УДК 004.432

ПЕРСПЕКТИВЫ РОСТА ПРОИЗВОДИТЕЛЬНОСТИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ С ПОМОЩЬЮ ТЕХНОЛОГИИ СУБИНТЕРПРЕТАТОРОВ В PYTHON

Р. Д. Синицын[0009-0003-6750-5222]

МИРЭА – Российский технологический университет

sinicinr2003@mail.ru

Аннотация

Рассмотрена проблема влияния глобальной блокировки интерпретатора на производительность многопоточных приложений в Python. Описана концепция субинтерпретаторов как одного из решений, позволяющих обходить ограничения GIL и обеспечивать эффективное параллельное выполнение кода. Проведен сравнительный анализ субинтерпретаторов с традиционными методами параллельных вычислений, такими как использование процессов и потоков. Результаты экспериментов показали, что субинтерпретаторы значительно повышают производительность в условиях высоких вычислительных нагрузок. Кроме того, исследованы возможности применения субинтерпретаторов в веб-разработках. Отмечены преимущества использования названного подхода для обработки запросов и управления ресурсами в современных веб-приложениях, что может значительно улучшить их масштабируемость и отклик. Новизна проведенного исследования заключается в глубоком анализе субинтерпретаторов в контексте конкретных сценариев использования, что ранее не получило достаточного освещения в научной литературе. Результаты работы подчеркивают необходимость дальнейшего изучения субинтерпретаторов как альтернативного подхода в Python и интерес к этому разработчиков и исследователей в области высокопроизводительных вычислений.

Ключевые слова*: Python, CPython, PEP, GIL, субинтерпретатор, многопоточность, многопроцессорность, асинхронность, интерпретатор, параллельные вычисления.*

ВВЕДЕНИЕ

В эпоху стремительного цифрового прогресса и роста вычислительных мощностей параллельные вычисления становятся неотъемлемой частью современных разработок. Они позволяют эффективно использовать доступные ресурсы, значительно повышая производительность и масштабируемость приложений. С выходом Python 3.12 [1] был представлен экспериментальный API для субинтерпретаторов, который предлагает инновационный подход к управлению параллелизмом, особенно в задачах, интенсивно использующих процессорные ресурсы. Этот шаг является значительным прорывом в контексте ограничений, связанных с глобальной блокировкой интерпретатора (GIL), которая долгое время сдерживала потенциал Python для истинного параллелизма.

Субинтерпретаторы позволяют реализовывать многопоточность без влияния GIL, обеспечивая возможность параллельного выполнения кода в рамках одного процесса. Эта модель открывает новые горизонты для программистов, стремящихся к более эффективному использованию многоядерных систем. В настоящей статье рассмотрены основные принципы работы субинтерпретаторов, их роль в параллельных вычислениях, а также принципиальные отличия от существующих подходов.

Разработка и интеграция эффективных методов обработки параллельных вычислений принципиально необходимы в свете растущих потребностей в высокопроизводительных вычислениях в таких областях, как наука о данных, искусственный интеллект и интернет вещей. Введение нового API для субинтерпретаторов не только открывает новые пути для повышения производительности, но и ставит перед разработчиками и исследователями важные вопросы о будущем параллельных вычислений на Python.

Цель настоящей статьи — дать всестороннее представление о новых возможностях, которые открывает использование субинтерпретаторов, а также оценить их потенциальное влияние на производительность параллельных вычислений и эффективность разработки приложений на Python. Читатели смогут глубже понять перспективы и вызовы, которые стоят перед сообществом Python в эру растущих требований к параллелизму и многопоточности.

ПРОБЛЕМА ИСПОЛЬЗОВАНИЯ GIL

Глобальная блокировка интерпретатора (GIL) — это механизм, используемый в CPython для синхронизации потоков и позволяющий выполнять только один поток в любой момент времени [2]. Эта блокировка связана с тем, что управление памятью в Python не является потокобезопасным. Однако GIL создает проблемы, особенно в многопоточных программах, ориентированных на вычисления. Например, при запуске четырех потоков Python на четырехъядерном процессоре одновременно будет работать только один поток. Это приводит к тому, что многопоточные программы, требующие высокой процессорной загрузки, будут работать медленнее, чем однопоточные. Хотя процессор имеет несколько ядер, Python может использовать только одно из них, что делает выполнение вычислений менее эффективным [3].

Дополнительная нагрузка на систему от переключений контекста также негативно сказывается на производительности многопоточных программ. В последних версиях Python стандартный интервал для переключения потоков составляет 5 миллисекунд. Уменьшив этот интервал, мы можем наблюдать заметное замедление работы многопоточных приложений. В итоге использование потоков в Python будет менее эффективным для задач с высокой вычислительной нагрузкой по сравнению с задачами, связанными с вводом-выводом. Убедиться в этом можно, например, создав список и рассчитав сумму всех чисел в этом списке (см. рис. 1):

```
import numpy


2 usages
def sum_range(range):
    res = 0
    for el in range:
        res += el
    return res


range = numpy.random.randint(0, 100, 100_000)
sum_range(range)
```

Рис. 1. Код расчёта суммы чисел последовательности, выполняемого в однопоточном режиме

Предполагается, что разделение вычислительной нагрузки на фрагменты и распределения ее по потокам приведёт к уменьшению времени исполнения программы.

```
from threading import Thread

import numpy

ranges = numpy.random.randint(0, 100, 100_000)
threads = []

for range in numpy.split(ranges, 100):
    thread = Thread(target=sum_range, args=(range,))
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
```

Рис. 2. Код расчёта суммы чисел последовательности, выполняемого в многопоточном режиме

Однако во втором примере расчёт выполняется в два раза медленнее, чем в первом. Это связано с тем, что все потоки привязаны к одному и тому же GIL, и в любой момент времени будет выполняться только один поток. Дополнительное время возникает из-за создания потоков и их переключений между собой, исходя из чего такой подход даёт малый прирост в производительности вычислений.

Запуск нескольких интерпретаторов в рамках одного процесса существовал с версии Python 1.5, но в таком подходе возникало достаточное большое количество проблем, связанных с разделением значительной части глобального состояния между интерпретаторами. PEP684 [4] окончательно изменил это, прекратив совместное использование GIL между интерпретаторами – теперь каждый интерпретатор в процессе Python имеет свой собственный GIL и, следовательно, может работать параллельно. Основной причиной того, что работа над GIL для каждого интерпретатора занимала много времени, было то, что CPython внутренне был основан на GIL как на источнике потокобезопасности. Из-за этого возникло много расширений, написанных на C и имеющих глобальное общее состояние.

В Python 3.12 и 3.13 расширения стандартной библиотеки Python, написанные на C, тестируются, и любое глобальное общее состояние перемещается в новый API, который помещает это состояние либо в модуль, либо в состояние интерпретатора. Даже когда эта работа будет окончательно завершена, сторонние расширения, написанные на C, придётся тестировать в парадигме субинтерпретаторов.

ЧТО ТАКОЕ СУБИНТЕРПРЕТАТОР

Как известно, системная архитектура языка программирования Python состоит из трёх основных частей [5]:

- Процесс Python, содержащий один или несколько интерпретаторов;
- Интерпретатор, содержащий GIL и один или несколько потоков Python;
- Поток, содержащий информацию о текущем выполняемом коде.

Субинтерпретатор — это интерпретатор, который запускает код Python параллельно с основным интерпретатором, в том же процессе и в том же адресном пространстве, но с полностью изолированными ресурсами выполнения. Это способ добиться изоляции памяти и разделить выполнение кода в зависимости от функциональности или предметной области.

Субинтерпретаторы в Python могут быть полезны в нескольких случаях использования. Их можно использовать для параллельного выполнения кода на отдельных ядрах процессора, что может позволить повысить производительность программ, связанных с нагрузкой на процессор. Субинтерпретаторы также можно использовать для изоляции различных частей программы для повышения безопасности и снижения риска конфликтов, поскольку каждый субинтерпретатор работает в своей отдельной изолированной среде со своими собственной памятью и состоянием. Это может быть полезно в ситуациях, когда необходимо запустить ненадежный код или есть потребность в изоляции определенных частей программы. Кроме того, субинтерпретаторы можно использовать для лучшего параллелизма в программах, связанных с вводом-выводом.

С момента появления Python 1.5 разработчики внедрили C-API, позволяющее использовать несколько интерпретаторов. Однако это решение сталкивалось с серьёзными ограничениями из-за наличия GIL, что препятствовало настоящему

параллелизму и вызывало сложности. Поэтому наиболее распространённым методом для параллельного выполнения кода был встроенный модуль multiprocessing [6].

Разработчики проводили исследование различных подходов для эффективной работы с многоядерным Python, но у каждого из предложенных решений были свои недостатки, и ни одно из них не оказалось простым:

- Существующая практика реализации GIL в модулях расширения никак не помогала с кодом, написанным на Python;
- Другие реализации языка Python, такие как Jython и IronPython, являлись не столь популярными решениями, так как CPython доминирует в сообществе;
- Удаление GIL являлось слишком большим техническим риском на тот момент;
- Проект Трента Нельсона PyParallel являлся неполным и только поддерживающим платформу Windows [7];
- Изменение модуля multiprocessing представляло собой обширный пласт работы, чтобы сделать его достаточно эффективным;
- Другие инструменты параллелизма, такие как dask, ray, MPI, не подходят для среды выполнения.

В ходе проведенных исследований в 2014 году стало достаточно ясно, что решение с использованием изолированных интерпретаторов не несёт в себе высокого уровня технического риска и что большую часть работы в любом случае стоит проделать [8].

В 2017 году разработчики ядра CPython представили инициативу по изменению структуры интерпретаторов с целью улучшения их изоляции от процесса владельца Python и возможности параллельной работы. Эта масштабная задача оказалась сложной и до сих пор остаётся до конца незавершённой. Она была разделена на два PEP: PEP684 [4], который предлагает замену глобальной блокировки интерпретатора (GIL) для каждого из интерпретаторов, и PEP554 [9], который вводит API для создания субинтерпретаторов и обмена данными между ними.

В 2023 году было одобрено предложение PEP703 [10] сделать GIL необязательным в CPython, которое работает совместно с GIL для каждого интерпретатора. Для этого имелись такие предпосылки, как «бессмертные» объекты и обновление расширений, написанных на C, чтобы не использовать общее глобальное состояние [11].

Ещё одним важным моментом является то, что, хотя PEP703 [10] был принят, это было сделано с оговоркой, что это необязательный флаг, и если он окажется слишком проблематичным или сложным, то изменения будут отменены в будущем. С другой стороны, GIL для каждого интерпретатора в значительной степени уже завершён и не требует альтернативного флага во время компиляции в CPython.

СРАВНЕНИЕ СУБИНТЕРПРЕТАТОРОВ С ДРУГИМИ МЕТОДАМИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Python имеет несколько вариантов параллельного выполнения, которые могут быть выбраны в зависимости от задачи:

- Потоки – создаются быстро и позволяют совместно использовать объекты с небольшими накладными расходами. Однако их основной недостаток заключается в том, что они подвержены глобальной блокировке интерпретатора (GIL). Это означает, что при высокой нагрузке на процессор прирост производительности будет минимальным. Тем не менее потоки отлично подходят для фоновых задач, таких как прослушивание сообщений в очереди;
- Сопрограммы – создаются очень быстро и позволяют совместно использовать любые объекты с минимальными накладными расходами. Сопрограммы отлично подходят для операций ввода-вывода [12];
- Многопроцессорность — этот подход создаёт отдельные процессы, которые работают параллельно и не зависят друг от друга. Создание процессов происходит медленно, поэтому для достижения заметного прироста производительности нагрузка должна быть достаточно высокой. В отличие от потоков, каждый процесс имеет свой собственный GIL, что позволяет реализовать полноценное параллельное выполнение;
- Субинтерпретаторы – новая концепция, обладающая параллелизмом многопроцессорной обработки, но с гораздо более быстрым запуском.

Для лучшего понимания итоговые результаты сравнительного анализа моделей параллельного выполнения обработки сведены в таблицу 1.

Таблица 1. Анализ моделей параллельности

Модель	Время запуска	Исполнение	Обмен данными
Сопрограммы	Самое малое	Условно параллельное	Любой
Потоки	Малое	Условно параллельное	Любой
Процессы	Долгое	Параллельное	Сериализация
Субинтерпретаторы	Среднее	Параллельное	Сериализация или общая память

Для сравнения производительности субинтерпретаторов в реальном выражении проведен тест скорости создания потоков, процессов и субинтерпретаторов. Результаты скорости создания 100 сущностей представлены на рис. 4.

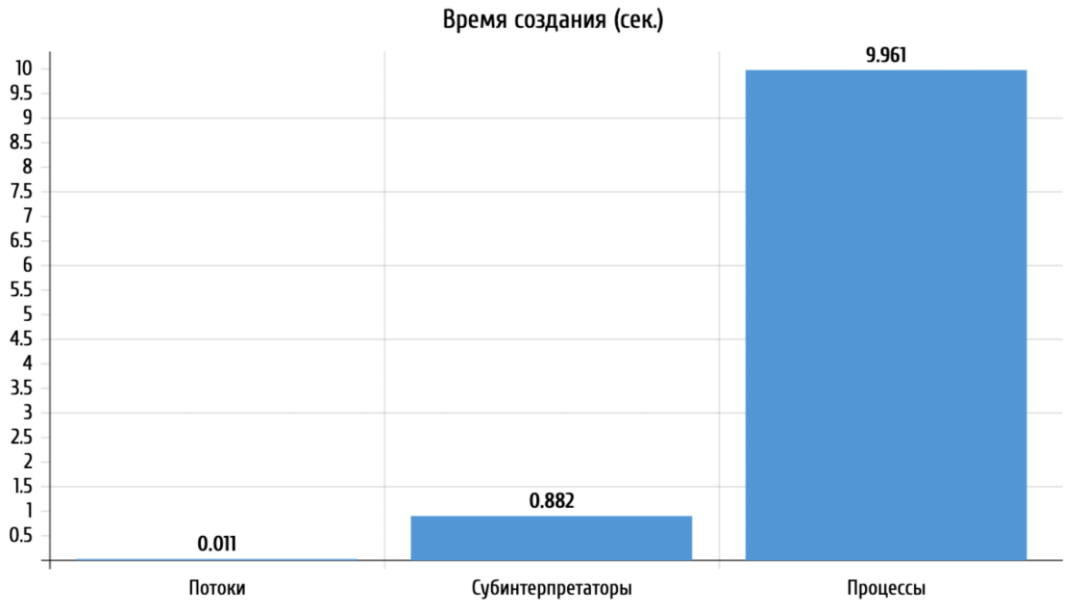

Рис. 4. График скорости запуска 100 сущностей в различных моделях параллельных вычислений

Данный тест показал, что многопоточность запускается примерно в 100 раз быстрее, чем субинтерпретаторы, которые примерно в 10 раз быстрее, чем многопроцессорная обработка. На приведенном рисунке явно видно, что различие в скорости создания интерпретаторов и потоков намного меньше, чем для процессов.

Далее проведено тестирование скорости выполнения параллельных вычислений с интенсивным использованием процессора для расчёта числа π с точностью до 2000 десятичных знаков. Результаты эффективности расчётов показаны на рис. 5.

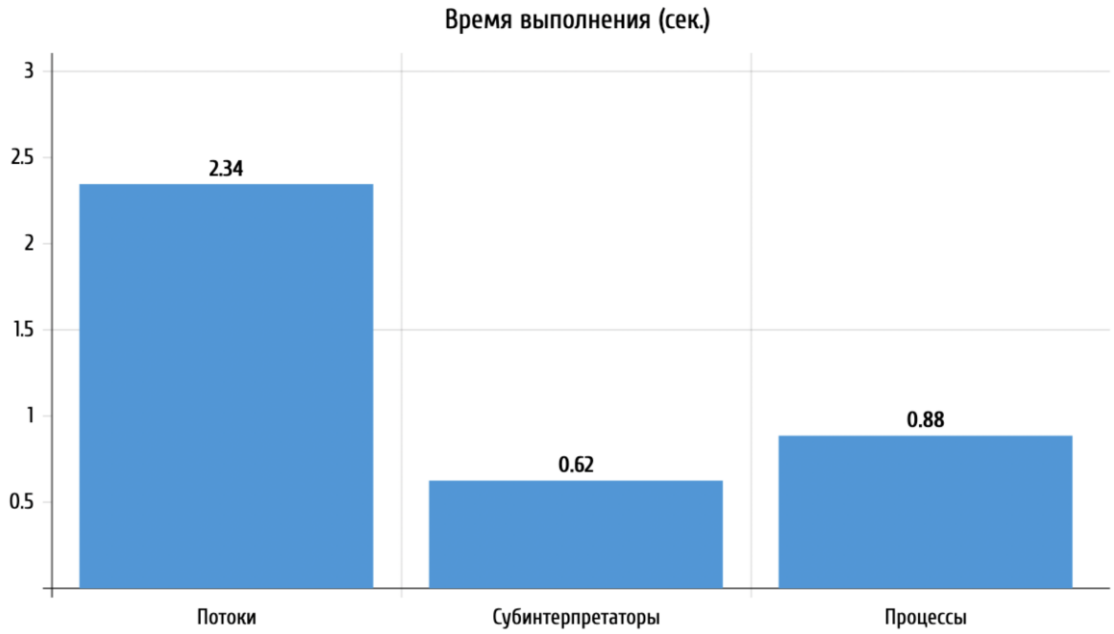

Рис. 5. Скорости вычисления числа π с точностью до 2000 десятичных знаков в разных моделях параллельных вычислений

Из рис. 5 видно, что в данном случае наиболее быстрым способом вычисления оказались субинтерпретаторы. Однако результаты первого теста показали, субинтерпретаторы также запускаются в 100 раз медленнее, чем потоки. Таким образом, если задача небольшая, например, вычисление числа π до 200 цифр, то преимущества накладных расходов при запуске перевешивают параллелизм, и обработка потоков выполняется быстрее, что видно на рис. 6 (скорости вычислении числа π до 200 знаков).

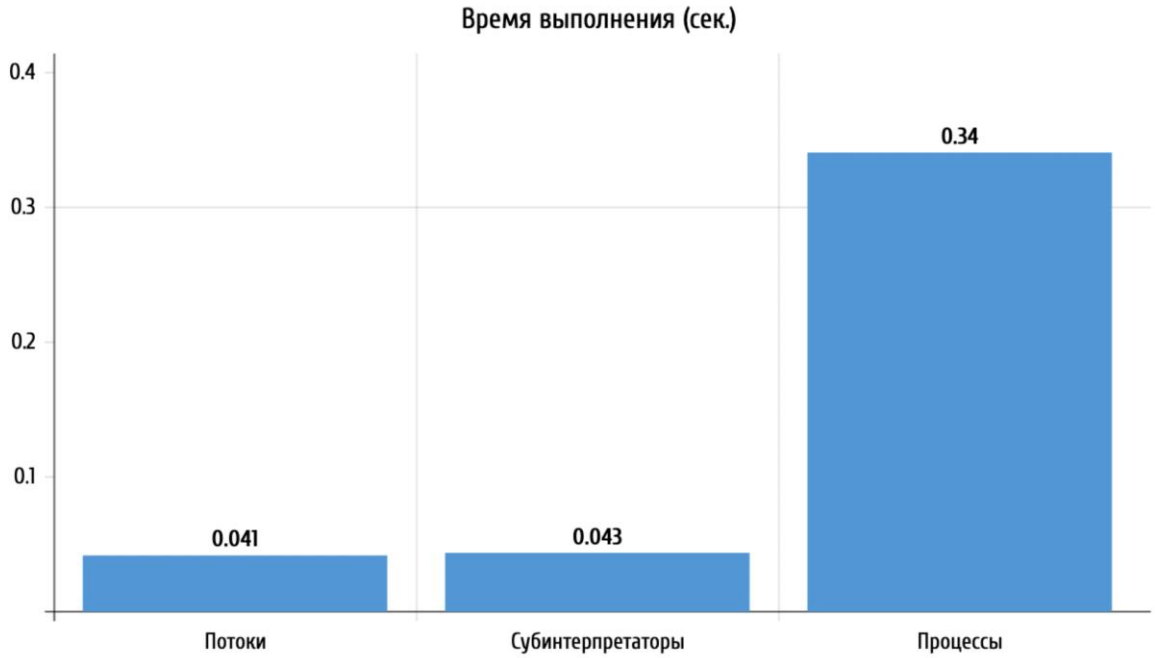

Рис. 6. График скорости вычисления числа π с точностью до 200 десятичных знаков в разных моделях параллельных вычислений

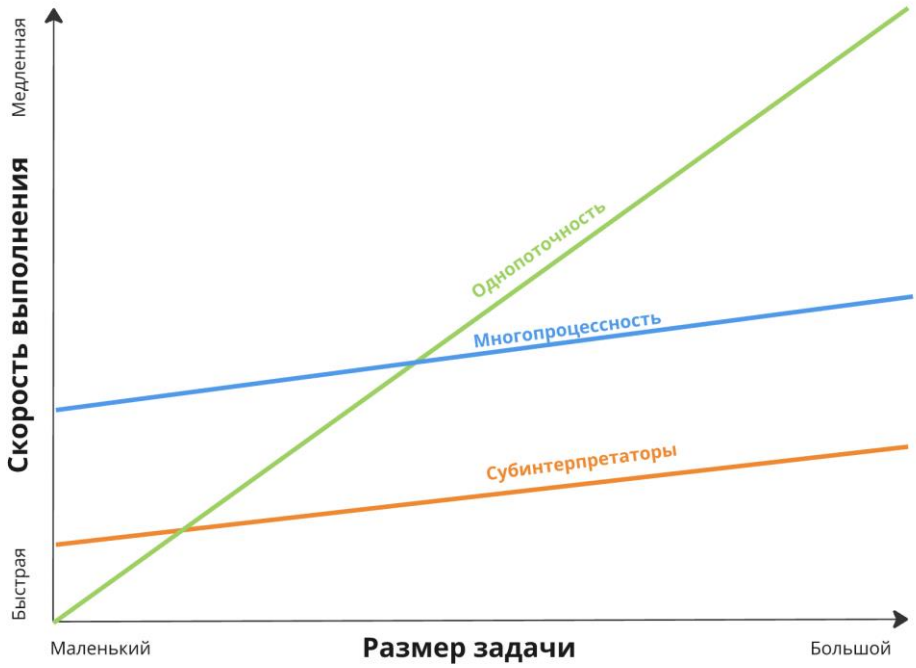

Рис. 7. График скорости выполнения от размера вычисляемых задач

Чтобы наглядно продемонстрировать концепцию границы, при которой параллелизм начинает демонстрировать более высокую эффективность, на рис. 7 представлены размер рабочей нагрузки и скорость увеличения времени выполнения. Эта граница является подвижным значением, зависящим от множества различных факторов.

Важным аспектом многопроцессорности является то, что она чаще всего применяется в моделях, где процессы выполняются длительное время и обрабатывают множество задач, а не создаются и уничтожаются для выполнения одной конкретной задачи. Прекрасным примером этого является популярный веб-сервер Gunicorn [13], который создает «работников», использующих многопроцессность, и эти рабочие процессы остаются активными на протяжении всего времени существования основного процесса. В таких условиях время запуска нового процесса или субинтерпретатора теряет свою значимость, особенно когда веб-воркер может функционировать продолжительное время. Оптимальным способом использования этих параллельных исполнителей для небольших задач является оставление их активными и использование основного процесса для координации и распределения нагрузки.

Многопроцессорность и субинтерпретаторы имеют собственное состояние импорта, что кардинально отличается от потоков и сопрограмм. При использовании асинхронного или многопоточного подходов не нужно беспокоиться об импорте необходимых модулей. Например, можно импортировать что-то в свой модуль и ссылаться на него изнутри функции потока.

Запуск интерпретатора занимает значительное время на выполнение модуля импорта site, который представлен файлом в установке Python. У каждого интерпретатора есть свои кэши и встроенные функции, которые делают его мини-процессом Python. Запуск потока или сопрограммы происходит очень быстро, поскольку они разделяют состояние с основным интерпретатором, что позволяет им избежать лишних вычислений. Однако при этом они подвержены блокировке и не могут работать параллельно.

Кроме того, важным моментом при использовании модели параллельного выполнения, такой как многопроцессорная обработка или субинтерпретаторы, является то, как происходит обмен данными. Независимо от того, используются

ли субинтерпретаторы или многопроцессорная обработка, нельзя просто отправлять существующие объекты Python в рабочие процессы.

Многопроцессорность использует по умолчанию pickle, когда происходят запуск процесса или использование пула процессов, и можно использовать каналы, очереди и общую память в качестве механизмов для отправки данных между рабочими процессами и основным процессом. Эти механизмы основаны на сериализации.

Pickle [14] – это встроенная библиотека сериализации, позволяющая преобразовать большинство объектов Python в байтовые строки и обратно. Она очень гибкая и поддерживает сериализацию различных типов объектов и даже позволяет определять методы сериализации для конкретных объектов. Кроме того, pickle справляется с вложенными объектами и их свойствами. Однако эта гибкость приводит к снижению производительности и может стать узким местом, особенно в моделях, основанных на частом обмене сложными обработанными данными между работниками, так как процесс сериализации – это достаточно медленная операция.

Субинтерпретаторы тоже могут принимать сериализованные данные, но у них также есть второй механизм, называемый общими данными. Общие данные – это высокоскоростное общее пространство памяти, в которое интерпретаторы могут записывать данные и обмениваться ими с другими интерпретаторами. Оно поддерживает только неизменяемые типы:

- Строки;
- Байтовые строки;
- Целые числа и числа с плавающей запятой;
- Булево значение и None;
- Кортежи.

Чтобы поделиться данными с интерпретатором, необходимо установить их как данные инициализации, использовав shared-аргумент, и предоставить словарь с общими значениями (int, float, bool, bytes, str, None, tuple). Если интерпретатор запущен, то можно обмениваться данными, используя канал. Модуль каналов также является частью PEP554 [9] и доступен с помощью секретного импорта.

ПРИМЕНЕНИЕ СУБИНТЕРПРЕТАТОРОВ В ВЕБ-РАЗРАБОТКЕ КАК ЗАМЕНА КАНОНИЧЕСКИМ МЕТОДАМ

Веб-серверы Python, расположенные перед фреймворками, такими как Django, Flask [15] или FastAPI, используют интерфейс WSGI или ASGI для синхронной и асинхронной обработки запросов соответственно [16]. Веб-серверы прослушивают HTTP-порт, на который приходят запросы, а затем распределяют запросы между набором предварительно проинициализированных процессов. В случае инициализации одного рабочего процесса пользователь, сделав HTTP-запрос, блокирует выполнение других запросов, пока не закончит обработку первого запроса.

Это означает, что при использовании многопроцессорной обработки существуют один GIL для каждого ядра процессора и пул потоков для конкурентной обработки входящих запросов. У такого подхода есть некоторые недостатки. Потоки не параллельны, поэтому, если существует два потока внутри одного рабочего процесса, которые очень заняты, Python не сможет «переместить» или запланировать решение этой задачи на другом ядре процессора.

В ходе настоящего исследования стояла цель разработать подход с субинтерпретаторами в качестве механизма работы воркеров, чтобы заменить подход с многопроцессорностью. Преимуществом нового подхода должно стать использование высокопроизводительного API каналов общей памяти для взаимодействия между работниками, вследствие чего воркеры будут меньше весить, занимая меньше памяти на хосте и оставляя больше памяти и ресурсов для обработки запросов.

Второй целью было скомпилировать основную ветку CPython 3.13 с реализацией потоков без GIL PEP703 [10], чтобы изучить возможность запуска потоков без GIL внутри этой модели.

Для достижения поставленных целей вручную был скомпилирован CPython 3.13a04 из исходного кода, а также разработан веб-воркер Hypercorn, работающий с помощью концепции субинтерпретаторов. В ходе работы также были решены следующие проблемы:

- Создание субинтерпретатора;

- Создание сигнального канала для передачи команд о запросах на выключение;
- Реализация подкласса threading.Thread и собственного метода .stop(), который отправляет сигнал субинтерпретатору;
- Запуск каждого дополнительного интерпретатора в потоке;
- Преобразование списка сокетов в кортеж кортежей.

Для сравнения производительности потоков, процессов и субинтерпретаторов было реализовано веб-приложение, написанное на Flask и вычисляющее факториал от 30. В качестве фреймворка для реализации веб-приложения был выбран Flask, так как, например, запустить приложение, написанное на Django, не получилось из-за тесной интеграции этого фреймворка с библиотекой для работы со временем datetime, которое использует расширение, написанное на C и использующее общее глобальное состояние.

Нагрузочное тестирование производилось с помощью инструмента Locust [17] с настройкой конкурентности, равной 100 пользователям, и скоростью появления 5 пользователей в секунду.

Результаты тестирования количества обрабатываемых запросов в секунду, потребление оперативной памяти и нагрузки процессора при использовании потоков, процессов и субинтерпретаторов представлены на рис. 8–13, для сравнения производительности были использованы следующие параметры запуска:

- Потоки – были запущены в 4 потока и 1 процесс;
- Процессы – был произведен запуск 4 рабочих процесса в однопоточном режиме;
- Субинтерпретаторы – запуск производился при использовании 4 субинтерпретаторов.

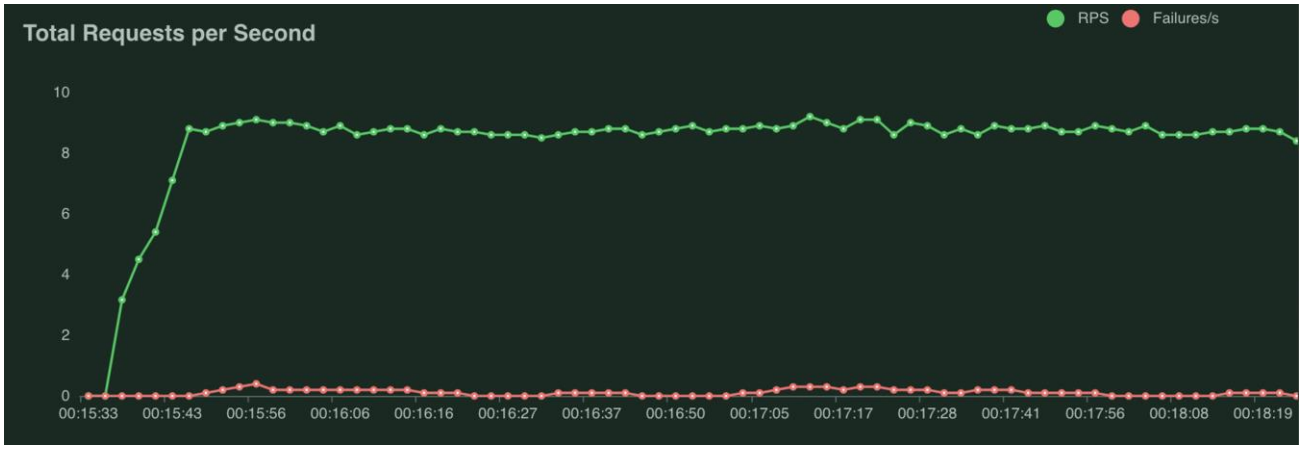

Рис. 8. График обрабатываемых запросов в секунду при многопоточной модели

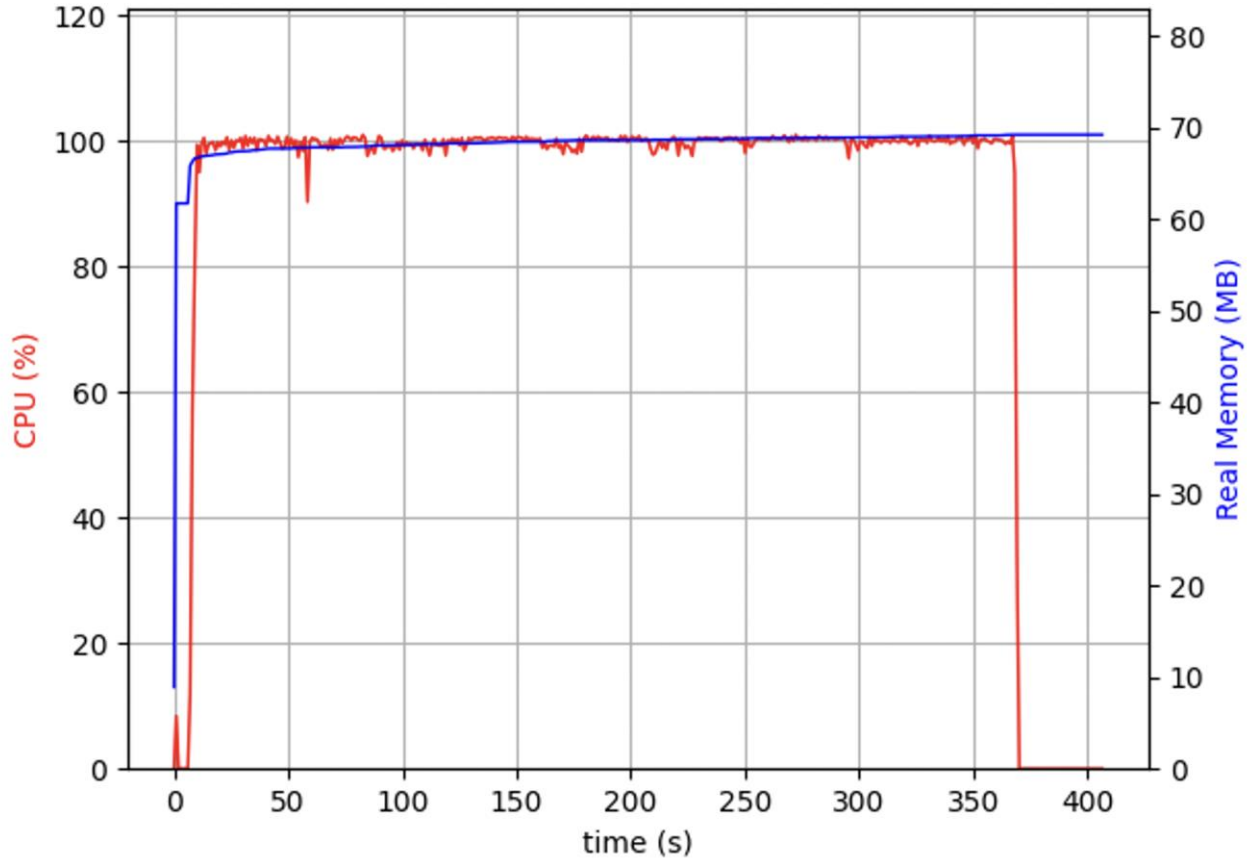

Рис. 9. График потребления памяти и ресурсов процессора при многопоточной модели

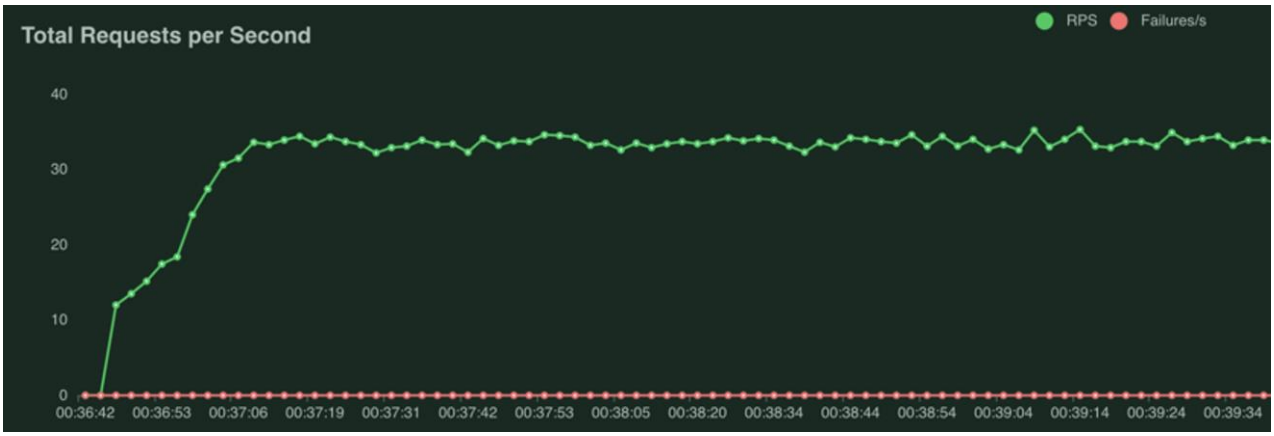

Рис. 10. График обрабатываемых запросов в секунду при многопроцессорной модели

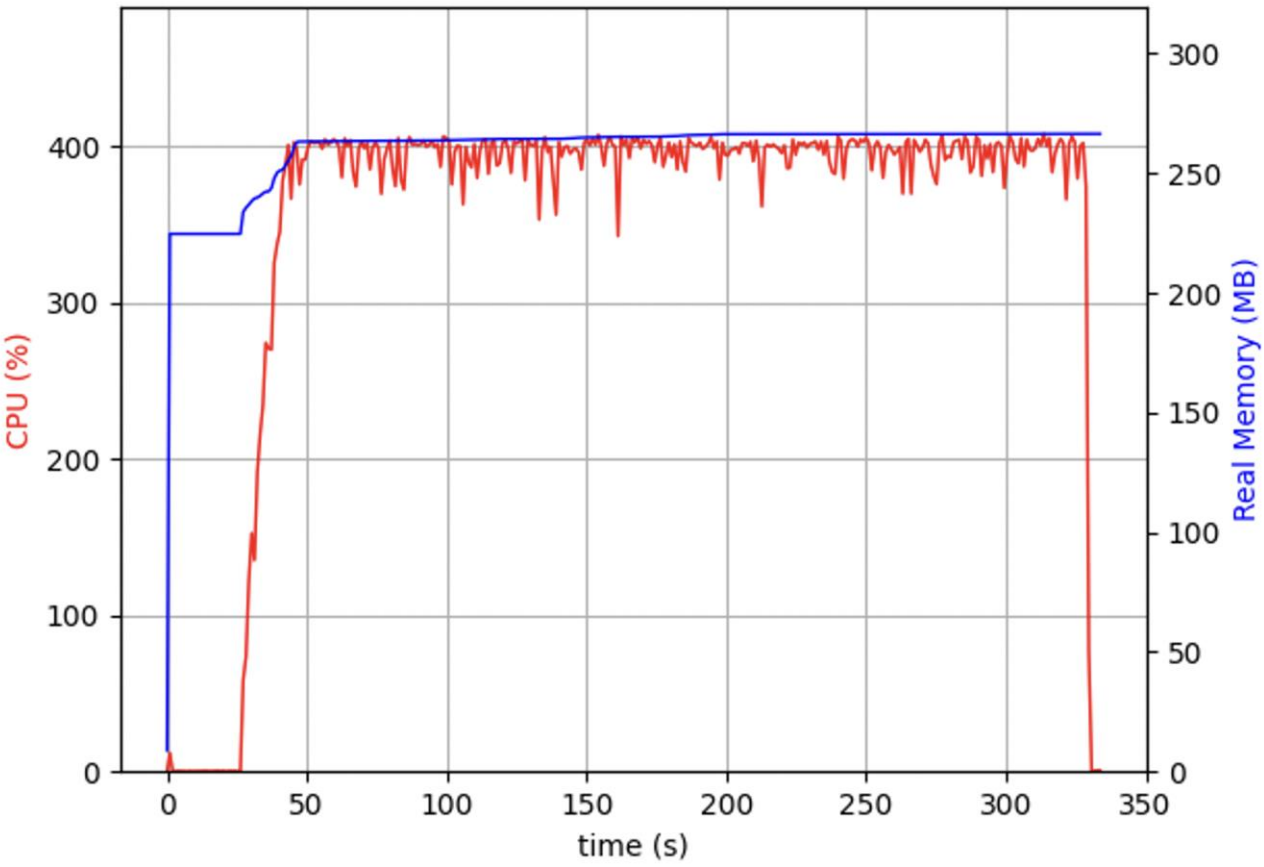

Рис. 11. График потребления памяти и ресурсов процессора при многопроцессорной модели

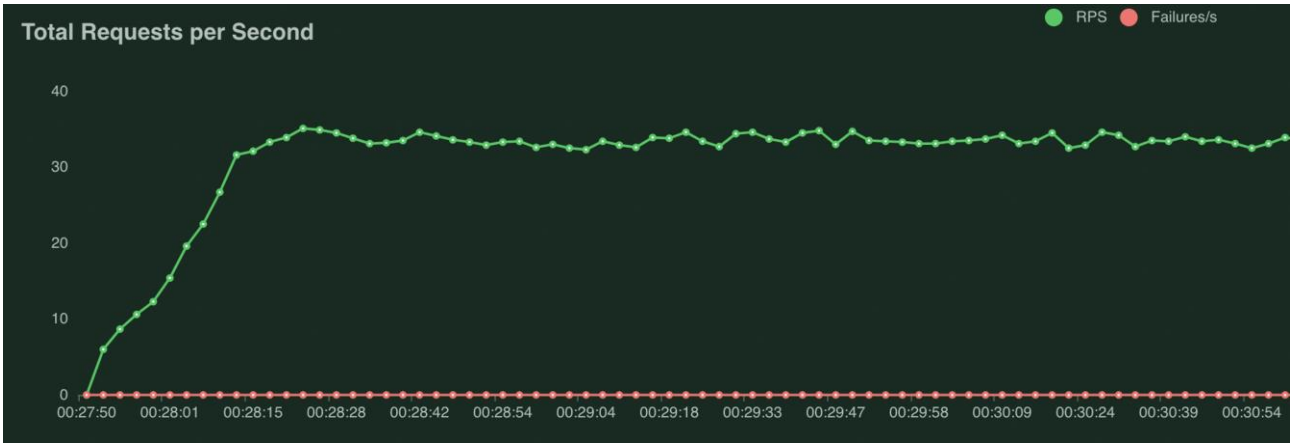

Рис. 12. График обрабатываемых запросов в секунду при использовании субинтерпретаторов

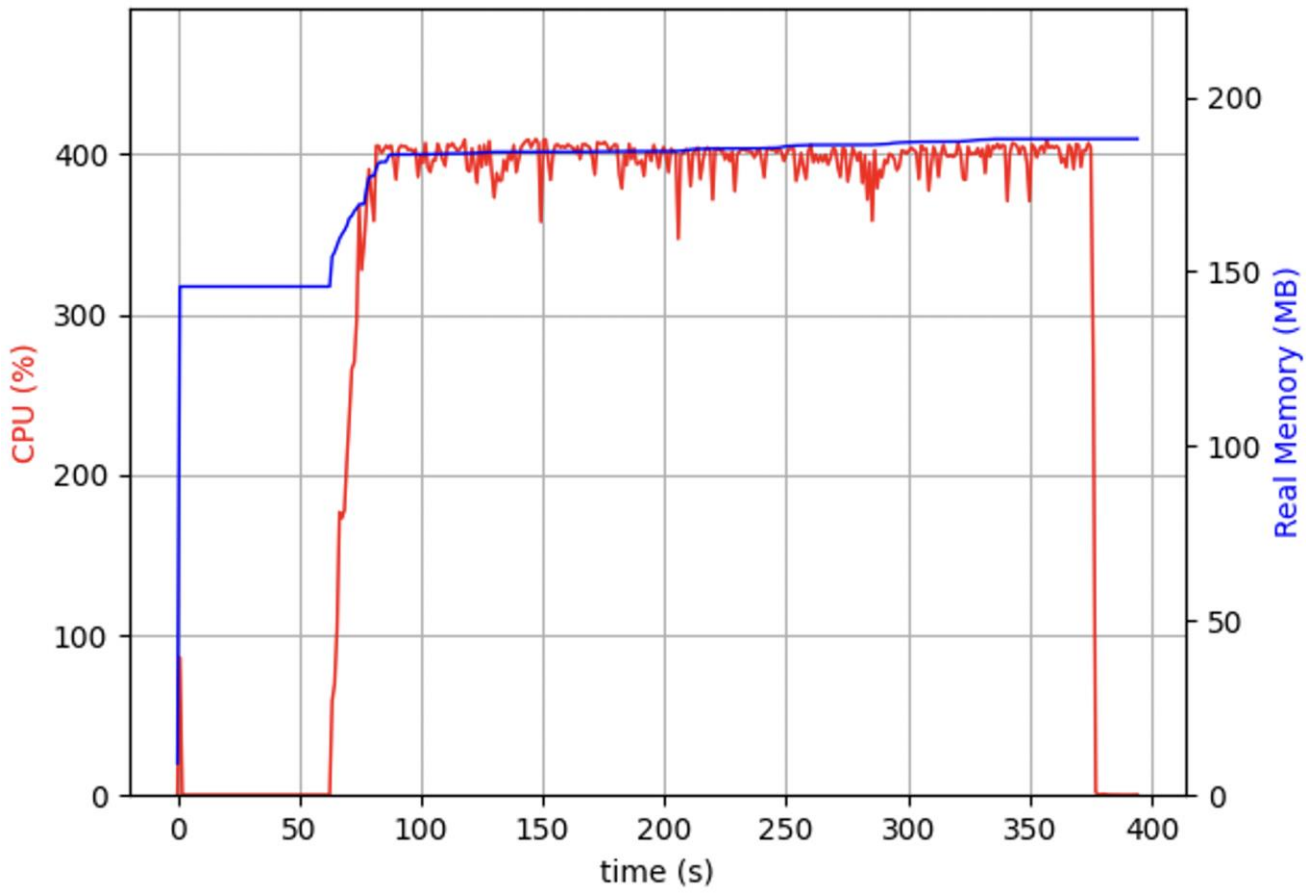

Рис. 13. График потребления памяти и ресурсов процессора при использовании субинтерпретаторов

Из полученных результатов видно, что при использовании многопоточности производительность приложения достаточно низкая и не превышает 10 запросов в секунду. При использовании субинтерпретаторов достигается производительность, не превышающая 40 запросов в секунду. Однако такой подход является самым экономным по памяти и занимает приблизительно 70 МБ оперативной памяти. Субинтерпретаторы, в свою очередь, являются средним вариантом между многопоточностью и процессами, сохраняя неменьшую производительность и потребляя 180 МБ оперативной памяти, в то время как многопроцессорность занимает 270 МБ оперативной памяти. Это доказывает преимущества и дальнейший потенциал использования предложенного подхода для веб-приложений.

ЗАКЛЮЧЕНИЕ

Рассмотрена проблема ограничений, накладываемых GIL на производительность многопоточных приложений в Python. Проанализирована концепция субинтерпретаторов как эффективного решения, способствующего повышению параллельной обработки задач. Результаты проведённых экспериментов показали, что субинтерпретаторы могут существенно улучшить производительность вычислений, особенно в сценариях с высокими нагрузками.

Кроме того, было проведено сравнение субинтерпретаторов с подходами многопроцессности и многопоточности. Результаты показали, что, в отличие от многопоточности, субинтерпретаторы позволяют избежать проблем, вызванных GIL, обеспечивая лучшую масштабируемость при выполнении CPU-bound задач. В то же время многопроцессный подход предоставляет преимущества в использовании многоядерных систем, но требует большего объёма ресурсов из-за необходимости создавать отдельные процессы и обмениваться данными между ними.

Сравнительный анализ показал, что субинтерпретаторы, будучи более лёгковесными, обеспечивают баланс между производительностью и эффективностью использования ресурсов. Особенно это заметно в ситуациях, где число задач превышает количество доступных ядер процессора. Таким образом, использование субинтерпретаторов обеспечивает синергетическую комбинацию лучших характеристик многопоточности и многопроцессности.

Выявлен значительный потенциал субинтерпретаторов для применения в различных областях, включая веб-разработку и обработку больших объёмов данных, что имеет важное значение в контексте современных вычислительных задач. Важно отметить, что, несмотря на частичную реализацию данной концепции, субинтерпретаторы представляют собой инновационный шаг, и существует необходимость в дальнейших исследованиях и разработках, направленных на полноценную интеграцию субинтерпретаторов в экосистему Python.

Благодарности

Автор выражает признательность преподавателям кафедры инструментального и прикладного программного обеспечения РТУ МИРЭА за ценные консультации по аспектам рассматриваемой предметной области.

СПИСОК ЛИТЕРАТУРЫ

1. Официальная документация Python. URL: https://www.python.org/
2. *Фаулер М.* Asyncio и конкурентное программирование на Python. М.: ДМК Пресс, 2023. 398 с.
3. *Beazley D.* Understanding the python GIL // PyCON Python Conference. Atlanta, Georgia, 2010.
4. PEP 684 – A Per-Interpreter GIL. URL: https://peps.python.org/pep-0684/
5. *Shaw A.* CPython Internals: Your Guide to the Python 3 Interpreter. 2021. 394 p.
6. *Brownlee J.* Python Multiprocessing Jump-Start: Develop Parallel Programs, Side-Step the GIL, and Use All CPU Cores (Python Concurrency Jump-Start Series). 2022. 144 p.
7. Официальный сайт PyParallel. URL: https://pyparallel.org/
8. *Roghult A.* Benchmarking Python Interpreters: Measuring Performance of CPython, Cpython, Jython and PyPy: Degree project in computer science and engineering. Sweden, 2016.
9. PEP 554 – Multiple Interpreters in the Stdlib.
URL: https://peps.python.org/pep-0554/
10. PEP 703 – Making the Global Interpreter Lock Optional in CPython.
URL: https://peps.python.org/pep-0703/

11. PEP 683 – Immortal Objects, Using a Fixed Refcount.
URL: https://peps.python.org/pep-0683/

12. *Савостин П.А., Ефремова Н.Э.* Практическое применение асинхронного программирования на языке Python при помощи пакета asyncio // Программные системы и вычислительные методы. 2018. №2. С. 11–16.

13. Gunicorn - Python WSGI HTTP Server for UNIX. URL: https://gunicorn.org/

14. Pickle — Python object serialization.
URL: https://docs.python.org/3/library/pickle.html

15. *Grinberg M.* Flask Web Development: Developing Web Applications with Python. O'Reilly Media, 2014. 258 p.

16. *Gardner J.* The Web Server Gateway Interface (WSGI) / In: The Definitive Guide to Pylons. Apress, 2009. 513 p.

17. Locust – A modern load testing framework. URL: https://locust.io/

PROSPECTS FOR IMPROVING THE PERFORMANCE OF PARALLEL COMPUTING USING PYTHON SUBINTERPRETER TECHNOLOGY

R. D. Sinitsyn[0009-0003-6750-5222]

MIREA – Russian Technological University

sinicinr2003@mail.ru

Abstract

The problem of the impact of global interpreter locking on the performance of multithreaded applications in Python is considered. The concept of subinterpreters is described as one of the solutions that circumvent the limitations of the GIL and ensure efficient parallel code execution. A comparative analysis of subinterpreters with traditional methods of parallel computing, such as the use of processes and threads, is carried out. The experimental results have shown that subinterpreters significantly increase performance in conditions of high computing loads. In addition, the possibilities of using subinterpreters in web development are explored. The advantages of using

this approach for query processing and resource management in modern web applications are indicated, which can significantly improve their scalability and responsiveness. The novelty of the research lies in the in-depth analysis of subinterpreters in the context of specific use cases, which had not previously received sufficient coverage in the scientific literature. The results of the work emphasize the need for further study of subinterpreters as an alternative approach in Python and the interest of developers and researchers in the field of high-performance computing in this.

Keywords*: Python, CPython, PEP, GIL, subinterpreter, multithreading, multiprocessing, asynchrony, interpreter, parallel computing.*

REFERENCES

1. Official Python Documentation. URL: https://www.python.org/
2. *Fowler M.* Asyncio i konkurentnoe programmirovanie na Python. M.: DMK Press, 2023. 398 p.
3. *Beazley D.* Understanding the python GIL // PyCON Python Conference. Atlanta, Georgia, 2010.
4. PEP 684 – A Per-Interpreter GIL. URL: https://peps.python.org/pep-0684/
5. *Shaw A.* CPython Internals: Your Guide to the Python 3 Interpreter. 2021. 394 p.
6. *Brownlee J.* Python Multiprocessing Jump-Start: Develop Parallel Programs, Side-Step the GIL, and Use All CPU Cores (Python Concurrency Jump-Start Series). 2022. 144 p.
7. PyParallel's official website. URL: https://pyparallel.org/
8. *Roghult A.* Benchmarking Python Interpreters: Measuring Performance of CPython, Cpython, Jython and PyPy: Degree project in computer science and engineering. Sweden, 2016.
9. PEP 554 – Multiple Interpreters in the Stdlib.
URL: https://peps.python.org/pep-0554/
10. PEP 703 – Making the Global Interpreter Lock Optional in CPython.
URL: https://peps.python.org/pep-0703/
11. PEP 683 – Immortal Objects, Using a Fixed Refcount.
URL: https://peps.python.org/pep-0683/

12. *Savostin P.A., Efremova N.E.* Prakticheskoe primenenie asinkhronnogo programmirovaniya na yazyke Python pri pomoshchi paketa asyncio // Programmnye sistemy i vychislitel'nye metody. 2018. №2. S. 11–16.

13. Gunicorn - Python WSGI HTTP Server for UNIX. URL: https://gunicorn.org/

14. Pickle — Python object serialization.
URL: https://docs.python.org/3/library/pickle.html

15. *Grinberg M.* Flask Web Development: Developing Web Applications with Python. O'Reilly Media, 2014. 258 p.

16. Gardner J. The Web Server Gateway Interface (WSGI) / In: The Definitive Guide to Pylons. Apress, 2009. 513 p.

17. Locust – A modern load testing framework. URL: https://locust.io/

СВЕДЕНИЯ ОБ АВТОРЕ

СИНИЦЫН Роман Дмитриевич – Студент Российского технологического университета МИРЭА по направлению «Программная инженерия».

Roman Dmitrievich SINITSYN – Student of the Russian Technological University MIREA with a degree in Software Engineering.

email: sinicinr2003@mail.ru

ORCID: 0009-0003-6750-5222

Материал поступил в редакцию 16 марта 2025 года