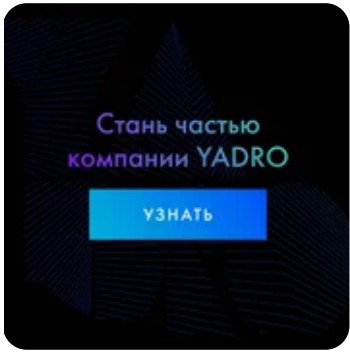


Способы реализации параллельных вычислений в программах на Python



oneweekoffer.yadro.com

РЕКЛАМА

Ищем ведущих программистов в команду YADRO.

Ждем амбициозных и талантливых, которые горят инновационными идеями

Заполни анкету

Узнать больше

[Справочник по языку Python3.](#) / Способы реализации параллельных вычислений в программах на Python

Что такое параллелизм?

Параллелизм дает возможность [работать над несколькими вычислениями одновременно](#) в одной программе. Такого поведения в Python можно добиться несколькими способами:

- Используя многопоточность [threading](#), позволяя нескольким потокам работать по очереди.
- Используя несколько ядер процессора [multiprocessing](#). Делать сразу несколько вычислений, используя несколько ядер процессора. Это и называется параллелизмом.
- Используя [асинхронный ввод-вывод](#) с [модулем asyncio](#). Запуская какую то задачу, продолжать делать другие вычисления, вместо ожидания ответа от сетевого подключения или от операций чтения/записи.

Разница между потоками и процессами.

Поток threading - это независимая последовательность выполнения каких то вычислений. Поток thread делит выделенную память ядру процессора, а также его процессорное время со всеми другими потоками, которые создаются программой в рамках одного ядра процессора. Программы на языке Python имеют, по умолчанию, один основной поток. Можно создать их больше и позволить Python переключаться между ними. Это переключение происходит очень быстро и кажется, что они работают параллельно.

Понятие **процесс в multiprocessing** - представляет собой также независимую последовательность выполнения вычислений. В отличие от потоков threading, процесс имеет собственное ядро и следовательно выделенную ему память, которое не используется совместно с другими процессами. Процесс может клонировать себя, создавая два или более экземпляра в одном ядре процессора.

Асинхронный ввод-вывод не является ни потоковым (threading), ни многопроцессорным (multiprocessing). По сути, это однопоточная, однопроцессная парадигма и не относится к параллельным вычислениям.

У Python есть одна особенность, которая усложняет параллельное выполнение кода. Она называется GIL, сокращенно от [Global Interpreter Lock](#). GIL гарантирует, что в любой момент времени работает только один поток. Из этого следует, что с потоками невозможно использовать несколько ядер процессора.

GIL был введен в Python потому, что управление памятью CPython не является потокобезопасным. Имея такую блокировку Python может быть уверен, что никогда не будет [условий гонки](#).

Что такое условия гонки и потокобезопасность?

- Состояние гонки** возникает, когда несколько потоков могут одновременно получать доступ к общей структуре данных или местоположению в памяти и изменять их, вследствие чего могут произойти непредсказуемые вещи...

Пример из жизни: если два пользователя одновременно редактируют один и тот же документ онлайн и второй пользователь сохранит данные в базу, то перезапишет работу первого пользователя. Чтобы избежать условий гонки, необходимо заставить второго пользователя ждать, пока первый закончит работу с документом и только после этого разрешить второму пользователю открыть и начать редактировать документ.

- Потокобезопасность** работает путем создания копии локального хранилища в каждом потоке, чтобы данные не сталкивались с другим потоком.

Алгоритм планирования доступа потоков к общим данным.

Как уже говорилось, потоки используют одну и ту же выделенную память. Когда несколько потоков работают одновременно, то нельзя угадать порядок, в котором потоки будут обращаться к общим данным. Результат доступа к совместно используемым данным зависит от **алгоритма планирования**, который решает, какой поток и когда запускать. Если такого алгоритма нет, то конечные данные могут быть не такими как ожидаешь.

Например, есть общая переменная `a = 2`. Теперь предположим, что есть два потока, `thread_one` и `thread_two`. Они выполняют следующие операции:

```
a = 2

# функция 1 потока
def thread_one():
    global a
    a = a + 2

# функция 2 потока
def thread_two():
    global a
    a = a * 3
```

Если поток `thread_one` получит доступ к общей переменной `a` первым и `thread_two` вторым, то результат будет 12:

- 1. $2 + 2 = 4$;
- 2. $4 * 3 = 12$.

или наоборот, сначала запустится `thread_two`, а затем `thread_one`, то мы получим другой результат:

- 1. $2 * 3 = 6$;
- 2. $6 + 2 = 8$.

Таким образом очевидно, что порядок выполнения операций потоками имеет значение

Без алгоритмов планирования доступа потоков к общим данным такие ошибки очень трудно найти и произвести отладку. Кроме того, они, как правило, происходят случайным образом, вызывая беспорядочное и непредсказуемое поведение.

Есть еще худший вариант развития событий, который может произойти без встроенной в Python блокировки потоков [GIL](#) . Например, если оба потока начинают читать глобальную переменную `a` одновременно, оба потока увидят, что `a = 2`, а дальше, в зависимости от того какой поток произведет вычисления последним, в конечном итоге и будет равна переменная `a` (4 или 6). Не то, что ожидалось!

Исследование разных подходов к параллельным вычислениям в Python.

Определим функцию, которую будем использовать для сравнения различных вариантов вычислений. Во всех следующих примерах используется одна и та же функция, называемая `heavy()` :

```
def heavy(n):
    for x in range(1, n):
        for y in range(1, n):
            x**y
```

Функция `heavy()` представляет собой вложенный цикл, который выполняет возведение в степень. Это функция связана со скоростью ядра процессора производить математические вычисления. Если понаблюдать за операционной системой во время выполнения функции, то можно увидеть загрузку ЦП близкую к 100%.

Будем запускать эту функцию по-разному, тем самым исследуя различия между обычной однопоточной программой Python, многопоточностью и многопроцессорностью.

Однопоточный режим работы.

Каждая программа Python имеет по крайней мере один основной поток. Ниже представлен пример кода для запуска функции `heavy()` в одном основном потоке одного ядра процессора, который производит все операции последовательно и будет служить эталоном с точки зрения скорости выполнения:

```
import time

def heavy(n):
    for x in range(1, n):
        for y in range(1, n):
            x**y

def sequential(n):
    for i in range(n):
        heavy(500)
    print(f"{n} циклов вычислений закончены")

if __name__ == "__main__":
    start = time.time()
    sequential(80)
    end = time.time()
    print("Общее время работы: ", end - start)

# 80 циклов вычислений закончены
# Общее время работы:  23.573118925094604
```

Использование потоков threading.

В следующем примере будем использовать несколько потоков для выполнения функции `heavy()`. Также произведем 80 циклов вычислений. Для этого разделим вычисления на 4 потока, в каждом из которых запустим 20 циклов:

```
import threading
import time

def heavy(n, i, thead):
    for x in range(1, n):
        for y in range(1, n):
            x**y
    print(f"Цикл № {i}. Поток {thead}")

def sequential(calc, thead):
    print(f"Запускаем поток № {thead}")
    for i in range(calc):
        heavy(500, i, thead)
    print(f"{calc} циклов вычислений закончены. Поток № {thead}")

def threaded(threads, calc):
    # threads - количество потоков
    # calc - количество операций на поток

    threads = []

    # делим вычисления на `threads` потоков
    for thead in range(threads):
        t = threading.Thread(target=sequential, args=(calc, thead))
        threads.append(t)
        t.start()

    # Подождем, пока все потоки
    # завершат свою работу.
    for t in threads:
        t.join()

if __name__ == "__main__":
    start = time.time()
    # разделим вычисления на 4 потока
    # в каждом из которых по 20 циклов
    threaded(4, 20)
    end = time.time()
    print("Общее время работы: ", end - start)
```

```
# Показано часть вывода
# ...
# ...
# ...
# Общее время работы: 43.33752250671387
```

[Однопоточный режим работы](#), оказался почти в 2 раза быстрее, потому что один поток не имеет накладных расходов на создание потоков (в нашем случае создается 4 потока) и переключение между ними.

Если бы у Python не было GIL, то вычисления функции `heavy()` происходили быстрее, а общее время выполнения программы стремилось к времени выполнения однопоточной программы. Причина, по которой многопоточный режим в данном примере не будет работать быстрее однопоточного - это вычисления, связанные с процессором и заключаются в GIL!

Если бы функция `heavy()` имела много блокирующих операций, таких как сетевые вызовы или операции с файловой системой, то применение многопоточного режима работы было бы оправдано и дало огромное увеличение скорости!

Это утверждение можно проверить смоделировав операции ввода-вывода при помощи [функции `time.sleep\(\)`](#).

```
import threading
import time

def heavy():
    # имитации операций ввода-вывода
    time.sleep(2)

def threaded(threads):
    threads = []

    # делим операции на `threads` потоков
    for thread in range(threads):
        t = threading.Thread(target=heavy)
        threads.append(t)
        t.start()

    # Подождем, пока все потоки
    # завершат свою работу.
    for t in threads:
        t.join()
    print(f"{threads} циклов имитации операций ввода-вывода закончены")

if __name__ == "__main__":
    start = time.time()
    # 80 потоков - это неправильно и показано
    # чисто в демонстрационных целях
    threaded(80)
    end = time.time()
    print("Общее время работы: ", end - start)

# 80 циклов имитации операций ввода-вывода закончены
# Общее время работы: 2.008725881576538
```

Даже если воображаемый ввод-вывод делится на 80 потоков и все они будут спать в течение двух секунд, то код все равно завершится чуть более чем за две секунды, т. к. многопоточной программе нужно время на планирование и запуск потоков.

Примечание! Каждый процессор поддерживает определенное количество потоков на ядро, заложенное производителем, при которых он работает оптимально быстро. Нельзя создавать безгранично много потоков. При увеличении числа потоков на величину, большую, чем заложил производитель, программа будет выполняться дольше или вообще поведет себя непредсказуемым образом (вплоть до зависания).

Использование многопроцессорной обработки multiprocessing.

Теперь попробуем настоящую параллельную обработку с использованием [модуля multiprocessing](#). Модуль multiprocessing во многом повторяет API [модуля threading](#), поэтому изменения в коде будут незначительны.

Для того, чтобы произвести 80 циклов вычислений функции `heavy()`, узнаем сколько процессор имеет ядер, а потом поделим циклы вычислений на количество ядер.

```
import multiprocessing
import time

def heavy(n, i, proc):
    for x in range(1, n):
        for y in range(1, n):
            x**y
    print(f"Цикл № {i} ядро {proc}")

def sequential(calc, proc):
    print(f"Запускаем поток № {proc}")
    for i in range(calc):
        heavy(500, i, proc)
    print(f"{calc} циклов вычислений закончены. Процессор № {proc}")

def processsesed(procs, calc):
    # procs - количество ядер
    # calc - количество операций на ядро

    processes = []

    # делим вычисления на количество ядер
    for proc in range(procs):
        p = multiprocessing.Process(target=sequential, args=(calc, proc))
        processes.append(p)
        p.start()

    # Ждем, пока все ядра
    # завершат свою работу.
    for p in processes:
        p.join()

if __name__ == "__main__":
    start = time.time()
    # узнаем количество ядер у процессора
    n_proc = multiprocessing.cpu_count()
    # вычисляем сколько циклов вычислений будет приходится
    # на 1 ядро, что бы в сумме получилось 80 или чуть больше
    calc = 80 // n_proc + 1
    processsesed(n_proc, calc)
    end = time.time()
    print(f"Всего {n_proc} ядер в процессоре")
    print(f"На каждом ядре произведено {calc} циклов вычислений")
    print(f"Итого {n_proc*calc} циклов за: ", end - start)

# Весь вывод показывать не будем
# ...
# ...
# ...
# Всего 6 ядер в процессоре
# На каждом ядре произведено 14 циклов вычислений
# Итого 84 циклов вычислений за:  5.0251686573028564
```

Код выполнялся почти в 5 раз быстрее. Это прекрасно демонстрирует линейное увеличение скорости вычислений от количества ядер процессора.

Использование многопроцессорной обработки с пулом.

Можно сделать предыдущую версию программы немного более элегантной, используя `multiprocessing.Pool()`. Объект пула, управляет пулом рабочих процессов, в который могут быть отправлены задания. Используя метод `Pool.starmap()`, можно произвести инициализацию функции `sequential()` для каждого процесса.

В целях эксперимента в функции запуска пула процессов `pooled(core)` предусмотрено ручное указание количества ядер процессора. Если не указывать значение `core`, то по умолчанию будет использоваться количество ядер процессора вашей системы, что является разумным выбором:

```
import multiprocessing
import time

def heavy(n, i, proc):
    for x in range(1, n):
        for y in range(1, n):
            x**y
        print(f"Вычисление № {i} процессор {proc}")

def sequential(calc, proc):
    print(f"Запускаем поток № {proc}")
    for i in range(calc):
        heavy(500, i, proc)
    print(f"{calc} циклов вычислений закончены. Процессор № {proc}")

def pooled(core=None):
    # вычисляем количество ядер процессора
    n_proc = multiprocessing.cpu_count() if core is None else core
    # вычисляем количество операций на процесс
    calc = int(80 / n_proc) if 80 % n_proc == 0 else int(80 // n_proc + 1)
    # создаем список инициализации функции
    # sequential(calc, proc) для каждого процесса
    init = map(lambda x: (calc, x), range(n_proc))
    with multiprocessing.Pool() as pool:
        pool.starmap(sequential, init)

    print (calc, n_proc, core)
    return (calc, n_proc, core)

if __name__ == "__main__":
    start = time.time()
    # в целях эксперимента, укажем количество
    # ядер больше чем есть на самом деле
    calc, n_proc, n = pooled(20)
    end = time.time()
    text = '' if n is None else 'задано '
    print(f"Всего {text}{n_proc} ядер процессора")
    print(f"На каждом ядре произведено {calc} циклов вычислений")
    print(f"Итого {n_proc*calc} циклов за: ", end - start)

# Весь вывод показывать не будем
# ...
# ...
# ...
# Всего задано 20 ядер процессора
# На каждом ядре произведено 4 циклов вычислений
# Итого 80 циклов за:  5.422096252441406
```

Из результатов работы видно, что время работы незначительно увеличилось.

Если запустить этот код, то можно проследить, что вычисления все равно происходят на том количестве ядер, которые имеются в процессоре. Только вычисления происходят поочередно - из за этого незначительное увеличение времени работы программы.

Выводы:

- Используйте модули `threading` или `asyncio` для программ, связанных с сетевым вводом-выводом, чтобы значительно повысить производительность.

- Используйте модуль multiprocessing для решения проблем, связанных с операциями ЦП. Этот модуль использует весь потенциал всех ядер в процессоре.

Содержание раздела:

- [КРАТКИЙ ОБЗОР МАТЕРИАЛА.](#)
- [Global Interpreter Lock \(GIL\)](#)

ХОЧУ ПОМОЧЬ
ПРОЕКТУ

