачі, можна кожному потоку передати свій елемент масиву, і підсумувати вже після join() потоків.
- ◇ Або, в кінці обчислень кожного потоку, скористатися м'ютексом.
- ◇  Важливо, що обчислення мають максимально відбуватися без потреби в синхронізації – вона дорога, сповільнює роботу. Тому, помилкою буде, наприклад, накопичувати результат у змінній, видимій всім потокам, правильно – накопичувати у локальній змінній, і додавати її до загального результату вже після завершення циклу.

<sup>10</sup>Вони не завжди будуть – але опцію потрібно завжди додавати. Вона створює макроси, які керують генерацією потокобезпечного коду для стандартної бібліотеки та лінують із необхідною бібліотекою. Частина IDE додають цю опцію автоматично.

<sup>11</sup>Приклади описано тут: <https://stackoverflow.com/questions/1620918/cmake-and-libpthread/29871891#29871891>, зокрема – не робіть так: "target\_link\_libraries(my\_app "\${CMAKE\_THREAD\_LIBS\_INIT}")".

<sup>12</sup>Як і для float та long double.

-  Обов'язково перевірте вашу роботу за допомогою valgrind!
- ◇ Окрім перевірки роботи із пам'яттю, ця утиліта вміє виявляти проблеми багатопоточності.
- ◇ Це здійснюють інструменти Helgrind та DRD – задачі у них схожі, але підходи дещо відрізняються. Варто користуватися обома:

```
$ valgrind --tool=helgrind <progname> <program options>
<купа виводу>
```

```
$ valgrind --tool=drd <progname> <program options>
```

- ◇ На жаль, це багато складніша задача, тому велика кількість false positive і false negative – однак, якщо уважно аналізувати вивід, дуже і дуже допомагає!
- ◇ Офіційна документація доволі хороша: <https://valgrind.org/docs/>.
- Об'єкти `std::thread` можна зберігати в `std::vector`.
- Випробуйте для "складних" кількостей потоків – зокрема, для заданих простими числами – 3, 7, 13 тощо.
- ◇ За таких поділів може дещо гірше збігатися процес адаптивного інтегрування, але суттєве сповільнення або сильно відмінний результат вказуватимуть на помилку.
- В ідеальному варіанті варто, щоб і потік, де виконується `main`, теж виконував обчислення – не простоював. Хоча іноді це й не зручно, в цій задачі досягається легко і може бути корисним.
- В останньому потоці потрібно дораховувати до кінця інтервалу із файлу конфігурації – розрахунки виду  $x_0 + N\Delta x$  завжди вноситимуть певну похибку.



## Розділ 4

# Створення та використання потокобезпечної черги

### 4.1 Ціль роботи

Ціллю роботи є перше знайомство із методологією РСАМ та чергою – важливими примітивом паралельного програмування.

#### 4.1.1 Аналіз проблеми

Описаний раніше метод розпаралелення інтегрування доволі продуктивний, однак має недоліки:

1. Постійне створення та знищення потоків. Це не безкоштовно – потребує часу.
2. На кожній ітерації поділу інтервалів очікує завершення всіх потоків.
3. Задачі в потоках можуть виконуватися різний час<sup>1</sup>, тоді повне вирішення буде очікувати на завершення найповільнішого потоку.

Вирішення проблеми 3 підказує методологія РСАМ, яка вивчалася на лекціях. Проблему 1 допомагає вирішити потокобезпечна черга, див. розділ 4.1.2. В наступному завданні ми також розглянемо альтернативний підхід. Щодо проблеми, яка залишилася, то вона мінімізується із використанням РСАМ, але глобальне її вирішення потребує складніших підходів.

---

<sup>1</sup>З найрізноманітніших причин. Наприклад, через те, що на якихось інтервалах

Розглянемо можливе застосування<sup>2</sup> РСАМ до цієї задачі:

- Partitioning: Елементарною дією є обчислення інтегралу в околі однієї точки – фактично, обчислення функції разом з множенням на  $\Delta x \Delta y$ .
- Communication: Глобальна, виду всі-одному – підсумувати значення<sup>3</sup>.
- Agglomeration: Об'єднати обчислення інтегралу для кількох елементарних інтервалів.
- Mapping: Тут ми не будемо ускладнювати, потоки запускаються наперед і існують до кінця.

Розмір інтервалу після Agglomeration визначається компромісом між:

- Потребою масштабування<sup>4</sup> і співмірності розміру – кількість таких задач має залишатися достатньо великою, на порядок більшою ніж кількість потоків.
- Потребою ефективності – обмін даними не безкоштовний, занадто частий обмін буде сповільнювати виконання.
- Співмірністю задач за розміром.

Оціночно можна сказати – варто мати кількість інтервалів не меншою, ніж на порядок більше, за обчислювачів, але не більше, ніж на два порядки більшою. Далі потрібно експериментувати – це теж мистецтво. Тобто, якщо потоків 4, можна розпочати із 100 інтервалів<sup>5</sup> і десь на 1000 зупинитися.

---

інтегрування стає більше денормалізованих чисел.

<sup>2</sup>Методологія радить використовувати кількох різних спроб проектування паралельного алгоритму, але, заради лаконічності тут зупинимося на найбільш очевидному.

<sup>3</sup>Тут це дешева операція, тому помітного виграшу від розпаралелення не буде

<sup>4</sup>Тобто, здатності добре опрацьовувати великі кількості точок інтегрування. В ідеалі – щоб кількість задач після агломерації, (тут – інтервалів), лінійно зростала із зростанням кількості точок.

<sup>5</sup>Варто пам'ятати, що порядок – це десь між множенням на 10 (або й трохи менше) і на 100, інтуїтивно – ближче до першого, але, все ж мова не про рівно 10.

### 4.1.2 Використання потокобезпечної черги

Важливим примітивом, образно кажучи – структурним елементом паралельного програмування, є черга. Для використання із різних потоків одного процесу вона має бути потокобезпечною – організованою так, щоб звертання із різних потоків не спричиняло гонитви даних (data race).

Подробиці розглядаються на лекціях, наведемо тезово незвичні властивості таких черг – на додачу до безпечності використання із різних потоків:

- При спробі отримати елемент із порожньої черги, повинно відбутися очікування. На відміну від черг однопоточних програм – це не помилка, елемент може з'явитися пізніше.
- ◇ Це очікування не має потребувати процесорного часу – busy loop<sup>6</sup> неприпустимий.
- Тому, потрібен ще якийсь інструмент повідомляти потоку, що даних більше не буде. Ми виділяли два можливих підходи – кожен із своїми плюсами та мінусами:
- ◇ Додаткові методи, що повідомляють, чи є плани ще додавати дані в чергу. Можуть покладатися на ті чи інші (атомарні) змінні.
- ◇ Підхід "poison pill" – в чергу додається елемент, який вказує, що даних більше не буде. Часто це змушує потік завершитися – звідти й назва.

Можливим підходом<sup>7</sup> є наступний:

- Головний потік створює дві черги: одну для запитів потокам (який інтервал інтегрувати), іншу – з їх відповідями.
- Запускає потрібну кількість потоків, передаючи їм посилання на ці черги та копії чи константні посилання на спільну для них інформацію – наприклад, яку функцію потрібно обчислювати.
- Додає в чергу всі інтервали та накопичує відповіді.
- ◇ З міркувань ефективності, множення на  $\Delta x \Delta y$  варто виконувати у головному потоці.

---

<sup>6</sup>Принаймні – довгий.

<sup>7</sup>Одним із дуже багатьох, і певне – не оптимальним для цієї задачі, але дидактично корисним методом.

- ◇ Прийнятним тут є очікувати стільки відповідей, скільки було передано інтервалів<sup>8</sup>.
- Коли отримав всі відповіді, перевіряє, чи можна зупинитися – через досягнуту похибку або максимальну кількість ітерацій. Якщо ні – розсилає нові інтервали потокам.
- ◇ Тут втрачається невелика частка паралелізму, але при тому сильно спрощується код.
- Якщо потрібно зупинитися – розсилає всім потокам `poison pill`, наприклад – порожній інтервал, очікує, поки вони завершаться та виводить результат.

### 4.1.3 Розробка потокобезпечної черги

Є багато способів реалізувати потокобезпечну чергу, розроблених під потреби різних задач – із різними інженерними компромісами.

В цій роботі можна скористатися найпростішим підходом, котрий розглядався на лекції:





- Дані зберігаються у `std::deque`.
- Реалізовано методи `enqueue()` та `deque()`, та, можливо, інші.
- Доступ до черги захищено з допомогою `std::mutex`.
- ◇ Використовуючи RAII-обгортки `std::lock_guard` та `std::unique_lock`.
- Для нотифікації про наявність даних використовується умовна змінна `std::condition_variable`.
- ◇ Обов'язковим є врахування можливості марних прокидань – `spurious wakeups`.

Чергу потрібно оформити як шаблон класу, щоб використовувати ту ж реалізацію для різних типів даних.

---

<sup>8</sup>Але для більш загальних задач це може бути поганим способом.

## 4.2 Завдання

-  Написати програму паралельного обчислення інтегралу, з використанням підходу, описаного в розділі 4.1. Алгоритм програми описано в 4.1.2.
-  Для цього – реалізувати потокобезпечну чергу, згідно обговореного на парах.
- ◇ Див. презентацію `cpp1_app_parallel_threads.pdf`, частину про обговорення умовних змінних та відповідної глави книги Anthony Williams, “C++ Concurrency in Action”. Коротко ці матеріали підсумовано підсумовано в 4.1.3.
- Решта релевантних вимог з попередніх робіт залишаються. Зокрема, `data race` є абсолютно неприйнятними.
-  Добитися, в тому числі – підбором оптимального розміру інтервалу, щоб код був не повільнішим за попередню реалізацію<sup>9</sup>.
-  За допомогою скрипта (розділ 3.2.2) виконати запуски для кількості потоків<sup>10</sup>: 1, 2, ..., 16, 100, 1000.
- Порівняти часи, для однакових кількостей потоків, із результатами попередньої роботи 3.
- Проаналізувати отримані результати, згідно описаного в розділі 4.2.3.

### 4.2.1 Інтерфейс

Останнім аргументом командного рядка потрібно вказати кількість точок, що будуть обчислюватися одним потоком за один раз. У всьому решта, інтерфейс – той же, що і в попередній роботі, див. розділ 3.2.1.



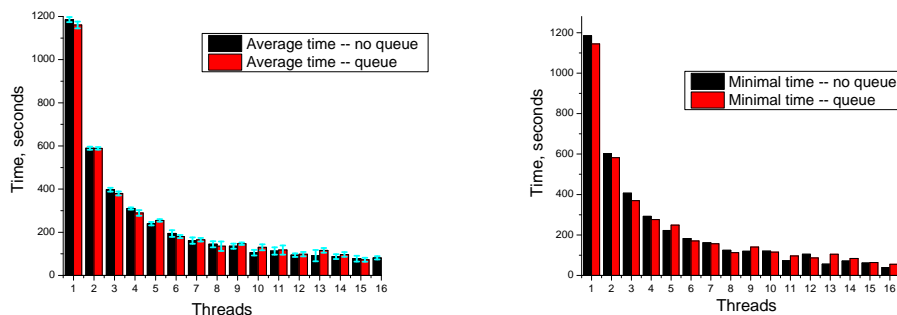
Виконавчий файл має називатися `integrate_parallel_queue`<sup>11</sup>.

---

<sup>9</sup>Досягнути в цій задачі прискорення важко, хоча й **можливо**, тому ми використовуємо її, в першу чергу, для відпрацювання методики, яку будемо використовувати для складніших задач.

<sup>10</sup>Число 16 обране як вдвічі більше за очікувану максимальну кількість фізичних ядер на ноутбуках студентів.

<sup>11</sup>Якщо ваша система – Windows, буде автоматично додаватися розширення `.exe` –



(a) Середні значення

(б) Мінімальні значення

Рис. 4.1: Приклад порівняння часу виконання. Ремарка: дані отримано на машині з 28 процесорами, вигляд ваших діаграм може відрізнятися.



Кількість точок, що будуть обчислюватися одним потоком за один раз передається четвертим аргументом командного рядка.



Вивід не відрізняється від попереднього варіанту.

Приклад командного рядка для (відсутнього) варіанту 7, із 4-ма потоками та 100 точок на одну задачу, що додається в чергу:

```
$ integrate_parallel_queue 7 data.cfg 4 100
```

### 4.2.2 Скрипт Python для автоматизованих запусків

В цілому, скрипт той же, що і в попередньому завданні, див. розділ 3.2.2, єдина відмінність – третім аргументом передається кількість точок, що будуть обчислюватися одним потоком за один раз. Тому список аргументів такий:

- скільки раз потрібно розрахувати інтеграл кожної із функцій,
- яку кількість потоків для цього використовувати,
- в якій кількості точок обчислювати за один раз.

### 4.2.3 Аналіз результатів запусків

- Визначити та занотувати оптимальний розмір інтервалу для однієї задачі, що передається в чергу – кількість точок, які обчислюватимуться потоком за раз.

- Використовуючи цей розмір<sup>12</sup>, проаналізувати різницю між часами виконання для кожної із кількості потоків у цій та попередній роботі.
- ◇ Для цього скориставшись методами перевірки гіпотез, які розглядалися в попередніх роботах.
- ◇ Також, для покращення інтерпретації результатів, зобразити результати на діаграмі. Приклад див. Рис. 4.1.
- Проаналізувати прискорення від розпаралелення в цій реалізації.
- ◇ Для цього побудувати графік коефіцієнту ефективності паралелізації<sup>13</sup>  $E(n)$ , використовуючи мінімальні часи виконання:

$$E(n) = \frac{S(n)}{n},$$

де  $S(n)$  – коефіцієнт прискорення:

$$S(n) = \frac{L(1)}{L(n)},$$

де, у свою чергу,  $L(1)$  – (мінімальний) час послідовного виконання задачі,  $L(n)$  – (мінімальний) час паралельного виконання із використанням  $n$  потоків.

- Пояснити отримані результати.

#### 4.2.4 Що потрібно здавати

Залишаються вимоги, описані в розділі 2.2.6.

#### 4.2.5 Методичні рекомендації

Будь ласка, користуйтеся і порадами з попередніх практичних, зокрема, 3.2.6 та 2.2.7 – тут лише доповнення.

---

це ОК, не потрібно якось турбуватися про це.

<sup>12</sup>А за потреби – уточнюючи його.

<sup>13</sup>Подробиці розглядалися на лекціях.

### Загальні

- Врахуйте граничні випадки – наприклад, коли точок чи задач стільки ж, скільки є потоків, або більше. Їх не обов’язково реалізовувати, але потрібно опрацювати та повідомити про помилку. Користуйтеся здоровим глуздом!
- Будьте акуратними! Помилки на границях областей інтегрування тут будуть проявлятися особливо гостро, оскільки задача розбита на багато менших і границь багато.

### C++

- Інтервал зручно задавати у вигляді структури, що містить всі необхідні величини:  $x_{0,i}$ ,  $y_{0,i}$  – початкові точки для задачі, що передається потоку,  $\Delta x$ ,  $\Delta y$ , та кількість точок по кожній із осей:  $N_x$ ,  $N_y$ .
- З командного рядка передається значення добутку:  $N_x \cdot N_y$ , тому потрібно забезпечувати, щоб  $N_x$  та  $N_y$  були його дільниками.
- ◇ Враховуючи, що одна задача для потоку – лише обчислення функції у певній кількості точок, зручно вважати, що  $N_y = 1$ .
- Цю структуру можна передавати (за допомогою черги) по значенню. Вона відносно велика, але проблемою тут це не буде.
- ◇ Альтернативи, типу використання динамічної пам’яті із розумними вказівниками чи й без, тут будуть надміру громіздкими<sup>14</sup>, тому категорично не рекомендуються.

#### 4.2.6 Додаткові завдання

- Визначте граничну кількість потоків, для якої ще відбувається прискорення – і граничну ефективність паралелізації.
- Реалізуйте альтернативний підхід до реалізації – нехай кожен потік ітерує свою задачу до досягнення вказаної похибки.
- ◇ Запропонуйте вибір методу для обчислення похибки, якою буде керуватися кожен із потоків.

---

<sup>14</sup>В майбутньому ми будемо стикатися із задачами, де саме використання динамічної пам’яті буде оптимальним.



- ◇ Якщо потік буде ітерувати більше певної кількості раз – нехай він ділить свій інтервал на дві частини та передає в чергу для виконання на окремих потоках.
- Запропонуйте та реалізуйте метод так організувати обчислення, щоб головний потік не очікував на завершення інтегрування для певного розбиття, перш ніж запускати на виконання задачі для наступного розбиття.
- Додаткове завдання – користуючись законом Амдала, оцініть, яку часту задачі ви змогли розпаралелити.
- ◇ Для цього можна скористатися fitting-ом або регресійним аналізом, знаючи вигляд закону Амдала.

Кожен потік до кінця рахує.

- Дослідити залежність часу виконання від кількості потоків на різних комп'ютерах і з різними наборами параметрів. Порівняти результати та проаналізувати їх.
- Провести порівняльний аналіз використання різних підходів до синхронізації – `std::mutex`, `std::atomic` тощо.
- Оцінити втрати продуктивності, від захоплення м'ютекса на кожній ітерації внутрішніх циклів<sup>15</sup>, та від накопичення у атомарній змінній, видимій всім потокам (C++20 або новіший).

---

<sup>15</sup>Так робити принципово невірно, суть цього завдання – показати, чому.