Использование многопоточности в сценариях, написанных на языке Python, в системах автоматизированного проектирования трёхмерной графики

Копылов М.С., **Бирюков Е.Д.**, ИПМ им. М.В. Келдыша РАН
kopylov@gin.keldysh.ru, birukov@gin.keldysh.ru

Аннотация

В работе рассматривается такой базовый функционал языка программирования Python как распределение вычислительной нагрузки на несколько процессоров или процессорных ядер. Описываются способы использования модулей threading и multiprocessing. Демонстрируются варианты использования многопоточных сценариев в комплексе оптического моделирования.

1 Введение

Многопоточность – свойство кода программы выполняться параллельно (одновременно) на нескольких процессорах (ядрах процессора) или псевдопараллельно на одном процессоре (ядре процессора). Особенно популярным этот метод распределения вычислительной нагрузки стал с наступлением эры многоядерных процессоров. В целом существует два основных подхода в распределении такой нагрузки: использование процессов и потоков.

Для применения многопоточности существует несколько причин. Например, приложение предпринимает попытку обращения к какому-то сетевому ресурсу, базе данных или устройству которое может занять определенное время. Вряд ли захочется, чтобы пользовательский интерфейс из-за этого блокировался, и пользователю пришлось бы просто дожидаться момента, когда от сервера или устройства вернётся ответ. Благодаря многопоточности, подобного рода задачи могут решаться гораздо эффективнее. К примеру, для всех видов активности, требующих ожидания, может быть запущен новый поток или процесс, позволяющий выполнить в это же время другие задачи. Также использование многопоточности может очень сильно уменьшить общее время выполнения насыщенных в плане обработки задач. В этом случае многочисленные потоки одного и того же процесса будут выполняться одновременно.

Язык программирования Python широко применяется для написания сценариев различного назначения благодаря элегантному дизайну, дисциплинирующему синтаксису, расширяемости, доступности на различных платформах (Windows, Linux, 32/64 бит). Именно благодаря этим качествам этот язык был интегрирован в наш комплекс оптического моделирования [Дерябин, Жданов, Соколов, 2017]. К тому же данный язык поддерживает оба вышеперечисленных подхода, предоставляя пользователю возможность написания программ, использующих как многопоточность, так и многопроцессность.

На текущий момент в рамках работы с данным комплексом оптического моделирования, написано достаточно большое количество различных сценариев. Часть из них используется непосредственно для задач моделирования, другая часть задействована в целях тестирования и самодиагностики компонентов комплекса. В целом наблюдается тенденция к постепенному усложнению этих сценариев. Это связано с одной стороны с интеграцией дополнительных пакетов расширения самого разного назначения, таких как numpy, scipy, matplotlib, imageio; с другой стороны с эволюционным расширением возможностей комплекса в рамках решаемых с его помощью задач.

В настоящее время всё чаще возникает необходимость использования многопоточности непосредственно в самих сценариях. Например, многие задачи пакетной обработки изображений не могут задействовать вычислительные ресурсы компьютера/группы компьютеров (кластера) на полную мощность, что приводит к неоптимальному использованию таких ресурсов. Это может быть вызвано разными причинами, такими как собственные ограничения алгоритмов обработки графических данных в многопроцессорных/многоядерных конфигурациях и конфигурациях с неоднородным доступом к памяти (NUMA). Бывают и обратные примеры, где выполнение пакетной обработки данных мо-

жет быть крайне не эффективным, если в нём вызываются некоторые специфические методы ядра комплекса оптического моделирования, которые однопоточны по своей природе.

В данной работе делается попытка разобраться с особенностями обоих подходов к многопоточности в Python, понять отличия между ними, выделить критерии, по которым должно отдаваться предпочтение тому или иному подходу в контексте использования этого языка в системе оптического моделирования и реалистичной компьютерной графики, разрабатываемой в ИПМ им. М. В. Келдыша РАН.

2 Многопоточность в Python

Как уже было сказано ранее, язык Python поддерживает оба базовых вида распределения вычислительной нагрузки - многопоточность и многопроцесность. Надо отметить, что потоки в Python управляются непосредственно операционной системой, т.е. их планирование и переключение не зависит от интерпретатора.

Являясь интерпретируемым языком, Python (а точнее его основная реализация CPython) содержит в себе специальный механизм - Global Interpreter Lock (GIL). Данный механизм в целом препятствует параллельному выполнению нескольких потоков, если они исполняются на многопроцессорной/многоядерной системе. Благодаря этому, только один поток в каждый момент времени может иметь доступ к ресурсам интерпретатора, другими словами, ресурсы принадлежат только этому потоку. Это сделано для того, чтобы между потоками не было конфликтов при доступе к отдельным переменным.

Подобная реализация потоков значительно упрощает работу с ними и даёт вполне достаточную потокобезопасность. Однако, большим минусом такой реализации потоков в Python может быть очень незначительный эффект от распараллеливания программы, а в некоторых ситуациях многопоточный код может выполняться по времени даже больше чем аналогичный однопоточный [Singh, Navtej, Lisa-Marie Browne, and Ray Butler, 2013].

Самым простым способом реализовать многопоточное выполнение в Python является использование модуля threading. Данный модуль содержит все, что нужно для многопоточного программирования - различные виды блокировок, семафоров, механизм событий. Пример простейшей программы, использующей данный модуль, показан на Рис. 1.

В данной программе создаются два потока `t1` и `t2`, которые одновременно печатают на экране текст. Анализируя данный пример, можно заметить, что код легко читаемый, последовательный и не нуждается в подробных комментариях.

Видно, что в данном примере используется объект типа Блокировка (Lock). Это необходимо для того, чтобы регулировать доступ к разделяемым ресурсам программы. В нашем случае таким разделяемым ресурсом является экран, на котором печатается текст оператором `print`.

Блокировка (Lock) – наиболее часто используемый объект синхронизации, исполь-

```
from threading import Thread, Lock

def f(lock):
    lock.acquire()
    print('hello world')
    lock.release()

if __name__ == '__main__':
    lock = Lock()
    # init threads
    t1 = Thread(target = f, args = (lock, ))
    t2 = Thread(target = f, args = (lock, ))
    # start threads
    t1.start()
    t2.start()
    # join threads to the main thread
    t1.join()
    t2.join()
```

Рис. 1. Пример использования модуля threading

зуемый при написании многопоточных приложений. Её особенностью является то, что в каждый момент времени она может принадлежать не более чем одному потоку. Захват блокировки осуществляется с помощью вызова метода `acquire`, а её последующее освобождение с помощью метода `release`.

Среди других объектов синхронизации в языке Python можно выделить блокировки с повторным входом (RLock), семафоры (Semaphore) и события (Event). Каждый из этих объектов имеет своё применение, например семафор удобно использовать в тех случаях, когда необходимо лимитировать доступ к какому либо ограниченному ресурсу.

Как уже было сказано, язык Python содержит в себе Global Interpreter Lock, благодаря которому синхронизацию доступа к разделяемым ресурсам, данным или переменным можно получить, вообще не прибегая к использованию каких либо явных блокировок [Masini, Bientinesi, 2010]. При этом данное исключение распространяется только на так называемые атомарные операции. Интерпретатор выполняет их без возможности прерывания со стороны других потоков. К таким операциям можно отнести:

- чтение или изменение одного атрибута объекта
- чтение или изменение одной глобальной переменной
- выборка элемента из списка
- модификация списка «на месте» т.е. с помощью метода `append`
- выборка элемента из словаря
- модификация словаря «на месте» т.е. добавление элемента, или вызов метода `clear`

Помимо модуля threading, Python включает в себя модуль multiprocessing. По функциональности этот модуль очень сильно напоминает вышеописанный модуль threading, однако он лишён некоторых его недостатков. Главное отличие заключается в том, что программы, использующие данный модуль, могут выполняться в параллельных процессах. В этом случае проблема, связанная с GIL, не играет негативной роли, так как каждый процесс использует свой собственный интерпретатор и GIL, но он не помешает им работать параллельно [Marowka, 2018].

Надо отметить, что синхронизации процессов и обмен данными между процессами является более сложной процедурой, чем обмен данными между потоками, выполняющимися в рамках одного процесса. Для этого в приложениях, выполняющих вычисления в нескольких процессах одновременно, принято использовать такие инструменты как очереди (Queue) и каналы (Pipe). Однако аналоги более простых объектов синхронизации, как блокировки, семафоры, события также доступны.

На Рис. 2. показан пример программы, использующей модуль multiprocessing. Данная программа запускает одновременно десять процессов, каждый из которых печатает свой порядковый номер на экране. Можно заметить, что код программы мало чем отличается от примера, использующего модуль threading.

Надо отметить, что в модуле multiprocessing имеются механизмы для работы с общей памятью. Для этого служат специальные классы переменной и массива, которые можно обобщать между несколькими процессами. Так же к особенностям данного модуля можно отнести его существенную платформозависимость.

3 Использование многопоточных сценариев на практике

Как уже было сказано, комплекс оптиче-

```
from multiprocessing import Process, Lock

def f(lock, i):
    lock.acquire()
    print(i)
    lock.release()

if __name__ == '__main__':
    lock = Lock()
    for num in range(10):
        Process(target = f, args = (lock, num)).start()
```

Рис. 2. Пример использования модуля multiprocessing

ского моделирования, разрабатываемый в ИПМ им. М. В. Келдыша РАН, позволяет запускать различные пользовательские сценарии благодаря встроенному языку Python. Подмножество внутренних объектов и классов комплекса доступно из этих сценариев через Python API [Дерябин, Жданов, Соколов, 2017]. Подобный функционал даёт возможность автоматизировать действия пользователя по работе с различными объектами комплекса, такими как сцена, модули рендеринга, различные симуляторы и.т.д.

Наличие подобной автоматизации является очень полезным при решении разносторонних задач, оперирующих большими объёмами данных, поэтому вопрос быстродействия сценариев в данных условиях стоит очень остро.

В рамках работы с комплексом были выявлены некоторые задачи, выполняющиеся не оптимально с точки зрения использования доступных вычислительных ресурсов, характеризующиеся не полной загрузкой всех доступных процессоров или процессорных ядер.

Одной из таких задач является применение медианного или усредняющего фильтра к изображению. В силу особенностей алгоритма подобных фильтров они по большей части являются однопоточными [Бирюков, 2014]. Данное обстоятельство не является особо критичным при работе с комплексом с помощью графического интерфейса, ведь в каждый определённый момент времени пользователь работает только с одним изображением, и общие потери времени на операцию наложения фильтра не столь заметны.

Однако при задействовании интегрированного языка сценариев Python подобные однопоточные и плохо поддающиеся распараллеливанию операции могут стать узким местом, особенно при использовании пакетной обработки данных. Простейший пример такого сценария, накладывающий фильтр на множество отдельных изображений, показан на Рис. 3.

В данном примере происходит циклическая загрузка изображений из файлов, хранящихся на диске, в ядро комплекса, и последующий вызов метода, накладывающего фильтр на это

```
def ApplyFilter(name):
    pp = PostProcessor(name)
    res = pp.ApplyAverageFilter()
    res.SaveToFile(…)

files = ["file1", "file2", …, "file99"]
for f in files:
    ApplyFilter(f)
```

Рис. 3. Однопоточный сценарий, применяющий фильтр к группе файлов

изображение. Результат работы фильтра затем сохраняется в новый файл.

Перепишем данный сценарий с поддержкой многопоточности, воспользовавшись модулем multiprocessing. Код подобного улучшенного сценария показан на Рис. 4. В данном примере операторы загрузки файлов, наложения фильтра и последующего сохранения результата вынесены в отдельный процесс.

4 Вычислительный эксперимент

Для замеров быстродействия однопоточного и многопоточного сценария, рассмотренных в предыдущей главе, был осуществлён вычислительный эксперимент на ПК со следующими характеристиками: Процессор: Intel Core i7-3770 3,4 ГГц; RAM: 32 Гб; ОС: Windows 8.1 x64. В качестве исходных данных использовались 100 графических файлов с разрешением 800 на 600 пикселей. Число одновременно порождаемых потоков было ограничено числом ядер процессора (4 ядра), технология Hyper-Threading была отключена. Сравнение замеров производительности приведено в Табл. 1.

Табл. 1. Сравнение времени выполнения сценариев

Сценарий	Время выполнения с.
Однопоточный	1480
многопоточный	430

```
def ApplyFilter(name):
    pp = PostProcessor(name)
    res = pp.ApplyAverageFilter()
    res.SaveToFile(…)

if __name__ == '__main__':
    from multiprocessing import Process
    files = ["file1", "file2", …, "file99"]
    for f in files:
        p = Process(target = ApplyFilter, args = f)
        p.start()
```

Рис. 4. Сценарий, реализующий многопоточность
при, применении фильтра к группе файлов

Эксперимент показал, что использование многопоточности в сценарии, осуществляющем пакетную обработку графических данных, позволило уменьшить общее время выполнения в 3.4 раза, что является очень хорошим результатом, учитывая ту легкость, с которой данное ускорение было достигнуто. При этом для такого ускорения оказалось достаточно лишь незначительно модифицировать первоначальный сценарий.

5 Заключение

При выполнении данной работы были сделаны следующие выводы:

- Язык Python обладает достаточно продвинутой поддержкой многопоточного программирования. При этом стоит отметить, что унификация интерфейсов модулей threading и multiprocessing делает их весьма удобными в использовании. В некоторых случаях в сценарии достаточно изменить лишь несколько строк кода, чтобы перейти от использования одного модуля к другому и наоборот. Более того, оба модуля поддерживают один и тот же набор примитивов синхронизации, что упрощает разработку программ.
- Модуль threading имеет некоторые ограничения при работе с потоками в плане эффективности, если эти потоки в основном работают с общими данными, такими как массивы, словари, списки.
- При использовании Python в комплексе оптического моделирования существует серьёзное ограничение на использование модуля threading. Данное ограничение связано с возможностью ситуации, при которой из двух потоков одновременно будет вызвана одна и та же функция ядра комплекса. Учитывая, что многие функции ядра комплекса не являются потокобезопасными, существует реальная угроза аварийного завершения при выполнении многопоточных сценариев, или возникновения ошибок в результатах вычислений. Если же вызовов ядра комплекса из сценария не предполагается, модуль threading может использоваться без ограничений.
- Модуль multiprocessing является более предпочтительным при использовании в сценариях, так как при использовании процессов можно добиться большего параллелизма, особенно при задачах пакетной обработки данных, при этом он не имеет каких либо ограничений.
- В качестве дальнейших исследований наиболее интересным представляется изучение возможностей модуля concurrent.futures, который является дальнейшим развитием многопоточного программирования в Python и предоставляет классы ThreadPoolExecutor и ProcessPoolExecutor, лишенные многих недостатков модулей, разобранных в этой работе.

Список литературы

Н.Б. Дерябин, Д.Д. Жданов, В.Г. Соколов. Внедрение языка сценариев в программные комплексы оптического моделирования // Программирование, 2017, № 1, с. 40-53.

Singh, Navtej, Lisa-Marie Browne, and Ray Butler. 2013. *"Parallel astronomical data processing with Python: Recipes for multicore machines."* Astronomy and Computing 2: 1-10.

Masini, S. and Bientinesi, P., 2010, *August. High-performance parallel computations using python as high-level language.* In European Conference on Parallel Processing (pp. 541-548). Springer, Berlin, Heidelberg.

Marowka, A., 2018. *On parallel software engineering education using python*. Education and Information Technologies, 23(1), pp.357-372.

Бирюков Е.Д. Использование медианного фильтра в системе обработки изображений реалистичной компьютерной графики // Новые информационные технологии в автоматизированных системах: материалы семнадцатого научно-практического семинара - М.: ИПМ им. М.В.Келдыша, 2014, с. 216-220.