Urban University

Д И П Л О М Н А Я Р А Б О Т А

Сравнение различных подходов к реализации асинхронного программирования: asyncio, threading и multiprocessing

Автор: Четвериков Артем Васильевич

г. Уфа, 2024

Оглавление

Введение

Параллельное программирование — это разработка программного обеспечения, которое выполняет значительную часть вычислений одновременно (параллельно).

Цели параллельного программирования:

- решение больших задач с объёмом данных, превосходящим возможности однопроцессорной вычислительной системы;

- увеличение эффективности программ за счёт параллельного выполнения как можно большего числа операций.

Возможность параллельного программирования обусловлена тем, что большие задачи часто возможно разделить на несколько меньших подзадач, которые могут выполняться одновременно.

Различают явный и неявный подходы к параллельному программированию. В первом случае данные и вычисления распределяются программистом по узлам параллельной системы явно при создании кода приложения. При неявном подходе автоматическое распределение данных и вычислений является задачей системы программирования.

Параллельное программирование используется в таких областях, как газовая динамика, ядерная физика, молекулярная биология, гидрометеорология, геологоразведка, автоматизированное проектирование, криптография, прогнозирование бизнес-процессов и др.

Обоснование выбора темы

Актуальность темы обусловлена тем, что возрастающая сложность промышленных, технических и научных задач, решаемых с помощью компьютерных систем, предъявляет к ним требования производительности, которые не могут быть выполнены из-за ограничений, накладываемых законами физики. Использование параллельного программирования позволяет максимально эффективно использовать возможности многоядерных процессоров и многопроцессорных систем.

Перспективность темы связана с тем, что параллельные вычислительные системы интенсивно развиваются, их стоимость падает, а сферы применения расширяются. Например, речь идёт не только о специальных мощных

дорогостоящих системах, но и об обычных персональных компьютерах, для которых уже стали привычными многоядерные процессоры и видеокарты, а значит, и параллельная обработка информации.

Таким образом, выбор темы для параллельного программирования должен быть обоснован её связью с современными технологиями и необходимостью решения сложных задач.

Определение цели и задач исследования

Реализовать асинхронные задачи с использованием asyncio, threading и multiprocessing, сравнить их производительность и уместность для различных типов задач.

1 Основные понятия и определения

Параллелизм заключается в выполнении нескольких операций одновременно.

Многопроцессорность — это способ реализации параллелизма, который предполагает распределение задач между центральными процессорами (ЦП) или ядрами компьютера. Многопроцессорность хорошо подходит для задач, связанных с ЦП: к этой категории обычно относятся тесно связанные for циклы и математические вычисления.

Параллельная обработка — это более широкое понятие, чем параллелизм. Оно предполагает, что несколько задач могут выполняться одновременно. (Есть поговорка, что параллельная обработка не подразумевает параллелизм.)

Многопоточность — это модель параллельного выполнения, при которой несколько потоков по очереди выполняют задачи. Один процесс может содержать несколько потоков. Что важно знать о многопоточности, так это то, что она лучше подходит для задач, связанных с вводом-выводом. В то время как задача, связанная с процессором, характеризуется тем, что ядра компьютера постоянно работают от начала до конца, в задаче, связанной с вводом-выводом, большую часть времени занимает ожидание завершения ввода/вывода.

2 Методы и подходы к разработке

Параллелизм включает в себя как многопроцессорность (идеальную для задач, связанных с процессором), так и многопоточность (подходящую для задач, связанных с вводом-выводом). Многопроцессорность — это форма параллелизма, а параллелизм — это конкретный тип (подмножество) параллельной обработки. Стандартная библиотека Python поддерживает оба этих варианта с помощью своих multiprocessing, threading, и concurrent.futures пакетов.

Также в Python встроен отдельный дизайн: асинхронный ввод-вывод, доступный через пакет стандартной библиотеки asyncio и ключевые слова async и await.

В документации Python этот asyncio пакет позиционируется как библиотека для написания параллельного кода. Однако асинхронный ввод-вывод не является ни многопоточностью, ни мультипроцессингом. Он не основан ни на том, ни на другом.

На самом деле асинхронный ввод-вывод — это однопоточная, однопроцессная конструкция: она использует кооперативную многозадачность, термин. Другими словами, асинхронный ввод-вывод создаёт ощущение параллелизма, несмотря на использование одного потока в одном процессе. Корутины (основная функция асинхронного ввода-вывода) могут планироваться одновременно, но они не являются по своей сути параллельными.

Асинхронный ввод-вывод — это стиль параллельного программирования, но не параллелизм. Он больше похож на многопоточность, чем на многопроцессорность, но сильно отличается от них обоих и является самостоятельным элементом в наборе инструментов для параллельного программирования.

Асинхронные процедуры могут «приостанавливаться» в ожидании конечного результата и тем временем запускать другие процедуры.

Асинхронный код с помощью описанного выше механизма обеспечивает параллельное выполнение. Другими словами, асинхронный код имитирует параллельное выполнение.

Асинхронный ввод-вывод позволяет сократить время ожидания, в течение которого функции в противном случае блокировались бы, и даёт возможность другим функциям выполняться во время простоя.

Многопоточность масштабируется менее эффективно, чем асинхронный

ввод-вывод, потому что потоки — это системный ресурс с ограниченной доступностью. Создание тысяч потоков приведёт к сбою на многих компьютерах, и это не рекомендуется делать. Создание тысяч задач асинхронного ввода-вывода вполне возможно.

Асинхронный ввод-вывод полезен, когда у вас есть несколько задач, связанных с вводом-выводом, в которых в противном случае преобладал бы блокирующий ввод-вывод, например:

- сетевой ввод-вывод, независимо от того, является ли программа серверной или клиентской
- бессерверные системы, такие как одноранговая многопользовательская сеть, например групповой чат
- операции чтения/записи, при которых нужно имитировать стиль «запустил и забыл», но при этом не беспокоиться о блокировке того, что читается и записывается.

3 Обзор библиотек для параллельного программирования на Python

3.1 Библиотека asyncio

Asyncio — это библиотека в стандартной библиотеке Python, которая предоставляет инфраструктуру для написания параллельного кода с использованием концепции асинхронного программирования.

Она позволяет эффективно обрабатывать многочисленные задачи ввода-вывода (например, сетевые операции или чтение/запись из файлов) без необходимости создавать множество потоков или процессов.

Некоторые ключевые концепции asyncio:

Событийный цикл (Event Loop). Это ядро asyncio, которое отвечает за планирование и выполнение задач (корутин). Работает по принципу однопоточной многозадачности, постоянно опрашивает очередь событий и выполняет соответствующие задачи.

Корутины (Coroutines). Это специальные функции, выполнение которых можно приостанавливать на определённых точках с помощью ключевого слова await. Используются для написания асинхронного, событийно-ориентированного кода.

Задачи (Tasks). Представляют собой обертки над корутинами для их планирования и выполнения. Задачи можно отменять, объединять с другими задачами и отслеживать на предмет исключений.

Будущие объекты (Futures). Представляют результат асинхронной операции, который может быть доступен позже. Могут быть использованы для координации между разными частями программы.

Asyncio оптимален именно для I/O-bound задач (сеть, файловые операции и т. д.), и не подходит для CPU-bound операций (вычислительно-интенсивных задач), поскольку в этом случае он не сможет переключаться на другие задачи во время блокировки.

Некоторые инструменты asyncio:

- пулы для управления ограниченными ресурсами (потоки, подпроцессы);
- синхронизаторы для координации между корутинами (Lock, Event, Condition);
- очереди для безопасной передачи данных между корутинами;
- сигналы для обработки внешних событий (UNIX-сигналы).

3.2 Библиотека threading

Библиотека threading в Python предназначена для многопоточности, то есть выполнения приложения сразу в нескольких потоках, которые отвечают за выполнение его функций одновременно.

Некоторые преимущества библиотеки:

Простое использование. Начать работу с ней достаточно легко.

Простота передачи данных из потока в основное приложение. Допускается использование глобальных переменных, но в этом случае программное обеспечение должно быть грамотно спроектировано.

Оптимальное решение для работы с потоками на одноядерном компьютере или при небольшой нагрузке на процессор.

Для подключения библиотеки её не нужно устанавливать, она поставляется вместе с интерпретатором. Достаточно подключить модуль с помощью команды: import threading.

В библиотеке представлен класс Thread для создания потока выполнения. Задание исполняемого кода в отдельном потоке возможно двумя способами: передача исполняемого объекта (функции) в конструктор класса или переопределение функции run() в классе-наследнике. После того, как объект создан, поток запускается путём вызова метода start().

Метод join() используется для блокирования исполнения родительского потока до тех пор, пока созданный поток не завершится. Это нужно в случаях, когда для работы потока-родителя необходим результат работы потока-потомка.

3.3 Библиотека multiprocessing

Библиотека multiprocessing в Python позволяет использовать несколько процессов для параллельного выполнения кода, что обеспечивает более эффективное использование многоядерных процессоров.

Для каждой задачи, которая должна выполняться одновременно, создаётся новый процесс. У каждого процесса свой интерпретатор Python и пространство памяти, что позволяет ему работать независимо от других процессов.

Некоторые ключевые компоненты библиотеки multiprocessing:

Класс Process. Используется для создания и управления независимыми процессами.

Класс Queue. Общая очередь заданий, которая позволяет безопасно обмениваться данными и координировать процессы. Используется для передачи сообщений или результатов между экземплярами процессов.

Класс Pipes. Предоставляет способ установить канал связи между процессами. Полезен для двунаправленной связи между двумя процессами.

Некоторые преимущества использования библиотеки multiprocessing:

- лучшее использование процессора для высокоинтенсивных задач;
- больший контроль над дочерними процессами по сравнению с потоками;
- простая реализация задач, подходящих для параллельного программирования.

Однако у использования multiprocessing есть и недостатки: код становится более сложным, так как приходится управлять несколькими процессами, особенно при работе с общими данными и синхронизацией процессов.

4 Проектирование приложения

4.1 Выбор библиотек и инструментов

При реализации приложения будем использовать наиболее популярные библиотеки для параллельного программирования: asyncio, threading и multiprocessing.

Так же для отображения и сохранения результатов исследований воспользуемся библиотеками pathlib, matplotlib, Pillow.

4.2 Определение структуры приложения

Реализуем приложение для анализа производительности работы библиотек asyncio, threading и multiprocessing в виде отдельных модулей.

Приложение будет реализовано без интерфейсной части, запускаются модули из командной строки или в IDE.

Замер производительности будем выполнять на нескольких типовых задачах:

- I/O-bound задача: работа с сетевыми запросами или операциями, которые включают ожидание отклика от внешнего ресурса, в нашем случае будем скачивать страницу сайта в виде html и записывать ее в файл;

- CPU-bound задача: выполнение сложных математических операций, в нашем случае — вычисление факториала числа;

- Задача с высокой параллельностью: массовая обработка небольших файлов, в нашем случае — поворот изображения на 90 градусов.

Также замер производительности выполним при обычном, синхронном, программировании.

4.3 Реализация основного функционала

Создаем по четыре модуля для каждого вида задач, всего 12 модулей.

Для уменьшения влияния случайных погрешностей пакеты однотипных задач будем запускать по 50 раз и вычислять среднее время выполнения обработки пакета задач.

Полученные значения времени выполнения отобразим на графике с возможностью его сохранения в файл изображения.

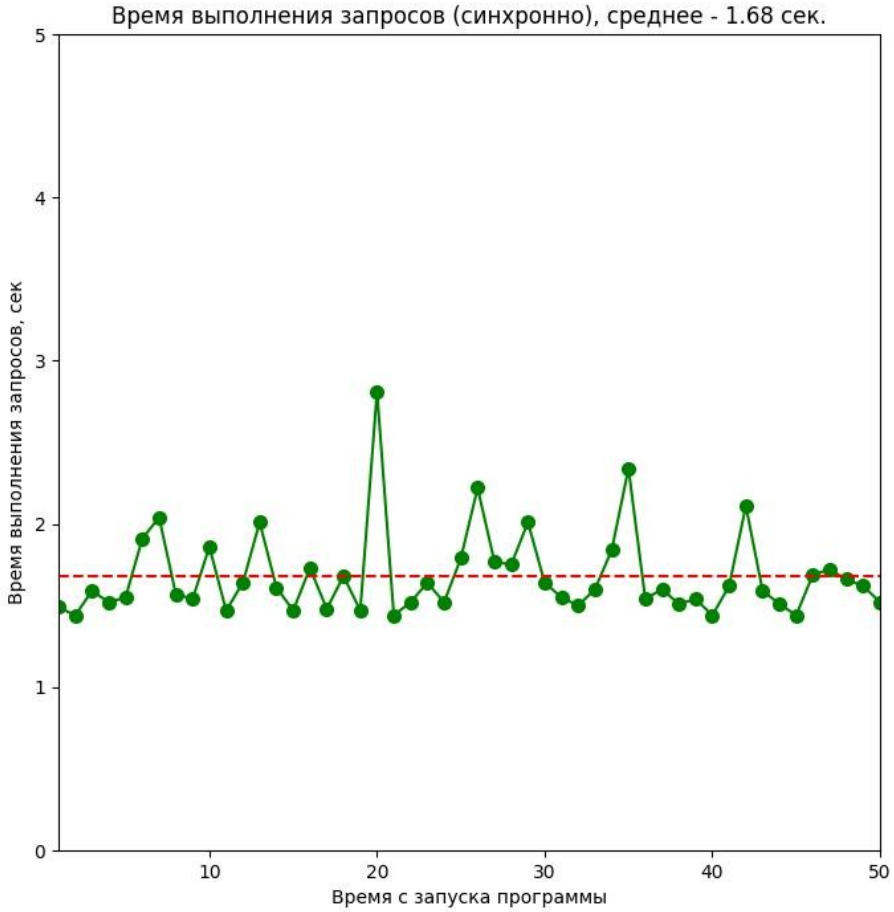
5 Анализ и интерпретация результатов

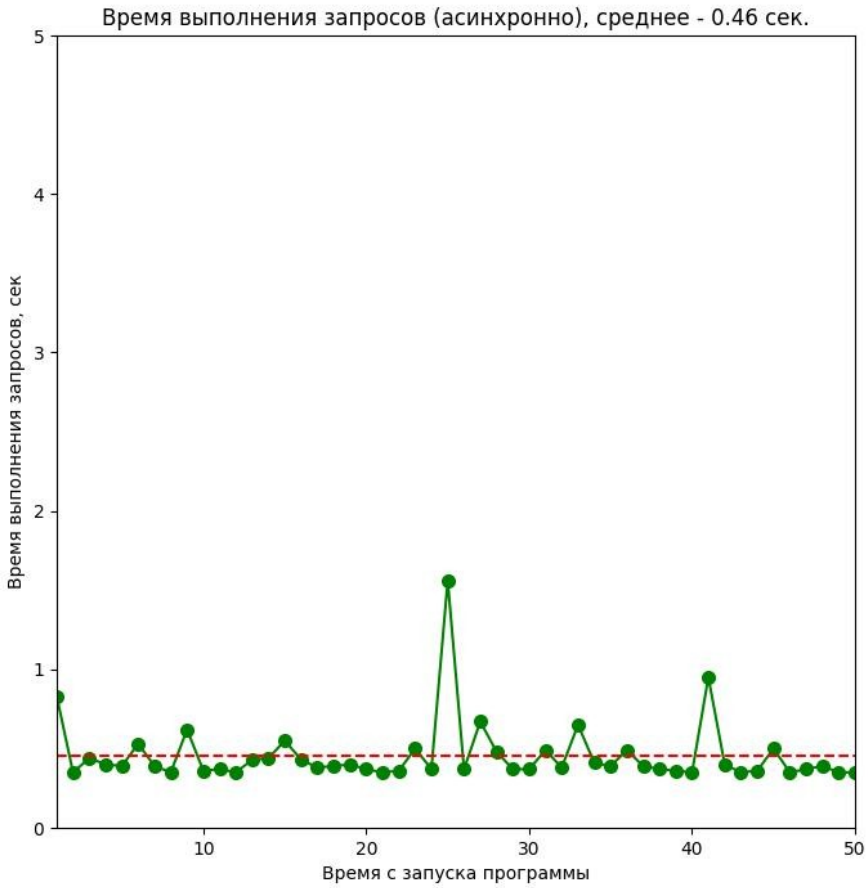
5.1 I/O-bound задача

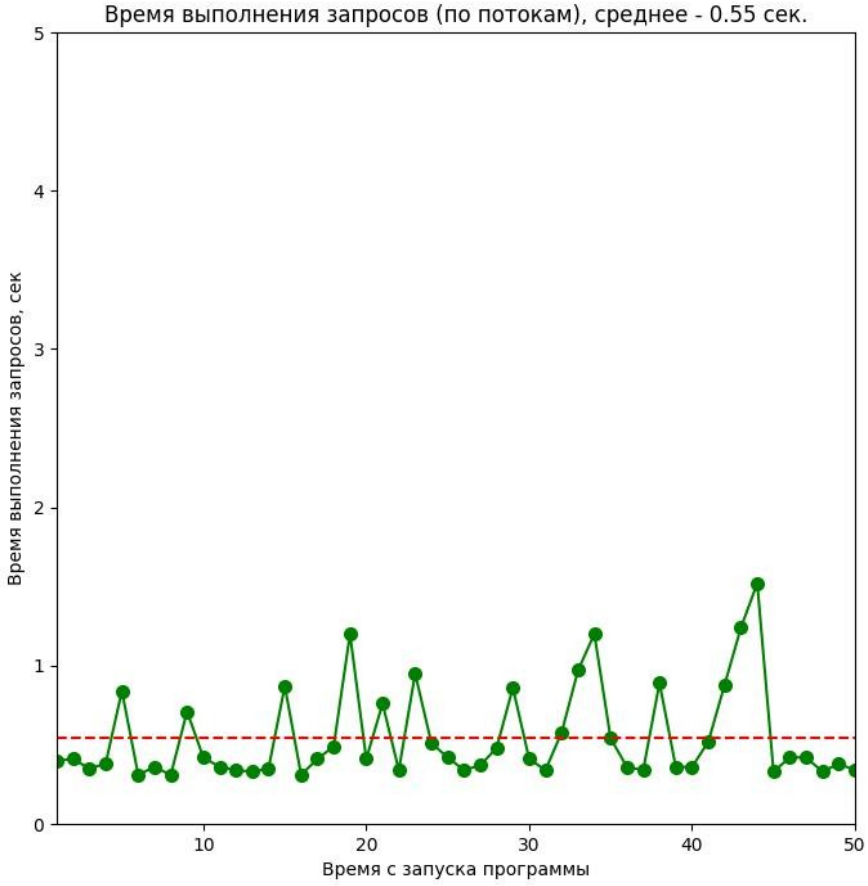
По итогам запуска четырех модулей для загрузки html страниц и записи их в файл получили следующие средние результаты:

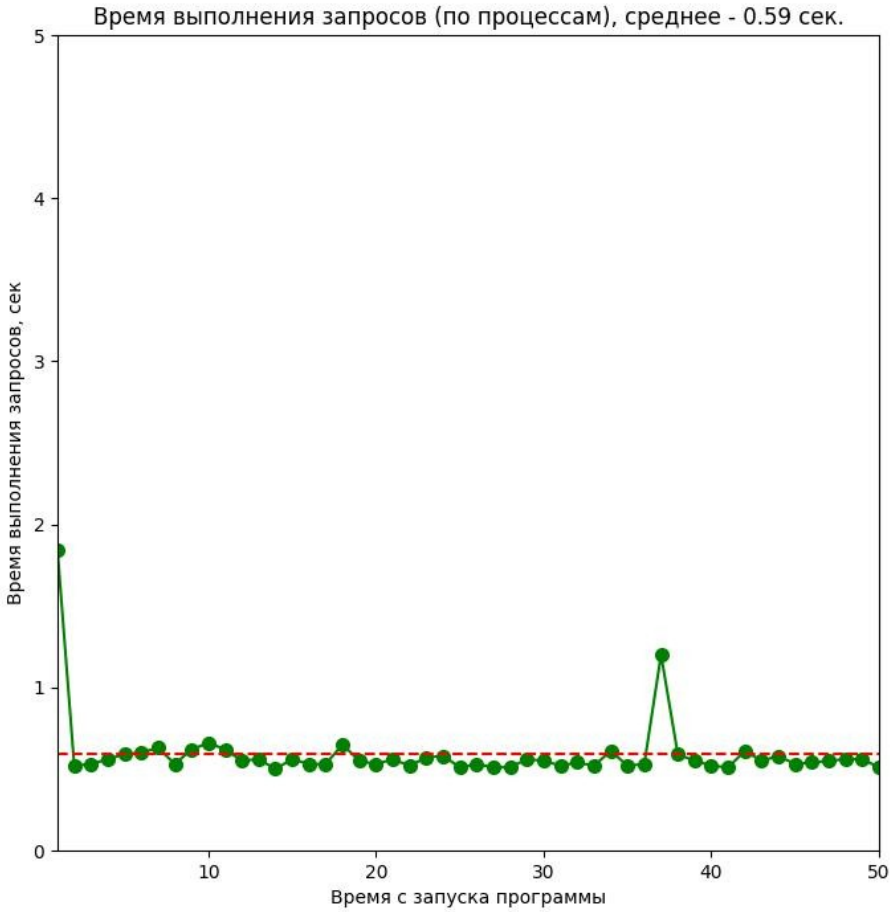
- синхронно: 1,68 сек;
- асинхронно: 0,46 сек;
- по потокам: 0,55 сек;
- по процессам: 0,59 сек.

Асинхронный подход показал преимущество при операции ввода-вывода (например, загрузка страниц и запись/чтение файлов), так как такие задачи обычно ждут, пока операционная система завершит запрос. Асинхронность позволяет выполнять другие задачи в это время, что и приводит к ускорению.




Время выполнения запросов (асинхронно), среднее - 0.46 сек.
Время выполнения запросов, сек
Время с запуска программы
0
1
2
3
4
5
10
20
30
40
50


Время выполнения запросов (по потокам), среднее - 0.55 сек.
Время выполнения запросов, сек
Время с запуска программы
0
1
2
3
4
5
10
20
30
40
50

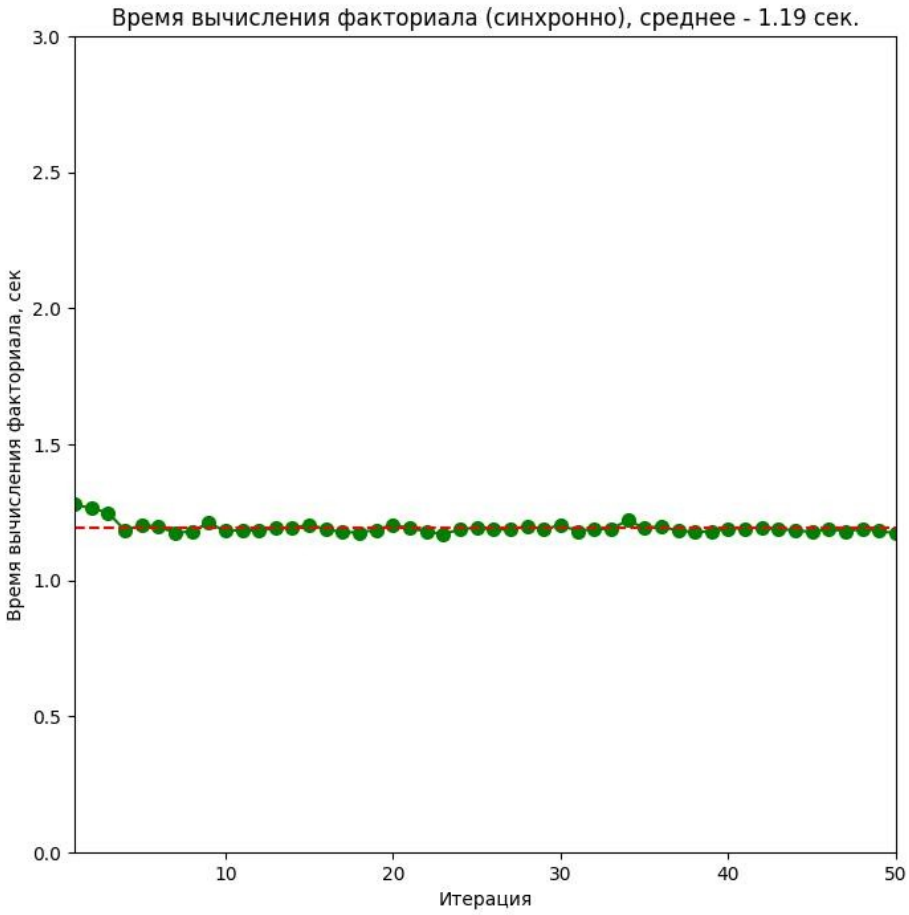

Время выполнения запросов (по процессам), среднее - 0.59 сек.
Время выполнения запросов, сек
Время с запуска программы
0
1
2
3
4
5
10
20
30
40
50

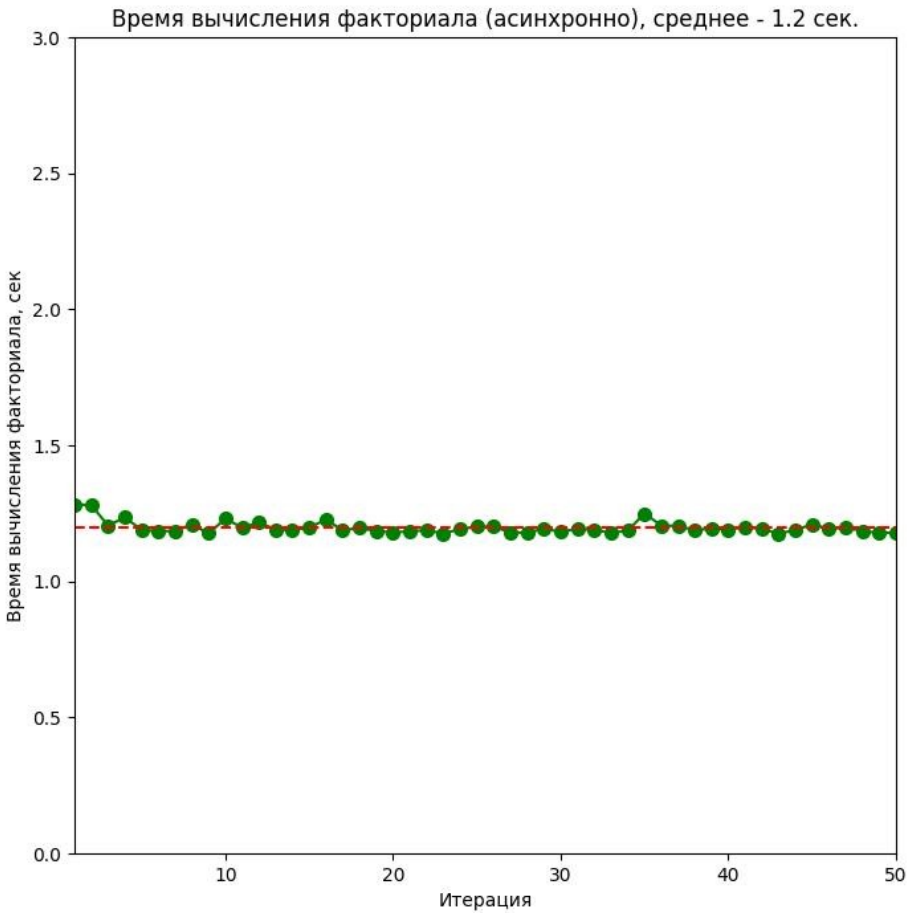
5.2 CPU-bound задача

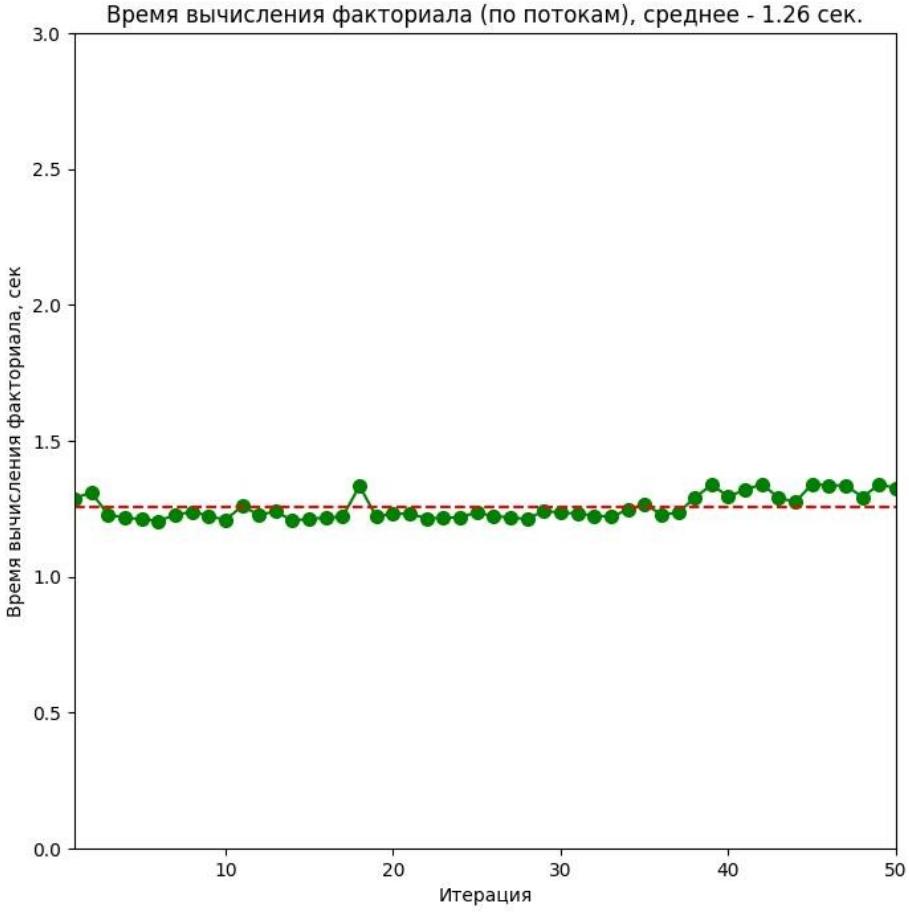
По итогам запуска четырех модулей для вычисления факториала получили следующие средние результаты:

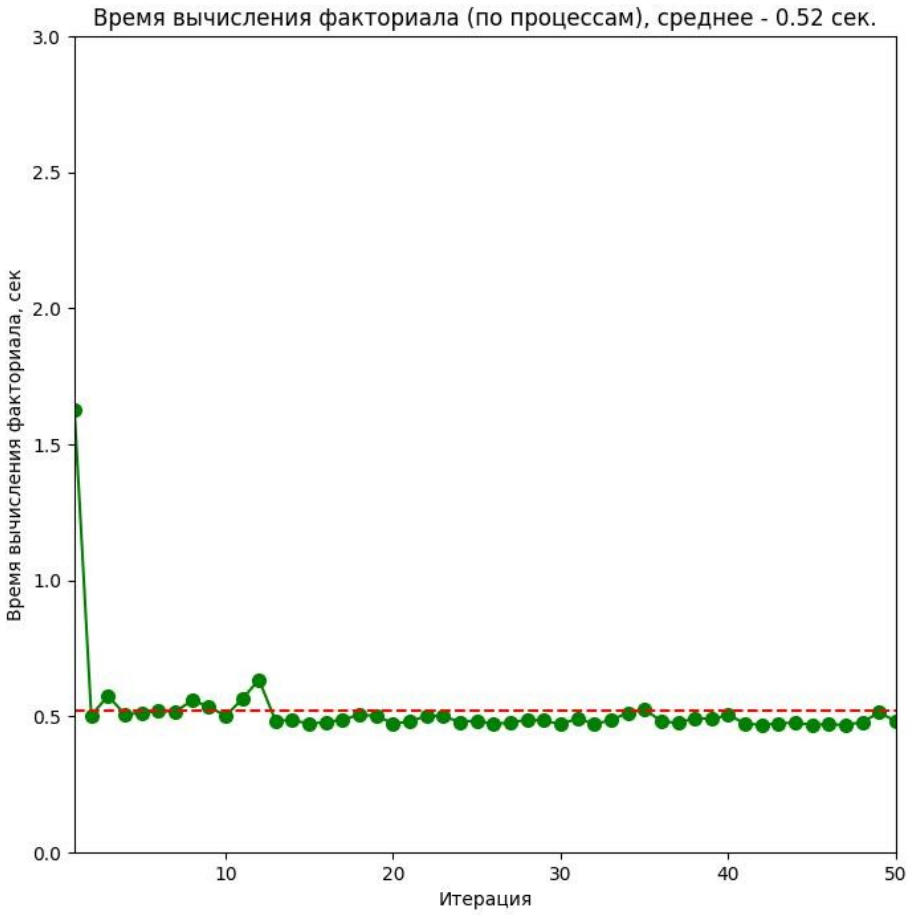
- синхронно: 1,19 сек;
- асинхронно: 1,20 сек;
- по потокам: 1,26 сек;
- по процессам: 0,52 сек.

Объяснением подобных результатов может служить то, что процессы лучше справляются с интенсивными вычислительными задачами, такими как вычисление факториала или любые операции, требующие значительных ресурсов CPU. Это связано с тем, что процессы используют отдельные ядра процессора, обходя ограничения GIL в Python.




Время вычисления факториала (асинхронно), среднее - 1.2 сек.
Время вычисления факториала, сек
Итерация
0.0
0.5
1.0
1.5
2.0
2.5
3.0
10
20
30
40
50


Время вычисления факториала (по потокам), среднее - 1.26 сек.
Время вычисления факториала, сек
Итерация
0.0
0.5
1.0
1.5
2.0
2.5
3.0
10
20
30
40
50

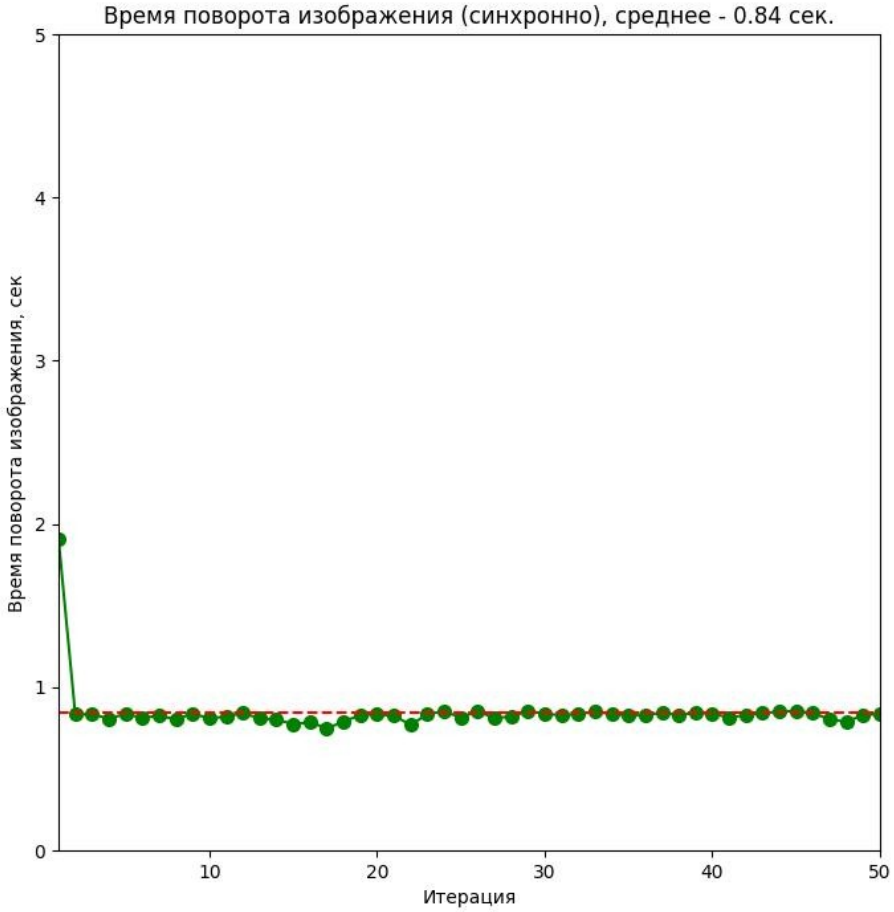

Время вычисления факториала (по процессам), среднее - 0.52 сек.
Время вычисления фактриала, сек
Итерация
3.0
2.5
2.0
1.5
1.0
0.5
0.0
10
20
30
40
50

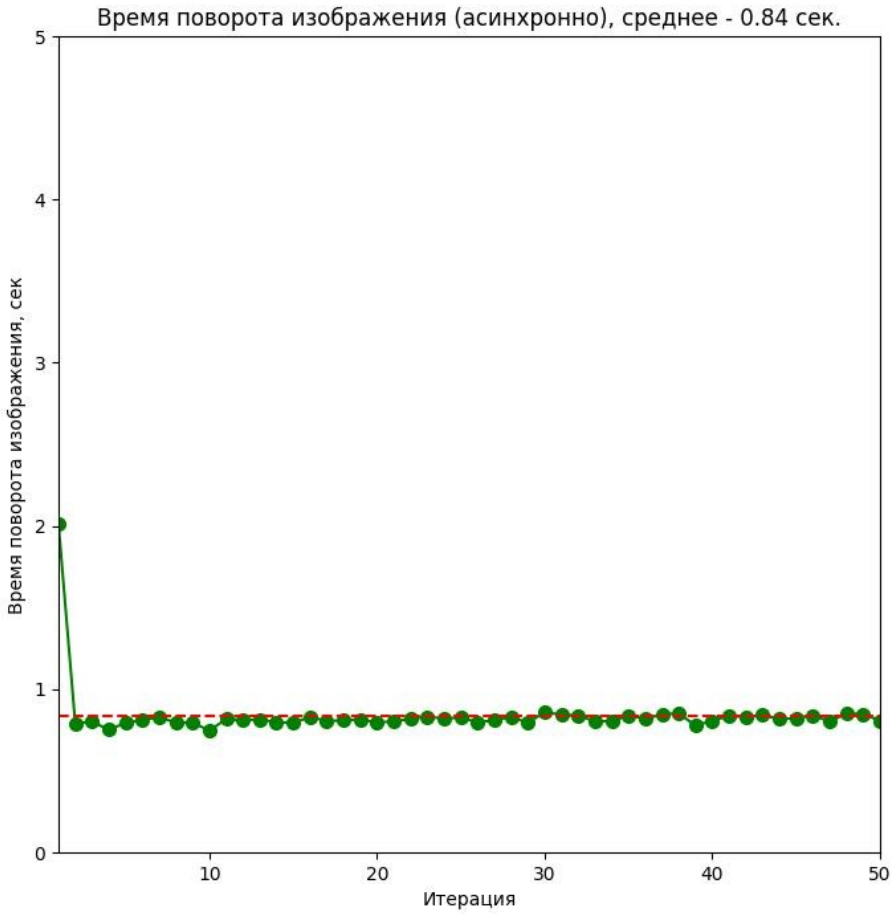
5.3 Задача с высокой параллельностью

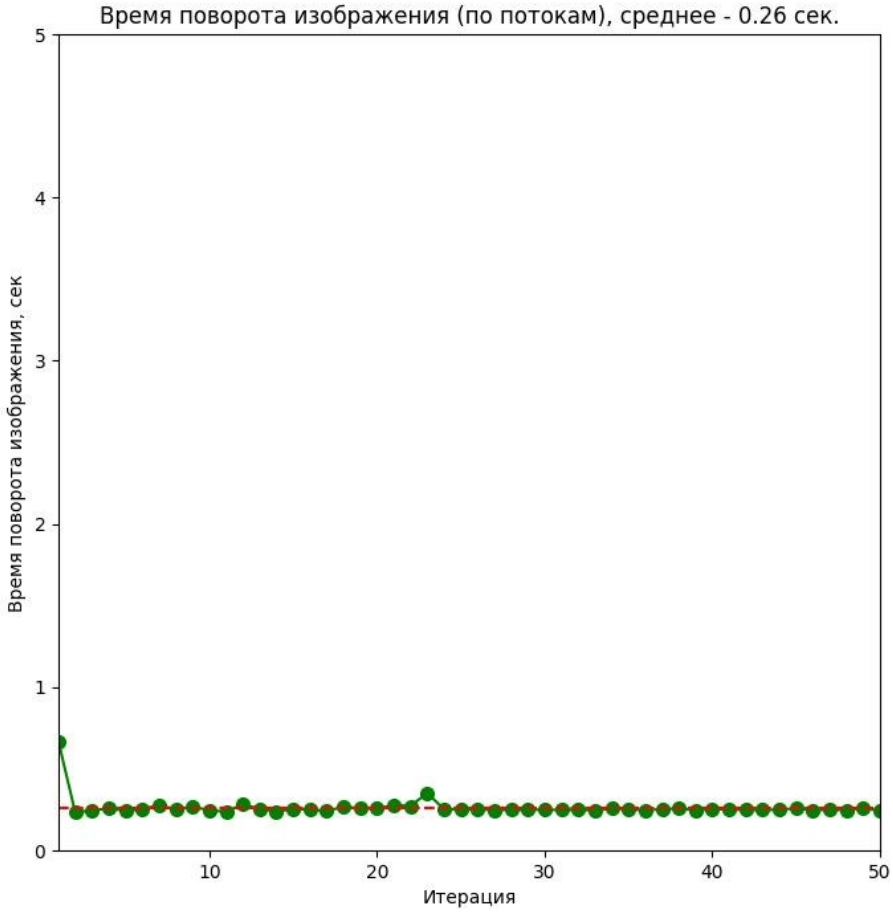
По итогам запуска четырех модулей для поворота изображений получили следующие средние результаты:

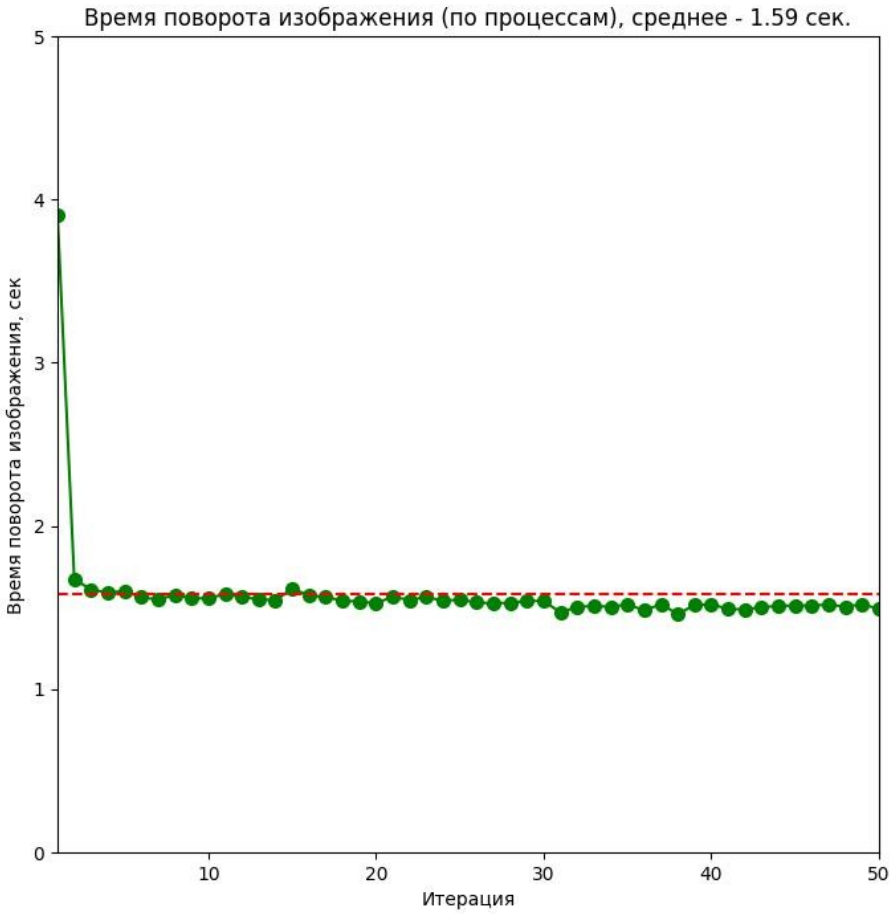
- синхронно: 0,84 сек;
- асинхронно: 0,84 сек;
- по потокам: 0,26 сек;
- по процессам: 1,59 сек.

При редактировании изображений потоки оказались эффективными. А процессы, напротив, показали себя наиболее медленными — возможно, это связано с необходимостью разделения довольно объемных изображений между процессами. Python-потоки хорошо работают с задачами, где значительная часть нагрузки связана с вводом-выводом или менее ресурсоемкими вычислениями.




Время поворота изображения (асинхронно), среднее - 0.84 сек.
Время поворота изображения, сек
Итерация
0
1
2
3
4
5
10
20
30
40
50


Время поворота изображения (по потокам), среднее - 0.26 сек.
Время поворота изображения, сек
Итерация
0
1
2
3
4
5
10
20
30
40
50


Время поворота изображения (по процессам), среднее - 1.59 сек.
Время поворота изображения, сек
Итерация
0
1
2
3
4
5
10
20
30
40
50

Заключение

В заключении можно сказать, что каждый из подходов к реализации асинхронного программирования (asyncio, threading и multiprocessing) имеет свои особенности и подходит для разных типов задач:

Asyncio идеален для задач, связанных с вводом-выводом, где требуется высокая производительность и масштабируемость.

Threading хорошо подходит для параллельной обработки задач, связанных с ожиданием, особенно в многозадачных приложениях.

Multiprocessing эффективен для задач, требующих интенсивных вычислений и использования нескольких ядер процессора.

Таким образом, выбор между этими подходами зависит от конкретных требований и задач разработки. Полученные результаты могут быть полезны для выбора оптимального подхода при разработке высокопроизводительных приложений на Python.