УДК 004.43 : 004.032.24

Дослідження ефективності розпаралелювання процесорозалежних задач мовою Python на основі механізму потоків

Роман Бабаков
доцент, д-р техн. наук
ORCID: 0000-0001-7196-0912
r.babakov@donnu.edu.ua

Донецький національний університет імені Василя Стуса

Олександр Баркалов
професор, д-р техн. наук
ORCID: 0000-0002-4791-2088
a.barkalov@imei.uz.zgora.pl

Університет Зеленогурський

Ключові слова:
процесорозалежні задачі;
швидкість обробки даних;
розпаралелювання;
механізм потоків;
мова Python.

Метою роботи є дослідження ефективності застосування паралельної обробки даних на основі потоків у мові Python для розв'язання задач, які потребують значних ресурсів центрального процесора. В якості такої задачі розглядалась задача обробки двовимірного масиву великих розмірів стандартними засобами Python без використання спеціалізованих бібліотек на кшталт Numpy. Виявлено непрогнозоване зменшення часу виконання багатопотокової програми на 30% у порівнянні з послідовною (непаралельною) реалізацією розв'язуваної задачі. Це дає змогу поставити під сумнів загальновідоме твердження про те, що багатопотокове розв'язання процесорозалежних задач саме в мові Python не є ефективним, оскільки не приводить до зменшення часу роботи програм. Раніше вважалося, що завдяки використанню так званого глобального замка (GIL) інтерпретатор мови Python у разі виконання багатопотокових програм із застосуванням модуля Threading спрямовує усі потоки програми на одне процесорне ядро, що виключає фізичне паралельне виконання потоків і за жодних умов не дає змоги зменшити час роботи програми. Відомою рекомендацією для отримання реального виграшу в часі є використання розпаралелювання на основі механізму процесів і модуля «multiprocessing», який допомагає задіяти кілька фізичних ядер процесора. Однак такий підхід потребує додаткових ресурсів процесора (ядер) та додаткових витрат часу на обмін даними між процесами, що може звести нанівець ефект від багатоядерної обробки. Проведені авторами експерименти довели, що застосування багатопотокового підходу також може бути доцільним у випадку процесорозалежних задач, оскільки виграш у часі, що досягається, не потребує додаткових ресурсів з боку центрального процесора комп'ютера та зайвих витрат часу на обмін даними між потоками.

DOI: 10.31558/2786-9482.2024.1.2

Вступ

Технологічний прогрес наблизився до межі можливостей збільшення швидкодії процесорних елементів комп'ютерної техніки. Частота роботи процесора на рівні 3–5 ГГц

була досягнута більше 10 років тому і зупинилась на цьому рівні. Подальше суттєве зростання частоти виявилось фізично неможливим, оскільки обмежується максимальною швидкістю проходження електричного струму в провідниках як у процесорному ядрі, так і між процесором та оперативною пам'яттю [1, 2].

Перспективним напрямом збільшення продуктивності обчислювальних систем виявилося застосування багатопроцесорних архітектур [3, 4]. У них використовується кілька процесорних елементів, які мають можливість працювати паралельно та зазвичай реалізуються у вигляді процесорних ядер на єдиному кристалі процесора. У сфері програмного забезпечення спостерігається переорієнтація на багатоядерні архітектури, що відображається на можливостях операційних систем і прикладних програм. У цьому аспекті очікуваним є зростання підтримки реалізації паралельних алгоритмів багатьма мовами програмування. Технології, як-от OpenMP, MPI та CUDA, добре відомі розробникам програмного забезпечення і дають змогу задіяти додаткове апаратне забезпечення для пришвидшення розв'язання обчислювальних задач [1, 3]. Альтернативою цим технологіям є використання функціоналу операційних систем, щоб задіяти ресурси сучасних багатоядерних процесорів.

Мова програмування Python містить 2 модулі підтримки паралельних алгоритмів [5–7]. Модуль *Multiprocessing* реалізує паралелізм на основі механізму процесів. Програма, написана з використанням цього модуля, дає змогу запустити декілька процесів, кожен із яких може виконуватись на окремому процесорному ядрі і має відокремлену область пам'яті. Перевага таких програм полягає у можливості одночасного виконання окремих ділянок програми, недоліком є значні витрати часу на обмін даними між процесами, які можуть нівелювати ефект від мультипроцесорної обробки даних.

Іншим модулем для розробки паралельних програм є модуль *Threading*, який реалізує паралелізм на основі механізму потоків. Усі потоки, які створено однією програмою, використовують спільну область пам'яті, що значно спрощує і пришвидшує обмін даними між потоками. Однак інтерпретатор мови Python запускає потоки лише на одному процесорному ядрі у режимі конкуренції [1, 6, 7]. Так само на одному ядрі виконуються звичайні послідовні Python-програми, що не використовують жодних засобів розпаралелювання. В тих Python-програмах, які значну частину часу витрачають на операції вводу-виводу, паралелізм на основі потоків дає змогу отримати значну економію часу. Наприклад, виконання одночасних запитів до великої кількості вебсайтів не потребує значних витрат з боку центрального процесора і може бути легко виконано на одному процесорному ядрі у псевдопаралельному режимі. Натомість складні обчислювальні алгоритми, які потребують безперервного використання процесора, у випадку багатопотокової реалізації будуть виконуватись не швидше, а довше, порівняно з їх послідовною реалізацією. Збільшення часу обумовлене додатковими діями інтерпретатора Python і операційної системи, спрямованими на створення, запуск і завершення множини потоків.

Отже, паралелізм на основі потоків є непридатним для розв'язання задач, що потребують значної потужності процесора. Втім це твердження є справедливим лише для мови Python, що зазначено у документації модуля Threading [5, 6]. Причина полягає у тому, що інтерпретатор Python використовує так званий глобальний замок інтерпретатора (GIL –

Global Interpreter Lock), який дає змогу виконувати байт-код тільки одному потоку в один момент часу. Застосування GIL забезпечує виконання тільки однієї атомарної інструкції за раз і гарантує безпеку та захист даних у багатопотокових програмах. Це стосується версії Python 3.12, що є актуальною сьогодні. Розробники Python мають плани на переробку GIL з метою одночасного доступу потоків до кількох ядер процесора, однак наразі використання механізму потоків для розпаралелювання складних обчислювальних алгоритмів вважається недоцільним.

Мета статті полягає в експериментальному дослідженні ефективності використання розпаралелювання програми на основі механізму потоків для розв'язання задачі, що потребує значних ресурсів центрального процесора, за критерієм витрат часу на виконання програми.

Постановка завдання дослідження

У роботі здійснюється експериментальна перевірка припущення про те, що використання паралелізму на основі потоків під час розв'язання процесорозалежних задач мовою Python не є доцільним і не приводить до зменшення часу роботи програми.

Для проведення експерименту обрано задачу з обробки двовимірного масиву, яка формулюється так.

Заданий двовимірний масив розміром M рядків на N стовпців, кожним елементом якого є псевдовипадкове ціле число в діапазоні від –1000 до 1000. Знайти серед усіх елементів масиву кількість таких елементів, для яких сума цифр у значенні елементу, незалежно від знака числа, дорівнює 10 (наприклад, 91, –253, 730, –28, 55). Потім замінити на це число усі елементи масиву, які є квадратами цілих чисел.

Запропоновано такий порядок проведення експерименту.

1. Розв'язати поставлену задачу за допомогою мови Python без використання засобів паралельного програмування. Результатом має бути послідовна програмна реалізація алгоритму задачі.
2. Розв'язати поставлену задачу за допомогою мови Python із використанням паралелізму на основі потоків на базі модуля Threading. Результатом має бути паралельна програмна реалізація алгоритму задачі.
3. Виміряти тривалість розв'язання задачі за послідовної і паралельної програмних реалізацій для різних розмірів масиву.
4. Порівняти результати виміру часу та зробити відповідні висновки.

Очікуваним результатом експерименту є те, що час роботи паралельної програми буде рівним або більшим за час роботи послідовної програми. Це пов'язано з тим, що в мові Python багатопотокові програми примусово виконуються на одному ядрі процесора. Отже, відсутня апаратна складова, що дає змогу прискорити виконання багатопотокової програми, порівняно з її послідовною версією.

Розв'язання задачі у вигляді послідовної реалізації

Розглянемо реалізацію алгоритму розв'язання поставленої задачі у послідовному вигляді, тобто без розпаралелювання.

Задамо розміри масиву у вигляді констант. Створимо двовимірний масив розміром $M \times N$, заповнений псевдовипадковими цілими числами в діапазоні [–1000, 1000]:

```
import random
M, N = 3, 4
mas = []                      # Підготовка порожнього масиву рядків
for i in range(0, M):         # Цикл за рядками
    mas_i = []                # Підготовка порожнього рядка
    for j in range(0, N):                # Цикл за стовпцями
        n = random.randint(-1000, 1000) # Генерація рандомного числа
        mas_i.append(n)                  # Додавання числа у рядок
    mas.append(mas_i)                    # Додавання рядка в масив
```

Знайдемо в масиві кількість елементів, сума цифр яких, незалежно від знака числа, дорівнює 10. Для зберігання кількості знайдених елементів будемо використовувати змінну `n10`:

```
n10 = 0
```

Організуємо вкладений цикл, всередині якого будемо брати з масиву черговий елемент та рахувати суму його цифр. Програмний код може виглядати так:

```
for i in range(0, M):            # Цикл за рядками
    for j in range(0, N):        # Цикл за стовпцями
        n = mas[i][j]            # Беремо елемент масиву
        s = str(n)               # Переводимо в символьний формат
        s = s.replace("-", "")   # Видаляємо знак "мінус", якщо він є
        m = 0                    # Початкове значення суми цифр
        for c in s:              # Перебирання символів рядка
            m += int(c)          # Додаємо цифру до суми
        if m == 10:              # Якщо сума цифр дорівнює 10
            n10 = n10 + 1        # Збільшуємо кількість
            print(n, end=" ")
print("\nЗнайдено чисел:", n10)
```

У наведеному фрагменті передостання команда `print` додана для перевірки працездатності програми і виводить на екран усі елементи масиву, сума цифр яких дорівнює 10. Ця команда потрібна лише на етапі тестування програми. Далі вона буде видалена, оскільки виведення інформації на екран може займати значний відсоток часу роботи програми.

Результат роботи програми для масиву розміром 10×10 виглядає так:

```
145 -64 712 640 -307 631 541 -181
Знайдено чисел: 8
```

Розв’яжемо другу частину задачі: замінимо всі елементи масиву, що є квадратами цілих чисел, на знайдене число `n10`. Для цього будемо проглядати в циклі всі елементи масиву та перевіряти, чи є черговий елемент додатним і чи є корінь із нього цілим числом. Для цього використаємо такий фрагмент програмного коду:

```
for i in range(0, M):              # Цикл за рядками
  for j in range(0, N):            # Цикл за стовпцями
    n = mas[i][j]                  # Беремо елемент масиву
```

```
if n > 0:                    # Якщо число додатне
   if n ** 0.5 == int(n ** 0.5):  # Якщо корінь є цілим числом
      mas[i][j] = n10        # Робимо заміну елементу на n10
      print("[", i, "][", j, "] = ",n, " -> ", n10, sep = "")
```

Як і в попередньому фрагменті, остання команда `print` потрібна лише на етапі налагодження програми.

Повний лістинг програми утворюється послідовним об'єднанням розглянутих вище фрагментів коду. Задавши розміри масиву $M = 10$, $N = 10$ і запустивши програму, отримаємо такий результат:

```
-451 505 -640 -343 -235 -82 64
Знайдено чисел: 7
[4][5] = 121 -> 7
[5][2] = 9 -> 7
[6][4] = 784 -> 7
[8][1] = 64 -> 7
```

У прикладі знайдено 7 чисел, сума цифр яких дорівнює 10. Потім було знайдено чотири числа, що є квадратами цілих чисел: 121, 9, 784, 64. Усі числа були замінені на число 7.

Масив розміром 10×10 не є достатньо великим, щоб у ньому зустрічалось багато псевдовипадкових чисел, що є квадратами цілих чисел. У наведеному вище прикладі таких чисел лише чотири. Якщо запустити програму ще кілька разів, результати можуть бути такими:

```
-370 -244 -280 190
Знайдено чисел: 4
[0][3] = 784 -> 4
[3][3] = 196 -> 4

-235 -190 -55 631 424 -730 910 -442 514
Знайдено чисел: 9
[3][2] = 484 -> 9

523 73 145 550 -811 -190 433 118 244 352
Знайдено чисел: 10
```

В останньому прикладі квадрати цілих чисел взагалі не були знайдені, і це цілком можлива ситуація для масиву невеликого розміру. Якщо задати розміри масиву як 20×20, знайдених квадратів зазвичай буде більше:

```
-55 -370 -370 226 -154 802 118 -271 -316 -253 -505 802 127 -19 352 253 -361 -190 433 433 -712 82 154 -64 -307
Знайдено чисел: 25
[1][14] = 676 -> 25
[3][9] = 529 -> 25
[4][5] = 625 -> 25
[4][14] = 121 -> 25
[7][0] = 900 -> 25
[12][2] = 25 -> 25
[18][0] = 49 -> 25
```

Додамо в програму команди для виміру часу виконання фрагментів коду. Для цього скористаємось можливостями стандартного модуля Time. Перший вимір поточного часу здійснюється відразу після псевдовипадкової генерації масиву, другий вимір – відразу після завершення програми. Різниця в часі буде дорівнювати кількості секунд, що витрачені на

виконання відповідного коду програми. Структурно розташування команд для виміру часу виглядатиме так:

```
import time
# Псевдовипадкова генерація масиву
t1 = time.time()         # Початковий час
# Обробка масиву
t2 = time.time()         # Кінцевий час
print("Час роботи програми:", t2-t1, "секунд.")
```

Розв'язання задачі у вигляді багатопотокової реалізації

Реалізуємо алгоритм розв'язання поставленої задачі обробки двомивірного масиву з використанням механізму потоків.

Задача складається з двох частин. У першій частині шукається кількість елементів масиву, сума цифр яких дорівнює 10. У другій частині шукаються і замінюються елементи, що є квадратами цілих чисел. У кожній частині іде робота з усіма елементами масиву, причому порядок перегляду елементів масиву для цієї задачі не має значення. Це дає змогу задіяти для обробки масиву декілька дочірніх потоків. Кожному потоку можна надавати якийсь фрагмент масиву, і потік буде виконувати завдання тільки для нього. Коли потік завершуватиме роботу, він передаватиме головному потоку результати своєї роботи і завершуватиметься. Головний потік буде швидко обробляти результати роботи дочірніх потоків.

Спроєктуємо програму так, щоб кожен дочірній потік виконував обробку тільки одного рядка двовимірного масиву. Результатом роботи є кількість цільових елементів, які містяться в цьому рядку масиву. Наприкінці роботи дочірнього потоку знайдена кількість додається до глобальної змінної `n10`, в якій ведеться накопичення результатів роботи кожного потоку. В даному випадку необхідна кількість потоків дорівнює кількості рядків масиву.

Команди створення і заповнення масиву залишимо такими, як у попередніх прикладах, за винятком того, що на початку програми під'єднаємо модуль Threading командою `import threading`.

Додамо в програму глобальний замок типу threading.Lock. Він буде застосовуватися для монопольного використання потоками команд `print` та доступу до глобальних змінних з метою упередження станів перегонів (race conditions):

```
L = threading.Lock()                # Замок
```

Як і для послідовної реалізації, створимо змінну `n10`, у якій буде накопичуватись кількість знайдених елементів масиву.

Напишемо функцію `f1`, яка приймає номер рядка двовимірного масиву, виконує в межах цього рядка пошук кількості елементів, сума цифр яких дорівнює 10, та додає цю кількість до глобальної змінної `n10`:

```
def f1(i):                    # Пошук у рядку з номером i
    global n10                # Будемо змінювати глобальну змінну
    k = 0                     # Кількість знайдених елементів у рядку
    for j in range(0, N):     # Цикл за стовпцями
```

```
        n = mas[i][j]         # Беремо елемент масиву
        s = str(n)            # Переводимо в символьний формат
        s = s.replace("-", "")  # Видаляємо знак "мінус", якщо він є
        m = 0                 # Початкове значення суми цифр
        for c in s:           # Перебирання символів рядка
            m += int(c)       # Додаємо цифру до суми
        if m == 10:           # Якщо сума цифр дорівнює 10
            k = k + 1         # Збільшуємо кількість
    L.acquire()
    n10 += k                  # Збільшуємо глобальну змінну
    L.release()
```

Функція працює безпосередньо з глобальним масивом `mas`. Це є можливим завдяки тому, що всі дочірні потоки в Python працюють з однією областю пам'яті і мають безпосередній доступ до глобальних змінних.

Створимо порожній список потоків, після чого в циклі створимо *M* дочірніх потоків. Кожен потік створимо на базі функції `f1` і передамо до неї номер рядка масиву:

```
t_list = []                   # Підготовка списку потоків
for i in range(0, M):         # Цикл зі створення потоків
    t = threading.Thread(target=f1, args=(i, ))
    t_list.append(t)
```

Виконаємо 2 цикли: цикл із запуску потоків та цикл з очікування потоків:

```
for t in t_list:              # Цикл із запуску потоків
    t.start()
for t in t_list:              # Цикл з очікування потоків
    t.join()
```

Після того, як робота всіх потоків завершиться, у змінній `n10` буде знаходитись кількість елементів масиву, сума цифр яких дорівнює 10.

Тепер розпаралелимо другу частину завдання, а саме процес заміни елементів масиву, що є квадратами цілих чисел, на значення кількості елементів, сума цифр яких дорівнює 10, тобто на значення змінної `n10`.

Як і для першої частини завдання, розпаралелимо обробку масиву на рівні окремих рядків – створимо на кожен рядок масиву один потік. Для цього напишемо функцію `f2`, яка приймає як аргумент номер рядка двовимірного масиву, проглядає цей рядок та замінює елементи, що є квадратами цілих чисел, на значення глобальної змінної `n10`:

```
def f2(i):                  # Заміна у рядку з номером i
  global mas                # Можливість змінювати глобальний масив
  for j in range(0, N):     # Цикл за стовпцями
    n = mas[i][j]           # Беремо елемент масиву
      if n > 0:             # Якщо число додатне
         if n ** 0.5 == int(n ** 0.5):  # Якщо корінь є цілим числом
            L.acquire()             # Захоплення замка
            mas[i][j] = n10         # Заміна елементу масиву на n10
            L.release()             # Вивільнення замка
```

Операція запису в глобальний масив виконується у «монопольному» режимі після захоплення замка. Хоча різні потоки працюють із різними рядками масиву і виникнення стану перегону даних не очікується, автори вважають використання замка правильним підходом, хоча це може трохи уповільнити роботи програми.

Розпаралелювання функції `f2` виконується так само, як і у випадку функції `f1`:

```
t_list = []                          # Підготовка списку потоків
for i in range(0, M):                # Цикл зі створення потоків
    t = threading.Thread(target=f2, args=(i, ))  # Функція f2
    t_list.append(t)
for t in t_list:                     # Цикл із запуску потоків
    t.start()
for t in t_list:                     # Цикл з очікування потоків
    t.join()
```

Об’єднаємо всі розглянуті вище фрагменти коду, пов’язані з багатопотоковою обробкою масиву, в єдину програму. Додатково додамо в програму команди виміру часу роботи подібно тому, як це було зроблено у випадку послідовної версії програми.

Результати експериментальних досліджень

Експерименти здійснено на комп’ютері з процесором i5-13500 та з оперативною пам’яттю типу DDR5 обсягом 64 Гб, який працює під управлінням ОС Windows 11. Використовувалась версія Python 3.12.

Порівняємо час виконання послідовної і паралельної версій програми. Для цього зробимо таке.

1. Визначимо такі розміри масиву, щоб його послідовна обробка тривала 1, 2, 5, 10 та 30 секунд. Під час цього будемо дотримуватись рівності значень M і N. Результати вимірів наведені в табл. 1. За розмірів масиву, вказаних у другому стовпці табл. 1, час обробки масиву за допомогою послідовної версії програми був найбільш близьким до значення, вказаного в першому стовпці таблиці (за усередненими результатами кількох запусків програми).

Таблиця 1. Тривалість послідовного розв’язання задачі

Тривалість (t_1), с	Розмір масиву
1	1230×1230
2	1770×1770
5	2800×2800
10	3950×3950
30	6850×6850

2. Визначимо тривалість виконання паралельної версії програми для розмірів масиву, вказаних у другому стовпці табл. 1. Результати вимірів наведені в табл. 2.

Таблиця 2. Тривалість паралельного розв'язання задачі із використанням механізму потоків

Розмір масиву	Тривалість (t_2), с
1230×1230	0.80
1770×1770	0.96
2800×2800	3.52
3950×3950	6.84
6850×6850	20.5

Порівнюючи значення у першому стовпці табл. 1 зі значеннями у другому стовпці табл. 2, можна бачити, що для усіх розглянутих розмірів масиву тривалість виконання паралельної програми виявилась меншою, ніж тривалість виконання еквівалентної послідовної програми.

3. Визначимо виграш у часі у разі використанні паралельної версії програми замість послідовної. Значення виграшу будемо виражати у відсотках:

$$E = \frac{t_2 - t_1}{t_1} \cdot 100\,\% . \tag{1}$$

Результати оцінювання ефекту від паралелізму (табл. 3) засвідчують, що виграш у часі для масивів великих розмірів сягає приблизно 30%, порівняно з послідовною обробкою масиву.

Таблиця 3. Виграш в часі за рахунок багатопотоковості

Розмір масиву	t_1 , с	t_2 , с	E, %
1230×1230	1	0.80	20
1770×1770	2	0.96	52
2800×2800	5	3.52	29.6
3950×3950	10	6.84	31.6
6850×6850	30	20.5	31.6

Обговорення отриманих результатів

Розглянута в цій роботі задача обробки двовимірного масиву може вважатись процесорозалежною, оскільки передбачає значний обсяг математичних обчислень та не використовує значною мірою операції вводу-виводу даних. Хоча вважається, що багатопотокова реалізація процесорозалежних задач мовою Python не приводить до зменшення часу, порівняно з послідовною реалізацією, отримані результати демонструють зворотне. Отриманий тридцятивідсотковий виграш у часі досягнутий лише за рахунок багатопотокової реалізації без будь-якої оптимізації алгоритму обробки. Водночас додатковий час на створення, обробку та завершення кількох тисяч потоків не завадив загальному виграшу в часі роботи програми.

За проведеними дослідженнями однозначно встановити причину отриманого виграшу не вдалося. Авторами були проведені додаткові дослідження, які полягали у подібному розв'язанні інших задач, пов'язаних з обробкою двовимірних масивів. Дослідження показали, що у випадку деяких задач виграш також сягав приблизно 30%, тоді як для інших

задач виграш був відсутній, і час роботи паралельної версії програми був рівним або більшим за час роботи послідовної версії. Якихось узагальнень та ґрунтовних висновків зробити не вдалося.

Для розглянутої в цій роботі задачі виграш зберігався за різних конфігурацій комп'ютера та версій ОС Windows. Хоча фактичні значення часу роботи програм були більшими (через менш продуктивний процесор), значення виграшу у кілька десятків відсотків зберігалось. Це дає змогу попередньо говорити про незалежність отриманого ефекту від апаратно-програмного забезпечення.

На основі отриманих результатів автори вважають за доцільне провести додаткові дослідження, які полягатимуть у такому:

1) дослідження впливу на виграш співвідношення кількості рядків і стовпців двовимірного масиву;

2) дослідження впливу на виграш загальної завантаженості процесора комп'ютера;

3) дослідження впливу на виграш кількості використовуваних потоків;

4) дослідження ефективності багатопотокової реалізації інших процесорозалежних задач, зокрема операцій множення матриць;

5) проведення зазначених досліджень на комп'ютерах під управлінням різних операційних систем.

Висновки

Експериментально досліджено ефективність розв'язання процесорозалежних обчислювальних задач мовою Python із використанням розпаралелювання на основі механізму потоків. Результати експериментів показали, що у деяких випадках багатопотокова реалізація алгоритму скорочує на 30% тривалість роботи програми, порівняно з послідовною реалізацією. Виграш досягається лише завдяки застосуванню багатопотоковості, без використання додаткових апаратних ресурсів та програмних засобів, тобто «безкоштовно».

Отримані результати ставлять під сумнів загальноприйняте твердження про те, що застосування механізму потоків у мові Python під час виконання процесорозалежних алгоритмів є недоцільним. Якщо розпаралелювання алгоритму на основі процесів за якихось причин є неможливим або недоцільним (наприклад, у випадку великих витрат часу на обмін даними між процесами), програмістам на Python рекомендується завжди розглядати можливість багатопотокової реалізації алгоритмів та оцінювати відповідний ефект у кожному конкретному випадку. Це може привести до отримання виграшу в часі виконання алгоритму без додаткових витрат. Така рекомендація визначає практичну цінність отриманих результатів та перспективність подальших досліджень у цьому напрямі.

Література

1. Качко, О. Г. (2016). *Паралельне програмування*. Харків: Харківський національний університет радіоелектроніки.
2. Мельник, А. О., Яковлєва, І. Д. (2018). *Структурний аналіз і синтез паралельних алгоритмів*. Чернівці: Чернівецький національний університет.

3. Семеренко, В. П. (2018). *Технології паралельних обчислень*. Вінниця: Вінницький національний технічний університет.
4. Burns, B. (2018). *Designing Distributed Systems. Patterns and Paradigms for Scalable, Reliable Services*. Sebastopol: O'Reilly Media.
5. Lutz, M. (2013). *Learning Python*, 5th Edition. Sebastopol: O'Reilly Media.
6. A Guide to Python Multiprocessing and Parallel Programming (2022). https://www.sitepoint.com/python-multiprocessing-parallel-programming/
7. Parallel Processing in Python (2019). https://www.geeksforgeeks.org/parallel-processing-in-python/

Рукопис отримано – 25/06/2024; прийнято до публікації – 03/07/2024.

Research on the efficiency of parallelization of processor-dependent tasks in Python based on the thread mechanism

Roman Babakov, Olexander Barkalov

Abstract

The objective of the paper is to study the effectiveness of using parallel data processing based on threads in the Python language to solve problems that require significant CPU resources. The task of processing a two-dimensional array of large sizes using standard Python tools without using specialized libraries such as Numpy was considered as such a problem. As a result of the research, an unexpected decrease in the execution time of the multi-threaded program by 30% was found in comparison with the sequential (non-parallel) implementation of the problem being solved. This makes it possible to question the well-known statement that multi-threaded solving of processor-dependent tasks in the Python language is not effective, as it does not lead to a decrease in the running time of programs. Previously, it was believed that due to the use of the so-called global lock (GIL), the Python language interpreter, when executing multithreaded programs using Threading module, directs all program threads to one processor core, which excludes the physical parallel execution of threads and under no circumstances allows to reduce time of program execution. A well-known recommendation for obtaining a real gain in time is the use of parallelization based on the process mechanism and the "multiprocessing" module, which allows several physical processor cores to be used. However, this approach requires additional processor resources (cores) and additional time spent on data exchange between processes, which can nullify the effect of multi-core processing. The experiments conducted by the authors proved that the use of a multithreaded approach can also be appropriate in the case of processor-dependent tasks, since the time gain achieved does not require additional resources from the computer's central processor and unnecessary time spent on data exchange between threads.

Keywords: processor-dependent tasks; speed of data processing; parallelization; thread mechanism; the Python language.

References

1. Kachko, O. H. (2016). *Paralelne prohramuvannia*. Kharkiv: Kharkivskyi natsionalnyi universytet radioelektroniky.
2. Melnyk, A. O., Yakovlieva, I. D. (2018). *Strukturnyi analiz i syntez paralelnykh alhorytmiv*. Chernivtsi: Chernivetskyi natsionalnyi universytet.
3. Semerenko, V. P. (2018). *Tekhnolohii paralelnykh obchyslen*. Vinnytsia: Vinnytskyi natsionalnyi tekhnichnyi universytet.
4. Burns, B. (2018). *Designing Distributed Systems. Patterns and Paradigms for Scalable, Reliable Services*. Sebastopol: O'Reilly Media.
5. Lutz, M. (2013). *Learning Python*, 5th Edition. Sebastopol: O'Reilly Media.
6. A Guide to Python Multiprocessing and Parallel Programming (2022). https://www.sitepoint.com/python-multiprocessing-parallel-programming/
7. Parallel Processing in Python (2019). https://www.geeksforgeeks.org/parallel-processing-in-python/