

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту

«Технології паралельних обчислень. Курсова робота»

Тема: Алгоритм сортування вибором та його паралельна реалізація мовою Python

|  |  |
| --- | --- |
| **Керівник**:  ст. викл. А. Ю. Дифучин  «Допущено до захисту»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2024 р.  Захищено з оцінкою  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Члени комісії:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Виконавець**:  Музичук Віталій Андрійович  студент групи ІП-13  залікова книжка № ІП-1322  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  «29» квітня 2024 р.  Інна СТЕЦЕНКО  Антон ДИФУЧИН |

**Київ – 2024**

**ЗАВДАННЯ**

1. Виконати розробку паралельного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.
2. Виконати тестування алгоритму, що доводить коректність результатів обчислень.
3. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.
4. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше.
5. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

**АНОТАЦІЯ**

Пояснювальна записка до курсової роботи складається з: 5 розділів, містить 20 рисунків, 1 таблиць, 7 джерел.

Об’єкт дослідження: метод сортування масивів вибором.

Мета роботи: дослідити послідовну і відомі паралельні реалізації алгоритму сортування вибором; спроєктувати та реалізувати паралельний і послідовний алгоритми; дослідити швидкодію алгоритмів в залежності від довжини вхідного масиву, кількості розбиттів масиву; зробити порівняльний аналіз на основі цих даних; оцінити ефективність паралелізації у поставленій задачі.

КЛЮЧОВІ СЛОВА: АЛГОРИТМ СОРТУВАННЯ ВИБОРОМ, MULTIPROCESSING, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ.

ЗМІСТ

[ВСТУП 6](#_Toc165325436)

[1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ 7](#_Toc165325437)

[1.1 Опис відомих паралельних реалізацій 8](#_Toc165325438)

[2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ 10](#_Toc165325439)

[2.1 Проєктування послідовного алгоритму 10](#_Toc165325440)

[2.2 Реалізація послідовного алгоритму 12](#_Toc165325441)

[2.3 Тестування коректності й аналіз швидкодії послідовного алгоритму 16](#_Toc165325442)

[3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС 22](#_Toc165325443)

[4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ 23](#_Toc165325444)

[4.1 Проєктування паралельного алгоритму 23](#_Toc165325445)

[2.2 Реалізація паралельного алгоритму 25](#_Toc165325446)

[4.3 Тестування паралельного алгоритму 27](#_Toc165325447)

[5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ 29](#_Toc165325448)

[ВИСНОВКИ 37](#_Toc165325449)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 38](#_Toc165325450)

[ДОДАТКИ 39](#_Toc165325451)

[Додаток А. Лістинг коду послідовної реалізації алгоритму 39](#_Toc165325452)

[Додаток Б. Лістинг коду паралельних реалізацій алгоритму 42](#_Toc165325453)

[Додаток В. Лістинг коду методів тестування 45](#_Toc165325454)

# ВСТУП

Однією з основних проблем в математиці та комп’ютерних науках є сортування елементів у списку. На даний момент вже створено багато варіантів вирішення цього завдання відомих як алгоритми сортування. Деякі з цих алгоритмів є прості і спонтанні, такі як сортування бульбашкою або ж вибором. Інші, такі як quick sort, є складними, але дають супер швидкі результати.

Сортування є одним з найбільш значущих і добре вивчених предметних областей у інформатиці. Більшість першокласних відомих алгоритмів відомі нам за їхню ефективність простоту, ощадливе використання пам'яті та інше. Однак ці алгоритми не враховують особливості багатоядерних архітектур сучасних процесорів, що істотно впливають на продуктивність [1].

Звідси виникає необхідність впровадження нових методів оптимізації роботи цих алгоритмів за допомогою використання паралельних обчислень. Цей підхід дозволяє значно зменшити час роботи сортування для великої кількості даних й використовувати більше ресурсів комп’ютера.

Темою цієї роботи буде алгоритм сортування вибором. У рамках цього дослідження буде проведено огляд існуючих версій паралельного алгоритму сортування вибором, а також буде розроблено один з таких варіантів. Крім того, буде проведений аналіз ефективності паралельної версії алгоритму у порівнянні з послідовним алгоритмом при різних початкових даних. Отримані під час тестування дані про швидкодію будуть представлені у вигляді таблиць і графіків.

# 1 ОПИС АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

Сортування вибором (англ. Selection Sort) – це дуже простий для розуміння алгоритм сортування, який корисний при роботі з невеликими обсягами даних. Однак, так само як і в сортуванні бульбашкою [2], велика кількість вхідних даних сповільнює його роботу. Сортування вибором має одну перевагу над іншими алгоритмами – він дуже мало переміщує дані, що сортуються. Таким чином, якщо кожен елемент даних досить великий і його важко перезаписувати в інше місце, то сортування вибором може бути найшвидшим.

Концепція існуючого алгоритму сортування вибором полягає в тому, що він простий, не потребує додаткової пам’яті для виконання сортування і легко реалізується порівняно з іншими алгоритмами.

Основна ідея цього алгоритму дуже проста – список ділиться на дві основні логічні частини, відсортовану та невідсортовану частини. Далі ітеративно знаходиться мінімальне (максимальне) значення з невідсортованої частини й додається до відсортованої частину списку. Таким чином для кожної ітерації відсортована область збільшується. Процес припиняється, коли кількість невідсортованих елементів дорівнює одиниці. Окрема процедура вибору значення для переміщення до відсортованого списку забезпечує правильне визначення індексу мінімального або максимального значення в несортованій частині [3].

Згідно нотації Ландау (англ. Big «O» notation) асимптотична часова складність даного алгоритму в будь-яких випадках дорівнює . Це означає, що незалежно від попередньої ступені відсортованості масиву, що подається на вхід алгоритму, він завжди буде виконуватися не менше ніж за часу де n – розмір вхідного масиву.

## 1.1 Опис відомих паралельних реалізацій

Хоча сортування вибором доволі популярний та давно відомий алгоритм, варіантів його паралельної реалізації існує небагато. Все це пов’язано з тим, що даний алгоритм має доволі низький рівень паралелізації і задля створення паралельних програм його поєднують з іншими видами сортування, наприклад сортування злиттям.

Незважаючи на це, все таки вдалося знайти декілька паралельних варіантів даного алгоритму.

Перший з них – це MinMax Bidirectional Parallel Sort [4]. Даний алгоритм це вдосконалена версія традиційного Bidirectional Parallel Sort. Основна ідея цього алгоритму полягає в наступному: він ділить список на дві рівних частини, в яких паралельно шукаються мінімальні й максимальні значення для кожної частини. Потім обидва значення мінімуму та максимуму з кожної частини порівнюються, щоб визначити мінімум і максимум цілого масиву. Далі мінімальне значення ми ставимо на початок списку й позначаємо цю частину як відсортовану й так само розміщуємо максимальний елемент (тільки в кінці масиву). Алгоритм повторюється, поки весь список не буде позначений, як відсортований.

Другий варіант – це Recursive divide-and-conquer selection sort [5]. Як виходить з назви, цей алгоритм заснований на рекурсивному поділі вхідного списку. Цей алгоритм складається з 2 основних процедур SelectionSort та Partition. Процедура Partition розділяє вхіднйи масив A на дві половини, де найбільший елемент у першій половині менше або дорівнює найменшому елементу в другій половині. Після такого рекурсивного поділу викликаються SelectionSort для двох половин, щоб сортувати їх. Псевдокод даного алгоритму зображений на Рисунках 1.1 та 1.2.

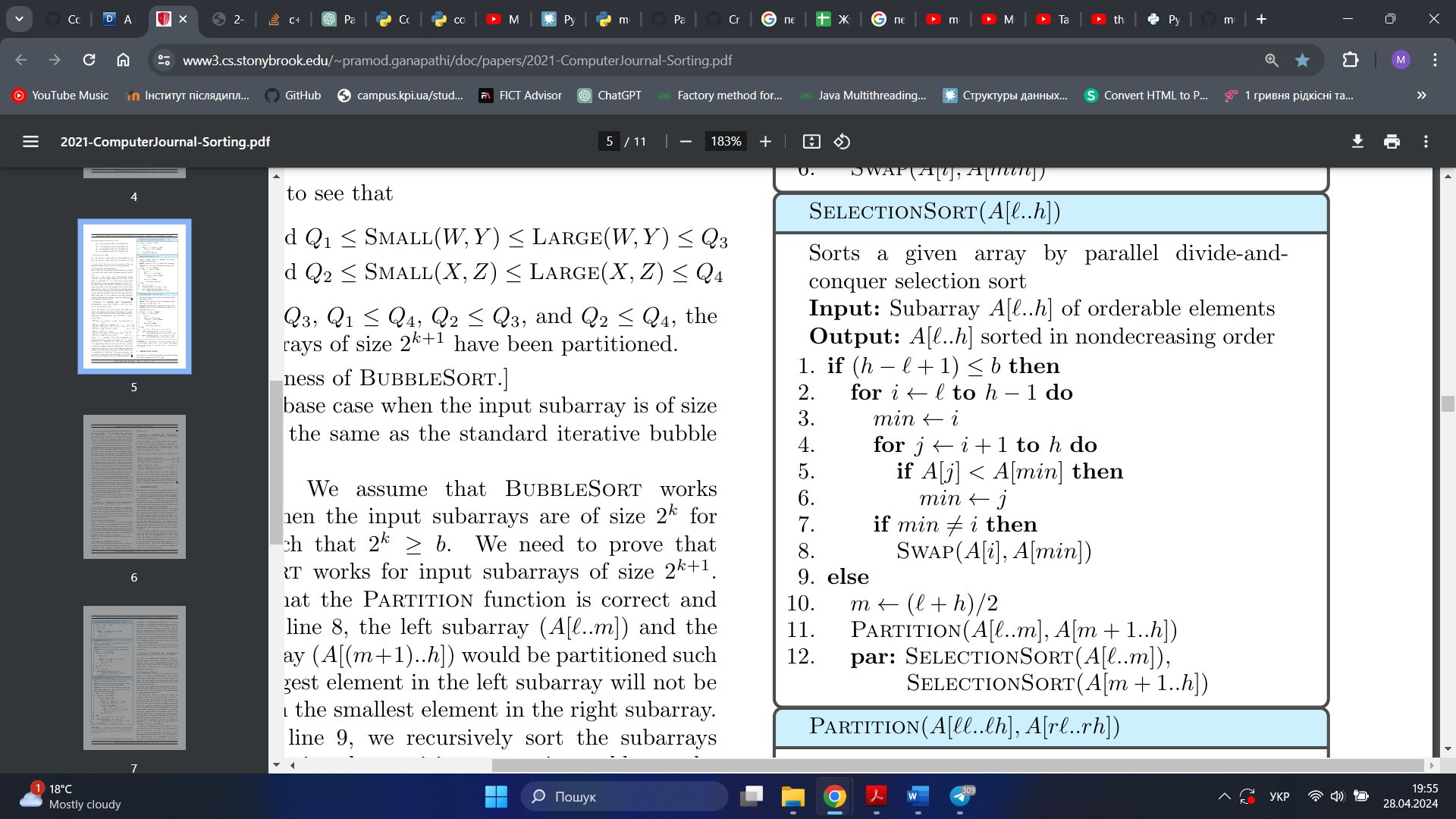


Рисунок 1.1 – Псевдокод процедури SelectionSort



Рисунок 1.2 – Псевдокод процедури Partition

# 2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

## 2.1 Проєктування послідовного алгоритму

Звичайну послідовну версію алгоритму сортвання вибором представимо за допомогою псевдокоду, де swap(a, b) – функція, що міняє місцями елементи в масиві:

**function** SelectionSort(array, size):

**Input:** array – input array for sorting, size – length of input array

**Output:** sorted array

**Start**

**for** i = 0 **to** size-1 **do**

min\_index = i

**for** j = i + 1 **to** size **do**

**if** array[j] < array[min\_index] **then**

min\_index = j

**if** min\_index != i **then**

**Swap**(array[i], array[min\_index])

**return** array

**End**

Оскільки паралельна версія даного алгоритму буде поєднанням сортування вибором та злиттям, то задля відповідності завданню курсової роботи й правильному порівнянню роботи паралельної та послідовної версії циї алгоритмів, необхідно імплементувати також алгоритм злиття в послідовну версію цього алгоритму. Послідовне поєднання сортування вибором та злиттям представлено за наступним псевдокодом, де функція partition(array, n\_split) виконує розбиття вхідного масиву на задану кількість підмасивів

**function** SelectionMergeSort(array, size, n\_split):

**Input:** array – input array for sorting, size – length of input array, n\_split – the number of partitions of the initial algorithm

**Output:** sorted array

**Start**

**if** size % n\_split != 0 **then**

**return**

sub\_arrays = **Partition**(araray, n\_split)

sub\_array\_size = size / n\_split

**for** **each** sub\_array **in** sub\_arrays **do**

**SelectionSort**(sub\_array, sub\_array\_size)

**return MergeArrays**(sub\_arrays)

**End**

Наступним опишемо функції, що відповідають за злиття попередньо відсортованих масивів.

**function** MergeArrays(arrays, size):

**Input:** arrays – input array of sorted subarrays, size – number of subarrays

**Output:** one merged sorted array

**Start**

**if** size == 0 **then**

**return** []

**if** size == 1 **then**

**return** arrays[0]

**if** size == 2 **then**

**return Merge**(arrays[0], arrays[1])

mid = size / 2

left\_half = **MergeArrays**(arrays[0:mid])

right\_half = **MergeArrays**(arrays[mid+1:size-1])

**return Merge**(left\_half, right\_half)

**End**

**function** Merge(array1, array2):

**Input:** array1, array2 – left and right subarray for merging

**Output:** one merged sorted array

**Start**

result = []

i = 0

j = 0

**while** i < len(array1) **and** j < len(array2) **do**

**if** array1[i] < array2[j] **then**

result.add(array1[i])

i = i + 1

**else**

result.add(array2[j])

j = j + 1

**while** i < len(array1) **do**

result.add(array1[i])

i = i + 1

**while** j < len(array2) **do**

result.add(array2[j])

j = j + 1

**return** result

**End**

## 2.2 Реалізація послідовного алгоритму

Отже для реалізації даного алгоритму спочатку необхідно визначити структуру об’єктів, які будуть сортуватися. Цими об’єктами будуть гвинтівки, які матимуть наступні характеристики:

* Калібр: від 5.6 до 12.7 мм з кроком 0.1 мм
* Довжина ствола: від 206.5 до 740.0 з кроком 0.1 мм
* Швидкість польоту кулі: від 310 до 900 м/с з кроком 1

Правила для порівняння даних об’єктів наступні:

1. Гвинтівка з більшим калібром вважається більшою, за гвинтівкою з меншим калібром
2. Якщо калібри гвинтівок однакові, тоді вважається більшою та, де добуток довжини ствола та швидкості польоту кулі буде більшим.

Дані об’єкти відображає наступний клас Rifle:

class Rifle(object):

"""

This class is an abstraction for automated and semi-automated rifles

with implemented equality methods

"""

def \_\_init\_\_(self, caliber=None, barrel\_length=None, bullet\_speed=None):

self.\_caliber = caliber if caliber else randint(56, 127) / 10

self.\_barrel\_length = barrel\_length if barrel\_length else randrange(2065, 7400, 5) / 10

self.\_bullet\_speed = bullet\_speed if bullet\_speed else randrange(310, 900, 5)

def \_\_str\_\_(self):

return (f"Caliber: {self.\_caliber:4.1f} mm; Barrel length: {self.\_barrel\_length:5.1f} mm; "

f"Bullet speed: {self.\_bullet\_speed:3d} m/s")

def \_\_eq\_\_(self, other):

if isinstance(other, Rifle):

return (self.\_caliber == other.\_caliber and

self.\_bullet\_speed == other.\_bullet\_speed and

self.\_barrel\_length == other.\_barrel\_length)

return False

def \_\_lt\_\_(self, other):

if not isinstance(other, Rifle):

return False

if self.\_caliber == other.\_caliber:

return (self.\_bullet\_speed \* self.\_barrel\_length) < (other.\_bullet\_speed \* other.\_barrel\_length)

else:

return self.\_caliber < other.\_caliber

def \_\_le\_\_(self, other):

if not isinstance(other, Rifle):

return False

if self.\_caliber == other.\_caliber:

return (self.\_bullet\_speed \* self.\_barrel\_length) <= (other.\_bullet\_speed \* other.\_barrel\_length)

else:

return self.\_caliber <= other.\_caliber

def \_\_gt\_\_(self, other):

if not isinstance(other, Rifle):

return False

if self.\_caliber == other.\_caliber:

return (self.\_bullet\_speed \* self.\_barrel\_length) > (other.\_bullet\_speed \* other.\_barrel\_length)

else:

return self.\_caliber > other.\_caliber

Тепер перейдемо безпосередньо до реалізації самого алгоритму. Функції MergeArrays(arrays) та Merge(array1, array2) в коді реалізовані наступним чином.

def merge(arr1: list, arr2: list):

"""

:param arr1, arr2: left and right subarray for merging

:return: one merged sorted array

"""

result = []

i = j = 0

while i < len(arr1) and j < len(arr2):

if arr1[i] < arr2[j]:

result.append(arr1[i])

i += 1

else:

result.append(arr2[j])

j += 1

while i < len(arr1):

result.append(arr1[i])

i += 1

while j < len(arr2):

result.append(arr2[j])

j += 1

return result

def merge\_arrays(arrays):

"""

:param arrays: input array of sorted sub\_arrays

:return: one merged sorted array

"""

if len(arrays) == 0:

return []

if len(arrays) == 1:

return arrays[0]

if len(arrays) == 2:

return merge(arrays[0], arrays[1])

mid = len(arrays) // 2

left\_half = merge\_arrays(arrays[:mid])

right\_half = merge\_arrays(arrays[mid:])

return merge(left\_half, right\_half)

Функція SelectionSort, що відповідає за класичний алгоритм сортування в точності повторює псевдокод, проте тут відсутній параметр довжини масиву, оскільки він визначається в коді через властивості переданого масиву.

def selection\_iterative\_sort(array: list):

"""

Classic iterative version of selection sort

:param array: input array for sorting

:return: sorted array

"""

size = len(array)

for i in range(size - 1):

min\_index = i

for j in range(i + 1, size):

if array[j] < array[min\_index]:

min\_index = j

if min\_index != i:

array[i], array[min\_index] = array[min\_index], array[i]

return array

Звідси кінцева версія SelectionMergeSort буде виглядати наступним чином

def sort(array, n\_split):

if len(array) % n\_split != 0:

print("It is not possible to split into so many subarrays because they will have different size")

sys.exit(1)

sub\_arrays = np.array(array).reshape(n\_split, -1)

for sub\_array in sub\_arrays:

selection\_iterative\_sort(sub\_array)

return merge\_arrays(sub\_arrays)

## 2.3 Тестування коректності й аналіз швидкодії послідовного алгоритму

Перевірити коректність роботи як послідовного так і паралельного алгоритму доволі просто – необхідно просто переконатися в тому, що кожен наступний елемент у вихідному масиві більший за попередній. Для цього побудуємо просту функцію, що робитиме це.

def check\_array(array: list[Rifle]) -> bool:

"""

Method checks if array are in ascending order

:param array:

:return:

"""

for i in range(len(array) - 1):

if array[i + 1] < array[i]:

return False

return True

Також для забезпечення якнайточнішого заміру часу виконання алгоритму проведемо попередній розігрів процеору, що довзолить забезпечити більш стабільні умови для вимірювань, що зробить результати більш об'єктивними та точними.

Будемо досліджувати вплив розміру вхідних даних та кількості розбиттів вхідного масиву на підмасиви.

Спочатку прослідкуємо за часом роботи алгоритму при збільшенні розмірів вхідних даних, але зі сталою кількістю підмасивів для розбиття. Заміри будемо проводити для масивів з розміром 1000, 2500, 5000, 1000, 15000, 20000 та 25000 елементів, які будуть заповненні випадково створеними об’єктами гвинтівок. Алгоритм буде виконуватися по 10 разів, з яких буде взято середнє значення часу його роботи. З кожною ітерацією алгоритму також буде перевірятися правильність сортування масиву. Для реалізації цього тесту напишемо наступну функцію:

def test\_sort\_method(sort\_method: Callable, len\_array: int, progon\_number: int, n\_split=None) -> float:

"""

This method tests sorting algo for correctness and measure its average time

:param sort\_method: Sorting method for testing

:param len\_array: Length of test array

:param progon\_number: Number of testing repeating

:param n\_split: Number of array division

:return average\_time

"""

array = [Rifle() for i in range(len\_array)]

# List for collecting time for each of progon

exec\_time = list()

# Flag that will save the wrong sort error

correctness\_flag = True

for i in range(progon\_number):

arr = deepcopy(array)

# Start of measuring

start = time()

if n\_split:

arr = sort\_method(arr, n\_split)

else:

arr = sort\_method(arr)

# End of measuring

end = time()

exec\_time.append(end - start)

if not check\_array(arr):

correctness\_flag = False

print(f"- {i + 1} progon passed -")

print(f"Correctness of sorting: {correctness\_flag}")

avarage\_time = sum(exec\_time) / progon\_number

print("Average execution time:", avarage\_time, "seconds")

return avarage\_time

Результати тестування алгоритму для 5 поділів й різної кількості вхідних даних наведені на рисунках 2.1 та 2.2.

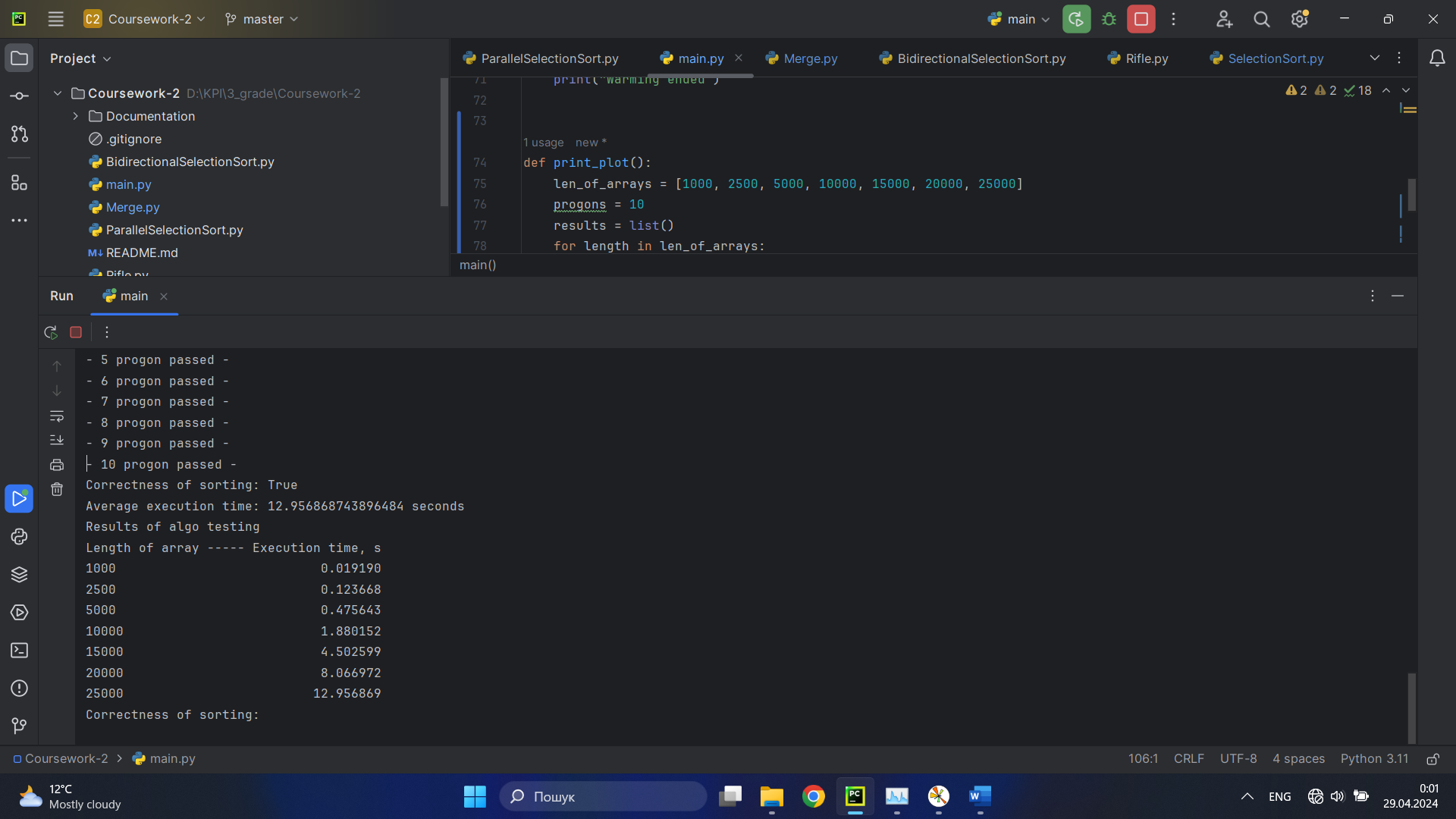


Рисунок 2.1 – Час виконання послідовного алгоритму при різній довжині масиву

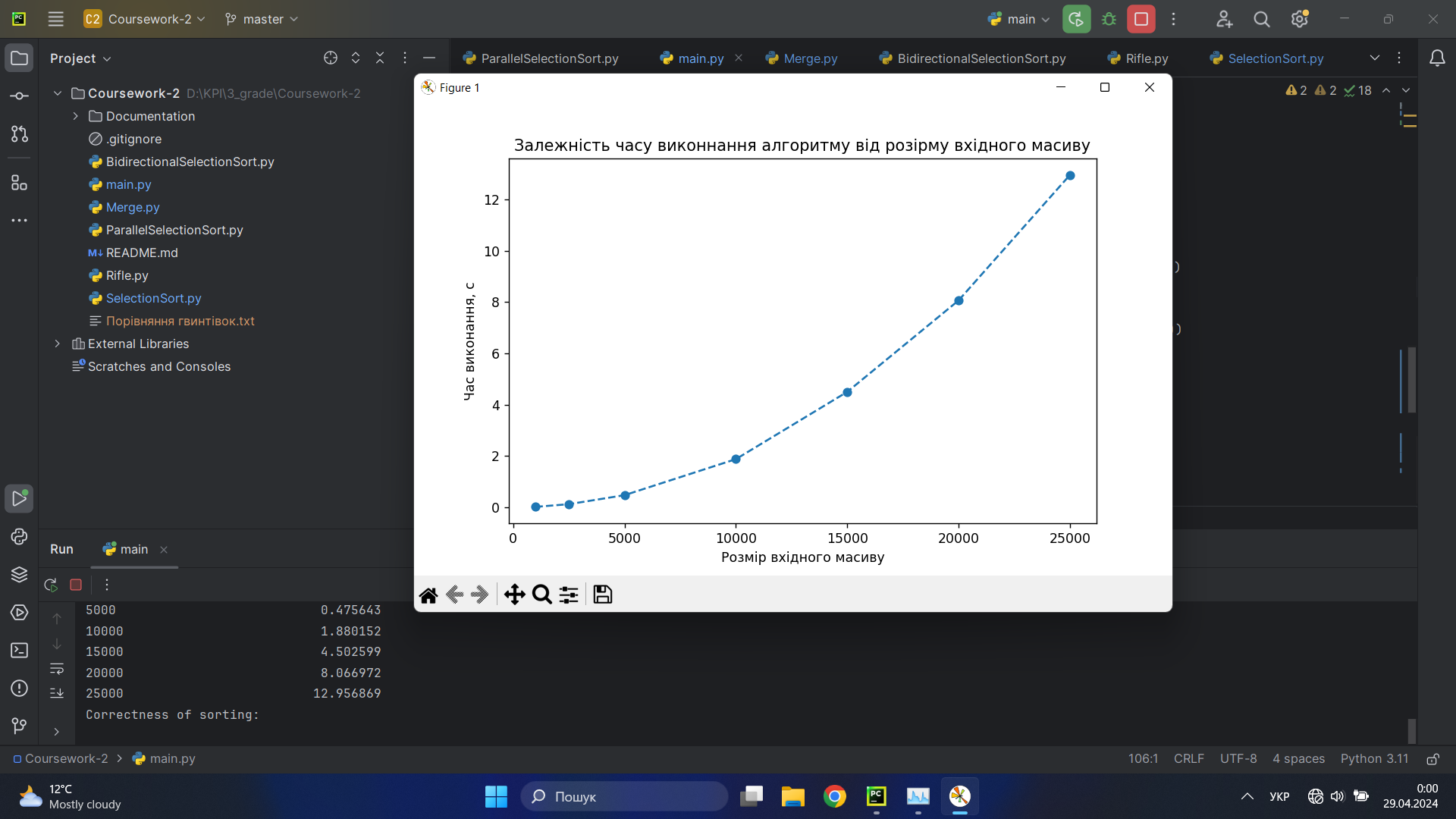


Рисунок 2.2 – Графік залежності часу виконання від розміру масиву

Як видно з обох малюнків, цілком справджується те, що асимптотична оцінка алгоритму дорівнює оскільки графік залежності візуально нагадує вітку параболи. Звідси також можна зробити висновок, що даний алгоритм краще використовувати лише при невеликих розмірах вхідного масиву.

Тепер проаналізуємо швидкодію алгоритму при зміні кількості розбиттів вхідного масиву. Для тестування будемо розбивати вхідний масив на таку кількість 1, 2, 5, 10, 20, 25, 50 рівних частин. Всі інші умови тестування не змінюються. Розмір вхідного масиву буде сталий і дорівнюватиме 10000. Результати тестування можна побачити на рисунках 2.3 та 2.4.

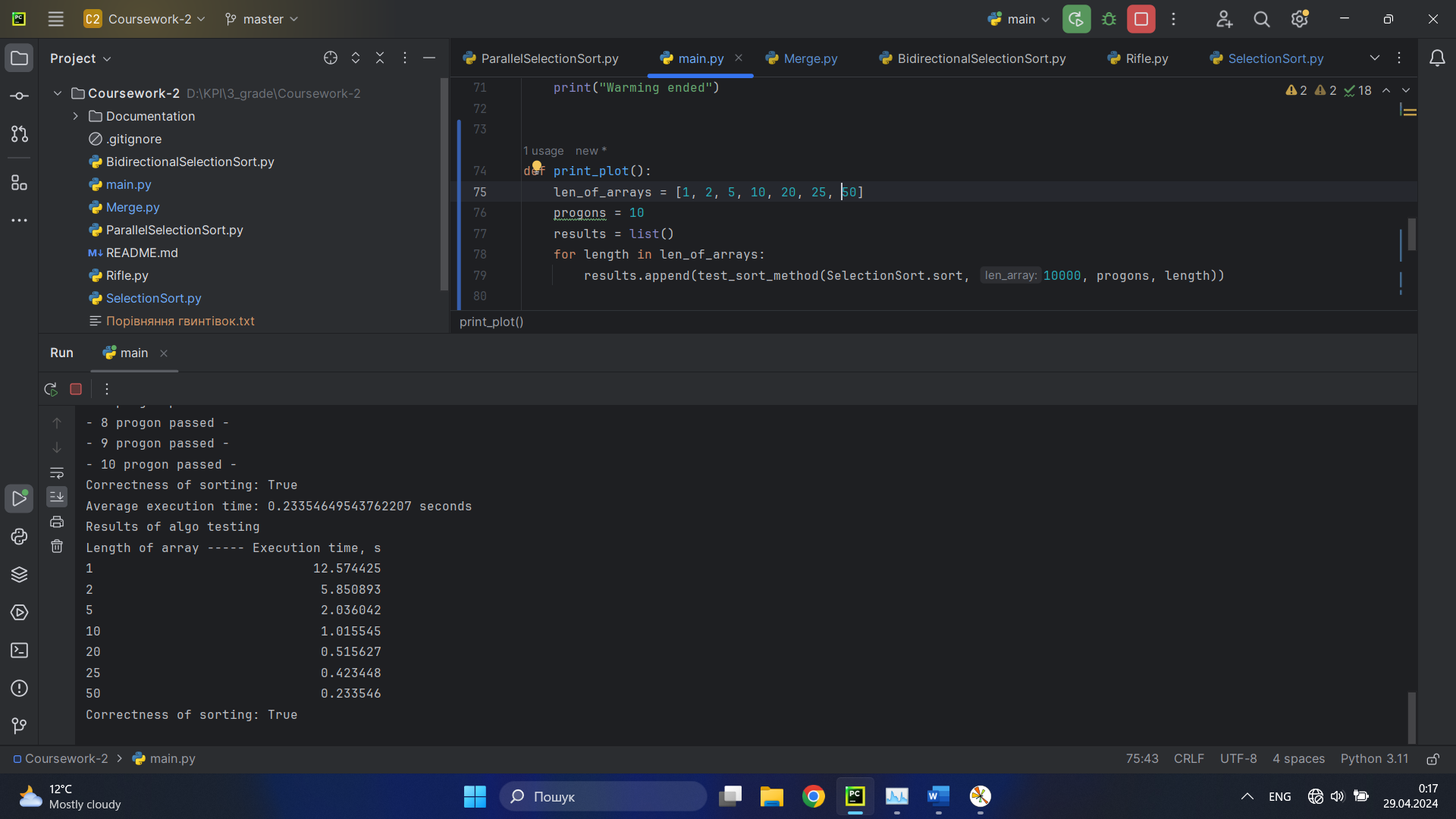


Рисунок 2.3 - Час виконання послідовного алгоритму при різній кількості розбиттів масиву

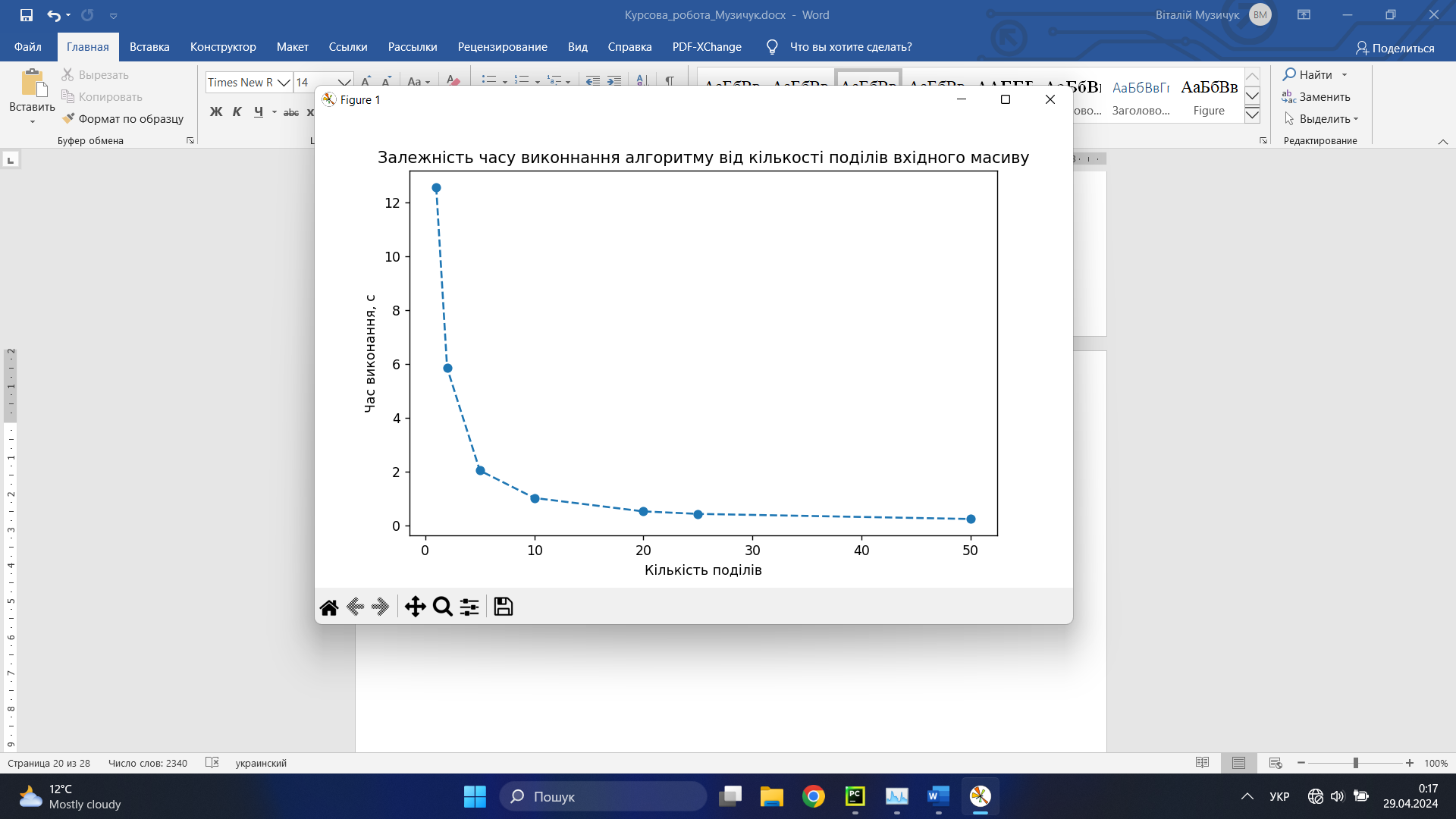


Рисунок 2.4 – Графік залежності часу виконання від кількості розбиттів

Бачимо, що при збільшені кількості розбиттів початкового масиву швидкість виконання алгоритму зменшується, проте більше 20 розбиттів встановлювати не варто, оскільки це особливо не покращує роботу алгоритму.

В обох тестуваннях паралельно перевірялася коректність вихідних масивів й помилок у сортуванні не було виявлено.

# 3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

У рамках даної роботи для реалізації паралельного алгоритму буде використовуватися мова програмування Python та її модуль multiprocessing [6], який дозволить побудувати паралельні програми.

Через присутність в Python GIL (Global Interpreter Lock) [7], який обмежує виконання Python коду в одному потоці за раз, навіть якщо система має багатоядерний процесор, використання модулю multithreading є недоцільним.

Враховуючи це використання модулю multiprocessing, де кожен процес має свій власний інтерпретатор Python, може бути більш ефективним рішенням. В цілому через присутність даного механізму використання багатопоточності у Python є недоцільним.

Також не менш важливою особливістю Python є те, що підпроцеси можуть запускатися лише з головного процесу, а це значить, що реалізація рекурсивних методів сортування, таких як Recursive divide-and-conquer selection sort, що описаний в п. 1.1, стає неможливою через це обмеження. Ще одним недоліком multiprocessing є неможливість передавати об’єкт пулу, як параметр методу, що теж частково відсікає деякі варіанти реалізації подібних алгоритмів.

Для розробки паралельної версії алгоритму сортування вибором буде використано клас Pool(). Цей клас представляє пул робочих процесів, що виконуються паралельно. Він має методи, які дозволяють завантажувати процеси до пулу.

# 4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЄКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

## 4.1 Проєктування паралельного алгоритму

Враховуючи обмеження розробки програм в модулі multiplocessing спробуємо реалізувати 2 паралельні версії алгоритму сортування вставками. Перша версія – це MinMax Bidirectional Parallel Sort, що описаний у п. 1.1. Для цього створимо фукнцію, що буде створювати пул потоків, за допомогою класу Pool, який прийматиме два процеси, які паралельно будуть шукати мінімальний та максимальний елементи в першій та другій половині масиву. Після цього визначаємо, загальний мінімум й максимум масиву й переміщуємо їх на початок (кінець) відсортованої частини масиву. Процеси будуть паралельно виконувати функцію **find\_min\_max**(array, start, end), яка повертатиме max та min елемент у заданому діапазоні. Детальніше псевдокод алгоритму наведений нище.

**function** find\_min\_max(array, start, end):

**Input:**  array – input array, start – start of subarray, end – end of subarray

**Output:** indexes of min and max elements

**Start**

min\_index = start

max\_index = start

**for** i = start+1 **to** end **do**

**if** array[i] < array[min\_index] **then**

min\_index = i

**if** array[i] > array[max\_index] **then**

max\_index = i

**return** min\_index, max\_index

**End**

**function** BidirectionalParallelSort(array, size):

**Input:**  array – input array, size – length of array

**Output:** sotred input array

**Start**

left = 0

right = size

middle = (left + right) // 2

**while** left < middle – 1 **do**

**do in parallel:**

left\_min\_idx, left\_max\_idx = **find\_min\_max**(array, left, middle)

right\_min\_idx, right\_max\_idx =**find\_min\_max**(array, middle, right)

**if** array[left\_min\_idx] <= arr[right\_min\_idx] **then**

min\_idx = left\_min\_idx

**else**

min\_idx = right\_min\_idx

**swap**(array[left], array[min\_idx])

**if** array[left\_max\_idx] <= arr[right\_max\_idx] **then**

max\_idx = right\_max\_idx

**else**

max\_idx = left\_max\_idx

**swap**(array[left], array[min\_idx])

left = left + 1

right = right -1

**if** array[middle-1] > array[middle] **then**

**swap**(array[middle-1], array[middle])

**return** array

**End**

Друга можлива реалізація паралельної версії даного алгоритму (ParallelSelectionMerge) буде полягати у наступному – вхідний масив array розбивається на n підмасивів і далі кожен підмасив паралельно сортується звичайним алгоритмом вибору. Після цього відбувається злиття всіх частин за допомогою функції **MergeArrays**(arrays), псевдокод якої також використовується в комбінованій версії послідовного алгоритму, що описаний в п. 2.1. Детальний паралельної версії наведений нище.

**function** ParallelSort (array, size, n\_split):

**Input:**  array – input array, n\_split – number of division of initial array

**Output:** sorted initial array

**Start**

**if** size % n\_split != 0 **then**

**return**

subarrays = **Partition**(array, n\_split)

**do in parallel:**

**for each** subarray **in** subarrays **do**

**SelectionSort**(array, size / n\_split)

**return** MergeArrays(subarrays)

**End**

В даному псевдокоді функція partition(array, n\_split) виконує розбиття вхідного масиву на задану кількість підмасивів, а SelectionSort(array, size) виконує звичайне сортування вибором.

## 2.2 Реалізація паралельного алгоритму

Для реалізації обох версій паралельного алгоритму скористаємося класом Pool, що створює пул паралельних процесів. Функція MinMax Bidirectional Parallel Sort буде містити пул потоків з 2 процесами, оскільки ми проводимо розбиття на 2 однакових частини, тому створення більше чи менше процесів ніж 2 у пулі буде зайвим. Процесу до пулу завантажуються через метод, apply\_async() Код реалізації наведений нище

def find\_min\_max(arr, start, end):

min\_index = start

max\_index = start

for i in range(start + 1, end):

if arr[i] < arr[min\_index]:

min\_index = i

elif arr[i] > arr[max\_index]:

max\_index = i

return min\_index, max\_index

def MinMaxBidirectional(arr):

size = len(arr)

left = 0

right = size

middle = (left + right) // 2

with mp.Pool(processes=2) as pool:

while left < middle-1:

t1 = pool.apply\_async(find\_min\_max, (arr, left, middle))

t2 = pool.apply\_async(find\_min\_max, (arr, middle, right))

left\_min\_idx, left\_max\_idx = t1.get()

right\_min\_idx, right\_max\_idx = t2.get()

if arr[left\_min\_idx] <= arr[right\_min\_idx]:

min\_idx = left\_min\_idx

else:

min\_idx = right\_min\_idx

arr[left], arr[min\_idx] = arr[min\_idx], arr[left]

if arr[left\_max\_idx] <= arr[right\_max\_idx]:

max\_idx = right\_max\_idx

else:

max\_idx = left\_max\_idx

arr[right - 1], arr[max\_idx] = arr[max\_idx], arr[right - 1]

left += 1

right -= 1

if arr[middle-1] > arr[middle]:

arr[middle-1], arr[middle] = arr[middle], arr[middle-1]

return arr

Другий спосіб сортування з розділенням масиву на n-ну кількість рівних підмасивів також буде використовувати пул процесів, проте завантаження процесів буде відбуватися через метод map(). Також не менш важливо зауважити, що пул створюється з такою кількістю процесів, що відповідають кількості підзадач з сортування, які будуть виконуватися паралельно.

def ParallelSelectionMerge(array: list, n\_split: int):

"""

:param array: input array

:param n\_split: number of division of initial array

:return:

"""

if len(array) % n\_split != 0:

print("It is not possible to split into so many subarrays because they will have different size")

sys.exit(1)

sub\_arrays = np.array(array).reshape(n\_split, -1)

pool = mp.Pool(n\_split)

res = pool.map(selection\_iterative\_sort, sub\_arrays)

pool.close()

pool.join()

return merge\_arrays(res)

## 4.3 Тестування паралельного алгоритму

Отже, тестування даних алгоритмів на коректність розрахунків не буде відрізнятися від тестування послідовного. Використаємо ту ж саму функцію, яка буде проганяти алгоритми n-ну кількість разів та перевіряти чи вихідні масиви відсортовані у порядку зростання. На вхід подається випадково згенерований масив з об’єктами гвинтівок. Результати тестування можна побачити на рисунках 4.1 та 4.2.

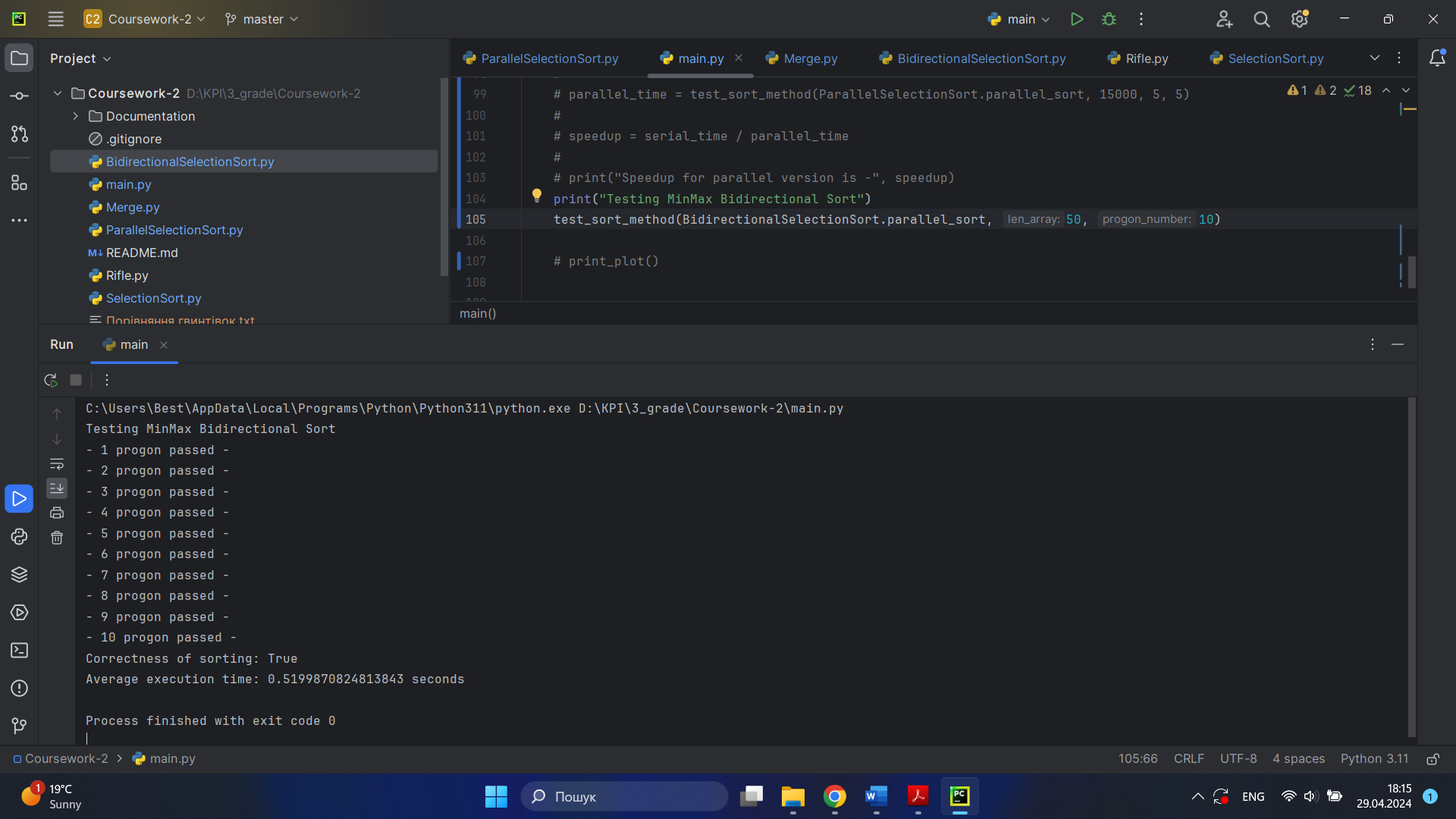


Рисунок 4.1 – Тестування першого варіанту паралельної реалізації

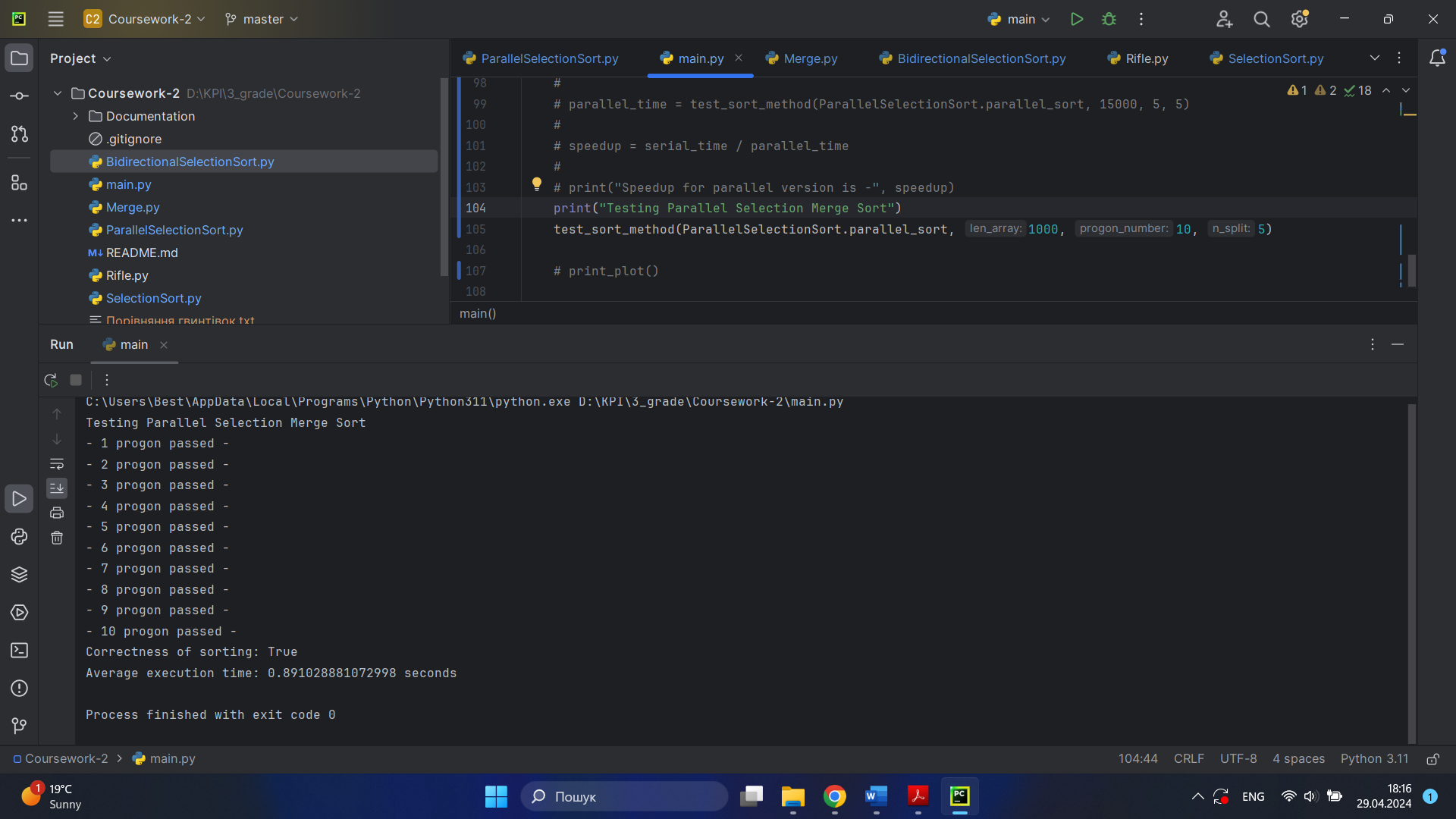


Рисунок 4.2 – Тестування другого варіанту паралельної реалізації

# 5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Дослідимо ефективність паралельних алгоритмів сортування вибором відносно їх послідовної версії. Для цього будемо досліджувати, як впливають такі параметри як розмір вхідного масиву та кількість поділів на прискорення паралельної версії відносно послідовної.

Умови тестування залишаються незмінними – кожен алгоритм проганяється по 10 разів, з чого вираховується середній час. Розмірність вхідних масивів залишається тією ж – 1000, 2500, 5000, 10000, 15000, 20000, 25000. Всі заміри й дослідження проводилися на ноутбуці з процесором Intel Core-i5-1135G7 (4 ядра, 8 логічних процесорів) та 8ГБ оперативної пам’яті на операційній системі Windows 11.

Спочатку дослідимо доцільність застосування MinMax Bidirectional Sorting відносно послідовного SelectionMergeSort. В даному випадку будемо досліджувати лише вплив розмірності масиву на прискорення, оскільки паралельна реалізація розбиває масив на сталих 2 підмасиви, тому послідовна версія буде тестуватися лише за розбиття масиву на 2 частини, для забезпечення достовірності порівняння алгоритмів. Резульатати порівняння відображенні в таблиці 5.1 та на рисунках 5.1, 5.2, та 5.3.

Таблиця 5.1. – Назва таблиці

|  |  |  |  |
| --- | --- | --- | --- |
| Кількість елементів | Час послідовного алгоритму, секунд | Час паралельного алгоритму, секунд | Коефіцієнт прискорення |
| 1000 | 0.0477 | 1.9696 | 0.024 |
| 2500 | 0.2942 | 10.4211 | 0.028 |
| 5000 | 1.1871 | 42.4518 | 0.028 |
| 10000 | 5.1917 | 164.1675 | 0.032 |
| 15000 | 12.5296 | 216.9856 | 0.058 |
| 20000 | 23.0279 | 389.8977 | 0.059 |
| 25000 | 42.1590 | 432.3216 | 0.098 |

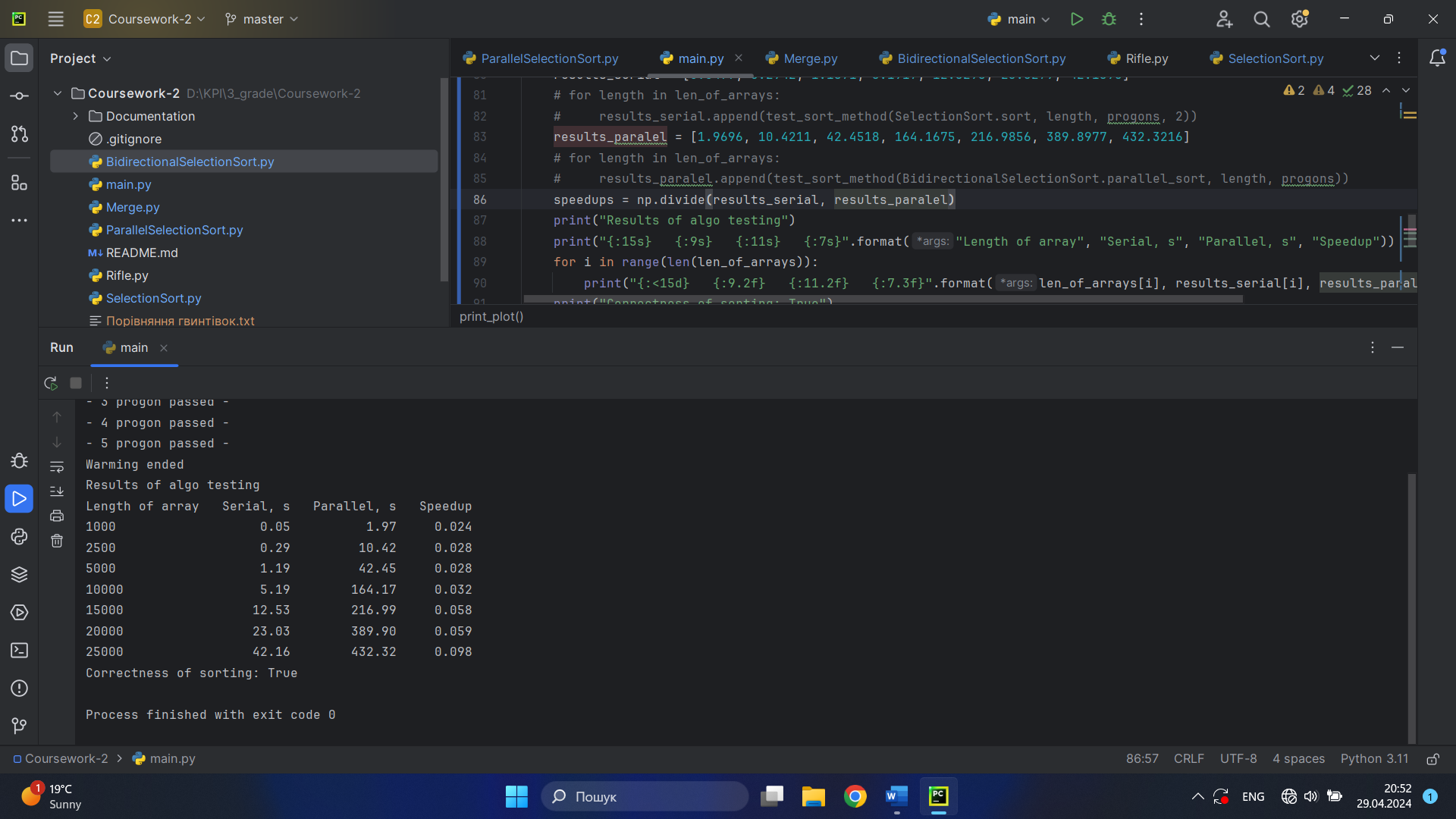


Рисунок 5.1 – Результати тестування MinMaxBidirectional у консолі

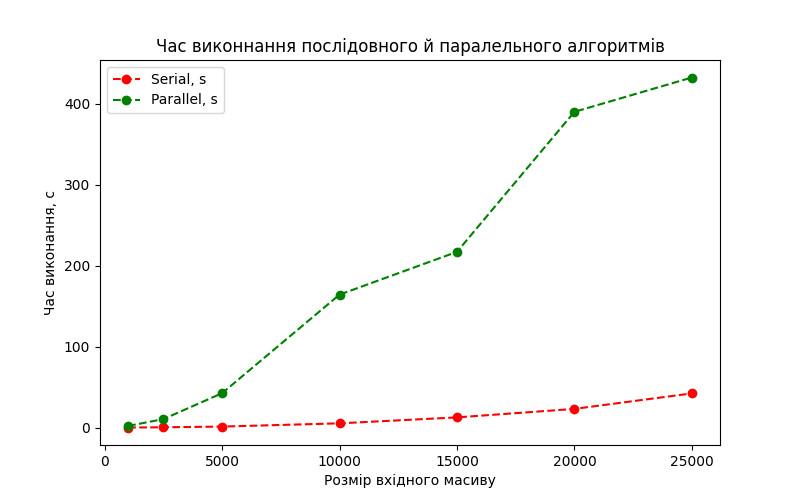


Рисунок 5.2 – Графік часу виконання послідовного й паралельного алгоритму

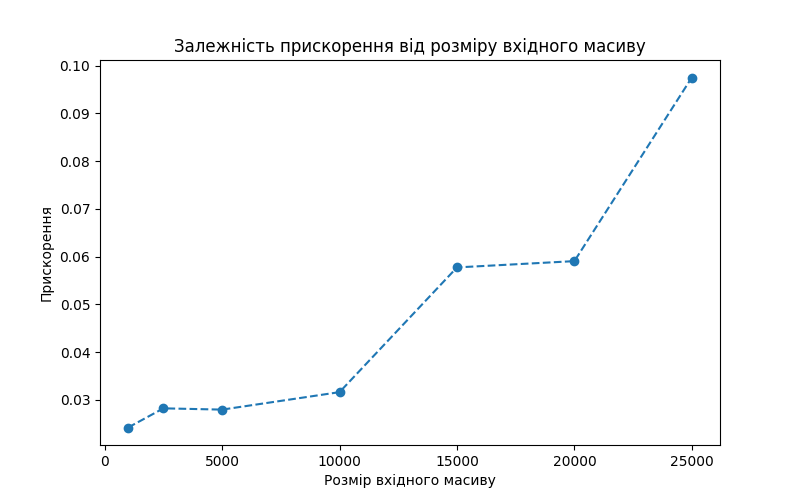


Рисунок 5.3 – Графік прискорення паралельного алгоритму

З наведених графіків та таблиць вище нескладно зрозуміти, що алгоритм MinMax Bidirectional Parallel Sorting дуже сильно поступається у швидкодії з його послідовною версією. Хоча на графіку прискорення можна спостерігати його ріст, проте це дуже низький показник для того, щоб сказати, що це алгоритм може бути швидшим для більших вхідних даних. Скоріш всього причиною такого колосально розкиду в часі роботи є те, що для кожної ітерації паралельного алгоритму в Python створюються 2 окремих процеси, що є дуже ресурсоємнісною операцією, оскільки вона вимагає створення окремого інтерпретатора, виділення пам’яті й копіювання даних до процесу. Можливо, якщо скористатися іншими засобами паралельного програмування даний алгоритм зможе досягнути істотного покращення, однак використання його у Python є абсолютно недоцільним за будь-яких умов.

Тепер проведемо тестування другого паралельного алгоритму ParallelSelectionMerge, для того щоб визначити його ефективність відносно аналогічного послідовного. У цьому випадку необхідно провести дослідження з визначення впливу 2 параметрів на ці алгоритми, а саме – довжина масиву та кількість його розбиттів. Спочатку розглянемо параметр довжини вхідного масиву.

Обидва алгоритми як послідовний, так і паралельний поєднані з сортуванням злиття. Попередньо тестувавши ефективність роботи послідовного алгоритму в п. 2.3 було встановлено, оптимальна кількість розбиттів для вхідного масиву становить 10, оскільки подальше збільшення цього параметру не дає значного прискорення. Спробуємо протестувати ці 2 алгоритми при сталій кількості розбиттів й поглянемо на результати.

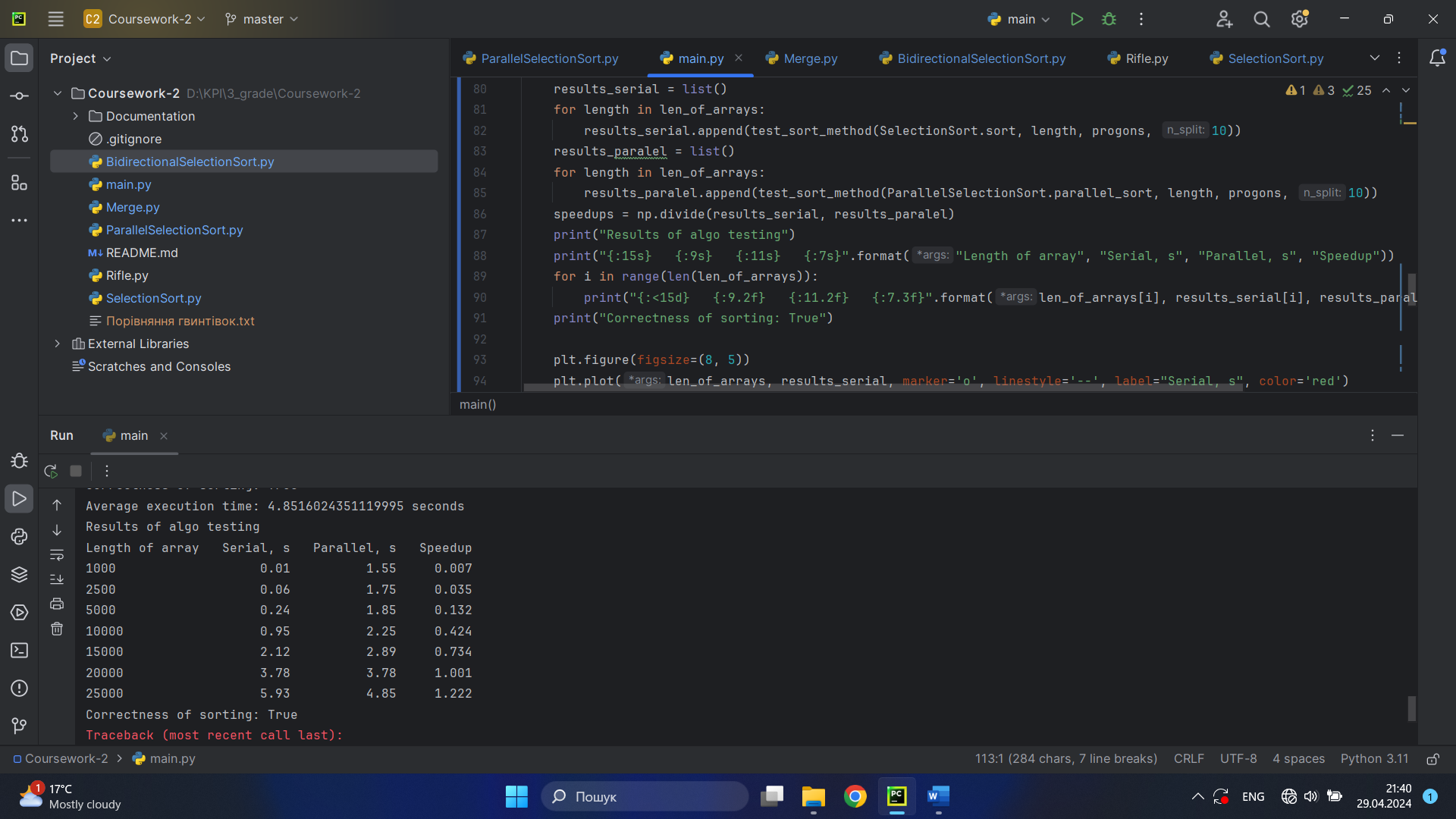


Рисунок 5.4 – Результати тестування алгоритму ParalleSelectionMerge

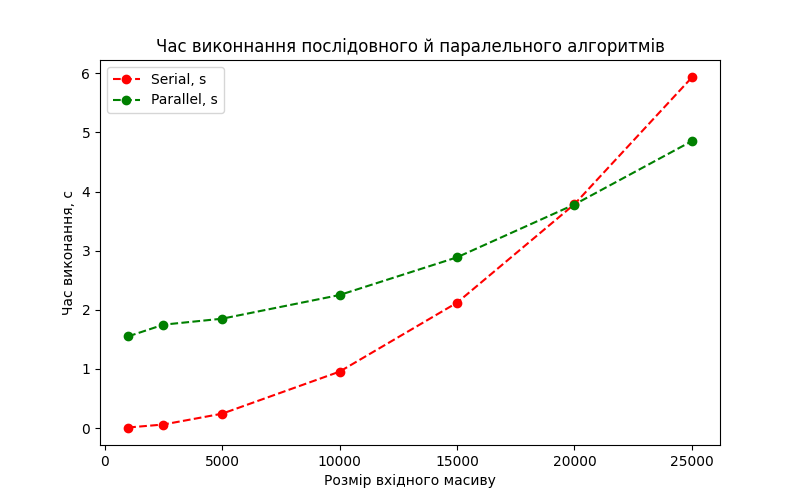


Рисунок 5.5 – Графік залежності часу від розміру масиву

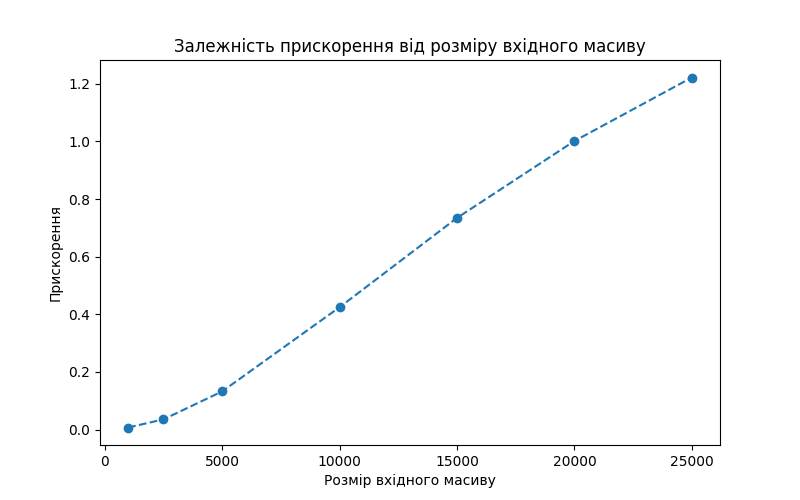


Рисунок 5.6 – Графік прискорення ParallelSelectionMerge

Загалом, як і очікувалося паралельний алгоритм показав кращий результат лише на доволі високих об’ємах данних, де прискорення стало більше 1.2. Проте результат можна покращити – спробуємо зменшити кількість розбиттів до 5, це зменшить використання ресурсів на створення процесів й дозволить досягнути більшого пришвидшення.

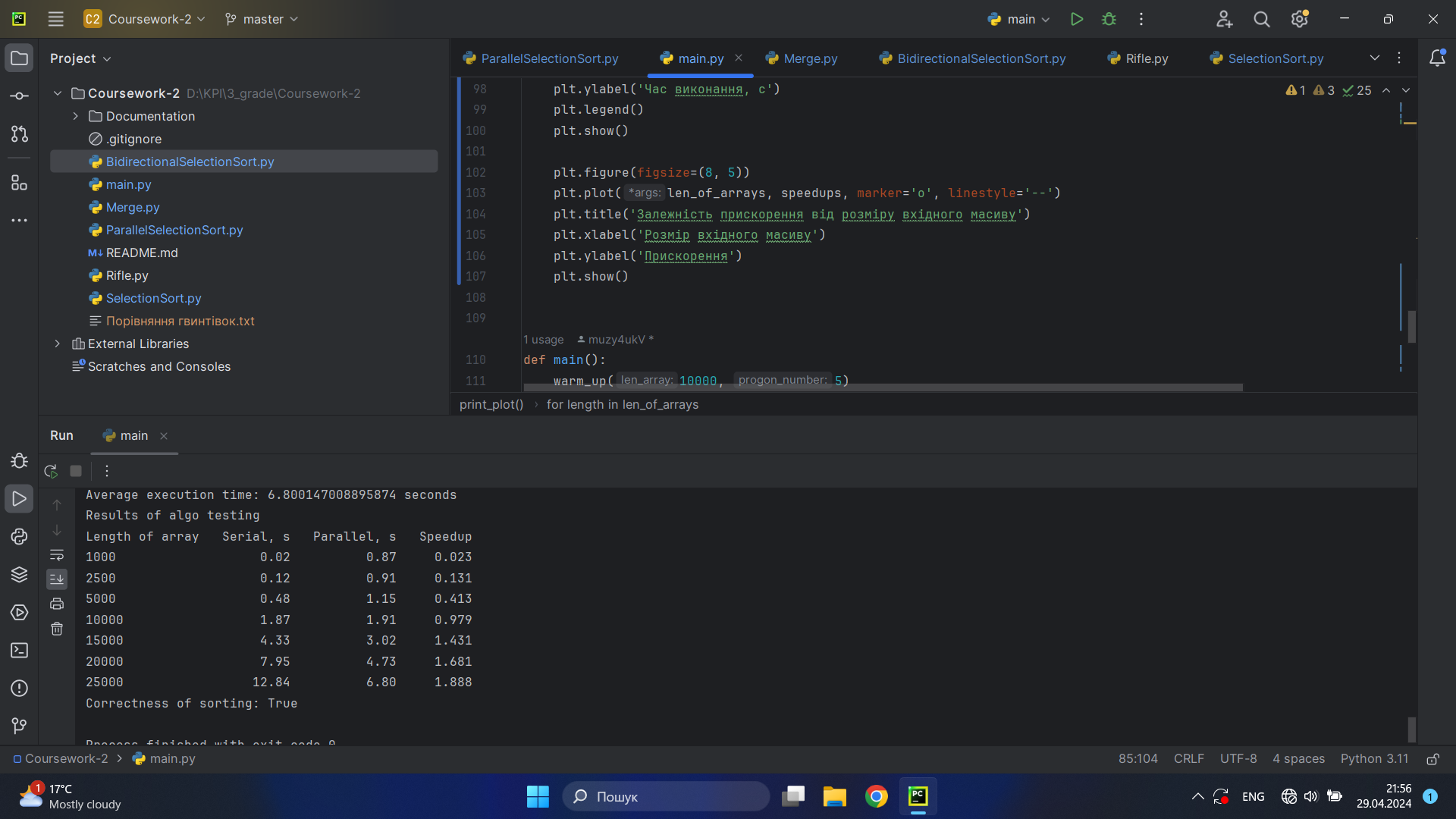


Рисунок 5.7 – Результат тестування ParallelSelectionMerge з n=5

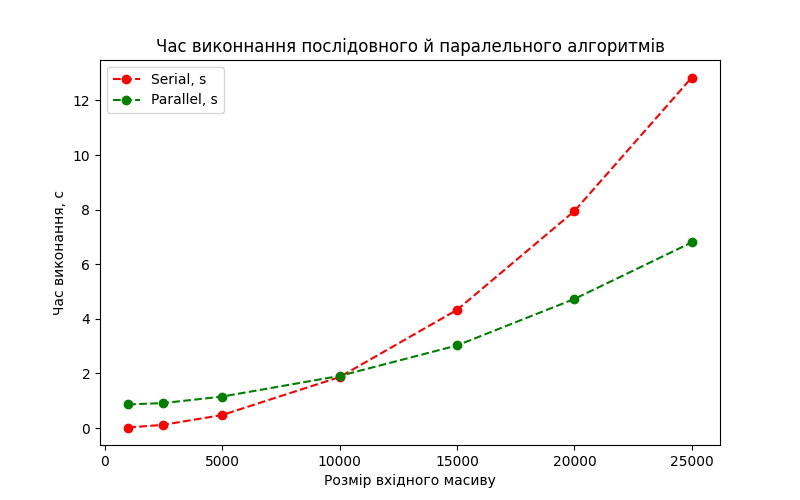


Рисунок 5.8 – Графік часу виконання ParallelSelectionMerge з n=5

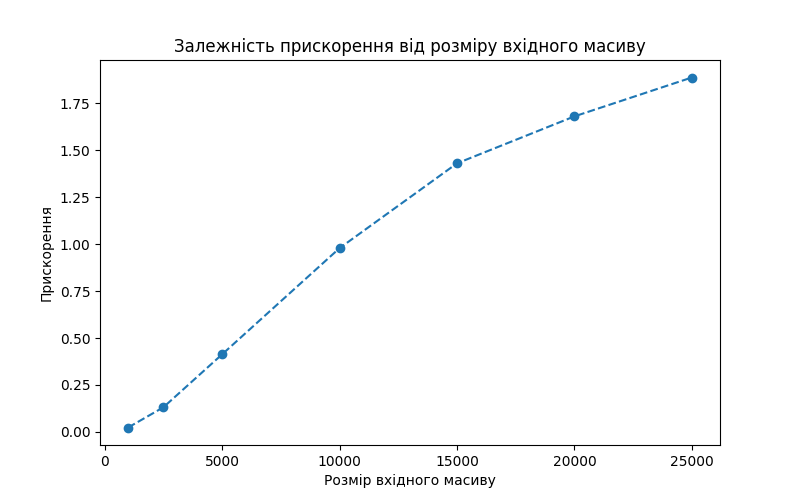


Рисунок 5.9 – Графік прискорення ParallelSelectionMerge з n=5

Справді, зменшення кількості розбиттів масиву допомогло отримати кращі результати прискорення на менших розмірах масивів та покращити час виконання паралельного алгоритму. Оскільки, від параметру розбиття масиву сильно залежить швидкодія алгоритму, спробуємо також дослідити і його. Так само, як при дослідженні послідовного алгоритму проженемо паралельний алгоритм зі сталою розміру вхідного масиву, що дорівнює 10000 але змінною кількістю розбиттів. Занотуємо результати у графіках.

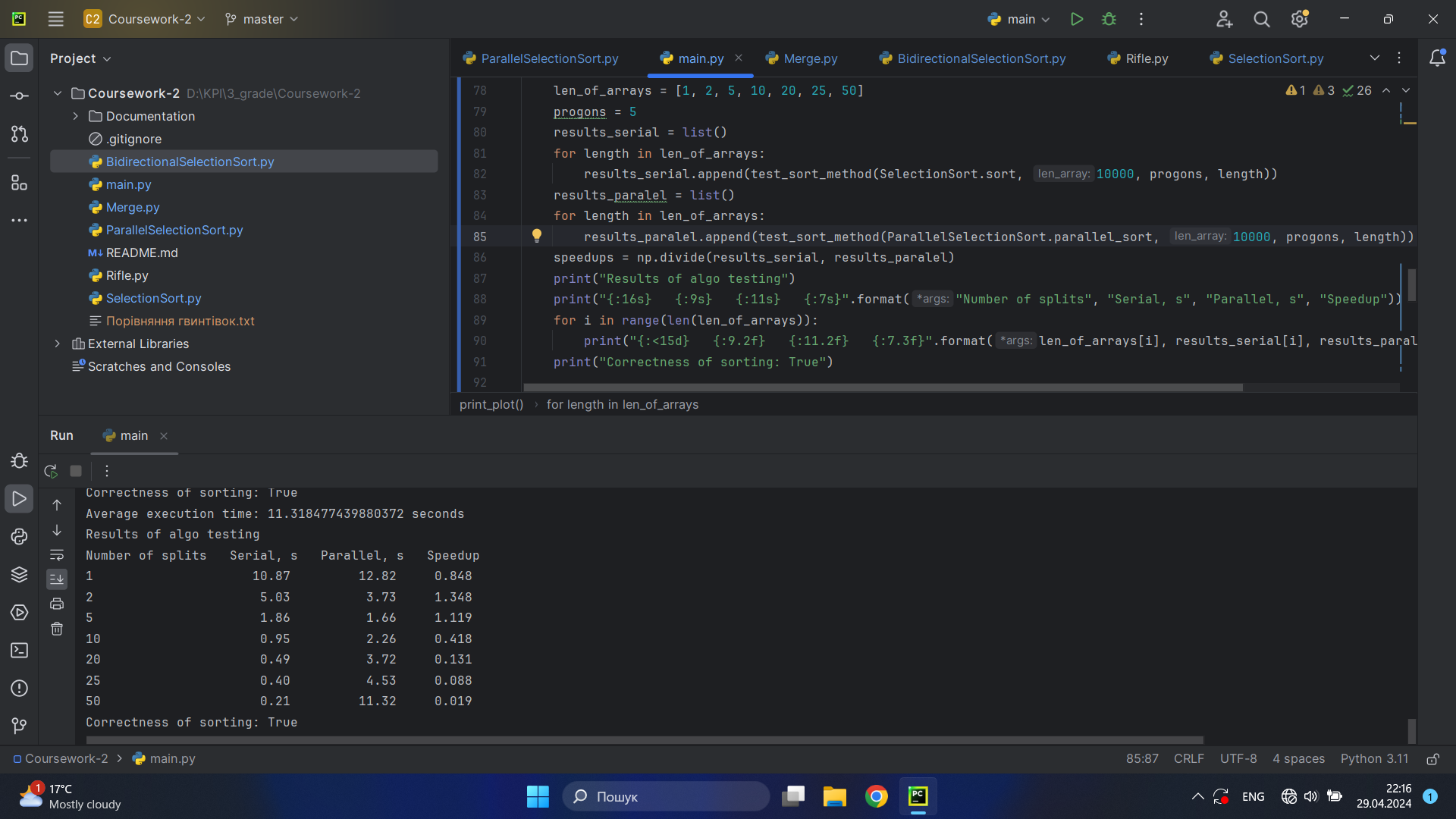


Рисунок 5.10 – Тестування кількості розбиттів алгоритму

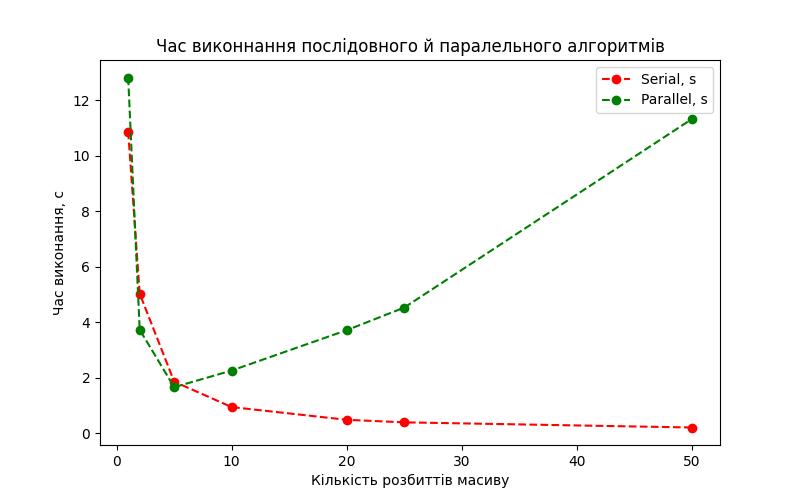


Рисунок 5.11 – Графік часу роботи від кількості розбиттів

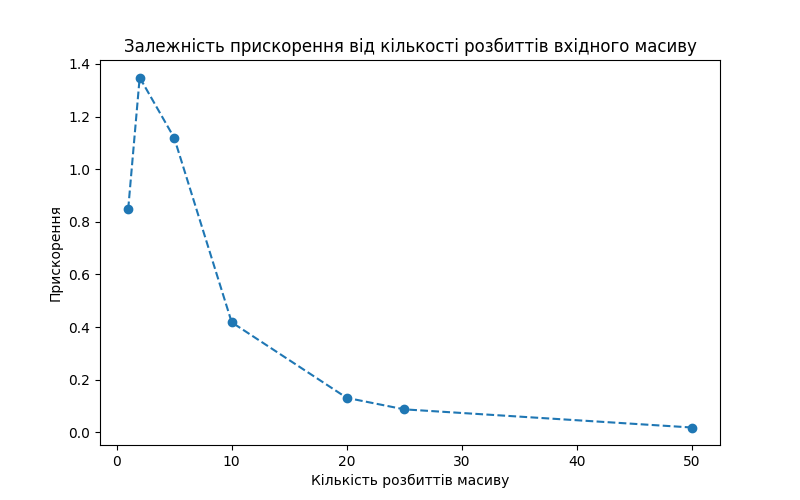


Рисунок 5.12 – Графік прискорення

Отже, як можна спостерігати, останній експеримент показав, що найдоцільніше масиви до 25000 елементів розбивати від 2 до 5 разів. Таку поведінку можна пояснити тим, що зі збільшенням кількості створюваних процесів, кількість ресурсів процесора, що витрачаються на їх створення необхідно більше, тому недоцільно використовувати велику кількість процесів.

Підсумовуючи всі вище проведені заміри швидкості роботи алгоритмів можна впевнено сказати, що

1. MinMax Bidirectional Selection Sort є неефективним алгоритмом сортування через специфіку створення паралельних програм на Python
2. ParallelSelectionMerge краще використовувати для великих вхідних масивів і його найкраще прискорення досягаєть при розбитті масиву на 5 рівномірних підмасивів.

# ВИСНОВКИ

У даній роботі було проведено дослідження щодо алгоритму сортування вибором, а також його паралельних реалізацій. Спочатку було описано принцип роботи даного алгоритму та проведено пошук його можливих паралельних реалізацій. На основі цього було спроєктовано та реалізовано гібридну послідовну версію сортування вибором й злиттям, а також дві паралельні. Всі алгоритми було подано у вигляді псевдокоду.

У реалізації паралельного алгоритму сортування вибором використовувався вбудований модуль multiprocessing мови програмування Python. Цей модуль дозволяє створювати та керувати процесами, а також виконувати обчислення паралельно, використовуючи доступні обчислювальні ресурси мультипроцесорних систем.

Аналіз швидкодії обох версій паралельних алгоритмів показав, що перша версія MinMaxBidirectionalSelectionSort має недостатню швидкість виконання через специфіку її реалізації. Другий варіант, ParallelSelectionMerge, виявився кращим за прискоренням на великих обсягах даних. Дослідження показало, що оптимальним є розбиття масиву на 5 рівномірних підмасивів для найкращого прискорення. Під час тестування даної паралельної версії алгоритму було досягнуто прискрення у 1.88 одиниць.

Отже, аналіз результатів дослідження дозволяє зробити висновок про те, що паралельна реалізація алгоритму сортування вибором на Python була виконана успішно. Прискорення порівняно з послідовною версією свідчить про ефективне використання обчислювальних ресурсів й коректність розробленого алгоритму.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sultanullah J., Salman F., Prof. Dr. Salim. *Analysis of Optimized Selection Sort Algorithm*. Режим доступу до ресурсу: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0047d34b344c05ffa4c0e94ab134d58546a7c6c2>
2. Oyelami Olufemi Moses. *Improving the performance of bubble sort using a modified diminishing increment sorting*. Режим доступу до ресурсу: <https://academicjournals.org/article/article1380628811_Olufemi.pdf>
3. J. B. Hayfron-Acquah, Ph.D. *Improved Selection Sort Algorithm*. Режим доступу до ресурсу: <https://www.researchgate.net/profile/Obed-Appiah-4/publication/272621833_Improved_Selection_Sort_Algorithm/links/54ea3c7d0cf27a6de113c461/Improved-Selection-Sort-Algorithm.pdf>
4. Khaled Thabit and Afnan Bawazir. *A Novel Approach of Selection Sort Algorithm with Parallel Computing and Dynamic Programing Concepts.* Режим доступу до ресурсу: <https://docplayer.net/45907190-A-novel-approach-of-selection-sort-algorithm-with-parallel-computing-and-dynamic-programing-concepts.html>
5. Pramod Ganapathi, Rezaul Chowdhury. *Parallel Divide-and-Conquer Algorithms for Bubble Sort, Selection Sort, and Insertion Sort.* Режим доступу до ресурсу: <https://www3.cs.stonybrook.edu/~pramod.ganapathi/doc/papers/2021-ComputerJournal-Sorting.pdf>
6. Python Documentation. *Multiprocessing.* Режим доступу до ресурсу: <https://docs.python.org/3/library/multiprocessing.html#>
7. Python Wiki. *Global Interpreter Lock.* Режим доступу до ресурсу: <https://wiki.python.org/moin/GlobalInterpreterLock>

# ДОДАТКИ

## Додаток А. Лістинг коду послідовної реалізації алгоритму

Повна версія коду доступна за посилання на GitHub: <https://github.com/muzy4ukV/parallel-selection-sort>

import sys

import numpy as np

from Merge import merge\_arrays

def sort(array, n\_split):

if len(array) % n\_split != 0:

print("It is not possible to split into so many subarrays because they will have different size")

sys.exit(1)

sub\_arrays = np.array(array).reshape(n\_split, -1)

for sub\_array in sub\_arrays:

selection\_iterative\_sort(sub\_array)

return merge\_arrays(sub\_arrays)

def selection\_iterative\_sort(array: list):

"""

Classic iterative version of selection sort

:param array: input array for sorting

:return: sorted array

"""

size = len(array)

for i in range(size - 1):

min\_index = i

for j in range(i + 1, size):

if array[j] < array[min\_index]:

min\_index = j

if min\_index != i:

array[i], array[min\_index] = array[min\_index], array[i]

return array

def merge(arr1: list, arr2: list):

"""

:param arr1, arr2: left and right subarray for merging

:return: one merged sorted array

"""

result = []

i = j = 0

while i < len(arr1) and j < len(arr2):

if arr1[i] < arr2[j]:

result.append(arr1[i])

i += 1

else:

result.append(arr2[j])

j += 1

while i < len(arr1):

result.append(arr1[i])

i += 1

while j < len(arr2):

result.append(arr2[j])

j += 1

return result

def merge\_arrays(arrays):

"""

:param arrays: input array of sorted sub\_arrays

:return: one merged sorted array

"""

if len(arrays) == 0:

return []

if len(arrays) == 1:

return arrays[0]

if len(arrays) == 2:

return merge(arrays[0], arrays[1])

mid = len(arrays) // 2

left\_half = merge\_arrays(arrays[:mid])

right\_half = merge\_arrays(arrays[mid:])

return merge(left\_half, right\_half)

## Додаток Б. Лістинг коду паралельних реалізацій алгоритму

import multiprocessing as mp

import sys

import numpy as np

from Merge import merge\_arrays

from SelectionSort import selection\_iterative\_sort

def parallel\_sort(array: list, n\_split: int):

"""

:param array: input array

:param n\_split: number of division of initial array

:return:

"""

if len(array) % n\_split != 0:

print("It is not possible to split into so many subarrays because they will have different size")

sys.exit(1)

sub\_arrays = np.array(array).reshape(n\_split, -1)

pool = mp.Pool(n\_split)

res = pool.map(selection\_iterative\_sort, sub\_arrays)

pool.close()

pool.join()

return merge\_arrays(res)

import multiprocessing as mp

def find\_min\_max(arr, start, end):

"""

Find the index of the minimum and maximum values in the array

:param arr: input array

:param start: start of subarray

:param end: end of subarray

:return: indexes of min and max

"""

min\_index = start

max\_index = start

for i in range(start + 1, end):

if arr[i] < arr[min\_index]:

min\_index = i

elif arr[i] > arr[max\_index]:

max\_index = i

return min\_index, max\_index

def parallel\_sort(arr):

size = len(arr)

left = 0

right = size

middle = (left + right) // 2

with mp.Pool(processes=2) as pool:

while left < middle-1:

t1 = pool.apply\_async(find\_min\_max, (arr, left, middle))

t2 = pool.apply\_async(find\_min\_max, (arr, middle, right))

left\_min\_idx, left\_max\_idx = t1.get()

right\_min\_idx, right\_max\_idx = t2.get()

if arr[left\_min\_idx] <= arr[right\_min\_idx]:

min\_idx = left\_min\_idx

else:

min\_idx = right\_min\_idx

arr[left], arr[min\_idx] = arr[min\_idx], arr[left]

if arr[left\_max\_idx] <= arr[right\_max\_idx]:

max\_idx = right\_max\_idx

else:

max\_idx = left\_max\_idx

arr[right - 1], arr[max\_idx] = arr[max\_idx], arr[right - 1]

left += 1

right -= 1

if arr[middle-1] > arr[middle]:

arr[middle-1], arr[middle] = arr[middle], arr[middle-1]

return arr

## Додаток В. Лістинг коду методів тестування

from typing import Callable

import numpy as np

from Rifle import Rifle

from copy import deepcopy

from time import time

import ParallelSelectionSort

import SelectionSort

import matplotlib.pyplot as plt

import BidirectionalSelectionSort

def test\_sort\_method(sort\_method: Callable, len\_array: int, progon\_number: int, n\_split=None) -> float:

"""

This method tests sorting algo for correctness and measure its average time

:param sort\_method: Sorting method for testing

:param len\_array: Length of test array

:param progon\_number: Number of testing repeating

:param n\_split: Number of array division

:return average\_time

"""

array = [Rifle() for i in range(len\_array)]

# List for collecting time for each of progon

exec\_time = list()

# Flag that will save the wrong sort error

correctness\_flag = True

for i in range(progon\_number):

arr = deepcopy(array)

# Start of measuring

start = time()

if n\_split:

arr = sort\_method(arr, n\_split)

else:

arr = sort\_method(arr)

# End of measuring

end = time()

exec\_time.append(end - start)

if not check\_array(arr):

correctness\_flag = False

print(f"- {i + 1} progon passed -")

print(f"Correctness of sorting: {correctness\_flag}")

avarage\_time = sum(exec\_time) / progon\_number

print("Average execution time:", avarage\_time, "seconds")

return avarage\_time

def check\_array(array: list[Rifle]) -> bool:

"""

Method checks if array are in ascending order

:param array:

:return:

"""

for i in range(len(array) - 1):

if array[i + 1] < array[i]:

return False

return True

def warm\_up(len\_array: int, progon\_number: int):

"""

Function that will warming up the processor before main work

:return: None

"""

print("Warming up CPU")

array = [Rifle() for i in range(len\_array)]

for i in range(progon\_number):

SelectionSort.sort(array, 5)

print(f"- {i + 1} progon passed -")

print("Warming ended")

def print\_plot():

len\_of\_arrays = [1, 2, 5, 10, 20, 25, 50]

progons = 5

results\_serial = list()

for length in len\_of\_arrays:

results\_serial.append(test\_sort\_method(SelectionSort.sort, 10000, progons, length))

results\_paralel = list()

for length in len\_of\_arrays:

results\_paralel.append(test\_sort\_method(ParallelSelectionSort.parallel\_sort, 10000, progons, length))

speedups = np.divide(results\_serial, results\_paralel)

print("Results of algo testing")

print("{:16s} {:9s} {:11s} {:7s}".format("Number of splits", "Serial, s", "Parallel, s", "Speedup"))

for i in range(len(len\_of\_arrays)):

print("{:<15d} {:9.2f} {:11.2f} {:7.3f}".format(len\_of\_arrays[i], results\_serial[i], results\_paralel[i], speedups[i]))

plt.figure(figsize=(8, 5))

plt.plot(len\_of\_arrays, results\_serial, marker='o', linestyle='--', label="Serial, s", color='red')

plt.plot(len\_of\_arrays, results\_paralel, marker='o', linestyle='--', label="Parallel, s", color='green')

plt.title('Час виконнання послідовного й паралельного алгоритмів')

plt.xlabel('Кількість розбиттів масиву')

plt.ylabel('Час виконання, с')

plt.legend()

plt.show()

plt.figure(figsize=(8, 5))

plt.plot(len\_of\_arrays, speedups, marker='o', linestyle='--')

plt.title('Залежність прискорення від кількості розбиттів вхідного масиву')

plt.xlabel('Кількість розбиттів масиву')

plt.ylabel('Прискорення')

plt.show()

def main():

warm\_up(10000, 5)

# serial\_time = test\_sort\_method(SelectionSort.sort, 15000, 5, 5)

#

# parallel\_time = test\_sort\_method(ParallelSelectionSort.parallel\_sort, 15000, 5, 5)

#

#

# speedup = serial\_time / parallel\_time

#

# print("Speedup for parallel version is -", speedup)

# print("Testing Parallel Selection Merge Sort")

# test\_sort\_method(ParallelSelectionSort.parallel\_sort, 1000, 10, 5)

print\_plot()

if \_\_name\_\_ == '\_\_main\_\_':

main()

from random import randint, randrange

class Rifle(object):

"""

This class is an abstraction for automated and semi-automated rifles

with implemented equality methods

"""

def \_\_init\_\_(self, caliber=None, barrel\_length=None, bullet\_speed=None):

self.\_caliber = caliber if caliber else randint(56, 127) / 10

self.\_barrel\_length = barrel\_length if barrel\_length else randrange(2065, 7400, 5) / 10

self.\_bullet\_speed = bullet\_speed if bullet\_speed else randrange(310, 900, 5)

def \_\_str\_\_(self):

return (f"Caliber: {self.\_caliber:4.1f} mm; Barrel length: {self.\_barrel\_length:5.1f} mm; "

f"Bullet speed: {self.\_bullet\_speed:3d} m/s")

def \_\_eq\_\_(self, other):

if isinstance(other, Rifle):

return (self.\_caliber == other.\_caliber and

self.\_bullet\_speed == other.\_bullet\_speed and

self.\_barrel\_length == other.\_barrel\_length)

return False

def \_\_lt\_\_(self, other):

if not isinstance(other, Rifle):

return False

if self.\_caliber == other.\_caliber:

return (self.\_bullet\_speed \* self.\_barrel\_length) < (other.\_bullet\_speed \* other.\_barrel\_length)

else:

return self.\_caliber < other.\_caliber

def \_\_le\_\_(self, other):

if not isinstance(other, Rifle):

return False

if self.\_caliber == other.\_caliber:

return (self.\_bullet\_speed \* self.\_barrel\_length) <= (other.\_bullet\_speed \* other.\_barrel\_length)

else:

return self.\_caliber <= other.\_caliber

def \_\_gt\_\_(self, other):

if not isinstance(other, Rifle):

return False

if self.\_caliber == other.\_caliber:

return (self.\_bullet\_speed \* self.\_barrel\_length) > (other.\_bullet\_speed \* other.\_barrel\_length)

else:

return self.\_caliber > other.\_caliber