Легкий старт

Параллельное программирование на Python

Код на Python для ПК и кластеров с использованием CUDA, PyTorch и Dask



Фабио Нелли

Фабио Нелли

Параллельное программирование на Python

Код на Python для ПК и кластеров
с использованием CUDA, PyTorch и Dask

Parallel and High Performance Programming with Python

Unlock Parallel and Concurrent Programming in Python using Multithreading, CUDA, Pytorch, and Dask

by **Fabio Nelli**



Параллельное программирование на Python

Код на Python для ПК и кластеров
с использованием CUDA, PyTorch и Dask

Фабио Нелли

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2026

УДК 004.438Python
ББК 32.973.2
Н49

Н49 **Фабио Нелли**

Параллельное программирование на Python. Код на Python для ПК и кластеров с использованием CUDA, PyTorch и Dask / пер. с англ. В. И. Бахура. – М.: ДМК Пресс, 2026. – 326 с.: ил.

ISBN 978-5-93700-451-2

Книга последовательно знакомит с методами параллельного программирования на Python – от базовых концепций до современных инструментов для распределенных и ускоренных вычислений. Практические примеры демонстрируют применение параллельных подходов в аналитике данных, машинном обучении и научном моделировании. Рассмотрены современные тенденции развития параллельных вычислений и их влияние на экосистему Python. Вы узнаете, какие ограничения и компромиссы сопровождают различные модели параллелизма, и научитесь выбирать архитектурные решения в зависимости от характера нагрузки.

Издание будет полезно как практикующим инженерам, так и студентам технических специальностей, изучающим параллельное и распределённое программирование. Желательны базовые навыки программирования на Python и понимание принципов работы операционных систем.

УДК 004.438Python
ББК 32.973.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-93-88590-73-0 (англ.)
ISBN 978-5-93700-451-2 (рус.)

Copyright © 2023, Orange Education Pvt Ltd, AVA™
© Оформление, перевод, издание, ДМК Пресс, 2026

Посвящается моей жене Памеле и моему сыну Валерио, которые поддерживали меня на протяжении всего процесса создания этой книги.

Лично я убежден, что информационные технологии имеют много общего с физикой. Обе дисциплины занимаются изучением мироздания на самых фундаментальных уровнях. Единственное отличие, безусловно, заключается в том, что в физике задача состоит в изучении мироздания, а в информационных технологиях – в его созидании.

ЛИНУС ТОРВАЛЬДС

Оглавление

Об авторе **9**
Технический рецензент **10**
Благодарности **11**
Предисловие **12**

Глава 1. Введение в параллельное программирование **14**

Параллельное программирование 15
Технологическая эволюция компьютеров и параллелизм 15
Процессоры, ядра, потоки и процессы 17
Многозадачное и параллельное программирование 20
Потоки и процессы в Python для моделей многозадачности и параллелизма 22
Распределение и использование памяти 30
Оценка эффективности параллельного программирования 34
Заключение 38

Глава 2. Параллельное программирование с потоками **40**

Потоки 40
Метод join() 42
Механизмы синхронизации 50
Заключение 77

Глава 3. Многопроцессорные вычисления и библиотека mpi4py **79**

Процессы и модуль multiprocessing 80
Каналы взаимодействия между процессами 89
Отображение функции через пул процессов 96
Параллельное отображение с chunksize 99
Класс ProcessPoolExecutor 100
Библиотека mpi4py 103
Параллелизм процессов 104
Основные области применения mpi4py 107
Реализация двухточечных коммуникаций 108
Коллективные коммуникации 109
Оптимизация коммуникаций при помощи топологий 116
Заключение 119

Глава 4. Асинхронное программирование с AsyncIO **120**

Асинхронное и синхронное программирование 120
Библиотека AsyncIO 123

Асинхронные итерации с async for и без async for 135
Очередь в асинхронной модели 138
Альтернативы библиотеке AsyncIO 140
Заключение 140

Глава 5. Реализация параллелизма с помощью распределенных систем 141

Распределенные системы и программирование 142
Celery 144
Библиотека Dramatiq как альтернатива Celery 162
Библиотека SCOOP 169
Заключение 175

Глава 6. Программирование GPU с CUDA для максимальной производительности 176

Архитектура GPU 176
Программирование GPU на Python 178
Numba 179
PyOpenCL 195
Заключение 211

Глава 7. Эпоха параллельных вычислений 212

Высокопроизводительные вычисления (HPC) 212
Параллельные вычисления 213
Проекты и примеры параллельных вычислений 215
Развлечения – игры и фильмы 231
Заключение 238

Глава 8. Масштабирование приложений для обработки данных с помощью Dask 240

Аналитика данных, библиотека Pandas и параллельные вычисления 240
Библиотека Dask 242
Начало работы на одном компьютере 244
Коллекции Dask 247
Методы для коллекций 252
Вычисления и графы задач 253
Низкоуровневое взаимодействие с Dask Delayed 255
Начало работы на кластере машин 256
Приступая к программированию кластера 260
Заключение 266

Глава 9. Раскрытие потенциала ИИ с параллельными вычислениями 268

Искусственный интеллект (ИИ) 269
ИИ, машинное обучение и глубокое обучение 270
Контролируемое и неконтролируемое обучение 272
Искусственный интеллект и параллельные вычисления 273

Параллельное и распределенное машинное обучение ... 274
Машинное обучение с помощью Scikit-learn ... 275
Масштабирование Scikit-learn с Dask-ML ... 278
Параллельное и распределенное глубокое обучение ... 280
PyTorch и TensorFlow ... 281
Пример глубокого обучения с PyTorch ... 282
Пример глубокого обучения с использованием PyTorch и GPU ... 292
Масштабирование PyTorch с помощью Dask ... 294
Заключение ... 304

Глава 10. Приложения параллельных вычислений ... 306

Массивно-параллельный искусственный интеллект ... 307
Пограничные вычисления ... 308
Распределенная вычислительная инфраструктура с киберфизическими системами ... 310
Искусственный интеллект в кибербезопасности ... 312
Вступление в эпоху Web 5.0 ... 313
Экзафлопсные вычисления ... 315
Квантовые вычисления ... 315
Новые профессиональные возможности ... 316
Необходимые улучшения в параллельных вычислениях для Python ... 317
Будущее PyTorch и TensorFlow ... 318
Заключение ... 320

Предметный указатель ... 322

Об авторе

Фабио Нелли (Fabio Nelli) получил степень магистра по химии и степень бакалавра по информационным технологиям и автоматизации. В настоящее время он сотрудничает с несколькими научно-исследовательскими институтами и частными компаниями, где ведет образовательные курсы по анализу данных и технологиям визуализации данных. Помимо профессиональной деятельности, он активно участвует в развитии отрасли благодаря публикации статей в интернете, прежде всего на своем веб-сайте *meccanismocomplesso.org*, а также написанию тематических книг с подробным изложением материала.

Технический рецензент

Прашант Рагху (Prashanth Raghu) является страстным поклонником технологий, который уделяет особое внимание изучению влияния технологий с открытым исходным кодом на приложения для крупных систем. Он окончил Технологический институт PES, а затем Национальный университет Сингапура и в настоящее время работает архитектором программного обеспечения в компании Zeta Technologies. Помимо увлечения технологиями, он интересуется искусством, играет на карнатической флейте и занимается спортом.

Благодарности

Хотелось бы поблагодарить нескольких человек за постоянную и непрерывную поддержку, которую они оказывали мне в процессе написания этой книги. Прежде всего я хотел бы поблагодарить свою жену за то постоянное вдохновение, которое она давала мне для написания этой книги; без ее поддержки я бы не смог ее завершить.

Я благодарен команде Orange Education, которая сопровождала меня на всех этапах создания этой книги. Особая благодарность Шали Дираджу (Shali Deeraj) и Шубхе Мурти (Shubha Murthy) за их содействие.

Предисловие

Эта книга познакомит вас с различными методами, доступными в настоящее время для параллельного и многозадачного программирования на языке Python. Существует множество библиотек и методов, позволяющих использовать преимущества различных существующих архитектур для повышения эффективности и производительности вычислений (GPU, процессорные ядра, распределенные сетевые ресурсы и т. д.). В этой книге будут постепенно представлены соответствующие концепции с примерами кода.

Глава 1 начинается с введения в параллелизм с использованием Python, а затем в ней описывается принцип обработки нескольких процессов и потоков на уровне операционной системы. Цель этой главы – познакомить вас с концепцией **параллельного программирования**, рассказав обо всех основных аспектах, которые необходимо учитывать для полного понимания его возможностей и применения. После введения в эти общие концепции вы познакомитесь с особенностями Python в этой области, прежде всего с потоками, а также с GIL (Global Interpreter Lock) и связанными с ним проблемами. Помимо этого, будут рассмотрены модули стандартной библиотеки Python, такие как threading и multiprocessing.

Глава 2 посвящена параллельному программированию на Python с использованием потоков в качестве элементов параллелизма. В частности, в этой главе будет рассмотрен модуль threading, который упрощает реализацию многопоточности и предоставляет множество полезных инструментов для синхронизации потоков.

Глава 3 посвящена параллельному программированию на Python с использованием процессов. Существует два основных подхода к реализации процессов в параллельном программировании: модуль multiprocessing стандартной библиотеки и библиотека mpi4py, которая также обеспечивает поддержку протокола MPI для языка Python. Обе библиотеки будут подробно рассмотрены в серии примеров, демонстрирующих принцип их работы и основные характеристики.

Глава 4 охватывает аспекты асинхронного программирования на Python, в том числе библиотеку AsyncIO в качестве примера.

Глава 5 посвящена распределенным системам, поскольку они также относятся к области параллельного и многопоточного программирования, что может оказаться эффективным подходом. Библиотека Celery в Python является эталоном для реализации распределенных систем. На ряде примеров будет показано, как использовать эту библиотеку для выполнения нескольких параллельных операций, называемых задачами, и как они распараллеливаются и выполняются в системе на базе Celery. Кроме того, далее будут рассмотрены другие альтернативные решения, начиная с похожих, таких как Dramatiq, и заканчивая более простыми, но отличными от них с концептуальной точки зрения, такими как SCOOP.

Глава 6 охватывает тему программирования GPU, поскольку эти процессоры предназначены для чрезвычайно быстрой и эффективной обработки векторных данных для рендеринга изображений, 3D-движков и работы с полигональными примитивами. Язык Python предлагает хорошие решения с различными библиотеками, такими как Numba (CUDA) и PyOpenCL.

Глава 7 познакомит вас с приложениями для параллельных вычислений и продемонстрирует их применение во многих научных и технических дисциплинах.

Глава 8 посвящена параллельным вычислениям, в частности в области науки о данных. Многие библиотеки, широко используемые в науке о данных, такие как NumPy, Pandas и Scikit-learn, могут быть адаптированы для параллельных вычислений благодаря библиотеке Dask. Эта библиотека предоставляет объекты, такие как nparrays, dataframes и machine learning api, предназначенные для работы в параллельном режиме. Существует множество других научных библиотек, которые могут быть интегрированы с Dask.

Глава 9 освещает еще одну очень интересную область применения, которая частично перекликается с материалами главы 8 об аналитике данных. Машинное обучение и глубокое обучение являются подразделами области искусственного интеллекта и являются эффективным инструментом для исследования данных. В последнее время эти технологии получают широкое распространение благодаря использованию параллельных и распределенных вычислений.

Глава 10 посвящена развитию и тенденциям в области программирования, которые в полной мере соответствуют последним инновационным технологиям.

Загрузка пакетов кода и цветных изображений

Загрузка пакетов кода из книги доступна по адресу *https://github.com/OrangeAVA/Parallel-Programming-with-Python*.

Пакеты кода и изображения из книги также можно найти по адресу *https://rebrand.ly/3127a9*.

В случае обновления кода он также будет обновлен в существующем репозитории GitHub.

Исправления

Мы в <Orange Education Pvt Ltd> испытываем огромную гордость за свою работу и следуем лучшим практикам в целях обеспечения точности нашего контента, чтобы предоставить нашим пользователям возможность насладиться чтением. Наши читатели – это наше зеркало, и мы используем их отзывы для анализа и исправления возможных ошибок, которые могли возникнуть в процессе публикации. Мы стремимся поддерживать высокое качество и помогать читателям, которые столкнулись с трудностями из-за непредвиденных ошибок. Пожалуйста, пишите нам по адресу *errata@orangeava.com*.

Мы высоко ценим вашу поддержку, предложения и отзывы.

Глава 1

Введение в параллельное программирование

В первой главе книги представлена идея параллельного программирования с описанием всех основных концепций, которые необходимо знать для полного понимания его особенностей и возможностей. Сначала рассмотрены аппаратные компоненты нового поколения компьютеров, такие как процессоры и ядра, которые позволяют выполнять параллельные вычисления. Затем описаны элементы операционной системы, которые непосредственно обеспечивают параллелизм: процессы и потоки. Далее подробно рассмотрены модели параллельного программирования с введением таких фундаментальных концепций, как параллелизм, синхронность и асинхронность.

После ознакомления с этими общими концепциями мы рассмотрим специфику использования языка Python применительно к этой области, в частности потоки, а также остановимся на Global Interpreter Lock (GIL) и связанных с ним проблемах. Мы упомянем стандартные модули библиотеки Python, такие как threading и multiprocessing, которые будут более подробно рассмотрены в следующих главах. Наконец, мы завершим главу обзором методов оценки эффективности программного обеспечения с параллельной обработкой данных, таких как ускорение и масштабирование, а также обсудим проблемы, которые могут возникнуть при программировании параллельных процессов (состояние конкурентной гонки, тупиковая ситуация и т. д.).

К концу этой главы вы ознакомитесь со всеми основными концепциями и терминологией, на которых основано параллельное программирование. Вы сможете составить в уме общую схему, в которой будут представлены все ключевые элементы процесса параллельного выполнения и их роль в его реализации. После этого вы будете готовы приступить к практической части обучения программированию, которая будет рассмотрена в следующих главах.

Структура

В этой главе будут рассмотрены следующие темы:

- центральные процессоры и ядра;
- процессы и потоки;
- параллельное и многопоточное программирование;

- GIL и потоки в Python;
- ускорение и масштабирование.

Параллельное программирование

Если вы читаете эту книгу, то, безусловно, вам уже понятна необходимость повышения потенциала вашего кода, поскольку вы столкнулись с ограничениями традиционных моделей, которые по историческим причинам (ограничения старых компьютеров) базируются на последовательном принципе.

Появление новых аппаратных технологий дало нам возможность запускать на наших компьютерах несколько программ одновременно. В действительности даже самые простые компьютеры оснащены многоядерной системой, которая позволяет программам работать в параллельном режиме. Почему бы тогда не воспользоваться преимуществами этой архитектуры?

Слишком часто нам приходится писать программы на Python для выполнения серии операций. В научной сфере для выполнения трудоемких вычислений зачастую необходимо использовать множество алгоритмов. Но в конце работы, запустив программу на своем компьютере, вы с разочарованием обнаруживаете, что она работает не так быстро, как вы надеялись, и время выполнения только увеличивается по мере возрастания объема обрабатываемой задачи. Но дело не только в скорости. Сегодня все чаще нам приходится иметь дело со все большими объемами данных, и для связанных с ними вычислений программам требуется все больше ресурсов памяти, с которыми, несмотря на свою мощность, наши компьютеры не справляются.

Параллельное программирование позволяет выполнять фрагменты кода одной из наших программ одновременно, что значительно повышает производительность. Таким образом, параллельное программирование позволяет сократить время выполнения программы, более эффективно использовать ресурсы и иметь возможность выполнять более сложные операции, которые ранее считались недостижимыми.

Технологическая эволюция компьютеров и параллелизм

Сегодня для многих программистов параллельное программирование по-прежнему остается чем-то непривычным, поскольку это довольно новая технология. Дело в том, что еще несколько лет назад все компьютеры, доступные разработчикам, были оснащены одним единственным арифметическим логическим устройством (Arithmetic Logic Unit, ALU), и возможным было только последовательное программирование. Инструкции программы выполнялись по одной в последовательном порядке (см. рис. 1.1):



Рис. 1.1. Последовательное выполнение

Многие из вас, безусловно, помнят характеристики компьютера, которые обычно отображались в виде частоты процессора в герцах (Гц), что указывало на количество инструкций, которые могут быть выполнены в секунду. Мощность компьютера в первую очередь измерялась его вычислительной частотой. Чем выше было это значение, тем быстрее работали программы.

Концепция параллелизма формировалась постепенно, по мере развития аппаратного обеспечения компьютеров. До 1980-х годов такие возможности компьютеров были весьма ограниченными: они выполняли одну программу за другой, строго последовательно, команда за командой. Разумеется, в таких технологических условиях концепция параллелизма была совершенно невообразима.

С появлением процессора Intel 80386 возникла возможность прерывать выполнение одной программы для работы с другой. В результате появились такие концепции, как превентивное программирование и чередование процессов. Этот технологический прорыв привел к появлению эффекта псевдопараллелизма, поскольку для пользователя создавалось впечатление одновременной работы нескольких программ. С появлением последующего процессора Intel 80486 ситуация значительно улучшилась благодаря внедрению конвейерной системы, основанной на разделении программ на подзадачи. Они выполнялись независимо, чередуясь друг с другом для различных программ. Кроме того, внутренняя архитектура впервые позволила собирать несколько разных инструкций (даже из разных программ) и выполнять их все вместе одновременно (но не синхронно). И именно на этом этапе произошло реальное становление параллельного программирования. Фрагменты инструкций различных подзадач выполняются по порядку, чтобы быть выполненными как можно скорее (см. рис. 1.2):

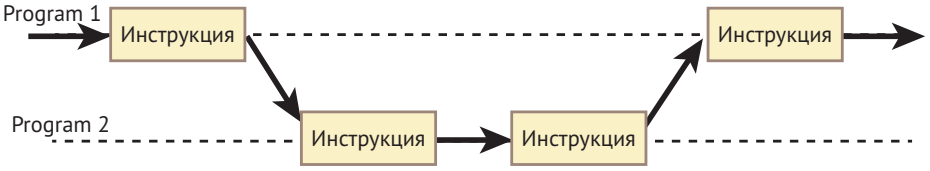


Рис. 1.2. Параллельное выполнение

Это продолжалось более десятилетия, в течение которого выпускались все более мощные модели процессоров, способные работать на более высоких частотах, чем предыдущие. Но вскоре ситуация перешла в кризисную фазу из-за ряда проблем и физических ограничений. Увеличение частоты исполнения приводит к одновременному увеличению тепловыделения и, как следствие, энергопотребления. Стало очевидно, что повышение частоты рано или поздно достигнет границ возможного.

Так процессоры достигли нового уровня инноваций с внедрением ядер в свою систему. Эти ядра, также известные как логические процессоры, позволяют имитировать наличие нескольких процессоров в одном центральном про-

цессоре, в результате чего получаются многоядерные процессоры. На практике это привело к появлению многопроцессорных компьютеров, способных одновременно и параллельно выполнять инструкции из разных программ. Именно поэтому в начале 2000-х годов появилось параллельное программирование, дающее разработчикам возможность одновременно выполнять различные фрагменты одной и той же программы.

Процессоры, ядра, потоки и процессы

Чтобы понять концепции, рассматриваемые в этой книге, необходимо сначала разобраться, что такое потоки (threads) и процессы (processes) и как они взаимосвязаны с режимами обработки данных центральным процессором и ядрами.

Это не какие-то абстрактные понятия, а реальные объекты, существующие в нашей операционной системе. Познакомиться с ними можно непосредственно в нашей операционной системе. Например, если вы работаете в Windows, откройте **Task Manager** (Диспетчер задач) и перейдите на вкладку **Performance** (Производительность).

Вы увидите окно, очень похожее на то, что показано на рис. 1.3, где можно в режиме реального времени отслеживать потребление различных ресурсов, таких как процессор, память и сеть Wi-Fi:

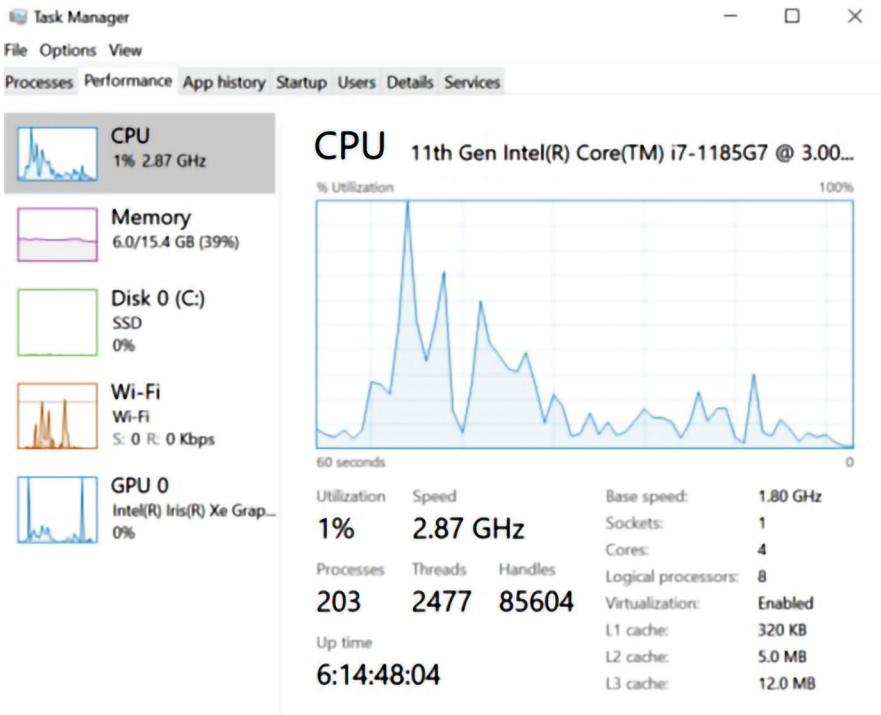


Рис. 1.3. Диспетчер задач в Windows

Кроме того, здесь также отображается различная информация, такая как количество процессов и запущенных в данный момент потоков. Справа перечислены некоторые характеристики системы, на которой мы работаем, например количество ядер.

Если же вы работаете в системах Linux, таких как Ubuntu, вы можете запустить соответствующее приложение, введя в терминале:

```
$ top
```

Появится экран, очень похожий на представленный на рис. 1.4:

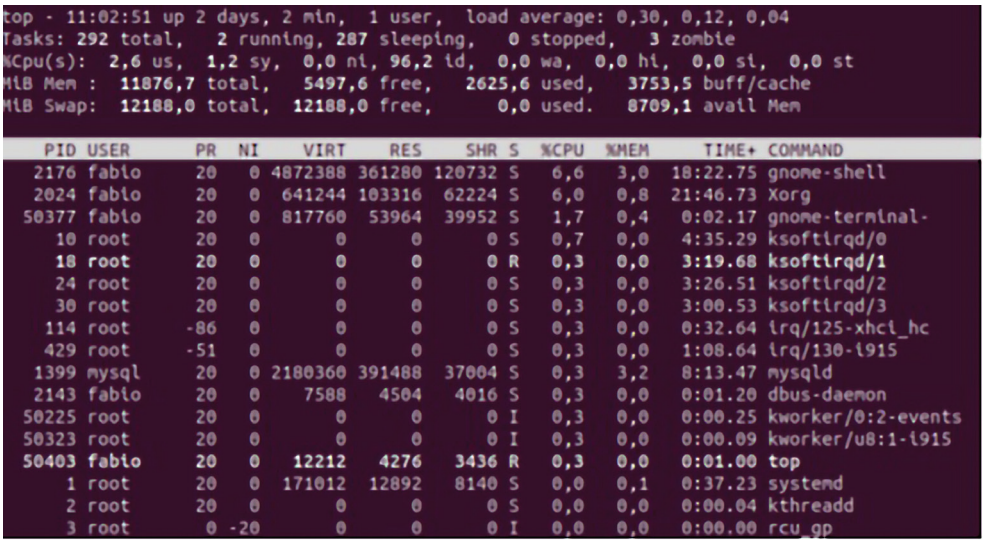


Рис. 1.4. top в терминале Ubuntu

Как можно видеть, в верхней части отображаются все используемые ресурсы с их значениями, которые обновляются в режиме реального времени. В нижней части находится список всех активных процессов в операционной системе. Каждый процесс идентифицируется уникальным номером – идентификационным номером процесса (process identification number, PID).

Поскольку системы Linux гораздо более гибкие и мощные прежде всего благодаря многочисленным командам оболочки, мы также можем отслеживать все потоки, связанные с каждой отдельной командой. Для этой цели воспользуемся более специфической командой для мониторинга процессов: `pid`.

```
$ pid -T <PID>
```

Опция `-T` используется для отображения потоков, которые являются вполне реалистичными для процесса. Затем `pid` процесса, который вы хотите отслеживать в деталях, передается команде `pid`. В данном случае, при выборе, например, процесса с `pid 2176`, будет получен результат, показанный на рис. 1.5, на котором все потоки предыдущего процесса отображаются с их идентификационным номером, `SPID`:

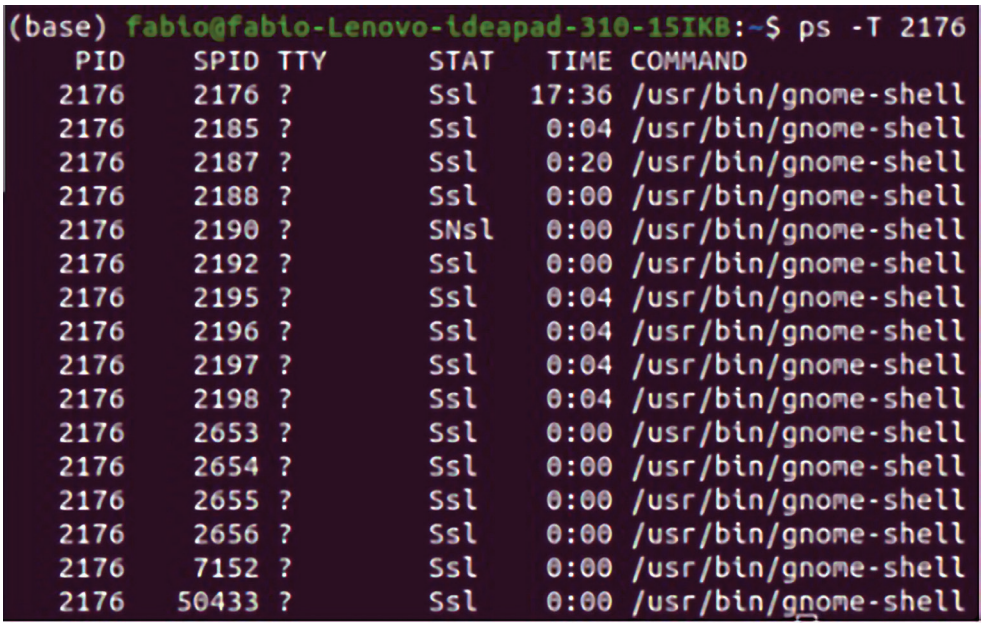


Рис. 1.5. Результаты выполнения команды pid в терминале Ubuntu

Центральный процессор (Central Processing Unit, CPU) – это истинный «мозг» нашего компьютера, где в основном и происходит обработка нашего кода. Производительность CPU характеризуется циклами, или единицами времени, которые он использует для выполнения операции. Обычно мы указываем мощность центрального процессора как частоту циклов в секунду (см. значение скорости 2,87 ГГц на рис. 1.3).

Центральный процессор может иметь одно (одноядерный процессор) или несколько ядер (многоядерный процессор). Ядра (cores) представляют собой блоки выполнения данных внутри центрального процессора. Каждое ядро способно выполнять несколько процессов. Процесс (process) – это, по сути, программа, которая запускается на компьютере и для которой зарезервирован участок памяти. Кроме того, каждый процесс сам может запускать другие процессы (подпроцессы) либо запускать один (`MainThread`) или несколько потоков (threads) внутри себя. Графическое представление всего этого показано на рис. 1.6.

Потоки, в свою очередь, можно рассматривать как подпроцессы, которые выполняются одновременно в рамках одного процессора. Как и процессы, потоки также имеют ряд схожих механизмов, которые управляют их синхронизацией, обменом данными и переходами состояний во время их выполнения: готово (ready), выполняется (running) и заблокировано (blocked).

Это общая структура, которую следует принимать во внимание, чтобы лучше понять принципы работы процессов и потоков в наших машинах и, следовательно, наилучшим образом реализовать программирование в параллельном режиме.

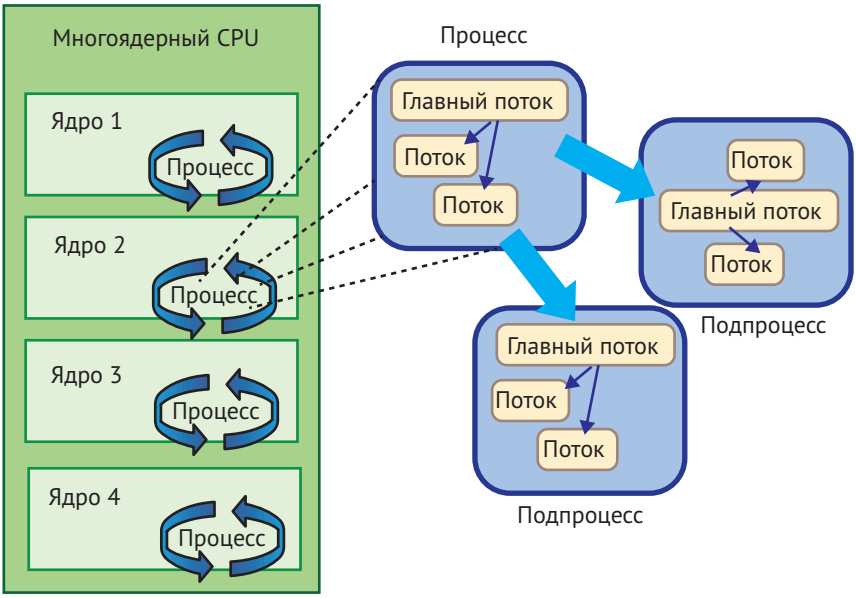


Рис. 1.6. Центральный процессор, ядро, процесс и потоки

Многозадачное и параллельное программирование

Очень часто возникает путаница между концепциями многозадачности (concurrency) и параллелизма (parallelism), поэтому эти два термина иногда используются как синонимы, но это нельзя считать корректным. Эти две концепции, хотя и взаимосвязаны, все же отличаются в контексте параллельного программирования, и очень важно понимать эти различия.

Начнем с того, что общего у этих двух концепций. И многозадачность, и параллелизм имеются в программах, которые должны выполнять несколько задач одновременно. Но именно в этом и заключается смысл многозадачности.

Многозадачность означает одновременное управление (а не выполнение) несколькими задачами, но они не обязательно будут выполняться одновременно.

Таким образом, даже программа, которая должна выполнять несколько задач одновременно, может делать это, только обрабатывая одну задачу за раз. Как только программа завершит выполнение инструкций, относящихся к какой-либо задаче или ее части (подзадаче), она перейдет к следующей задаче и т. д. Одна задача будет сменяться другой, и программа завершит свою работу. Если это поможет, можно представить себе, что задачи конкурируют друг с другом за право на выполнение.

Таким образом, даже если наш компьютер оснащен одноядерным процессором, конкурирующая программа без проблем запустится (см. рис. 1.7):

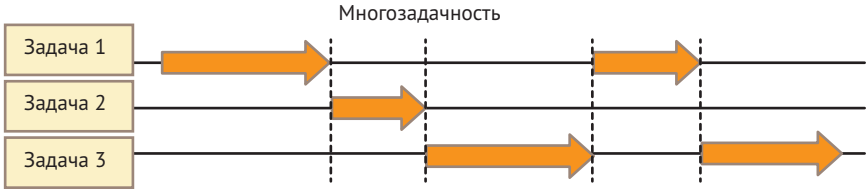


Рис. 1.7. Многозадачность на одноядерном процессоре

Пользователь видит одновременное выполнение нескольких задач, но внутри процессора в каждый момент времени выполняется только одна задача.

Однако многозадачное программирование также относится к многоядерным процессорам или многопроцессорным компьютерам. В этом случае может возникнуть следующая конкурентная ситуация.

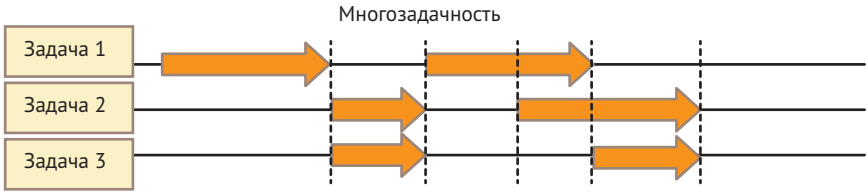


Рис. 1.8. Многозадачность на многоядерном процессоре

Как видно на рис. 1.8, ситуация усложняется. С учетом наличия нескольких процессоров (нескольких ядер) подзадачи могут быть распределены между ними и выполняться одновременно. Так возникает явление параллелизма.

Параллелизм означает одновременное выполнение нескольких задач. С учетом этого параллелизм представляет собой особый случай многозадачного (конкурентного) программирования.

Параллелизм имеет место при назначении программой каждой задачи отдельному ядру центрального процессора, чтобы каждая из них могла обрабатываться одновременно, то есть параллельно, как показано на рис. 1.9.



Рис. 1.9. Параллелизм в многоядерном процессоре

Отсюда следует, что для реализации параллелизма требуется аппаратное обеспечение с несколькими процессорными ядрами, а именно многоядерный процессор. В одноядерном процессоре можно лишь имитировать многозадачность, но невозможно реализовать параллелизм.

Потоки и процессы в Python для моделей многозадачности и параллелизма

Теперь, когда разница между многозадачным и параллельным программированием понятна, перейдем к следующему шагу. Во многих языках программирования принято связывать потоки с многозадачностью, а процессы – с параллелизмом. На самом деле эти два элемента операционной системы объединяют в себе две разные функции: многозадачность и параллелизм.

Однако в случае с Python целесообразно разделить эти случаи на две отдельные модели программирования. На самом деле потоки в Python не ведут себя так же идеально, как потоки в операционной системе. Потоки в Python не могут выполняться одновременно и, следовательно, не могут работать параллельно. Работа с потоками в Python похожа на работу с одноядерным процессором, хотя в действительности это не так.

Проблема потоков в Python: GIL

Тот факт, что потоки в Python, в отличие от других языков программирования, не могут выполняться одновременно на двух разных ядрах, напрямую связан с самим интерпретатором Python. Дело в том, что интерпретатор, на котором всегда выполнялся код Python, был реализован в **CPython**, и во время его реализации было обнаружено, что он не является в полной мере ориентированным на работу в многопоточной среде. Другими словами, чем больше потоков обращалось к одному и тому же объекту (память является общей для всех потоков), тем чаще возникала несогласованность из-за возникновения конкурентных условий доступа. Чтобы избежать этой серьезной проблемы, в интерпретатор была включена Глобальная блокировка интерпретатора (Global Interpreter Lock, GIL). В результате разработчики Python приняли решение, что в рамках одного процесса может одновременно выполняться только один поток, что исключает параллелизм для этого типа объектов (отсутствие многопоточности).

Глобальная блокировка интерпретатора может быть получена только одним потоком за одно время, в то время как все остальные потоки находятся в режиме ожидания. Как только поток завершает свою задачу, GIL освобождается и переходит к следующему потоку. Такое выполнение является действительно многозадачным. Многозадачные программы, как правило, менее затратны в плане ресурсов, чем параллельные программы, поскольку создание новых процессов значительно дороже, чем создание потоков. Однако следует иметь в виду, что операции установки и снятия блокировки замедляют выполнение всей программы.

Но все не так плохо. Позже вы увидите, как адаптировать эту особенность языка Python к параллельным моделям программирования. Кроме того, многие внешние библиотеки не зависят от GIL, поскольку они были реализованы на других языках, таких как C и Fortran, и поэтому они используют внутренние механизмы, поддерживающие многозадачность. Одной из них является `NumPy`, ключевая библиотека для числовых вычислений в Python.

Отказ от GIL для достижения многопоточности

Возможность исключения GIL из интерпретатора Python всегда вызывала горячие дискуссии. Однако с течением времени эта возможность становится все более труднодостижимой, поскольку исключить GIL без исключения многих официальных и сторонних пакетов и модулей, используемых в Python, слишком сложно.

Другой возможностью могло бы стать использование других реализаций Python, отличных от CPython. Самая распространенная из них, PyPy, известная своей более высокой производительностью, к сожалению, также реализовала GIL, очень похожий на GIL в CPython. Зато Jython, версия Python, реализованная на Java, и IronPython, реализованная на .NET, не имеют GIL и могут использовать многопоточность, тем самым используя преимущества наличия нескольких ядер или процессоров.

Потоки и процессы в Python

Подводя итог, можно сказать, что потоки и процессы – это инструменты, которые Python предоставляет нам для создания программ в многозадачной и параллельной форме соответственно.

В табл. 1.1 представлено сравнение некоторых характеристик этих двух концепций, которые необходимо учитывать при программировании.

Таблица 1.1. Сравнение характеристик потоков и процессов в Python

Потоки	Процессы
Совместное использование памяти (процесса)	Без совместного использования памяти
Незначительное потребление ресурсов	Требуют большого количества ресурсов
Быстрое создание с малой нагрузкой	Более медленное и ресурсоемкое создание
Требуются механизмы синхронизации	Синхронизация не требуется

Многозадачность и параллелизм в Python

Итак, для многозадачного программирования в Python, учитывая поведение потоков при использовании этого языка, необходимо скорректировать ранее данное определение многозадачности, исключив возможность параллелизма.

Многозадачность означает управление несколькими задачами одновременно, но при этом они не обязательно будут выполняться одновременно.

Таким образом, параллельное программирование в Python можно представить в виде потоков, каждый из которых выполняет свои задачи независимо и в конкуренции друг с другом. Они будут чередоваться друг с другом в общем потоке выполнения до момента завершения, как показано на рис. 1.10.

В случае параллельного программирования на Python процессы идеально подходят для одновременного, а именно параллельного выполнения задач. Каждому из них будет назначена задача, и все они вместе смогут одновременно выполнять инструкции внутри них до завершения программы, как показано на рис. 1.11.

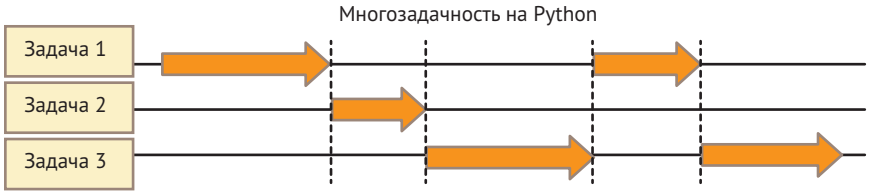


Рис. 1.10. Многозадачность в Python



Рис. 1.11. Параллелизм в Python

Из этого следует, что, в то время как для других языков программирования термины «многозадачность» и «параллельность» могут вызывать путаницу, в Python многозадачность и параллелизм, не имеющие общих аспектов одновременности, являются двумя совершенно разными концепциями.

Облегченная многозадачность с гринлетами

Как только что было показано, многозадачность использует потоки в качестве эффективного инструмента для реализации своих программных моделей.

Однако, помимо потоков, Python предлагает еще одну возможную альтернативу: гринлеты (greenlets). С точки зрения многозадачности использование гринлетов или потоков равнозначно, поскольку в Python потоки никогда не выполняются параллельно, и поэтому в этом языке программирования оба варианта отлично работают в многозадачном программировании. Но создание гринлетов и управление ими требует гораздо меньше ресурсов, чем потоки. Именно поэтому их использование в программировании определяется как легкая многозадачность (light concurrency). По этой причине гринлеты часто используются при необходимости управления большим количеством простых функций ввода-вывода, как, например, в веб-серверах. Позже в этой книге на нескольких простых примерах будет показано, как создавать гринлеты и как ими управлять.

Параллельное программирование с помощью Python

Понадобится понимание роли, которую могут играть потоки и процессы в Python. После этого мы сможем углубиться в параллельное программирование, тесно связанное с языком Python.

В этом языке параллельное программирование выражается исключительно в процессах. Программа делится на несколько параллельных подзадач, каждая из которых назначается отдельному процессу. Внутри каждого из них мы можем выбирать, выполнять ли различные шаги синхронно или асинхронно.

Синхронное и асинхронное программирование

В этой книге, а также во многих других онлайн-источниках о параллельном программировании нередко упоминаются термины *синхронный* (synchronous) и асинхронный (asynchronous), которые иногда для краткости называют **sync** и **async**. Во всех этих случаях речь идет о двух различных моделях программирования.

Осознанно или нет, но, когда мы разрабатываем программу для параллельного выполнения или для конкурирующих между собой процессов или потоков, вполне естественно использовать синхронную структуру. Это связано с тем, что, как правило, у всех нас есть опыт последовательного программирования, и это влияет на наше мышление. Иными словами, при наличии двух или более процессов (но это могут быть как потоки, так и простые функции внутри программы) один процесс (Процесс 1 на рис. 1.12) продолжает свое выполнение до того момента, когда он выполняет внешний вызов, передавая выполнение другому процессу для получения услуги, вычисления или любой другой операции. Другой процесс (Процесс 2 на рис. 1.12) будет выполнен для завершения своей задачи, а затем вернет результат выполнения функции исходному процессу, который в это время находился в режиме ожидания. После получения необходимого результата исходный процесс возобновит свое выполнение.

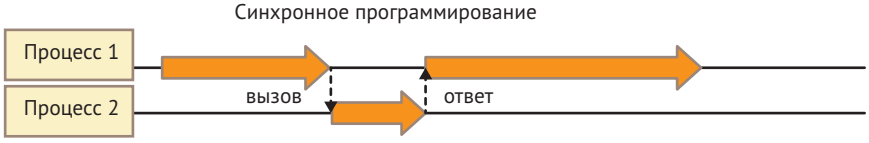


Рис. 1.12. Синхронное программирование

В действительности асинхронные модели программирования гораздо более эффективны, чем синхронные, как с точки зрения более рационального потребления вычислительных ресурсов, так и с точки зрения затрат времени на выполнение программы. Давайте рассмотрим предыдущий случай, но на этот раз посмотрим на него с точки зрения асинхронности, как показано на рис. 1.13.

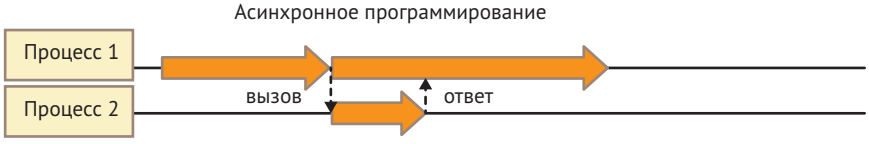


Рис. 1.13. Асинхронное программирование

Как и в случае синхронного выполнения, начальный процесс (Процесс 1 на рис. 1.13) продолжит свое выполнение до вызова, который запустит второй процесс (Процесс 2 на рис. 1.13). Но на этот раз начальный процесс не будет приостанавливать свое выполнение, чтобы дождаться завершения второго процесса. Он будет продолжать свое выполнение независимо от того, когда и как он получит результат выполнения второго процесса.

Как можно догадаться, асинхронное программирование дает нам преимущество во многих случаях, когда мы тратим много времени на ожидание операций, требующих внешнего ответа или длительного времени выполнения. Поэтому очень важно хорошо разбираться в обеих моделях, если вы хотите использовать весь потенциал параллельного программирования.

Что касается практической реализации, то, хотя она еще не совсем очевидна для нас, она является вполне осуществимой. Все языки программирования имеют внутренние механизмы для ее реализации. Мы подробно рассмотрим асинхронное программирование в главе 6 «Программирование GPU с CUDA для максимальной производительности».

Предварительная обработка и свертка

В параллельном программировании широко используется схема Map-Reduce, которая главным образом основана на двух этапах:

- предварительной обработке – отображении, распределении (mapping);
- свертке – редукции (reducing).

Первый этап предварительной обработки заключается в разделении задач, которые должны быть выполнены программой, на несколько частей (задач), а затем в распределении их между различными процессами, которые будут выполнять их одновременно, то есть параллельно. Зачастую при выполнении каждого процесса получается результат. Поэтому после этапа, непосредственно связанного с параллельным выполнением, следует этап, на котором все результаты должны быть объединены, а именно этап свертки. На рис. 1.14 представлена диаграмма, которая помогает лучше понять суть изложенного выше.

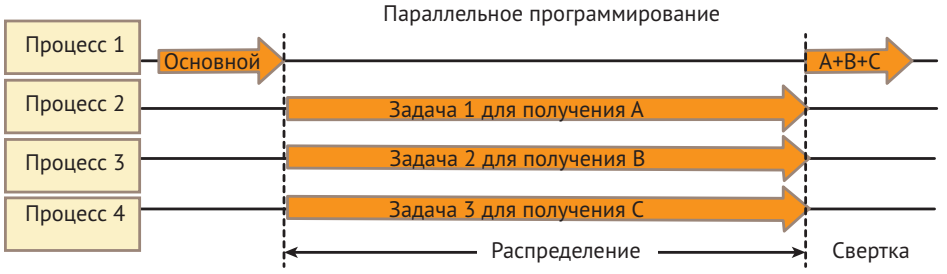


Рис. 1.14. Предварительная обработка и свертка при параллельном программировании

Операции, ограниченные CPU и вводом-выводом

В ходе стадии разработки параллельной программы необходимо уделить должное внимание отдельным задачам и оценить, не требуют ли некоторые из них слишком много времени для выполнения. Если это так, то может прои-

зойти значительное снижение производительности, поскольку все остальные процессы будут ждать завершения фазы предварительной обработки. В действительности для перехода к фазе свертки потребуются все результаты, полученные для каждого процесса. Рассмотрим случай, подобный тому, который представлен на рис. 1.15, где один из параллельных процессов требует слишком много времени на выполнение по сравнению с другими. В этом случае все остальные процессы будут ждать, чтобы продолжить выполнение и передать результаты на этап свертки. В этом случае параллельное программирование теряет свою эффективность.

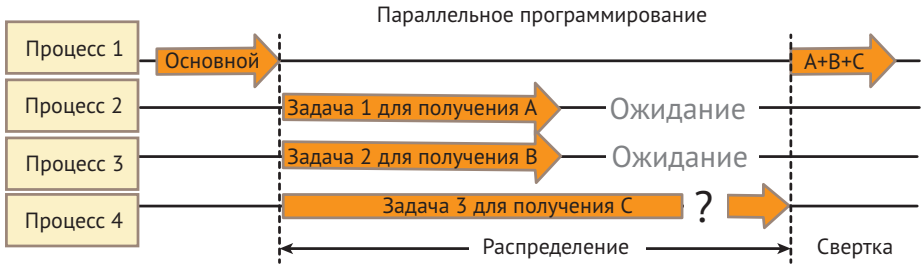


Рис. 1.15. Параллельное программирование с низкой производительностью

В таких случаях необходимо учитывать различные операции, выполняемые в каждом отдельном процессе (задаче). Эти задачи могут включать внутренние операции, такие как чтение файла или вызов внешнего веб-сервиса. В этом случае процесс будет вынужден ждать ответа от внешнего устройства, и поэтому время выполнения может быть непредсказуемым. Такой тип операций называется I/O-связанным, или зависимым от ввода-вывода (I/O bound). Операции, которые включают только внутренние вычисления CPU, называются зависимыми от центрального процессора, или CPU-связанными (CPU bound).

При параллельном программировании, когда речь идет о подзадачах или операциях, связанных с центральным процессором, использование нескольких процессов, которые выполняют инструкции параллельно, делает программу более эффективной. Но в случае ограничений ввода-вывода приходится работать по-другому.

В этом случае наиболее подходящим является многозадачное программирование, и здесь на сцену выходят потоки. Внутри процесса мы можем создать несколько потоков. Один из них будет продолжать заниматься операцией, связанной с CPU, а другие будут заниматься различными операциями, связанными с вводом-выводом. Когда один из этих потоков, содержащих связанные с вводом-выводом операции, ожидает данных или ответа от внешнего источника, другие потоки продолжают выполнять свои операции.

В этом случае благодаря одновременному запуску потоков мы сокращаем время выполнения.

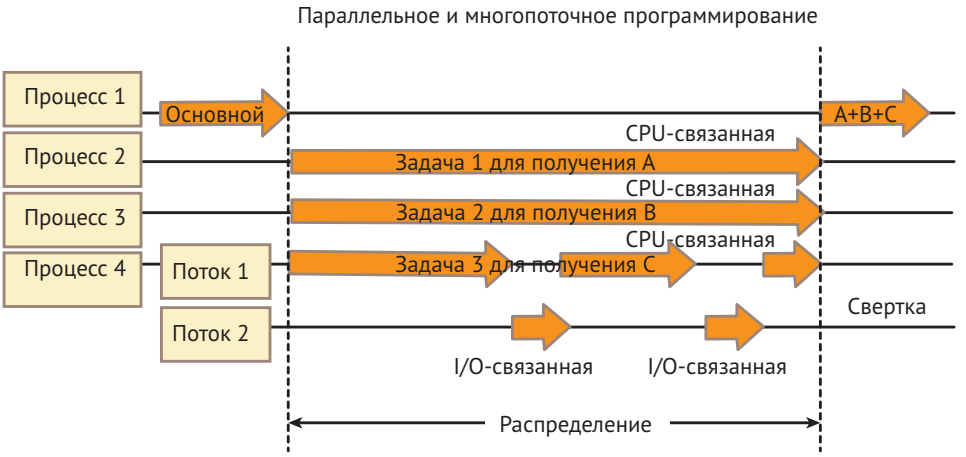


Рис. 1.16. Многозадачность для операций, связанных с вводом-выводом

Как видно на рис. 1.16, создание дополнительного потока, помимо **MainThread** (который всегда присутствует в процессе), позволяет управлять (асинхронно или синхронно) операцией, связанной с вводом-выводом, в отдельном потоке, в то время как **MainThread** продолжает обработку данных.

Дополнительные меры предосторожности при параллельном программировании

При многозадачном программировании необходимо уделять особое внимание использованию потоков, в частности управлению распределенными данными. На рис. 1.17 наглядно показано, что потоки в рамках одного процесса имеют как собственную память (недоступную для других потоков), так и распределенное пространство памяти, в котором находятся объекты, доступные для всех потоков.

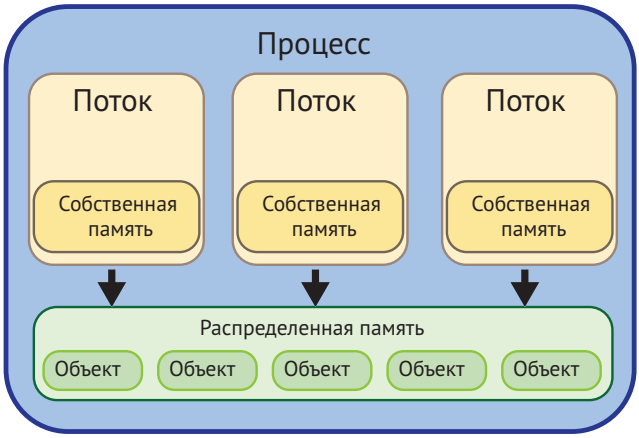


Рис. 1.17. Процесс с потоками

Несмотря на присутствие GIL в интерпретаторе Python, работающем с несколькими потоками, для предотвращения проблем с несогласованностью данных по-прежнему необходимо блокировать глобальные объекты, присутствующие в общем пространстве памяти. Но как это возможно, если GIL обеспечивает одновременное выполнение только одного потока в процессе?

Дело в том, что интерпретатор будет заботиться только о внутренних объектах Python, но что касается объектов, которые мы определяем и создаем во время выполнения нашей программы, то никакого контроля или блокировки не будет, они управляются независимо. Заблокировать глобальные объекты, которые мы создаем, чтобы не получить неожиданных результатов, придется нам самим.

Для этого, как будет показано вскоре, в стандартной библиотеке Python есть модули, которые, помимо реализации процессов и, в частности, потоков, предоставляют множество инструментов для управления блокировкой упомянутых выше глобальных объектов.

Модули потоков и многопроцессорности

Приступая к непосредственной реализации, мы можем воспользоваться преимуществами как многозадачных, так и параллельных моделей с *потоками* и *процессами*, с помощью двух модулей из стандартной библиотеки: threading и multiprocessing. Они предоставляют набор функций Python, которые взаимодействуют с операционной системой для создания, выполнения и управления процессами и потоками в Python.

Обратите внимание: если вы заметили, что не существует специального модуля для потоков под названием *multithreading* по аналогии с модулем для процессов, то это лишь потому, что Python на самом деле не поддерживает многопоточность, а может выполнять только один поток в один момент времени.

Модуль **threading** предоставляет уровень абстракции для `_thread` – низкоуровневого модуля, который реализует примитивы для работы с несколькими потоками. Кроме того, он включает в себя множество инструментов, которые помогают программисту в сложной задаче управления параллельными системами, такими как потоки: **lock** (блокировка), **condition** (состояние) и **semaphores** (семафоры). Функциональность этого модуля вместе с этими инструментами будет подробно рассмотрена в следующей главе 2 «Параллельное программирование с потоками» с рядом примеров кода, которые помогут вам лучше понять, как и когда их использовать.

Модуль **multiprocessing**, в свою очередь, предлагает эффективный API для реализации *параллелизма на основе процессов* (process-based parallelism).

Помимо создания и управления процессами, этот модуль также предлагает большое количество функций, которые помогают управлять одновременно существующими процессами в рамках одной программы. Например, **Queue** и **Pipe** – это объекты, которые позволяют обмениваться информацией (объектами) между различными процессами, или пулы, которые упрощают управление несколькими процессами одновременно. Этот модуль и его функциональные возможности будут подробно рассмотрены в главе 3 «Многопроцессорные вычисления и библиотека mpi4py».

Распределение и использование памяти

До сих пор мы обсуждали параллельное программирование исключительно с точки зрения его выполнения, но при работе с аппаратными системами, допускающими параллельную обработку, очень важно разбираться в организации памяти.

На практике, даже если у нас есть очень мощный и быстрый процессор с множеством вычислительных модулей, таких как ядра, производительность программы все равно будет во многом зависеть от организационной структуры памяти.

Все операции, связанные с передачей данных в память, как правило, не так быстры, как операции в процессоре, и во время выполнения этих операций память остается занятой до конца цикла памяти, и никакой другой компонент не может ее использовать.

С организацией памяти тесно связана концепция взаимодействия между различными компонентами, участвующими в работе программы. Формы взаимодействия зависимы непосредственно от конкретной организации памяти, и поэтому для идеальной работы параллельной программы потребуется оптимизировать передачу информации между выполняемыми объектами, такими как процессы и потоки, которую обеспечивает память. В зависимости от организации памяти необходимо использовать различные механизмы для синхронизации передачи информации между различными объектами программы, чтобы избежать риска несогласованности данных или других проблем, таких как зависания и некорректное выполнение.

Таким образом, можно сказать, что память играет очень важную роль в эффективности параллельного программирования, и поэтому очень важно анализировать ее поведение в процессе реализации наших проектов.

Распределение памяти в процессах

В языке Python возможны три различные модели организации памяти в рамках одной программы:

- полностью параллельная;
- распределенная или общая память;
- передача сообщений.

Эти модели можно описать следующим образом.

- Первая модель, полностью параллельная, относится к особому случаю, когда программные объекты, будь то потоки или процессы, не требуют обмена информацией и могут выполняться до конца, после чего отдельные результаты объединяются. На самом деле существуют особые алгоритмы, которые соответствуют этой модели и которые, собственно, по этой причине называются «неудобно параллельными» (embarrassingly parallel).
- Общая (shared) память – это модель организации памяти, обычно используемая для потоков в Python. Потоки внутри процесса могут обмениваться данными через общую память, доступную самому процессу. Существует несколько возможных механизмов обмена данными, которые соответствуют этой модели, некоторые из них являются допустимыми, а другие – нет, и мы рассмотрим некоторые из них в этой книге.
- Передача сообщений (message passing), с другой стороны, является моделью организации памяти для процессов. Они фактически не имеют общей памяти, и поэтому их единственным способом взаимодействия является обмен сообщениями. Существует несколько решений этой проблемы, обычно предлагаемых в виде готовых пакетов под названием Message Passing Interface (MPI). Стандартная библиотека Python предлагает модуль MPI, специально предназначенный для выполнения этой задачи.

Распределение памяти между несколькими процессорами

Пока что речь шла о системе с одним многоядерным процессором. Но в реальности параллельное программирование естественным образом распространяется на использование нескольких процессоров. Они могут находиться на одной или на разных машинах, подключенных друг к другу каким-либо образом.

Несомненно, что и в этом случае организация памяти играет важную роль в обеспечении эффективной работы параллельных вычислений.

Две предыдущие модели, с общей памятью и с передачей сообщений, также используются в этой области в виде следующих двух моделей:

- общая (shared) память;
- распределенная (distributed) память.

В системах на основе **модели общей памяти** все имеющиеся процессоры могут обращаться к определенной области памяти для обмена данными и пересылки информации. Системы такого типа обычно построены на основе физической шины, которая позволяет подключать определенное количество физически разделенных процессоров (которые могут находиться на одной или разных машинах), как показано на рис. 1.18.

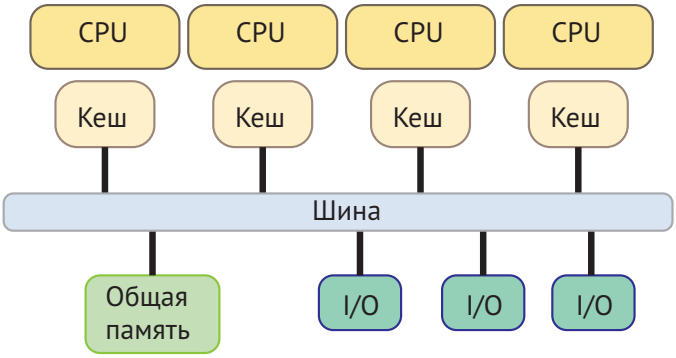


Рис. 1.18. Системы с общей памятью

Каждый процессор имеет свою собственную локальную память в виде кеша, как правило, с высокой производительностью, поскольку обмен данными между центральным процессором и этой областью памяти происходит очень часто. Однако эта область имеет ограниченный объем, и зачастую различным процессорам приходится распределять данные для работы, что возможно только с помощью общей памяти, подключенной к общей шине.

На этом этапе ситуация становится сложной. Программисту необходимо внимательно контролировать синхронизацию данных, которые одновременно используются различными процессорами.

Один из процессоров берет значение данных из общей памяти и копирует его в свой кеш для дальнейшей обработки. В это время другой процессор нуждается в том же значении и также копирует его из общей памяти в свой кеш. Через некоторое время первый процессор заканчивает обработку и записывает результат в общую память, обновляя ее содержимое. Однако тем временем второй процессор обрабатывает значение, которое уже не является актуальным, и поэтому здесь происходит потеря согласованности данных. Таким образом, становится очевидным, что необходимо реализовать механизмы управления параллелизмом и синхронизации, аналогичные механизмам для потоков в рамках одного процесса (с помощью аппаратных или программных средств).

Так зачем же использовать эту модель? Главная причина заключается в том, что системы с общей памятью работают очень быстро, поскольку в значительной степени зависят от аппаратного обеспечения, а не от программного. По сути, многие механизмы управления и синхронизации доступа процессоров к ресурсам общей памяти могут быть реализованы с помощью аппаратного обеспечения.

Другая модель, получившая широкое распространение в настоящее время, – это модель распределенной памяти (Distributed Memory). В отличие от предыдущей модели вместо физической шины, соединяющей различные процессоры между собой, здесь используется сетевое соединение. В действительности эта модель в основном используется для процессоров, расположенных на физически разделенных машинах, даже на значительном расстоянии друг от друга. Схема системы этого типа показана на рис. 1.19.

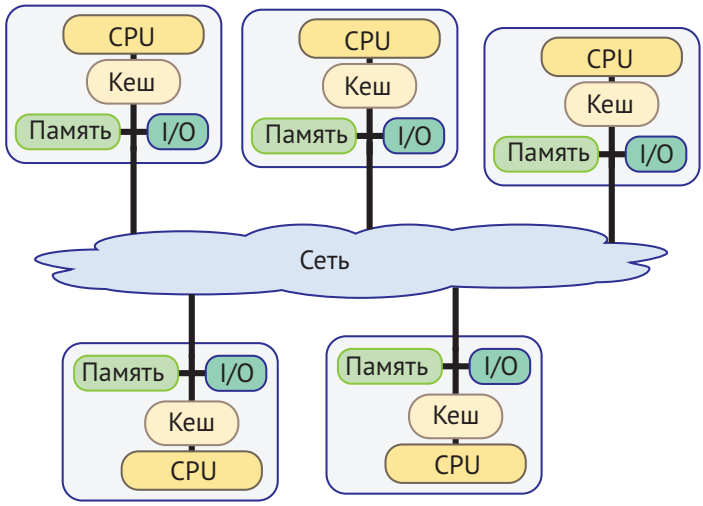


Рис. 1.19. Система распределенной памяти

В модели этого типа каждый процессор, помимо собственного выделенного кеша, имеет также локальную память, которую он обычно может использовать в полном объеме. Когда возникает необходимость обмена данными с другим процессором, данные передаются по сети. Таким образом, больше не возникает проблем с согласованностью данных, поскольку каждый процессор отвечает за свои собственные данные. Еще одним преимуществом является то, что, поскольку больше нет физической шины, а есть только сетевое соединение, количество процессоров, которые можно добавить к этой системе, является теоретически бесконечным.

Однако недостатком этой модели является то, что сетевое соединение работает не так быстро, как физическая шина, но прежде всего для взаимодействия и обмена данными между различными процессорами требуется механизм передачи сообщений. Управление сообщениями, их создание, отправка и чтение отдельными процессорами, с одной стороны, устраняет проблемы с синхронизацией, но с другой стороны, значительно замедляет выполнение программ.

Распределенное программирование

Существование сложных моделей, таких как распределенная память, привело к дальнейшей эволюции параллельного программирования, в результате чего его стали называть распределенным программированием (distributed programming). При этом программа может выполняться параллельно различными процессами, запущенными на разных машинах, подключенных к сети. Сегодня такая система получила широкое распространение и используется в такой степени, что появилось множество пакетов Python, предлагающих решения в этой области. В этой книге мы также рассмотрим это дальнейшее расширение параллельного программирования с использованием некоторых пакетов, доступных бесплатно в сети, и посвятим им отдельную главу 5 «Реализация параллелизма с помощью распределенных систем».

Оценка эффективности параллельного программирования

Одним из аспектов, тесно связанных с параллельным программированием, является необходимость разработки комплекса методов для оценки его эффективности. Такая оценка необходима на этапе разработки программы, чтобы определить, являются ли принятые решения целесообразными, или же следует рассмотреть другие варианты.

Когда вы принимаете решение использовать параллелизм, это означает, что вами движет необходимость решить большие проблемы в кратчайшие сроки. Однако для достижения этой цели необходимо учитывать множество факторов, таких как степень используемого параллелизма, аппаратное обеспечение и прежде всего модель программирования. Поэтому необходимо провести анализ производительности для оценки правильности выбранных решений в процессе разработки программы.

Существует целое множество индексов производительности, которые можно использовать для оценки производительности нашей программы путем оценки их количественных показателей. Это не что иное, как числовые значения, полученные в результате соответствующих вычислений, которые позволяют нам систематически и с высокой точностью сравнивать программы или алгоритмы друг с другом. Среди этих индексов наиболее известным и наиболее используемым является ускорение (speedup).

Ускорение

Ускорение (speedup) – это численное значение, которое выражает разницу между производительностью двух систем, выполняющих одну и ту же задачу. В нашем случае ускорение S можно рассматривать как отношение времени, затраченного на выполнение последовательной программы t_S, к времени t_p, затраченному на выполнение нашей параллельной программы, которая была создана для выполнения тех же операций. Время t является функцией количества процессоров N, которые могут быть CPU, ядрами или GPU, но обычно N обозначает количество процессоров. Таким образом, время, затрачиваемое в параллельной системе, может быть выражено как $t(N)$, а время в последовательной системе – как $t(1)$, поскольку оно эквивалентно системе с одним процессором:

$$S = \frac{t_S}{t_p} S = \frac{t_S}{t_p} = \frac{t(1)t(1)}{t(N)t(N)}.$$

Другими словами, ускорение дает нам информацию о преимуществах производительности, полученных благодаря применению параллельного решения по сравнению с последовательным. Кроме того, если мы сравним количество процессоров N с индексом ускорения, то сможем дополнительно классифицировать наш алгоритм или программу.

- Если $S = N$, то ускорение является линейным или идеальным.
- Если $S < N$, то ускорение является действительным.
- Если $S > N$, то ускорение является чрезвычайно действительным.

Индекс ускорения также связан с законом Амдаля, широко используемым в параллельных вычислениях. Этот закон позволяет прогнозировать максимальное ускорение, достижимое программой при использовании бесконечного числа процессоров. Он описывает, как в программе процент присутствующего последовательного кода определяет максимально достижимое значение ускорения:

$$S \le \frac{1}{1-\alpha}.$$

Скорость работы S – это индекс ускорения, а α – это доля времени, затрачиваемого на выполнение части программы, реализованной параллельно.

Таким образом, если у нас есть программа, на параллельное выполнение кода которой уходит 90 мин, а на последовательное – 10 мин, итого 100 мин, то $\alpha = 0{,}9$. В этом случае максимальное достижимое ускорение составит:

$$S \le \frac{1}{1-0,9} S \le \frac{1}{1-0,9} = \frac{1}{0,1} \frac{1}{0,1} = 10.$$

Максимальное ускорение, которое может быть достигнуто нашей программой, будет равно 10. Постепенно добавляя ядра или процессоры для выполнения, мы сможем постепенно улучшить производительность нашей программы, пока не достигнем ускорения, близкого к 10. Как только мы достигнем этого значения, дальнейшее добавление большего числа процессоров или ядер для параллельного выполнения не поможет добиться дальнейшего улучшения (см. график, показанный на рис. 1.20).

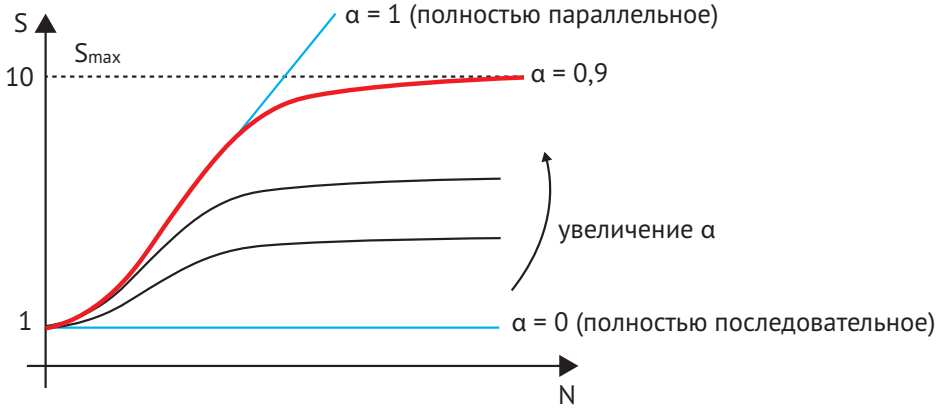


Рис. 1.20. Закон Амдаля

Таким образом, если мы хотим все больше и больше увеличивать скорость работы программы или алгоритма, понадобится свести к минимуму последо-

вательные части кода. Только в этом случае при увеличении количества процессоров (или ядер) производительность будет расти все более линейно.

Кроме того, после максимального увеличения объема параллельной части кода следующей задачей будет поиск оптимального компромисса, при котором ускорение будет максимально высоким, но при этом будет использоваться оптимальное количество процессоров (см. рис. 1.21).

Так можно избежать ненужной нагрузки из-за чрезмерного параллелизма, вызванного добавлением слишком большого количества процессоров.

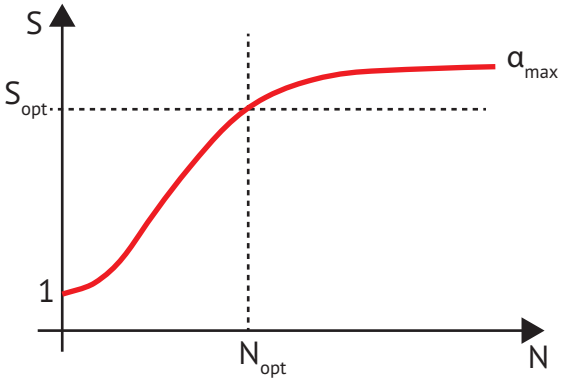


Рис. 1.21. Оптимизация количества необходимых процессоров

Масштабируемость

Масштабируемость (scaling) – это способность системы повышать свою вычислительную эффективность за счет добавления дополнительного оборудования (числа процессоров). В контексте параллельных вычислений масштабируемость относится к вопросу эффективности параллелизации, то есть к соотношению между действительной и идеальной скоростью работы при увеличении числа используемых процессоров.

Масштабируемость можно разделить на два типа:

- сильную масштабируемость;
- слабую масштабируемость.

Сильная масштабируемость происходит при увеличении количества процессоров при постоянном размере задачи. В идеальной ситуации это должно привести к постепенному снижению нагрузки на каждый процессор.

Слабая масштабируемость имеет место при увеличении количества процессоров одновременно с увеличением их производительности. В этом случае нагрузка на каждый процессор остается постоянной.

Сильное масштабирование было продемонстрировано с помощью закона Амдаля, согласно которому ускорение рассчитывается при постоянном размере задачи и увеличении количества процессоров. Уже в этом случае закон демонстрирует, что ускорение все же имеет максимальный предел из-за отсут-

ствия возможности 100 % параллелизации кода. Кроме того, существует множество других факторов, которые затрудняют достижение хорошего сильного масштабирования по мере увеличения количества процессоров. Например, увеличение количества процессоров приведет к росту нагрузки для обеспечения взаимодействия между ними.

Что касается слабой масштабируемости, то в этом случае ускорение не имеет верхних пределов и поэтому может расти бесконечно (теоретически). Это подтверждается законом Густафсона, который определяет расчет ускорения иначе, чем закон Амдаля:

$$S = (1 - \alpha) + \alpha * N.$$

В действительности Густафсон обратил внимание на то, что при увеличении размера задачи увеличение числа процессоров приводило к увеличению ускорения только параллельной части кода α, в то время как ускорение последовательной части $(1 - \alpha)$ не изменялось. И действительно на графике, представленном на рис. 1.22, видны тенденции линейного роста различных ускорений по мере постепенного добавления процессоров и, следовательно, без достижения предела.

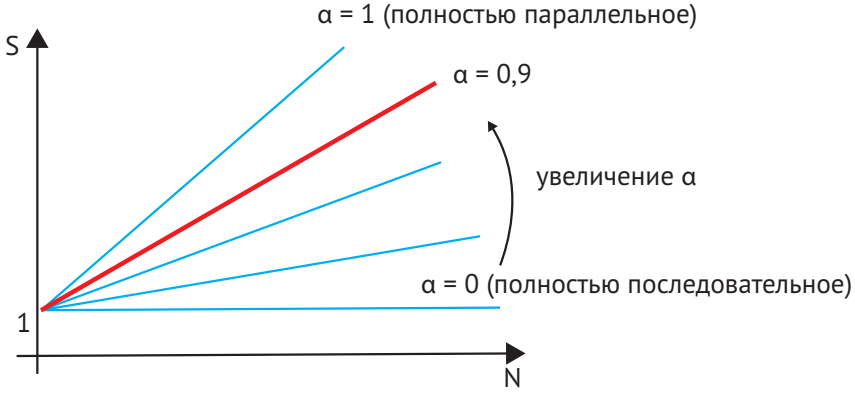


Рис. 1.22. Закон Густафсона

Благодаря концепции сильного и слабого масштабирования, а также двум законам Амдаля и Густафсона можно сделать следующие полезные выводы: для решения небольших задач лучше использовать небольшие системы, тогда как для решения более сложных задач лучше использовать крупные системы.

Тестирование производительности в Python

Систематическое тестирование производительности программ в различных условиях называется **сопоставительным анализом**, или **бенчмаркингом** (benchmarking). До сих пор мы рассматривали оценку производительности программы с теоретической точки зрения. А как обстоят дела на практике? В Python существует множество инструментов, позволяющих измерить производительность программы или фрагмента кода.

В следующих главах мы рассмотрим несколько практических примеров проведения таких измерений. Например, для вычисления времени (что также полезно для расчета ускорения) выполнения фрагмента кода мы будем использовать модуль time из стандартной библиотеки Python. Этот модуль предоставляет доступ к нескольким типам таймеров, и с помощью вызова метода `time()` мы сможем получить реальные показания секундомера. Затем, вычисляя разницу между прочитанными значениями времени, мы получим время, затраченное кодом на выполнение операций между двумя вызовами:

```
started = time.time()

# Код здесь

elapsed = time.time()
print("Elapsed time=", elapsed - started)
```

Профилирование

Профилирование (profiling) – это анализ частей программы, влияющих на ее производительность, и выявление узких мест.

В настоящее время в Python для этого доступно несколько инструментов, каждый из которых полезен по-своему. Что касается потребления ресурсов памяти, можно использовать мощный инструмент: **package memory profiler**. Этот модуль позволяет отслеживать потребление памяти различными процессами/задачами в Python. Кроме того, он позволяет выполнять построчный анализ кода на предмет потребления ресурсов и поэтому может быть полезен в качестве линейного профилировщика.

Заключение

В этой главе было подробно рассмотрено большинство концепций, лежащих в основе параллельного программирования. С течением времени параллельное программирование развивалось в соответствии с появлением новых технологий, совмещая концепции и сущности, которые постепенно развивались. Процессы и потоки, исполняемые в операционной системе, сопоставляются в параллельном программировании с помощью объектов process и thread, которые могут быть реализованы с помощью модулей *threading* и *multiprocessing* из стандартной библиотеки Python. В следующих двух главах мы рассмотрим использование этих двух модулей для параллельного программирования, а также возможности максимально эффективного применения всех их функций.

Что следует помнить

- Многозадачность: подразумевает одновременное управление несколькими задачами, но они не обязательно выполняются одновременно.

- Параллелизм: подразумевает одновременное выполнение нескольких задач.
- Потоки: в Python потоки не могут выполняться одновременно, а значит, не могут работать параллельно.

Вопросы

1. В чем разница между параллелизмом и многозадачностью?
2. Каковы преимущества и недостатки системы распределенной памяти?
3. Когда и как поддерживается слабое и сильное масштабирование?

Список литературы

- *https://wiki.python.org/moin/GlobalIntepreterLock*.
- *https://www.udacity.com/blog/2020/04/what-is-python-parallelization. html*.
- *https://www.infoworld.com/article/3542595/6-python-libraries-for-parallel-processing.html*.
- *https://en.wikipedia.org/wiki/Monkey_patch*.
- *https://greenlet.readthedocs.io*.
- *https://www.koyeb.com/blog/introduction-to-synchronous-and-asynchronous-processing*.
- *https://medium.com/fintechexplained/advanced-python-concurrency-and-parallelism-82e378f26ced*.
- *https://sebastianraschka.com/Articles/2014_multiprocessing.html*.
- *https://medium.com/fullstackai/concurrency-in-python-cooperative-vs-preemptive-scheduling-5feaed7f6e53*.
- *https://carpentries-incubator.github.io/lesson-parallel-python/*.
- *http://code.tutsplus.com/articles/introduction-to-parallel-and-concurrent-programming-in-python--cms-28612*.
- *https://en.wikipedia.org/wiki/Speedup*.
- *https://hpc-wiki.info/hpc/Scaling*.
- *https://www.meccanismocomplesso.org/python/*.

Глава 2

Параллельное программирование с потоками

В этой главе мы рассмотрим потоки и их параллельное программирование. В стандартной библиотеке Python есть модуль threading, который упрощает реализацию потоков и предоставляет множество полезных инструментов для их синхронизации. Вся глава будет посвящена описанию этого модуля, и мы по очереди рассмотрим все предоставляемые классы и их функциональность.

Структура

В этой главе мы обсудим следующие темы:

- потоки;
- Lock и Rlock;
- Semaphore;
- Condition;
- Event;
- ThreadPoolExecutor.

Потоки

Основными элементами параллельного программирования являются именно потоки, и для этого модуль `threading` предоставляет класс `Thread`:

```
class threading.Thread(group=None,
                       target=None,
                       name=None,
                       args=(),
                       kwargs={},
                       *, daemon=None)
```

Конструктор `Thread()` принимает ряд аргументов, среди которых наиболее важными и полезными являются `target` и `args`. Функция, которая будет вызвана в потоке, передается в `target`, а аргументы, которые будут ей переданы, передаются в `args`. Мы можем сразу оценить функциональность этого типа объекта

на практическом примере. В программе мы определяем пять потоков, которые будут конкурировать друг с другом. Все они будут ориентированы на одну и ту же функцию, которую для удобства назовем `function()`. Эта функция не будет выполнять задачи, а только займет короткий промежуток времени, чтобы имитировать время, затрачиваемое на выполнение набора инструкций:

```
import threading import time

def function(i):
  print ("start Thread %i\n" %i)
  time.sleep(2)
  print ("end Thread %i\n" %i)
  return

t1 = threading.Thread(target=function , args=(1,))
t2 = threading.Thread(target=function , args=(2,))
t3 = threading.Thread(target=function , args=(3,))
t4 = threading.Thread(target=function , args=(4,))
t5 = threading.Thread(target=function , args=(5,))
t1.start()
t2.start()
t3.start()
t4.start()
t5.start()
print("END Program")
```

Как видно из кода, сначала определяются пять экземпляров класса `Thread`, соответствующих такому же количеству потоков, с использованием переменных `t1`, `t2` и т. д. После этого запуск потоков будет инициирован вызовом метода `start()`. При запуске программы мы получим:

```
start  Thread 1
start  Thread 2
start  Thread 3
start  Thread 4
start  Thread 5
END Program
$ end Thread 1 end Thread 2
end Thread 4
end Thread 3
end Thread 5
```

Если проанализировать результат выполнения, можно заметить несколько важных моментов. Во-первых, программа запускает все пять потоков одновременно, а затем завершает работу, не дожидаясь их завершения, в результате чего появляется приглашение ввести новую команду. На самом деле пять потоков продолжают выполняться в фоновом режиме, выводя результаты в командную строку.

Еще один интересный момент, который можно увидеть в выводе, заключается в том, что порядок завершения их выполнения отличается от порядка запуска и может варьироваться от выполнения к выполнению. Это нормальное поведение потоков, поскольку они выполняются совместно. Продолжительность и порядок их выполнения обычно непредсказуемы. Поэтому важно использовать методы синхронизации, которые мы рассмотрим позже в этой главе.

Метод join()

В приведенном выше примере было показано, что при запуске потоков во время выполнения программы можно наблюдать, как программа заканчивает работу до завершения работы потоков. Эта проблема решается очень просто, поскольку модуль threading предоставляет для этой цели метод `join()`. Этот метод, запускаемый в потоке, заставляет программу ждать завершения его выполнения перед закрытием. Таким образом, имея несколько потоков, мы вызовем метод `join()` для каждого из них.

Затем применим его к предыдущему примеру:

```
import threading
import time

def function(i):
  print ("start Thread %i" %i)
  time.sleep(2)
  print ("end Thread %i" %i)
  return

t1 = threading.Thread(target=function , args=(1,))
t2 = threading.Thread(target=function , args=(2,))
t3 = threading.Thread(target=function , args=(3,))
t4 = threading.Thread(target=function , args=(4,))
t5 = threading.Thread(target=function , args=(5,))
t1.start()
t2.start()
t3.start()
t4.start()
t5.start() t1.join()
t2.join()
t3.join()
t4.join()
t5.join()
print("END Program")
```

В этом случае запуск программы даст следующий результат:

```
start Thread 1
start Thread 2
```

```
start Thread 3
start Thread 4
start Thread 5
end Thread 3
end Thread 1
end Thread 2
end Thread 4
end Thread 5
END Program
```

Как можно видеть, теперь программа будет ждать завершения выполнения всех потоков перед своим закрытием.

Рассмотрим этот вопрос более подробно. Вызовы метода `join()` могут использоваться в определенный момент выполнения программы в качестве простого примера синхронизации. Например, мы хотим, чтобы была выполнена только часть кода основной программы, а затем хотим дождаться начала выполнения потоков. Затем мы хотим перезапустить программу, выполнить другие операции или запустить другие потоки.

Давайте рассмотрим простой пример, полученный после изменения предыдущего кода:

```
…
t1.start()
t2.start()
t1.join()
t2.join()
print("First set of threads done")
print("The program can execute other code here")
t3.start()
t4.start()
t5.start()
t3.join()
t4.join()
t5.join()
print("Second set of threads done")
print("END Program")
```

Запустив вновь измененный код, мы получим следующий результат:

```
start Thread 1
start Thread 2
end Thread 2
end Thread 1
First set of threads done
The program can execute other code here
start Thread 3
start Thread 4
start Thread 5
```

```
end Thread 3
end Thread 5
end Thread 4
Second set of threads done
END Program
```

Как можно видеть, с помощью методов `join()` можно добавить в программу место, где вы ожидаете начала выполнения потоков, а затем снова начинаете со следующих строк кода.

Типичный шаблон синхронизации потоков

В приведенных выше примерах при использовании нескольких потоков можно заметить, что одинаковые строки кода используются несколько раз. Каждый раз мы определяли для каждого потока вызов методов `start()` и `join()`, записывая множество похожих строк. По мере увеличения количества потоков ситуация только ухудшается. Однако есть и другие, более удобные способы написания кода.

В нашем примере с 5 потоками мы можем написать код следующим образом:

```
import threading
import time

def function(i):
  print ("start Thread %i" %i)
  time.sleep(2)
  print ("end Thread %i" %i)
  return

n_threads = 5
threads = [ ]
for i in range(n_threads):
  t = threading.Thread(target=function , args=(i,))
  threads.append(t)
  t.start()

for i in range(n_threads):
  threads[i].join()
```

В этом виде код стал гораздо более читабельным и простым. Используя цикл for, который повторяется для любого желаемого количества потоков, мы избежали необходимости определять каждый поток отдельно (`t1`, `t2`, `t3`, ...) и вызывать методы `start()` и `join()` для каждого из них.

При выполнении мы получим результат, идентичный предыдущим примерам:

```
start Thread 0
start Thread 1
```

```
start Thread 2
start Thread 3
start Thread 4
end Thread 0
end Thread 1
end Thread 2
end Thread 3
end Thread 4
```

Обратите внимание: постарайтесь не использовать следующий код, если только это не делается намеренно:

```
for i in range(n_threads):
  t = threading.Thread(target=function , args=(i,))
  threads.append(t)
  t.start()
  t.join()
```

Если поместить вызов метода `join()` в тот же цикл `for`, то мы не получим предыдущий случай, когда все потоки запускаются одновременно. Но в этом случае будет всего лишь последовательная обработка потоков. При этом сначала запускается первый поток, и только после его завершения запускается второй поток и т. д.

Модуль concurrent.futures и класс ThreadPoolExecutor

Помимо модуля `threading`, в стандартной библиотеке есть еще один модуль, который является полезным инструментом для работы с потоками, а в следующей главе мы увидим, что он также подходит для процессов. Этот модуль называется `concurrent.futures`. Он представляет собой высокоуровневый интерфейс для асинхронного выполнения вызываемых объектов.

Внутри него находится класс `ThreadPoolExecutor`, который очень полезен для одновременного управления несколькими потоками. Когда в нашей программе необходимо управлять множеством потоков, наиболее эффективным способом сделать это является создание `ThreadPoolExecutor`.

Рассмотрим пример, в котором мы хотим запустить четыре потока одновременно, и все они связаны с одной или несколькими функциями. Вместо создания определения четырех экземпляров класса `Thread`, а затем четырех вызовов метода `start()` и четырех вызовов метода `join()` гораздо проще использовать `ThreadPoolExecutor`.

Примером этого является следующий код:

```
import concurrent.futures
import time
```

```
def thread(num,t):
  print("Thread %s started" %num)
  time.sleep(t)
  print("Thread %s ended" %num)

with concurrent.futures.ThreadPoolExecutor(max_workers=4) as t:
   t.submit(thread(1,10))
   t.submit(thread(2,1))
   t.submit(thread(3,10))
   t.submit(thread(4,4))

print("Program ended")
```

Запуск этого кода приводит к следующему результату.

```
.
Thread 1 started
Thread 1 ended
Thread 2 started
Thread 2 ended
Thread 3 started
Thread 3 ended
Thread 4 started
Thread 4 ended
Program ended
```

Потоки выполняются последовательно и, судя по всему, отдельно друг от друга. На основании этого очень аккуратного результата можно сделать вывод, что `ThreadPoolExecutor` предоставляет некую систему синхронизации внутри себя. Даже если внутри не должно быть ничего подобного. Если же мы напишем предыдущий код без использования `ThreadPoolExecutor`, то получим следующее:

```
import threading
import time

def thread(num,t):
  print("Thread %s started" %num) time.sleep(t)
  print("Thread %s ended" %num)

t1 = threading.Thread(target=thread, args=(1,10,))
t2 = threading.Thread(target=thread, args=(2,1,))
t3 = threading.Thread(target=thread, args=(3,10,))
t4 = threading.Thread(target=thread, args=(4,4,))
t1.start()
t2.start()
t3.start()
t4.start()
t1.join()
```

```
t2.join()
t3.join()
t4.join() print("Program ended")
```

При запуске этого кода получится совершенно иной результат:

```
Thread 1 started
Thread 2 started
Thread 3 started
Thread 4 started
Thread 2 ended
Thread 4 ended
Thread 1 ended
Thread 3 ended
Program ended
```

Здесь потоки запускаются одновременно и поочередно (в конкуренции). Поэтому время выполнения каждого исполняемого потока зависит от его продолжительности. В этом примере мы использовали несколько потоков, которые нацелены на одну и ту же функцию. Другой очень распространенный случай – когда каждому потоку назначено выполнение своей функции. Давайте посмотрим, как изменяется синтаксис в таком случае.

Конкуренция потоков

Хорошим и простым примером для демонстрации поведения двух конкурирующих потоков (параллельное программирование) является следующий код. Каждому потоку назначена своя функция, `addA()` и `addB()`, которые имитируют определенную циклическую операцию, занимающую соответственно `timeA` и `timeB` при каждой итерации. Два потока запускаются одновременно, и поскольку в Python они не могут выполняться параллельно, а только по одному за раз, во время выполнения программы они будут конкурировать, чтобы как можно скорее завершить свой цикл (`COUNT` соответствует `5` итерациям). Чтобы отслеживать последовательность выполнения двух потоков, две функции в каждом цикле будут добавлять буквы `A` и `B`, соответствующие этим двум потокам, в строку символов:

```
import threading
import time

sequence = "" COUNT = 5
timeA = 5
timeB = 10


def addA():
  global sequence
  for i in range(COUNT):
```

```
    time.sleep(timeA)
    sequence = "%sA" %sequence
    print("Sequence: %s" %sequence)

def addB():
  global sequence
  for i in range(COUNT):
    time.sleep(timeB)
    sequence = "%sB" %sequence
    print("Sequence: %s" %sequence)

# Основная программа
t1 = threading.Thread(target = addA)
t2 = threading.Thread(target = addB)
t1.start()
t2.start()
t1.join()
t2.join()
```

Во время выполнения программы вы сможете увидеть последовательность выполнения. Итак, если вы запустите только что написанный код, вы получите последовательность, похожую на следующую:

```
Sequence: A
Sequence: AA
Sequence: AAB
Sequence: AABA
Sequence: AABAA
Sequence: AABAAB
Sequence: AABAABA
Sequence: AABAABAB
Sequence: AABAABABB
Sequence: AABAABABBB
```

Как видно из результата, два потока чередуются друг с другом во время выполнения произвольным образом. Вы увидите, что последовательность меняется в зависимости от выполняемых операций. Вы также можете поразвлечься, изменяя время выполнения каждого потока посредством изменения значений переменных `timeA` и `timeB`. Это, в свою очередь, повлияет на последовательность выполнения двух потоков, которые будут конкурировать друг с другом.

Использование подклассов Thread

В предыдущем примере мы использовали определение потока через конструктор `Thread()`, которому имя функции передается в качестве аргумента через параметр `target`:

```
t = threading.Thread(target = function_name)
```

В данном случае мы разделили потоки, определенные кодом внутри функций, на два отдельных объекта.

Другой способ реализации кода заключается в определении нового потока в качестве подкласса Thread с собственными методами и, следовательно, собственным кодом для выполнения внутри него, без дополнительных вызовов внешних функций. В этом случае потоки являются реальными объектами, что соответствует принципам объектно ориентированного программирования.

Сначала давайте напрямую импортируем класс Thread из модуля threading:

```
from threading import Thread
```

Теперь мы можем определить новый подкласс класса Thread и переопределить метод __init __ (self [, args]), добавив любые другие определения во время инициализации подкласса:

```
class myThread(Thread):
  def __init__ (self):
    Thread.__init__(self)
    #добавьте здесь свой код
```

Затем мы переопределим метод run(self [, args]) для реализации того, что поток должен будет выполнить при запуске с помощью вызова start():

```
  def run(self):
    #код здесь
```

Затем создается экземпляр только что определенного нового класса myThread и вызывается конструктор с любыми аргументами, если они есть. Наконец, поток запускается путем вызова метода start(), который выполнит код, реализованный в методе run():

```
t = myThread()
t.start()
```

Давайте вернемся к предыдущему примеру, где два разных метода передаются двум экземплярам одного и того же класса Thread с помощью опции target. Исходя из новых соображений, две функции можно заменить двумя отдельными подклассами. Каждый из них будет иметь соответствующий код функции в методе run().

Таким образом, предыдущий код можно преобразовать следующим образом:

```
from threading import Thread
import time

sequence = "" COUNT = 5
timeA = 5
timeB = 10
```

```
class ThreadA(Thread):
  def __init__ (self):
    Thread.__init__(self)

  def run(self):
    global sequence
    for i in range(COUNT):
      time.sleep(timeA)
      sequence = "%sA" %sequence
      print("Sequence: %s" %sequence)

class ThreadB(Thread):
  def __init__ (self):
    Thread.__init__(self)

  def run(self): global sequence
    for i in range(COUNT):
      time.sleep(timeB)
      sequence = "%sB" %sequence
      print("Sequence: %s" %sequence)

# Главная программа
t1 = ThreadA()
t2 = ThreadB()
t1.start()
t2.start()
t1.join()
t2.join()
```

Механизмы синхронизации

Как было показано в предыдущем разделе, потоки выполняются на конкурентной основе и, таким образом, работают параллельно (но не одновременно). В результате могут возникать непредсказуемые ситуации, которые, если их не контролировать, могут привести к проблемам из-за конфликта доступа, особенно в случае конкуренции за доступ к общему ресурсу. Для решения этой проблемы модуль `threading` предоставляет несколько классов, которые можно использовать для реализации механизмов синхронизации потоков. Существуют различные типы, каждый со своими особенностями. На протяжении этой главы мы рассмотрим их все, и каждый из них будет реализован в простом примере, полезном для лучшего понимания их работы.

Объекты синхронизации, предоставляемые модулем `threading`:

- Lock,
- RLock,
- Semaphore,

- Condition,
- Event.

Класс Lock

Lock – это класс с самым низким уровнем синхронизации среди всех классов, предоставляемых модулем `threading`. Класс Lock может принимать два состояния:

- Locked,
- Unlocked.

Он имеет два метода:

- `acquire()`,
- `release()`,

которые служат для переключения статуса блокировки между заблокированным (locked) и разблокированным (unlocked). При определении блокировки с помощью конструктора `Lock()`он находится в разблокированном состоянии. Когда поток вызывает метод `lock.acquire()`, блокировка переключается в состояние locked и блокирует выполнение потока, который переходит в состояние ожидания. Когда другой поток вызывает метод `lock.release()`, блокировка возвращается в состояние unlocked, и ожидающий поток возобновляет свое выполнение (см. рис. 2.1).

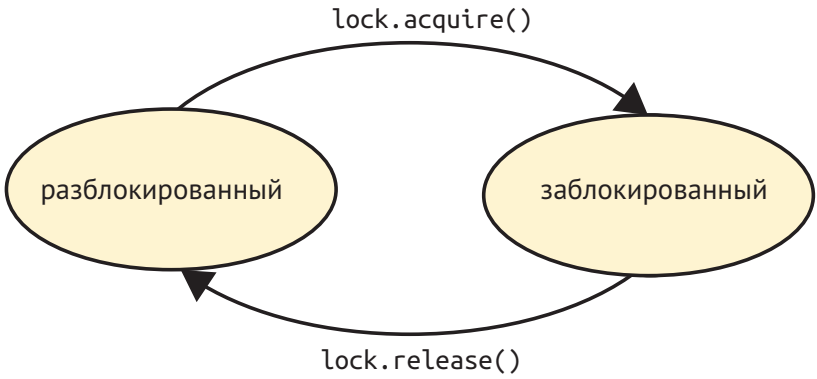


Рис. 2.1. Состояния класса Lock

Если этот механизм синхронизации не управляется должным образом, может возникнуть еще большая путаница в синхронизации, чем при отказе от его использования. В действительности может быть несколько потоков, которые вызвали метод `lock.acquire()`, и все они ожидают, пока хотя бы один из других потоков вызовет `lock.release()`, чтобы изменить статус блокировки с заблокированного на разблокированный. В этом случае невозможно предсказать, какой именно из ожидающих потоков возобновит выполнение, и результат может варьироваться в зависимости от реализации.

Для иллюстрации принципа работы блокировок рассмотрим пример с двумя потоками, каждый из которых выполняет свою функцию. Мы назовем эти две функции `funcA()` и `funcB()`. В первой главе мы видели, что конкурирующие потоки используют общую память процесса. Поэтому в нашем примере мы просто используем переменную `shared`, которая содержит целое число, доступное для обоих потоков. Первый поток, связанный с `funcA()`, увеличит это значение на `10`, а другой поток, связанный с `funcB()`, уменьшит значение на `10`. Обе функции выполнят эту операцию `10` раз.

Итак, напишем следующий код:

```
import threading
import time

shared_data = 0

def funcA():
  global shared_data
  for i in range(10):
    local = shared_data
    local += 10
    time.sleep(1)
    shared_data = local
    print("Thread A wrote: %s" %shared_data)

def funcB():
  global shared_data
  for i in range(10):
    local = shared_data
    local -= 10
    time.sleep(1)
    shared_data = local
    print("Thread B wrote: %s" %shared_data)

t1 = threading.Thread(target = funcA)
t2 = threading.Thread(target = funcB)
t1.start()
t2.start()
t1.join()
t2.join()
```

Это может служить отличным примером конкуренции между потоками. Запускаем код и получаем следующий результат:

```
Thread A wrote: 10
Thread B wrote: -10
Thread A wrote: 20
Thread B wrote: -20
Thread B wrote: -30
```

```
Thread A wrote: 30
Thread A wrote: 40
Thread B wrote: 20
Thread A wrote: 50
Thread B wrote: 10
Thread A wrote: 60
Thread B wrote: 0
Thread A wrote: 70
Thread B wrote: -10
Thread B wrote: -20
Thread A wrote: 80
Thread B wrote: 70
Thread A wrote: 90
Thread B wrote: 60
Thread A wrote: 100
```

Для начала итерации циклов `for` внутри двух функций разделены и выполняются отдельно. В этом случае потоки осуществляют атомарное выполнение каждого цикла `for`, и каждый из этих циклов конкурирует между двумя функциями. Таким образом, на каждом этапе выполнения один из циклов for функции `funcA()` или `funcB()` будет преобладать над другим, и будет выполняться первым (в Python потоки не могут выполняться параллельно). Однако это именно то поведение, которое мы ожидали от нескольких потоков, выполняющихся одновременно. Пока все правильно.

Проблема заключается в значении переменной `shared`, используемой на различных этапах. Значение в конце выполнения должно быть `0`, но в данном случае оно равно `100` (хотя это совершенно случайное значение, которое отличается от выполнения к выполнению). Таким образом, мы имеем дело со случаем конкурентного доступа к данным. Кроме того, из полученных данных видно, что это произошло не один раз, а происходит очень часто даже за 10 циклов и при использовании всего 2 потоков.

Поэтому совершенно очевидно, что для правильной работы этой программы необходимо использовать механизм синхронизации, который координирует доступ двух потоков к переменной `shared`, чтобы избежать конкурентных условий. Класс `Lock`, предоставляемый модулем `threading`, является самым простым вариантом.

Далее мы определяем экземпляр класса `Lock` в начале программы и вставляем вызовы методов `acquire()` и `release()` в два потока, как показано в следующем коде:

```
import threading
import time

shared = 0
lock = threading.Lock()

def funcA():
```

```
    global shared
    for i in range(10):
      lock.acquire()
      local = shared
      local += 10
      time.sleep(1)
      shared = local
      print("Thread A wrote: %s" %shared)
      lock.release()

def funcB():
    global shared
    for i in range(10):
      lock.acquire()
      local = shared
      local -= 10
      time.sleep(1)
      shared = local
      print("Thread B wrote: %s" %shared)
      lock.release()

t1 = threading.Thread(target = funcA)
t2 = threading.Thread(target = funcB)
t1.start()
t2.start()
t1.join()
t2.join()
```

В этот раз запуск кода даст совершенно другой результат:

```
Thread A wrote: 10
Thread A wrote: 20
Thread A wrote: 30
Thread A wrote: 40
Thread A wrote: 50
Thread A wrote: 60
Thread A wrote: 70
Thread A wrote: 80
Thread A wrote: 90
Thread A wrote: 100
Thread B wrote: 90
Thread B wrote: 80
Thread B wrote: 70
Thread B wrote: 60
Thread B wrote: 50
Thread B wrote: 40
Thread B wrote: 30
```

```
Thread B wrote: 20
Thread B wrote: 10
Thread B wrote: 0
```

Как можно видеть, проблем с конкурентными условиями больше нет. Переменная `shared` считывается и изменяется синхронно только одним потоком за раз, что предотвращает появление неверных значений. Однако следует отметить, что с введением этого механизма синхронизации потоки, казалось бы, утратили свою одновременность. На самом деле потоки демонстрируют одновременность только в пределах синхронизированного кода. По результатам отчетливо видно, что сначала полностью выполняется функция `funcA()` со всеми ее `10` циклами `for`, которая обращается только к переменной `shared` и приводит ее значение к `100`. Затем полностью выполняется поток `funcB()`, который возвращает значение `shared` к `0`.

Протокол управления контекстом с использованием Lock

Все объекты в модуле потоков, которые используют методы `acquire()` и `release()`, такие как объекты `Lock`, могут использоваться в контекстном менеджере с помощью оператора `with` (см. примечание ниже).

Обратите внимание: в языке Python оператор `with` создает контекст выполнения, который позволяет запускать блок операторов под управлением контекстного менеджера (Context Manager).

```
with expression:
    #код
```

Контекстный менеджер выполняет оценку выражения применительно к блоку кода, связанному с with (контекст). Поэтому выражение должно возвращать объект, который реализует протокол управления контекстом и который, по сути, состоит из двух методов:

- `__enter__()`, который вызывается при входе в контекст;
- `__exit__()`, который вызывается при выходе из контекста.

В дополнение к этому у оператора `with` есть еще одно преимущество: он включает в себя функциональность конструкции `try ... finally`.

Таким образом, вы получаете более понятный и удобный для повторного использования код. Благодаря этим многочисленным преимуществам многие классы стандартной библиотеки допускают использование операторов `with` в качестве альтернативы традиционным конструкциям.

В случае блокировок при входе в блок вызывается метод `acquire()`, а при выходе – метод `release()`.

Таким образом, форма

```
lock.acquire() try:
#код
finally:
        lock.release()
```

может быть записана следующим образом:

```
with lock:
        #code
```

Затем мы переписываем предыдущий код с использованием поддерживаемого протокола контекстного менеджера:

```
import threading
import time

shared_data = 0
lock = threading.Lock()

def funcA():
  global shared_data
  for i in range(10):
    with lock:
      local = shared_data
      local += 10
      time.sleep(1)
      shared_data = local
      print("Thread A wrote: %s" %shared_data)

def funcB():
  global shared_data
  for i in range(10):
    with lock:
      local = shared_data
      local -= 10
      time.sleep(1)
      shared_data = local
      print("Thread B wrote: %s" %shared_data)

t1 = threading.Thread(target = funcA)
t2 = threading.Thread(target = funcB)
t1.start()
t2.start()
t1.join()
t2.join()
```

Как можно видеть, код стал гораздо более читабельным. При запуске кода мы не заметим никаких отличий в его поведении по сравнению с предыдущим кодом.

Протокол Context Manager с использованием оператора `with` также поддерживается другими объектами модуля `threading`, которые мы рассмотрим позже в этой главе:

- RLock,
- Condition,
- Semaphore.

Все эти объекты, как и `Lock`, используют методы `acquire()` и `release()` в своем механизме синхронизации.

Другой возможный вариант синхронизации с Lock

Продолжим анализ предыдущего кода. Как уже было сказано, мы добавили механизм синхронизации, который, похоже, полностью (или почти полностью) отменяет конкурентное поведение двух потоков.

Созданный нами механизм является наиболее интуитивным: мы добавили вызовы методов `acquire()` и `release()` попарно в каждом потоке, чтобы разграничить части блоков. Все было довольно понятно при использовании контекстных менеджеров с оператором `with`: у нас есть две симметричные части кода в обоих потоках. Но мы не обязаны действовать именно так. Вы можете попробовать найти более сложные состояния синхронизации. На свой страх и риск. В действительности можно вставлять вызовы методов `acquire()` и `release()` асимметрично, в тех местах кода, которые не являются симметричными друг относительно друга, и иногда использовать `acquire()` в одном потоке, а `release()` – в другом. Таким образом, контекстный менеджер теряет возможность идентифицировать блоки кода, и синхронизация становится гораздо более сложной. В таких случаях могут возникнуть не только проблемы с конкурентными условиями, но даже тупиковые ситуации. Кроме того, метод `release()` выдает ошибку выполнения, если он вызывается при разблокированном состоянии блокировки. Однако не отчаивайтесь, попробуйте провести несколько тестов, и, возможно, вы обнаружите, что синхронизированное решение с конкурентным выполнением все же возможно.

Например, если изменить код следующим образом:

```
import threading
import time

shared = 0
lock = threading.Lock()

def funcA():
  global shared
```

```
    for i in range(10):
      time.sleep(1)
      shared += 10
      print("Thread A wrote: %s" %shared)
      lock.acquire()

  def funcB():
    global shared
    lock.acquire()
    for i in range(10):
      time.sleep(1)
      shared -= 10
      print("Thread B wrote: %s" %shared)
      lock.release()

  t1 = threading.Thread(target = funcA)
  t2 = threading.Thread(target = funcB)
  t1.start()
  t2.start()
  t1.join()
  t2.join()
```

Изменений достаточно много. Например, функция `funcA()` больше не вызывает метод `release()`, а метод `acquire()` вызывается в конце каждой итерации. В функции `funcB()`, напротив, метод `acquire()` вызывается в начале выполнения, вне цикла `for`. Кроме того, функции больше не используют локальные переменные, а обе напрямую изменяют переменную `shared`.

Запустив измененный код, мы получим следующий результат:

```
Thread A wrote: 10
Thread B wrote: 0
Thread B wrote: -10
Thread A wrote: 0
Thread B wrote: -10
Thread A wrote: 0
Thread A wrote: 10
Thread B wrote: 0
Thread B wrote: -10
Thread A wrote: 0
Thread B wrote: -10
Thread A wrote: 0
Thread A wrote: 10
Thread B wrote: 0
Thread B wrote: -10
Thread A wrote: 0
Thread A wrote: 10
Thread B wrote: 0
```

```
Thread B wrote: -10
Thread A wrote: 0
```

Как можно видеть, на этот раз все, похоже, работает нормально. В конце концов мы получили итоговое общее значение `0` и восстановили возможность одновременного выполнения двух потоков. В действительности последовательность итераций между циклами `for` в двух потоках снова стала случайной и конкурентной. Запустив программу несколько раз, можно заметить, что поведение остается корректным, хотя даже в таких случаях абсолютной уверенности не будет.

Если вы не уверены, что два потока продвигаются поочередно, при этом один из них остается заблокированным в своем выполнении, а другой продолжает свое выполнение, вы можете (на этапе отладки) добавить в предыдущий код некоторые выводимые значения для подтверждения хода работы обоих потоков. В нашем случае можно добавить номер итерации к строкам, отображающим результаты подсчета для каждого потока. В этом случае мы сможем убедиться в ходе выполнения двух потоков одновременно:

```
import threading
import time

shared = 0
lock = threading.Lock()

def funcA():
  global shared
  for i in range(10):
    time.sleep(1)
    shared += 10
    print("Thread A wrote: %s, %i" %(shared,i))
    lock.acquire()

def funcB():
  global shared
  lock.acquire()
  for i in range(10):
    time.sleep(1)
    shared -= 10
    print("Thread B wrote: %s, %i" %(shared,i))
    lock.release()

t1 = threading.Thread(target = funcA)
t2 = threading.Thread(target = funcB)
t1.start()
t2.start()
t1.join()
t2.join()
```

Запустив код сразу после внесения изменений, вы увидите, что оба потока завершают свое выполнение, чередуясь между собой:

```
Thread A wrote: 10, 0
Thread B wrote: 0, 0
Thread B wrote: -10, 1
Thread A wrote: 0, 1
Thread B wrote: -10, 2
Thread A wrote: 0, 2
Thread B wrote: -10, 3
Thread A wrote: 0, 3
Thread B wrote: -10, 4
Thread A wrote: 0, 4
Thread B wrote: -10, 5
Thread A wrote: 0, 5
Thread B wrote: -10, 6
Thread A wrote: 0, 6
Thread B wrote: -10, 7
Thread A wrote: 0, 7
Thread B wrote: -10, 8
Thread A wrote: 0, 8
Thread B wrote: -10, 9
Thread A wrote: 0, 9
```

Механизм Lock в этом случае продолжает работу, даже если он выполняется в коде асимметрично и без уверенности в том, что все операции будут выполняться именно таким образом.

RLock

Еще один класс, используемый для синхронизации потоков, называется RLock. Он представляет собой **блокировку с повторным запуском** (reentrant lock). Этот класс очень похож на класс Lock, но, в отличие от него, может быть использован несколько раз одним и тем же потоком. Внутри него, помимо состояния «заблокировано-разблокировано», также хранится информация о владельце потока и уровне рекурсии.

Как и Lock, класс RLock можно привязать к потоку с помощью метода `acquire()`. В этот момент RLock блокируется, а вызывающий поток становится одним из его владельцев. Таким же образом RLock можно разблокировать с помощью метода `release()`. Однако, в отличие от механизма синхронизации Lock, в данном случае вызовы методов `acquire()` и `release()` могут быть множественными и вложенными друг в друга. Другие потоки, которые вызовут метод `acquire()`, будут добавлены в список владельцев. Только окончательный вызов `release()` сможет разблокировать RLock и обеспечить возможность возобновления работы другого потока.

Итак, перейдем к примеру. Давайте используем три потока с функцией, в которой есть два вложенных цикла for с доступом к переменной `shared` на обоих

уровнях. Кроме того, мы сможем различить три потока, изменяя время выполнения каждого из них, чтобы еще больше акцентировать внимание на их конкурентном поведении:

```
import threading
import time

shared = 0
rlock = threading.RLock()

def func(name, t):
  global shared
  for i in range(3):
    rlock.acquire()
    local = shared
    time.sleep(t)
    for j in range(2):
      rlock.acquire()
      local += 1
      time.sleep(2)
      shared = local
      print("Thread %s-%s wrote: %s" %(name, j, shared))
      rlock.release()
    shared = local + 1
    print("Thread %s wrote: %s" %(name, shared))
    rlock.release()

t1 = threading.Thread(target = func,args=('A',2,))
t2 = threading.Thread(target = func,args=('B',10,))
t3 = threading.Thread(target = func,args=('C',1,))
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

Запустив приведенный выше код, мы получим следующий результат:

```
Thread A-0 wrote: 1
Thread A-1 wrote: 2
Thread A wrote: 3
Thread A-0 wrote: 4
Thread A-1 wrote: 5
Thread A wrote: 6
Thread A-0 wrote: 7
Thread A-1 wrote: 8
Thread A wrote: 9
```

```
Thread B-0 wrote: 10
Thread B-1 wrote: 11
Thread B wrote: 12
Thread B-0 wrote: 13
Thread B-1 wrote: 14
Thread B wrote: 15
Thread B-0 wrote: 16
Thread B-1 wrote: 17
Thread B wrote: 18
Thread C-0 wrote: 19
Thread C-1 wrote: 20
Thread C wrote: 21
Thread C-0 wrote: 22
Thread C-1 wrote: 23
Thread C wrote: 24
Thread C-0 wrote: 25
Thread C-1 wrote: 26
Thread C wrote: 27
```

Кроме того, в этом случае, как и в случае с Lock, синхронизация привела к идеальному управлению переменной `shared`, но к потере одновременного выполнения потоков. Без использования механизма синхронизации RLock (то есть без вызовов методов `acquire()` и `release()`) программа дала бы результат, подобный следующему:

```
Thread C-0 wrote: 1
Thread A-0 wrote: 1
Thread C-1 wrote: 2
Thread C wrote: 3
Thread A-1 wrote: 2
Thread A wrote: 3
Thread C-0 wrote: 4
Thread C-1 wrote: 5
Thread C wrote: 6
Thread A-0 wrote: 4
Thread B-0 wrote: 1
Thread A-1 wrote: 5
Thread A wrote: 6
Thread C-0 wrote: 7
Thread B-1 wrote: 2
Thread B wrote: 3
Thread C-1 wrote: 8
Thread C wrote: 9
Thread A-0 wrote: 7
Thread A-1 wrote: 8
Thread A wrote: 9
Thread B-0 wrote: 4
```

```
Thread B-1 wrote: 5
Thread B wrote: 6
Thread B-0 wrote: 7
Thread B-1 wrote: 8
Thread B wrote: 9
```

Как видно из результата, возникает конкурентная ситуация, в результате которой переменная `shared` получает неверные значения.

Semaphore

Еще один механизм синхронизации, доступный в модуле `threading`, основан на принципе семафора (semaphore). Этот примитив является старейшей формой синхронизации в истории компьютерных наук, изобретенной Эдсгером В. Дейкстрой (Edsger W. Dijkstra) еще в 1962 году.

С его помощью осуществляется синхронное управление использованием общих ресурсов несколькими потоками в рамках одного процесса. Для этого каждый семафор связывается с общим ресурсом, предоставляя доступ всем потокам до тех пор, пока значение его внутреннего счетчика не станет отрицательным.

Семафор – это объект, который, подобно Lock, работает через вызовы методов `acquire()` и `release()`. Внутри него находится счетчик, который уменьшается на единицу при каждом вызове `acquire()` и увеличивается на единицу при каждом вызове `release()`.

Следовательно, если поток должен получить доступ к общему ресурсу, защищенному семафором, н сначала ему необходимо вызвать метод `acquire()`. Внутренний счетчик семафора уменьшается на единицу. Если значение больше или равно нулю, поток получает доступ к ресурсу, в противном случае он блокируется и ожидает, пока другой поток не вызовет `release()` для того же ресурса. Только в этот момент поток сможет продолжить свое выполнение, получив доступ к необходимому ресурсу.

Поэтому очень важно, чтобы для каждого потока, вызвавшего `acquire()`, в конце операций с общим ресурсом был вызов метода `release()`, чтобы другие потоки также могли получить доступ к ресурсам, избегая таким образом любых тупиковых ситуаций.

Для синхронизации с помощью семафора в примере кода мы будем использовать модель «производитель–потребитель» (Producer-Consumer), см. рис. 2.2. Эта модель программирования основана на двух типах объектов, которые воздействуют на поток данных. **Производитель** генерирует данные, обычно получая их из внешнего ресурса, а **потребитель** использует данные от производителя. Проблема заключается в том, что эти два объекта работают независимо друг от друга, с разной и переменной скоростью. Их количество также может варьироваться. Например, может быть только один производитель и несколько потребителей, или наоборот. Эта модель очень хорошо подходит для потоков (как и для процессов), поэтому в этих примерах ее использование вполне оправдано.



Рис. 2.2. Модель «производитель–потребитель»

В качестве примера синхронизации с использованием семафора мы определим два подкласса `Thread`: `Consumer` и `Producer`. В их методах `run()` мы реализуем их код для выполнения. В случае `Producer` мы реализуем функцию `request()`, которая имитирует запрос данных из внешнего источника с определенным временем, затрачиваемым посредством `time.sleep()`.

Итак, давайте напишем следующий код.

```
from threading import Thread, Semaphore
import time
import random

semaphore = Semaphore(1)
shared = 1

class Consumer(Thread):
  def __init__ (self):
    Thread.__init__(self)
    global semaphore

  def run(self):
    global shared
    semaphore.acquire()
    print("consumer has used this: %s" %shared)
    shared = 0
    semaphore.release()

class Producer(Thread):
  def __init__ (self):
    Thread.__init__(self)
    global semaphore
    def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    global shared
    semaphore.acquire()
    shared = self.request()
    print("producer has loaded this: %s" %shared)
```

```
        semaphore.release()

t1 = Producer()
t2 = Consumer()
t1.start()
t2.start()
t1.join()
t2.join()
```

При запуске этого кода получится примерно такой результат:

```
producer has loaded this: 60
consumer has used this: 60
```

Поскольку механизм синхронизации семафоров использует методы `acquire()` и `release()`, то семафоры также поддерживают протокол управления контекстом. Таким образом, мы можем записать приведенный выше код следующим образом:

```
from threading import Thread, Semaphore
import time
import random

semaphore = Semaphore(1)
shared = 1

class Consumer(Thread):
  def __init__ (self):
    Thread.__init__(self)
    global semaphore

  def run(self):
    global shared
    with semaphore:
      print("consumer has used this: %s" %shared)
      shared = 0

class Producer(Thread):
  def __init__ (self):
    Thread.__init__(self)
    global semaphore

  def request(self):
    time.sleep(1)
    return random.randint(0,100)


  def run(self):
```

```
      global shared
      with semaphore:
        shared = self.request()
        print("producer has loaded this: %s" %shared)

t1 = Producer()
t2 = Consumer()
t1.start()
t2.start()
t1.join()
t2.join()
```

Запуск этого кода приводит к точно такому же результату.

```
producer has loaded this: 2
consumer has used this: 2
```

В данном случае у нас есть два потока, `Producer` и `Consumer`, которые осуществляют одну операцию, то есть генерируют и потребляют одно значение. Но что произойдет, если мы заставим поток Producer генерировать больше значений (например, 5), а поток Consumer будет потреблять столько же?

Мы переписываем код таким образом, чтобы каждый поток выполнял описанную выше операцию пять раз:

```
from threading import Thread, Semaphore
import time
import random

semaphore = Semaphore(1)
shared = 1
count = 5

class consumer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    global semaphore
    self.count = count

  def run(self):
    global shared
    for i in range(self.count):
      semaphore.acquire()
      print("consumer has used this: %s" %shared)
      shared = 0
      semaphore.release()

class producer(Thread):
  def __init__(self, count):
```

```
    Thread.__init__(self)
    self.count = count
    global semaphore

  def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    global shared
    for i in range(self.count):
      semaphore.acquire()
      shared = self.request()
      print("producer has loaded this: %s" %shared)
      semaphore.release()

t1 = producer(count)
t2 = consumer(count)
t1.start()
t2.start()
t1.join()
t2.join()
```

Запуск этого кода дает следующий результат:

```
producer has loaded this: 59
producer has loaded this: 85
producer has loaded this: 20
producer has loaded this: 4
producer has loaded this: 7
consumer has used this: 7
consumer has used this: 0
consumer has used this: 0
consumer has used this: 0
consumer has used this: 0
```

Это определенно не то поведение, на которое мы рассчитываем. В действительности поток `Producer` продолжает генерировать значения, которые перезаписывают общий ресурс, но потребитель блокируется и не запускается до тех пор, пока производитель не завершит свою работу (пять циклов). Поток `Consumer`, запускающийся после завершения пяти циклов потока `Producer`, потребляет только последнее сгенерированное значение и теряет четыре предыдущих.

Кроме того, в этом случае, как и в примере с блокировкой, атомарное управление блоком кода потока с помощью семафора больше не подходит для наших нужд. Под атомарностью (atomic) мы подразумеваем следующее:

```
semaphore.acquire()
#код
semaphore.release()
или даже так:
with semaphore:
    #код
```

Итак, давайте разделим механизм вызова `acquire()` и `release()` между двумя потоками. Поток `Producer`, который будет первым получать доступ к общему ресурсу, вызовет метод `acquire()` перед перезаписью данных. Поток `Consumer` после использования общего ресурса вызовет метод `release()` для высвобождения ресурса:

```
from threading import Thread, Semaphore
import time
import random

semaphore = Semaphore(1)
shared = 1
count = 5

def request():
  time.sleep(1)
  return random.randint(0,100)

class consumer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    global semaphore
    self.count = count

  def run(self):
    global shared
    for i in range(self.count):
      semaphore.acquire()
      print("consumer has used this: %s" %shared)
      shared = 0

class producer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    self.count = count
    global semaphore

  def run(self):
    global shared
    for i in range(self.count):
```

```
        shared = request()
        print("producer has loaded this: %s" %shared)
        semaphore.release()

t1 = producer(count)
t2 = consumer(count)
t1.start()
t2.start()
t1.join()
t2.join()
```

Выполнение кода дает следующий результат:

```
consumer has used this: 1
producer has loaded this: 0
consumer has used this: 0
producer has loaded this: 47
consumer has used this: 47
producer has loaded this: 25
consumer has used this: 25
producer has loaded this: 82
consumer has used this: 82
producer has loaded this: 23
```

Как видно из результата, синхронизация все еще не идеальна. Мы наблюдаем чередование действий между потоком `Consumer` и потоком `Producer`. За исключением того, что первым к общему ресурсу обращается `Consumer`, и поэтому первым потребляемым значением является начальное значение по умолчанию `1`, в то время как значение, произведенное `Producer` в конце программы, не потребляется. Чтобы решить эту проблему и правильно настроить порядок выполнения, достаточно установить начальное внутреннее значение `semaphore` равным `0` вместо `1`:

```
semaphore = Semaphore(0)
```

При повторном запуске программы в этот раз мы получим правильное поведение:

```
producer has loaded this: 25
consumer has used this: 25
producer has loaded this: 45
consumer has used this: 45
producer has loaded this: 60
consumer has used this: 60
producer has loaded this: 9
consumer has used this: 9
producer has loaded this: 4
consumer has used this: 4
```

В результате достигается идеальная синхронизация. Это объясняется тем, что в нашем случае атомарная операция, подлежащая синхронизации, представляет собой операцию производства-потребления одного цикла, в которой одновременно участвуют два потока:

```
Phase 1 synchronized: Producer (1 ciclo for)+ Consumer (1 ciclo for)
Phase 2 synchronized: Producer (1 ciclo for)+ Consumer (1 ciclo for)
Phase 3 synchronized: Producer (1 ciclo for)+ Consumer (1 ciclo for)
Phase 4 synchronized: Producer (1 ciclo for)+ Consumer (1 ciclo for)
Phase 5 synchronized: Producer (1 ciclo for)+ Consumer (1 ciclo for)
```

Тогда как в предыдущем примере схема синхронизации была неверной:

```
Phase 1 synchronized: Producer ( 5 cicli for)
Phase 2 synchronized: Consumer ( 5 cicli for)
```

Из всего этого хорошо видно, что при вставке механизма синхронизации в код очень важно сначала разобраться в том, как работают задействованные потоки. Затем из нашего представления о правильном выполнении извлечь атомарную фазу, которая будет повторяться и синхронизироваться и которая, как мы видели, ограничена блоком кода, присутствующим в одном потоке, но может включать несколько потоков одновременно. Затем необходимо применить правильный механизм синхронизации.

В случае, если в синхронизированной фазе участвует больше потоков, мы должны обойтись без протокола контекстного менеджера с оператором `with`.

Класс Condition

Альтернативой **Semaphore** при синхронизации потоков является класс `Condition`. У класса `Condition` есть внутренняя блокировка (Lock), которая с помощью `acquire()` и `release()` обеспечивает переход из заблокированного состояния в разблокированное. Помимо этого, у него есть и другие связанные методы. Метод `wait()` освобождает блокировку, но блокирует поток до тех пор, пока другой поток не вызовет методы `notify()` и `notify_all()`.

Метод `notify()` активирует только один из потоков, ожидающих переменную условия, если таковая имеется. Метод `notify_all()` активирует все ожидающие потоки.

Вернемся к предыдущему коду, использовавшему семафор, и на этот раз воспользуемся переменной `Condition` в качестве системы синхронизации потоков. Затем мы изменяем ранее созданный код и в итоге получаем следующие строки:

```
from threading import Thread, Condition
import time
import random

condition = Condition()
shared = 1
```

```
count = 5

class Consumer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    global condition
    self.count = count

  def run(self):
    global shared
    for i in range(self.count):
      condition.acquire()
      if shared == 0:
        condition.wait()
      print("consumer has used this: %s" %shared)
      shared = 0
      condition.notify()
      condition.release()

class Producer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    self.count = count
    global condition

  def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    global shared
    for i in range(self.count):
      condition.acquire()
      shared = self.request()
      print("producer has loaded this: %s" %shared)
      condition.wait()
      if shared == 0:
        condition.notify()
      condition.release()

t1 = Producer(count)
t2 = Consumer(count)
t1.start()
t2.start()
t1.join()
t2.join()
```

При запуске этого кода получается следующий результат:

```
producer has loaded this: 43
consumer has used this: 43
producer has loaded this: 98
consumer has used this: 98
producer has loaded this: 51
consumer has used this: 51
producer has loaded this: 57
consumer has used this: 57
producer has loaded this: 40
consumer has used this: 40
```

Кроме того, в этом случае система синхронизации работает так же, как и система Semaphore.

Объект Event

Помимо `Semaphore` и `Condition`, существует еще один механизм синхронизации. Использование `Event` в концептуальном плане является одним из самых простых. Все это можно рассматривать как базовый механизм коммуникации между потоками, в котором один поток сигнализирует другому потоку, ожидающему его наступления, об определенном событии.

Объект `Event` управляет внутренним булевым флагом. Существует также два метода, которые определяют его значение. Метод `set()` устанавливает значение флага в True (истина), а метод `clear()` – в False (ложь). False – это значение объекта `Event` по умолчанию при его создании. Существует также третий метод `wait()`, который блокирует поток до тех пор, пока флаг не примет значение True.

Другими словами, поток во время своего выполнения «зависает», вызывая метод `wait()` в ожидании события, после которого он сможет продолжить работу. Когда это событие происходит в другом потоке, вызывается метод `set()`, который разблокирует предыдущий поток, выполняющий свои операции, а затем вызывает `clear()`, чтобы сбросить все.

В нашем примере с двумя потоками `Producer` и `Consumer` мы можем заменить предыдущие методы синхронизации механизмом синхронизации на основе Event.

Изменим код предыдущих примеров следующим образом:

```
from threading import Thread, Event
import time
import random

event = Event()
shared = 1
count = 5
```

```
class Consumer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    global event
    self.count = count

  def run(self):
    global shared
    for i in range(self.count):
      event.wait()
      print("consumer has used this: %s" %shared)
      shared = 0
      event.clear()

class Producer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    self.count = count
    global event

  def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    global shared
    for i in range(self.count):
      shared = self.request()
      print("producer has loaded this: %s" %shared)
      event.set()

t1 = Producer(count)
t2 = Consumer(count)
t1.start()
t2.start()
t1.join()
t2.join()
```

После запуска этого кода получается следующий результат:

```
producer has loaded this: 40
consumer has used this: 40
producer has loaded this: 100
consumer has used this: 100
producer has loaded this: 15
consumer has used this: 15
producer has loaded this: 27
```

```
consumer has used this: 27
producer has loaded this: 94
consumer has used this: 94
```

Можно также заметить, что в данном случае синхронизация между двумя потоками идеальна.

Queue

Продолжим развитие предыдущего примера. До сих пор мы использовали только один поток `Producer` и один поток `Consumer`. Но что произойдет при увеличении их количества?

Изменим предыдущую программу и на этот раз запустим одновременно четыре потока: два `Producer` и два `Consumer`.

```
from threading import Thread, Event
import time
import random

event = Event()
shared = 1
count = 5

class Consumer(Thread):
  def __init__(self, count):
    Thread.__init__(self) global event
    self.count = count

  def run(self):
    global shared
    for i in range(self.count):
      event.wait()
      print("consumer has used this: %s" %shared)
      shared = 0
      event.clear()

class Producer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    self.count = count
    global event

  def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    global shared
```

```
        for i in range(self.count):
            shared = self.request()
            print("producer has loaded this: %s" %shared)
            event.set()

t1 = Producer(count)
t2 = Producer(count)
t3 = Consumer(count)
t4 = Consumer(count)
t1.start()
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()
```

При запуске этого кода получается следующий результат:

```
producer has loaded this: 92
consumer has used this: 92
consumer has used this: 0
producer has loaded this: 53
consumer has used this: 53
consumer has used this: 0
producer has loaded this: 62
consumer has used this: 62
producer has loaded this: 70
consumer has used this: 70
consumer has used this: 0
producer has loaded this: 7
consumer has used this: 7
producer has loaded this: 46
consumer has used this: 46
producer has loaded this: 30
producer has loaded this: 43
consumer has used this: 43
producer has loaded this: 36
producer has loaded this: 59
```

Как видно из результата, работа всех четырех потоков происходит некорректно. Синхронизация при использовании переменной `shared` больше не работает, что приводит к потере большого количества значений, созданных двумя потоками `Producer`, а в потоках `Consumer` время от времени считывается значение `0` без ожидания генерации нового значения.

Как же решить эту дополнительную проблему?

В таких случаях на помощь приходит Queue (очередь).

Изменив предыдущий код, мы удаляем переменную shared и заменяем ее на Queue, которая способна правильно управлять распределением между параллельными потоками, будь то поток Producer либо поток Consumer:

```
from threading import Thread
from queue import Queue
import time
import random

queue = Queue() shared = 1
count = 5

class Consumer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    self.count = count

  def run(self):
    global queue
    for i in range(self.count):
      local = queue.get()
      print("consumer has used this: %s" %local)
      queue.task_done()

class Producer(Thread):
  def __init__(self, count):
    Thread.__init__(self)
    self.count = count

  def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    global queue
    for i in range(self.count):
      local = self.request()
      queue.put(local)
      print("producer has loaded this: %s" %local)

t1 = Producer(count)
t2 = Producer(count)
t3 = Consumer(count)
t4 = Consumer(count)
t1.start()
```

```
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()
```

Запуск этого кода дает следующий результат:

```
producer has loaded this: 31
producer has loaded this: 33
consumer has used this: 31
consumer has used this: 33
producer has loaded this: 11
consumer has used this: 11
producer has loaded this: 2
consumer has used this: 2
producer has loaded this: 27
consumer has used this: 27
producer has loaded this: 68
consumer has used this: 68
producer has loaded this: 92
consumer has used this: 92
producer has loaded this: 19
consumer has used this: 19
producer has loaded this: 87
consumer has used this: 87
producer has loaded this: 91
consumer has used this: 91
```

Теперь код работает полностью корректно.

Заключение

В этой главе мы подробно рассмотрели всю информацию, которую нужно знать об инструментах модуля `threading`. Мы рассмотрели различные способы определения набора потоков в программе: вызов функций, использование подклассов или вставка в `ThreadPoolExecutor`. Мы также рассмотрели различные механизмы синхронизации потоков и их отличия друг от друга. Как бы то ни было, нам удалось убедиться в непредсказуемости поведения потоков и в том, как легко столкнуться с проблемами конкурентных условий. В следующей главе мы перейдем к реальному параллельному программированию, которое в Python реализуется исключительно с помощью процессов. Мы познакомимся с модулем multiprocessing, предоставляемым стандартной библиотекой Python.

Что следует помнить

- Методы синхронизации: они незаменимы в тех случаях, когда необходимо использовать память, совместно используемую несколькими потоками.
- Чрезмерная синхронизация: препятствует одновременному выполнению потоков, что приводит к их последовательному выполнению.

Вопросы

1. В чем разница между классами Lock и RLock?
2. Что такое протокол контекстного менеджера и для чего он нужен?
3. Существует ли механизм синхронизации, более эффективный, чем другие?

Список литературы

- *https://realpython.com/python-with-statement/*.
- *https://peps.python.org/pep-0343/*.
- *https://en.wikipedia.org/wiki/Semaphore_(programming)*.
- *https://www.meccanismocomplesso.org/i-thread-in-python-threading-parte-1/*.
- *https://www.meccanismocomplesso.org/thread-in-python-lock-e-deadlock-parte-4/*.
- *https://www.meccanismocomplesso.org/thread-in-python-il-modello-producer-consumer-parte-5/*.

Глава 3

Многопроцессорные вычисления и библиотека mpi4py

В этой главе мы приступим к изучению параллельного программирования. Дело в том, что в Python только процессы могут выполняться в параллельном режиме, что и обеспечивает реальные преимущества параллельных вычислений. В Python существует два основных подхода к реализации процессов в параллельном программировании: модуль multiprocessing стандартной библиотеки и библиотека `mpi4py`, которая позволяет использовать протокол MPI и в языке Python. Модуль multiprocessing использует парадигму программирования с общей памятью, при которой процессы внутри программы могут обращаться к общей области памяти. Как и в случае с потоками, этот подход может привести к возникновению условий конкуренции доступа и проблем синхронизации. В связи с этим в модуле multiprocessing предусмотрены два канала связи между процессами: **Queue** и **Pipe**. Они дают параллельным процессам возможность обмениваться данными полностью синхронно и без появления проблем, связанных с общей памятью. Библиотека `mpi4py`, с другой стороны, основана на парадигме передачи сообщений. В этой модели нет общих ресурсов, и все взаимодействия между процессами происходят путем обмена процессами.

Структура

В этой главе мы обсудим следующие темы:

- модуль multiprocessing;
- процесс как класс и подкласс;
- каналы обмена данными – Queue и Pipe;
- пул процессов;
- ProcessPoolExecutor;
- библиотека Mpi4py;

- двухточечная коммуникация;
- коллективная коммуникация;
- топологии.

Процессы и модуль multiprocessing

В языке Python настоящими участниками параллельного программирования являются процессы, поскольку только они способны обеспечить реальное одновременное выполнение кода. Это связано с тем, что, как было описано в предыдущих главах, потоки в Python не могут действовать параллельно, и самое большее, что можно реализовать с их помощью, – это совместное выполнение.

В связи с этим программирование процессов в Python имеет чрезвычайно важное значение, настолько, что в стандартной библиотеке имеется специальный модуль, полностью посвященный их реализации: multiprocessing.

Этот модуль позволяет выполнять все возможные операции по созданию и управлению процессами в рамках программы и делает это похожим образом, как и модуль threading с потоками. Инструкции и конструкции, используемые в этом модуле, почти идентичны:

- конструктор `Process()` позволяет создавать процесс;
- вызов метода `start()` запускает процесс;
- вызов метода `join()` приводит к тому, что выполнение программы (главного процесса – `Main`) ожидает завершения выполнения всех параллельно запущенных процессов.

Жизненный цикл процесса основан на трех состояниях:

- готов (ready);
- выполнение (running);
- ожидание (waiting).

Далее модуль `multiprocessing` структурирует параллельное программирование следующим образом. Основной процесс запускается с выполнением программы. Внутри него с помощью конструктора `Process()` будет определено несколько процессов. На этом этапе только что созданные процессы находятся в состоянии Ready. Затем в определенный момент часть программы запускается для параллельного выполнения. Все объявленные процессы, то есть дочерние (child) процессы, будут активированы одновременно путем вызова метода `start()` для каждого из них. Главный процесс, в данном случае также являющийся родительским процессом, продолжит свое выполнение в асинхронном режиме и не будет ждать завершения выполнения дочерних процессов (см. рис. 3.1).

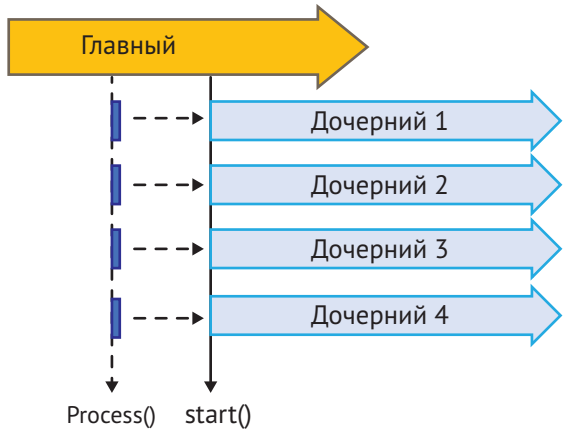


Рис. 3.1. Асинхронный режим без join()

Если же мы хотим использовать механизм синхронизации и нам нужно, чтобы основной процесс ждал результата работы своих дочерних процессов, мы вызовем метод `join()` для каждого процесса (см. рис. 3.2):

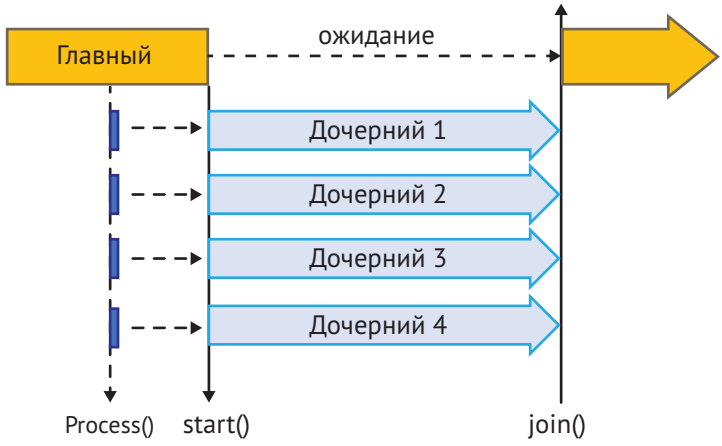


Рис. 3.2. Синхронный режим с join()

Как и в случае с потоками модуля threading, мы пишем код с использованием процессов. Как видите, в дизайне API между потоками и процессами есть хорошая сопоставимость.

Каждый параллельно активированный процесс будет выполнять определенные операции, указанные целевой (target) функцией. Все процессы могут выполнять одну и ту же операцию, или каждая из них может выполнять разные операции. На самом деле целевая функция указывается для каждого отдельного процесса путем передачи его имени в качестве аргумента target в конструкторе.

Чтобы лучше понять описанные выше концепции, давайте рассмотрим пример и напишем следующий код:

```
import multiprocessing
import time

def function(i):
  print ("start Process %i" %i)
  time.sleep(2)
  print ("end Process %i" %i)
  return

if __name__ == '__main__':
  p1 = multiprocessing.Process(target=function, args=(1,))
  p2 = multiprocessing.Process(target=function, args=(2,))
  p3 = multiprocessing.Process(target=function, args=(3,))
  p4 = multiprocessing.Process(target=function, args=(4,))
  p5 = multiprocessing.Process(target=function, args=(5,))
  p1.start()
  p2.start()
  p3.start()
  p4.start()
  p5.start()
  p1.join()
  p2.join()
  p3.join()
  p4.join()
  p5.join()
  print("END Program")
```

Как видно из кода, с помощью конструктора `Process()` определяется параллельное выполнение пяти процессов. Все они будут выполнять одну и ту же целевую функцию, которую для простоты мы здесь назвали function. Если для этой функции требуются какие-либо аргументы, их также можно передать в конструктор `Process()` в виде кортежа в аргументе `args`. В коде мы передаем число, необходимое для идентификации процесса, запущенного внутри функции. Все эти процессы будут активированы с помощью метода `start()`, а затем синхронизированы с основными процессами с помощью метода `join()`.

После запуска кода получаем следующий результат.

```
Start Process 1
Start Process 2
Start Process 3
Start Process 4
Start Process 5
End Process 1
End Process 2
End Process 4
```

```
End Process 3
End Process 5
END Program
```

Результат очень похож на тот, что был получен с помощью потоков и модуля threading, но в данном случае все пять процессов работали параллельно.

Способ реализации кода в параллельном режиме, как в случае выше, очень понятен, все инструкции выражены по отдельности и явным образом. Поскольку нам необходимо использовать пять процессов в параллельном режиме, мы определили пять строк для конструкторов, пять строк для их активации с помощью метода start и пять строк для их синхронизации с основным процессом с помощью метода `join()`. Но если количество процессов станет значительно больше, реализация кода подобным образом может стать весьма проблематичной.

Существует много более эффективных конструкций, которые используют итерации и другие механизмы для обобщения всех предыдущих шагов. Модификация предыдущего кода может выглядеть следующим образом:

```
import multiprocessing
import time

def function(i):
  print ("start Process %i" %i)
  time.sleep(2)
  print ("end Process %i" %i)
  return

if __name__ == '__main__':
  processes = []
  n_procs = 5

  for i in range(n_procs):
    p = multiprocessing.Process(target=function, args=(i,))
    processes.append(p)
    p.start()

  for i in range(n_procs):
    processes[i].join()

  print("END Program")
```

Если вы запустите измененный код, то увидите, что полученный результат не изменится.

Использование идентификаторов процесса

В приведенном выше примере мы использовали передачу аргументов в конструкторе `Process()` для пересылки номера процесса через итератор `i`. Суще-

ствует еще один способ идентификации запущенных процессов, а именно с помощью их идентификаторов – **Process ID** (**PID**). Это очень простая операция: из стандартной библиотеки мы импортируем модуль `os`, который предоставляет нам функцию `getpid()` для получения PID каждого запущенного процесса.

В рассмотренный выше код внесем изменения, заменив числа идентификаторами процессов, как показано в коде ниже:

```
import multiprocessing
import os
import time

def function():
  pid = os.getpid()
  print ("start Process %s" %pid)
  time.sleep(2)
  print ("end Process %s" %pid)
  return

if __name__ == '__main__':
  processes = []
  n_procs = 5

  for I in range(n_procs):
    p = multiprocessing.Process(target=function)
    processes.append(p)
    p.start()

  for i in range(n_procs):
    processes[i].join()

  print("END Program")
```

Как видите, конструктор `Process()` больше не нуждается в использовании `args` в качестве аргумента, потому что функция сама будет его обрабатывать. Внутри него фактически находится вызов функции `getpid()`, которая вернет PID процесса, в котором она будет выполняться. Мы получим пять разных PID из одной и той же функции без необходимости использования аргументов, переданных из главного процесса.

Если мы запустим этот код, то получим следующий результат:

```
start Process 20644
start Process 20000
start Process 16240
start Process 1988
start Process 24388
end Process 20644
end Process 20000
```

```
end Process 16240
end Process 1988
end Process 24388
END Program
```

Как видно из результата, на этот раз вместо последовательных номеров у нас есть PID, которые позволяют идентифицировать различные параллельно запущенные процессы. Эти PID совпадают с теми, которые распознает операционная система. Это означает, что их выполнение можно будет отслеживать и с помощью других инструментов или приложений, которые контролируют потребление ресурсов или управляют активными процессами, не входящими в сферу действия интерпретатора Python.

Пул процессов

Дальнейшее развитие шаблона программирования для совместного управления несколькими процессами заключается в использовании **пула процессов** (process pool), который в Python представлен классом `multiprocessing.Pool`.

Пул – это объект, отвечающий за определенное количество процессов. С его помощью вы можете контролировать их состояние от создания до использования, а также определять, следует ли приостанавливать некоторые из них для экономии вычислительных ресурсов. Таким образом, `multiprocessing.Pool` – это не что иное, как интерфейс для выполнения конкретной задачи с использованием набора процессов без необходимости указывать, сколько и какие процессы должны выполнять эту задачу. Это позволяет значительно упростить код и сделать его гораздо более читабельным.

Возьмем предыдущий код и преобразуем его – на этот раз с помощью `multiprocessing.Pool`.

```
import multiprocessing
import time

def function(i):
  process = multiprocessing.current_process()
  print ("start Process %i(pid:%s)" %(i,process.pid))
  time.sleep(2)
  print ("end Process %i(pid:%s)" %(i,process.pid))
  return

  if __name__ == '__main__':
    pool = multiprocessing.Pool()
    print("Processes started: %s" %pool._processes)
    for i in range(pool._processes):
      results = pool.apply(function, args=(i,))
    pool.close()

  print("END Program")
```

В коде есть много изменений, которые необходимо принять во внимание. Во-первых, для определения пула процессов мы использовали конструктор класса:

```
pool = multiprocessing.pool.Pool()
```

Это позволит создать пул процессов, где будет использоваться количество рабочих процессов, соответствующее количеству логических ядер процессора в вашей системе. Доступ к этому числу в коде можно получить через `pool._processes`. Затем используется `pool.apply()` для выполнения пулом задачи, представленной функцией `function()`, путем передачи ей номера процесса, сгенерированного через `args` в качестве аргумента. После выполнения всех заданных в коде задач пул будет закрыт с помощью функции `pool.close()`.

Что касается функции `function()`, мы добавили PID процесса для его однозначной идентификации. Чтобы получить это значение, сначала с помощью `current_process()` определяется запущенный процесс, а затем вызывается атрибут `pid`.

Запустив написанный выше код, мы получим следующий результат:

```
Processes started: 12
start Process 0(pid:18196)
end Process 0(pid:18196)
start Process 1(pid:5300)
end Process 1(pid:5300)
start Process 2(pid:11984)
end Process 2(pid:11984)
start Process 3(pid:23436)
end Process 3(pid:23436)
start Process 4(pid:16224)
end Process 4(pid:16224)
start Process 5(pid:3784)
end Process 5(pid:3784)
start Process 6(pid:9196)
end Process 6(pid:9196)
start Process 7(pid:6248)
end Process 7(pid:6248)
start Process 8(pid:9272)
end Process 8(pid:9272)
start Process 9(pid:4168)
end Process 9(pid:4168)
start Process 10(pid:22676)
end Process 10(pid:22676)
start Process 11(pid:16500)
end Process 11(pid:16500)
END Program
```

Как видно из результата, сначала создаются двенадцать рабочих процессов по количеству ядер в компьютере, на котором выполняется код (это значение может меняться от системы к системе), и каждый из них выполняет вызванную функцию в соответствии с шагами, заданными циклом `for`. Как видно из результата, выполнение каждой задачи происходит последовательно. Позже вы узнаете, как максимально эффективно использовать пулы процессов за счет их параллельного запуска с помощью функции `map()`.

Давайте внесем небольшое изменение для лучшего понимания работы пула процессов. При вызове `Pool()` также можно задать фиксированное количество процессов внутри пула независимо от количества ядер в процессоре. В конструкторе количество процессов, которые станут частью пула, определяется с помощью параметра `processes`:

```
pool = multiprocessing.Pool(processes=4)
```

Мы вносим соответствующие изменения в предыдущий код, заменяя определение `Pool()` и оставляя цикл `for` с двенадцатью вызовами функции. Внутри `function()` мы заменяем термин **Process** на более подходящий термин **Task** (задача):

```
import multiprocessing
import time

def function(i):
  process = multiprocessing.current_process()
  print ("start Task %i(pid:%s)" %(i,process.pid))
  time.sleep(2)
  print ("end Task %i(pid:%s)" %(i,process.pid))
  return

if __name__ == '__main__':
  pool = multiprocessing.Pool(processes=4)
  print("Processes started: %s" %pool._processes)
  for i in range(12):
    results = pool.apply(function, args=(i,))
  pool.close()

  print("END Program")
```

Итак, в этом случае у нас будет двенадцать задач, которые будут выполняться всего 4 процессами. При выполнении вновь измененного кода мы получим следующий результат:

```
Processes started: 4
start Task 0(pid:5284)
end Task 0(pid:5284)
start Task 1(pid:2220)
end Task 1(pid:2220)
```

```
start Task 2(pid:15236)
end Task 2(pid:15236)
start Task 3(pid:4692)
end Task 3(pid:4692)
start Task 4(pid:5284)
end Task 4(pid:5284)
start Task 5(pid:2220)
end Task 5(pid:2220)
start Task 6(pid:15236)
end Task 6(pid:15236)
start Task 7(pid:4692)
end Task 7(pid:4692)
start Task 8(pid:5284)
end Task 8(pid:5284)
start Task 9(pid:2220)
end Task 9(pid:2220)
start Task 10(pid:15236)
end Task 10(pid:15236)
start Task 11(pid:4692)
end Task 11(pid:4692)
END Program
```

Как видно из результата, все двенадцать задач выполняются последовательно, как и в предыдущем случае, но на этот раз для их выполнения будут поочередно активироваться и деактивироваться четыре (рабочих) процесса.

Определение процессов как подклассов

До сих пор мы изучали способы определения с помощью конструктора `Process()` процессов, которые будут работать *параллельно*. Затем для них назначалась целевая функция с параллельно выполняемым кодом. Другой способ реализации схемы параллельных процессов заключается в определении этих процессов в качестве подклассов `Process()`. Далее функциональность этих процессов будет задаваться путем переопределения методов `__init__)` и `run()`. В этом случае код, который будет выполняться параллельно, будет вставлен в метод `run()` этого класса.

Для лучшего понимания этой концепции рассмотрим в качестве примера следующий код:

```
from multiprocessing import Process
import time
import random

class ChildProcess(Process):
  def __init__(self, count):
    Process.__init__(self)
```

```
      self.count = count

    def run(self):
      print ("start Process %s" %self.count)
      time.sleep(2)
      print ("end Process %s" %self.count)

if __name__ == '__main__':
  processes = []
  n_procs = 5

  for i in range(n_procs):
    p = ChildProcess(i)
    processes.append(p)
    p.start()

  for i in range(n_procs):
    processes[i].join()
```

Как видно из кода, мы определяем подкласс `Process()`, который мы назвали `ChildProcess()`. Внутри класса у нас есть два переопределенных метода. В `__init__()` мы определим добавляемые атрибуты подкласса, так как в данном случае аргумент `count` передается в конструктор. В методе `run()` мы вставим код, который был в целевой функции.

Запустив код, мы получим следующий результат:

```
start Process 0
start Process 1
start Process 2
start Process 3
start Process 4
end Process 0
end Process 1
end Process 2
end Process 4
end Process 3
```

Каналы взаимодействия между процессами

Процессы, в отличие от потоков, как правило, не используют общее пространство памяти, и поэтому им нужен другой механизм для взаимодействия друг с другом и обмена данными. Однако модуль multiprocessing построен на основе парадигмы общего доступа к памяти и поэтому может обеспечить распределение ресурсов между процессами. Поскольку такая практика приводит к тем же проблемам, что и в случае с потоками, а именно к конфликту приоритетов и отсутствию синхронизации между процессами, ее использование крайне не-

желательно. Как и модуль threading, модуль multiprocessing предоставляет аналогичные инструменты для синхронизации процессов, такие как семафоры, блокировки, события и т. д., но использовать этот подход не рекомендуется. В отличие от модуля threading, для параллельной работы с несколькими процессами модуль multiprocessing предоставляет каналы коммуникации, которые используются в качестве безопасных и синхронизированных механизмов обмена данными. Существует два различных режима работы каналов коммуникации:

- очереди (queues);
- каналы (pipes).

Эти каналы определяются в коде с помощью классов Queue и Pipe, которые предоставляются собственно модулем multiprocessing. Оба класса оснащены конструкторами `Queue()` и `Pipe()`, а также множеством методов, которые управляют обменом данными между процессами и их внутренними механизмами синхронизации. Эти коммуникационные каналы, однажды определенные на уровне главного процесса, являются отличными инструментами для обмена данными в идеально синхронизированном режиме. На самом деле благодаря этим каналам процессы могут безопасно и синхронно отправлять и получать данные без риска возникновения конкурентных конфликтов. Кроме того, если количество отправленных данных превышает количество запрошенных, они не будут перезаписаны, но будут накоплены в этих объектах, благодаря чему обеспечивается буферизация данных.

Очереди

Очереди, которые мы будем использовать в качестве канала взаимодействия между процессами, реализованы в модуле multiprocessing. В Python также есть очереди, относящиеся к модулю queue (который мы уже использовали с потоками). Но на самом деле их внутренняя реализация отличается, и `multiprocessing.Queue` использует иной механизм передачи данных, специфичный для работы с процессами. Иными словами, они используют парадигму передачи сообщений, применяемую исключительно для процессов, и избегают использования механизмов синхронизации для добавления в очереди.

Queue (очередь) – это структура данных, которая реализует очередь по принципу «первым вошел – первым вышел» (First IN-First OUT, FIFO). Процессы смогут добавлять данные в очередь с помощью метода `put()` или получать данные из нее с помощью метода `get()`. Порядок ввода данных в очереди остается неизменным, и отправка данных процессам, которые их запрашивают, будет происходить в том же порядке, в котором они были введены.

В очереди данные могут накапливаться, и это могут быть как простые числовые данные, так и более сложные объекты. Очередь не имеет ограничений по размеру, поэтому проблем с переполнением данных не возникает. Однако ограничение емкости можно задать с помощью аргумента конструктора

Queue(), где значение maxsize указывает максимальное количество объектов, которые может содержать очередь:

```
queue = multiprocessing.Queue(maxsize=100)
```

Тем не менее в контексте емкости очереди иногда важно во время выполнения знать актуальное количество содержащихся в ней объектов. Для этой цели существует метод qsize(), который возвращает такое число:

```
size = queue.qsize()
```

Тем не менее в контексте управления очередью, когда необходимо добавить условия в поток программы, может быть полезно выяснить, пустая очередь или полная. Метод empty() возвращает булево значение True, если очередь пуста, а метод full() возвращает True, если очередь заполнена:

```
if queue.empty():
   ...

if queue.full():
   ...
```

Для лучшего понимания принципа работы очереди можно использовать парадигму «производитель–потребитель», которая очень полезна именно в таких случаях отправки и получения данных. Производителем будет любой процесс, который в результате своего выполнения будет предоставлять данные, а потребителем будет любой процесс, которому эти данные понадобятся для завершения своего выполнения. Для работы этого механизма, созданного как минимум двумя процессами, необходим обмен данными. Поскольку процессы не имеют общей области памяти, они обмениваются данными через очередь. Производитель отправляет произведенные данные в очередь, а потребитель запрашивает их у производителя. Потребители и производители всегда асинхронны, они обычно не зависят друг от друга, за исключением потока данных, который происходит через очередь. Поскольку время работы потребителя и производителя обычно не совпадает, могут быть случаи, когда производитель генерирует данные для потребителя быстрее, и эти данные могут упорядоченно накапливаться в рамках очереди, при этом не перезаписываясь. Таким образом, у потребителя будет достаточно времени для упорядоченного использования этих данных.

Следующий код реализует простую парадигму «производитель–потребитель» на основе двух параллельных процессов, которые используют очередь для обмена данными.

```
from multiprocessing import Process, Queue
import time
import random

class Consumer(Process):
```

```
    def  __init__(self, count, queue):
      Process.__init__(self)
      self.count = count
      self.queue = queue

    def run(self):
      for i in range(self.count):
        local = self.queue.get()
        time.sleep(2)
        print("consumer has used this: %s" %local)

class Producer(Process):
    def __init__ (self, count, queue):
      Process.__init__(self)
      self.count = count
      self.queue = queue

    def request(self):
      time.sleep(1)
      return random.randint(0,100)

    def run(self):
      for i in range(self.count):
        local = self.request()
        self.queue.put(local)
        print("producer has loaded this: %s" %local)

if __name__ == '__main__':
    queue = Queue()
    count = 5
    p1 = Producer(count, queue)
    p2 = Consumer(count, queue)
    p1.start()
    p2.start()
    p1.join()
    p2.join()
```

Запуск этого кода приведет к следующему результату:

```
producer has loaded this: 55
producer has loaded this: 30
consumer has used this: 55
producer has loaded this: 60
producer has loaded this: 14
consumer has used this: 30
producer has loaded this: 97
consumer has used this: 60
```

```
consumer has used this: 14
consumer has used this: 97
```

Как можно видеть, значения, сгенерированные процессом производителя, накапливаются в очереди и впоследствии используются процессом потребителя в идеальной синхронизации, без потери или перезаписи значений.

Каналы

Еще одним классом, выполняющим функции средства коммуникации между процессами, является **канал** (Pipe). Канал реализует двустороннее взаимодействие между двумя процессами, рассчитанное на отправку и прием данных между ними. Как и класс Queue, он имеет собственный конструктор `Pipe()`, который при объявлении создает два разных объекта `multiprocessing.connection.Connection`. Обычно при объявлении конструктора `Pipe()` оба экземпляра подключений размещаются отдельно:

```
conn1, conn2 = multiprocessing.Pipe()
```

Одно из этих соединений будет использоваться для отправки данных, а другое – для их приема. По умолчанию первое соединение (`conn1`) используется только для приема данных, а второе соединение (`conn2`) – только для отправки данных.

В дополнение к двунаправленному режиму Pipe также может быть создан с дуплексным режимом, где каждое из двух соединений может использоваться как для отправки, так и для приема данных. В этом случае необходимо установить аргумент `duplex`, передаваемый в конструкторе, в значение `True`:

```
conn1, conn2 = multiprocessing.Pipe(duplex=True)
```

В обоих режимах процессы через два сгенерированных объекта Connection смогут отправлять данные с помощью метода `send()` и принимать их с помощью метода `recv()`:

```
conn2.send(object) object = conn1.recv()
```

Как и в случае с очередями, размер отправляемых объектов при работе с каналами не ограничен; единственное условие – они должны быть извлекаемыми. При отправке они будут извлечены, а после получения – автоматически выгружены.

Как и в случае с очередями, класс `Pipe` также предоставляет набор методов для управления соединениями, созданными при его использовании. Например, состояние Pipe можно контролировать с помощью функции `poll()`. Эта функция возвращает булево значение True, если в соединении есть данные, ожидающие приема процессом. Этот метод очень полезен для управления потоком выполнения программы:

```
if conn1.poll():
```

Рассмотрим только что изложенные концепции на примере следующего кода:

```
from multiprocessing import Process, Pipe
import time
import random

class Consumer(Process):
  def __init__(self, count, conn):
    Process.__init__(self)
    self.count = count
    self.conn = conn

  def run(self):
    for i in range(self.count):
      local = self.conn.recv()
      time.sleep(2)
      print("consumer has used this: %s" %local)

class Producer(Process):
  def __init__(self, count, conn):
    Process.__init__(self)
    self.count = count
    self.conn = conn

  def request(self):
    time.sleep(1)
    return random.randint(0,100)

  def run(self):
    for i in range(self.count):
      local = self.request()
      self.conn.send(local)
      print("producer has loaded this: %s" %local)

if __name__ == '__main__':
  recver, sender = Pipe()
  count = 5
  p1 = Producer(count, sender)
  p2 = Consumer(count, recver)
  p1.start()
  p2.start()
  p1.join()
  p2.join()
  recver.close()
  sender.close()
```

Кроме того, в данном случае мы использовали парадигму «производитель–потребитель» и максимально сохранили код, использовавшийся ранее с Queue. Это сделано для того, чтобы помочь читателю провести соответствующие аналогии между этими двумя режимами работы. При запуске кода вы получите следующий результат:

```
producer has loaded this: 0
producer has loaded this: 76
consumer has used this: 0
producer has loaded this: 28
producer has loaded this: 90
consumer has used this: 76
producer has loaded this: 75
consumer has used this: 28
consumer has used this: 90
consumer has used this: 75
```

По окончании использования Pipe мы вызвали метод `close()` для каждого из двух соединений. Эта операция очень важна для освобождения всех ресурсов, используемых Pipe.

Наиболее важным и, возможно, наиболее ограничивающим аспектом работы с Pipe является то, что он работает исключительно между двумя конечными точками, а значит, только с двумя процессами. Если вам нужно работать одновременно с несколькими процессами, вам потребуется установить Pipe для каждой пары процессов. Очевидно, что, когда количество параллельно работающих процессов становится большим, управление Pipe становится проблематичным.

Pipe в сравнении с Queue

Как только что было показано, в качестве каналов взаимодействия модуль многопроцессорной обработки предлагает два решения: Pipe и Queue. Какое из них выбрать для обмена данными между параллельными процессами?

На первый взгляд, с учетом ограничения на использование Pipe только между двумя процессами, может показаться очевидным, что лучшим решением всегда будет Queue. Но это не всегда так.

Pipe – это намного более простой класс, чем Queue. Его реализация происходит на низком уровне, и именно эта особенность делает его более эффективным и потенциально более быстрым для обмена данными между двумя процессами. Таким образом, если этот тип соединения может быть использован между парами процессов, он будет предпочтительнее, чем Queue. Если же, с другой стороны, обмен данными является постоянным, а количество процессов, участвующих в этом обмене, велико, то использование Queue может оказаться единственно возможным.

Отображение функции через пул процессов

Еще одним очень важным аспектом параллельного программирования является отображение (mapping) функции. В Python можно расширить функциональность `map()` между несколькими процессами в параллельном режиме. Эта функция очень полезна в тех случаях, когда необходимо применить функцию к объекту, поддерживающему **итерацию**. В действительности `map()` применяет функцию к каждому элементу объекта и в результате возвращает массив элементов одинакового размера, каждый из которых содержит результат применения функции к каждому элементу.

Для лучшего понимания работы функции `map()` нет ничего лучше, чем прямой пример. Рассмотрим следующий код:

```
import time
import math
import numpy as np

def func(value):
    result = math.sqrt(value)
    print("The value %s and the elaboration is %s" %(value, result) )
    return result

if __name__ == '__main__':
    data = np.array([10,3,6,1]) results = map(func, data)
    for result in results:
      print("This is the result: %s" %result)
```

В коде мы выбрали массив `NumPy` из четырех элементов, к которому будем применять функцию `func()`. В результате работы функции получаем массив из четырех элементов, каждый из которых содержит возвращаемое значение функции, примененной к соответствующему элементу.

При запуске кода получим следующий результат:

```
The value 10 and the elaboration is 3.1622776601683795
This is the result: 3.1622776601683795
The value 3 and the elaboration is 1.7320508075688772
This is the result: 1.7320508075688772
The value 6 and the elaboration is 2.449489742783178
This is the result: 2.449489742783178
The value 1 and the elaboration is 1.0
This is the result: 1.0
```

Этот механизм отображения объектов `iterable` можно использовать в параллельном программировании. Эффективность может значительно возрасти, если вычисление функции для каждого элемента будет выполняться параллельно. Например, каждый элемент может быть назначен процессу, целью выполнения которого является функция, переданная в `map()`. В этом случае все

элементы будут вычисляться одновременно. Для массива из *n* элементов обычная программа потребует времени *n* * *t*, где *t* – это время выполнения функции. В параллельном режиме тот же массив из *n* элементов будет передан *n* процессам, а время выполнения не превысит *t* (если в системе *n* ядер или *n* процессоров). Таким образом, повышение эффективности не вызывает никаких сомнений.

Для параллельного отображения функции модуль multiprocessing предоставляет класс `multiprocessing.pool.Pool`. В этом классе для этого предусмотрены два метода:

- `map()`;
- `map_async ()`.

В случае параллельного программирования есть два метода из-за различных возможных вариантов синхронизации. Оба они работают почти одинаково, за исключением того, что функция `map()` должна использоваться в тех случаях, когда необходимо заблокировать выполнение программы до завершения работы всех целевых функций в параллельном режиме. Функция `map_ async()` должна использоваться в тех случаях, когда блокировка всего выполнения программы не требуется.

Метод `map()` принимает два аргумента: целевую функцию и объект `iterable`:

```
results = pool.map(target, iterable):
```

Как следует из названия, класс pool будет создавать пул процессов, равный количеству элементов итерабельного объекта, и передавать целевую функцию этим процессам.

Но для лучшего понимания этих концепций и принципов их работы необходимо перейти к практическим примерам.

```
import time
import math
import numpy as np

from multiprocessing.pool import Pool

def func(value):
    result = math.sqrt(value)
    print("The value %s and the elaboration is %s" %(value, result) )
    time.sleep(value)
    return result

if __name__ == '__main__':
    with Pool() as pool:
        data = np.array([10,3,6,1])
        results = pool.map(func, data)
        print("The main process is going on...")
```

```
        for result in results:
            print("This is the result: %s" %result)
    print("END Program")
```

При запуске этой программы получается следующий результат:

```
The value 10 and the elaboration is 3.1622776601683795
The value 3 and the elaboration is 1.7320508075688772
The value 6 and the elaboration is 2.449489742783178
The value 1 and the elaboration is 1.0
The main process is going on...
This is the result: 3.1622776601683795
This is the result: 1.7320508075688772
This is the result: 2.449489742783178
This is the result: 1.0
END Program
```

Что касается функции `map_async()`, мы можем изменить предыдущий код с помощью соответствующих замен, как показано в следующем коде:

```
import time
import math
import numpy as np
from multiprocessing.pool import Pool

def func(value):
    result = math.sqrt(value)
    print("The value %s and the elaboration is %s" %(value, result) )
    time.sleep(value)
    return result

if __name__ == '__main__': with Pool() as pool:
    data = np.array([10,3,6,1])
        results = pool.map_async(func, data)
        print("Main Process is going on...")
        for result in results.get():
            print("This is the result: %s" %result)
    print("END Program")
```

При запуске этой программы можно получить следующий результат:

```
Main Process is going on...
The value 10 and the elaboration is 3.1622776601683795
The value 3 and the elaboration is 1.7320508075688772
The value 6 and the elaboration is 2.449489742783178
The value 1 and the elaboration is 1.0
This is the result: 3.1622776601683795
This is the result: 1.7320508075688772
```

```
This is the result: 2.449489742783178
This is the result: 1.0
END Program
```

Как видно из двух разных результатов, мы получили ожидаемое поведение. Используя метод `map()`, мы получаем механизм синхронизации, который прерывает выполнение процесса `main`. Он не возобновляется до тех пор, пока все параллельные процессы, запущенные `map()`, не завершат свое выполнение. В действительности текст «`Main process is going on…`» («основной процесс продолжается…») появляется только после завершения всех дочерних процессов и возврата их результатов в `results`. При использовании `map_ async()` выполнение основного процесса продолжается и не ожидает полной оценки результатов. Текст «`Main process is going on…`» появляется в начале текста.

Параллельное отображение с chunksize

Функция `map()` применяет функцию к каждому элементу в объекте `iterable`. Если объект `iterable` содержит большое количество элементов, вызов большого количества процессов для их выполнения может оказаться очень неэффективным.

Более эффективный подход заключается в разделении элементов объекта `iterable` на части с определенным количеством элементов, каждая из которых будет назначена процессу. Это можно сделать посредством передачи аргумента `chunksize` функции `map()`.

Давайте изменим пример кода, который мы использовали ранее для метода `map()`, увеличив количество элементов в `np.array()`. Затем мы установим части массива из четырех элементов для каждого процесса, передавая аргумент `chunksize`, равный 4, методу `map()`:

```
import time
import math
import numpy as np
from multiprocessing.pool import Pool

def func(value):
    result = math.sqrt(value)
    print("The value %s and the elaboration is %s" %(value, result) )
    time.sleep(value)
    return result

if __name__ == '__main__':
    with Pool() as pool:
        data = np.array([10,3,6,1,4,5,2,9,7,3,4,6])
        results = pool.map(func, data, chunksize=4)
        print("The main process is going on...")
        for result in results:
            print("This is the result: %s" %result)
    print("END Program")
```

Запуск этого кода дает следующий результат:

```
The value 10 and the elaboration is 3.1622776601683795
The value 4 and the elaboration is 2.0
The value 7 and the elaboration is 2.6457513110645907
The value 5 and the elaboration is 2.23606797749979
The value 3 and the elaboration is 1.7320508075688772
The value 2 and the elaboration is 1.4142135623730951
The value 3 and the elaboration is 1.7320508075688772
The value 4 and the elaboration is 2.0
The value 9 and the elaboration is 3.0
The value 6 and the elaboration is 2.449489742783178
The value 6 and the elaboration is 2.449489742783178
The value 1 and the elaboration is 1.0
The main process is going on...
This is the result: 3.1622776601683795
This is the result: 1.7320508075688772
This is the result: 2.449489742783178
This is the result: 1.0
This is the result: 2.0
This is the result: 2.23606797749979
This is the result: 1.4142135623730951
This is the result: 3.0
This is the result: 2.6457513110645907
This is the result: 1.7320508075688772
This is the result: 2.0
 This is the result: 2.449489742783178
END Program
```

В дополнение к функции `map()` класс `Pool` предоставляет ряд методов с аналогичными свойствами, таких как `imap()` и `apply()`, и все они также имеют асинхронные версии.

Класс ProcessPoolExecutor

Механизм, аналогичный пулам процессов, реализован в классе `ProcessPoolExecutor`, который доступен в модуле `concurrent.future`. Этот класс также предоставляет пул многократно используемых процессов для выполнения определенных операций, в том числе `map()`.

Кроме того, в этом случае вы указываете имя целевой функции для выполнения и итератор, к которому будет применена функция. После завершения всех назначенных задач `ProcessPoolExecutor` необходимо закрыть для освобождения всех использованных ресурсов. Для закрытия используется функция `shutdown()`.

Но даже в этом случае существуют гораздо более простые и эффективные конструкции. В действительности `ProcessPoolExecutor` совместим с контекстным менеджером и объявлением `with`:

```
with ProcessPoolExecutor() as executor:
```

В этом случае, когда выполнение покидает блок контекстного менеджера, функция shutdown() вызывается автоматически. Контекстный менеджер соответствует следующей формулировке:

```
try:
    executor = ProcessPoolExecutor()
finally:
    executor.shutdown()
```

В связи с этим можно сделать вывод, что использование контекстного менеджера в таких случаях является весьма удобным. Класс ProcessPoolExecutor в качестве пула очень полезен для параллельного выполнения функции map(). Это можно увидеть, например, в следующем коде:

```
import time
import math
import numpy as np
from concurrent.futures import ProcessPoolExecutor

def func(value):
    result = math.sqrt(value)
    print("The value %s and the elaboration is %s" %(value, result) )
    time.sleep(value)
    return result

if __name__ == '__main__':
    with ProcessPoolExecutor(10) as executor:
        data = np.array([10,3,6,1])
        for result in executor.map(func, data):
            print("This is the result: %s" %result)
    print("END Program")
```

Запуск этого кода дает следующий результат:

```
The value 10 and the elaboration is 3.1622776601683795
The value 3 and the elaboration is 1.7320508075688772 The value 6 and the
elaboration is 2.449489742783178
The value 1 and the elaboration is 1.0
This is the result: 3.1622776601683795
This is the result: 1.7320508075688772
This is the result: 2.449489742783178
This is the result: 1.0
END Program
```

Ранее мы ознакомились с методом map() класса Pool(), который давал возможность использовать части нескольких элементов, назначаемых каждому процессу, при помощи аргумента chunksize. Что ж, в случае ProcessPoolExecutor метод map() принимает тот же аргумент.

Далее мы модифицируем предыдущий код, увеличивая количество элементов в `np.array()` и добавляя аргумент `chunksize`, равный 4. Кроме того, чтобы распознать, какой процесс обрабатывает различные элементы `np.array`, мы импортируем модуль `os`, который использует `getpid()` для получения ID различных процессов.

```
import time
import math
import os
import numpy as np
from concurrent.futures import ProcessPoolExecutor

def func(value):
    result = math.sqrt(value)
    pid = os.getpid()
    print("[Pid:%s] The value %s and the elaboration is %s" %(pid, value,
result) )
    time.sleep(value)
    return result

if __name__ == '__main__':
    with ProcessPoolExecutor(10) as executor:
        data = np.array([10,3,6,1,4,5,2,9,7,3,4,6])
        for result in executor.map(func, data, chunksize=4):
            print("This is the result: %s" %result)
    print("END Program")
```

Запуск этого кода дает следующие результаты:

```
[Pid:14716] The value 10 and the elaboration is 3.1622776601683795
[Pid:22496] The value 4 and the elaboration is 2.0
[Pid:6508] The value 7 and the elaboration is 2.6457513110645907
[Pid:22496] The value 5 and the elaboration is 2.23606797749979
[Pid:6508] The value 3 and the elaboration is 1.7320508075688772
[Pid:22496] The value 2 and the elaboration is 1.4142135623730951
[Pid:14716] The value 3 and the elaboration is 1.7320508075688772
[Pid:6508] The value 4 and the elaboration is 2.0
[Pid:22496] The value 9 and the elaboration is 3.0
[Pid:14716] The value 6 and the elaboration is 2.449489742783178
[Pid:6508] The value 6 and the elaboration is 2.449489742783178
[Pid:14716] The value 1 and the elaboration is 1.0
This is the result: 3.1622776601683795
This is the result: 1.7320508075688772
This is the result: 2.449489742783178
This is the result: 1.0
This is the result: 2.0
This is the result: 2.23606797749979
This is the result: 1.4142135623730951
```

```
This is the result: 3.0
This is the result: 2.6457513110645907
This is the result: 1.7320508075688772
This is the result: 2.0
This is the result: 2.449489742783178
END Program
```

Библиотека mpi4py

Совершенно иным подходом в параллельном программировании на Python с использованием процессов является использование **интерфейса передачи сообщений** (Message Passage Interface, MPI). Этот коммуникационный протокол является практически стандартом в области параллельного программирования, особенно для систем с несколькими узлами (несколькими процессорами). Этот протокол используется во многих языках программирования, в том числе Fortran и C.

В Python также имеется ряд модулей MPI, полезных для параллельного программирования. Наиболее интересным из них является библиотека `mpi4py`. Эта библиотека была разработана на основе спецификации MPI-1/2. Она предоставляет объектно ориентированный интерфейс на основе соединений MPI-2 C++.

Эта библиотека опирается на общие концепции, которые значительно отличаются от тех, что мы видели в модуле multiprocessing и в стандартной библиотеке Python. Эти концепции фактически основаны на стандартах, разработанных специально для многопроцессорных и распределенных архитектур, а также для кластеров. Из этого следует, что методы выполнения и конструкции кода в этом случае значительным образом отличаются от тех, что мы видели в первой части этой главы.

Почему же стоит использовать модули MPI вместо модуля multiprocessing? Тому есть несколько причин. В первую очередь MPI практически стал стандартом параллельного программирования во многих языках программирования, включая Python, и он хорошо адаптируется к самым разным архитектурам. В то же время его можно использовать и на простых системах с одним многоядерным процессором, где эффективность вычислений не так очевидна, но где этот модуль может быть полезен для практических экспериментов и разработки программ, которые в дальнейшем можно будет адаптировать к более сложным архитектурам.

Первым делом необходимо установить библиотеку `mpi4py`. Для Anaconda просто запустите следующую команду:

```
conda install mpi4py
```

Если вы используете Python, вам понадобится `pip`:

```
pip install mpi4py
```

Для запуска программ на Python вместо классической команды `python` вам необходимо использовать следующую команду из оболочки:

```
mpiexec -n x python mpi4py _name.py
```

Команда `mpiexec` обычно используется для прямого запуска нескольких процессов в параллельном режиме, а `x` обозначает количество процессов в параллельном режиме.

Для ознакомления с функциональностью библиотеки начнем с самой простой программы, которую можно запустить с помощью `mpiexec`:

```
from mpi4py import MPI comm = MPI.COMM_WORLD
rank = comm.rank
print("The process %d is started" %rank)
```

Для выполнения кода нам понадобится команда `mpiexec`. Чтобы запустить `2` процесса параллельно, выполним:

```
$ mpiexec -n 2 python mpi01.py
```

Мы получим вывод двух процессов:

```
The process 0 is started
The process 1 is started
```

Если же мы хотим запустить `6` процессов в параллельном режиме, достаточно изменить значение параметра `n`, передаваемого в командной строке:

```
$ mpiexec -n 6 python mpi01a.py
```

При запуске кода будет получен следующий результат:

```
The process 2 is started
The process 5 is started
The process 1 is started
The process 4 is started
The process 3 is started
The process 0 is started
```

Как видно из результатов, порядок следования процессов не является упорядоченным, а представляет собой случайный результат, который меняется от запуска к запуску. При анализе кода параметр `comm` выражает коммуникатор и определяет группу процессов, которые могут обмениваться данными через механизм передачи сообщений:

```
comm = MPI.COMM_WORLD
```

Ранг представляет собой идентификационный номер процесса в группе взаимодействующих процессов. Это означает, что все процессы, запущенные командой `mpiexec`, становятся частью этой группы взаимодействия, и каждый из них идентифицируется номером, рангом.

Параллелизм процессов

Как уже было описано в разделе о модуле multiprocessing, процессы позволяют выполнять код в параллельном режиме. Мы можем провести те же тесты с биб-

лиотекой mpi4py, и на этот раз мы проведем тестирование производительности посредством измерения времени выполнения. Для этого нужно запустить следующую программу:

```
import time
from mpi4py import MPI comm = MPI.COMM_WORLD
rank = comm.rank
print("The process %d is started" %rank) time.sleep(10)
print("The process %d is ended" %rank)
```

Для измерения времени, затраченного на выполнение команды mpiexec, нельзя использовать модуль time, но можно запустить команду time в Linux или Windows PowerShell. После запуска полученный результат будет примерно таким:

```
> Measure-Command { mpiexec -n 4 python mpi01b.py | Out-Host }
The process 2 is started
The process 2 is ended
The process 3 is started
The process 3 is ended
The process 1 is started
The process 1 is ended
The process 0 is started
The process 0 is ended

Days : 0
Hours : 0
Minutes : 0
Seconds : 10
Milliseconds : 298
Ticks : 102985516
TotalDays : 0,000119196199074074
TotalHours : 0,00286070877777778
TotalMinutes : 0,171642526666667
TotalSeconds : 10,2985516
TotalMilliseconds : 10298,5516
```

Как можно видеть, выполнение занимает около 10 с, что соответствует ожидаемому времени выполнения каждого процесса. Последовательная версия предыдущей программы дает следующий код:

```
import time

for i in range(4):
      print("The process %d is started" %i)
      time.sleep(10)
      print("The process %d is ended" %i)
```

Мы измеряем время, затраченное на повторное выполнение предыдущей программы, используя команду `Measure-Command` в PowerShell (хотя можно было бы использовать модуль `time`).

```
> Measure-Command { python mpi01c.py }

Days : 0
Hours : 0
Minutes : 0
Seconds : 40
Milliseconds : 99
Ticks : 400994538
TotalDays : 0,000464114048611111
TotalHours : 0,0111387371666667
TotalMinutes : 0,66832423
TotalSeconds : 40,0994538
TotalMilliseconds : 40099,4538
```

Как и предполагалось, последовательное выполнение занимает `40` с (четыре операции по `10` с каждая), в то время как параллельное выполнение четырех операций (четырех процессов) занимает около `10` с. В результате мы получаем значительное повышение эффективности.

Эффективность параллелизма с учетом количества процессоров/ядер

Что касается эффективности параллелизма с учетом количества процессоров или вычислительных ядер, различные показатели времени могут быть измерены путем изменения количества параллельных процессов. Для этого можно использовать следующий код:

```
import time
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
print("The process %d is started" %rank)
n = 0
for i in range(100000000):
    n += i
print("The process %d is ended" %rank)
```

Мы осуществляем выполнение кода с помощью `mpiexec`, варьируя количество процессоров. Результаты измерений представлены в табл. 3.1. В первом столбце указано количество параллельно работающих процессов. Во втором столбце представлены результаты измерений времени работы в зависимости от количества процессов.

Таблица 3.1. Время выполнения в зависимости от количества процессов

Кол-во параллельных процессов, N	Время работы, t	Соотношение, s
1	10,7	10,7
2	13,8	6,9
4	16,0	4
6	19,9	3,317
8	27,6	3,45
12	39,6	3,3

Однако нас интересует соотношение $s = t / N$, то есть время выполнения задачи для каждого процесса. Как можно видеть, это значение по мере увеличения количества процессов постепенно уменьшается. Тем не менее можно заметить, что значение стремится к определенному порогу, за которым дальнейшее снижение не происходит, и этот порог находится на уровне примерно 3,3 с. Более наглядно это показано на рис. 3.3.

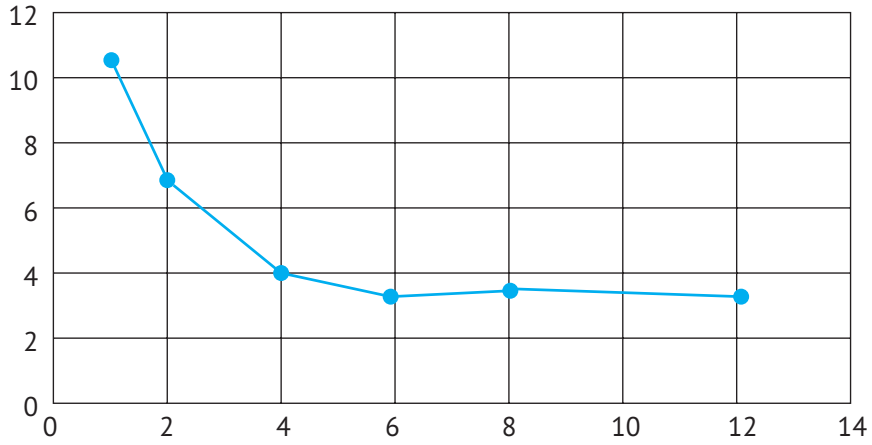


Рис. 3.3. Время выполнения в зависимости от количества процессов

Из рис. 3.3 видно, что при количестве процессов более шести время выполнения каждой задачи практически не изменяется. Это происходит по той причине, что в системе автора используется шесть ядер, и свыше этого количества работа в параллельном режиме уже невозможна.

Основные области применения mpi4py

Библиотека `mpi4py` построена на принципе передачи сообщений в соответствии со стандартами MPI. Области ее применения в мире параллельного программирования можно разделить на три группы:

- двухточечные коммуникации,
- коллективные коммуникации,
- топологии.

Во второй половине этой главы все три эти области применения будут рассмотрены на практических примерах.

Реализация двухточечных коммуникаций

Двухточечные («точка–точка» – point-to-point) операции представляют собой обмен сообщениями между двумя процессами. Теоретически каждая операция отправки должна быть идеально синхронизирована с соответствующей операцией приема. Однако на практике это не всегда так, и реализация MPI должна обеспечивать сохранность отправленных данных даже в случае отсутствия синхронизации между процессами отправки и приема. Обычно для этого используется буфер, который скрыт от разработчика и полностью управляется библиотекой `mpi4py`.

Модуль `mpi4py` обеспечивает двухточечную коммуникацию с помощью двух функций:

- `comm.send(data, receiver_process);`
- `comm.recv(sender_process).`

Функция `send()` отправляет данные процессу приема через его ранг в группе коммуникации. Функция `recv()` принимает данные от процесса отправки, который также идентифицируется по номеру ранга в группе коммуникации.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
print("Process %s started" %rank)
if rank==0:
    msg ="This is my message"
    receiver = 1
    comm.send(msg,dest=receiver)
    print("Process 0 sent: %s " %msg + "to %d" %receiver)
if rank==1:
    source=0
    msg=comm.recv(source)
    print("Process 1 received is: %s" %msg + "from %d" %source)
```

Для запуска приведенного выше кода в параллельном режиме необходимо выполнить следующую команду оболочки:

```
$ mpiexec -n 9 python programma01.py
```

В результате получаем следующий результат:

```
Process 0 started
Process 1 started
Process 2 started
Process 3 started
Process 4 started
Process 5 started
Process 6 started
Process 7 started
Process 8 started
Process 0 sent: This is my message to 1
Process 1 received is: This is my message from 0
```

Методы `send()` и `recv()` являются блокирующими, то есть они блокируют вызывающий процесс до тех пор, пока буферизованные данные не будут готовы к безопасному использованию. Порядок следования процессов в этом случае не является упорядоченным ввиду асинхронного запуска параллельных процессов, и по этой причине порядок строк вывода может отличаться.

Коллективные коммуникации

Наиболее часто используемые виды коллективных (collective) коммуникаций:

- барьерная синхронизация;
- трансляция данных;
- сбор данных;
- распределение данных;
- операция свертки (редукции).

Коллективная коммуникация с трансляцией данных

При параллельном программировании мы нередко сталкиваемся с ситуацией, когда в момент выполнения необходимо поделиться значением определенной переменной между несколькими процессами или распределить различные операции над переменными, для которых каждый процесс предоставляет частичное решение. В подобных ситуациях используются группы древовидной (tree) коммуникации (например, процесс 0 отправляет данные процессам 1 и 2, которые, в свою очередь, отправляют их процессам 3, 4, 5, 6).

Метод взаимодействия, который включает в себя все процессы коммуникатора, называется **коллективной коммуникацией** (collective communication). Соответственно, коллективная коммуникация базируется на более чем двух процессах. В этом случае мы можем назвать ее трансляцией (broadcast), когда один процесс отправляет одни и те же данные другим процессам (см. рис. 3.4):

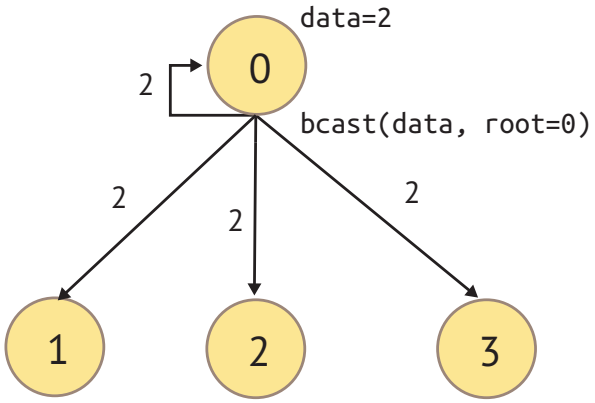


Рис. 3.4. Трансляция

Чтение данных, определенных в процессе 0, который является корневым (root), и их распределение между всеми процессами, включая сам корневой процесс, выглядит следующим образом:

```
from mpi4py import MPI
import random
comm = MPI.COMM_WORLD
rank = comm.rank
if rank == 0:
    data=random.randint(1, 10)
else:
    data=None
data = comm.bcast(data, root=0)
if rank == 1:
    print("The square of %d is %d" %(data,data*data))
if rank == 2:
    print("Half of %d is %d" %(data,data/2))
if rank == 3:
    print("Double of %d is %d" %(data,data*2))
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 4 python mpi04a.py
The square of 6 is 36
Half of 6 is 3
Double of 6 is 12
```

Коллективная коммуникация с распределением данных

Функция распределения (scattering) очень похожа на функцию трансляции, но есть одно отличие. В то время как `comm.bcast()` отправляет один и тот же тип данных всем процессам, которые находятся в режиме прослушивания,

comm.scatter() отправляет части данных в массиве разным процессам, как на рис. 3.5.

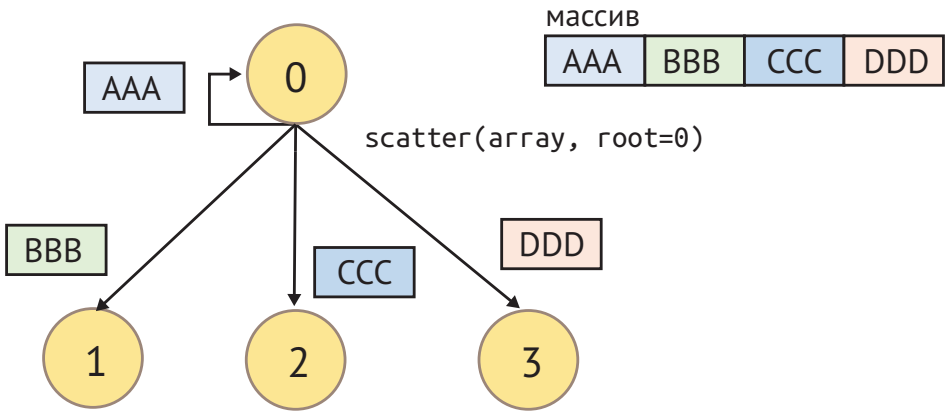


Рис. 3.5. Распределение

Функция comm.scatter() берет элементы массива и распределяет их между процессами в зависимости от их ранга, так что первый элемент будет отправлен процессу 0, второй элемент – процессу 1 и т. д.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
if rank == 0:
    array = ['AAA','BBB','CCC','DDD']
else:
    array = None
data = comm.scatter(array,root=0)
print("Process %d is working on %s element" %(rank,data))
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 4 python mpi05a.py
Process 0 is working on AAA element
Process 1 is working on BBB element
Process 2 is working on CCC element
Process 3 is working on DDD element
```

Коллективная коммуникация со сбором данных

Функция сбора (gathering function) выполняет операцию, обратную распределению. В этом случае все процессы, участвующие в вычислении, отправляют свои фрагменты данных в принимающий корневой процесс, который собирает их все вместе, как на рис. 3.6.

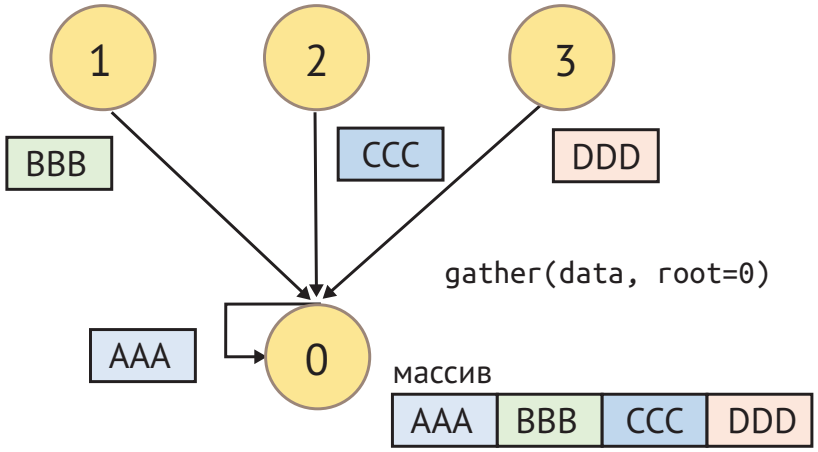


Рис. 3.6. Сбор данных

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
if rank == 0:
    data = 'AAA'
if rank == 1:
    data = 'BBB'
if rank == 2:
    data = 'CCC'
if rank == 3:
    data = 'DDD'
array = comm.gather(data, root=0)
if rank == 0:
    print("The new array is %s " %array)
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 4 python mpi06a.py
The new array is ['AAA', 'BBB', 'CCC', 'DDD']
```

Этот метод передачи сообщений может оказаться очень полезным для выполнения параллельных вычислений. Многие алгоритмы основаны на вычислениях, которые можно разбить на мелкие идентичные фрагменты. Назначив каждый из них параллельному процессу, вы можете эффективно сократить время выполнения. По окончании вычислений будет получено множество частичных результатов, которые затем будут сложены для получения окончательного результата.

Рассмотрим простой пример, чтобы понять, как эта концепция гармонично сочетается с коллективной коммуникацией со сбором данных.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
size = comm.size
data = 2*rank + 1
print("Process %d calculated this value: %d" %(rank,data))
array = comm.gather(data, root=0)
if rank == 0:
    result = 0
    for i in range(0, size):
        result += array[i]
print("The result of the parallel computation is %d" %result)
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 4 python mpi06b.py
Process 3 calculated this value: 7
Process 1 calculated this value: 3
Process 2 calculated this value: 5
Process 0 calculated this value: 1
The result of the parallel computation is 16
```

Как видно из результата, мы разделили вычисление на четыре равные части. В таких случаях в присутствующих в коде вычислениях нередко используется количество рангов каждого отдельного процесса для итераций, а также индексируются части вычислений. Затем отдельные результаты суммируются для получения окончательного результата. Меняя количество параллельных процессов, можно расширить итерацию вычислений до более обширных значений.

Например, расширив параллельные вычисления до восьми процессов, мы получим следующий результат:

```
$mpiexec -n 8 python mpi06b.py
Process 5 calculated this value: 11
Process 7 calculated this value: 15
Process 3 calculated this value: 7
Process 6 calculated this value: 13
Process 1 calculated this value: 3
Process 2 calculated this value: 5
Process 4 calculated this value: 9
Process 0 calculated this value: 1
The result of the parallel computation is 64
```

Коллективная коммуникация в режиме AlltoAll

Другой способ коллективной коммуникации заключается в использовании режима **AlltoAll** («все-ко-всем»). В этом случае процессы ведут себя одновре-

менно как в режиме сбора и как в режиме распределения. В этой группе коммуникации каждый процесс отправляет и получает части массива данных от других процессов, принадлежащих к группе, как на рис. 3.7.

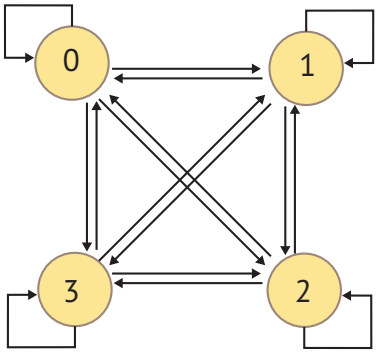


Рис. 3.7. Режим AllToAll

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
if rank==0:
    output = ['0A','0B','0C','0D']
if rank==1:
    output = ['1A','1B','1C','1D']
if rank==2:
    output = ['2A','2B','2C','2D']
if rank==3:
    output = ['3A','3B','3C','3D']


input = comm.alltoall(output)
print("Process %s received %s" %(rank,input))
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 4 python mpi07a.py
Process 0 received ['0A', '1A', '2A', '3A']
Process 2 received ['0C', '1C', '2C', '3C']
Process 3 received ['0D', '1D', '2D', '3D']
Process 1 received ['0B', '1B', '2B', '3B']
```

Результат можно увидеть на рис. 3.8.

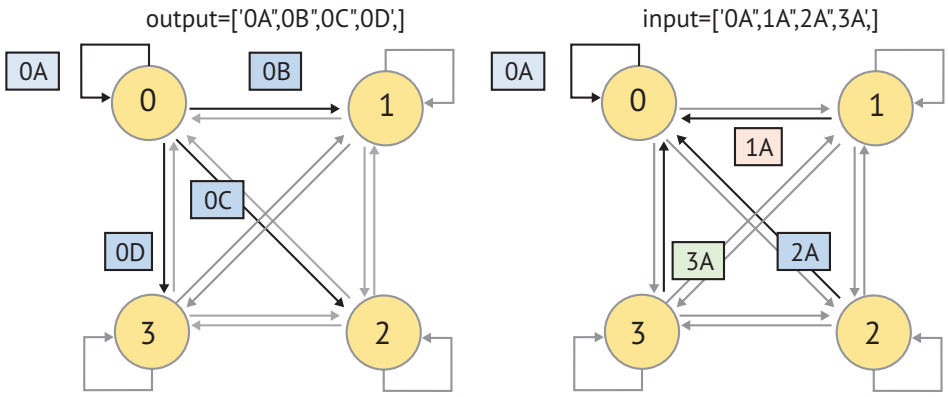


Рис. 3.8. AllToAll для процесса 0

Операция свертки

Операция свертки, или редукции (reduction operation), выполняемая `comm.reduce()`, принимает массив входных элементов от каждого процесса и возвращает массив выходных элементов корневому процессу. Выходные элементы содержат свернутый (агрегированный) результат.

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.rank

if rank == 0:
  output = np.array([0,0,0,0])
if rank == 1:
  output = np.array([1,1,1,1])
if rank == 2:
  output = np.array([2,2,2,2])
if rank == 3:
  output = np.array([3,3,3,3])

print("Process %d. Sending %s" %(rank,output))
input = comm.reduce(output, root=0, op=MPI.SUM)
if rank == 0:
  print("The result of the parallel computation is %s" %(input))
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 4 python mpi08a.py
Process 3. Sending [3 3 3 3]
Process 2. Sending [2 2 2 2]
```

```
Process 1. Sending [1 1 1 1]
Process 0. Sending [0 0 0 0]
The result of the parallel computation is [6 6 6 6]
```

При свертке доступны различные операции. Полный список этих операций приведен в табл. 3.2.

Таблица 3.2. Список операций свертки

SUM	Сумма элементов
PROD	Умножение элементов
MAX	Максимальное значение элементов
MAXLOC	Максимальное значение и ранг связанного процесса
MIN	Минимальное значение элементов
MINLOC	Минимальное значение и ранг связанного процесса
LAND	Логическое И по элементам
LOR	Логическое ИЛИ по элементам
BAND	Побитовое И по элементам
BOR	Побитовое ИЛИ по элементам

Оптимизация коммуникаций при помощи топологий

Интересной особенностью MPI является возможность вносить изменения в виртуальные топологии. Все взаимодействие между процессами происходит через коммуникационные группы. До сих пор при использовании **MPI_COMM_WORLD** все процессы назначались в группу, и каждому из них присваивался номер ранга от 0 до $n - 1$, где n – это количество процессов, запущенных в параллельном выполнении.

Библиотека `mpi4py` позволяет нам определить виртуальную топологию для этого коммуникатора. Это означает, что способ присвоения рангов можно изменить. При определении виртуальной топологии каждый узел (процесс) сможет обмениваться данными только с соседними узлами внутри этой виртуальной топологии. Благодаря этому сократится время выполнения, что повысит производительность. Дело в том, что во многих сложных случаях ранги назначаются различным процессам случайным образом, и сообщения могут проходить через множество процессов, прежде чем достигнут правильного адресата, а именно процесса, для которого это сообщение было предназначено.

MPI предоставляет два способа построения топологии:

- декартова топология;
- матрица смежности.

В первом случае создается декартова топология, во втором необходимо использовать матрицу смежности для создаваемого графа.

В этом примере мы будем использовать только декартову топологию, с помощью которой можно построить многие широко используемые топологии, в частности кольца и тороиды.

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank

comm_3D = comm.Create_cart(dims = [3,3,3],
                           periods=[False,False,False],
                           reorder=False)
xyz = comm_3D.Get_coords(rank)
print ("In this 3D topology, process %s has coordinates %s " %(rank,xyz))
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 27 python mpi09.py
In this 3D topology, process 0 has coordinates [0, 0, 0]
In this 3D topology, process 1 has coordinates [0, 0, 1]
In this 3D topology, process 4 has coordinates [0, 1, 1]
In this 3D topology, process 6 has coordinates [0, 2, 0]
In this 3D topology, process 7 has coordinates [0, 2, 1]
In this 3D topology, process 8 has coordinates [0, 2, 2]
In this 3D topology, process 12 has coordinates [1, 1, 0]
In this 3D topology, process 10 has coordinates [1, 0, 1]
In this 3D topology, process 9 has coordinates [1, 0, 0]
In this 3D topology, process 11 has coordinates [1, 0, 2]
In this 3D topology, process 13 has coordinates [1, 1, 1]
In this 3D topology, process 14 has coordinates [1, 1, 2]
In this 3D topology, process 15 has coordinates [1, 2, 0]
In this 3D topology, process 16 has coordinates [1, 2, 1]
In this 3D topology, process 17 has coordinates [1, 2, 2]
In this 3D topology, process 20 has coordinates [2, 0, 2]
In this 3D topology, process 5 has coordinates [0, 1, 2]
In this 3D topology, process 2 has coordinates [0, 0, 2]
In this 3D topology, process 18 has coordinates [2, 0, 0]
In this 3D topology, process 24 has coordinates [2, 2, 0]
In this 3D topology, process 26 has coordinates [2, 2, 2]
In this 3D topology, process 21 has coordinates [2, 1, 0]
In this 3D topology, process 3 has coordinates [0, 1, 0]
In this 3D topology, process 22 has coordinates [2, 1, 1]
In this 3D topology, process 19 has coordinates [2, 0, 1]
In this 3D topology, process 25 has coordinates [2, 2, 1]
In this 3D topology, process 23 has coordinates [2, 1, 2]
```

Этот результат позволяет нам изобразить типологию графически, как показано на рис. 3.9.

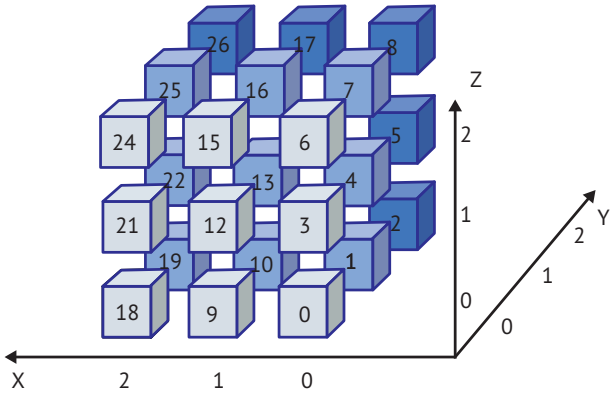


Рис. 3.9. Декартово ранжирование трехмерной топологии

Для любого из процессов можно определять соседние процессы с помощью следующей функции:

```
shift(direction,displacement)
```

Эта функция возвращает два целых числа, которые соответствуют рангам двух соседних процессов на одной из декартовых осей. Аргумент direction задает направление, в котором необходимо определить соседние процессы, в соответствии с декартовыми осями. Диапазон значений аргумента direction составляет [0, *n* – 1], где *n* – это размер декартовой сетки. Поэтому в нашем случае **0** будет соответствовать Z, **1** будет соответствовать X, а **2** будет соответствовать Y (исходя из приведенного выше рисунка). Другой аргумент задает направление перемещения.

```
displacement (> 0: upwards shift, < 0: downwards shift) (integer)
```

Однако в нашем случае, когда мы хотим определить соседние процессы, это значение всегда будет равно **1**. Продолжая рассматривать этот пример, для определения соседей процесса 12 необходимо добавить в код следующие строки:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
comm_3D = comm.Create_cart(dims = [3,3,3],
                           periods =[False,False,False],
                           reorder=False)
xyz = comm_3D.Get_coords(rank)

if rank==12:
  print ("In this 3D topology, process %s has coordinates %s " %(rank,xyz))
  right,left = comm_3D.Shift(0,1)
  up,down = comm_3D.Shift(1,1)
```

```
forward,backward = comm_3D.Shift(2,1)
print("Neighbors (left-right): %s %s" %(left,right))
print("Neighbors (up-down): %s %s" %(up,down))
print("Neighbors (forward-backward): %s %s" %(forward,backward))
```

Запуск этого кода дает следующий результат:

```
$mpiexec -n 27 python mpi09b.py
In this 3D topology, process 12 has coordinates [1, 1, 0]
Neighbors (left-right): 21 3
Neighbors (up-down): 9 15
Neighbors (forward-backward): -1 13
```

Таким образом, если во время наших параллельных вычислений требуется обмен информацией только между топологически смежными процессами, мы можем напрямую вставить их ранг в код, не используя коллективные формы коммуникации, с вложенными if-операторами для выбора между различными процессами, чтобы выбрать только необходимые.

Следовательно, мы можем напрямую использовать методы

- `comm.send(data, neighbor)`,
- `comm.recv(neighbor)`

для прямого и эффективного отправления и получения сообщений между смежными процессами.

Заключение

После изучения этой главы вы получили все основные инструменты для параллельного программирования. Мы рассмотрели концепции, связанные с механизмами родительских и дочерних процессов в модуле многопроцессорной обработки, и проиллюстрировали их рядом примеров. Вы также научились применять мощные возможности функции `map()` в параллельных системах, что позволяет значительно повысить эффективность ваших программ. В то же время мы рассмотрели библиотеку `mpi4py` на основе стандарта MPI, которая позволяет запускать определенный набор родительских процессов в параллельном режиме и использует парадигму обмена сообщениями для обработки данных. Однако в следующих главах мы подробно рассмотрим более сложные концепции параллельного программирования, а также несколько библиотек, которые еще больше раскрывают этот потенциал.

Список литературы

- *https://mpi4py.readthedocs.io/en/stable/*.
- *https://docs.python.org/3/library/multiprocessing.html*.
- *https://github.com/PowerShell/PowerShell/releases/tag/v7.2.6*.
- *https://superfastpython.com/multiprocessing-pipe-in-python/*.

Глава 4

Асинхронное программирование с AsyncIO

Во второй главе книги было рассказано о параллельном программировании на основе потоков и, в частности, о модуле threading. Ограничения, налагаемые GIL, создают серьезные проблемы, которые необходимо преодолеть при работе с потоками в Python. Эта особенность Python накладывает ограничения на режим многопоточности, что представляет собой неоспоримый недостаток по сравнению с параллельным программированием. Следовательно, многопоточное программирование на Python становится незаменимым при управлении задачами, связанными с вводом-выводом, где невозможно предсказать время ожидания при выполнении других задач. В этой главе мы вернемся к этому типу программирования, но с использованием совершенно другого подхода: **асинхронной модели** и **библиотеки AsyncIO**.

Структура

В этой главе будут рассмотрены следующие темы:

- асинхронное и синхронное программирование;
- библиотека AsyncIO;
- синтаксис async/await;
- сопрограммы;
- задачи;
- футуры;
- цикл событий;
- концепция async for;
- очереди.

Асинхронное и синхронное программирование

В мире программирования существуют две значительно отличающиеся модели:

- синхронная модель (также называемая **sync**);
- асинхронная модель (также называемая **async**).

Асинхронное программирование, которое является темой этой главы, основано на модели, где несколько объектов действуют одновременно. Этими объектами могут быть потоки, процессы или что-либо еще, что может действовать одновременно и выполняться в любом порядке, независимо друг от друга. Асинхронная модель является неблокирующей архитектурой, что означает, что в программах, созданных с использованием этой модели, нет прерываний выполнения, даже если одна или несколько операций в ней ожидают какого-либо события или обслуживания.

В частности, с помощью асинхронного программирования можно одновременно обрабатывать несколько операций, каждая из которых имеет свою конкретную задачу и выполняется независимо от других. Их выполнение не связано с выполнением других операций, и все они стараются одновременно выполнить свою задачу (конкуренция). Кроме того, процесс выполнения каждого компонента программы можно контролировать таким образом, чтобы активировать его только тогда, когда это удобно или возможно. Это позволяет управлять несколькими запросами одновременно, с максимальной эффективностью используя как ресурсы, так и время выполнения. В качестве примера можно привести серверные приложения, к которым обращаются несколько клиентов. Каждый из них запрашивает у сервера услугу посредством сообщения, а затем сервер управляет процессом обработки сообщений и отправкой ответов клиенту. Этот подход не является обязательным, но считается предпочтительным с учетом времени ожидания при работе с данными в сети.

Синхронное программирование, в свою очередь, является наиболее знакомым нам типом кода. Оно основано на модели зачастую с одним исполняющим объектом (один поток в одном процессе), где код представляет собой не что иное, как последовательность выполняемых операций. На самом деле синхронные серверные приложения обычно используют один поток на каждый запрос. Выполнение этих операций взаимозависимо. Только после завершения одной операции выполнение переходит к следующей. Синхронная модель представляет собой блокирующую архитектуру, а это значит, что программы, созданные с использованием этой модели, характеризуются строгой последовательностью выполнения, и, если какая-либо операция блокируется в ожидании ответа, вся программа дает сбой.

Плюсы и минусы асинхронного и синхронного программирования

Асинхронное программирование, при условии грамотного проектирования для правильного и последовательного выполнения, является более плавным и динамически более восприимчивым к внешним событиям, таким как действия пользователя или внешний сервис предоставления данных и т. д. Этот тип программирования в основном опирается на принцип конкуренции (даже если это не одно и то же), и поэтому использующие его программы могут быть более быстрыми, производительными и эффективными, значительно сокращая время выполнения кода при ожидании ответов от внешних сервисов.

Наиболее очевидным примером такого программирования является разработка графических интерфейсов пользователя (graphical user interfaces, GUI). Благодаря различным обработчикам событий, которые кажутся работающими одновременно (но на самом деле это не так), пользователь видит динамичное функционирование без задержек между нажатиями кнопок, взаимодействием с текстом, появлением и исчезновением раскрывающихся меню.

В свою очередь, синхронное программирование следует заранее определенному порядку выполняемых действий, где задачи обычно выполняются последовательно, а не одновременно, и каждая задача дожидается выполнения предыдущей. Даже если в программе есть параллельные задачи, все они выполняют одну и ту же функцию, и все они должны быть выполнены для продолжения выполнения программы. Это программирование используется в тех случаях, когда для выполнения задачи требуется результат предыдущей задачи, и так далее по всей программе.

Преимущество синхронного программирования заключается в том, что этот тип программирования практически не вызывает проблем во время выполнения. Поведение программы легко предсказуемо, а ее реализация генерирует простой для создания и чтения код. Напротив, асинхронное программирование может быть очень сложным как в реализации кода, так и в его правильном прочтении. Такая программа может вызывать непредсказуемое и аномальное поведение, которое трудно предугадать на этапе разработки и еще труднее исправить. Поэтому асинхронное программирование требует больших затрат на обслуживание.

Конкурентное программирование и асинхронная модель

Концепции, лежащие в основе многопоточного (конкурентного – concurrent) программирования, уже были достаточно подробно рассмотрены в предыдущих главах. В отличие от параллельного программирования этот режим может работать как в однопроцессорных, так и в многопроцессорных системах. Поскольку ранее мы уже сравнивали асинхронные и синхронные модели, давайте теперь сравним их с многопоточным программированием.

Асинхронная модель имеет больше ограничений, чем модель, используемая для потоков, поскольку ее работа ограничена единственным потоком, исполняемым на одноядерном процессоре, а не несколькими потоками одновременно. В большинстве асинхронных моделей для операций используются фоновые рабочие потоки, которые просто не упоминаются в основной технической документации. В потоковой модели различные задачи распределяются между несколькими потоками, которые одновременно (но не параллельно) конкурируют за одни и те же общие ресурсы и время выполнения. В асинхронной модели, с другой стороны, все задачи привязаны к одному потоку и выполняются поочередно в едином потоке выполнения, асинхронно. В любой момент выполнение каждой задачи может быть приостановлено для освобождения места для других ожидающих задач, а по завершении выполнения других задач – возобновлено, и т. д., пока все операции не будут завершены, как показано на рис. 4.1.

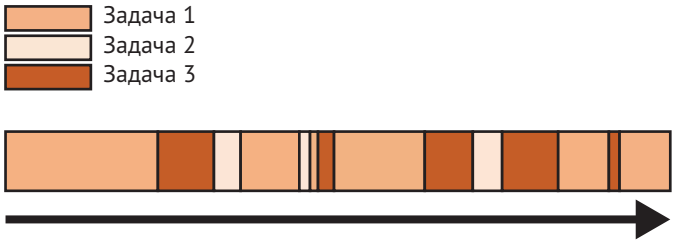


Рис. 4.1. Асинхронное выполнение

Асинхронная модель в Python зарекомендовала себя как гораздо более эффективная и безопасная, чем модель, используемая с потоками, и в последние годы она занимает все большее место в программировании, до такой степени, что в многопоточном программировании полностью заменяет модуль потоков во всех случаях, когда необходимо управлять задачами ввода-вывода. Кроме того, она идеально подходит для работы в полной синергии с параллельным программированием, поскольку последнее, в свою очередь, эффективнее всего справляется с управлением задачами с использованием ресурсов центрального процессора. Такое взаимное сочетание двух программ нередко приводит к необходимости использования обеих.

Библиотека AsyncIO

В языке Python асинхронная модель поддерживается библиотекой AsyncIO. Хотя в настоящее время разрабатываются и другие альтернативы, библиотека AsyncIO по-прежнему остается эталоном для параллельного программирования в асинхронном режиме.

В этой главе мы познакомимся с концепциями, лежащими в основе асинхронной модели, используя только библиотеку AsyncIO, где эти концепции представлены в виде компонентов. Это позволит вам ознакомиться с асинхронной моделью на конкретных примерах.

Основные компоненты модуля:

- сопрограмма;
- задача;
- футура;
- цикл событий.

Синтаксис async/await

В последние годы библиотека AsyncIO претерпела множество изменений, при этом использовались самые разные шаблоны программирования. В интернете нередко можно наткнуться на код с устаревшими структурами, связанными со старыми ключевыми словами и шаблонами (такими как `@asyncio.coroutine` и т. д.). В настоящее время используется синтаксис, основанный на ключевых словах `async-await`.

Рассмотрим очень простой пример реализации кода с помощью библиотеки `asyncio` в соответствии с этим синтаксисом.

```
import asyncio

async def main():
    print('Awaiting for ...')
    await asyncio.sleep(1)
    print('... AsyncIO!')

asyncio.run(main())
```

Как можно видеть, код представляет собой то, что мы рассматриваем как определение функции, которую мы назвали `main()`, с префиксом, являющимся ключевым словом `async`. В данном случае мы определили не функцию, а сопрограмму (coroutine), которую можно рассматривать как особый элемент модели асинхронного программирования.

Затем внутри кода мы находим вызов `asyncio.sleep()` (это тоже не функция, а сопрограмма), предваряемый ключевым словом `await`. Это указывает на то, что вызываемый объект будет включен в график параллельного выполнения. Выполнение этого механизма будет подвержено прерываниям в ожидании выполнения других подобных объектов, а затем возобновится после того, как эти объекты завершат свою задачу.

Сопрограмма `asyncio.sleep()` работает так же, как функция `time.sleep()`, с задержкой выполнения на определенное количество секунд, что задается в качестве параметра. Разница заключается в том, что, будучи сопрограммой, она работает корректно в контексте асинхронного выполнения.

В библиотеке AsyncIO такие объекты, как сопрограммы, задачи (tasks) и операторы будущего, или футуры (futures), классифицируются как объекты, ожидающие выполнения (awaitable objects). Все эти объекты после определения и создания в коде могут затем быть запланированы для совместного выполнения с помощью префикса await.

Причина, по которой в Python было решено добавить синтаксис `async-await`, заключается в необходимости устранить любые формы неоднозначности в коде. Как мы уже видели, `async` позволяет быстро отличать сопрограммы от функций, а `await` дает возможность быстро распознавать вызовы, которые следует планировать асинхронно.

Сопрограммы

Центральным элементом асинхронной модели являются сопрограммы. **Сопрограмма** (coroutine) – это специализированная версия генераторной функции в Python.

Обратите внимание: генераторные функции.

В Python генераторная функция (generator function) представляет собой функцию, которая возвращает итератор.

```
def generator():
  yield "A"
  yield "B"
  yield "C"
```

Вместо `return` генераторы используют оператор `yield`. В этом случае генератор при вызове функции `next()` будет возвращать по одному элементу при каждой итерации.

```
iterator = generator()
print("First call: %s" %next(iterator))
print("Second call: %s" %next(iterator))
print("Third call: %s" %next(iterator))
```

При запуске кода вы получите следующий результат:

```
First call: A
Second call: B
Third call: C
```

Как видно из результата, при каждом вызове мы будем получать разный результат, а именно соответствующее значение в последовательности итераций, определенной `yield` в генераторной функции.

Сопрограмма – это функция, которая может приостановить свое выполнение до достижения return и на некоторое время передать управление другой сопрограмме.

В AsyncIO мы можем определить функцию как сопрограмму с помощью ключевого слова `async`. В приведенном выше примере сопрограммой является функция `main()`, определение которой предваряется `async`:

```
async def main():
    print('Awaiting for ...')
    await asyncio.sleep(1)
    print('... AsyncIO!')
```

Попробуйте вызвать сопрограмму саму по себе, как любую другую функцию:

```
>>> main()
```

Вы не увидите никаких результатов: на самом деле она не будет выполнена.

На самом деле сопрограммы – это не простые функции, и для их выполнения библиотека AsyncIO предоставляет три разных способа.

- Первый способ – это именно тот, который показан в примере. Функция `asyncio.run()` принимает в качестве аргумента сопрограмму, которая будет считаться точкой входа при выполнении программы. Чтобы выполнить предыдущую сопрограмму, мы можем запустить следующую инструкцию:

  ```
  >>> asyncio.run(main())
  ```

- Другой способ выполнения сопрограммы – ее запуск через другую сопрограмму. Для примера определим вторую сопрограмму, также с префиксом `async`:

```
async def other():
  print("I am a coroutine")
```

В сопрограмме `main()` вызов новой сопрограммы для выполнения помещается после `await`:

```
import asyncio

async def other():
  print("I am a coroutine")

async def main():
    print('Awaiting for ...')
    await asyncio.sleep(1)
    await other()
    print('... AsyncIO!')

asyncio.run(main())
```

Ее исполнение даст следующий результат:

```
Awaiting for ... I am a coroutine
... AsyncIO!
```

- Существует также третий метод, в котором используется класс `task` в AsyncIO. Вызывая функцию `asyncio.create_task()`, мы создаем новый объект класса `Task`, который будет выполнять сопрограмму многопоточно.

```
import asyncio

async def other():
  print("I am a coroutine")

async def main():
    task = asyncio.create_task(other())
    print('Awaiting for ...')
    await asyncio.sleep(1)
    await task
    print('... AsyncIO!')

asyncio.run(main())
```

При выполнении кода будет получен тот же результат, что и ранее. В последнем случае задачи выполняются в заранее заданном порядке, что будет рассмотрено в следующем разделе.

Задача

Задачи (tasks) в AsyncIO дают возможность планировать работу сопрограмм в порядке конкуренции (в заданном порядке). Сопрограммы могут быть обернуты в объекты класса `Task` с помощью функции `asyncio.create_task()`, как было показано в предыдущем коде. Созданные таким образом задачи затем вызываются с помощью await в сопрограмме `main()` и выполняются в конкурирующем режиме. Рассмотрим это на практическом примере. Изменим предыдущий код таким образом, чтобы создать сопрограмму с двумя параметрами: `id` – это целое число, которое мы будем использовать в качестве идентификатора сопрограммы, и другое целое число в качестве индикатора времени выполнения сопрограммы, которое мы будем моделировать с помощью следующей функции:

```
asyncio.sleep().
import asyncio

async def other(id,t):
  await asyncio.sleep(t)
  print("I am a coroutine %s" %id)

async def main():
    task1 = asyncio.create_task(other(1,10))
    task2 = asyncio.create_task(other(2,4))
    task3 = asyncio.create_task(other(3,1))
    await task1
    await task2
    await task3

asyncio.run(main())
```

Запуск этого кода даст следующий результат:

```
I am a coroutine 3
I am a coroutine 2
I am a coroutine 1
```

Как видно из результата, три сопрограммы выполнялись одновременно, и поскольку на выполнение `coroutine 3` ушло меньше всего времени, она и закончила работу раньше всех, не дожидаясь результатов других сопрограмм.

Если бы мы использовали сопрограммы без задач, никакого многопоточного выполнения бы не произошло. Ниже приведена предыдущая версия кода, написанная с использованием только сопрограмм:

```
import asyncio

async def other(id,t):
  await asyncio.sleep(t)
  print("I am a coroutine %s" %id)
```

```
async def main():
    t1 = time.perf_counter()
    await other(10,1)
    await other(4,2)
    await other(3,3)
    t2 = time.perf_counter()
    elapsed_time = t2 - t1
    print("Elapsed time %s" %elapsed_time)

asyncio.run(main())
```

Запуск этого кода даст следующий результат:

```
I am a coroutine 1
I am a coroutine 2
I am a coroutine 3
Elapsed time 17.021660084999894
```

Как видно из результата, выполнение не является одновременным (конкурентным), поскольку сопрограммы завершают свою задачу в той же последовательности, в которой они были запущены, даже если первая сопрограмма занимает гораздо больше времени, чем остальные. Общее время выполнения равно сумме времени выполнения трех сопрограмм (10 + 4 + 3 = 17).

Если измерить время выполнения версии кода с задачами, то получим следующее:

```
I am a coroutine 3
I am a coroutine 2
I am a coroutine 1
Elapsed time 10.006599820999327
```

Как можно видеть, время выполнения почти совпадает со временем выполнения самой медленной сопрограммой. В действительности преимущество конкурентного программирования заключается в том, чтобы использовать время простоя выполнения, останавливая медленные сопрограммы и позволяя в это время работать более быстрым. Этот код является прекрасным примером такого поведения.

Продолжим работу с задачами и внесем изменения в приведенный выше код. Созданные в программе задачи могут иметь идентифицирующее имя. Его можно присвоить во время создания задачи с помощью опционального параметра `name`:

```
asyncio.create_task(coroutine,name="..")
```

Или позже с помощью метода `set_name()`:

```
task.set_name('...')
```

В сопрограмме `other()` мы использовали входной параметр для идентификации задачи, в которую она завернута. Этого можно избежать, поскольку можно

отслеживать актуальную задачу в сопрограмме без необходимости передавать внешние параметры. Функция `asyncio.current_task()` возвращает актуальную задачу. Поскольку нам нужно только имя задачи, мы можем вызвать метод `get_name()`.Тогда ранее приведенный код можно изменить следующим образом:

```
import asyncio

async def other(t):
  await asyncio.sleep(t)
  id = asyncio.current_task().get_name()
  print("I am a coroutine %s" %id)

async def main():
    task1 = asyncio.create_task(other(10),name="1")
    task2 = asyncio.create_task(other(4),name="2")
    task3 = asyncio.create_task(other(3),name="3")
    await task1
    await task2
    await task3

asyncio.run(main())
```

Запуск этого кода даст следующий результат:

```
I am a coroutine 3
I am a coroutine 2
I am a coroutine 1
```

Как видно из результата, использование задач вместо сопрограмм демонстрирует наличие многопоточности. В действительности первой будет выполнена третья сопрограмма, и на ее выполнение потребуется всего 2 с. Первая сопрограмма, требующая больше времени (10 с), будет завершена последней.

Сбор объектов awaitable для параллельного выполнения

Еще более явным способом запуска группы ожидающих выполнения (awaitable) объектов, таких как сопрограммы, задачи и футуры, является функция `asyncio.gather()`. Эта функция принимает в качестве параметров последовательность объектов awaitable. Если в последовательности есть сопрограммы, они будут автоматически запланированы как задачи. Порядок выполнения, даже если оно будет многопоточным, соответствует порядку последовательности. Когда все ожидающие выполнения объекты будут завершены, функция `asyncio.gather()` вернет агрегированный список значений, порядок которых соответствует порядку последовательности ожидающих выполнения объектов, которые были переданы в качестве параметра.

```
import asyncio
async def coroutine(t,id):
    await asyncio.sleep(t)
```

```
    print("I am the coroutine %s" %id)
    return t+2

async def main():
    results = await asyncio.gather(
        coroutine(10,"A"),
        coroutine(4,"B"),
        coroutine(2,"C"),
    )
    print("The results are: %s" %results)

asyncio.run(main())
```

Запуск этого кода даст следующий результат:

```
I am the coroutine C
I am the coroutine B
I am the coroutine A
The results are: [12, 6, 4]
```

Как видно из результата, при использовании сопрограмм вместо задач получилось одновременно выполняемое поведение. В действительности они были переданы функции `asyncio.gather()`, что привело к их неявной конвертации, и код стал гораздо более читабельным и интуитивно понятным. Вот как бы выглядел соответствующий код, если бы были использованы задачи:

```
async def main():
    task1 = asyncio.create_task(coroutine(10,"A"))
    task2 = asyncio.create_task(coroutine(4,"B"))
    task3 = asyncio.create_task(coroutine(2,"C"))
    r1 = await task1
    r2 = await task2
    r3 = await task3
    results = [ r1, r2, r3]
    print("The results are: %s" %results)

asyncio.run(main())
```

Футура

Еще одним концептом, связанным с асинхронным программированием, является использование операторов будущего, или футуров (futures), которые реализованы в модуле `asyncio` классом `Future`. По своей сути оператор `future` концептуально представляет собой результат (включая исключение), который еще не доступен на момент его обработки.

Рассмотрим следующий код:

```
import asyncio

async def get_result(result):
```

```
    await asyncio.sleep(5)
    result = '...an awaited result'

async def main():
    my_result = ' '
    task1 = asyncio.create_task(get_result(my_result))
    await task1

    print("I'm waiting for ...")
    print(my_result)

asyncio.run(main())
```

Запуск этого кода даст следующий результат:

```
I'm waiting for ...
```

Как можно видеть, ничего не было выведено на печать: программа завершила свое выполнение в ожидании результата.

Именно в таких случаях используется `Future`. Сделав соответствующие замены в предыдущем коде, мы получим следующий код:

```
import asyncio

async def get_result(future):
    await asyncio.sleep(5)
    future.set_result('...a future result')

async def main():
    my_future = asyncio.Future()
    task1 = asyncio.create_task(get_result(my_future))
    await task1

    print("I'm waiting for ...")
    print(await my_future)
    print('Before continuing with my execution')

asyncio.run(main())
```

Запуск этой программы дает следующий результат:

```
I'm waiting for ...
...a future result
Before continuing with my execution
```

Как видно из результата, на этот раз программа дождалась генерации результата (5 с), а затем вывела на экран всю информацию.

При использовании футуры вместе с `await` программе придется дождаться, пока сопрограмма `get_result()` сгенерирует результат (собственно функцию будущего), и только после этого она сможет продолжить свое выполнение.

Таким образом, механизм Future основан на функциях обратного вызова, таких как get_result() в предыдущем примере, задача которых состоит в обработке любого результата и присвоении его экземпляру класса Future() с помощью метода set_result().

Обратите внимание: Функции обратного вызова

Обратный вызов (callback) – это функция, которая передается в качестве аргумента другой функции.

```
def callbackFunc():
    #Делает что-либо
def mainFunc(callbackFunc):
     ...
    callbackFunc()
    ...
```

Экземпляр класса Future создается с помощью его конструктора:

```
future = asyncio.Future()
```

Новый экземпляр передается в качестве аргумента в функцию обратного вызова, которая его оценивает. Когда содержимое future необходимо использовать, оно вставляется в код с префиксом await, чтобы попасть в механизм планирования AsyncIO. Таким образом, все выполнение программы будет скорректировано в соответствии с объектами, ожидающими выполнения (awaitable), которые будут учитываться в соответствии с их временем выполнения, асинхронно. Таким образом, программа будет ждать все время, необходимое для оценки future, выполняя в то же время всю возможную работу в параллельном режиме, а затем продолжит поток выполнения.

Таким образом, Future также является объектом awaitable. В то же время future – это результат выполнения процесса (функции обратного вызова), статус выполнения которого неизвестен, но который требует внимания до его завершения (в будущем). Следовательно, при вызове future с await уже предполагается, что на этом этапе программы результат еще не будет доступен. В этом случае могут возникнуть следующие ситуации:

- процесс, который представляет future, завершился, и результат уже введен для футуры. В этом случае оператор await немедленно возвращает это значение;
- процесс, который представляет future, завершился с ошибкой, вернув исключение. В этом случае оператор await немедленно генерирует это исключение;
- процесс, который представляет future, все еще выполняется, и функция будущего еще не оценена. В этом случае все остальные задачи ожида-

ют завершения процесса и оценки `future`. Как только статус `future` будет завершен, оператор `await` вернет результат или вызовет исключение, в зависимости от ситуации, и выполнение программы будет продолжено.

Различие между сопрограммой (Coroutine) и футурой (Future) очень важно. Код сопрограммы не будет выполняться, пока он находится в состоянии ожидания. Футура представляет собой нечто, что в любом случае выполняется, и просто дает вашему коду возможность дождаться ее завершения, проверить, выполнена ли она, а затем передать полученный результат.

Обратите внимание: `asyncio.futures` и `concurrent.futures`. Несмотря на сходство, необходимо быть очень осторожным, чтобы не перепутать эти два типа классов `Future`. Футуры модуля `concurrent` основаны на многопоточном конструкторе, в то время как футуры `asyncio` являются исключительно однопоточными. Таким образом, хотя семантически они выполняют одинаковые задачи, но работают совершенно по-разному и не являются взаимозаменяемыми.

Цикл событий

В плане событий в моделях программирования мы имеем дело с тремя концепциями:

- источником событий (event source),
- обработчиком событий (event handler),
- циклом событий (event loop).

В вычислительной системе источник событий определяется как объект, генерирующий события. Последний управляется другим объектом, определяемым как обработчик событий. Цикл событий действует как связующее звено между источниками событий и обработчиком событий, см. рис. 4.2.



Рис. 4.2. Цикл событий

В действительности события генерируются рандомно и асинхронно, поэтому прямая связь между генерацией событий и управлением ими возникает очень редко. Между этими двумя операциями всегда есть период време-

ни (асинхронность), и в течение этого времени могут генерироваться новые управляемые события.

С точки зрения программирования библиотека AsyncIO реализует цикл событий через класс `Loop` с помощью серии методов, которые позволяют им управлять:

- `asyncio.new_event_loop()`: создает и возвращает новый цикл событий;
- `asyncio.get_event_loop()`: возвращает цикл событий текущего контекста;
- `asyncio.set_event_loop()`: устанавливает цикл событий для текущего контекста;
- `Loop.call_later()`: планирует обратный вызов, который будет выполнен после заданной задержки в секундах;
- `Loop.call_soon()`: планирует обратный вызов, который будет выполнен при следующей итерации цикла событий;
- `Loop.run_until_complete()`: работает до тех пор, пока сопрограмма (переданная в качестве аргумента) не будет завершена;
- `Loop.run_forever()`: запускает цикл событий до вызова `stop()`;
- `Loop.stop()`: останавливает цикл событий;
- `Loop.close()`: закрывает цикл событий.

Цикл событий является неотъемлемым объектом в механизме асинхронного выполнения `asyncio` и активируется неявным образом при запуске команды.

```
asyncio.run(main())
```

Цикл событий можно рассматривать как цикл `while True`, созданный для отслеживания статуса выполнения сопрограмм, которые участвуют в асинхронном планировании через конструкцию `async` / `await`. Таким образом, цикл событий в любой момент выполнения программы сможет определить, какие сопрограммы находятся в состоянии простоя и одновременно какие задачи необходимо выполнить в это время. Поэтому именно цикл событий выводит сопрограмму из состояния простоя и заставляет ее выполнять свои операции. Когда каждая из сопрограмм завершается, цикл событий помечает соответствующую задачу как выполненную. Наконец, когда все задачи завершены, цикл событий закрывается.

Таким образом, цикл событий можно считать главным диспетчером асинхронных операций библиотеки AsyncIO. Обычно цикл событий запускается, выполняет свои операции и закрывается в неявной форме. В действительности мы не видели никаких упоминаний о нем в предыдущих примерах с сопрограммами, задачами и футурами.

Однако также существует возможность явного управления циклом событий в коде. Типичный сценарий выглядит следующим образом:

```
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
```

```
finally:
    loop.close()
```

Это полностью эквивалентно команде `asyncio.run(main())`, как можно увидеть непосредственно в следующем коде:

```
import asyncio

async def main():
  print('Starting...')
  await asyncio.sleep(10)
  print('...Ending')

#asyncio.run(main())
loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(main())
finally:
    loop.close()
```

В коде можно увидеть, как сначала используется функция `asyncio.get_event_loop()` для присвоения текущего цикла событий ссылочной переменной цикла. Это позволяет явно контролировать рабочий процесс цикла событий. В конструкции `try-finally` мы запускаем планирование и выполнение сопрограммы `main()` и одновременно все объекты awaitable, вызываемые в ней. По завершении всех запланированных асинхронных операций или в случае возникновения исключений цикл событий должен быть закрыт с помощью метода `loop.close()` для высвобождения ресурсов.

Это самый простой и понятный случай, но можно реализовать гораздо более сложные сценарии, когда требуется более тщательный контроль за работой цикла. Существует целое множество методов, которые регулируют его работу или позволяют ее контролировать. Например, методы `loop.is_running()` и `loop.is_closed()` дают возможность проверить статус выполнения цикла событий.

Однако следует отметить, что во многих случаях необходимость явного указания цикла событий в нашем коде просто не возникает.

Асинхронные итерации с async for и без async for

Концепция `async for` позволяет выполнять итерации с помощью асинхронного итератора, цель которого заключается в обеспечении возможности выполнения асинхронных вызовов на каждом этапе итерации. Расширением этой концепции является асинхронный генератор, сопрограмма, в которой можно использовать `await` и `yield` вместе:

```
async def gen(n):
  for i in range(n):
```

```
    await asyncio.sleep(1)
    yield i
```

Этот асинхронный генератор можно использовать в качестве асинхронного итератора в `async for`, как в приведенном ниже коде.

```
import asyncio

async def gen(n):
  for i in range(n):
    await asyncio.sleep(1)
    yield i
async def main():
    async for i in gen(10):
      print(i)

asyncio.run(main())
```

Запуск этого кода даст следующий результат:

```
0
1
2
3
4
5
6
7
8
9
```

Тем не менее, как видно из результата, использование асинхронных генераторов не позволяет сделать итерацию многопоточной. Результаты различных шагов отображаются в той же последовательности итераций. Таким образом, Async for на данный момент не позволяет реализовать многопоточное выполнение. Все, что он предоставляет, – это возможность оставить цикл событий под контролем для какой-либо другой работающей сопрограммы.

Если же мы хотим получить итерацию, в которой различные шаги конкурируют друг с другом, нам понадобится другой способ. Функция `asyncio.as_completed()` аналогично тому, что мы видели с `asyncio.gather()`, принимает последовательность `awaitable`. Эта функция возвращает итератор сопрограмм, которые будут действовать в конкуренции друг с другом.

Мы можем увидеть это на простом примере, воплощенном в следующем коде:

```
import asyncio

async def f(i):
```

```
    print('start iteration step %s' %i)
    await asyncio.sleep(1)
    print('end iteration step %s' %i)
    return i

async def main():
  for j in asyncio.as_completed([f(i) for i in range(10)]):
      result = await j
      print('result received: %s' %result)

asyncio.run(main())
```

Как видно из кода, мы передаем последовательность из десяти сопрограмм `f(i)`, от `f(0)` до `f(9)`, в функцию `asyncio.as_completed()`. Это вернет итератор, шаги которого будут представлять сопрограммы в порядке завершения (а не в порядке поступления последовательности). Сопрограмма последовательности, которая закончится первой, будет первым шагом итерации, и ее результат будет первым, который будет собран и отображен в выводе. Затем наступит очередь той, которая закончится сразу после нее, и т. д. до последней.

Выполнив приведенный выше код, вы получите результат, который полностью отражает только что описанное поведение.

```
start iteration step 2
start iteration step 0
start iteration step 6
start iteration step 9
start iteration step 3
start iteration step 7
start iteration step 4
start iteration step 8
start iteration step 5
start iteration step 1
end iteration step 2
end iteration step 0
end iteration step 6
end iteration step 9
end iteration step 3
end iteration step 7
end iteration step 4
end iteration step 8
end iteration step 5
end iteration step 1
result received: 2
result received: 0
result received: 6
result received: 9
result received: 3
```

```
result received: 7
result received: 4
result received: 8
```

Как можно видеть, выполнение различных шагов итерации больше не следует последовательному порядку, но каждый шаг конкурирует в своем выполнении, пытаясь завершить свою задачу как можно скорее.

Очередь в асинхронной модели

Библиотека AsyncIO предоставляет множество инструментов для эффективной реализации асинхронной модели и, как и модули threading и multiprocessing, позволяет применять очереди (queues) для оптимального управления общей памятью. В данном случае за этот ресурс будут конкурировать не потоки, а сопрограммы. Для этой цели предусмотрен класс `asyncio.Queue`, который также позволяет реализовать модель «производитель–потребитель» в асинхронном программировании способом, очень похожим на очередь, рассмотренную в двух предыдущих главах.

Создание общего пространства, организованного с учетом асинхронного программирования, больше не связано с проблемой гонки, как это было в случае с потоками. Вспомним еще раз, что AsyncIO работает в одном потоке, и поэтому вам не придется сталкиваться с проблемами безопасной работы потоков в языке Python.

В случае AsyncIO очередь выполняет задачу подключения между сопрограммой-производителем и сопрограммой-потребителем, которые могут не знать о существовании друг друга и могут быть включены в разные схемы планирования. Таким образом, появится группа сопрограмм-производителей, которые будут добавлять элементы в очередь по одному, асинхронно. Группа сопрограмм-потребителей не будет ожидать завершения всех сопрограмм-производителей, а получит сигнал о том, что они могут, в свою очередь, работать и принимать элементы, уже обработанные в очереди.

Итак, давайте создадим пример кода, в котором мы установим количество производителей и потребителей с помощью двух переменных, передаваемых в качестве аргументов сопрограмме `main()`:

```
import asyncio
import random

async def producer(name, queue):
    n = random.randint(0, 10)
    await asyncio.sleep(n)
    await queue.put(n)
    print("Producer %s adds %s to the queue" %(name,n))

async def consumer(name, queue):
    while True:
```

```
        n = await queue.get()
        await asyncio.sleep(n)
        print("Consumer %s receives %s from the queue" %(name,n))
        queue.task_done()

async def main(nproducers, nconsumers):
    q = asyncio.Queue()
        producers = [asyncio.create_task(producer(n, q)) for n in range(n-
producers)]
        consumers = [asyncio.create_task(consumer(n, q)) for n in range(n-
consumers)]

    await asyncio.gather(*producers)
    await q.join()
    for c in consumers:
        c.cancel()

asyncio.run(main(4,2))
```

Как видно из кода, в качестве примера выбраны четыре производителя и два потребителя. Все производители сопрограмм вставляют в столбец случайное целочисленное значение от 0 до 9 с, и при этом они используют столько секунд, сколько составляет это случайное значение, благодаря функции `asyncio.sleep()`. Все производители сопрограмм запускаются в конкуренции друг с другом с помощью функции `asyncio.gather()`. Создается объект очереди, и значения, сгенерированные производителями, добавляются в него в соответствии со временем выполнения с помощью метода `queue.put()`. Как видно из кода, этот метод также является сопрограммой с префиксом `await`. Затем создаются сопрограммы потребителей и планируются отдельно для производителей. На самом деле они не присутствуют в методе `asyncio.gather()`, но активируются неявно с помощью очереди при каждом вызове `queue.get()`, благодаря которому сопрограммы получают содержащиеся в ней значения. Эта взаимосвязь между потребителями и очередью обеспечивает идеальное функционирование всей системы «производитель–потребитель», поскольку очередь позаботится об активации доступных потребителей при наличии элементов внутри.

При выполнении кода мы получим результат, соответствующий описанному выше.

```
Producer 1 adds 0 to the queue
Consumer 0 receives 0 from the queue
Producer 2 adds 1 to the queue
Consumer 1 receives 1 from the queue
Producer 3 adds 8 to the queue
Producer 0 adds 9 to the queue
Consumer 0 receives 8 from the queue
Consumer 1 receives 9 from the queue
```

Альтернативы библиотеке AsyncIO

В последние годы, в том числе благодаря успеху асинхронного программирования, не использующего потоки, появились другие интересные библиотеки, которые можно считать достойной альтернативой библиотеке AsyncIO.

- **Curio** (*https://curio.readthedocs.io/*) – это библиотека, которая использует сопрограммы, задачи и очереди, а также имеет синтаксис `async` / `await`. Поэтому она рекомендуется для тех, кто уже знаком с `asyncio` и концепциями, лежащими в основе реализации асинхронной модели.
- **Trio** (*https://trio.readthedocs.io/*) – еще более амбициозный проект, поскольку он явно представлен как прямая альтернатива библиотеке `asyncio`, позиционируя себя как более простой и более отвечающий требованиям постоянных инноваций, добавляемых в язык Python. Молодой, интересный проект, заслуживающий особого внимания.

Заключение

В этой главе мы подробно рассмотрели концепцию асинхронного программирования и различные случаи, в которых оно становится многопоточным. Вы наверняка осмыслили тонкости различных типов выполнения и то, как они, в свою очередь, отличаются от результатов параллельного программирования. Важно понять, что там, где параллельное программирование теряет свои преимущества из-за тесной связи с операциями, связанными с центральным процессором, асинхронное программирование, а тем более многопоточное программирование, наоборот, показывают свои лучшие стороны, позволяя эффективно управлять операциями ввода-вывода. Все эти концепции были реализованы благодаря библиотеке AsyncIO, с которой вы уже ознакомились, и основным инструментам. В следующей главе мы рассмотрим распределенное программирование, где применим изученные концепции в системах, не ограниченных одной машиной, а включающих несколько процессоров, распределенных по часто разным устройствам и расположенных в разных местах.

Список литературы

- *https://docs.python.org/3/library/asyncio.html*.
- *https://realpython.com/async-io-python/*.
- *https://trio.readthedocs.io/en/stable/*.
- *https://curio.readthedocs.io/en/latest/*.

Глава 5

Реализация параллелизма с помощью распределенных систем

Распределенные системы являются одним из направлений параллельного и многопоточного программирования. Они представляют собой эффективное решение благодаря высокому уровню производительности. В связи с этим необходимо хорошо разбираться в таких системах и связанных с ними решениях, представленных в языке Python. В этой главе мы рассмотрим несколько вариантов таких решений. Сначала мы обсудим фреймворк Celery, который в настоящее время является основным инструментом Python для реализации распределенных систем. Эта библиотека представляет собой эффективное средство для построения систем такого типа и является отличным образцом для понимания возможных архитектур, элементов, из которых они состоят, а также различных механизмов, лежащих в их основе. На ряде примеров мы покажем, как использовать эту библиотеку для выполнения нескольких одновременно выполняемых операций, называемых задачами, как они распределяются и вычисляются в параллельном режиме в системе на базе Celery. Затем мы перейдем к другим альтернативным решениям, начиная с очень похожего, такого как Dramatiq, и заканчивая более простым, но концептуально отличающимся от них, таким как SCOOP.

Структура

В этой главе мы обсудим следующие темы:

- распределенные системы;
- модель «клиент–сервер»;
- фреймворк Celery;
- обмен сообщениями;
- задача;
- рабочий процесс;
- Dramatiq;

- Docker;
- SCOOP.

Распределенные системы и программирование

Распределенная (distributed) **система** – это любая система, в которой составляющие ее программные элементы работают на разных машинах и взаимодействуют друг с другом для достижения одной и той же цели. Эти элементы, находясь на разных машинах и, возможно, даже географически удаленных друг от друга системах, могут обмениваться сообщениями посредством сети связи.

Наличие объектов, выполняющих свои задачи независимо и одновременно, является дальнейшим развитием концепций потоков и процессов, рассмотренных в предыдущих главах. Таким образом, распределенные системы, помимо выполнения полезных операций в рамках модели «клиент–сервер», вполне способны выполнять ряд операций в режиме многопоточности или даже параллельно. Из этого следует, что распределенные системы прекрасно вписываются в контекст параллельного программирования, и большинство рассматриваемых концепций в полной мере применимы к этим системам.

Итак, мы можем использовать параллельное программирование в модели «клиент–сервер» (см. рис. 5.1), где клиент запрашивает у сервера серию задач, а сервер в свою очередь выполняет поиск в распределенной системе из равного числа вычислительных объектов под названием рабочий процесс (worker), каждый из которых способен выполнять эти задачи в параллельном режиме.

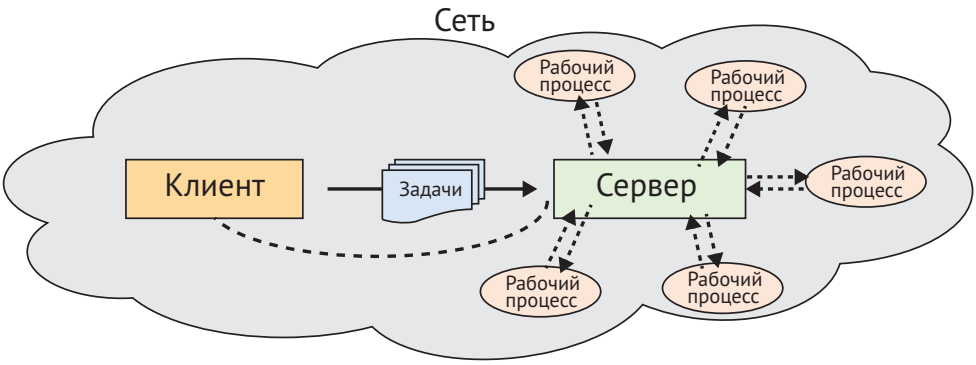


Рис. 5.1. Распределенная система в параллельном программировании

Сервер собирает результаты, полученные в асинхронном режиме, и отправляет их запрашивающему клиенту. Этот тип программирования определяется как распределенное программирование, и именно ему посвящена эта глава.

Плюсы и минусы распределенных систем

Эти системы обеспечивают отличную производительность благодаря таким факторам, как:

- масштабируемость,
- параллелизм,
- гибкость,
- надежность (отказоустойчивость).

Все эти факторы преимущественно обусловлены распределенным характером работы этих систем.

- *Масштабируемость*: гарантируется возможностью увеличения количества элементов в системе по мере роста запроса на ресурсы для решения задачи. С теоретической точки зрения в распределенных системах нет ограничений на это увеличение. Если задача становится более сложной, достаточно добавить другие машины в другие копии элементов системы, чтобы они взяли на себя выполнение конкурирующих вычислений. Совершенно очевидно, что распределенные системы ориентированы на горизонтальную, а не вертикальную масштабируемость.
- *Параллелизм*: еще одна особенность архитектуры этих систем. На самом деле при выполнении ряда задач распределенная система может назначить каждую из них отдельному блоку обработки, который будет действовать одновременно с другими блоками. Таким образом, все задачи будут выполняться в конкурирующем режиме, а зачастую и параллельно.
- *Гибкость*: в распределенной системе она обусловлена множеством элементов, каждый из которых выполняет определенную функцию и находится на отдельном, подключенном к системе компьютере. Эти компьютеры могут быть разными, а компоненты системы можно рассматривать как действительно независимые приложения, которые можно заменить другими совместимыми приложениями того же назначения. Следовательно, при построении распределенной системы существует возможность выбора приложения, которое будет играть роль определенного компонента. Выбор может быть обусловлен выбором операционной системы, функциональностью самого приложения или определенными характеристиками, которые необходимо внести в систему. Все это дает возможность создавать распределенные системы, которые всегда будут соответствовать конкретным запросам пользователя.
- *Надежность или отказоустойчивость системы*: она также гарантируется избыточностью распределенных в системе элементов. Наличие копий элементов на разных машинах гарантирует возможность продолжения работы даже после сбоя приложения или всей машины. Такие системы способны прозрачно и быстро управлять перемещением задачи из сбойного приложения в свободное, находящееся в режиме ожидания. С позиции клиента даже после сбоя в работе не возникнет никаких перебоев в выполнении запрашиваемых услуг.

Однако на деле не все так идеально, и, несмотря на все преимущества распределенных систем, необходимо учитывать, что за это приходится платить определенную цену.

Тот факт, что распределенные системы основаны на большом количестве элементов, которые могут быть разбросаны по разным машинам, требует наличия механизма взаимодействия между различными элементами: обмена сообщениями (messaging). Компоненты распределенных систем используют сообщения для взаимодействия друг с другом и обмена информацией. Таким образом, в каждом случае необходимо создать сообщение и отправить его по сети нужному получателю, который должен будет его принять и распаковать в поисках нужной информации. Совершенно очевидно, что такая система также нуждается в средствах управления, а все это предполагает значительное потребление ресурсов.

Еще одна проблема, связанная с наличием множества различных элементов в распределенных системах, заключается в сложности их обслуживания. Существует множество приложений, которые могут выполнять эти задачи, и все они должны быть реализованы для поддержки всех систем и выполнения всех возможных функций. Но на практике это просто нереальная задача. Поэтому при построении распределенной системы может случиться так, что наш выбор оказался недостаточно удачным и некоторые элементы не поддерживают все необходимые функции в полной мере. Могут возникнуть ситуации несовместимости, особенно если у вас установлены обновленные версии некоторых элементов наряду с более старыми у других элементов.

Кроме того, по мере увеличения количества элементов системы возрастает и сложность ее эксплуатации. Поэтому в случае возникновения проблем или необходимости администрирования становится все труднее проверить правильность ее функционирования.

Последним фактором, который необходимо принимать во внимание, является зависимость сетевой инфраструктуры распределенной системы от той сети, к которой она подключена. Эта инфраструктура имеет решающее значение для эффективности распределенной системы. Сбои в работе сети, неправильные настройки или другие проблемы чаще всего являются признаком неудовлетворительного управления сетью. Эта задача, как правило, выходит за рамки компетенции разработчика.

Celery

Celery – это фреймворк на языке Python, предназначенный для создания распределенных систем на основе этого языка.

Соответственно, Celery с помощью множества компонентов и инструментов позволяет без особых усилий и в доступной для всех форме создавать распределенные системы, отвечающие разнообразным потребностям. Его особые характеристики позволяют нам создавать системы с возрастающей сложностью, начиная от распределенной системы на одной машине и заканчивая сетевыми системами из бесчисленного количества машин, работающих на разных технологических платформах. Эта возможность в значительной степени предоставляется благодаря высокой степени портируемости Python, что обеспечивает возможность его работы на разных архитектурах и операционных системах.

Углубляясь в детали, характеристики распределенной системы Celery можно выразить в трех ключевых моментах:

- портируемость,
- высокая доступность,
- масштабируемость.

В такой системе, даже если ее архитектура преимущественно контролируется Celery, различные составные компоненты могут относиться к другим технологиям и добавляться, заменяться и модифицироваться без особых проблем (концепция подключаемых компонентов). Благодаря этому система обладает высокой степенью портируемости в соответствии с различными потребностями и используемыми технологиями. Этот подход отлично подходит для распределенных систем, поскольку каждый из таких компонентов может находиться на отдельном компьютере, быть физически отделенным и даже географически удаленным, подключаясь к системе через сетевое соединение.

Еще одним преимуществом этого типа архитектуры является тот факт, что количество компонентов, предоставляющих услуги для клиентов (таких как брокеры и рабочие процессы) может быть переменным, и они могут быть распределены на разных машинах. Тем самым гарантируется непрерывная работоспособность даже в случае возникновения аппаратных или программных проблем на некоторых машинах, что обеспечивает высокую доступность всей системы. Кроме того, количество таких компонентов может варьироваться в зависимости от запроса, и с теоретической точки зрения это количество не имеет максимального предела. К системе может быть просто добавлено дополнительное оборудование через сетевые соединения, что позволит распределить использование ресурсов, сохранив его неизменным для каждого компонента. Таким образом, по мере роста задачи можно добавлять дополнительные компоненты, эффективно распределяя ресурсы и обеспечивая горизонтальное масштабирование.

Архитектура систем Celery

В распределенной системе на основе Celery задействовано множество различных компонентов, каждый из которых играет определенную роль в механизме распределенных вычислений (см. рис. 5.2). К основным компонентам относятся:

- брокер;
- очередь задач;
- клиенты;
- рабочие процессы;
- бэкенд результатов (опционально).

Эти компоненты обмениваются между собой данными и сообщениями с *помощью механизма обмена сообщениями* (message mechanism). Работа этого

механизма обеспечивается приложением для передачи сообщений, которое в контексте системы Celery называется брокером (broker). Существует несколько приложений для передачи сообщений, совместимых с системами Celery, и каждое из них использует свой протокол связи.

Клиенты (clients) – это процессы, которые при выполнении своих задач запрашивают определенные службы, предоставляемые распределенной системой. Эти запросы осуществляются посредством сообщений, содержащих указание на выполняемую задачу. Эти сообщения отправляются брокеру, который будет ими управлять. По окончании обработки задачи клиент найдет результат запроса в бэкенде результатов, обычно представляющем собой базу данных. Схема такой архитектуры показана на рис. 5.2.

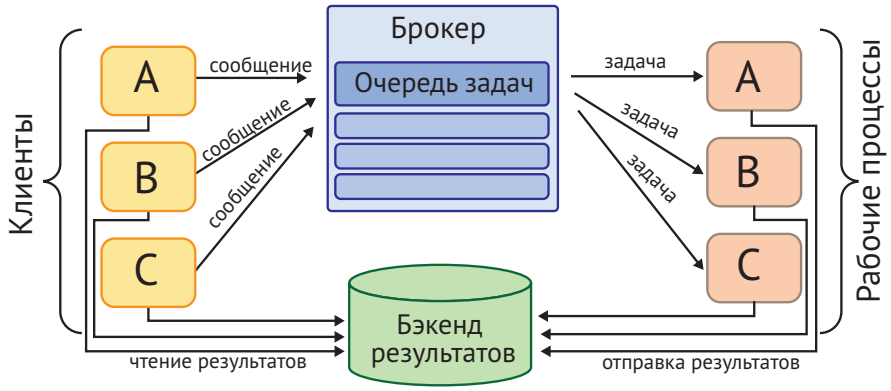


Рис. 5.2. Архитектура Celery

Рабочие процессы (workers) – это процессы, работающие либо на том же компьютере, либо на других компьютерах, распределенных по сети и подключенных к системе Celery. Эти процессы были специально разработаны для выполнения задач, которые передаются им брокером. После запуска рабочие процессы находятся в режиме ожидания, а после получения назначенной им задачи выполняют ее и затем отправляют результат в бэкенд результатов.

Как правило, эти рабочие могут выполнять различные задачи. Это только повышает эффективность системы. С учетом того, что в отсутствие определенной задачи рабочий процесс не остается в режиме ожидания, а может выполнять другие типы задач из очереди задач.

В контексте компонентов распределенной архитектуры Celery, брокер и очереди задач внутри нее играют ключевую роль. Брокер выполняет две основные задачи.

- Первая задача заключается в приеме сообщений, отправленных различными клиентами, путем ввода соответствующих задач в очередь задач.
- Вторая задача заключается в распределении различных задач, находящихся в очереди, между соответствующими ожидающими рабочими процессами.

Очереди задач (task queues) накапливают задачи, запрошенные различными клиентами. Благодаря этому, если при определенном запросе на обслуживание со стороны клиента рабочий процесс не доступен сразу, задача остается в очереди задач в ожидании обработки. Брокер занимается распределением различных задач между ожидающими рабочими процессами. Это позволяет управлять выполнением задач асинхронно и в многопоточном режиме.

Бэкенд результатов (result backend) занимается сбором результатов, полученных различными рабочими процессами во время выполнения задач. Эти результаты затем будут доступны клиентам для считывания. Кроме того, на этом этапе возвращаемые значения попадают в своего рода асинхронный механизм, который позволяет накапливать результаты без необходимости управления каждым результатом сразу после его возврата. Эти механизмы являются обязательными в распределенной системе, где различные компоненты действуют независимо друг от друга, асинхронно, выполняя свои действия по запросу, обработке и управлению результатов в неподдающиеся прогнозированию промежутках времени, в которых возникает необходимость в ожидании.

Задачи

Как было отмечено в предыдущих разделах, системы Celery (и распределенные системы в целом) построены на принципе эффективного распределения задач между различными компонентами. В частности, в этих случаях используется концепция **задачи** (task). Любая задача, которая должна быть выполнена и распределена в системе, должна быть инкапсулирована в форму задачи. **Задача** также может быть определена как единица работы, которая может быть распределена в системе этого типа. Это последнее определение весьма полезно для нас, поскольку оно восходит к концепции многопоточности и параллельного программирования, где любая задача, как правило, может быть разделена на небольшие *атомарные* части, то есть части, которые могут быть выполнены отдельно и независимо друг от друга и которые также определяются как задачи.

Именно здесь находится точка соприкосновения двух технологий: распределенных систем и многопоточного программирования. После разбиения нашей проблемы на задачи, каждая из которых может выполняться независимо, в распределенной системе они могут быть отправлены брокеру и выполнены одновременно разными рабочими процессами, распределенными по сети. Полученные результаты будут собраны в бэкенде результатов и перекомпонованы клиентской программой для получения окончательного результата. Мы уже сталкивались с подобным поведением при работе с многопоточностью и асинхронными сопрограммами. Однако в данном случае у нас есть рабочие процессы – такая концептуальная сущность, к которой относятся многие различные реализации, такие как процессы, потоки, события и эвентлеты (eventlets). Все они управляются открыто или с помощью соответствующих настроек.

Установка системы Celery

После короткого, но содержательного описания теоретической части компонентов распределенной системы Celery перейдем к практической реализации, установке и настройке всех ее компонентов.

Установка Anaconda

Практический совет: когда вы хотите создать тестовые системы, как в примерах этой книги, рекомендуется создать виртуальную среду, в которой будут установлены различные пакеты и модули Python. В случае возникновения ошибок или отпадения необходимости ее можно без проблем удалить, не создавая конфликтов между базовыми версиями Python, установленными в нашей системе приложений, и всеми взаимозависимостями, удалениями и ручными установками, выполненными во время установки нашей тестовой среды.

Мы можем воспользоваться платформой разработки Anaconda, которая значительно облегчает управление виртуальными средами и установку пакетов. Для тех, кто еще не знаком с ней: эту платформу можно бесплатно загрузить из интернета (*https:// www.anaconda.com/products/distribution*). Веб-страница сразу предложит вам самую актуальную и стабильную дистрибутивную версию для вашей операционной системы, как показано на рис. 5.3.



Рис. 5.3. Веб-страница дистрибутива Anaconda

После установки для создания виртуальной среды необходимо перейти в панель **Environments** (Виртуальные среды), см. рис. 5.4.

Затем следует нажать кнопку **Create** (Создать), которая расположена внизу по центру, как показано на рис. 5.5.

При нажатии этой кнопки откроется диалоговое окно, где вы можете ввести желаемое название виртуальной среды, которую хотите использовать, и выбрать версию Python, на которой она будет работать, как показано на рис. 5.6.

После определения имени и версии Python просто нажмите кнопку **Create** для создания виртуальной среды, а затем найдите ее на панели **Anaconda Navigator**. На этом этапе вы можете активировать виртуальную среду простым щелчком по ее имени, которое появится в списке по центру. В случае автора книги виртуальная среда получила название **Parallel**. Когда виртуальная среда активна, сбоку появится зеленый значок (PLAY), как показано на рис. 5.7.

В правой части панели появится список всех пакетов Python, установленных в виртуальной среде. Пакеты со значком Anaconda (зеленый кружок) будут установлены с помощью самого дистрибутива, а пакеты со значком Python будут установлены сторонними средствами с помощью команды `pip`:

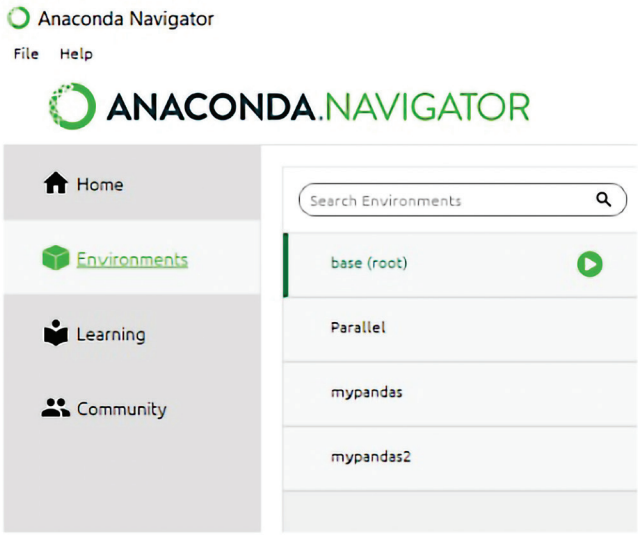


Рис. 5.4. Панель **Anaconda Navigator**

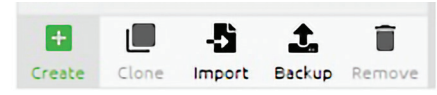


Рис. 5.5. Кнопка создания виртуальной среды в Anaconda Navigator

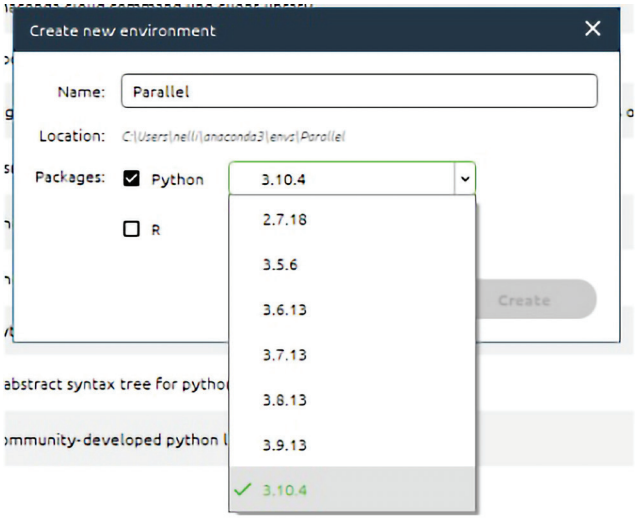


Рис. 5.6. Диалоговое окно для создания виртуальных сред в Anaconda

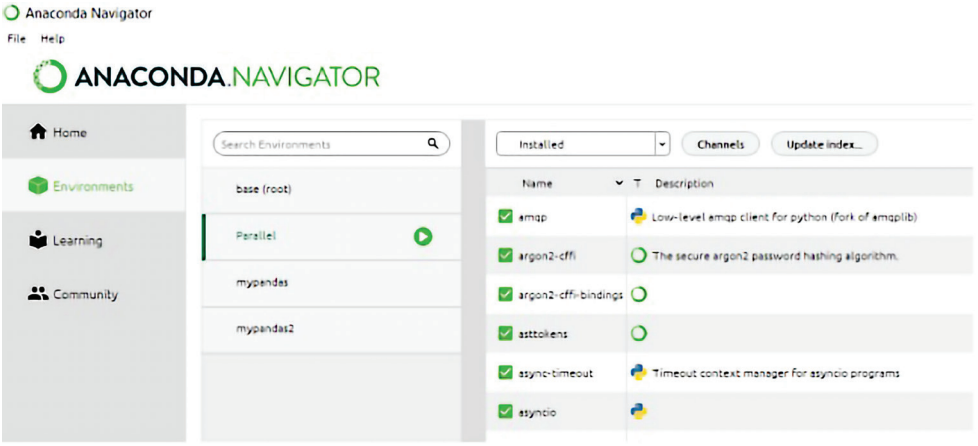


Рис. 5.7. Виртуальная среда с пакетами в Anaconda Navigator

Anaconda – это чрезвычайно полезная и гибкая платформа. Она предоставляет весь спектр приложений для разработки, которые можно использовать самыми разными способами. На панели **Home** (Главная) отображаются все приложения, доступные для только что созданной виртуальной среды, как показано на рис. 5.8.

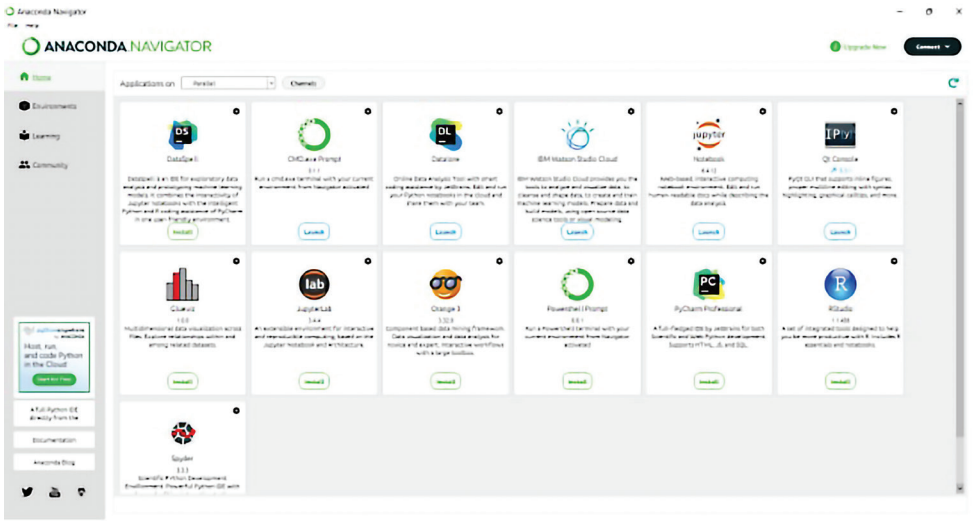


Рис. 5.8. Главная страница Anaconda Navigator

Как можно видеть, здесь доступно множество приложений, в том числе такие полезные, как **Jupyter Notebook** и командная строка `CMD.exe`. В последнем случае нажатие кнопки **Launch** (Запуск) откроет командную консоль, полностью настроенную для работы с виртуальной средой, как показано на рис. 5.9.



Рис. 5.9. Командная консоль виртуальной среды

Установка Celery

В только что открытой командной консоли мы установим библиотеку Celery с помощью PIP.

```
pip install celery
```

> **Обратите внимание**: Пакет `celery` входит в дистрибутив Anaconda, но он неполный (по крайней мере, в версии, имеющейся у автора). Поэтому автор загружает Celery напрямую из PIP, который также предоставляет все остальные связанные пакеты, необходимые для выполнения примеров в этой книге.

Установка Docker

Еще одно приложение, которое хорошо подходит для тестирования распределенных сред вроде тех, что будут представлены в этой главе, называется Docker Desktop. Оно позволяет изолировать приложения, предоставляющие какие-то сервисы, например серверы, базы данных и т. д., в виде пакетов, называемых контейнерами (containers). Приложение чрезвычайно полезно в средах разработки, поскольку запущенные в контейнерах программы весьма просто активировать, деактивировать, удалять и клонировать. Этот инструмент отлично подходит для тестирования распределенной среды, в которой работает множество активных служб прослушивания, таких как Celery. Принимая во внимание сложность версий Python и взаимозависимости пакетов, Docker дает возможность быстро менять и тестировать версии различных серверных приложений без необходимости проведения сложных установок и настроек. Кроме того, также в случае Celery, существуют различные приложения, которые выполняют ту же роль и могут быть оценены в режиме реального времени непосредственно в распределенной системе Celery. Для этого их можно установить все вместе в виде контейнеров, а затем активировать и тестировать одно за другим, в то время как остальная часть системы остается неизменной.

Для установки Docker Desktop просто перейдите на официальную страницу приложения (*https://www.docker.com/*). На отобразившейся веб-странице вам

будет предложено загрузить последнюю стабильную версию, подходящую для вашей операционной системы, как показано на рис. 5.10.

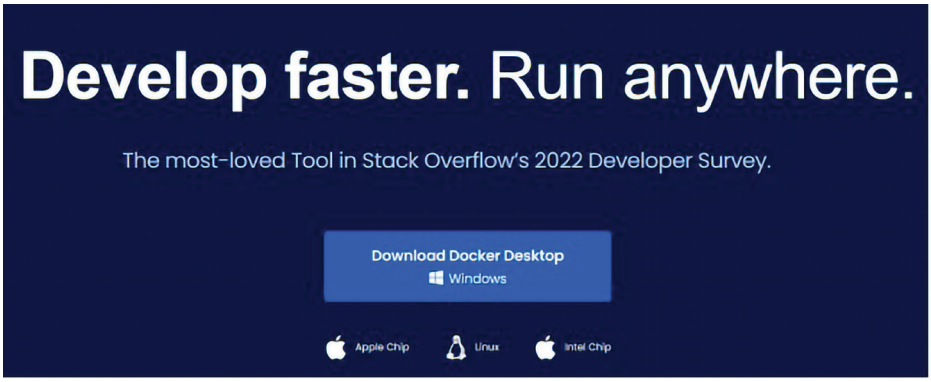


Рис. 5.10. Веб-страница Docker

После установки Docker Desktop появится возможность инсталлировать целую серию готовых контейнеров различных приложений в различных распределенных версиях, воспользовавшись всего лишь командной строкой:

```
docker run -d appname:version
```

Установка службы передачи сообщений (Broker)

После установки Celery необходимо инсталлировать ключевой компонент – брокер (Broker). Существует несколько приложений для передачи сообщений, совместимых с Celery, однако не все из них полностью поддерживают всю функциональность брокеров Celery. Наиболее функциональными и совместимыми приложениями для передачи сообщений являются:

- RabbitMQ;
- Redis.

Оба приложения просты в установке в системах.

1. Для установки приложения Redis на сервер, если вы используете Linux:

   ```
   $ sudo apt-get install redis-server
   ```

2. Затем запустите сервер, введя в командной строке:

   ```
   $ redis-server
   ```

3. В случае системы автора книги, поскольку он использует одну машину (клиент–сервер), было предпочтительнее использовать приложение Docker, в котором можно установить и запустить все службы, в том числе Redis:

   ```
   $ docker run -d -p 6379:6379 redis
   ```

4. Затем установите специальный модуль Celery для использования Redis:

```
pip install celery[redis]
```

5. Если же вы предпочитаете использовать приложение RabbitMQ, его можно установить следующим образом:

```
$ sudo apt-get install rabbitmq-server
```

6. Для установки RabbitMQ на Docker:

```
docker run -d -p 5672:5672 rabbitmq:3.9
```

В конце установки Docker Desktop у нас появятся активные службы RabbitMQ и Redis, как показано на рис. 5.11.

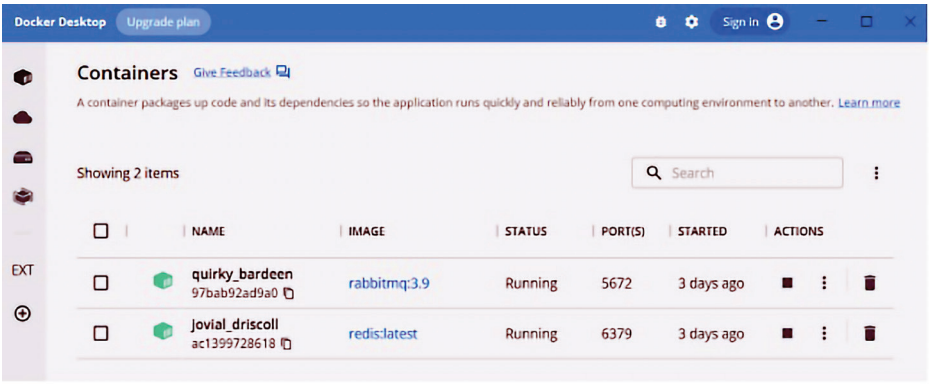


Рис. 5.11. Docker Desktop

Также в этом случае необходимо установить модуль Celery, созданный специально для использования RabbitMQ внутри него:

```
pip install celery[librabbitmq]
```

Установка бэкенда результатов

Существует множество различных приложений для такого компонента, как бэкенд результатов (result backend). К числу таких бэкендов, полностью поддерживаемых Celery, относятся:

- RabbitMQ;
- Redis;
- MongoDB;
- Memcached.

Для удобства мы будем использовать в качестве бэкенда результатов RabbitMQ и Redis. Такое решение позволяет избежать дополнительных установок в рамках этой книги.

Установка системы Celery

Для работы Celery не требуется специальной настройки, по крайней мере, в большинстве случаев. Программно выполняется лишь несколько настроек. Более того, вы можете определить компоненты, которые станут частью системы Celery, во время ее инстанциирования.

1. Для создания экземпляра Celery, который по соглашению будем называть `app` (но можно назвать как угодно), используем конструктор `Celery()`. Он принимает в качестве аргументов модуль с задачами, которые система будет предоставлять клиентам (`tasks.py`), а также различные URL-адреса и протоколы компонентов, входящих в систему.

   Например, если мы хотим использовать серверное приложение RabbitMQ в качестве брокера, напишем следующее:

   ```
   from celery import Celery
   app = Celery('tasks', broker='pyamqp://guest@localhost//', backend='rpc://')
   ```

   Как видите, сервер RabbitMQ работает по протоколу pyamqp и доступен по адресу локального хоста (то есть на том же компьютере, что и наш) в качестве гостевого пользователя.

2. Если же в качестве брокера нужно использовать приложение Redis, напишем следующее:

   ```
   from celery import Celery
   app = Celery('tasks', broker='redis://localhost//', backend='redis://localhost')
   ```

   Эти фрагменты кода необходимо вставить в файл `tasks.py`.

3. Теперь, когда соответствующие настройки выполнены, можно запустить службу Celery:

   ```
   $ celery -A tasks worker --loglevel=INFO
   ```

   При выполнении этой команды будет получен следующий результат:

   ```
   -------------- celery@LAPTOP-1STIRGLU v5.2.7 (dawn-chorus)
   --- ***** -----
   -- ******* ---- Windows-10-10.0.22000-SP0 2022-09-22 14:40:30
   - *** --- * ---
   - ** ---------- [config]
   - ** ---------- .>              tasks:0x2c647cdad40
   - ** ---------- .> transport:   amqp://guest:**@localhost:5672//
   - ** ---------- .> results:     rpc://
   - *** --- * --- .> concurrency: 12 (prefork)
   -- ******* ---- .> task events: OFF (enable -E to monitor tasks in this worker)
   --- ***** -----
    -------------- [queues]
                .> celery          exchange=celery(direct) key=celery

   [tasks]
   ```

```
  . tasks.add
[2022-09-22 14:40:30,583: INFO/MainProcess] Connected to amqp://
guest:**@127.0.0.1:5672//
[2022-09-22 14:40:30,604: INFO/MainProcess] mingle: searching for neighbors
[2022-09-22 14:40:31,535:
28788 calling self.run() INFO/SpawnPoolWorker-1] child process
[2022-09-22 14:40:31,535:
27892 calling self.run() INFO/SpawnPoolWorker-2] child process
[2022-09-22 14:40:31,550:
19868 calling self.run() INFO/SpawnPoolWorker-3] child process
[2022-09-22 14:40:31,566:
19072 calling self.run() INFO/SpawnPoolWorker-4] child process
[2022-09-22 14:40:31,566:
31208 calling self.run() INFO/SpawnPoolWorker-8] child process
[2022-09-22 14:40:31,566: INFO/SpawnPoolWorker-7] child process
30672 calling self.run()
[2022-09-22 14:40:31,582: INFO/SpawnPoolWorker-5] child process
30480 calling self.run()
[2022-09-22 14:40:31,582: INFO/SpawnPoolWorker-6] child process
6328 calling self.run()
[2022-09-22 14:40:31,582: INFO/SpawnPoolWorker-9] child process
14124 calling self.run()
[2022-09-22 14:40:31,598: INFO/SpawnPoolWorker-11] child process
28616 calling self.run()
[2022-09-22 14:40:31,598: INFO/SpawnPoolWorker-10] child process
26540 calling self.run()
[2022-09-22 14:40:31,613: INFO/SpawnPoolWorker-12] child process
4888 calling self.run()
[2022-09-22 14:40:31,726: INFO/MainProcess] mingle: all alone
[2022-09-22 14:40:31,779: INFO/MainProcess] celery@LAPTOP-1STIRGLU
ready.
```

4. Как можно видеть, сервер по умолчанию запустил двенадцать рабочих процессов. Это число зависит от характеристик машины, на которой запущен Celery. В нашем случае имеется двенадцать ядер, поэтому было создано двенадцать рабочих процессов. Брокер будет оставаться активным в ожидании вызовов задач, которые будут назначены рабочим процессам.

   Если вы работаете в Windows, необходимо ввести:

   ```
   $ celery -A tasks worker --pool=solo --loglevel=INFO
   ```

   Но в этом случае вы будете работать с одним потоком, и конкурентной среды больше не будет. В этом случае необходимо настроить рабочие процессы с помощью инструмента `eventlet`. Установим его.

   ```
   pip install eventlet
   celery -A tasks worker --pool=eventlet --loglevel=INFO
   ```

По окончании работы с Celery для завершения нажмите **CTRL**+**C**. На консоли Celery появятся следующие сообщения:

```
worker: Hitting Ctrl+C again will terminate all running tasks!
worker: Warm shutdown (MainProcess)
```

5. После завершения всех процессов вам будет возвращена командная строка.
6. Для получения полного списка опций командной строки воспользуйтесь следующей командой:

```
$ celery worker -help
```

Существует также множество других команд. Чтобы ознакомиться с ними, введите следующую команду в командной строке:

```
$ celery -help
```

Определение задач

Теперь, когда у нас есть простая, но достаточно функциональная распределенная система на базе Celery, можно приступить к определению задач и тестированию примеров.

В Celery на стороне сервера, где находится файл модуля tasks.py, вы можете реализовать задачи, которые система будет предоставлять клиентам. Для идентификации задачи в коде используется декоратор `@app.task`, где `app` – это экземпляр Celery, используемый в файле `tasks.py`.

```
@app.task
def my_task():
    return "This is a task"
```

Вызов задач

Задачи – это услуги, доступные в распределенной системе, которые могут запрашивать клиенты. Поэтому в коде клиента мы можем в любой момент отправить запрос на услугу, которая представляет собой вызов задачи. Для этого Celery предоставляет два различных метода:

- `delay()`;
- `apply_async()`.

Оба метода отправляют задачу брокеру через сообщение, а различие между ними заключается в типе передаваемых аргументов. Метод `delay()` принимает аргументы, размещенные отдельно.

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

В то время как `apply_async()` объединяет их в список и словарь:

```
task.apply_async(args=[arg1, arg2], kwargs={'kwarg1': 'x', 'kwarg2': 'y'})
```

Когда мы вызываем задачу, в качестве возвращаемого значения мы получаем значение **AsyncResult**. Этот объект позволит нам проверить статус задачи со стороны клиента.

Пример задачи

В файле модуля на сервере `tasks.py` мы определяем задачу следующим образом:

```
@app.task
def add(x, y):
    return x + y
```

На стороне клиента мы откроем сессию Python на уровне того же каталога, где находится файл `tasks.py`, что значительно упрощает задачу. После открытия сессии мы сначала импортируем задачу, созданную в файле `tasks.py`, а затем вызываем ее простым использованием метода `delay()`.

```
>>> from tasks import add
>>> add.delay(4,4)
```

При выполнении клиентского кода задача будет инкапсулирована в сообщении и отправлена прослушивающему брокеру. После обработки задачи рабочим процессом вы сможете увидеть результат на консоли машины, на которой находится рабочий процесс.

```
[2022-09-22 15:30:46,024: INFO/MainProcess] Task tasks.add[4966d730-5334-
4089-8395-cb04f40de3c9] received
[2022-09-22 15:30:46,041: INFO/MainProcess] Task tasks.add[4966d730-5334-
4089-8395-cb04f40de3c9] succeeded in 0.01600000000325963s: 8
```

Как можно видеть, в выводе отображаются две строки. Первая показывает, что брокер получил сообщение, а вторая – что задача была успешно выполнена, с указанием времени выполнения и возвращенного результата.

Если бы мы хотели использовать `apply_async()`, нам пришлось бы передать два аргумента, необходимых для задачи, в виде списка.

```
>>> add.apply_async([4,4])
```

Если же мы настроили Celery с использованием бэкенда результатов, мы также можем получить это значение на стороне клиента следующим образом:

```
>>> ares = add.delay(4,4)
>>> ares.get()
8
```

Вызов задачи возвращает объект `AsyncResult`. С помощью метода `get()` мы запросим у объекта считывание результата из бэкенда результатов. Это значение будет доступно только после обработки задачи, поэтому на это уйдет

некоторое время. Поскольку вызов `get()` должен быть асинхронным, сначала необходимо проверить состояние вызванной задачи. Например, с помощью метода `ready()` можно узнать, завершена ли задача:

```
>>> ares.ready()
True
```

Это вернет булево значение: `True`, если задача завершена, и `False` в противном случае. Еще одна важная информация – определение, была ли задача выполнена успешно или завершилась сбоем с генерацией исключения, с помощью методов `successful()` и `failed()` соответственно. Они также возвращают булевы значения. Все эти и другие подобные методы используются для управления выполнением программы в зависимости от статуса задачи.

```
>>> ares.successful()
True
>>> ares.failed()
False
```

Сигнатуры и примитивы

Теперь вы знаете, как вызывать задачи в клиентской программе. Однако возможности Celery на этом не заканчиваются. Эта библиотека также предоставляет набор инструментов, которые позволяют создавать сложные рабочие процессы, состоящие из нескольких одновременно выполняемых задач, при этом код остается очень простым.

Прежде всего в Celery используется такой концепт, как сигнатуры (signatures), которые позволяют упростить вызов задачи. Например:

```
>>> s1 = add.signature((2,2), countdown=10)
```

В данном случае мы сохранили вызов задачи в сигнатуре, что позволит нам использовать его в коде, не переписывая все заново, а просто используя только что объявленную сигнатуру:

```
>>> s1.delay()
```

Также существует стандартная сигнатура `s()`, которая просто заменяет вызов `delay()`:

```
>>> add.delay(4,5)
```

С этой сигнатурой все выглядит очень просто:

```
>>> add.s(4,5)
```

Это может значительно облегчить жизнь программиста. На самом деле эти сигнатуры необходимы для использования в качестве аргументов в ряде функций, доступных в библиотеке Celery:

- `group()`;
- `chain()`;
- `chord()`.

Эти функции основаны на примитивах (primitives), которые являются ничем иным, как сигнатурами со сложной структурой вызова и которые, именно за счет своей формы, генерируют определенные рабочие процессы в распределенных системах.

Например, метод `group()` принимает в качестве аргумента список задач, которые будут выполняться одновременно. Возвращаемое значение является особым, что позволяет рассматривать его как группу, получая в конце список с элементами, следующими в порядке вызова задач:

```
>>> from celery import group
>>> g = group(add.s(i,i) for i in range(10))()
>>> g.get()
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

В консоли сервера появится следующий вывод:

```
[2022-09-22 16:24:24,645: INFO/MainProcess] Task tasks.add[4dfadc20-7deb-
459c-8dc5-9db5ea377263] succeeded in 0.031000000017229468s: 2
[2022-09-22 16:24:24,670: INFO/MainProcess] Task tasks.add[fb8fa5fc-7c9f-
4659-8168-26133100f17b] succeeded in 0.01600000000325963s: 6
[2022-09-22 16:24:24,695: INFO/MainProcess] Task tasks.add[1b9b73e9-a8f4-
43d5-aacc-791a19789a1c] succeeded in 0.030999999959021807s: 4
[2022-09-22 16:24:24,721: INFO/MainProcess] Task tasks.add[f2764fb6-ed38-
4f8c-927a-0b4c4810a03f] succeeded in 0.031000000017229468s: 14
[2022-09-22 16:24:24,746: INFO/MainProcess] Task tasks.add[9e6d6b83-1b3f-
4b9f-963f-2a94bea21b93] succeeded in 0.01600000000325963s: 18
[2022-09-22 16:24:24,771: INFO/MainProcess] Task tasks.add[c4b3b795-329b-
4f88-8409-4f425ad986a7] succeeded in 0.031000000017229468s: 8
[2022-09-22 16:24:24,796: INFO/MainProcess] Task tasks.add[72bc2b1c-189b-
4d56-921d-decbc17cc29f] succeeded in 0.01600000000325963s: 16
[2022-09-22 16:24:24,825: INFO/MainProcess] Task tasks.add[30d86bf8-2f46-
4e50-b2df-1a9b051c283a] succeeded in 0.01600000000325963s: 12
[2022-09-22 16:24:24,850: INFO/MainProcess] Task tasks.add[bdbc022a-2e01-
4cf9-870a-3ae0cf20ea9e] succeeded in 0.031000000017229468s: 10
```

Как можно видеть, `10` отправленных задач были выполнены в разное время без соблюдения порядка итерации, использованного в вызове. Можно также заметить, что время выполнения различных задач было разным. Однако результаты отдельных операций были собраны и отсортированы в соответствии с этим порядком.

Еще один очень полезный метод – это `chain()`. Он дает возможность связать несколько задач в цепочку, чтобы результат одной задачи использовался в качестве аргумента второй задачи и т. д.

```
>>> from tasks import add, mul
>>> from celery import chain
>>> c = chain(add.s(4,4) | mul.s(8))()
>>> c.get()
```

В консоли сервера появится следующий вывод:

```
[2022-09-22 16:41:52,501: INFO/MainProcess] Task tasks.add[9bad176a-937d-
4742-8698-469b0918fe69] received
[2022-09-22 16:41:52,539: INFO/MainProcess] Task tasks.add[9bad176a-937d-
4742-8698-469b0918fe69] succeeded in 0.031000000017229468s: 8
[2022-09-22 16:41:52,540: INFO/MainProcess] Task tasks.mul[21339f93-56f3-
45c1-a09a-09b24b36999a] received
[2022-09-22 16:41:52,565: INFO/MainProcess] Task tasks.mul[21339f93-56f3-
45c1-a09a-09b24b36999a] succeeded in 0.030999999959021807s: 64
```

Из этого примера видно, что произошло именно то, что мы описали. Сообщение от первой задачи в цепочке принимается и немедленно выполняется. Затем результат передается в качестве аргумента второй задаче и отправляется обратно в виде сообщения брокеру. После этого выполняется вторая задача, и результат возвращается клиенту.

Еще один метод – `chord()` использует группу задач в обратном вызове.

```
>>> from tasks import add, mul, xsum
>>> from celery import chord
>>> c = chord((add.s(i,i) for i in range(10)), xsum.s())()
>>> c.get() 90
```

Обратите внимание: если вы попытаетесь запросить `chord` с помощью RPC, вы получите сообщение об ошибке, так как эта функция не реализована в этом типе бэкенда результатов.

```
NotImplementedError: The "rpc" result backend does not support chords!
(бэкенд результатов "rpc" не поддерживает chord!)
```

Обратите внимание, что группа, связанная с задачей, также обновляется до `chord`, поскольку этот шаблон требует синхронизации.

Бэкенды результатов, поддерживающие `chord`: Redis, Database, Memcached и др.

В этом случае в качестве бэкенда результатов необходимо использовать Redis. Замените настройки и перезапустите сервер Celery.

В консоли сервера Celery появится следующий результат:

```
[2022-09-22 16:59:09,209: INFO/MainProcess] Task tasks.add[c4a4c71c-3c4b-
45ce-aef2-842cfe024330] received
```

```
[2022-09-22 16:59:09,242: INFO/MainProcess] Task tasks.add[a03d2f04-347b-
47da-b052-087ef34f478f] received
[2022-09-22 16:59:09,279: INFO/MainProcess] Task tasks.add[e3af658d-d41f-
4714-87e1-1af6a63c2401] received
[2022-09-22 16:59:09,309: INFO/MainProcess] Task tasks.add[920cbf97-7a20-
4c57-8024-22f69fbdccea] received
[2022-09-22 16:59:09,335: INFO/MainProcess] Task tasks.add[c4a4c71c-3c4b-
45ce-aef2-842cfe024330] succeeded in 0.125s: 0
[2022-09-22 16:59:09,344: INFO/MainProcess] Task tasks.add[a03d2f04-347b-
47da-b052-087ef34f478f] succeeded in 0.10899999999674037s: 2
[2022-09-22 16:59:09,349: INFO/MainProcess] Task tasks.add[2bc571ef-794a-
4581-8cbe-3a8cf6dea136] received
[2022-09-22 16:59:09,379: INFO/MainProcess] Task tasks.add[e3af658d-d41f-
4714-87e1-1af6a63c2401] succeeded in 0.09399999998277053s: 4
[2022-09-22 16:59:09,382: INFO/MainProcess] Task tasks.add[2e5bc833-b47e-
40a7-85e9-74fc152eeb16] received
[2022-09-22 16:59:09,398: INFO/MainProcess] Task tasks.add[920cbf97-7a20-
4c57-8024-22f69fbdccea] succeeded in 0.0940000000409782s: 6
[2022-09-22 16:59:09,419: INFO/MainProcess] Task tasks.add[004d45e8-1b93-
4377-9695-fe000432be75] received
[2022-09-22 16:59:09,448: INFO/MainProcess] Task tasks.add[2bc571ef-794a-
4581-8cbe-3a8cf6dea136] succeeded in 0.09399999998277053s: 8
[2022-09-22 16:59:09,451: INFO/MainProcess] Task tasks.add[ff29e563-cc91-
44f9-a3cb-b2a1895444e3] received
[2022-09-22 16:59:09,484: INFO/MainProcess] Task tasks.add[2e5bc833-b47e-
40a7-85e9-74fc152eeb16] succeeded in 0.10899999999674037s: 10
[2022-09-22 16:59:09,488: INFO/MainProcess] Task tasks.add[760cf014-8613-
44f5-bbc3-77abcf6a7901] received
[2022-09-22 16:59:09,507: INFO/MainProcess] Task tasks.add[004d45e8-1b93-
4377-9695-fe000432be75] succeeded in 0.09399999998277053s: 12
[2022-09-22 16:59:09,521: INFO/MainProcess] Task tasks.add[2938bca7-6566-
4c47-8d0d-c8e94045ea52] received
[2022-09-22 16:59:09,538: INFO/MainProcess] Task tasks.add[ff29e563-cc91-
44f9-a3cb-b2a1895444e3] succeeded in 0.07800000003771856s: 14
[2022-09-22 16:59:09,560: INFO/MainProcess] Task tasks.add[760cf014-8613-
44f5-bbc3-77abcf6a7901] succeeded in 0.061999999976251274s: 16
[2022-09-22 16:59:09,575: INFO/MainProcess] Task tasks.xsum[3329ba27-eb79-
47b2-9618-7e71a8cad91e] received
[2022-09-22 16:59:09,616: INFO/MainProcess] Task tasks.add[2938bca7-6566-
4c47-8d0d-c8e94045ea52] succeeded in 0.09399999998277053s: 18
[2022-09-22 16:59:09,622: INFO/MainProcess] Task tasks.xsum[3329ba27-eb79-
47b2-9618-7e71a8cad91e] succeeded in 0.0470000000204891s: 90
```

Метод `chunks()` позволяет разделить итерируемую структуру на более мелкие фрагменты. В этом случае, если у нас есть, например, 100 объектов, их можно разделить на 10 задач по 10 объектов в каждой.

```
>>> from tasks import add
>>> from celery import chunks
>>> z = zip(range(10),range(10))
>>> list(z)
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), (8, 8),
(9, 9)]
>>> ch = add.chunks(z,10)()
>>> ch.get()
[[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]]
```

Библиотека Dramatiq как альтернатива Celery

Несмотря на то что Celery в настоящее время считается эталонной библиотекой для создания распределенных систем на Python, существуют и другие достойные альтернативы, которые постепенно становятся все более популярными. Одной из них, имеющей минимальные отличия от Celery, является библиотека Dramatiq. Многочисленные общие характеристики с Celery делают ее подходящим кандидатом для тех, кто уже знаком с Celery, поскольку обе библиотеки имеют много одинаковых свойств. Обе библиотеки используют асинхронные задачи и такие приложения, как Redis или RabbitMQ.

Одним из недостатков Celery, как считают некоторые разработчики, является ее достаточно высокая сложность, в то время как Dramatiq задумывалась как более простая и интуитивно понятная библиотека.

Установка Dramatiq

Прежде чем приступить к рассмотрению примеров, необходимо установить последнюю версию Dramatiq (в настоящее время версия 1.13) в нашей виртуальной среде или в среде, специально предназначенной для этих целей. Откроем командную консоль и введем следующее:

```
pip install dramatiq
```

Для наших примеров с Dramatiq мы будем использовать приложение Redis, поэтому нам нужно установить необходимые для его использования зависимые компоненты. Для этого введем следующую команду:

```
pip install dramatiq[redis]
```

Но если вы хотите использовать RabbitMQ, можете ввести вместо этого следующее:

```
pip install dramatiq[rabbitmq]
```

На этом этапе остается только активировать контейнер Redis (или RabbitMQ), который мы загрузили в Docker Desktop, как показано на рис. 5.12.

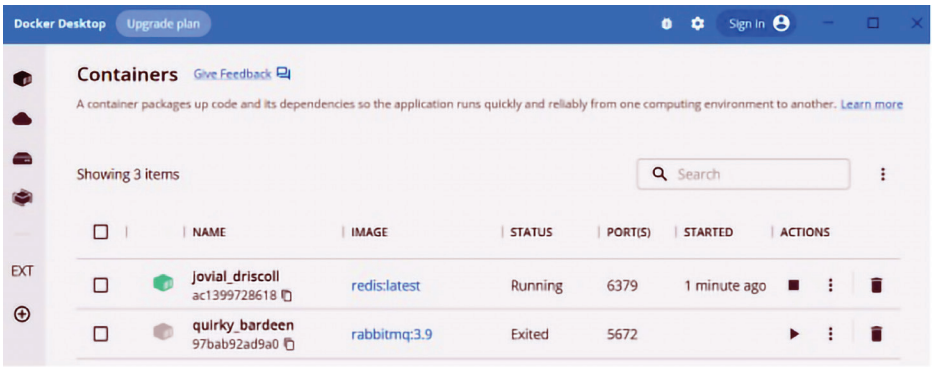


Рис. 5.12. Активация контейнера Redis в Docker Desktop

Начало работы с Dramatiq

Теперь, когда мы установили Dramatiq и все другие необходимые зависимые компоненты, можно приступать к работе с этой библиотекой.

В Dramatiq задачи заменяются акторами (actors), которые представляют собой не что иное, как функции со встроенным кодом, который будет выполняться рабочими процессами, созданными Redis. В этой архитектуре Redis будет работать аналогично тому, как он работает в Celery. Он будет заниматься инкапсуляцией деятельности акторов (*задач*) в сообщения, которые будут отправляться различным ожидающим рабочим процессам (см. рис. 5.13).

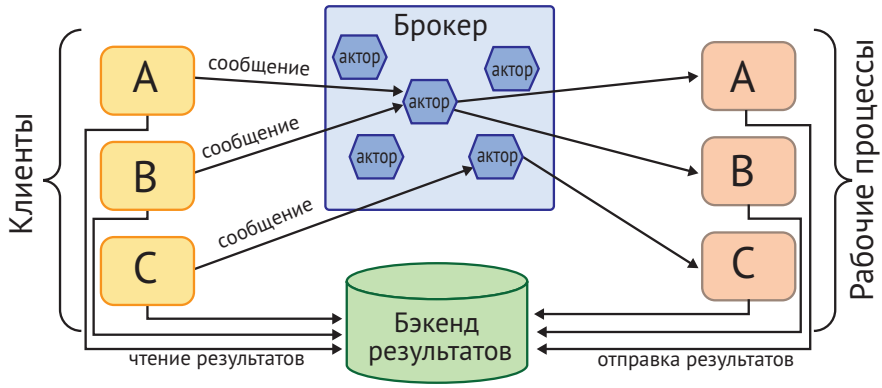


Рис. 5.13. Архитектура Dramatiq

Для этого необходимо выполнить следующие шаги.

1. Сначала реализуем код, который будет выступать в роли сервера и который сохраним как `dramaserver.py`.

```
import dramatiq
import time
```

```
@dramatiq.actor
def wait(t,n):
    time.sleep(t)
    print("I am the actor %s and I will wait for %s secs" %(n,t))
```

Как видно из кода, сначала мы импортируем необходимые библиотеки, включая `dramatiq`. Затем мы определяем функции, которые будут выполнять действия рабочих процессов, которые можно распознать в коде по декоратору `@dramatiq.actor`. В нашем примере в качестве актора мы определим функцию `wait()`, которая не делает ничего, кроме как выжидает определенное количество секунд, а затем отображает сообщение о количестве секунд в результатах.

2. Теперь, когда мы определили очень простой пример файлового сервера с описанием предоставляемых услуг (акторов), выполняемых распределенной системой Dramatiq, мы можем активировать последнюю простым запуском следующей команды из консоли виртуальной среды:

```
dramatiq dramaserver
```

Как видите, вы вводите имя файлового сервера для активации определенных в нем акторов. После запуска команды в выводе появится следующий текст:

```
[2022-09-27 16:46:00,775] [PID 18332] [MainThread] [dramatiq.MainPro-
cess] [INFO] Dramatiq '1.13.0' is booting up.
[2022-09-27 16:46:00,685] [PID 7452] [MainThread] [dramatiq.WorkerPro-
cess(0)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,580] [PID 25400] [MainThread] [dramatiq.Worker-
Process(1)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,685] [PID 25192] [MainThread] [dramatiq.Worker-
Process(2)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,705] [PID 6828] [MainThread] [dramatiq.WorkerPro-
cess(3)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,705] [PID 17300] [MainThread] [dramatiq.Worker-
Process(4)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,700] [PID 18248] [MainThread] [dramatiq.Worker-
Process(5)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,666] [PID 10216] [MainThread] [dramatiq.Worker-
Process(6)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,661] [PID 9372] [MainThread] [dramatiq.WorkerPro-
cess(7)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,736] [PID 25952] [MainThread] [dramatiq.Worker-
Process(8)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,705] [PID 20300] [MainThread] [dramatiq.Worker-
Process(9)] [INFO] Worker process is ready for action.
```

```
[2022-09-27 16:46:00,721] [PID 15876] [MainThread] [dramatiq.Worker-
Process(10)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:00,770] [PID 6644] [MainThread] [dramatiq.WorkerPro-
cess(11)] [INFO] Worker process is ready for action.
[2022-09-27 16:46:01,270] [PID 6640] [MainThread] [dramatiq.ForkPro-
cess(0)] [INFO] Fork process'dramatiq.middleware. prometheus:_run_ex-
position_server' is ready for action.
```

Как можно видеть, в распределенной системе Dramatiq было создано двенадцать рабочих процессов, соответствующих двенадцати различным процессам (PID указаны в тексте), которые будут ждать получения сообщений от Redis, где будут содержаться исполняемые акторы.

Обратите внимание: В Windows для завершения работы сервера нажмите кнопку **X** в правом верхнем углу. Окно не закроется, но все активные службы в Dramatiq будут закрыты.

---

3. На этом этапе мы создаем клиентскую часть, которую сохраним как `dramaclient.py`.

```
import dramatiq
import time
from dramaserver import wait

[wait.send(10,i) for i in range(10)]
print("End Program")
```

4. Осталось только открыть еще одну консоль виртуальной среды и запустить клиентский код.

```
python dramaclient.py
```

5. После запуска кода в консоли сразу же появится текст о завершении программы и приглашение к вводу новой команды.

```
End Program
```

Спустя примерно 10 с в консоли сервера одновременно появится следующий текст:

```
I am the actor 10 and I will wait for 0 secs
I am the actor 10 and I will wait for 7 secs
I am the actor 10 and I will wait for 3 secs
I am the actor 10 and I will wait for 9 secs
```

```
I am the actor 10 and I will wait for 1 secs
I am the actor 10 and I will wait for 6 secs
I am the actor 10 and I will wait for 4 secs
I am the actor 10 and I will wait for 2 secs
I am the actor 10 and I will wait for 5 secs
I am the actor 10 and I will wait for 8 secs
```

В предыдущем коде мы использовали встроенную команду `for` для одновременного запуска нескольких акторов. Но этот способ можно использовать только в некоторых случаях, когда все акторы одинаковы, а одним из параметров обычно является значение самого итератора. Более универсальный способ параллельного запуска нескольких акторов – группировка (grouping).

6. Перепишем предыдущий код клиента следующим образом:

```
import dramatiq
import time
from dramaserver import wait
from dramatiq import group

g = group([
    wait.message(10,'A'),
    wait.message(5,'B'),
    wait.message(4,'C'),
    wait.message(7,'D'),
]).run()

print("End Program")
```

7. Как можно видеть, здесь используется функция `group()`, принимающая список вызовов акторов, которые также могут отличаться друг от друга. Но на этот раз акторы будут вызывать метод `message()` для отправки необходимых параметров.

   При выполнении на выходе консоли сервера получается следующий результат:

```
I am the actor C and I will wait for 4 secs
I am the actor B and I will wait for 5 secs
I am the actor D and I will wait for 7 secs
I am the actor A and I will wait for 10 secs
```

На этот раз время выполнения будет другим, поэтому строки будут отображаться в порядке выполнения с разным временем.

Обработка результатов

До сих мы работали с акторами, которые не генерировали результаты, а именно не возвращали значения. Для них в Dramatiq требуется специальная обработка. Их тоже необходимо поместить в сообщения и собрать в бэкенде результатов, очень похожем на тот, который мы видели в Celery. Здесь в качестве приложения для бэкенда результатов также можно выбрать Redis или RabbitMQ.

Для добавления всех необходимых настроек и активации служб брокера и бэкенда результатов необходимо внести изменения и дополнения в ранее написанный код, на этот раз завершив полную картину распределенной реализации с помощью Dramatiq.

В отношении кода, обслуживающего серверную сторону, необходимо внести следующие изменения:

```
import dramatiq
import time

from dramatiq.brokers.redis import RedisBroker
#from dramatiq.brokers.rabbitmq import RabbitmqBroker
from dramatiq.results import Results
from dramatiq.results.backends import RedisBackend

broker = RedisBroker(host="localhost")
#broker = RabbitmqBroker(host="localhost")
dramatiq.set_broker(broker)
result_backend = RedisBackend(host="localhost") broker.add_middleware(Resul
ts(backend=result_backend))

@dramatiq.actor(store_results=True)
def wait(t,n):
    time.sleep(t)
    print("I am the actor %s and I will wait for %s secs" %(n,t))
    return "I waited for {0} secs".format(t)
```

Как видно из кода, в него было внесено много дополнений. В первой части на этот раз необходимо импортировать все классы, отвечающие за службы Broker и Result Backend. В нашем примере было решено использовать Redis для обеих функций. Однако в классах, соответствующих RabbitMQ, были оставлены комментарии, чтобы дать подсказки тем, кто хочет использовать это приложение. Поскольку пример выполняется локально на одном компьютере, в качестве хоста для обоих сервисов был указан local host. Если вы будете работать в сети, замените эти значения соответствующим IP-адресом. Что касается актора, было добавлено возвращаемое значение, которое по-прежнему является строкой. В данном случае наша цель заключается в отображении этой строки со стороны клиента на экране консоли.

Код на стороне клиента также требует некоторых изменений.

```
import dramatiq
import time
from dramaserver import wait
from dramatiq import group

from dramatiq.brokers.redis import RedisBroker
from dramatiq.results import Results
from dramatiq.results.backends import RedisBackend

broker = RedisBroker(host="localhost")
dramatiq.set_broker(broker)
result_backend = RedisBackend(host="localhost") broker.add_middleware(Results
(backend=result_backend))

g = group([
    wait.message(10,'A'),
    wait.message(5,'B'),
    wait.message(4,'C'),
    wait.message(7,'D'),
]).run()

for res in g.get_results(block=True, timeout=12000):
    print(res)

print("End Program")
```

Как можно видеть, в код клиента также необходимо импортировать те же классы служб Broker и Result Backend, которые мы использовали для серверной стороны. Как правило, это также позволяет клиенту понять, где находятся службы, составляющие распределенную систему Dramatiq, путем указания различных IP-адресов.

Что касается акторов, то на этот раз, будучи объединенными в группу, они хранят свои результаты в памяти бэкенда результатов в течение определенного периода времени. Получить их из группы можно с помощью метода `get_results()`. По умолчанию время хранения результатов в бэкенде результатов составляет `10` с, и, поскольку в нашем примере мы работаем с такими временными интервалами, будет лучше увеличить тайм-аут до `12` с (12 000 мс).

После запуска этих двух фрагментов кода в созданной нами системе Dramatiq на консоли клиентской стороны мы получим следующий результат:

```
I waited for 10 secs
I waited for 5 secs
I waited for 4 secs
I waited for 7 secs
End Program
```

На консоли серверной стороны получится следующий результат:

```
I am the actor C and I will wait for 4 secs
I am the actor B and I will wait for 5 secs
I am the actor D and I will wait for 7 secs
I am the actor A and I will wait for 10 secs
```

Библиотека SCOOP

Еще одна библиотека, доступная онлайн для распределенных вычислений с использованием Python, называется Scalable Concurrent Operation in Python (SCOOP – масштабируемые параллельные вычисления на Python). Этот проект все еще находится в стадии бета-тестирования (на момент написания книги доступна версия 0.7) и состоит из одного модуля под названием scoop. Его задачей является распределение многопоточных задач, называемых в данном случае футурами (futures), между рабочими процессами на разных подключенных к сети машинах.

Основной целью разработчиков SCOOP является максимальное упрощение и прозрачность управления распределенным и параллельным программированием. С помощью SCOOP можно управлять различными задачами на нескольких уровнях обработки с помощью механизмов отображения и свертки (редукции), см. рис. 5.14. Далее модуль обрабатывает физические аспекты, касающиеся возможности распределения задач на основе физического потенциала подключенных к системе машин и методов параллелизации, которые будут применяться в конкретных случаях.

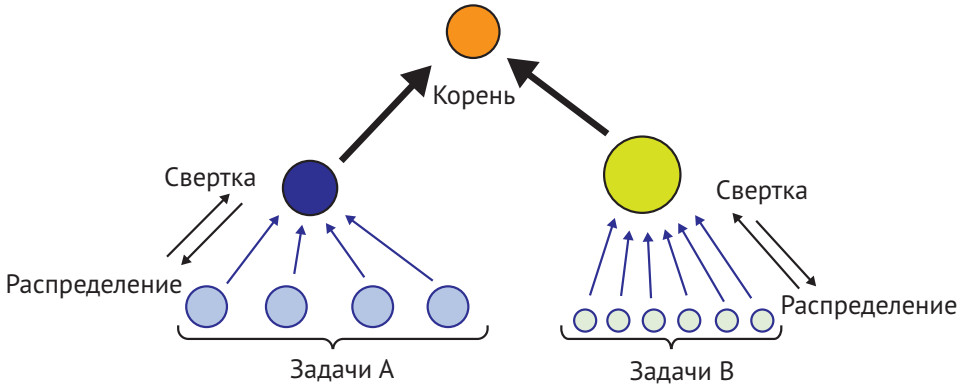


Рис. 5.14. Многоуровневый механизм SCOOP

Если говорить об элементах, составляющих архитектуру, SCOOP в этом плане гораздо проще, чем распределенные системы, такие как Celery, поскольку в нем есть только рабочие процессы (все одинаковые) и брокер, как показано на рис. 5.15.

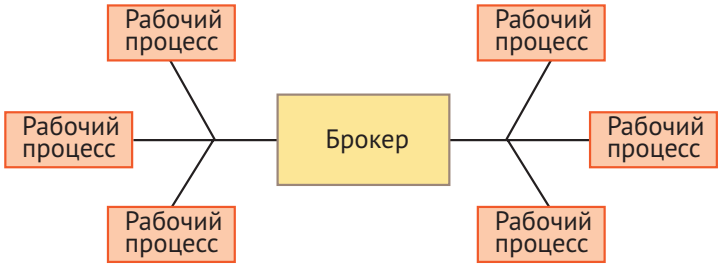


Рис. 5.15. Архитектура SCOOP

Центральным элементом системы обмена данными является брокер, который взаимодействует со всеми независимыми рабочими процессами для обмена сообщениями между ними. Футуры создаются в элементах рабочего процесса, а не в брокере, с помощью централизованной процедуры сериализации. Это повышает надежность архитектуры и улучшает производительность. В действительности основная нагрузка брокера заключается в сетевом обмене и взаимодействии ввода-вывода между различными рабочими процессами с относительно низким временем отклика.

Установка SCOOP

Для установки SCOOP можно использовать созданную ранее виртуальную среду или создать новую специально для этой цели. После активации виртуальной среды необходимо открыть консоль командной строки и установить SCOOP с помощью `pip`.

```
pip install SCOOP
```

Для непосредственного тестирования SCOOP необходимо написать в текстовом редакторе и сохранить в каталоге рабочей среды виртуальной среды файл `scoopy.py` со следующим кодом:

```
from scoop import futures
def worker(value):
    print("I am the Worker %s" %value)

if __name__ == "__main__":
    list(futures.map(worker, range(4)))
```

Перед запуском кода быстро просмотрим его. Из модуля `scoop` мы импортируем футуры, которые представляют собой не что иное, как пул задач, которые брокер отправляет рабочим процессам. Брокер создает задачи с помощью метода `map()`, который, как и его аналог из стандартной библиотеки, принимает в качестве второго аргумента итератор, в данном случае `range(4)`. Это не что иное, как последовательность из четырех элементов с целыми числами в порядке возрастания: `(0,1,2,3)`. Функция `map()` должна быть передана в качестве аргумента функции `list()`, чтобы обеспечить синхронизацию по окончании

выполнения всех рабочих процессов. Таким образом, программа будет ждать результатов всех рабочих процессов, а затем завершит работу.

Каждый элемент итерабельного объекта передается в первый аргумент `map()`, а именно worker, который представляет собой не что иное, как имя определенной в коде функции `worker()`. Эта функция включает в себя операции, которые будут назначены каждому отдельному рабочему процессу. Итак, в нашем примере будет четыре задачи (по количеству элементов итератора). Они будут назначены доступным рабочим процессам, каждый из которых выведет строку с добавлением своего идентификационного номера, точно соответствующего значению итератора.

Для запуска кода с использованием SCOOP необходимо указать опцию `-m` scoop в команде выполнения кода Python. Итак, напишем:

```
python -m scoopy.py
```

Выполнение кода приведет к выводу в консоль следующего результата:

```
[2022-09-26 15:01:08,933] launcher INFO SCOOP 0.7 2.0 on win32 using
Python 3.10.4 | packaged by conda-forge | (main, Mar 30 2022, 08:38:02)
[MSC v.1916 64 bit (AMD64)], API: 1013
[2022-09-26 15:01:08,933] launcher INFO Deploying 12 worker(s) over 1 host(s).
[2022-09-26 15:01:08,933] launcher INFO Worker distribution:
[2022-09-26 15:01:08,933] launcher INFO 127.0.0.1: 11 + origin
Launching 12 worker(s) using an unknown shell.
I am the Worker 0
I am the Worker 1
I am the Worker 3
I am the Worker 2
[2022-09-26 15:01:09,934] launcher (127.0.0.1:64064) INFO Root process is done.
[2022-09-26 15:01:09,934] launcher (127.0.0.1:64064) INFO Finished cleaning
spawned subprocesses.
```

Как видно из результатов выполнения, SCOOP автоматически создал двенадцать рабочих процессов на компьютере (11 + исходный). Это значение автоматически устанавливается SCOOP в соответствии с характеристиками машины, на которой выполняется программа. Как видно из вывода, отображаемого отдельными функциями `worker()` со случайным порядком выполнения, четыре задачи выполняются одновременно.

Давайте усложним предыдущий пример, добавив математическое вычисление с результатом к функции, которую мы назовем `func()`.

```
from scoop import futures
import math
import numpy as np

def func(value):
    result = math.sqrt(value)
```

```
    print("The value %s and the elaboration is %s" %(value, result))
    return result

if __name__ == "__main__":
    data = np.array([10,3,6,1,4,8,25,9])
    results = list(futures.map(func, data))
    for result in results:
        print("This is the result: %s" %result)
```

Как видно из кода, теперь функция `func()`, обрабатываемая рабочими процессами, помимо записи строки в качестве вывода, будет возвращать результат брокеру, который затем будет передан в качестве возвращаемого значения функции `map()`, которое через `list()` будет сохранено в переменной `results`. Мы также заменили упорядоченную последовательность `range(4)` массивом `NumPy` из восьми элементов. В конце программы результаты будут выведены на экран.

Давайте сохраним код в файле и назовем его, например, `scoopy2.py`. В предыдущем примере мы видели, что SCOOP автоматически создал двенадцать рабочих процессов. Это число можно установить в командной строке, добавив опцию `-n` к команде `python`. Например, если мы хотим использовать `4` рабочих процесса, вам нужно будет запустить из консоли виртуальной среды следующую команду:

```
python -m scoop -n 4 scoopy2.py
```

Запуск кода даст следующий результат:

```
[2022-09-26 15:12:25,473] launcher INFO SCOOP 0.7 2.0 on win32 using
Python 3.10.4 | packaged by conda-forge | (main, Mar 30 2022, 08:38:02)
[MSC v.1916 64 bit (AMD64)], API: 1013
[2022-09-26 15:12:25,473] launcher INFO Deploying 4 worker(s) over 1 host(s).
[2022-09-26 15:12:25,489] launcher INFO Worker distribution:
[2022-09-26 15:12:25,489] launcher INFO 127.0.0.1: 3 + origin
Launching 4 worker(s) using an unknown shell.
The value 10 and the elaboration is 3.1622776601683795
The value 3 and the elaboration is 1.7320508075688772
The value 6 and the elaboration is 2.449489742783178
The value 1 and the elaboration is 1.0
The value 4 and the elaboration is 2.0
The value 8 and the elaboration is 2.8284271247461903
The value 25 and the elaboration is 5.0
The value 9 and the elaboration is 3.0
This is the result: 3.1622776601683795
This is the result: 1.7320508075688772
This is the result: 2.449489742783178
This is the result: 1.0
This is the result: 2.0
This is the result: 2.8284271247461903
This is the result: 5.0
```

```
This is the result: 3.0
[2022-09-26 15:12:26,474] launcher (127.0.0.1:64354) INFO Root process is done.
[2022-09-26 15:12:26,474] launcher (127.0.0.1:64354) INFO Finished cleaning
spawned subprocesses.
```

Как видно из результата, порядок выполнения различных задач по-прежнему случайный. Однако в выходных результатах следует отметить, что `list(map())` вернет все результаты в том же порядке, в котором они были определены в массиве `NumPy`. На этом этапе происходит синхронизация, которая ожидает получения всех результатов для присвоения значения результатам, а именно возвращаемой переменной.

После того как стал понятен процесс управления задачами с помощью отображения в архитектуре SCOOP, можно перейти к следующему этапу и добавить этап свертки. В предыдущем примере мы увидели, как при передаче итерабельного объекта из `n` элементов можно получить итерабельный объект из `n` результатов. Теперь можно добавить этап свертки (редукции), добавив функцию агрегирования-свертки результата. Например, мы можем заменить функцию `list()`, которая синхронизирует все результаты различных рабочих процессов и сортирует их в список, функцией свертки, такой как, например, функция `sum()`. Эта функция будет суммировать все результаты различных рабочих процессов. Независимо от времени их выполнения она будет постепенно суммировать все результаты, ожидая окончания работы последнего рабочего процесса, а затем возвращая результат. Здесь снова будет использоваться механизм синхронизации.

Изменим предыдущий код следующим образом:

```
if __name__ == "__main__":
    data = np.array([10,3,6,1,4,8,25,9])
    result = sum(futures.map(func, data))
    print("This is the reduction result: %s" %result)
```

При выполнении вы получите результат свертки, который представляет собой сумму всех результатов рабочих процессов:

```
…
This is the reduction result: 21.172245335266624
…
```

В SCOOP есть еще один метод, `mapReduce()`, который объединяет обе предыдущие операции, а именно отображение и свертку. Мы продолжаем изменять предыдущий код и импортируем модуль `operator`, который позволяет нам вызывать операторы в качестве методов. В нашем примере мы будем использовать `add()` для сложения. Наконец, мы заменяем `map()` на `mapReduce()`, добавляя метод агрегирования `operator.add` в качестве второй применяемой функции.

```
from scoop import futures
import math
import numpy as np
import random
```

```
import operator

def func(value):
    result = math.sqrt(value)
    print("The value %s and the elaboration is %s" %(value, result))
    return result

if __name__ == "__main__":
    data = np.array([10,3,6,1,4,8,25,9])
    result = futures.mapReduce(func, operator.add, data)
    print("This is the reduction result: %s" %result)
```

Запустив приведенный выше код, вы получите тот же результат, что и в предыдущем случае. Задачу можно усложнить еще больше, объединив функции отображения и сокращения.

Например, после вычисления квадратных корней введенных данных мы можем преобразовать их в целочисленные значения и наконец вычислить среднее значение, как показано в следующем модифицированном коде:

```
if __name__ == "__main__":
    data = np.array([10,3,6,1,4,8,25,9])
    result = np.mean(list(futures.map(int, futures.map( func, data))))
    print("This is the reduction result: %s" %result)
```

Запуск этого кода даст следующий результат:

```
[2022-09-26 16:09:50,661] launcher INFO SCOOP 0.7 2.0 on win32 using
Python 3.10.4 | packaged by conda-forge | (main, Mar 30 2022, 08:38:02)
[MSC v.1916 64 bit (AMD64)], API: 1013
[2022-09-26 16:09:50,661] launcher INFO Deploying 4 worker(s) over 1 host(s).
[2022-09-26 16:09:50,661] launcher INFO Worker distribution:
[2022-09-26 16:09:50,661] launcher INFO 127.0.0.1: 3 + origin
Launching 4 worker(s) using an unknown shell.
The value 10 and the elaboration is 3.1622776601683795
The value 3 and the elaboration is 1.7320508075688772
The value 6 and the elaboration is 2.449489742783178
The value 1 and the elaboration is 1.0
The value 4 and the elaboration is 2.0
The value 8 and the elaboration is 2.8284271247461903
The value 25 and the elaboration is 5.0
The value 9 and the elaboration is 3.0
This is the reduction result: 2.375
[2022-09-26 16:09:52,039] launcher (127.0.0.1:65396) INFO Root process is done.
[2022-09-26 16:09:52,054] launcher (127.0.0.1:65396) INFO Finished cleaning
spawned subprocesses.
```

Как можно видеть, ситуация может постепенно усложняться, но при этом распределенные вычисления становятся все более эффективными.

Заключение

В этой главе была представлена концепция распределенных систем и их применение в контексте параллельного программирования. Кроме того, был подробно рассмотрен фреймворк Celery, который в настоящее время является основным ориентиром для реализации распределенных систем на Python. В частности, мы рассмотрели наиболее полезные функции, связанные с параллельным выполнением задач. Затем мы также рассмотрели две возможные альтернативы этому фреймворку, которые предлагают аналогичные решения в нашем контексте: Dramatiq и SCOOP.

В следующей главе мы рассмотрим еще одну возможность использования параллельного программирования: использование графических процессоров (GPU) и их библиотек. Это позволит нам использовать вычислительную мощность процессоров такого типа.

Список литературы

- *https://www.capitalone.com/tech/software-engineering/distributed-systems-using-python-ray/*.
- *https://scoop.readthedocs.io/en/0.7/*.
- *https://pypi.org/project/scoop/*.
- *https://www.pedaldrivenprogramming.com/2018/07/dramatiq-celery-alternative/*.
- *https://dramatiq.io/*.

Глава 6

Программирование GPU с CUDA для максимальной производительности

До сих пор в контексте параллельного программирования мы рассматривали только центральные процессоры (CPU) в наших вычислительных системах. Однако CPU – отнюдь не единственные чипы, подходящие для параллельных вычислений. Кроме них, существуют еще графические процессоры (graphics processing units, GPU). Эти процессоры предназначены для чрезвычайно быстрой и эффективной обработки векторных данных при рендеринге изображений, работе 3D-движков и обработке полигональных примитивов. Развитие графических процессоров в основном обусловлено активным ростом рынка игр, что позволило достичь впечатляющих результатов за последние годы. Интересным моментом является тот факт, что графические процессоры и связанные с ними технологии также оказались крайне эффективными инструментами для научных вычислений.

Структура

В этой главе мы обсудим следующие темы:

- архитектура GPU;
- программирование GPU;
- Numba для CUDA;
- OpenCL.

Архитектура GPU

В практическом плане GPU можно представить в виде совокупности миллионов параллельных ядер, каждое из которых способно выполнять вычисления независимо друг от друга. Такая структура с высокой степенью параллелизма позволяет выполнять вычисления над очень большими векторами и матрицами за очень короткое время, что, безусловно, не под силу CPU. Эти харак-

теристики в последнее время привлекают внимание научного сообщества, что привело к разработке моделей и инструментов разработки, а также высокоспециализированных библиотек для научных целей.

Рассмотрим подробнее внутреннюю структуру графического процессора. Эти знания пригодятся для лучшего понимания моделей программирования GPU. Внутри графические процессоры разделены на два уровня параллелизма. Первый уровень состоит из нескольких вычислительных блоков, составляющих графический процессор, которые называются потоковыми мультипроцессорами (streaming multiprocessor, SM). Каждый из этих блоков работает независимо от других, и все вместе они выполняют задачи одновременно. Второй уровень параллелизма имеет место внутри самого потокового мультипроцессора. Каждый потоковый мультипроцессор, в свою очередь, подразделяется на дополнительные вычислительные блоки, называемые потоковыми процессорами (stream processor, SP). Каждый потоковый процессор имеет ядро, которое может независимо и одновременно с другими потоковыми процессорами обрабатывать поток. Схема этой структуры показана на рис. 6.1.

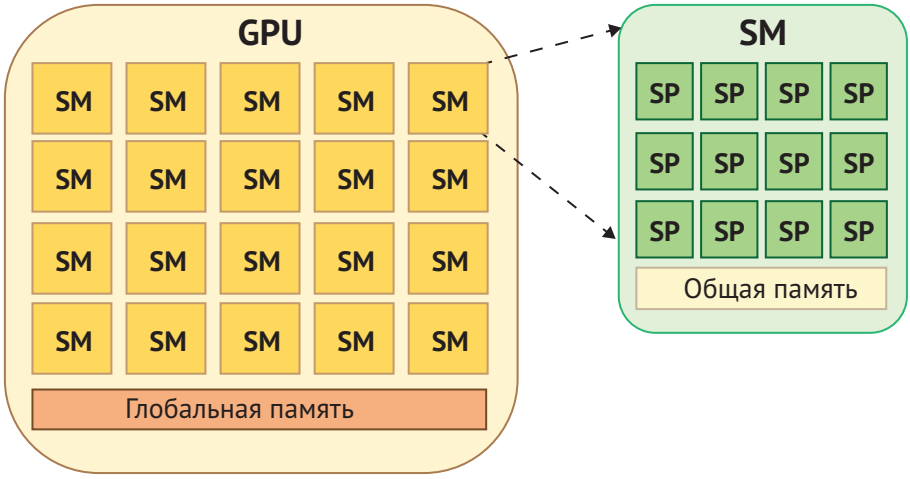


Рис. 6.1. Внутренняя структура графического процессора

Разделение на SM и SP является структурным, и уже на основе этого разделения можно строить логическую организацию для модели программирования GPU. Потоковые процессоры, которые являются базовыми вычислительными модулями GPU, могут быть сгруппированы в блоки. Эти логические блоки можно использовать для выполнения того или иного режима работы. В этом случае все ядра в блоке будут одновременно выполнять одну и ту же инструкцию, но с указанием на различные области памяти, при этом каждое ядро будет обрабатывать назначенную ему часть данных. Память, к которой они будут обращаться, находится внутри самого GPU, что позволяет максимально быстро и оперативно получать доступ к хранящимся в ней значениям. Затем эти данные передаются и повторно загружаются в центральный процессор для дальнейшей обработки.

Высокая степень параллелизма с тысячами или миллионами вычислительных модулей (или потоков), способных выполнять столько же вычислений в параллельном режиме, делает GPU чрезвычайно мощными, но при этом совершенно отличными от центральных процессоров. В связи с этим потребовалось разработать специальные, отличные от традиционных, методы программирования. Для этого необходимо использовать различные драйверы и компиляторы, способные собирать части кода для конкретного языка, которые могут обрабатываться GPU, а они, в свою очередь, выпускаются разными производителями, и поэтому каждый из них требует специального подхода.

В настоящее время есть более эффективные решения, которые позволяют использовать вычислительную мощность графических процессоров и интегрировать ее с классическим, зачастую последовательным процессом выполнения операций на центральных процессорах. Среди различных производителей графических карт NVIDIA является, пожалуй, единственной компанией, которая достигла наибольшего успеха как с точки зрения мощности вычислений, так и в плане использования на наших компьютерных системах. Для этого типа карт существует специальная среда разработки под названием CUDA – Compute Unified Device Architecture (унифицированная архитектура вычислительных устройств), которая была специально разработана для использования с графическими картами NVIDIA. Эта среда разработки в основном построена на языке C со специальными расширениями, которые позволяют задействовать специфические возможности графических процессоров, но при этом можно использовать и другие языки программирования, такие как Fortran, Java, и в особой степени Python.

Еще одной технологией, которая позволяет программировать графические процессоры вместе с центральным процессором для использования его потенциала параллельных вычислений, является фреймворк OpenCL. Он также основан на языках C и C++, но со временем его удалось интегрировать и в другие языки программирования, такие как Python. Эта технология является более универсальной, нежели созданная специально для карт NVIDIA технология CUDA. В действительности OpenCL может работать с видеокартами других производителей (Intel, AMD и т. д.), а также с картами NVIDIA, что стало возможным благодаря выпуску наборов инструментов SDK и специальных драйверов для интеграции OpenCL. Что касается видеокарт NVIDIA, то OpenCL по-прежнему опирается на технологию CUDA, которая полностью совместима и может с ней интегрироваться.

Программирование GPU на Python

Программирование GPU на Python стало возможным благодаря появлению множества библиотек, в основном представляющих собой оболочки соответствующих библиотек C на Python, которые за последние годы претерпели огромные изменения. Их развитие идет настолько быстрыми темпами, что со временем некоторым из них удалось добиться большого успеха, в то время как другие постепенно утратили свою актуальность в связи с постоянной эволю-

цией этих технологий. В настоящее время для параллельного программирования GPU лучше всего подходят две библиотеки:

- Numba,
- PyOpenCL.

Безусловно, они не единственные библиотеки такого рода. Существует множество других библиотек, которые также используют те же базовые технологии, такие как CUDA и OpenCL, но часть из них довольно сложна в использовании, порой несовместима с другими библиотеками и требует компиляции базы на C (с установкой компиляторов, которые должны соответствовать определенным версиям). Нередко настройка среды разработки с такими библиотеками занимает несколько дней, а документация обновляется редко и не всегда помогает добиться желаемого результата. Библиотеки Numba и PyOpenCL полностью интегрированы с такими средами управления пакетами Python, как Pip и Anaconda, поэтому их установка проходит быстро и без проблем, что позволяет приступить к работе без промедления.

Numba

Библиотека Numba с открытым исходным кодом включает в себя JIT-компилятор, способный переводить части кода Python и объекты NumPy непосредственно в соответствующий машинный код с помощью технологии LLVM. Эта технология, предоставляемая пакетом llvmilite, интегрирована в Numba. Благодаря такой системной интеграции Numba предоставляет возможность создавать код, максимально использующий возможности центрального процессора и графического процессора, без значительных правок в языке Python.

Благодаря этому Numba позволяет нам писать классический код на Python с добавлением особых обозначений (соответствующих декораторов Python), которые будут идентифицировать части кода, предназначенные для компиляции с учетом особенностей различных моделей графических процессоров.

Numba для CUDA

В рамках нашей книги мы будем использовать Numba для работы исключительно с графическими процессорами NVIDIA, поэтому нам понадобится CUDA. Чтобы работать с этой технологией, сначала необходимо ознакомиться с ее моделью программирования и понять ее основные концепции. Благодаря им мы сможем хорошо понять различные шаги, которые необходимо выполнить для максимального раскрытия потенциала графического процессора.

В первую очередь в этой модели программирования необходимо определить, на каком центральном процессоре будет выполняться программа, написанная на Python, и какие графические процессоры будут с ним взаимодействовать. Центральный процессор определим в качестве хоста, поскольку он будет выполнять основную часть программы, как правило, последовательно. Графические процессоры, напротив, определим в качестве устройств, поскольку они могут быть разнородными и для их функционирования могут потре-

боваться разные драйверы и компиляторы. Код Python, написанный специально для одновременной работы на CPU и GPU, будет программироваться и выполняться традиционным способом на хосте. При обнаружении частей кода, предназначенных для выполнения на графических процессорах, они будут компилироваться специальным образом с учетом используемой технологии. Поскольку мы рассматриваем графические процессоры NVIDIA, то Numba будет использовать технологию CUDA и встроенный в нее компилятор для создания специально скомпилированных пакетов, называемых ядрами (kernels) и предназначенных для отправки на эти графические процессоры для выполнения. Как будет показано ниже, эти ядра легко идентифицируются в коде как функции, определяемые специальными декораторами.

Помимо ядер, существуют и другие функции для GPU, которые называются функциями устройств (device functions). Это всегда функции, скомпилированные специально для запуска на различных устройствах, но вызываемые ядрами во время их работы на GPU (а не хостом).

Что касается памяти, необходимо сделать некоторые пояснения. Те фрагменты программы, которые выполняются на центральном процессоре, имеют доступ, как и все традиционные программы Python, к системной памяти, которую мы будем называть памятью хоста (host memory). Что касается ядер, они будут выполняться на графических процессорах, и, следовательно, им потребуется доступ к специальной памяти на этих картах, которую мы будем называть памятью устройства (device memory).

В модели программирования GPU вам придется явным образом выделять этот тип памяти (память хоста и память устройства) и управлять обменом данными между ними. Поскольку GPU оптимально работают с векторами, они будут использоваться в модели массивов NumPy на основе CUDA, специально оптимизированной для этих операций передачи и выделения памяти на графических процессорах.

Перечислив еще раз основные компоненты модели программирования графических процессоров с помощью CUDA, получаем следующий список:

- хост (CPU);
- устройство (GPU);
- ядро (функция GPU, запускаемая хостом и выполняемая на устройстве);
- функция устройства (функция GPU, запускаемая на устройстве и выполняемая на устройстве);
- память хоста (основная память системы);
- память устройства (память на карте GPU).

Логическая иерархия программной модели GPU

Как было отмечено выше, модель выполнения CUDA отличается от традиционной последовательной модели, используемой для программирования центрального процессора. В CUDA код выполняется несколькими потоками одновременно (нередко сотнями или тысячами). Помимо многоуровневой фи-

зической структуры, включающей блоки SM и SP, модель программирования CUDA предусматривает логическое разделение, структурированное по иерархии потоков, которые сгруппированы в сетки (grids) и блоки (blocks). Типичная схема логической иерархии показана на рис. 6.2.

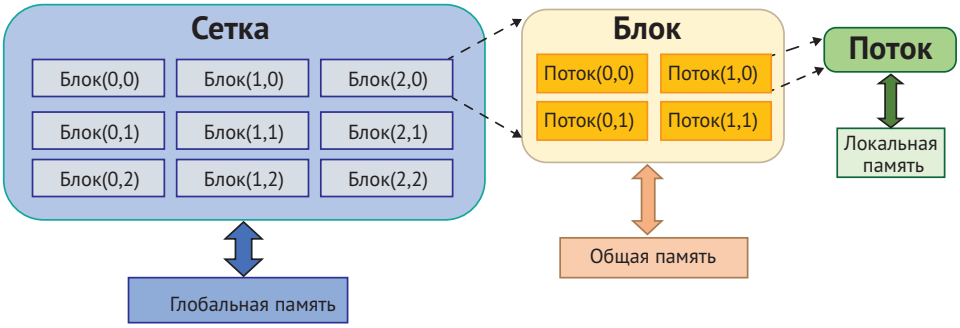


Рис. 6.2. Логическая иерархия потоков в GPU

В Numba имеется полный набор функций, классов и атрибутов, которые можно использовать для управления этой иерархией и составляющими ее элементами. Как показано на рис. 6.2, каждому уровню иерархии соответствует определенный тип памяти в GPU:

- глобальная память;
- общая память;
- локальная память.

Разрабатывая программу с помощью Numba (а также с помощью других библиотек, использующих CUDA), необходимо всегда учитывать эти различные типы памяти. Создание эффективных алгоритмов непременно предполагает их оптимизированное использование, минимизацию доступа и обмена данными, а также операций, требующих ресурсов времени.

Установка CUDA

Первым делом необходимо установить набор инструментов CUDA из интернета. Его можно бесплатно загрузить на сайте разработчиков NVIDIA на специальной странице (*https://developer.nvidia.com/cuda-downloads*). После загрузки страницы откроется окно, в котором можно выбрать одну из нескольких опций для выбора среды разработки, как показано на рис. 6.3.

В нашем случае после выбора характеристик системы (Windows 11, x86_64) была автоматически загружена следующая версия:

```
cuda_11.7.1_516.94_windows.exe
```

Однако в вашем случае вы, безусловно, получите наиболее свежую версию. После загрузки исполняемого файла установите его на свой компьютер. Следуйте инструкциям, которые будут отображаться во время установки.

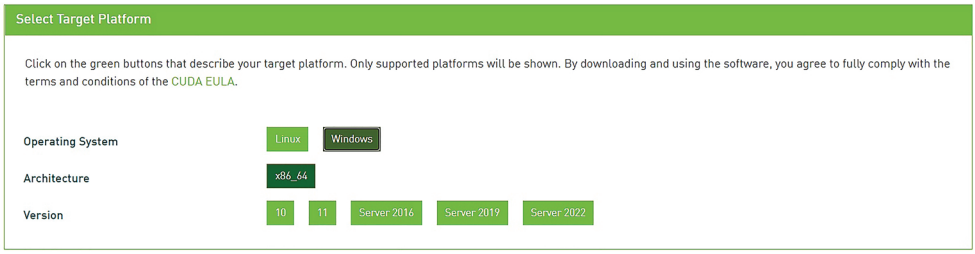


Рис. 6.3. Окно опций системы разработки

Установка Numba для CUDA

Установка Numba с помощью платформы Anaconda очень проста. Для установки Numba просто откройте консоль виртуальной среды, которую вы специально подготовили для работы с GPU, и введите следующую команду:

```
conda install numba cudatoolkit
```

Как видите, также необходимо установить пакет `cudatoolkit`. Многие библиотеки, использующие графические процессоры NVIDIA, уже содержат в себе все необходимые подключаемые модули для этого пакета, но Numba – это библиотека, которая имеет множество применений и не всегда используется для работы с графическими процессорами такого типа. Поэтому необходима отдельная установка `cudatoolkit`.

Объявление и вызов ядра

Перед запуском программы было показано, что части кода, предназначенные для выполнения на графических процессорах, определяются как функции ядра. Этот тип функций имеет две основные характеристики, которые отличают их от обычных функций в Python. Прежде всего ядра – это функции, которые не могут возвращать значение явным образом, поэтому они не могут использовать return для возврата значения за пределы функции. Эти функции работают исключительно с данными из массивов, переданных в качестве аргументов функции, и поэтому один из этих массивов должен быть определен для хранения результатов вычислений. Также при объявлении ядра необходимо учитывать иерархию потоков, которая будет установлена при вызове. Поэтому необходимо указать количество блоков потоков, участвующих в вычислениях, и количество потоков, присутствующих в каждом блоке.

В коде Python можно запросто определить часть кода, которая реализует функцию ядра, с помощью специального декодера `@cuda.jit`.

```
@cuda.jit
def kernel_func(array):
    """
Code here
    """
```

После определения ядра вам при необходимости понадобится вызвать его в коде. Однако перед этим, как уже было сказано, необходимо определить иерархию потоков, которая должна быть настроена и использована для его выполнения. После этого мы установим количество потоков для каждого блока, а затем количество блоков на сетку.

```
threadsperblock = 32
blockspergrid = (array.size // threadsperblock) + 1
```

Совершенно очевидно, что, определяя эти два параметра таким образом, мы сможем гарантировать, что каждый раз будет учитываться размер используемого массива и что будет организовано соответствующее количество потоков в иерархии, подходящей для наиболее оптимального их вычисления.

Затем мы вызовем функцию `kernel`, указав эти параметры в качестве индексов функции. Произведение этих двух значений фактически будет соответствовать потокам, используемым в вычислениях.

```
kernel_func[blockspergrid, threadsperblock](array)
```

После вызова функция `kernel` будет скомпилирована и отправлена в качестве ядра на соответствующее устройство для выполнения всеми вызванными потоками. Все ядра будут работать в параллельном асинхронном режиме, каждое из них будет запускать ядро независимо. Для синхронизации вычислений можно добавить `cuda.synchronize()`. В этом случае код дождется завершения работы всех ядер, после чего продолжит свое выполнение.

Как будет показано в следующих примерах, выбор подходящей иерархии потоков имеет решающее значение для оптимальной производительности вычислений на GPU. Размер блока определяет количество потоков, которые будут совместно использовать определенную область памяти (общую память). Он также должен быть достаточно большим, чтобы обеспечить вычисление входных данных за один шаг.

Кроме того, в объявлении функции ядра нам необходимо оптимально управлять индексами задействованных потоков и блоков, чтобы каждый поток мог точно идентифицировать элемент входного массива, с которым он работает. Для этого CUDA предоставляет специальные объекты. Например, `threadIdx`, `threadIdy` и `threadIdz` определяют индексы различных потоков, используемых в блоке по трем измерениям. А `blockIdx`, `blockIdy` и `blockIdz` указывают индексы различных блоков в пределах сетки. Размер блоков можно указать с помощью `blockDim.x`, `blockDim.y` и `blocDim.z`.

В свете этого, если вы хотите, например, реализовать ядро, которое выполняет только инкремент на `1` для всех элементов в массиве, нам придется использовать предыдущие объекты индекса следующим образом:

```
@cuda.jit
def add_one(array):
    tx = cuda.threadIdx.x
    ty = cuda.blockIdx.x
```

```
dim = cuda.blockDim.x

pos = tx + ty * dim
if pos < array.size:
    array[pos] += 1
```

Поскольку рассматриваемый массив является одномерным, мы можем рассматривать внутри ядра одномерную геометрию, в которой присутствует только x. Таким образом, итерация между всеми индексами, участвующими в вычислении, может быть связана с положением элементов массива с помощью следующего выражения:

$$pos = tx + ty * dim.$$

Кроме того, во время вычисления полезно установить условие, чтобы гарантировать наличие полученного индекса в границах массива. Далее мы добавляем это условие.

$$if\ pos < array.size:$$

В конечном итоге все ядра, индексированные с помощью pos с каждым элементом массива, увеличат свое значение на одну единицу одновременно, параллельно и асинхронно.

Функции устройства

Функции устройства несколько отличаются от функций ядра в плане их объявления. Для этого достаточно добавить к декоратору `device = True`.

```
@cuda.jit(device=True)
def device_func(a, b):
    return a + b
```

В отличие от функций `kernel`, функция `device` может возвращать значение, как и все обычные функции.

Пример программирования с Numba

Полученных к настоящему моменту знаний о Numba вполне достаточно, чтобы применить их на практике и скомпилировать небольшой пример кода. Используем ранее рассмотренное ядро, которое увеличивает значения элементов массива на единицу.

Напишем следующий код:

```
import numpy as np
from numba import cuda

@cuda.jit
def add_one(a):
```

```
    tx = cuda.threadIdx.x
    ty = cuda.blockIdx.x
    dim = cuda.blockDim.x

    pos = tx + ty * dim
    if pos < a.size:
        a[pos] += 1


n = 10
a_host = np.random.random(n)
print("Vector a: %s" %a_host)
a_dev = cuda.to_device(a_host)

threadsperblock = 128
blockspergrid = (a_host.size // threadsperblock) + 1
add_one[threadsperblock, blockspergrid](a_dev)

a_host = a_dev.copy_to_host()
print("New Vector a: %s" %a_host)
```

Как можно видеть, сначала мы импортировали необходимые модули, чтобы иметь возможность интегрировать ядра в код. В частности, для тех, кто хочет работать с графическими процессорами NVIDIA, мы импортируем часть, связанную с CUDA.

```
from numba import cuda
```

Также важна библиотека `numpy`, поскольку предпочтительными данными для этой технологии как по распределению памяти, так и по передаче являются именно массивы NumPy. Поэтому мы также импортировали этот модуль.

```
import numpy asn p
```

Затем мы скопировали ядро, которое ранее определили для примера.

```
@cuda.jit
def add_one(a):
    tx = cuda.threadIdx.x
    ty = cuda.blockIdx.x
    dim = cuda.blockDim.x

    pos = tx + ty * dim
    if pos < a.size:
        a[pos] += 1
```

Затем мы определили простой массив `a_host` (поскольку он находится в памяти хоста) из `10` элементов, значения которых генерируются случайным образом (числа с плавающей запятой).

```
n = 10
a_host = np.random.random(n)
print("Vector a: %s" %a_host)
```

На этом этапе мы перенесли этот массив в память устройства с помощью функции `cuda.to_device()`, которая выделяет в памяти устройства пространство, идентичное вектору `a_host`, а затем копирует его содержимое, обозначенное массивом `a_dev` (поскольку он находится в памяти устройства).

Следующим шагом является определение иерархии потоков, участвующих в выполнении ядра. Здесь мы также скопировали уже известные определения.

```
threadsperblock = 128
blockspergrid = (a_host.size // threadsperblock) + 1
```

Мы вставили большое количество потоков (`n = 128`), чтобы избежать вывода вот такого предупреждающего сообщения при выполнении:

```
NumbaPerformanceWarning: Grid size 10 will likely result in GPU
under-utilization due to low occupancy.
```

Это сообщение об ошибке (`NumbaPerformanceWarning: Размер сетки 10, вероятно, приведет к неполной загрузке графического процессора из-за низкой заполняемости`) появляется из-за того, что программа рекомендует использовать сетки такого размера, который сопоставим с количеством SP в GPU и не слишком мал, чтобы не допустить недостаточного использования потенциала этого оборудования.

После определения соответствующей иерархии для потоков функция `kernel` вызывается на массиве `a_dev` с только что определенными параметрами.

```
add_one[threadsperblock, blockspergrid](a_dev)
```

После вызова ядра необходимо обработать результат вычисления, а именно вектор `a_dev`, элементы которого были увеличены на одно значение. Но этот вектор находится в GPU, поэтому его необходимо перенести обратно в память хоста. Для этого используется функция `copy_to_host()`, которая копирует область памяти из GPU (устройства) на хост.

```
a_host = a_dev.copy_to_host ()
```

Теперь переменная `a_host` содержит обновленные значения. Мы можем вывести ее результаты на экран и сравнить с исходными значениями до вычисления.

```
print("New Vector a: %s" %a_host)
```

Запустив код полностью, мы получим следующий результат:

```
Vector a: [0.41384874 0.45213153 0.65042446 0.08814434 0.1537844 0.1184357
0.61263638 0.86474909 0.91423873 0.34284854]
```

```
New Vector a: [1.41384874 1.45213153 1.65042446 1.08814434 1.1537844
1.1184357 1.61263638 1.86474909 1.91423873 1.34284854]
```

Как видите, все вычисления были выполнены корректно.

Дальнейшие изменения

Продолжим модифицировать написанный ранее код и рассмотрим несколько других полезных концепций. Обратимся к определению геометрии потока (thread geometry). Как было показано выше, управление потоками может быть сложным и трудным для понимания на базовом уровне. К счастью, существует альтернативный метод, который значительно упрощает задачу: определение абсолютного положения.

В действительности очень простые алгоритмы, такие как рассмотренный в нашем примере, как правило, используют все индексы потоков одним и тем же способом. В Numba есть функции, которые значительно упрощают задачу определения положений индексов потоков в ядре.

Функция `cuda.grid(ndim)` дает возможность узнать абсолютное положение текущего потока в текущей иерархии. С помощью `ndim` мы определяем количество измерений, на которых основано ядро. Благодаря этой функции код функции `kernel` значительно упрощается.

```
@cuda.jit
def increment_by_one(an_array):
    pos = cuda.grid(1)
    if pos < a.size:
        a[pos] += 1
```

Подстановка этого выражения в предыдущий код никак не влияет на результаты.

Расширение на матрицы (массив 2D)

Еще одно возможное изменение заключается в расширении предыдущего примера на случай двумерных массивов, поэтому мы будем говорить о матрицах, а не о векторах.

Для этого расширения необходимо внести некоторые изменения в код. Для начала вместо вектора `a` определим матрицу `A`. Затем внесем соответствующие изменения и уменьшим количество элементов до `4` для лучшей наглядности результата.

```
n = 4
A_host = np.random.random(n*n).reshape(n,n)
print("Matrix A: \n %s" %A_host)
A_dev = cuda.to_device(A_host)
```

Таким образом, мы заменили исходный вектор матрицей 4×4 как на хосте (`A_host`), так и на устройстве (`A_dev`). Затем мы модифицируем функцию ядра

таким образом, чтобы можно было увеличить все элементы матрицы на одно значение.

```
@cuda.jit
def add_one_2D(A):
    x, y = cuda.grid(2)
    if x < A.shape[0] and y < A.shape[1]:
        A[x, y] += 1
```

Благодаря абсолютному позиционированию можно очень просто определить двухмерную функцию ядра. Что касается иерархии потоков, то для них потребуется новая геометрия, способная оптимально взаимодействовать с двухмерными векторами. Таким образом, чтобы можно было назначить поток для каждого элемента матрицы, необходимо определить новую иерархию следующим образом:

```
import math
threadsperblock = (128,128)
blockspergrid_x = math.ceil(A_host.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(A_host.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
add_one_2D[threadsperblock, blockspergrid](A_dev)
```

Наконец, перенесем полученный результат с устройства обратно на хост и отобразим его в выводе.

```
A_host = A_dev.copy_to_host()
print("New Matrix A: \n %s" %A_host)
```

Рассмотрим полностью измененный код.

```
import numpy as np
from numba import cuda
import math

@cuda.jit
def add_one_2D(A):
    x, y = cuda.grid(2)
    if x < A.shape[0] and y < A.shape[1]:
        A[x, y] += 1

n = 4
A_host = np.random.random(n*n).reshape(n,n)
print("Matrix A: \n %s" %A_host)
A_dev = cuda.to_device(A_host)

threadsperblock = (128,128)
blockspergrid_x = math.ceil(A_host.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(A_host.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
```

```
add_one_2D[threadsperblock, blockspergrid](A_dev)

A_host = A_dev.copy_to_host()
print("New Matrix A: \n %s" %A_host)
```

Запуск этого кода даст следующий результат:

```
Matrix A:
  [[0.00993464 0.99662295 0.6271492 0.39916735]
  [0.81896746 0.9282727 0.74802307 0.06418345]
  [0.54618904 0.92098866 0.13560085 0.41312367]
  [0.34544717 0.90828693 0.8784179 0.96724232]]
New Matrix A:
  [[1.00993464 1.99662295 1.6271492 1.39916735]
  [1.81896746 1.9282727 1.74802307 1.06418345]
  [1.54618904 1.92098866 1.13560085 1.41312367]
  [1.34544717 1.90828693 1.8784179 1.96724232]]
```

Передача данных через очередь

Еще одно изменение, которое можно внести и которое применимо ко всем примерам (как для векторов, так и для матриц), состоит в использовании очереди для передачи данных на GPU и с GPU. Редактирование в данном случае очень простое, для этого достаточно создать поток CUDA.

```
stream = cuda.stream()
```

Затем добавьте его к двум функциям, которые управляют передачей данных.

```
A_dev = cuda.to_device(A_host, stream=stream)
A_host = A_dev.copy_to_host(stream=stream)
```

В результате вы получите более эффективную передачу данных, управляемую специальной очередью, которая обеспечивает корректный и эффективный обмен данными между хостом и устройством.

Суммирование двух матриц

Расширим пример с матрицами еще больше и добавим операцию между несколькими двумерными векторами. В этом случае мы сложим элементы двух матриц `A` и `B`, чтобы получить третью матрицу. В этом примере рассматривается способ добавления нескольких входных векторов в ядро и управление ими в остальной части кода.

Теперь у нас больше нет одного вектора (матрицы), а есть три, один из которых будет пустым, поскольку в нем будут храниться результаты вычислений.

```
n = 4
A_host = np.random.random(n*n).reshape(n,n)
```

```
B_host = np.random.random(n*n).reshape(n,n)
C_host = np.zeros((n,n))
print("Matrix A: \n %s" %A_host)
print("Matrix B: \n %s" %B_host)
```

Передача данных с хоста на устройство и обратно должна осуществляться отдельно для каждого устройства.

```
Stream = cuda.stream()
A_dev = cuda.to_device(A_host, stream=stream)
B_dev = cuda.to_device(B_host, stream=stream)
C_dev = cuda.to_device(C_host, stream=stream)
```

Передача данных с устройства на хост: в данном случае нам нужно будет передать только `C_dev`, поскольку только в нем содержатся результаты вычислений ядра. Раз уж `A_dev` и `B_dev` используются только для чтения и идентичны данным на хосте, нет необходимости в их повторном копировании.

```
C_host = C_dev.copy_to_host(stream=stream)
print("Matrix C: \n %s" %C_host)
```

Теперь, когда мы разобрались с двумерными векторами, перейдем к ядру, которое потребует некоторых изменений, хотя все по-прежнему остается довольно простым.

```
@cuda.jit
def add_one_2D(A,B,C):
    x, y = cuda.grid(2)
    if x < A.shape[0] and y < A.shape[1]:
        C[x,y] = A[x, y] + B[x, y]
```

Положение потоков в предыдущем примере остается неизменным, поскольку сохраняется согласованность вычислений между элементами двух матриц `A` и `B`. Только соответствующие элементы `A` и `B`, а именно те, которые имеют одинаковые `x` и `y`, будут сложены, и результатом будет элемент `C`, который всегда находится в том же положении. В данном случае для этого было достаточно просуммировать элементы различных матриц.

```
C[x,y] = A[x, y] + B[x, y]
```

Что касается иерархии потоков, то здесь все по-прежнему без изменений.

Остается изменить вызов функции `kernel`, которая на этот раз требует трех аргументов, а именно трех матриц `A`, `B` и `C`.

```
add_one_2D[threadsperblock, blockspergrid](A_dev, B_dev, C_dev)
```

Перепишем измененный код целиком.

```
import numpy as np
from numba import cuda
```

```
import math

@cuda.jit
def add_one_2D(A,B,C):
    x, y = cuda.grid(2)
    if x < A.shape[0] and y < A.shape[1]:
        C[x,y] = A[x, y] + B[x, y]

n = 4
A_host = np.random.random(n*n).reshape(n,n)
B_host = np.random.random(n*n).reshape(n,n)
C_host = np.zeros((n,n))
print("Matrix A: \n %s" %A_host)
print("Matrix B: \n %s" %B_host)

stream = cuda.stream()
A_dev = cuda.to_device(A_host, stream=stream)
B_dev = cuda.to_device(B_host, stream=stream)
C_dev = cuda.to_device(C_host, stream=stream)

threadsperblock = (128,128)
blockspergrid_x = math.ceil(A_host.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(A_host.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
add_one_2D[threadsperblock, blockspergrid](A_dev, B_dev, C_dev)

C_host = C_dev.copy_to_host(stream=stream)
print("Matrix C: \n %s" %C_host)
```

Запуск этого кода даст следующий результат:

```
Matrix A:
  [[0.82645664 0.14077644 0.36426624 0.65039297]
  [0.10257125 0.30826244 0.82165832 0.90301203]
  [0.19817223 0.95656813 0.61620533 0.1200333]
  [0.8798604 0.11889796 0.31937825 0.0725755]]
Matrix B:
  [[0.83451822 0.37569725 0.91302873 0.061711]
  [0.04279979 0.13255439 0.96567441 0.51154525]
  [0.8677725 0.34670277 0.2593976 0.50304275]
  [0.91463535 0.96564344 0.91629866 0.17960097]]
Matrix C:
  [[1.66097486 0.51647368 1.27729497 0.71210397]
  [0.14537104 0.44081682 1.78733273 1.41455728]
  [1.06594473 1.3032709 0.87560292 0.62307604]
  [1.79449575 1.0845414 1.23567691 0.25217646]]
```

Умножение матриц

Последнее изменение в предыдущем коде заключается в использовании умножения матриц, например A * B = C. Это довольно затратно для последовательной программы, выполняемой на центральном процессоре. Вместо этого, как оказалось, эта операция идеально подходит для обработки на графических процессорах, которые демонстрируют действительно впечатляющую производительность по сравнению с центральным процессором.

```
import numpy as np
from numba import cuda
import math

@cuda.jit
def matmul(A, B, C):

    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp

n = 4
A_host = np.random.random(n*n).reshape(n,n)
B_host = np.random.random(n*n).reshape(n,n)
C_host = np.zeros((n,n))
print("Matrix A: \n %s" %A_host)
print("Matrix B: \n %s" %B_host)

stream = cuda.stream()
A_dev = cuda.to_device(A_host, stream=stream)
B_dev = cuda.to_device(B_host, stream=stream)
C_dev = cuda.to_device(C_host, stream=stream)

threadsperblock = (128,128)
blockspergrid_x = math.ceil(A_host.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(A_host.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
matmul[threadsperblock, blockspergrid](A_dev, B_dev, C_dev)

C_host = C_dev.copy_to_host(stream=stream)
print("Matrix C: \n %s" %C_host)
```

Как можно видеть, разница при исключении ядра очень незначительна. При выполнении приведенного выше кода будет получен следующий результат:

```
Matrix A:
  [[0.75926968 0.92657886 0.68471105 0.17324253]
  [0.40790221 0.45988319 0.39122078 0.6198121 ]
  [0.10698571 0.43140089 0.58631454 0.38644777]
  [0.17504734 0.76003441 0.68634873 0.94149139]]
Matrix B:
  [[0.79381329 0.96132662 0.16316954 0.5995909 ]
  [0.74119391 0.85658564 0.85753895 0.65944347]
  [0.98787907 0.04619411 0.70690497 0.51689578]
  [0.85581547 0.67852783 0.84931887 0.30510469]]
Matrix C:
  [[2.11416833 1.6727798 1.54963095 1.47305894]
  [1.58158443 1.22468845 1.263899 0.93916938]
  [1.31461425 0.76167987 1.13008593 0.76960271]
  [2.18606024 1.48984562 1.96513125 1.24818073]]
```

Выбранный нами пример требует небольшого времени выполнения, поскольку были выбраны матрицы небольшого размера. Но с увеличением размера ситуация быстро меняется. Теперь давайте добавим счетчик `time` из модуля `time`. Мы увидим, например, как время выполнения меняется при таких операциях в зависимости от количества потоков на блок. Изменим код, чтобы можно было записывать время выполнения.

```
import numpy as np
from numba import cuda
import time
import math

@cuda.jit
def matmul(A, B, C):

    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp

print("Matrix Size: ")
n = input()
print("\n")
A_host = np.random.random(n*n).reshape(n,n)
B_host = np.random.random(n*n).reshape(n,n)
C_host = np.zeros((n,n))

stream = cuda.stream()
A_dev = cuda.to_device(A_host, stream=stream)
```

```
B_dev = cuda.to_device(B_host, stream=stream)
C_dev = cuda.to_device(C_host, stream=stream)

threadsperblock = (128,128)
blockspergrid_x = math.ceil(A_host.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(A_host.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
matmul[threadsperblock, blockspergrid](A_dev, B_dev, C_dev)

t1 = time.perf_counter()
C_host = C_dev.copy_to_host(stream=stream)
t2 = time.perf_counter()
elapsed_time = t2 - t1
print("Elapsed time %s" %elapsed_time)
```

Как видно из кода, мы исключили вывод значений матриц и вставили размер матриц из консоли во время выполнения программы, чтобы не приходилось каждый раз изменять код. Другим решением могло бы быть использование входных параметров при запуске программы.

Если мы измерим время, затрачиваемое при изменении размера матриц и размера блоков (количество потоков на блок), получим таблицу, похожую на приведенную ниже табл. 6.1.

Таблица 6.1. Затраты времени на умножение матриц

Размер матрицы	Потоков на блок			
	n = 128	n = 256	n = 512	n = 1024
1024	0,06	0,06	0,22	0,71
2048	0,68	0,42	0,42	1,36
3072	2,14	1,78	1,28	2,26
4092	7,24	4,87	2,04	3,28
5120	–	12,32	7,22	5,29
6144	–	17,08	14,79	10,37
7168	–	40,73	25,6	17,51
8192	–	67,36	40,24	23,36
9216	–	–	63,97	46,58
10 240	–	–	121,47	60,23

Если построить график значений, приведенных в табл. 6.1, получится следующий любопытный результат, см. рис. 6.4.

Вы можете увидеть зависимость времени, затрачиваемого на умножение матриц, от увеличения их размера. Приведены три зависимости, каждая с разным количеством потоков на блок. Как можно видеть, чем выше это число, тем выше эффективность работы графического процессора.

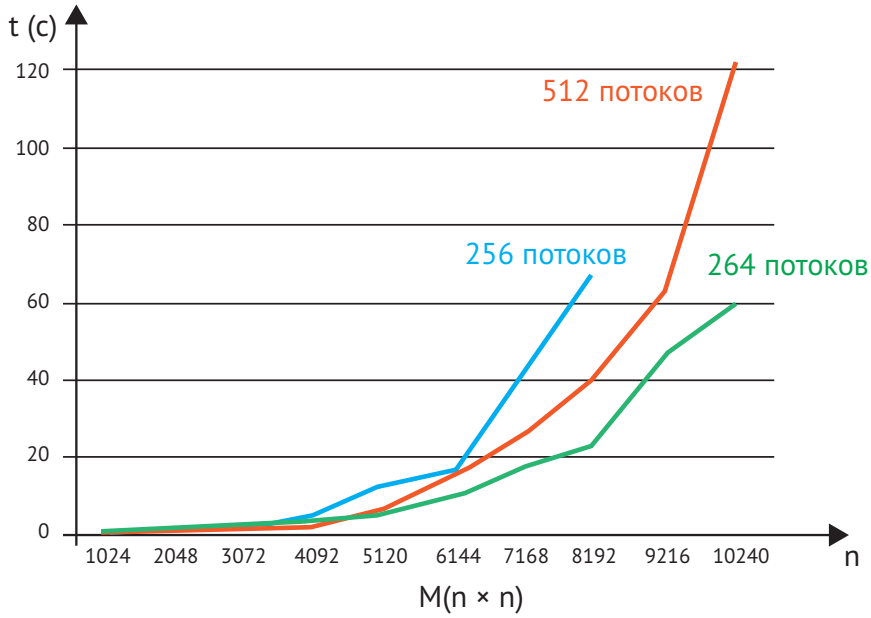


Рис. 6.4. Время умножения матриц с Numba для CUDA при изменении количества потоков в блоке

PyOpenCL

Еще одной библиотекой, которая дает возможность максимально использовать потенциал графических процессоров с помощью Python, является PyOpenCL. Этот пакет представляет собой оболочку Python для Open Computing Language (OpenCL). Этот фреймворк, основанный на использовании языка C99, позволяет работать на различных и разнородных платформах с использованием как центрального, так и графического процессов. Программы, написанные с помощью OpenCL, далее компилируются соответствующими драйверами на разных языках, специфичных для каждой технологии. По этой причине его использование не ограничивается исключительно картами NVIDIA и библиотекой CUDA. Поэтому OpenCL является отличным решением для расширения программирования на многие другие различные среды, которые используют графические процессоры различных производителей, таких как IBM, AMD или Intel.

Изначально OpenCL была создана Apple, но позже этот проект был передан Khronos Group, некоммерческому консорциуму, который по сей день продолжает работу над проектом и выпускает новые версии с постоянными обновлениями. Поскольку OpenCL не зависит исключительно от CUDA, она является отличной альтернативой для тех, кто не имеет графических карт NVIDIA, но хочет использовать другие GPU от разных производителей для параллельных вычислений по той же методике.

Установка PyOpenCL

Подготовить компьютер под управлением *Windows* к работе с OpenCL совсем несложно. Необходимо установить последние версии драйверов для всех устройств, совместимых с OpenCL.

Вы можете просто установить `pyOpenCL` с помощью стандартной установки `conda`:

```
conda install -c conda-forge pyopencl
```

Для использования преимуществ драйверов процессоров Intel можно также добавить

```
conda install intel-opencl-rt
```

Для других платформ существуют специальные пакеты, которые можно найти в интернете. Разумеется, если мы хотим использовать графические процессоры NVIDIA, в системе должен быть установлен набор инструментов CUDA. Если вы использовали Numba в первой части этой главы, дополнительные установки не потребуются.

После завершения установки этих пакетов в системе вы можете проверить, все ли прошло успешно, с помощью следующего примера. Очень полезной в этом плане функцией, позволяющей исследовать работу вашей системы, является `get_platforms()`. Эта функция предоставляет список платформ, поддерживающих OpenCL, и с помощью функции `get_devices()` можно получить дополнительную информацию об устройствах (графических процессорах), установленных на вашей системе. Затем мы используем эти две функции для получения общего представления об используемой системе с помощью следующего кода:

```
import pyopencl as cl

for platform in cl.get_platforms():
    print("Platform %s " %platform.name)
    print(" Vendor: %s " %platform.vendor)
    print(" Version: %s " %platform.version)
    for device in platform.get_devices():
        print(" Device %s " %device.name)
        print(" Max Clock Speed: %s MHz " %device.max_clock_frequency)
        print(" Compute Units: %s " %device.max_compute_units)
        print(" Local Memory: %s Kb" %(device.local_mem_size/1024.0))
        print(" Global Memory: %s Gb " %(round(device.global_mem_size/
1073741824.0,2)))
```

Запуск этого кода даст примерно подобный результат:

```
Platform NVIDIA CUDA
  Vendor: NVIDIA Corporation
```

```
    Version: OpenCL 3.0 CUDA 11.7.101
      Device NVIDIA GeForce GTX 1650 Ti
        Max Clock Speed: 1485 MHz
        Compute Units: 16
        Local Memory: 48.0 Kb
        Global Memory: 4.0 Gb
Platform Intel(R) OpenCL HD Graphics
  Vendor: Intel(R) Corporation
  Version: OpenCL 2.1
    Device Intel(R) UHD Graphics
      Max Clock Speed: 1150 MHz
      Compute Units: 24
      Local Memory: 64.0 Kb
      Global Memory: 6.31 Gb
```

Как можно видеть, мы получаем список платформ в нашей системе, которые совместимы с OpenCL, и устройств, которые можно использовать в них. В приведенном примере есть две видеокарты: NVIDIA и интегрированная в материнскую плату Intel.

Модель программирования PyOpenCL

Программы, созданные с помощью PyOpenCL, необходимо структурировать очень точно, с учетом конкретной архитектуры используемых систем, таких как GPU, и основополагающих концепций. Поэтому в коде всегда необходимо будет ссылаться на хост (CPU) и устройства (GPU).

Для начала необходимо разработать программу таким образом, чтобы она могла запускаться на хосте. Затем код будет скомпилирован с указанием различных фрагментов, предназначенных для выполнения на устройствах. В результате будет сгенерировано несколько исполняемых пакетов, каждый из которых компилируется в соответствии с особенностями соответствующего устройства (модели разных производителей имеют разные драйверы), после чего они будут отправлены на выполнение во время работы программы. Эти пакеты идентифицируются ядрами, которые представляют собой фрагменты кода для выполнения на устройствах.

Системы под управлением фреймворка OpenGL можно рассматривать как распределенные системы (мы говорили о них в предыдущей главе), где разнородные процессоры соединяются между собой путем обмена ядрами (вместо сообщений). Таким образом, как и в распределенных системах, потребуется система для управления обменом и распределением этих ядер. Поэтому в OpenCL присутствует **Command Queue** (очередь команд) – структура данных, где собираются и накапливаются различные получаемые устройством ядра. Эта структура отвечает за управление порядком выполнения различных ядер. Кроме того, в рамках модели программирования OpenCL также определяется **Context** (контекст), который идентифицирует определенную группу устройств и облегчает управление передачей ядер и данных в системе хост–устройство, как на рис. 6.5.

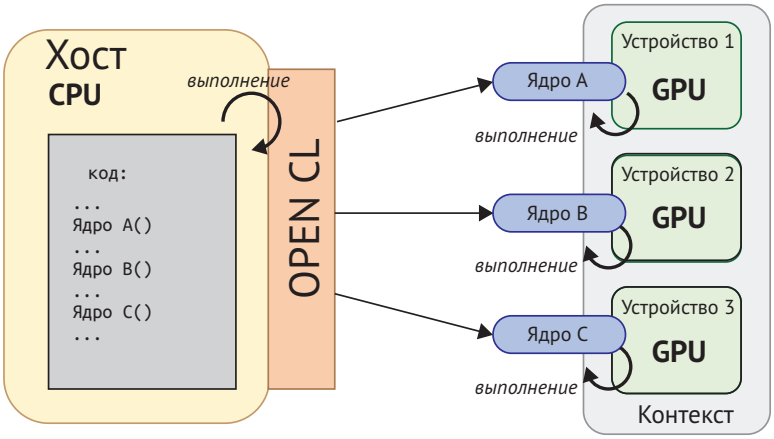


Рис. 6.5. Модель программирования OpenCL

Разработка программ с PyOpenCL

Реализация программы с использованием PyOpenCL должна соответствовать рекомендациям модели программирования OpenCL. Поскольку OpenCL главным образом основана на операциях с массивами NumPy, мы можем начать с простого примера, где элементы одного из этих векторов будут удвоены. Эта операция будет реализована в ядре, которое затем будет скомпилировано для запуска на GPU (устройстве).

Сначала следует импортировать базовые модули, необходимые для реализации программы с помощью PyOpenCL:

```
import numpy as np
import pyopencl as cl
```

На этом этапе в рамках модели программирования GPU первым шагом будет определение области памяти в хосте и содержащихся в ней данных. С помощью OpenCL это сделать очень просто при использовании массивов NumPy. В нашем простом примере мы можем определить пару векторов: `a_host` из 4 элементов со значениями для вычислений и `res_host`, также из 4 элементов, который будет содержать результаты вычислений.

```
a_host = np.array([0.1 , 1.4, 2.3, 1.7])
a_host = a_host.astype(np.float32)
res_host = np.zeros((4), dtype=np.float32)
```

Следующим шагом будет определение объектов управления распределением ядра: контекст будет создан с помощью конструктора `Context()`, а его очередь – с помощью конструктора `CommandQueue()`.

```
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
```

Поскольку мы имеем дело с графическими процессорами, необходимо явным образом управлять распределением памяти на устройствах и передачей данных из памяти хоста в память устройств. В OpenCL это осуществляется простым способом с помощью класса `Buffer`. Затем мы определяем буфер, который будет заниматься передачей значений из `a_host` в область памяти устройства GPU, определяемую переменной `a_dev`. Эта область памяти, поскольку она имеет только функции чтения, будет определена как `READ_ONLY`. То же самое будет сделано с вектором, который будет содержать результаты. Создается вектор `res_dev` того же размера, что и `a_dev`, но, поскольку он предназначен для хранения результатов, он будет памятью `WRITE_ONLY`.

```
mf = cl.mem_flags
a_dev = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_host)
res_dev = cl.Buffer(ctx, mf.WRITE_ONLY, a_host.nbytes)
```

После завершения выделения памяти в устройстве и последующей передачи данных можно приступить к реализации ядра. В PyOpenCL код `kernel` вставляется в качестве аргумента конструктора класса `Program()`.

```
prg = cl.Program(ctx, """
__kernel void doubled(
    __global const float *a_dev, __global float *res_dev)
{
  int gid = get_global_id(0);
  res_dev[gid] = a_dev[gid]*2;
}
""").build()
```

Как можно видеть, ядро имеет структуру классической функции, где аргументы представляют как входные, так и возвращаемые значения. На самом деле ядра не могут иметь возвращаемых значений, но работают с векторами, переданными в качестве аргументов. После определения ядра его необходимо вызвать для выполнения. После объявления и вызова ядра оно назначается очереди, которая позаботится не только о его отправке на устройство, но и о передаче области памяти, содержащей результат, с устройства на хост с помощью метода `enqueue_copy()`.

```
knl = prg.doubled
knl(queue, a_host.shape, None, a_dev, res_dev)
cl.enqueue_copy(queue, res_host, res_dev)
```

Теперь, когда результаты скопированы на хост в массив `res_host`, полученные результаты могут быть выведены.

```
print("Vector a: %s" %a_host)
print("Result: %s" %res_host)
```

Рассмотрим весь только что написанный код.

```
import numpy as np
import pyopencl as cl

a_host = np.array([0.1 , 1.4, 2.3, 1.7])
a_host = a_host.astype(np.float32)
res_host = np.zeros((4), dtype=np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_dev = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_host)
res_dev = cl.Buffer(ctx, mf.WRITE_ONLY, a_host.nbytes)

prg = cl.Program(ctx, """
__kernel void doubled(
    __global const float *a_dev, __global float *res_dev)
{
  int gid = get_global_id(0);
  res_dev[gid] = a_dev[gid]*2;
}
""").build()

knl = prg.doubled
knl(queue, a_host.shape, None, a_dev, res_dev)
cl.enqueue_copy(queue, res_host, res_dev)

print("Vector a: %s" %a_host)
print("Result: %s" %res_host)
```

Запуск этого кода даст примерно следующий результат:

```
Choose platform:
[0]  <pyopencl.Platform 'NVIDIA CUDA' at 0x1612071dd60>
[1]  <pyopencl.Platform 'Intel(R) OpenCL HD Graphics' at 0x1611ff6cb80>
Choice [0]:0
Set the environment variable PYOPENCL_CTX='0' to avoid being asked again.
Vector a: [0.1 1.4 2.3 1.7]
Result: [0.2 2.8 4.6 3.4]
```

Вы сразу же увидите, что сначала вам будет предложено выбрать среду для работы (это происходит в том случае, если в вашей системе есть несколько платформ с поддержкой технологии OpenCL). Введите нужное значение. Если нажать **Enter**, программа выполнит вычисления на платформе по умолчанию (значение `0`). Чтобы изменить этот выбор позже, можно вставить в код переменную среды, как предлагает программа. Поэтому мы вставляем следующие инструкции в начальную часть предыдущего кода:

```
import os
os.environ['PYOPENCL_CTX'] = '0'
```

Таким образом, мы заранее сообщим компилятору, что хотим работать на платформе NVIDIA. Перезапустив программу после изменения, мы сразу получим желаемый результат.

```
Vector a: [0.1 1.4 2.3 1.7]
Result: [0.2 2.8 4.6 3.4]
```

Умножение матриц: пример

Как уже было сказано ранее, отличным примером измерения производительности GPU с помощью библиотеки Numba является умножение матриц. Давайте на этот раз воспользуемся для этого библиотекой PyOpenCL. Например, с помощью следующего кода можно умножить две матрицы `A * B` размером `8` и получить третью матрицу `C`, содержащую результат этой операции.

```
import pyopencl as cl
import numpy as np

import os
os.environ['PYOPENCL_CTX'] = '0'

n = 8

a = np.random.randint(10, size=(n*n))
b = np.random.randint(10, size=(n*n))
c = np.zeros((n*n), dtype=np.float32)

a = a.astype(np.float32)
b = b.astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, c.nbytes)

prg = cl.Program(ctx, """
    __kernel void multiply(ushort n,
    ushort m, ushort p, __global float *a,
    __global float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = 0.0f;
```

```
        int rowC = gid/p;
        int colC = gid%p;
        __global float *pA = &a[rowC*m];
        __global float *pB = &b[colC];
        for(int k=0; k<m; k++)
        {
          pB = &b[colC+k*p];
          c[gid] += (*(pA++))*(*pB);
        }
      }
""").build()

prg.multiply(queue, c.shape, None,
             np.uint16(n), np.uint16(n), np.uint16(n),
             a_buf, b_buf, c_buf)

cl.enqueue_copy(queue, c, c_buf)

print("Matrix A")
print(a.reshape(n,n))
print("Matrix B")
print(b.reshape(n,n))
print("Matrix A*B")
print(c.reshape(n, n))
```

Запуск этого кода даст следующий результат:

```
Matrix A
[[6. 4. 2. 5. 3. 7. 4. 7.]
 [3. 0. 2. 0. 8. 5. 4. 4.]
 [4. 6. 3. 3. 3. 8. 7. 2.]
 [7. 4. 7. 6. 0. 3. 3. 2.]
 [8. 1. 6. 1. 5. 9. 4. 4.]
 [9. 9. 6. 4. 4. 3. 5. 3.]
 [9. 4. 9. 0. 1. 8. 7. 4.]
 [7. 7. 8. 0. 8. 9. 2. 5.]]
Matrix B
[[6. 3. 1. 0. 0. 6. 8. 7.]
 [3. 0. 1. 0. 9. 3. 3. 4.]
 [2. 4. 7. 1. 5. 1. 8. 1.]
 [3. 3. 4. 8. 1. 4. 6. 3.]
 [7. 3. 3. 8. 4. 0. 8. 9.]
 [6. 3. 0. 6. 0. 8. 8. 4.]
 [3. 0. 8. 9. 2. 6. 3. 4.]
 [2. 0. 2. 7. 8. 6. 5. 3.]]
Matrix A*B
[[156.  71.  99. 193. 127. 192. 233. 167.]
```

```
 [128.  56.  81. 160.  82. 108. 176. 143.]
 [151.  66. 112. 176. 114. 175. 211. 157.]
 [117.  76. 112. 114.  99. 139. 203. 120.]
 [175.  93. 110. 172. 100. 181. 265. 178.]
 [172.  84. 134. 154. 165. 175. 257. 194.]
 [168.  90. 143. 156. 131. 205. 269. 169.]
 [205. 104. 120. 179. 179. 185. 308. 216.]]
```

Теперь внесем некоторые изменения в предыдущую программу и добавим измерение времени, затрачиваемого на вычисление произведения матриц.

```
import pyopencl as cl
import numpy as np
import time

import os
os.environ['PYOPENCL_CTX'] = '0'

n = 8

a = np.random.randint(10, size=(n*n))
b = np.random.randint(10, size=(n*n))
c = np.zeros((n*n), dtype=np.float32)

a = a.astype(np.float32)
b = b.astype(np.float32)

ctx = cl.create_some_context() queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, c.nbytes)

prg = cl.Program(ctx, """
    __kernel void multiply(ushort n,
    ushort m, ushort p,
    __global float *a,
    __global float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = 0.0f;
      int rowC = gid/p; int colC = gid%p;
      __global float *pA = &a[rowC*m];
      __global float *pB = &b[colC];
      for(int k=0; k<m; k++)
      {
        pB = &b[colC+k*p];
```

```
          c[gid] += (*(pA++))*(*pB);
        }
      }
       """).build()


t1 = time.perf_counter() prg.multiply(queue, c.shape, None,
                                      np.uint16(n), np.uint16(n),
np.uint16(n),
                                      a_buf, b_buf, c_buf)
t2 = time.perf_counter()
cl.enqueue_copy(queue, c, c_buf)
elapsed_time = t2 - t1
print("Elapsed time %s" %elapsed_time)
```

Мы удалили вывод результатов вычислений матриц и вместо этого вставили функции `perf_counter()` модуля time для вычисления времени, затраченного на умножение матриц.

При запуске предыдущего кода мы получим следующий результат:

```
Elapsed time 0.007469899996067397
```

Таким образом, в качестве результата мы получили время в секундах. Эта информация будет полезна для оценки времени, необходимого для умножения матриц по мере увеличения их размера, который в нашем коде выражается переменной `n`. Мы проводим серию измерений на нашем компьютере, изменяя значение `n` для каждого выполнения, назначая ему, например, степени числа 2 (4, 8, 16, 32, 64 и т. д.). В результате получим таблицу значений, аналогичную табл. 6.2.

Таблица 6.2. Время, затраченное на умножение матриц

n	t(GPU)
4	0,0063
8	0,0056
16	0,0066
32	0,0058
64	0,0058
128	0,0065
256	0,0076
512	0,0082
1024	0,0086
2048	0,0164
4096	0,0484

Как видно из результатов, время выполнения почти одинаково до 1024, а затем увеличение происходит с увеличением степени 2. Это позволяет предположить, что время выполнения зависит от количества доступных потоков в графическом процессоре. До определенного значения параллельно будет работать n потоков, которые затратят одинаковое время на вычисление произведения матриц ($n \times n$). Таким образом, согласно этому механизму, вычисление произведения двух матриц (4 × 4) займет столько же времени, сколько и вычисление произведения двух матриц (512 × 512).

Для лучшего понимания потенциала программирования и вычислений на графических процессорах рассмотрим умножение двух матриц, выполняемое последовательно. Пример реализации произведения двух матриц `A` и `B` в классической программе на Python, не использующей графический процессор, представлен в следующем коде:

```
import numpy as np

n = 8
a = np.random.randint(10, size=(n*n))
b = np.random.randint(10, size=(n*n))
c = np.zeros((n*n), dtype=np.float32)

a = a.astype(np.float32)
b = b.astype(np.float32)


def matmul(n, A, B, C):
    # Итерация по строкам A.
    for i in range(n):
        # Итерация по столбцам B
        for j in range(n):
            tmp = 0.0
            # Итерация по строкам B.
            for k in range(n):
                tmp += A[i * n + k] * B[k * n + j]
            C[i * n + j] = tmp

matmul(n, a, b, c)
print("Matrix A")
print(a.reshape(n,n))
print("Matrix B")
print(b.reshape(n,n))
print("Matrix A*B")
print(c.reshape(n,n))
```

При выполнении кода мы получим результат, идентичный результату, полученному в предыдущем примере с PyOpenCL. Значения генерируются случай-

ным образом, поэтому результаты будут различаться при каждом запуске, но вычисления и результаты будут практически одинаковыми.

```
Matrix A
[[9. 6. 2. 0. 1. 5. 4. 2.]
 [4. 6. 1. 2. 1. 3. 0. 2.]
 [8. 9. 3. 3. 4. 3. 3. 0.]
 [5. 2. 0. 1. 4. 7. 6. 6.]
 [3. 5. 2. 6. 7. 6. 1. 8.]
 [0. 2. 2. 4. 0. 2. 5. 6.]
 [9. 9. 0. 2. 6. 7. 7. 3.]
 [8. 1. 4. 5. 8. 7. 7. 2.]]
Matrix B
[[6. 6. 8. 3. 0. 6. 6. 8.]
 [9. 8. 5. 9. 3. 9. 4. 4.]
 [9. 9. 6. 1. 9. 9. 0. 5.]
 [2. 6. 0. 7. 0. 7. 5. 2.]
 [2. 9. 9. 4. 9. 5. 1. 9.]
 [2. 4. 6. 3. 8. 0. 1. 7.]
 [5. 3. 3. 6. 0. 8. 8. 8.]
 [8. 4. 4. 7. 0. 5. 4. 3.]]
Matrix A*B
[[174. 169. 173. 140.  85. 173. 124. 188.]
 [115. 122. 103. 108.  60. 116.  70. 101.]
 [191. 222. 190. 172. 114. 221. 130. 202.]
 [150. 158. 170. 155.  98. 153. 126. 201.]
 [188. 234. 195. 206. 144. 206. 121. 203.]
 [121. 105.  73. 126.  40. 134.  94.  98.]
 [224. 253. 246. 230. 137. 250. 181. 280.]
 [184. 251. 236. 181. 167. 234. 156. 281.]]
```

На этом этапе мы также модифицируем этот код таким образом, чтобы иметь возможность выполнять измерения времени, затрачиваемого на умножение матриц при изменении их размера с изменением значения `n`.

```
import numpy as np
import time

n = 8
a = np.random.randint(10, size=(n*n))
b = np.random.randint(10, size=(n*n))
c = np.zeros((n*n), dtype=np.float32)

a = a.astype(np.float32)
b = b.astype(np.float32)

def matmul(n, A, B, C):
```

```
    # Iterate over the rows of A.
    for i in range(n):
        # Iterate over the columns of B
        for j in range(n):
            tmp = 0.0
            # Iterate over the rows of B.
            for k in range(n):
                tmp += A[i * n + k] * B[k * n + j]
            C[i * n + j] = tmp

t1 = time.perf_counter()
matmul(n, a, b, c)
t2 = time.perf_counter()
elapsed_time = t2 - t1
print("Elapsed time %s" %elapsed_time)
```

Выполнив серию вычислений для значений *n*, используемых с кодом в PyOpenCL, мы получим серию показателей времени, как показано в табл. 6.3.

Таблица 6.3. Сравнение времени выполнения для CPU и GPU

n	t(CPU)	t(GPU)
4	0,000074	0,0063
8	0,00035	0,0056
16	0,0034	0,0066
32	0,014	0,0058
64	0,12	0,0058
128	0,8	0,0065
256	6,77	0,0076
512	61,5	0,0082
1024		0,0086
2048		0,0164
4096		0,0484

Как можно видеть, в данном случае динамика изменения времени кардинально отличается от предыдущего случая. Мы наблюдаем экспоненциальный рост по мере увеличения размера используемых матриц, настолько значительный, что при превышении определенного размера время достигает недопустимых значений. Интересно отметить, что для небольших массивов (<32) последовательные вычисления гораздо более эффективны, чем вычисления с использованием графических процессоров с помощью PyOpenCL. На рис. 6.6 наглядно представлены временные тенденции для обоих типов программирования.

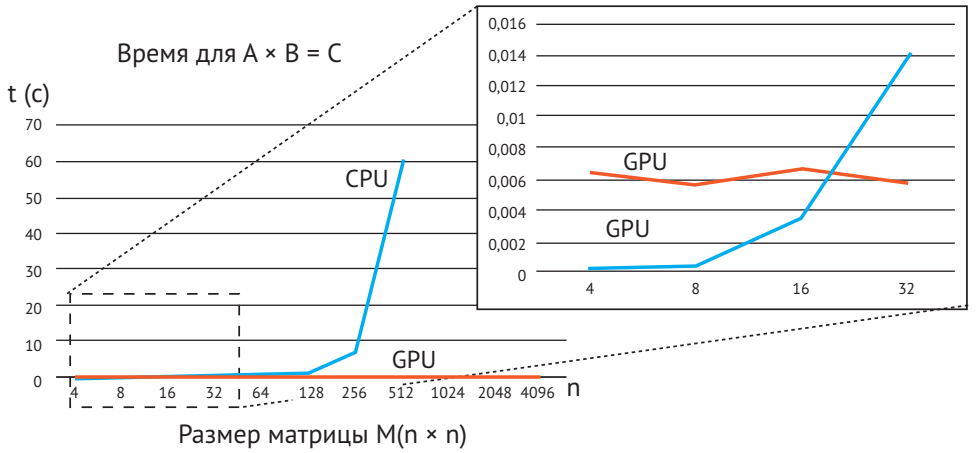


Рис. 6.6. Время на вычисление A * B на CPU (последовательная программа) и на GPU (с OpenCL)

Как видно из графика для небольших матриц (до порядка 16), последовательные вычисления, выполняемые на CPU для вычисления произведения матриц, оказываются более эффективными, чем вычисления на GPU с использованием параллельных потоков. Однако динамика затрачиваемого времени на CPU имеет экспоненциальный характер, и уже для матриц большего размера вычисления становятся непомерно длительными, а время вычислений постоянно увеличивается. При этом мы наблюдаем совершенно другой характер ведения вычислений на GPU, где время выполнения остается постоянным, несмотря на увеличение размера используемых для умножения матриц. Это в значительной степени обусловлено большим количеством потоков на графическом процессоре, которые способны чрезвычайно эффективно обрабатывать параллельные вычисления тысяч потоков одновременно.

Поэлементные вычисления с помощью PyOpenCL

С помощью PyOpenCL можно выполнять поэлементные вычисления неявным способом с использованием сложных выражений за один единственный вычислительный цикл. Для этого используется функция `ElementwiseKernel()`.

Рассмотрим следующий пример:

```
import numpy as np
import pyopencl as cl
import pyopencl.array
from pyopencl.elementwise import ElementwiseKernel

import os
os.environ['PYOPENCL_CTX'] = '0'

a_host = np.array([0.1 , 1.4, 2.3, 1.7])
```

```
a_host = a_host.astype(np.float32)
b_host = np.array([0.2 , 0.3, 1.0, 0.5])
b_host = b_host.astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_dev = cl.array.to_device(queue, a_host)
b_dev = cl.array.to_device(queue, b_host)

sum = ElementwiseKernel(ctx,
    "float *a_dev, float *b_dev, float *res_dev",
    "res_dev[i] = a_dev[i] + b_dev[i]",
    "sum")

res_dev = cl.array.empty_like(a_dev)
sum(a_dev, b_dev, res_dev)

print("Vector A: %s" %a_host)
print("Vector B: %s" %b_host)
print("Vector Result: %s" %res_dev)
```

Запуск этого кода даст следующий результат:

```
Vector A: [0.1 1.4 2.3 1.7]
Vector B: [0.2 0.3 1. 0.5]
Vector Result: [0.3 1.7 3.3 2.2]
```

Вычисления MapReduce с помощью PyOpenCL

Как и в предыдущем примере, имеется еще одна конструкция, которая допускает неявное использование сложных выражений, например, предназначенных для MapReduce. Для этой цели в PyOpenCL предусмотрена функция `ReductionKernel()`.

```
import numpy as np
import pyopencl as cl
import pyopencl.array
import pyopencl.reduction

import os
os.environ['PYOPENCL_CTX'] = '0'

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

ah = np.array([0.1 , 1.4, 2.3, 1.7])
```

```
ah = ah.astype(np.float32)
bh = np.array([0.2 , 0.3, 1.0, 0.5])
bh = bh.astype(np.float32)

a = cl.array.to_device(queue, ah)
b = cl.array.to_device(queue, bh)

krnl = pyopencl.reduction.ReductionKernel(ctx, np.float32, neutral="0",
        reduce_expr="a+b", map_expr="x[i]*y[i]",
        arguments="__global float *x, __global float *y")

res = krnl(a, b).get()

print("Vector A: %s" %a)
print("Vector B: %s" %b)
print("Vector Result: %s" %res)
```

Запуск этого кода даст следующий результат:

```
Vector A: [0.1 1.4 2.3 1.7]
Vector B: [0.2 0.3 1.  0.5]
Vector Result: 3.59
```

Эта программа для начала выполняет произведение между соответствующими элементами двух векторов `a` и `b` (процесс отображения, выраженный выражением `x [i] * y [i]`, которое присваивается параметру `map_expr`). Векторы `x` и `y`, передаваемые в качестве аргументов `kernel`, должны быть индексированы с помощью индекса `i`.

$$0{,}1 \times 0{,}2 = 0{,}02$$
$$1{,}4 \times 0{,}3 = 0{,}42$$
$$2{,}3 \times 1{,}0 = 2{,}3$$
$$1{,}7 \times 0{,}5 = 0{,}85$$

Все результаты будут сложены вместе в качестве операции редукции в выражении `a + b`, которое присвоено параметру `reduce_expr`. В этом случае формальные значения `a` и `b` используются для обозначения операндов бинарной операции редукции.

$$0{,}02 + 0{,}42 + 2{,}3 + 0{,}85 = 3{,}59$$

Чтобы рассмотреть другой пример, можно заменить предыдущее ядро `mapreduce` на это.

```
krnl = pyopencl.reduction.ReductionKernel(ctx, np.float32, neutral="0",
        reduce_expr='max(a,b)', map_expr="x[i]+y[i]",
        arguments="  global float *x,   global float *y")
```

В этот раз запуск кода даст следующий результат:

```
Vector A: [0.1 1.4 2.3 1.7]
Vector B: [0.2 0.3 1.  0.5]
Vector Result: 3.3
```

В этой операции складываются соответствующие элементы двух векторов, при этом выражение `x[i] + y[i]` присваивается параметру `map_expr`.

0,1 + 0,2 = 0,3
1,4 + 0,3 = 1,7
2,3 + 1,0 = 3,3
1,7 + 0,5 = 2,2

Из этих результатов будет выбрано наибольшее значение с помощью двоичного выражения в виде `max (a, b)`, присвоенного параметру `reduce_expr`.

Заключение

В этой главе мы рассмотрели возможности повышения эффективности параллельных вычислений с помощью программирования графических процессоров. Благодаря таким библиотекам, как Numba и PyOpenCL, теперь мы можем использовать вычислительный потенциал графических процессоров, чья особая внутренняя архитектура позволяет выполнять даже сложные вычисления в параллельном режиме. В частности, эти библиотеки основаны на технологии CUDA, которая специально разработана для графических карт NVIDIA и обеспечивает максимальное использование их возможностей. Затем мы рассмотрели модели программирования, лежащие в основе CUDA и позволяющие вставлять в код на Python ядра – фрагменты кода, скомпилированные для выполнения исключительно на GPU. Эта глава завершает рассмотрение всех моделей параллельного программирования и связанных с ними библиотек. В следующей главе мы проанализируем все возможные области применения технологий и моделей, рассмотренных в предыдущих главах.

Список литературы

- *https://www.tutorialspoint.com/cuda/cuda_introduction_to_the_gpu.htm*.

Глава 7

Эпоха параллельных вычислений

В этой главе мы рассмотрим практические аспекты параллельного программирования, которые были затронуты в предыдущих главах. С появлением параллельных вычислений сфера применения этой технологии значительно расширилась и теперь включает не только само параллельное программирование и связанные с ним аспекты, но также специально разработанные инфраструктуры для управления такими вычислениями. Суперкомпьютеры обычно определяются как высокопроизводительные системы, состоящие из множества центральных и графических процессоров, где применяются методологии параллельных и высокопроизводительных вычислений (HPC). В этой главе представлен обзор секторов и дисциплин, для которых параллельные вычисления стали эффективным инструментом исследований и развития.

Структура

В этой главе мы обсудим следующие темы:

- высокопроизводительные вычисления (HPC);
- параллельные вычисления;
- моделирование;
- параллельная обработка сигналов;
- метод конечных элементов;
- молекулярный докинг;
- геномика;
- игровые движки;
- суперкомпьютеры и HPC.

Высокопроизводительные вычисления (HPC)

Высокопроизводительные вычисления (High-Performance Computing, HPC), безусловно, не являются чем-то новым. Суперкомпьютеры десятилетиями играют важную роль в научных исследованиях.

В последние годы объем данных растет с огромной скоростью, и существует множество областей применения систем HPC, а именно систем, способных выполнять ресурсоемкие вычисления с помощью общих вычислительных ресурсов для получения результатов в более короткие сроки.

Технологии HPC могут применяться на одном компьютере или узле, но их истинная сила заключается в создании кластеров, где множество компьютеров объединены в суперкомпьютер.

Суперкомпьютеры преимущественно используются для моделирования, развития искусственного интеллекта и анализа очень больших объемов данных. Эти три задачи лежат в основе многих приложений, которые нашли свое применение и развиваются в различных научных и промышленных секторах.

Хотя суперкомпьютеры не являются новшеством, но используемые в них технологии за последние 20 лет претерпели огромные изменения. Экспоненциальный прирост вычислительной мощности и скорости передачи данных превращает параллельную архитектуру не просто в хорошую идею, а в необходимость. Новые технологии, такие как большие данные (Big Data) и Интернет вещей (Internet of Things, IoT), генерируют триллионы потоков данных, которые необходимо анализировать синхронно и в режиме реального времени. Совершенно очевидно, что только параллельные вычисления, применяемые на все более мощных устройствах, смогут обеспечить обработку такого объема данных.

Параллельные вычисления

Параллельные вычисления представляют собой более обширную концепцию, которая также включает в себя параллельное программирование. В действительности под этим подразумевается одновременное выполнение одной или нескольких программ на архитектурах с несколькими процессорами, будь то центральные процессоры, вычислительные ядра или графические процессоры.

Так, параллельные вычисления могут осуществляться как в простой системе с одним процессором и несколькими ядрами (например, Intel Xeon W имеет 28 ядер), так и в суперкомпьютерах или кластерах компьютеров с тысячами микропроцессоров. В последнем случае речь идет о массовых параллельных вычислениях.

Мы используем технологии параллельных вычислений даже в повседневной жизни, хотя и не всегда замечаем это. Наши смартфоны, которые мы не выпускаем из рук в течение дня, используют эту технологию. Например, если бы iPhone мог выполнять только одну операцию за раз, все операции выполнялись бы гораздо медленнее. Открытие электронного письма могло бы занимать несколько секунд, а переключение между различными приложениями не было бы таким быстрым. Последние смартфоны с последовательной архитектурой вышли на рынок в 2010 году (iPhone 4), что является доисторическим периодом, если подумать о том, на что способны современные смартфоны. Уже следующая модель, iPhone 5, была оснащена двухъядерным процессором

с тактовой частотой 1,5 ГГц. А в современном iPhone 11 используется 6 ядер. Смартфон Samsung Galaxy Note 10 имеет 8 ядер. Все эти смартфоны являются примерами параллельных вычислений на ладони.

Преимущества параллельных вычислений

Программы запускаются быстрее благодаря более высокой тактовой частоте, и это может быть связано не только с параллельным выполнением, в то время как параллелизм позволяет одновременно выполнять разные части одной и той же программы. Параллельные вычисления имеют преимущество в виде сокращения времени выполнения этих программ или увеличения объема данных, которые могут быть обработаны одновременно. Чем более мощными являются параллельные вычисления, тем более сложные задачи они могут решать. Экономия времени обычно означает экономию денег, и именно по этой причине параллельные вычисления становятся обязательным выбором для многих отраслей промышленности и проектирования. В научном мире основной целью является не столько экономия времени, сколько возможность решать все более сложные задачи. В обоих случаях основным режимом применения параллельных вычислений является моделирование (simulation).

Как показывает практика, параллельные вычисления идеально подходят для моделирования окружающего нас реального мира. В действительности реальный мир не является последовательным, и происходящее вокруг нас не разворачивается по одному событию за другим, а происходит одновременно. Все более сложные математические модели разрабатываются параллельно с целью реконструкции фрагментов реального мира, как макроскопического, такого как звезды или черные дыры, так и микроскопического, такого как молекулы и ферменты. С развитием параллельных вычислений становится возможным использование все более сложных математических моделей с меньшим количеством приближений, что позволяет генерировать все более точные симуляции. Изучение поведения и свойств систем, воссозданных в симуляциях, является основой для инноваций и исследований во многих областях науки и промышленности.

Еще одним примером, демонстрирующим необходимость использования параллельных вычислений, является одновременный мониторинг нескольких сигналов данных. Иногда приходится работать в режиме реального времени с несколькими независимыми потоками данных, которые предоставляют огромные объемы информации, но должны обрабатываться как единое целое. Компьютеры должны не только обрабатывать эти потоки данных, но также извлекать из них в режиме реального времени информацию, которая будет очень полезна для дальнейших действий. Такого рода проблемы можно решить с помощью алгоритмов параллельной обработки сигналов, которые являются расширением классических методов обработки сигналов с использованием параллельных вычислений. Такое применение мы находим в медицине при работе с биологическими сигналами, в финансах для контроля тенденций фондового рынка, в сейсмологии при изучении волновых колебаний, передаваемых в землю, и во многих других областях.

Проекты и примеры параллельных вычислений

Осознав важность параллельной обработки для наиболее распространенных видов анализа и исследований, проводимых во многих отраслях науки и промышленности, давайте вместе рассмотрим некоторые примеры более подробно.

Метеорология

Метеорология – это наука, основанная на использовании чрезвычайно сложных математических моделей, которые учитывают множество постоянно меняющихся климатических и экологических факторов, с целью составления максимально точных прогнозов погоды на ближайшие дни.

Поэтому очевидно, что для выполнения таких вычислений потребуются чрезвычайно мощные вычислительные машины, основанные в основном на параллельных вычислениях.

В этой конкретной области следует упомянуть методику численного прогнозирования погоды (Numeric Weather Prediction, NWP).

Эта методика прогнозирования погоды основана на системе динамических уравнений, описывающих движение жидкостей. Эта система уравнений является очень сложной с математической точки зрения, и без дополнительных действий найти ее решение невозможно. Для того чтобы эти уравнения можно было ввести в компьютер, их необходимо подвергнуть ряду операций по преобразованию в численные методы. Затем производятся параметризации и устанавливаются начальные и граничные условия. К сожалению, эти шаги вводят приближения с упущениями, оценками параметров и другими факторами, которые упрощают вычисления в целях их осуществимости, но в то же время влекут за собой погрешности.

Модель NWP предназначена для моделирования атмосферы определенной географической области с учетом ее временной динамики с целью получения прогнозов климатических тенденций. Занимаемый атмосферой объем географической области делится на сетку трехмерных точек, которые определяют ее разрешение. Достижения в области параллельной вычислительной обработки позволяют увеличить разрешение модели и повысить скорость вычислений для получения результатов. Чем выше разрешение модели, тем выше точность прогнозов. Однако работа с моделями такого уровня становится все более сложной. Несмотря на увеличение разрешения и скорости моделирования, современные алгоритмы начинают достигать своих пределов из-за требований взаимодействия математических методов решения уравнений, которые прогнозируют изменение погоды с течением времени.

В настоящее время эти алгоритмы объединяют вычислительный параллелизм с географическим параллелизмом. Анализируемая географическая область делится на перекрывающиеся области, и каждая область назначается отдельному процессору. Во время расчета прогноза каждый процессор обменивается данными о перекрывающейся области с другими процессорами, участвующими в расчете, чтобы учесть климатические воздействия соседних областей.

Увеличение разрешения и ускорение работы модели NWP привело к возможности анализа небольших географических областей, однако при этом увеличилась площадь пересечения различных процессоров, что привело к увеличению обмена информацией и снижению производительности модели.

Еще одним критическим моментом модели NWP, требующим дальнейшего совершенствования, является параметризованная часть. В метеорологии существует множество физических процессов, которые невозможно предсказать напрямую, см. рис. 7.1. Здесь обозначены такие явления, как: интенсивность солнечного излучения (1), рассеяние взвешенными частицами и молекулами (2), снег (3), дождь (4), образование конденсата (5), глубокая конвекция (6), длинноволновое излучение от облаков (7), растительность (8), испарение (9), отражение/поглощение облаками (10), топография (11), водное покрытие (12), турбулентности (13), отражение/поглощение на поверхности земли (14), длинноволновое излучение от поверхности земли (15).

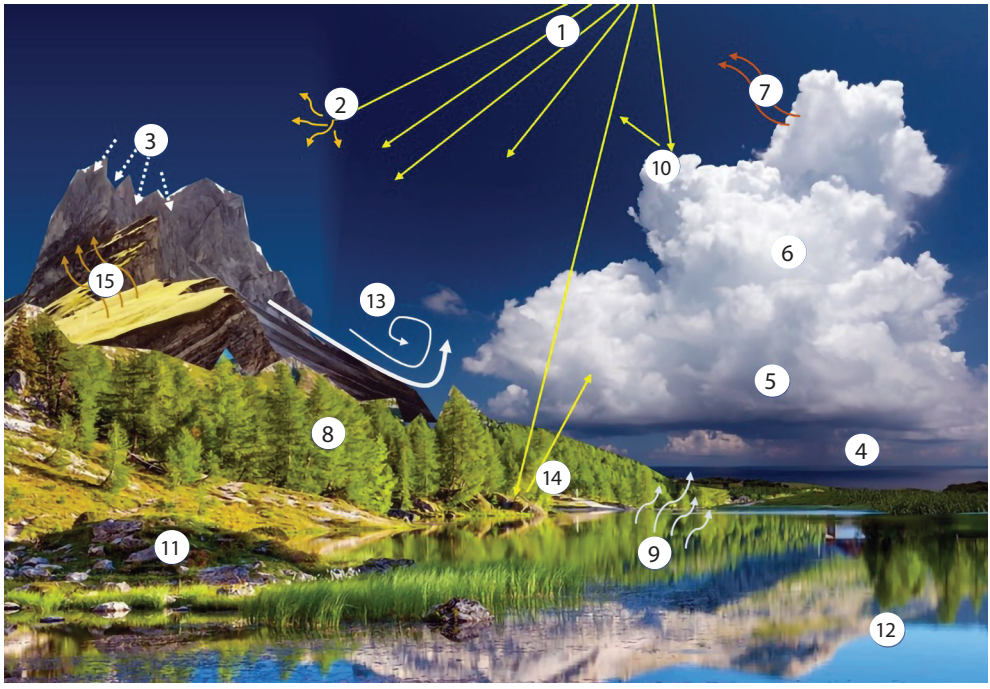


Рис. 7.1. Основные факторы, влияющие на погодные условия

Все это кратковременные, сложные или еще плохо изученные явления, и поэтому они вводятся в модель с помощью соответствующих параметров на основе эмпирических или статистических оценок. Кроме того, в этом случае разработка новых алгоритмов, максимально использующих параллельные вычисления, поможет лучше реагировать на эти дополнительные факторы, повышая эффективность и точность разрабатываемых прогнозов.

Что касается языка Python, здесь можно упомянуть MetView. Эта платформа представляет собой практичную метеорологическую рабочую станцию, пред-

назначенную для обеспечения комплексной среды исследования и анализа метеорологических данных, см. рис. 7.2. Она включает в себя множество инструментов, облегчающих анализ на всех этапах. Доступ к множеству имеющихся источников данных обеспечивает наличие средств конвертации различных форматов. Кроме того, имеется множество инструментов для их обработки и отображения. Все эти операции управляются с помощью команд сценариев на языке Python.

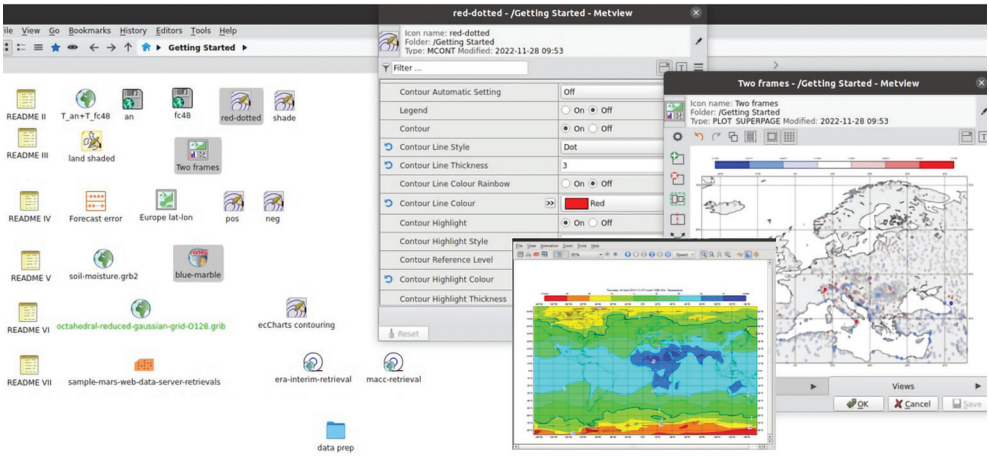


Рис. 7.2. Платформа MetView

Платформа MetView доступна на Anaconda только для Linux и macOS. Для ее установки достаточно ввести следующую команду:

```
$ conda install -c conda-forge metview
```

После установки она обновляет свое содержимое и связанные пакеты с помощью

```
$ conda update -c conda-forge metview
```

После установки платформа запускается с помощью команды metview:

```
$ metview
```

Через несколько секунд приложение начнет работу, и на экране появится окно, как показано на рис. 7.2. Существует много интересных примеров его использования. Они представлены в серии Jupyter Notebooks на странице *https://confluence.ecmwf.int/display/METV/Python+Jupyter+Notebooks*.

Океанография

Океанография – это отрасль естественных наук, где применяются модели, аналогичные метеорологическим. Здесь вместо атмосферы учитываются огромные объемы воды в океанах и все факторы и процессы, которые опи-

сывают их эволюцию во времени. Для получения достоверных результатов моделирования здесь также предполагается использование сложных моделей, которые могут быть обработаны только высокопроизводительными компьютерами. И в этом случае также необходимо использование параллельных вычислений.

На основе этих исследований собираются данные, которые затем распространяются среди исследователей и аналитиков через базы данных. Большая часть этих данных поступает из спутниковых наблюдений, собираемых государственными агентствами, такими как Европейская организация по эксплуатации метеорологических спутников (European Organization for the Exploitation of Meteorological Satellites, EUMETSAT) или Национальный центр экологической информации (National Centers for Environmental Information, NOAA). Эти базы данных предоставляют информацию о наблюдаемых из космоса параметрах, таких как температура поверхности, цвет воды, высота волн и толщина льда. В других базах данных содержатся океанографические данные, касающиеся морского дна и других параметров, относящихся к промежуточным глубинам.

Наиболее часто используемым типом данных в океанографии являются данные CTD (conductivity, temperature, depth – проводимость, температура, глубина). Эти значения позволяют определять физические свойства различных слоев воды в океанах, что чрезвычайно важно для получения профилей TS (temperature-salinity – температура-соленость), широко используемых в океанографии.

Все эти данные используются для моделирования (очень похожего на атмосферное моделирование в метеорологии), которое позволяет прогнозировать состояние океанов с течением времени. И в этом случае используемые модели чрезвычайно сложны и требуют применения суперкомпьютеров, которые максимально используют возможности параллельных вычислений.

Что касается языка Python, существует сайт, на котором собрана вся серия библиотек Python, связанных с различными операционными аспектами в области океанографических исследований – SEA-PY, по адресу *https://pyoceans.github.io/sea-py/*, как показано на рис. 7.3.



Рис. 7.3. Официальный сайт SEA-PY

Сейсмология

Мы рассмотрели модели для изучения атмосферы и модели для изучения больших океанических масс, поэтому не могли обойти вниманием модели для изучения перемещений тектонических плит. В частности, подвижки земной коры и прогнозирование их самых страшных последствий: землетрясений.

Сейсмология также является сложной наукой, где исследования опираются на использование сложных моделей, расчет которых невозможен без использования большого количества собранных данных. Делать прогнозы в этой области еще сложнее, чем в метеорологии и океанографии. Во многом это связано с тем, что природа глубинных слоев земной коры с большим трудом поддается изучению.

В данном случае используется метод сейсмической разведки, основанный на отражении сейсмических волн при их прохождении через земную кору. При прохождении сейсмической волны через земную кору, когда она встречает пограничный слой, разделяющий два разных слоя горных пород, происходит ее частичное отражение. В результате получается серия отраженных волн различной формы и интенсивности, которые необходимо обнаружить, записать в виде данных, а затем экстраполировать из них модель для реконструкции нижележащих горизонтов горных пород.

Расчеты этих моделей также основаны исключительно на параллельных вычислениях. В Python для этого существует специальная библиотека Obs-Py (*https://docs.obspy. org/*). Это библиотека с открытым исходным кодом, которая предоставляет исследователям эффективную среду для обработки сейсмологических данных, предлагая весь набор инструментов для этих специфических задач, таких как обработка сигналов сейсмических волн, как показано на рис. 7.4.

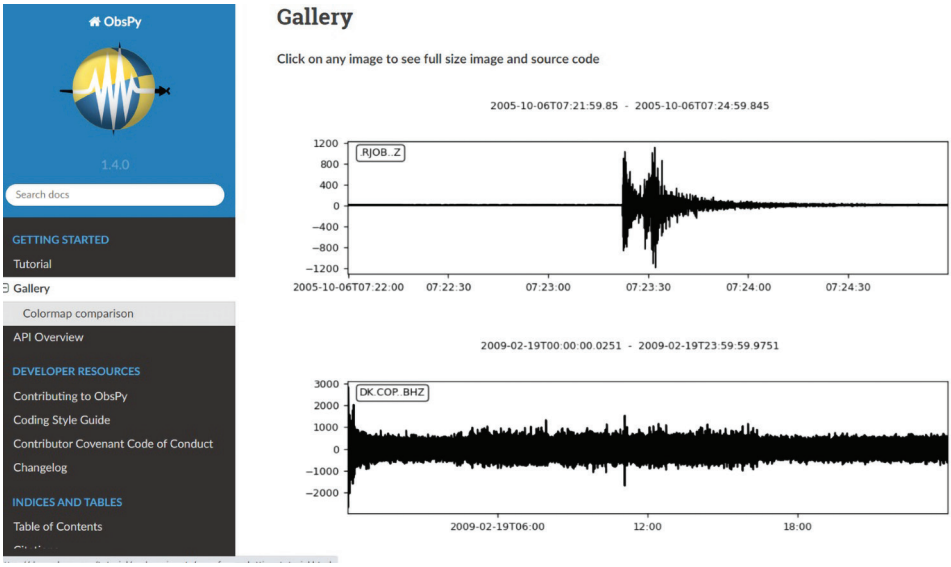


Рис. 7.4. Официальная страница библиотеки Obs-Py

Астрофизика

Еще одной областью науки, в которой последние достижения в основном связаны с моделированием с помощью параллельной обработки, является астрофизика.

В астрофизике предметом изучения являются вопросы поведения и эволюции звезд. В настоящее время в астрофизике широко применяется моделирование с использованием сложных физических законов, учитывающих гидродинамику и магнитную гидродинамику. Система отсчета представляет собой сферическую геометрию, а модель использует спектральные коды, где физические величины подразделяются на основе сферических гармоник в ортогональных направлениях и полиномов Чебышева в радиальном направлении. Разумеется, подобные вычисления являются чрезвычайно сложными и могут быть выполнены только с помощью высокопроизводительных вычислительных систем, таких как массивно-параллельные архитектуры (суперкомпьютеры).

Результаты, полученные с помощью подобного типа моделирования, оказались очень точными и позволили добиться колоссального прогресса в этой области астрофизики.

В Python доступны модели, описывающие поведение и эволюцию звезд, которые используют более современный подход, называемый «звезда в коробке». Эти симуляции основаны на другой модели с использованием декартовой геометрии, в которой звезда заключена в кубическую коробку, как на рис. 7.5. Полученные результаты показывают, что с помощью научных библиотек Python, таких как SciPy, NumPy, Matplotlib и Pypar (еще одна библиотека, облегчающая параллельную реализацию с Python), можно получить довольно точные результаты.

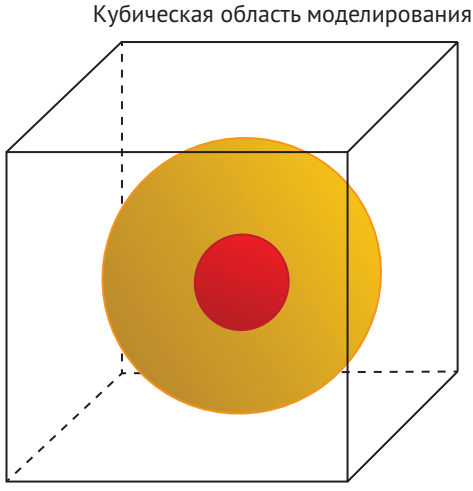


Рис. 7.5. Модель «звезда в коробке»

Помимо изучения поведения звезд, астрофизика в последнее время занимается проблемами, которые до недавнего времени считались неразрешимыми.

Взрыв сверхновой, движение галактик с их вероятным столкновением и слиянием, а также поведение черной дыры при поглощении звезды – все это чрезвычайно сложные явления. Используемые для их моделирования модели настолько трудноразрешимы, что требуют огромных вычислительных мощностей.

Для лучшего понимания взаимосвязи этих научных дисциплин с технологическим прогрессом в области параллельной обработки данных приведем один пример. В 2019 году была решена проблема, которая более сорока лет не давала покоя астрофизикам. Речь о проблеме черных дыр. Исследователи из Северо-Западного университета, университета Амстердама и университета Оксфорда в своих симуляциях показали, что внутренняя материя, которая вращается вокруг черных дыр, а затем коллапсирует в них, совмещается с этими черными дырами. Эта демонстрация стала возможной только благодаря разрешению на использование мощного суперкомпьютера для симуляции. Поэтому очевидно, что с ростом вычислительной мощности будут решены многие другие проблемы, которые до сих пор считались непреодолимыми.

Что касается изучения и моделирования взрывов и конвективных движений звезд, таких как сверхновые и новые, в Python существует ряд алгоритмов, реализованных в различных модулях, как, например, в разработанных Майклом Зингале (Michael Zingale), см. по адресу *https://zingale.github.io/index.html*. На его сайте есть несколько интересных проектов по этой теме, код которых можно скачать с GitHub, см. рис. 7.6.

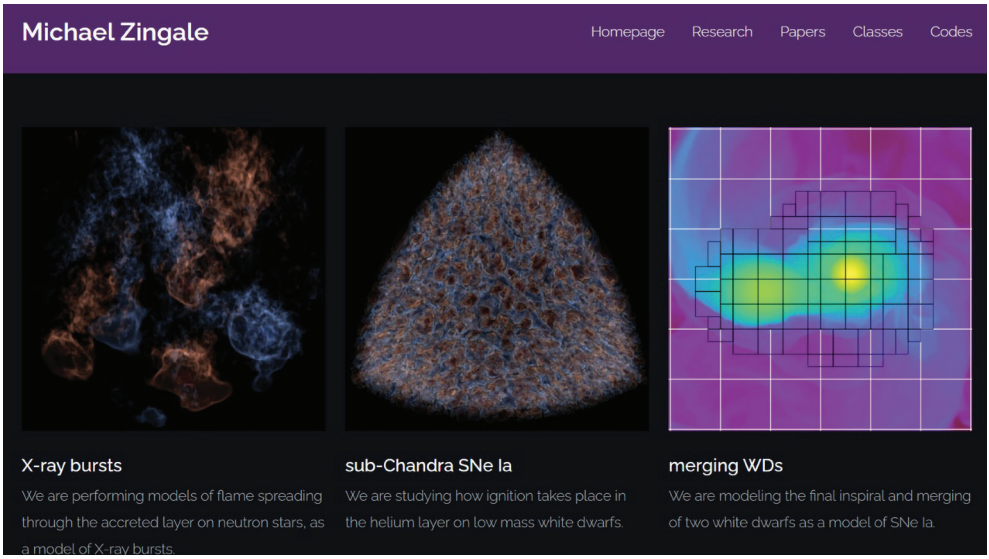


Рис. 7.6. Веб-страница Майкла Зингале

Одним из инструментариев, используемых астрофизическим сообществом, является yt. Он предназначен для анализа и визуализации объемных данных, и в частности его расширение `yt_astro_analysis`. Оно позволяет анализировать и визуализировать данные, полученные в результате моделирования астрофизических явлений на высокопроизводительных вычислительных системах

(HPC). Данные, генерируемые на разных платформах, чаще всего являются несовместимыми, и такой инструмент способен интегрировать их, формировать из них визуальные представления и выполнять точные анализы с помощью параллельных вычислений. Этот пакет также присутствует в Anaconda и может быть установлен с помощью следующей команды:

```
conda install -c conda-forge yt_astro_analysis
```

Существует отличный сайт с обширной документацией и примерами (*https://yt-project.org/doc/cookbook/index.html*), многие из которых представлены в Jupyter Notebooks, как показано на рис. 7.7.

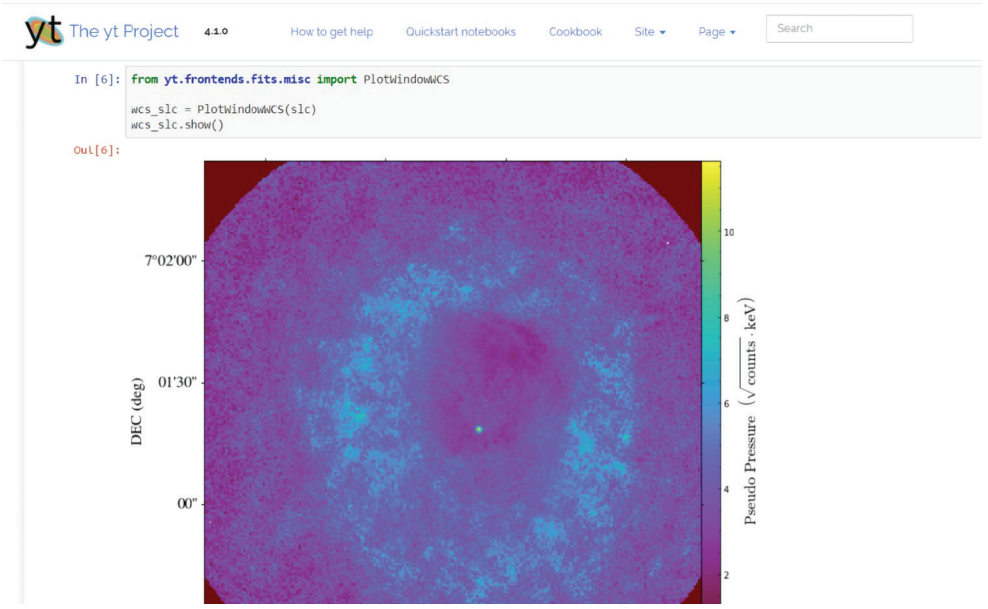


Рис. 7.7. Пример использования yt_astro_analysis в Jupyter Notebook на веб-странице проекта yt

Нефтегазовая промышленность и энергетика

Разведка ресурсов – это метод сбора и анализа информации об энергетических ресурсах, таких как нефть, природный газ и т. д. Компьютеры используются для разведки и оценки таких ресурсов. Нефтегазовые компании используют всевозможные современные методы для увеличения вероятности выявления углеводородных месторождений.

Данные, собираемые с помощью каротажа скважин, представляют собой несколько гигабайтов информации от зондов и датчиков. Эти данные требуют анализа для определения наличия, протяженности и глубины залегания слоев горных пород, нефти, газа и воды. Для обработки такого большого объема данных необходимы системы и методологии HPC. Только с их помощью можно обработать такой объем данных и извлечь необходимую информацию.

Например, в США, в Хьюстоне, находится суперкомпьютер (один из самых быстрых на планете) под названием Bubba, состоящий из тысяч мультипроцессоров Intel Xeon Phi, которые охлаждаются в масляных ваннах. Эта особая технология дает возможность повысить производительность и скорость вычислений такого типа процессоров. Этот суперкомпьютер принадлежит компании DownUnder GeoSolutions, которая специализируется на геопроцессинге, и используется именно для выполнения таких вычислений. На этом примере можно понять, в какой мере все отраслевые исследовательские центры в академических и коммерческих структурах в настоящее время крайне серьезно относятся к роли моделирования и используемых для этого параллельных вычислений.

Что касается Python, то в последние годы в специализированной литературе появилось немало публикаций, в которых приводятся примеры анализа и моделирования с объяснением их преимуществ и недостатков.

Финансы

Наряду с научными и государственными исследовательскими центрами не менее активно системы обработки данных на основе параллельных вычислений используют финансовые учреждения.

Практически все основные аспекты современного банковского дела в настоящее время основаны на операциях, выполняемых на параллельных вычислительных системах. Криптовалюты также нуждаются в мощных вычислительных системах, поскольку, учитывая сложность лежащей в их основе технологии блокчейн, для их работы требуется использование параллельных вычислительных ресурсов.

Одним из наиболее перспективных методов является финансовая математика (quantitative finance), которую также называют количественным анализом (quantitative analysis). Эта дисциплина основана на применении математических и статистических методов в сфере финансов. В ее основе лежит поиск закономерностей или корреляций в огромных массивах маркетинговых данных, которые могут в той или иной степени указывать на тенденции и, следовательно, позволяют применять определенные стратегии. Другими целями могут быть анализ рыночных тенденций, управление рисками и управление инвестициями.

Современные технологии позволяют получить доступ к огромным потокам данных через Big Data и извлекать из них информацию, которая будет использоваться для финансовой математики. По этой причине существует множество алгоритмов и методов, которые, как правило, используют технологии Big Data наряду с технологиями анализа и обработки данных на параллельных вычислительных системах.

В Python также имеется множество примеров, упоминаемых в специализированной литературе и в интернет-источниках, где описываются исследования и применение алгоритмов и методов на основе параллельных вычислений. Во многих из этих приложений используется машинное обучение, для которого в Python предусмотрено множество библиотек, но также имеется множество

специализированных библиотек для финансовой деятельности, имеющих фундаментальное значение для создания и изучения финансовых моделей. Мы рассмотрим некоторые из них, которые могут быть полезны для интересующихся этой темой.

Quandl – это библиотека, которая позволяет получить доступ к обширной подборке экономических, финансовых и рыночных данных из центральных банков, государственных и международных организаций, а также из многих других источников. Большинство наборов данных доступны бесплатно после регистрации, а некоторые предоставляются за плату.

Zipline и Pyfolio – две другие библиотеки, которые позволяют выполнять первичную обработку финансовых данных, например собранных Quandl, который имеет хорошо проработанный интерфейс. Обе библиотеки обеспечивают мощную поддержку специализированных алгоритмов для торговых операций. Кроме того, Pyfolio дает возможность получить отчет со статистикой результативности, включающий такие важные параметры, как годовая/месячная доходность, квантили доходности, скользящие коэффициенты бета/Шарпа и оборачиваемость портфеля, см. рис. 7.8.



Рис. 7.8. Официальный сайт Pyfolio

Кроме того, существуют библиотеки, такие как QuantLib и Technical Analysis Library (TA-Lib), которые изначально создавались на языке C++, но стали доступны на Python. Они предоставляют еще более продвинутые и специализированные инструменты для использования в финансовой сфере. Так, QuantLib позволяет разрабатывать модели для проведения торговых операций и управления рисками. Эта библиотека предлагает набор инструментов для проектирования сложных алгоритмов, включая рыночные конвенции, модели кривой доходности, сольверы, дифференциальные уравнения в частных производных и метод Монте-Карло. В TA-Lib доступны дополнительные финансовые ин-

струменты, которые можно интегрировать с QuantLib, например, для анализа пересечений, индикаторов импульса, индикаторов объема, индикаторов волатильности, преобразования цен, индикаторов циклов, а также для анализа паттернов и функций финансовой статистики.

Все эти библиотеки доступны на платформе Anaconda и могут быть установлены с помощью следующих команд:

```
conda install -c anaconda quandl
conda install -c conda-forge zipline
conda install -c conda-forge quantlib
conda install -c conda-forge pyfolio
conda install -c conda-forge ta-lib
```

Проектирование

Анализ методом конечных элементов (Finite Element Analysis, FEA) – это моделирование любого физического явления с помощью численного метода, называемого методом конечных элементов (Finite Element Method, FEM).

Инженеры используют моделирование FEA для воссоздания поведения виртуальных прототипов в различных условиях без их физического воплощения. Исходя из данных, полученных в результате этих симуляций, становится возможным оптимизировать выбор компонентов, их материалов и формы, не создавая и не тестируя их каждый раз на практике, что значительно экономит время и ресурсы. Однако для этого необходимо использовать математические модели с максимально точным моделированием реальности, способные количественно оценивать результаты воздействия физических и структурных явлений, таких как теплопередача, распространение волн, сопротивление давлению и т. д. В связи с этим большинство таких моделей описываются уравнениями в частных производных (Partial Differential Equations, PDEs).

Метод конечных элементов является наиболее успешной на сегодняшний день технологией для выполнения такого рода расчетов и дает отличные результаты. Для проведения максимально точных и эффективных расчетов по методу конечных элементов необходимо использовать высокопроизводительные компьютеры, работающие по принципу параллельных вычислений. При расчетах по методу конечных элементов одновременно решается огромное количество уравнений в частных производных, которые затем обрабатываются в параллельном режиме.

Для выполнения симуляций необходимо создать сетку, состоящую из миллионов мелких элементов, которые вместе формируют общую структуру.

Расчеты производятся по каждому отдельному элементу. При объединении отдельных результатов получается окончательный результат, описывающий всю структуру. Это означает, что в пространстве между значениями, полученными из точек (узлов), описывающих элементы, вводится приближение интерполяции.

Другими словами, вы знаете значение некоторых точек, но не всех. Интерполяцию можно улучшить, используя все более сложные системы или увеличивая

количество конечных элементов, учитываемых при анализе. Все это требует большей вычислительной мощности, более сложных вычислений и большего объема памяти. Таким образом, для повышения производительности и точности этих симуляций необходимо повысить эффективность вычислительного инструмента, на котором они выполняются.

Достижения в области параллельных вычислений фактически приводят к улучшению и в области FEA, и в результатах, получаемых в результате имитационного моделирования.

Что касается методов FEM в Python, для этих целей существует множество проектов и готовых библиотек.

PolyFem – это библиотека, разработанная на C++, но также представленная на Python, в которой реализованы методы вычисления различных дифференциальных уравнений в частных производных, таких как:

- Лапласа,
- Гельмгольца,
- линейной упругости,
- Сен-Венана–Кирхгофа,
- Нео-Хука,
- Стокса,
- Навье–Стокса.

Здесь заданные для расчета задачи удобно вводить через интерфейс JSON или программно, путем определения класса.

Эта библиотека также доступна на платформе Anaconda. Для ее установки необходимо ввести следующую команду:

```
conda install -c conda-forge polyfempy
```

Альтернативой может служить SfePy – библиотека для решения систем связанных дифференциальных уравнений в частных производных с помощью метода конечных элементов в 1D, 2D и 3D, как показано на рис. 7.9. Для ее установки рекомендуется загрузить исходный код с GitHub и скомпилировать его, однако на платформе Anaconda доступна уже скомпилированная версия:

```
conda install -c conda-forge sfepy
```

Медицина и разработка лекарственных препаратов

Параллельные вычисления широко используются в медицине, особенно при обработке изображений. Диагностические устройства, сканирующие тело и мозг человека, используют для получения снимков самые передовые технологии. Одной из наиболее распространенных технологий в этой области является реконструкция магнитно-резонансной томографии, МРТ (Magnetic Resonance Imaging, MRI), которая используется для обработки рентгеновских снимков и исследования структуры мозга.

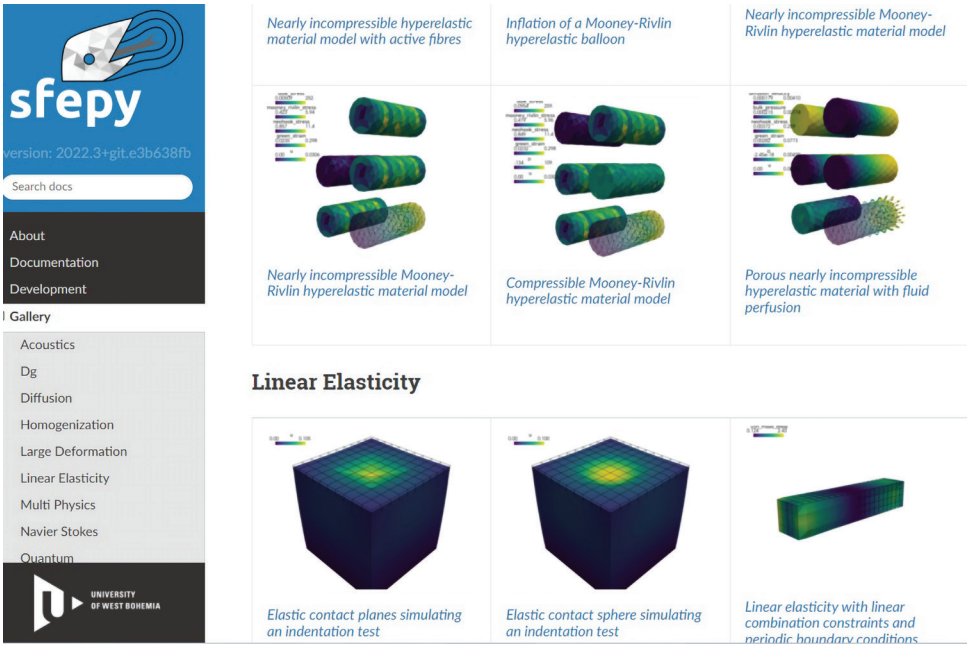


Рис. 7.9. Официальный сайт SfePy с множеством примеров

Python, как язык научного программирования, предлагает в этой области множество инструментов, в частности библиотеки, такие как `mripy`, которые включают в себя ряд вычислительных инструментов для реконструкции МРТ, таких как функции машинного обучения, преобразование Фурье и специализированные нейронные сети. Библиотека Python, которая полностью использует потенциал параллельных вычислений (как CPU, так и GPU), называется `SigPy`. Эта библиотека предлагает набор инструментов для выполнения задач по обработке сигналов, однако она была разработана специально для обработки МРТ-изображений и включает в себя полностью специализированный модуль `sigpy.mri`. Для его установки можно использовать платформу Anaconda:

```
conda install -c frankong sigpy
# (опционально для поддержки графиков) conda install matplotlib
# (опционально для поддержки CUDA) conda install cupy
# (опционально для поддержки MPT) conda install mpi4py
```

Методы параллельной обработки сигналов также находят все более широкое применение в других областях диагностики. Пациенты с нервно-мышечными или кардиореспираторными заболеваниями нуждаются в постоянном мониторинге. Постоянный мониторинг генерирует большой объем данных в виде сигналов с различной полезной информацией, которую необходимо обрабатывать в кратчайшие сроки. Поэтому и в этом случае необходимо использовать алгоритмы обработки сигналов, адаптированные для подобного типа задач. Отличительной особенностью данного случая является то, что эти данные не-

обходимо обрабатывать в режиме реального времени, а значит, за очень короткий промежуток времени. Полученные в результате мониторинга данные состоят из биосигналов различного типа (как правило, электрокардиограмма, дыхание, электроэнцефалограмма и электромиограмма), которые регистрируются параллельно. Их обработка с целью извлечения полезной информации достаточно сложна и требует значительного времени. Несомненно, что классический подход приводит к задержкам и снижению эффективности, что может иметь критические последствия, поскольку речь идет о здоровье человека.

Поэтому переход к параллельной обработке является необходимым шагом для повышения эффективности существующих систем мониторинга и обеспечения возможности получения результатов в режиме реального времени, чтобы при необходимости можно было незамедлительно принять меры. Выявление внезапных изменений в показателях жизненных функций пациента должно быть быстрым и точным, чтобы предотвратить серьезные осложнения.

В Python для такой обработки сигналов разработана библиотека BioSPPy. Это набор инструментов, включающий различные методы обработки сигналов и распознавания образов, которые предназначены для анализа наиболее распространенных биологических сигналов, таких как фотоплетизмография (PPG), электрокардиография (ЭКГ), кожно-гальваническая реакция (EDA), электроэнцефалография (ЭЭГ), электромиография (ЭМГ) и дыхание. К сожалению, эта библиотека отсутствует в дистрибутиве Anaconda, но ее можно установить на PyPI с помощью команды `pip`:

```
pip install biosppy
```

Даже в научно-исследовательском секторе, таком как разработка лекарственных препаратов и геномика, интенсивно используются сложные алгоритмы. Несомненно, преобразование этих алгоритмов из последовательных в параллельные и использование вычислительных систем (суперкомпьютеров), которые эффективно используют возможности такого программирования, переживают бурное развитие и способствуют значительным новаторским изменениям в этом секторе.

Разработка лекарственных препаратов

Разработка лекарственных препаратов – одна из самых дорогостоящих, трудоемких и сложных задач, стоящих перед фармацевтическими компаниями. Идентификация и оптимизация молекул или пептидов осуществляется на основе огромного количества возможных комбинаций химических соединений. Исследуемые молекулы должны отвечать определенным требованиям, таким как высокая аффинность и специфичность по отношению к соответствующей целевой молекуле, как правило, ферменту (метод молекулярного докинга). Этот фермент каким-то образом связан с заболеванием, и его активация-деактивация приводит к тому, что молекула становится эффективным лекарственным средством. Но не только это: молекула также должна обладать благоприятными фармакодинамическими и фармакокинетическими свойствами, иначе она рискует быть не усвоенной или будет слишком быстро метаболизирована та-

ким образом, что не достигнет целевого фермента. Моделирование такого анализа потребляет столько вычислительных ресурсов и времени, что внедрение параллельных вычислений становится крайне необходимым.

В связи с этим фармацевтическая промышленность также оказывает сильное влияние на развитие и инновации в области современных методов анализа и вычислительных систем, на которых осуществляется их обработка.

Разработка новых лекарственных препаратов – это огромный рынок, требующий огромных финансовых вложений. С другой стороны, согласно статистике, только одно из 10 000 соединений, предложенных отделом исследований и разработок, в конечном итоге будет признано успешным лекарственным препаратом. Таким образом, очевидно, что любые меры, направленные на ускорение и облегчение этапов моделирования и анализа потенциального лекарственного препарата, приведут к сокращению расходов.

Молекулярный докинг (molecular docking) – это метод, включающий в себя выборку и оценку. Этот процесс состоит из выборки возможных пространственных конфигураций, которые может принимать молекула, и расчета оценочных функций для каждой из этих конфигураций при связывании с ферментом.

Этот процесс является очень ресурсоемким в плане вычислений, но легко поддается разделению на параллельные задачи. Кроме того, многие алгоритмы, используемые при оценке лекарственного препарата, также могут быть разделены на параллельные задачи, что повышает производительность всего процесса.

Благодаря этим методам, которые применяются на суперкомпьютерах, происходит увеличение скорости вычислений и, следовательно, сокращение времени, необходимого для выделения эффективных лекарственных препаратов.

Однако исследования в этой области продолжаются, что открывает возможности для многих других инноваций.

Существует много программ для молекулярного докинга, в том числе Autodock, разработанная на C/C++. Недавно была выпущена новая версия этой программы под названием Autodock-GPU, которая в значительной степени использует потенциал графических карт. Многие научные работы свидетельствуют о том, что скорость работы этой последней версии в сотни раз превышает скорость работы первоначальной версии.

Что касается языка Python, существует версия Autodock под названием AutodockFR, которую можно загрузить в виде пакета по адресу *https://ccsb.scripps.edu/adfr/downloads/*.

Однако существуют и другие библиотеки для молекулярного докинга. Одной из них является `pyscreener`, которая имеет внутреннюю систему задач, способную эффективно адаптироваться к количеству доступных вычислительных ядер и одновременно масштабировать параллельно выполняемые вычисления между ядрами и доступными процессорами.

Среди коммерческих решений стоит упомянуть библиотеки, предоставляемые **Acellera** (см. рис. 7.10), которые можно использовать и бесплатно (соглашение EULA) с ограничением на использование одного графического процес-

сора. Среди них – HTMD, программируемая среда для моделирования, анализа и визуализации молекулярных систем.

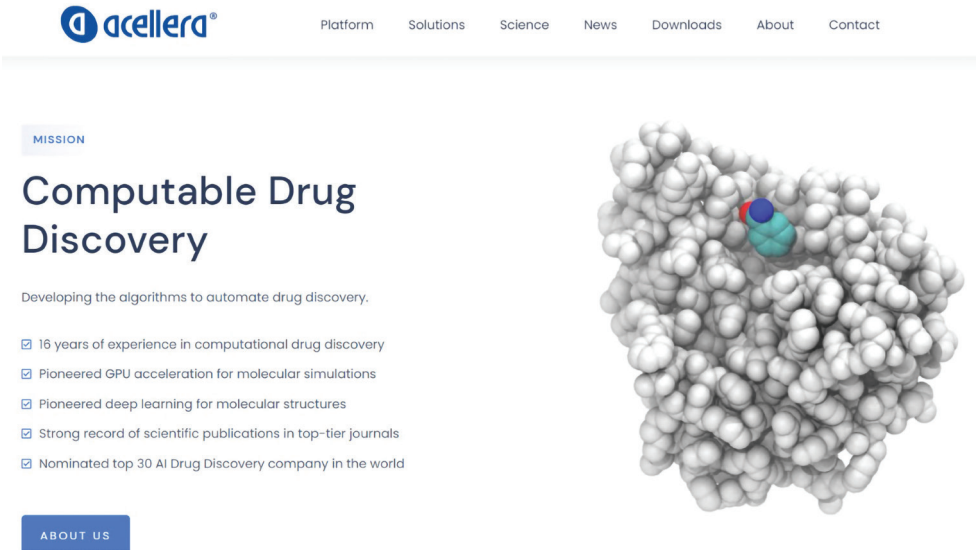


Рис. 7.10. Веб-страница Acellera о разработке лекарств с помощью вычислений

Геномика

В геномике также существует большая потребность в массовом использовании суперкомпьютеров, основанных на параллельных вычислениях. Исследование генома организма требует умения работать с очень большими объемами данных и информацией, что крайне сложно обеспечить без алгоритмов, способных работать в параллельном режиме.

Геномика является наиболее представительной из дисциплин биоинформатики. Она занимается сравнением структур, таких *как последовательности* ДНК, гены и РНК, между различными организмами. При этом речь идет о данных огромных размеров (терабайты), настолько больших, что их можно классифицировать как биологические большие данные (Biological Big Data). При использовании обычных компьютеров и без применения параллельных вычислений на один эксперимент может уйти несколько недель, а то и месяцев. Поэтому для сокращения времени обработки и обеспечения возможности работы с данными такого типа, их обработки и анализа необходимо применять методы параллелизации и высокопроизводительных вычислений (HPC).

Однако интеграция алгоритмов биоинформатики с параллельными вычислениями – задача не из простых, и она все еще находится в стадии изучения и разработки. Существует два подхода к решению этой задачи. Современное программное обеспечение, такое как FASTA, BLAST, HMMER, ClustalW и RaxML, в настоящее время дорабатывается с целью использования параллельных

вычислений, например с использованием интерфейса MPI (см. рис. 7.11). Альтернативным подходом является использование распределенных систем, так называемых облаков. В них для биоинформатических вычислений создаются конвейеры, которые можно представить в виде рабочих процессов.

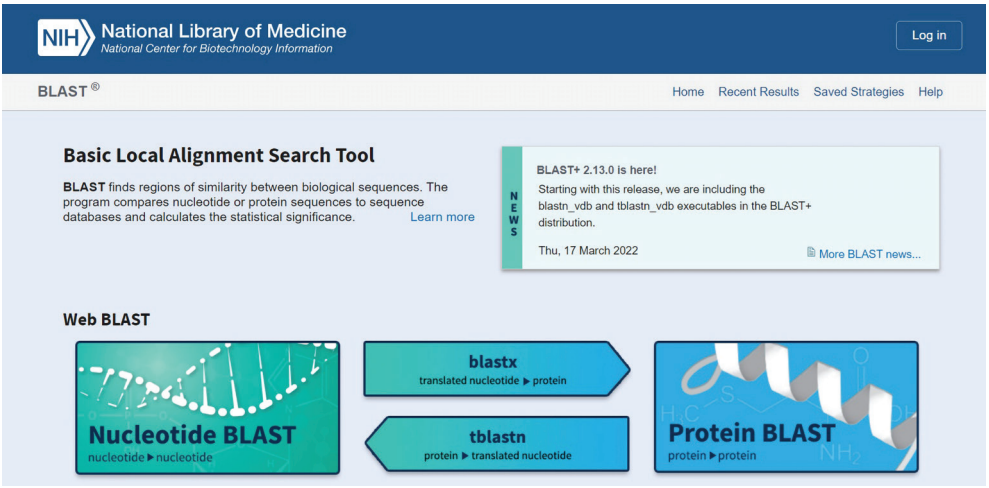


Рис. 7.11. Официальная веб-страница BLAST

В настоящее время уже существуют системы управления рабочими процессами, такие как Tavaxy, Pegasus, Swift/T и SciCumulus, которые могут обрабатывать данные биоинформатики в распределенных инфраструктурах.

Что касается Python, существует библиотека `pyrpipe`, предназначенная для обработки данных RNA-Seq путем создания вычислительных конвейеров полностью на языке Python. Эта библиотека может использоваться как на локальных компьютерах, так и в средах HPC, а также легко интегрируется в системы управления рабочими процессами, такие как Snakemake и Nextflow.

Эту библиотеку можно без проблем установить на платформе Anaconda:

```
conda install -c bioconda pyrpipe
```

Наряду с `pyrpipe` дистрибутив Anaconda включает в себя огромное количество библиотек для биоинформатики, которые собраны в специальном репозитории под названием bioconda по адресу *https://anaconda.org/bioconda/repo*.

Кроме того, в Anaconda есть еще одна библиотека под названием BioPython, которая содержит ряд полезных инструментов для молекулярной биологии. Ее можно загрузить с помощью следующей команды:

```
conda install -c conda-forge biopython
```

Развлечения – игры и фильмы

После такого большого количества примеров использования в различных научных и промышленных отраслях может показаться удивительным, что

параллельные вычисления уходят своими корнями в мир развлечений. На самом деле это не должно вызывать удивления, поскольку графические процессоры изначально были разработаны для обработки больших графических нагрузок в видеоиграх. Именно рынок видеоигр стал мощным стимулом для разработки специальных процессоров, способных улучшить качество игр, в частности трехмерных изображений. Игры постоянно эволюционируют, и в настоящее время их развитие привело к появлению продуктов с плавной графикой, богатыми интерактивными функциями (нередко управляемыми искусственным интеллектом), которые образуют все более реалистичные сценарии (иногда даже выходящие за пределы реальности). Именно здесь и проявляется весь потенциал высокой степени параллелизма, присущей графическим картам.

Игровые движки

В основе большинства игр лежит сложная структура, называемая игровым движком (game engine), который является центральным элементом их рабочего процесса.

На рынке представлено несколько моделей игровых движков для создания видеоигр. Среди наиболее актуальных можно выделить следующие:

- Unreal Engine;
- Unity;
- Godot.

Эти мощные игровые движки просты в использовании и позволяют использовать интеллектуальные функции для создания красивых приложений, обеспечивающих максимально реалистичный игровой процесс (см. рис. 7.12).

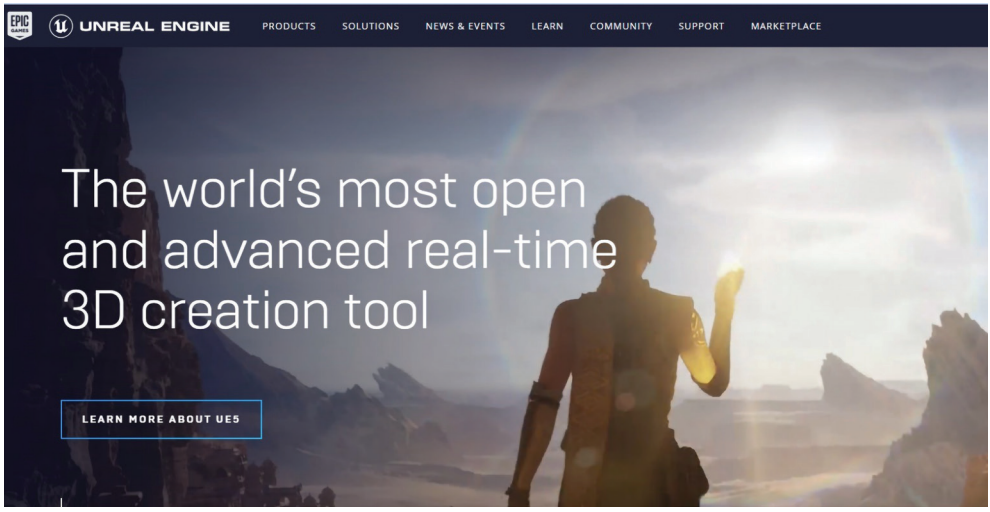


Рис. 7.12. Официальная веб-страница Unreal Engine

Для разработчиков предоставляются пользовательские интерфейсы, которые позволяют разрабатывать игры с помощью простых средств, с максимальным использованием самых мощных доступных технологий, как графических (поддерживаемых графическими процессорами), так и обучающих (искусственный интеллект). Это открывает возможность добиться более реалистичных движений объектов в трехмерной среде и еще большей интерактивности. Эти игровые движки постоянно совершенствуются и включают в себя все возможные инновации в области параллельного программирования, поскольку позволяют максимально использовать все доступные технологии. Нередко эти технологии затем находят повторное применение в других, упомянутых ранее областях.

Мощь игровых движков также основана на 3D-рендеринге. Именно этот процесс позволяет игрокам получить максимально реалистичные впечатления от игры. 3D-рендеринг – это не что иное, как создание изображения на основе трехмерного моделирования.

Трехмерные модели объектов виртуальной среды, созданной игровым движком, проходят этап текстурирования. На поверхность этих объектов наносятся двухмерные изображения, что придает им реалистичный вид, а блики рассчитываются с учетом направления освещения. Добавляются другие эффекты, такие как тени, и в результате создается 2D-изображение, которое, как фотография, преобразует виртуальную реальность в правдоподобное изображение.

Чем более мощные вычислительные устройства будут использоваться, тем лучше будут результаты и тем более реалистичными будут изображения. Кроме того, эти изображения необходимо генерировать для каждого кадра видеоигры, которые, как и кадры фильма, будут плавно сменять друг друга. В связи с этим несложно понять, каких вычислительных мощностей требует такая технология. К этому следует добавить вычисления, необходимые для обработки звука, искусственного интеллекта, сети и памяти.

Параллельные вычисления – это мощный инструмент для оптимизации базовых процессов в игровом движке. Наиболее мощные игровые движки в настоящее время используют ядра центрального процессора и графические процессоры в режиме многопоточности. В этой книге было показано, что Python не очень хорошо обрабатывает многопоточность на центральных процессорах, и на самом деле эти игровые движки основаны на языках программирования, таких как C# и C++, которые, напротив, допускают применение многопоточности на центральных процессорах.

Так, например, в игровом движке Unity (см. рис. 7.13) многопоточность применяется для параллельного выполнения процессов рендеринга в разных потоках. Параллельное выполнение нескольких потоков значительно сокращает время выполнения процессов в рамках одного кадра. Таким образом, можно генерировать больше кадров в секунду, и, следовательно, игры будут работать более плавно. Именно здесь вступают в игру графические процессоры, архитектура которых разработана для более эффективного вычисления текстур пикселей, чем это может сделать центральный процессор. Кроме того, благодаря высокой степени параллелизма внутренней структуры эти вычисления

могут быть распределены между большим количеством потоков одновременного выполнения, что значительно облегчает нагрузку и сокращает время обработки.

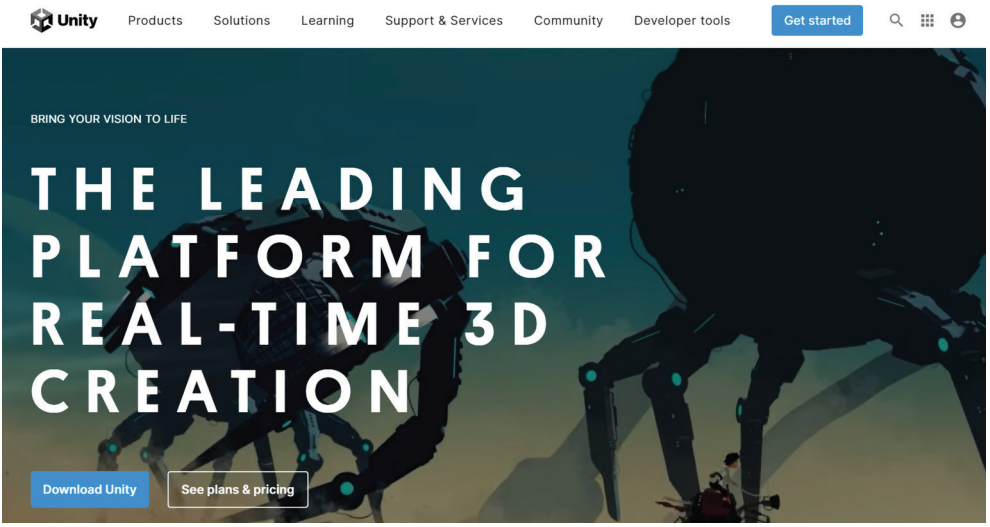


Рис. 7.13. Официальная веб-страница Unity

В то же время центральный процессор может выполнять другие задачи. Поскольку процессоры обладают высокой мощностью при выполнении математических вычислений, их используют для параллельного выполнения таких вычислений, чтобы сделать 3D-среду еще более реалистичной. Поведение объектов, эффекты ударов и колебаний водной поверхности, реакции на взаимодействие с пользователем и даже интеллект неигровых персонажей (Non-Player Characters, NPC) обрабатываются специальными алгоритмами для процессора, ядра которого распределяют между множеством потоков для параллельных вычислений.

Как уже было сказано ранее, для Python таких игровых движков не существует. Доступные движки распространяются в виде библиотек. PyGame и другие библиотеки для разработки игр на Python не поддерживают многопоточность на процессорах и не достигли таких уровней производительности, как библиотеки на других языках программирования, например Unreal Engine и Unity. Однако ничто не мешает новым проектам достичь достойных результатов в будущем.

Разработка параллельного игрового движка

На сайте Intel по адресу *https://www.intel.com/content/dam/develop/external/us/en/documents/designing-a-parallel-game-engine-155986.pdf* есть замечательная статья под названием *Designing the Framework of a Parallel Game Engine* («Проектирование структуры параллельного игрового движка»). В ней описывается процесс разработки одного из возможных игровых движков.

Это может стать хорошим стимулом для разработки нового игрового движка на Python.

Хороший дизайн игрового движка заключается в максимальном использовании всех доступных процессоров. Как уже было сказано ранее, другие игровые движки используют преимущества многопоточности на центральных процессорах. В Python у нас нет такой возможности, но мы можем использовать многопроцессорную обработку. Что касается графических процессоров, то, как уже было сказано ранее, Python в этом плане не имеет серьезных ограничений по сравнению с другими языками программирования.

Как мы уже выяснили, игровой движок работает в цикле обработки, направленном на создание одного кадра за другим. Каждый кадр является результатом целой серии процессов, включающих движение объектов в трехмерном пространстве, текстурирование, рендеринг и взаимодействие с пользователем. Совершенно очевидно, что генерация кадра приводит к последовательному выполнению игры, где кадры генерируются в упорядоченной последовательности один за другим. Но между генерацией одного кадра и следующего можно максимально использовать весь потенциал параллельных вычислений. Между двумя кадрами будут запускаться потоки и процессы, которые будут работать в параллельном режиме для выполнения всех необходимых задач, что значительно сократит время обработки. Затем генерация кадра станет моментом синхронизации, когда все потоки или процессы должны будут завершить свое выполнение и выдать результат. Обмен информацией между ними будет происходить именно в этот момент, что значительно снизит нагрузку при синхронизации и позволит системе быстрее переходить от одного кадра к другому.

В статье описаны два режима работы.

Первый: режим Lock Step Mode (пошаговая синхронизация) – именно этот режим был описан выше, см рис. 7.14.

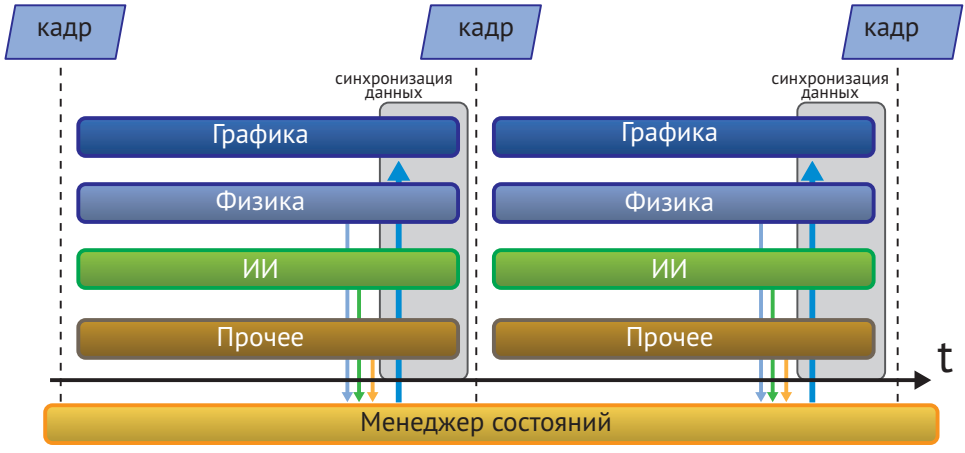


Рис. 7.14. Режим Lock Step Mode

Второй: режим Free Step Mode (свободная синхронизация), где не все компоненты обязательно должны синхронизироваться на каждом этапе кадра, но могут быть распределены на несколько циклов (см. рис. 7.15).

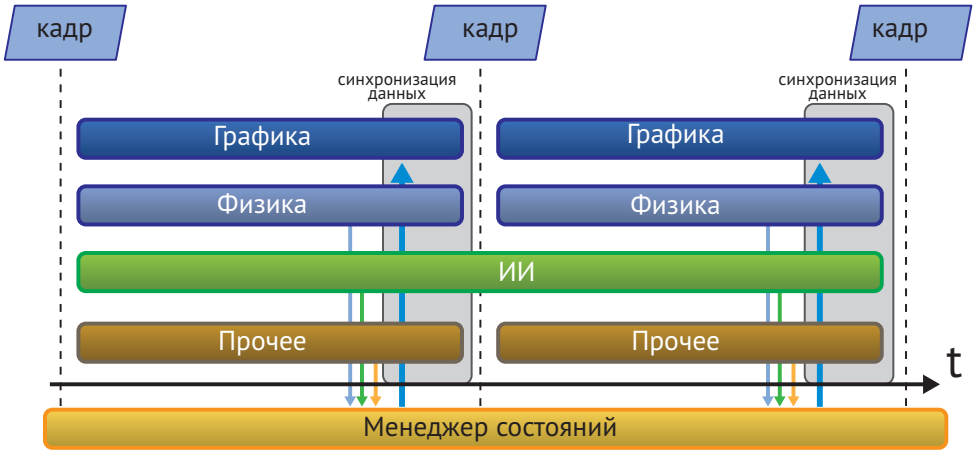


Рис. 7.15. Режим Free Step Mode

Фильмы и 3D-анимация

Возможности игровых движков по созданию трехмерных сред привели к результатам, которые порой превосходят саму реальность. Некоторые технологии, используемые при создании изображений для видеоигр, также применяются при производстве фильмов. На этапе постпродакшна многих современных фильмов все чаще используются эффекты (очень часто заимствованные из видеоигр), которые позволяют улучшить реальные изображения или создать новые сюрреалистические или дорогостоящие декорации, в которых движутся актеры.

Некоторые технологические приемы, такие как цветовая коррекция и трассировка лучей, используют всю мощь графических процессоров для улучшения изображений, снятых для фильма. Все эти визуальные эффекты обычно называют VFX (Visual Effects – визуальные эффекты). Они являются результатом обработки фотореалистичных изображений, сгенерированных цифровым способом, или результатом обработки других изображений.

Специалисты по спецэффектам для этой работы используют особенное программное обеспечение. Например, DaVinci Resolve Studio – это платформа для обработки спецэффектов, которая активно применяет графические процессоры (см. рис. 7.16). В таких фильмах, как «Тор: Любовь и гром» (2022) и «Скоростной поезд» (2022), это программное обеспечение использовалось для цветокоррекции изображений, снятых с помощью высококачественного рендеринга на основе методов трассировки лучей, чтобы подчеркнуть отражения, контрасты и световые эффекты. Вся эта работа выполнялась с помощью параллельных вычислений на графических процессорах.

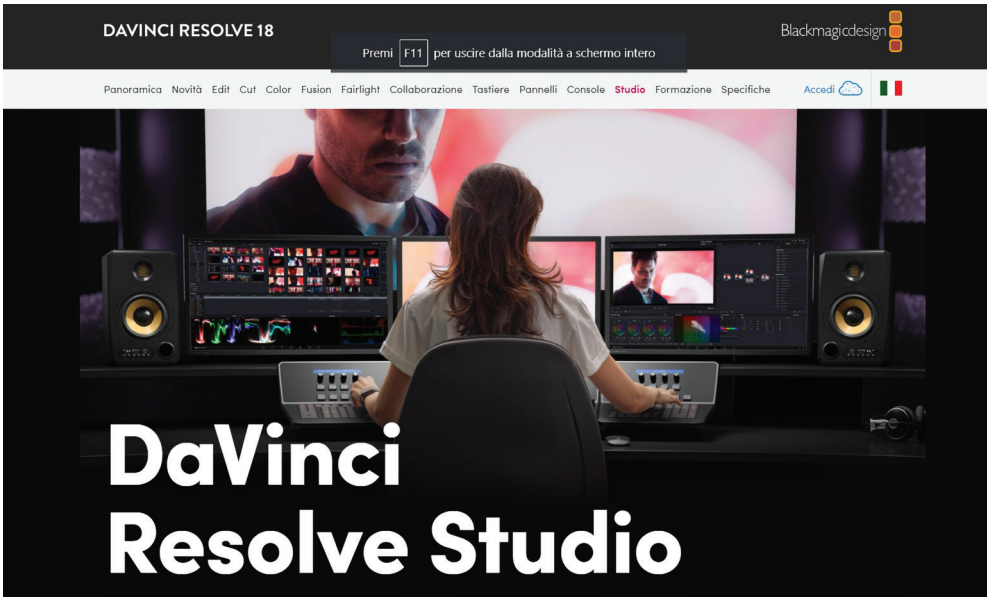


Рис. 7.16. Веб-страница DaVinci Resolve Studio

Трассировка лучей (Ray Tracing) – это разновидность алгоритма рендеринга, который создает реалистичные световые эффекты в компьютерной графике посредством серии векторных вычислений. Эта техника требует огромного количества ресурсов (очень затратно в плане вычислений). Если представить себе затраты на обработку отдельных фрагментов изображений в фильме, то можно себе представить, сколько ресурсов требуется для анимационных фильмов, таких как фильмы Pixar. Длительность этих фильмов составляет около 2 ч, и все изображения в них являются результатом рендеринга сред, полностью построенных на искусственных трехмерных моделях.

Питер Коллингридж (Peter Collingridge), фриланс-программист Pixar Animation Studios, поведал, что Pixar для создания своих анимационных фильмов имеет один из 25 самых мощных в мире суперкомпьютеров, который состоит из 2000 машин и 24 000 вычислительных ядер. Несмотря на наличие такой вычислительной мощности, для рендеринга таких фильмов, как «Корпорация монстров» (2001), потребовалось около двух лет вычислений.

Несомненно, в последние годы Pixar еще больше увеличила мощность своего суперкомпьютера, но в то же время понятно, что для реализации этих фильмов, полностью основанных на 3D-анимации с чрезвычайно реалистичными эффектами рендеринга, требуются огромные вычислительные мощности, даже для лучших в мире суперкомпьютеров.

Начиная с «Корпорации монстров», Pixar и другие студии производят все более динамичные и реалистичные анимационные фильмы, в которых повышается детализация и сложность движений.

Это явный признак того, что в основополагающих технологиях произошло значительное развитие, в том числе в области параллельных вычислений. Учи-

тывая особенности этих технологий, общественности доступно не так много информации, но для тех, кто заинтересован в работе в этом секторе, однозначно следует принять во внимание параллельное программирование.

Заключение

Этой главой мы заканчиваем обзор направлений и областей применения, в которых параллельные вычисления являются эффективным инструментом для внедрения инноваций и проведения исследований. Мы не рассматривали анализ данных и искусственный интеллект, которые будут подробно описаны в следующих главах этой книги.

Список литературы

- *https://www.intel.in/content/www/in/en/high-performance- computing/what-is-hpc.html*.
- *https://www.hp.com/us-en/shop/tech-takes/parallel-computing-and-its-modern-uses*.
- *https://builtin.com/hardware/parallel-processing-example*.
- *https://www.eoas.ubc.ca/books/Practical_Meteorology/prmet102/Ch20-nwp-v102.pdf*.
- *https://pyoceans.github.io/sea-py/*.
- *https://biostats.w.uib.no/ctd/*.
- *https://library.lanl.gov/cgi-bin/getfile?00285657.pdf*.
- *https://events.ecmwf.int/event/279/contributions/2700/attachments/1869/3361/2022-05-17-nwp-primer-metview-intro.pdf*.
- *https://confluence.ecmwf.int/display/METV/Metview*.
- *https://prappleizer.github.io/textbook.pdf*.
- *https://www.lupm.in2p3.fr/users/jmorin/pres/euroscipy2011-jmorin-pencil.pdf*.
- *https://arxiv.org/pdf/1807.06128.pdf*.
- *https://yt-astro-analysis.readthedocs.io/en/latest/*.
- *https://github.com/yt-project*.
- *https://www.researchgate.net/publication/346249862_DEVELOPING_PAR ALLEL_COMPUTING_ALGORITHMS_USING_GPU'S_TO_DETERMINE_OIL_AND_GAS_RESERVES_PRESENTED_IN_THE_ UPSTREAM_EXPLORATION_SECTOR*.
- *https://iopscience.iop.org/article/10.1088/1742-6596/2032/1/012038/pdf*.
- *https://abhyankar-ameya.medium.com/parallel-computing-for-finance-ec053d8fb20f*.
- *https://www.activestate.com/blog/top-10-python-packages-for-finance-and-financial-modeling/*.
- *https://www.simscale.com/docs/simwiki/fea-finite-element-analysis/what-is-fea-finite-element-analysis/*.

- *https://getfem-examples.readthedocs.io/en/latest/demo_unit_disk.html*.
- *https://sfepy.org/doc-devel/index.html*.
- *https://www.unrealengine.com/en-US*.
- *https://unity.com/*.
- *https://godotengine.org/*.
- *https://www.intel.com/content/dam/develop/external/us/en/documents/designing-a-parallel-game-engine-155986.pdf*.
- *https://www.gamedeveloper.com/blogs/advantages-of-parallel-computing-in-unity*.
- *https://www.dcp.dev/projects/distributed-ray-tracing*.
- *https://sigpy.readthedocs.io/en/latest/mri.html*.
- *https://run.unl.pt/bitstream/10362/8249/1/Chorao_2012.pdf*.
- *https://www.mdpi.com/1424-8247/15/1/63/htm*.
- *https://ccsb.scripps.edu/adfr/*.
- *https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4655901/*.

Глава 8

Масштабирование приложений для обработки данных с помощью Dask

В предыдущей главе мы рассмотрели основные возможности параллельных вычислений в различных научных и промышленных отраслях, в том числе в контексте Python, и предложили несколько доступных инструментов и библиотек для этого языка. Однако мы сознательно не затронули тему обработки данных, решив посвятить ей отдельную главу. В этой главе мы в основном рассмотрим возможные области применения анализа данных с помощью параллельных вычислений, а именно библиотеку Dask. Это мощный инструмент, который позволяет расширить возможности широко используемых библиотек для анализа данных, таких как Pandas и NumPy, в распределенных системах чрезвычайно простым и быстрым способом.

Структура

В этой главе мы обсудим следующие темы:

- библиотека Dask;
- Pandas и Dask DataFrame;
- распределенные параллельные вычисления;
- коллекции Dask;
- графы задач и вычисления;
- Kaggle;
- кластер Saturn.io.

Аналитика данных, библиотека Pandas и параллельные вычисления

Python зарекомендовал себя как язык программирования, который отлично подходит для анализа и обработки данных, и постепенно становится конку-

рентом таких традиционных для этой сферы языков, как R. В основном это связано с потенциалом библиотек, доступных сегодня для Python, которые в последние годы завоевали заслуженное место среди инструментов для анализа данных. Одной из таких библиотек является, например, Pandas, которая благодаря DataFrame (см. рис. 8.1) позволяет достаточно эффективно и быстро выполнять операции с огромными объемами данных и управлять ими. Кроме того, эта библиотека, наряду с такими, как NumPy, Scikit-Learn, Matplotlib и многими другими, сформировала комплекс основных инструментов для научной деятельности в области данных, обеспечив создание среды для исследований и анализа, которая исчерпывающе покрывает все потребности в рамках языка Python.

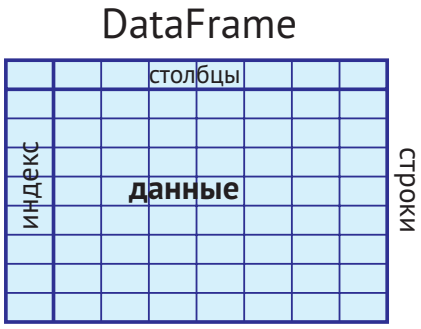


Рис. 8.1. DataFrame

Специалист по работе с данными последовательно устанавливает все эти библиотеки, например ряд специфических виртуальных сред, вроде тех, что мы создали с помощью дистрибутива Anaconda, и таким образом оперативно формирует необходимую среду для разработки и анализа. Вы импортируете данные (зачастую CSV-файлы из интернета) на свой компьютер, с помощью библиотеки Pandas преобразуете их в объекты DataFrame, очищаете и обрабатываете в соответствии с вашими задачами, проводите анализ и извлекаете полезную информацию. Все кажется идеальным. Pandas – это очень эффективная библиотека, способная обрабатывать сотни тысяч строк данных за несколько секунд или даже быстрее, в зависимости от вычислительной мощности вашего компьютера. Но каким бы мощным и эффективным ни был ваш компьютер, рано или поздно вам понадобится импортировать CSV-файлы или данные в других форматах, размер которых превышает объем оперативной памяти вашей системы. При попытке импортировать такие данные вы получите сообщение об ошибке. Кроме того, с увеличением размера активных данных DataFrame и по мере использования все более сложных процедур для обработки и преобразования этих данных время выполнения существенно увеличивается, отчасти из-за используемых алгоритмов, но также и из-за компьютерных ресурсов, которые постепенно иссякают по мере увеличения объема обрабатываемых данных. Это особенно актуально при работе с Big Data.

Что же делать дальше? Добравшись до критической точки, ситуацию можно улучшить за счет горизонтального масштабирования проблемы. В случае с

анализом данных лучшим решением будет не только распараллеливание вычислений с помощью параллельных вычислений, но и распределение обрабатываемых данных по разным вычислительным системам с собственной памятью. Среди уже рассмотренных в этой книге систем распределенные системы лучше всего соответствуют таким требованиям. Системы из нескольких вычислительных установок, каждая со своей памятью, такие как компьютерные кластеры, являются лучшей средой для исследователя данных, который намерен расширить возможности своего ограниченного, хотя и мощного, отдельного компьютера, как показано на рис. 8.2.

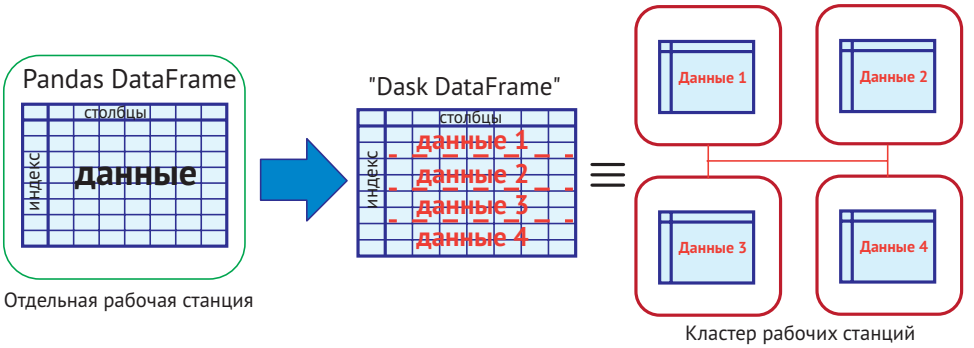


Рис. 8.2. Горизонтальное масштабирование DataFrame

Pandas – это библиотека, разработанная для оптимальной работы на отдельных машинах, но для перехода на распределенную систему, такую как кластер компьютеров, была разработана соответствующая библиотека с теми же характеристиками, что и Pandas, под названием Dask.

Библиотека Dask

Dask – это гибкая библиотека с поддержкой множества библиотек для работы с данными, таких как Pandas, NumPy, Scikit-learn и многих других, которая позволяет выполнять распределенные вычисления и параллельную обработку данных с использованием Python. Эти библиотеки являются незаменимыми инструментами для всех, кто занимается анализом данных на Python. Библиотека Dask использует тот же синтаксис, методы и режимы работы, что и перечисленные выше библиотеки. Таким образом, все пользователи этих библиотек смогут использовать свой уже написанный код, расширив его с помощью библиотеки Dask с помощью простых изменений. Благодаря этому масштабирование кода с одного компьютера на распределенные системы происходит действительно очень быстро.

Итак, Dask – это настоящий фреймворк для создания распределенных приложений, набор инструментов, позволяющий масштабировать систему всего за несколько шагов, переходя от чисто последовательной системы к многопроцессорной системе с параллельной обработкой данных, или осуществлять

горизонтальное масштабирование, а именно переносить выполнение задач на внешние распределенные системы, такие как облако.

Библиотека Dask концептуально разделена на две отдельные части:

- динамическое планирование задач;
- сбор больших данных.

Динамическое планирование задач (dynamic task scheduling) – это компонент для оптимизации вычислений в распределенной системе, в которой они должны выполняться. Преимущественно он занимается полностью прозрачным для разработчика преобразованием кода Python, в котором содержатся функции, методы и определения объектов, во многом похожих на используемые в библиотеках типа Pandas. Результатом преобразования становятся операции для параллельных вычислений, которые полностью соответствуют исходным, но гораздо более производительны и эффективны.

В свою очередь, коллекции больших данных (Big Data collections) – это все те серии объектов с данными, очень похожими на массивы NumPy и DataFrame, которые вместо этого предназначены для распределения по кластерам. Все это происходит при сохранении интерфейсов исходных структур данных, а значит, и названий методов, передаваемых параметров и синтаксиса операций, которые обычно используются с этими объектами.

Таким образом, библиотека Dask предоставляет специалистам по обработке данных возможность расширения до распределенных систем со структурами высокого уровня параллелизма DataFrame и массивами. При этом максимально сохраняется синтаксис исходных библиотек, что позволяет очень просто и быстро переносить существующий код и адаптировать его для параллельных вычислений. Кроме того, после интеграции кода с библиотекой Dask этот код сможет работать на кластерах с более чем 1000 ядрами, как на одном ноутбуке с одним процессом. В этом случае Dask возьмет на себя оптимизацию выполнения кода в соответствии с условиями обработки.

Проще говоря, работу Dask можно обобщить в представлении, показанном на рис. 8.3.

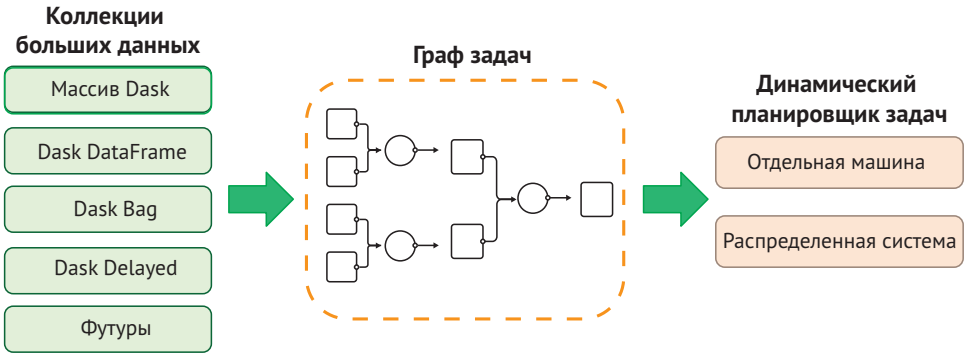


Рис. 8.3. Схема работы библиотеки Dask

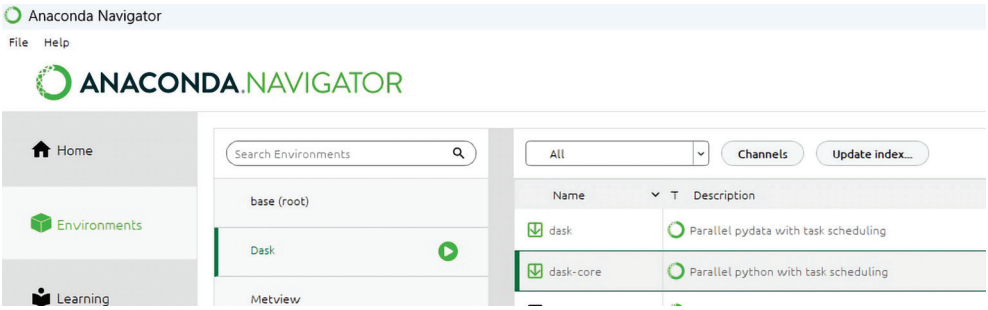
Библиотека Dask предоставляет три различных компонента:

- *коллекции больших данных* (Big Data collections): на высоком уровне эта структура данных очень похожа на соответствующие объекты других библиотек, такие как DataFrame в Pandas. На низком уровне они предназначены для управления распределенными параллельными вычислениями;
- *графы задач* (task graphs): они генерируются Dask из коллекций больших данных и из операций, которые должны быть выполнены над ними. Их цель – разделить эти операции на более мелкие задачи. Затем они будут распределены между различными процессорами системы, рабочими процессами, которые будут выполнять эти задачи в параллельном режиме;
- *динамический планировщик задач* (dynamic task scheduler): этот компонент отвечает за распределение задач между доступными рабочими процессами.

Начало работы на одном компьютере

Лучший способ начать знакомство с Dask и его функциями – использовать свой собственный компьютер. В дальнейшем всегда можно будет перенести разработанный код на распределенные системы, такие как кластер (за определенную плату).

Библиотека Dask корректно работает на всех операционных системах, поэтому ее можно использовать как с Linux, так и с macOS и Windows. Опять же, рекомендуется использовать Anaconda Navigator. Мы создаем новую виртуальную среду для работы над нашими проектами с Dask с помощью панели **Environments** (Среды) в Anaconda Navigator, а затем устанавливаем два пакета `dask` и `dask-core`, как показано на рис. 8.4.



Рис/ 8.4. Установка пакетов Dask в новой виртуальной среде Anaconda Navigator

Поскольку мы работаем с аналитикой данных, нам понадобятся две основные библиотеки Python: NumPy и Pandas. Эти две библиотеки необходимо установить в виртуальной среде, поэтому мы воспользуемся следующими командами из командной оболочки:

```
conda install numpy
conda install pandas
```

Либо всегда можно установить их через Anaconda Navigator, как это было сделано для Dask, см. рис. 8.5.

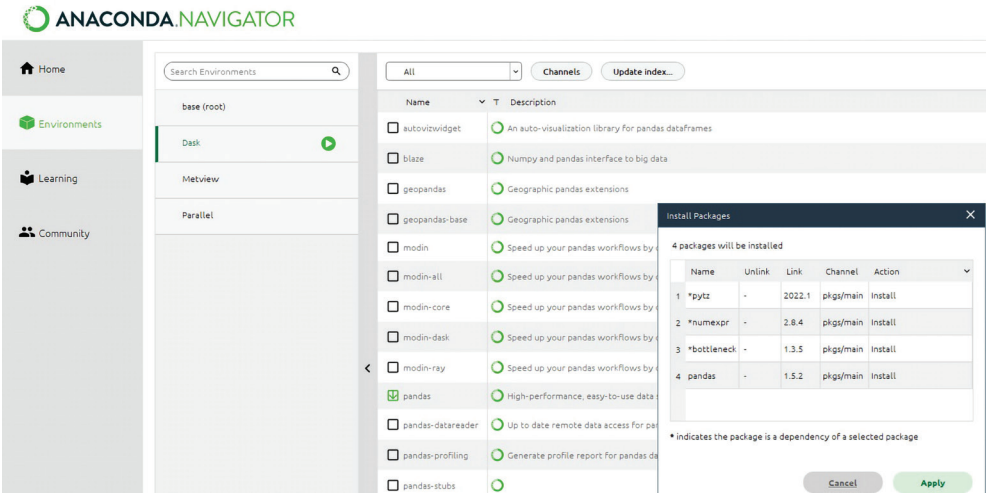


Рис. 8.5. Установка пакетов numpy и pandas в Anaconda

Еще одна библиотека, которая позже пригодится для отображения потока выполнения и моделей распределения памяти через Dask, называется `graphviz`. Эту библиотеку также нужно установить с помощью командной оболочки виртуальной среды, выполнив следующую команду:

```
conda install graphviz
conda install python-graphviz
```

Или графически, выбрав его из пакетов, доступных в Anaconda Navigator.

После установки для проверки ее корректности мы импортируем все необходимые пакеты в наш код следующим образом:

```
import numpy as np
import pandas as pd

import dask.dataframe as dd
import dask.array as da
import dask.bag as db
```

Для запуска программ, представленных в этой главе, можно использовать командную оболочку, как и раньше, но для анализа данных мы будем использовать другой подход: нам понадобится известный блокнот Jupyter Notebook, с помощью которого можно работать с кодом Python в интерактивном режиме, а результаты отображаются в режиме реального времени как в текстовом, так и в графическом формате. Вы обнаружите, что для анализа данных использова-

ние Jupyter Notebook окажется лучшим выбором, облегчающим задачу разработчика и аналитика по тестированию кода, но прежде всего для одновременного изучения анализируемых данных.

Чтобы активировать Jupyter Notebook, вновь из Anaconda на главном экране нажмите кнопку **Launch** (Запуск) непосредственно над квадрантом, соответствующим приложению Jupyter, как показано на рис. 8.6.

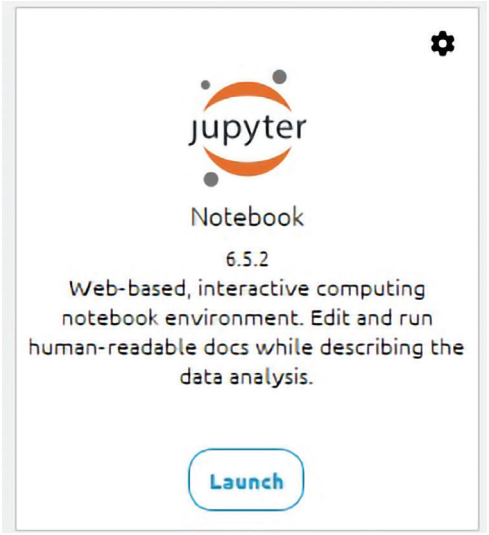


Рис. 8.6. Панель Jupyter Notebook для запуска Jupyter Notebook

После нажатия кнопки через несколько секунд приложение автоматически откроется на странице в браузере по умолчанию. Здесь будут представлены папки, в которых можно открывать, создавать и сохранять страницы Notebook, как показано на рис. 8.7.

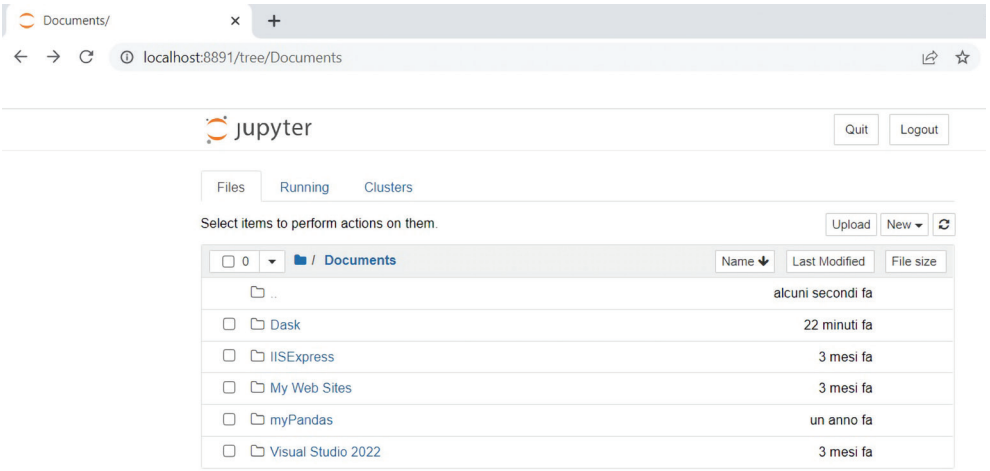


Рис. 8.7. Jupyter Notebook

Итак, давайте создадим новую заметку и запишем в первой ячейке модули, соответствующие импортируемым библиотекам. Если при запуске первой ячейки не появится никаких сообщений об ошибках, значит установка прошла успешно, см. рис. 8.8.

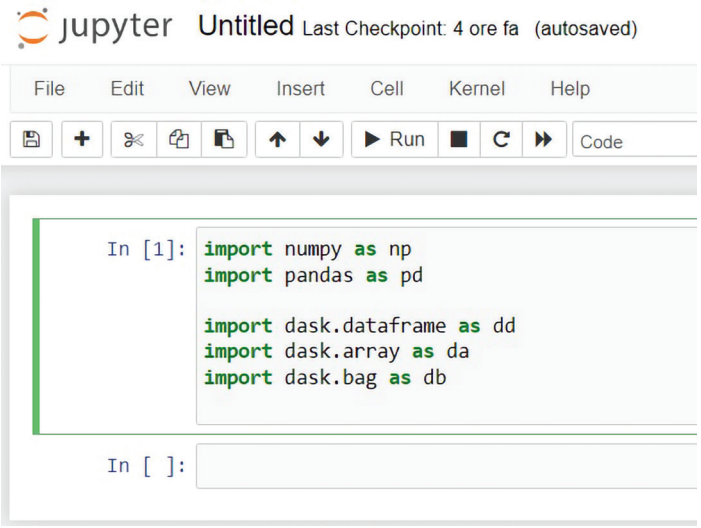


Рис. 8.8. Импорт модулей, необходимых для работы с Dask

Коллекции Dask

Одной из специфических характеристик Dask является возможность расширения структур данных, таких как DataFrame из библиотеки `pandas` и массивы NumPy, для эффективной работы в распределенных системах. Для этого Dask предлагает пользователю набор развертываемых объектов, называемых коллекциями, среди которых наиболее актуальными являются:

- Dask Array;
- Dask DataFrame.

Dask DataFrame представляет собой множество небольших структур DataFrame из pandas, которые находятся в вашем кластере. При взаимодействии с Dask DataFrame вызываемые вами функции и методы преобразуются во множество более мелких функций, которые выполняются в параллельном режиме на нескольких меньших структурах Pandas DataFrame.

Коллекция Dask Array представлена небольшими массивами NumPy, которые находятся в вашем кластере. При взаимодействии с коллекцией Dask Array вызываемые вами функции и методы преобразуются в более мелкие функции, которые выполняются в параллельном режиме на небольших массивах NumPy.

Теперь рассмотрим эти концепции более подробно на практике. Сначала создадим новый Python 3 Notebook в Jupyter и в первую ячейку импортируем необходимые библиотеки и модули:

```
import numpy as np
import pandas as pd

import dask.dataframe as dd
import dask.array as da
```

Запустим ячейку, а затем перейдем к вставке кода в следующую. Сначала попробуем создать простой DataFrame с помощью pandas, который мы будем использовать для сравнения с DataFrame Dask:

```
df = pd.DataFrame(np.random.rand(60).reshape(10,6))
df
```

Запустив ячейку, мы получим результат, подобный показанному на рис. 8.9 (значения меняются, поскольку они случайны).

	0	1	2	3	4	5
0	0,149034	0,604988	0,070560	0,298244	0,455134	0,896560
1	0,854942	0,291555	0,686519	0,856187	0,559002	0,737303
2	0,333051	0,779722	0,075977	0,891700	0,577997	0,544064
3	0,032463	0,374616	0,252517	0,291193	0,669397	0,687053
4	0,607324	0,337583	0,270796	0,799909	0,810546	0,530320
5	0,683711	0,048430	0,268431	0,286249	0,526409	0,981537
6	0,903745	0,823235	0,883192	0,379180	0,031492	0,094687
7	0,893327	0,700813	0,876160	0,313225	0,318481	0,726146
8	0,939292	0,308199	0,041690	0,667480	0,076510	0,504596
9	0,153619	0,025283	0,544786	0,339783	0,459579	0,020543

Рис. 8.9. Пример Pandas DataFrame

Как можно видеть, в созданном нами простом DataFrame содержится 60 случайных значений, собранных в шести разных столбцах и десяти записях. Мы сделали это только в качестве примера, но в реальности DataFrames обычно формируются с помощью CSV-файла, содержащего импортируемые данные. В интернете есть множество источников данных, которые предоставляют CSV-файлы для импорта с ценной информацией для анализа.

Итак, для практики давайте экспортируем вновь сгенерированные данные в файл CSV, который мы сможем использовать для наших примеров. Это несложно сделать с помощью метода `to_csv()`, вызванного на DataFrame, который нужно экспортировать:

```
df.to_csv("myDataFrame.csv", index=False)
```

Мы выполняем команду в ячейке Notebook и переходим к проверке текущего каталога. Здесь мы найдем новый файл `myDataFrame.csv`, содержащий 60 случайных значений, сгенерированных ранее. Параметр `index=False` используется для того, чтобы индексы нумерации строк DataFrame не отображались в файле CSV.

Таким образом, если необходимо импортировать данные в DataFrame pandas в нашем ноутбуке, достаточно использовать функцию `from_csv()`, которая вызывается непосредственно в библиотеке:

```
df = pd.read_csv("myDataFrame.csv")
df
```

При запуске ячейки в результате получим DataFrame, идентичный предыдущему, показанному на рис. 8.9.

То же самое можно сделать для создания Dask DataFrame. Синтаксис практически такой же.

```
ddf = dd.read_csv("myDataFrame.csv")
ddf
```

Однако, в отличие от предыдущего случая, мы получим результат, представленный на рис. 8.10.

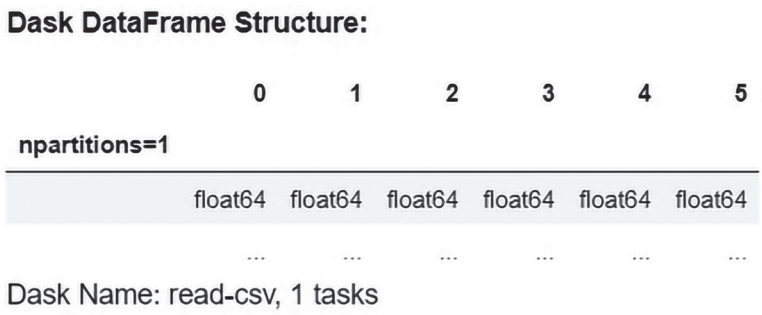


Рис. 8.10. Структура Dask DataFrame без данных

Как можно видеть, в DataFrame нет данных, а отображается только внутренняя структура DataFrame с названиями столбцов и соответствующими типами данных. Где же данные?

Хотя синтаксис Pandas и Dask очень похож, как и структура их DataFrame, они ведут себя по-разному. Схема поведения Dask отличается «ленивостью». Это означает, что он не выполняет операции, необходимые при вызове метода или определении объекта, такого как DataFrame. Для этого он ждет специальной команды, называемой `compute()`.

Итак, теперь в следующей ячейке Jupyter Notebook мы запускаем следующую команду:

```
ddf.compute()
```

В результате мы получим заполнение DataFrame теми же данными, что и в случае с `pandas`.

Еще один возможный способ импорта данных в Dask DataFrame состоит в том, чтобы получить их напрямую из другого Pandas DataFrame. Эта операция выполняется гораздо чаще, чем можно себе представить, особенно когда нужно масштабировать уже существующий код. В связи с этим для выполнения такой операции существует метод `from_pandas()`. В этом случае у нас есть возможность выбрать количество фрагментов, на которые будет разделен исходный DataFrame. Это число определяется с помощью опции `npartitions`, передаваемой в качестве аргумента функции:

```
ddf = dd.from_pandas(df, npartitions=4)
ddf
```

В результате выполнения приведенных выше строк кода мы получим результат, показанный на рис. 8.11.

Dask DataFrame Structure:

npartitions=3	0	1	2	3	4	5
0	float64	float64	float64	float64	float64	float64
3	...	...	...	...	...	...
6	...	...	...	...	...	...
9	...	...	...	...	...	...

Dask Name: from_pandas, 3 tasks

Рис. 8.11. Структура Dask DataFrame, разделенного на четыре раздела

В данном случае видно, что, по сравнению с предыдущим случаем, также отображаются четыре индекса строк, соответствующие первым строкам каждого раздела, на который будет разделен Dask DataFrame. Если нам нужно получить более подробную информацию, мы также можем выполнить следующую команду:

```
ddf.divisions
```

В результате мы получим кортеж индексов, соответствующих первым строкам каждого раздела DataFrame:

```
(0, 3, 6, 9)
```

Как можно видеть, результат соответствует тому, что показано на рис. 8.11.

Нам просто нужно улучшить DataFrame, как и ранее, с помощью функции `compute()`:

```
ddf.compute()
```

И мы получим тот же DataFrame, что и в предыдущих случаях.

Что касается массивов Dask, здесь ситуация очень похожа, только в этом случае вместо `pandas` DataFrame используются массивы NumPy. Затем мы определяем массив NumPy из 100 случайных целых значений от 0 до 9, собранных в матрице 10×10.

```
narr = np.random.randint(0,10,100).reshape(10,10)
narr
```

При выполнении мы получим результат, подобный следующему:

```
array([[5, 2, 5, 7, 2, 3, 9, 8, 2, 6],
       [9, 6, 3, 5, 0, 8, 5, 5, 1, 9],
       [6, 0, 3, 4, 3, 8, 1, 1, 7, 5],
       [3, 5, 5, 5, 3, 8, 3, 9, 3, 9],
       [3, 8, 5, 6, 0, 6, 0, 9, 7, 2],
       [7, 9, 4, 0, 4, 4, 3, 0, 6, 9],
       [9, 1, 7, 2, 4, 8, 9, 2, 6, 6],
       [5, 5, 3, 8, 1, 5, 7, 6, 4, 2],
       [8, 4, 8, 7, 0, 7, 6, 6, 9, 0],
       [1, 2, 8, 6, 8, 8, 0, 3, 0, 8]])
```

Вы можете напрямую импортировать значения этого массива NumPy непосредственно в массив Dask, используя на этот раз функцию `from_array()` библиотеки Dask. Опция `chunks` требует кортежа, содержащего размер `chunks` (фрагментов), на которые будет разбит массив Dask:

```
darr = da.from_array(narr, chunks=(5, 5))
darr
```

Выполнив приведенный выше код, вы получите графический результат, как показано на рис. 8.12.

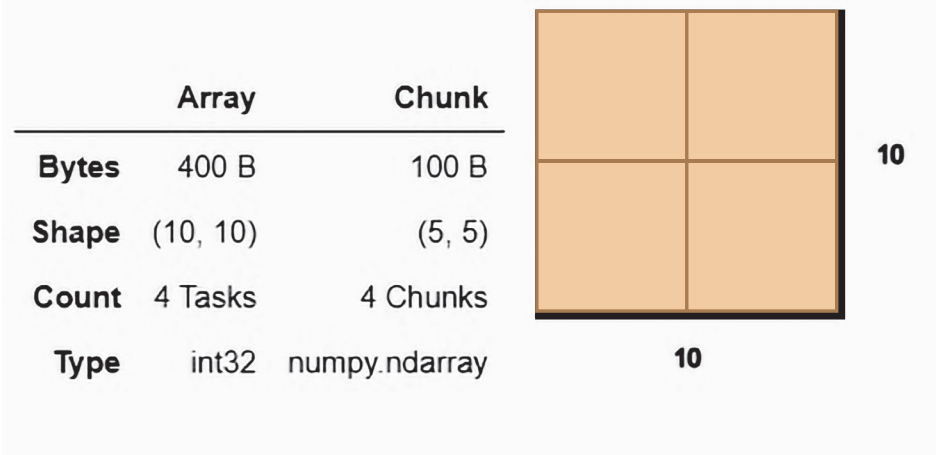


Рис. 8.12. Структура массива Dask

Как можно видеть, в этом случае мы также получили в результате только структуру массива Dask, поскольку он еще не получил значения (помните, что Dask является «ленивым»). В описании мы видим размеры каждой части или фрагмента (chunk), на которые будет разделен массив Dask. Каждому фрагменту будет назначена задача (task), поэтому заполнение массива (array) Dask также будет происходить в параллельном режиме.

Если нужна текстовая информация о разбиении массива Dask, на этот раз нужно использовать атрибут `chunks`:

```
darr.chunks
((5, 5), (5, 5))
```

Чтобы оценить массив Dask, снова запускаем функцию `compute()`:

```
darr.compute()
```

В результате в выводе будет отображено содержимое массива.

```
array([[2, 7, 3, 1, 3, 8, 1, 9, 0, 0],
       [0, 8, 0, 6, 3, 7, 7, 3, 4, 5],
       [0, 1, 2, 8, 6, 4, 3, 8, 5, 6],
       [1, 8, 4, 7, 8, 2, 4, 5, 1, 8],
       [4, 6, 4, 5, 5, 7, 4, 3, 8, 2],
       [8, 5, 3, 3, 9, 4, 5, 4, 2, 5],
       [9, 3, 1, 4, 0, 9, 5, 6, 7, 3],
       [5, 7, 1, 4, 8, 2, 2, 9, 3, 2],
       [1, 7, 9, 4, 7, 8, 9, 7, 2, 0],
       [6, 4, 3, 0, 6, 6, 6, 6, 4, 3]])
```

Методы для коллекций

Теперь, когда вы знаете, как определять наиболее распространенные коллекции, такие как DataFrame и Array, давайте попробуем применить к ним методы. Что касается номенклатуры и синтаксиса методов, все должно остаться таким же (или почти таким же), как и в соответствующих методах исходных библиотек.

Например, если мы хотим вычислить среднее значение первого столбца DataFrame Pandas, нам нужно написать следующее:

```
df['1'].mean()
```

В результате мы получим среднее значение, которое в данном конкретном случае составляет

```
0.42944239885994107
```

Так что в отношении Dask DataFrame ничего особо не изменится:

```
m = ddf['1'].mean()
```

В этом случае функция присваивается переменной. В действительности, как мы видели в случае с расширением DataFrame, то же правило применяется и к методам. Методы не выполняются до тех пор, пока не будет вызвана функция `compute()` для соответствующего метода. Таким образом, чтобы выполнить метод, следует написать:

```
m.compute()
```

И таким образом получить тот же результат, что и с Pandas DataFrame:

```
0.42944239885994107
```

То же самое относится к массивам Dask. Если вы хотите вычислить среднее значение содержимого массива, просто напишите:

```
mm = np.mean(darr)
```

Это вычисление будет выполнено только при вызове метода `compute()`:

```
mm.compute()
```

В нашем случае это даст следующий результат:

```
4.52
```

Пока что все просто, но давайте проанализируем причину этого странного ленивого поведения Dask и то, что на самом деле происходит внутри.

Вычисления и графы задач

Как можно видеть из предыдущих примеров, особенность библиотеки Dask, по сравнению с традиционными библиотеками, такими как Pandas и NumPy, заключается в ее поведении. Dask – это ленивая библиотека, и мы видели, что улучшение структуры данных, то есть коллекции в случае Dask, выполняется позднее, а не во время определения самой коллекции. Во время определения создается только структура данных, которую необходимо будет заполнить позднее. То же самое касается вызова методов, за которым никогда не следует вычисление, а только ожидание через присвоение переменной.

Заполнение коллекции и выполнение метода над ней происходит только тогда, когда метод `compute()` вызывается в коде на соответствующем объекте, определенном ранее. Эта операция называется вычислением (computing). Она выполняется иначе, чем в Pandas или NumPy. Как вы наверняка заметили, при определении объекта также происходит его разбиение, то есть определение количества частей, на которые он будет разделен. Каждой части будет назначена задача, которая, в свою очередь, будет назначена другому рабочему процессу. Затем будет выполнена оценка и выполнение вычислений внутри коллекции в параллельном режиме. Различные части, например DataFrame, будут заполняться и обрабатываться параллельно, что сократит время выполнения этой

операции. С небольшими CSV-файлами, по крайней мере с файлами размером менее 1 Гб, мы никогда не встречались с этой проблемой, но с более крупными CSV-файлами даже при расширении данных наблюдается огромное потребление ресурсов, особенно времени на заполнение. Таким образом, Dask, помимо распределенного упорядочения своих структур данных, части которых могут обрабатываться параллельно, также сокращает время оценки этих огромных объектов.

С вычислениями связана еще одна концепция – граф задач (Task Graph). Эти графы генерируются до начала операции и описывают процесс ее выполнения в системе или на отдельном компьютере. Они представляют собой путь выполнения, которому должен следовать планировщик задач. Графы задач являются динамическими и изменяются в зависимости от системы, на которой выполняется операция, что позволяет максимально оптимизировать параллельные вычисления.

Вы можете просмотреть эти графы задач, сгенерированные во время вычислений, в Jupyter Notebook. Например, если мы хотим увидеть граф задач среднего значения по массиву Dask в предыдущем случае, то можем ввести следующую команду:

```
mm.dask
```

В результате мы получим список слоев, на которые разделен граф задач, как показано на рис. 8.13.

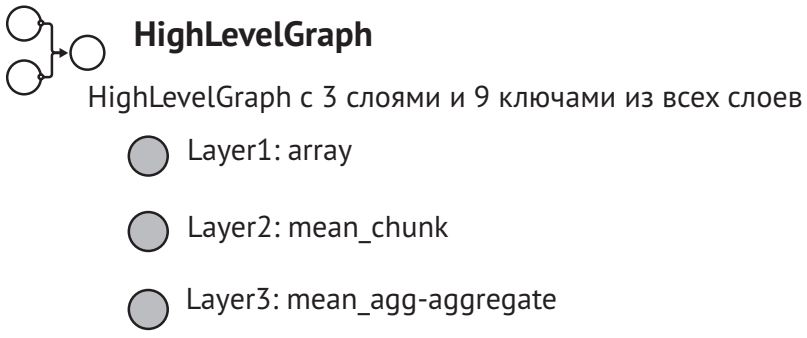


Рис. 8.13. Слои в графе задач

В то же время, если нам нужно увидеть граф задач, нужно написать следующее:

```
mm.visualize()
```

В результате мы получим граф, как показано на рис. 8.14.

Как можно видеть, вычисление среднего значения выполняется в параллельном режиме между четырьмя частями (фрагментами) матрицы. Затем четыре результата, полученные из отдельных задач, объединяются для получения окончательного результата.

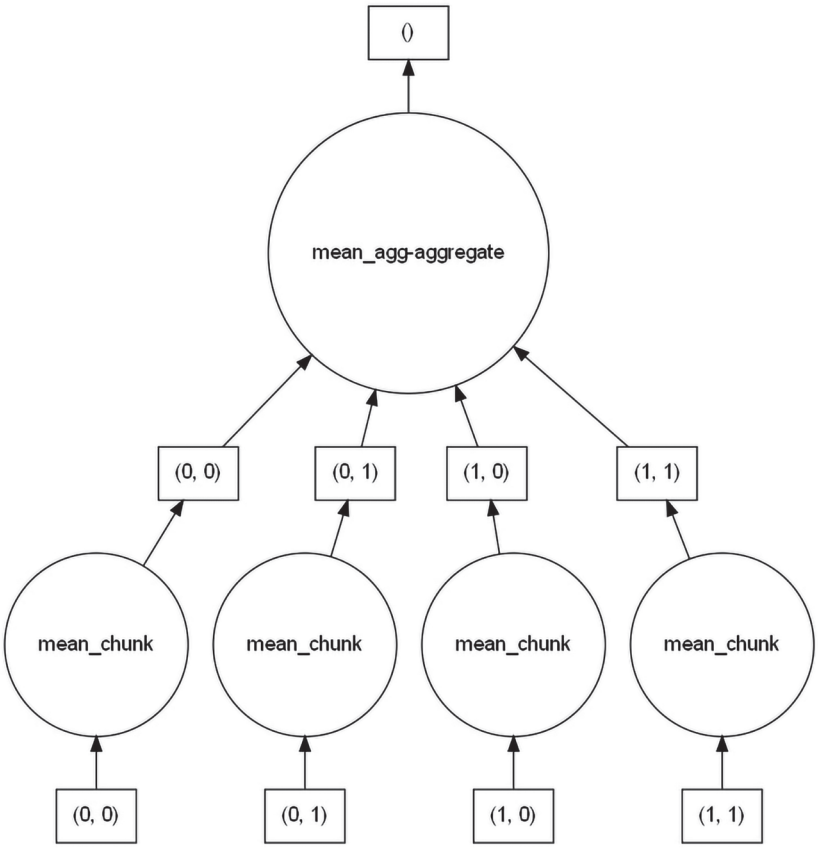


Рис. 8.14. Граф задач для вычисления среднего значения по массиву Dask

Низкоуровневое взаимодействие с Dask Delayed

В дополнение к методам, предоставляемым библиотеками Pandas и NumPy, которые затем были воспроизведены в соответствующих коллекциях Dask, в коде зачастую может возникнуть необходимость определить новые методы, применимые к этому типу объектов. Но как в этом случае мы можем расширить эти новые методы для параллельных вычислений с Dask, если они в нем не реализованы?

В этом отношении на помощь приходит **Dask Delayed** (Dask отложенный). Этот инструмент позволяет обернуть отдельные функции, присвоив им граф задач. Эти функции будут включены в низкоуровневый механизм ленивой конструкции и будут вести себя так же, как методы и коллекции Dask.

На практике функции, подлежащие преобразованию, будут определены декоратором `@dask.delayed`. Например, при определении функции, которая работает с массивом Dask, удваивая все его элементы, мы будем использовать декоратор.

```
import dask
@dask.delayed
def doubled(x):
    return 2*x
```

И поэтому, если мы сейчас вызовем эту функцию, мы получим объект Dask `Delayed`.

```
doubled(darr)
Delayed('doubled-5f08d4cc-aaeb-43bb-9577-7c8f055af1fd')
```

Итак, чтобы выполнить вычисление функции в параллельном режиме, мы сначала вызываем ее, присваивая переменной Dask Delayed, а затем запускаем для нее метод `compute()`.

```
d = doubled(darr)
d.compute()
```

В результате получаем матрицу с удвоенными элементами, вычисленную параллельно четырьмя разными задачами.

```
array([[ 4, 14,  6,  2,  6, 16,  2, 18,  0,  0],
       [ 0, 16,  0, 12,  6, 14, 14,  6,  8, 10],
       [ 0,  2,  4, 16, 12,  8,  6, 16, 10, 12],
       [ 2, 16,  8, 14, 16,  4,  8, 10,  2, 16],
       [ 8, 12,  8, 10, 10, 14,  8,  6, 16,  4],
       [16, 10,  6,  6, 18,  8, 10,  8,  4, 10],
       [18,  6,  2,  8,  0, 18, 10, 12, 14,  6],
       [10, 14,  2,  8, 16,  4,  4, 18,  6,  4],
       [ 2, 14, 18,  8, 14, 16, 18, 14,  4,  0],
       [12,  8,  6,  0, 12, 12, 12, 12,  8,  6]])
```

Начало работы на кластере машин

До сих пор мы использовали Dask на одном компьютере, чтобы изучить принцип работы этой библиотеки и ее уникальные особенности. Теперь пришло время перейти к более реалистичным задачам, где потребуется обрабатывать огромные наборы данных и запускать программу в среде кластера из нескольких компьютеров.

Сообщество Kaggle

Для начала поговорим о данных. Как уже упоминалось ранее, в интернете есть множество сайтов с наборами данных для просмотра и анализа. Одним из наиболее интересных сайтов можно назвать Kaggle (kaggle.com). Kaggle – это веб-сообщество специалистов по обработке данных, преимущественно занимающихся машинным обучением. Помимо множества интересных фактов, конкурсов и тренингов по анализу данных и изучению моделей прогнозиро-

вания, сайт предоставляет своим посетителям множество наборов данных по разным темам.

Просматривая различные доступные наборы данных, выберем наиболее подходящий для наших целей. Выберем, например, набор данных по истории акций Russell 3000 по адресу *https://www.kaggle.com/datasets/williecosta/russell-3000-stock-history*, как показано на рис. 8.15.

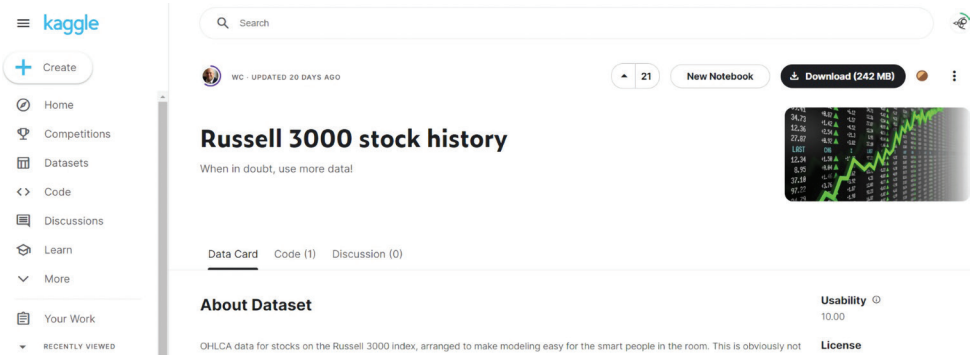


Рис. 8.15. Набор данных по истории акций Russell 3000 в Kaggle

На странице набора данных можно загрузить файл `archive.zip` размером 242 Мб, нажав черную кнопку в правом верхнем углу. После загрузки ZIP-архива его необходимо распаковать. Внутри вы найдете файл `prices.csv`, который после извлечения займет более 800 Мб памяти.

Кластер Saturn.io

В интернете можно найти различные платные сервисы, которые предоставляют кластеры для выполнения вычислений. Среди них есть платформа с поддержкой библиотеки Dask, которая предлагает удобную среду для работы: Saturn. io. Этот сервис предлагает ряд приложений для работы с данными, которые упрощают распределенное программирование с помощью Python и Dask, а также других языков, используемых для анализа данных и статистики, таких как R, как показано на рис. 8.16. Кроме того, Saturn.io предоставляет онлайн-поддержку через чат, а также ряд интересных статей о Dask, кластерах и других приложениях, связанных с анализом данных на кластерах.

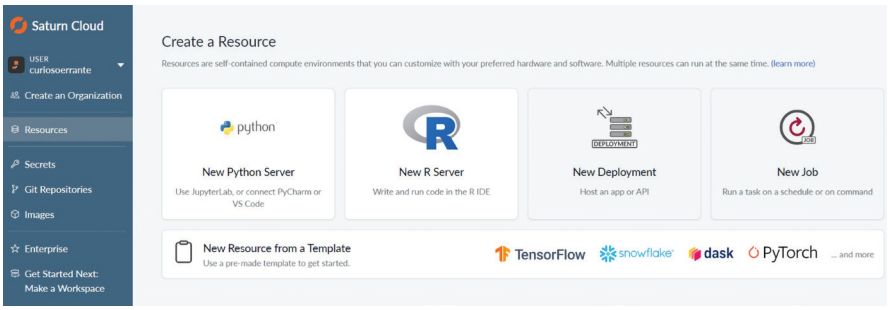


Рис. 8.16. Saturn.io

Примечательно, что после регистрации пользователю предоставляется возможность бесплатно настроить и использовать простой кластер из четырех машин в течение 30 ч. В случае автора этой книги была настроена система из сервера Jupyter, непосредственно подключенного к кластеру Dask с планировщиком задач и тремя рабочими процессами. Все это было сделано за несколько минут с помощью простого выбора характеристик каждой машины на веб-странице, как показано на рис. 8.17.

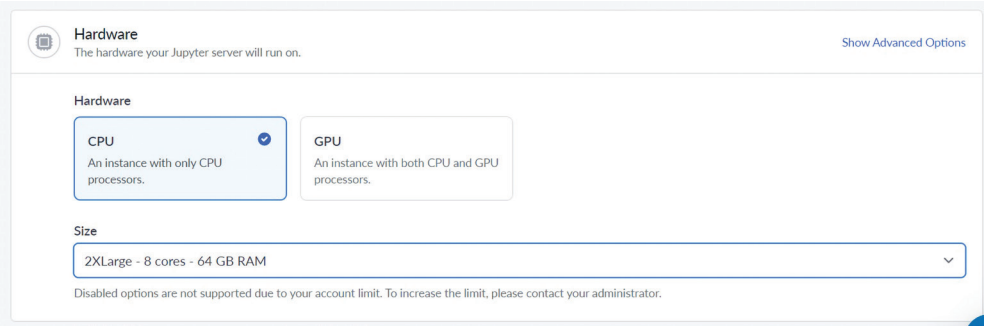


Рис. 8.17. Настройка отдельных машин кластера на Saturn.io с помощью инструкций

После настройки необходимо запустить все машины. Нажмите кнопку **Start** (Запуск) как для сервера Jupyter, так и для кластера Dask. Вы сразу же увидите этапы запуска отдельных машин, которые отображаются в полях с указанием количества машин, присутствующих на каждом этапе, как показано на рис. 8.18. Весь процесс загрузки занимает несколько минут.

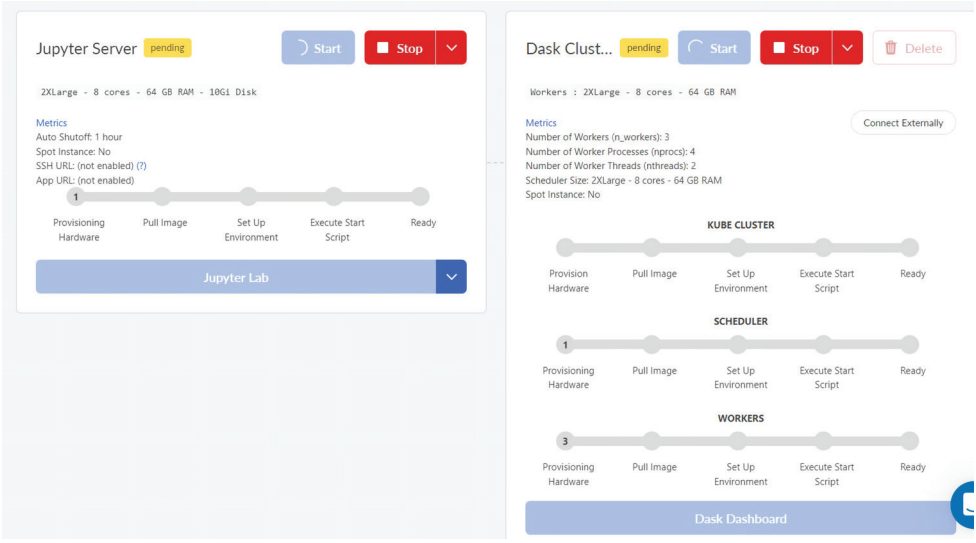


Рис. 8.18. Этапы запуска машин кластера Dask

Наконец, когда все машины будут готовы к работе, этапы загрузки будут скрыты, как показано на рис. 8.19.

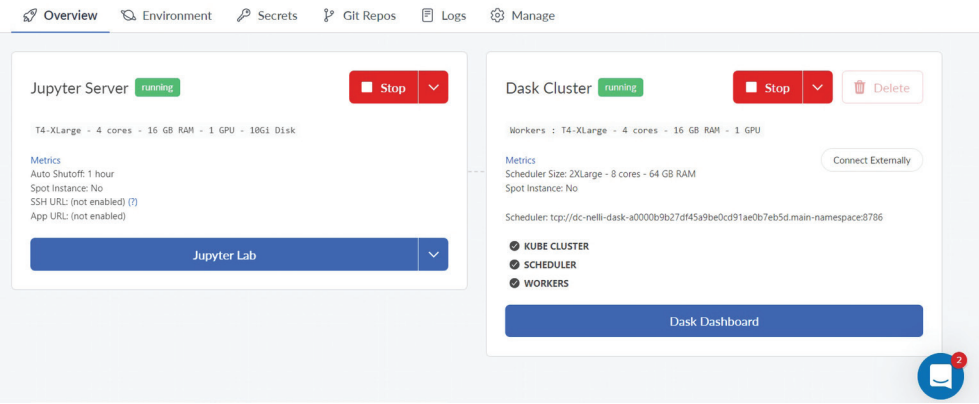


Рис. 8.19. Рабочий кластер Dask со всеми запущенными машинами

Загрузка данных в кластер

Прежде чем приступить к программированию кластера, необходимо загрузить файл данных `prices.csv` в вашу систему. Для этого следует воспользоваться Jupyter Lab. Откройте приложение на сервере Jupyter, нажав синюю кнопку внизу окна, и перейдите в файловую систему Jupyter. Создайте папку в рабочей области, назвав ее, например, data, а затем начните загрузку файла. С учетом размера файла операция займет несколько минут. По окончании операции вы увидите значок файла `prices.csv` в только что созданном каталоге, как на рис. 8.20.

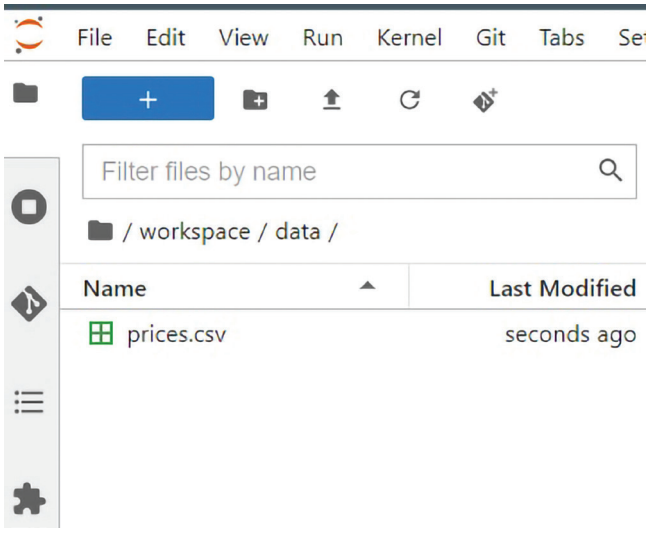


Рис. 8.20. Загрузка файла CSV с набором данных в Jupyter Lab

Приступая к программированию кластера

Теперь все должно быть готово и настроено для начала работ по программированию. Откройте новый Jupyter Notebook на сервере Jupyter и начните импортировать следующие модули:

```
from dask_saturn import SaturnCluster
from distributed import Client

cluster = SaturnCluster()
client = Client(cluster)
client
```

Как можно видеть, в нашей системе с одной машиной их не было. На самом деле для работы в кластере необходимо импортировать определенные модули кластерной среды, в которой вы работаете. Однако они предоставляются хост-системой, и по этому поводу нет причин для беспокойства. Затем импортируйте класс `SaturnCluster` из модуля `dask_saturn` и запустите экземпляр кластера. Кроме того, для того чтобы Dask распознал, что вы работаете на системе этого типа, а не на одной машине (ситуация по умолчанию), вам также необходимо создать экземпляр клиента, предоставляемого модулем distributed.

Наконец, мы готовы импортировать библиотеки, необходимые для анализа данных. Это практически те же библиотеки, которые используются в системах с одной машиной:

```
import dask.dataframe as dd
import pandas as pd
```

В предыдущем разделе мы загрузили файл `prices.csv` на сервер Jupyter. Но работа еще не завершена. В действительности наша система включает пять разных машин, каждая со своей памятью. Поэтому необходимо добавить код синхронизации данных между сервером Jupyter и рабочими процессами, чтобы эти данные были доступны для всех машин.

```
from dask_saturn import RegisterFiles, sync_files

client.register_worker_plugin(RegisterFiles())
sync_files(client, 'data')
```

Таким образом, все наборы данных, импортированные в папку данных Jupyter Server, становятся доступны для всех машин в кластере. Запустив эту ячейку в Notebook, вы заметите, что выполнение операции займет несколько минут. По окончании выполнения вы получите в качестве результата файловую систему, синхронизированную между всеми машинами в кластере.

```
./
./prices.csv
```

Теперь мы наконец можем импортировать данные из файла `prices.csv` в Dask DataFrame. Для этого напишем ту же строку кода, что и в предыдущем случае:

```
ddf = dd.read_csv('data/prices.csv')
```

Выполнение будет очень быстрым, потому что на самом деле никакие данные загружаться не будут. Фактически если мы посмотрим на содержимое вновь созданного Dask DataFrame, то увидим, что загрузилась только структура без данных в ней:

```
ddf
```

Получение структуры данных показано на рис. 8.21.

Dask DataFrame Structure:

	Unnamed: 0	symbol	date	open	high	low	close	volume	adjusted
npartitions=13									
	int64	object	object	float64	float64	float64	float64	int64	float64
	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...	...

Dask Name: read-csv, 13 tasks

Рис. 8.21. Структура Dask DataFrame в кластере

Чтобы загрузить данные, нужен следующий код:

```
ddf.compute()
```

Только после вызова метода `compute()` мы сможем проанализировать данные из файла CSV. Но в данном случае появится сообщение об ошибке, связанной с типами данных:

```
ValueError: Mismatched dtypes found in `pd.read_csv`/`pd.read_table`.
```

Как можно видеть, несмотря на схожий синтаксис Pandas и Dask, для Dask метод `read_csv()` более требователен, чем для его аналога в `pandas`. В действительности при попытке загрузить CSV-файл с помощью этого метода никаких сообщений об ошибках не появляется. Однако в следующем тексте вывода (см. рис. 8.22) сообщение об ошибке подсказывает способ решения проблемы.

В данном случае речь о несовпадении обнаруженного (Found) и ожидаемого (Expected) формата данных (float64 вместо int64). Пояснение гласит, что «обычно это происходит из-за сбоя в определении типа данных Dask и *может* быть исправлено путем ручного указания типов данных путем добавления» (usually this is due to dask's dtype inference falling, and *may* be fixed by specifying dtypes manually by adding) необходимых строк кода. Ниже в качестве альтернатив-

ного решения предлагается «указать `assume_missing=True`, чтобы интерпретировать все неуказанные столбцы целых чисел как числа с плавающей запятой» (provide `assume_missing=True` to interpret all unspecified integer columns as floats).

```
+--------+---------+----------+
| Column | Found   | Expected |
+--------+---------+----------+
| volume | float64 | int64    |
+--------+---------+----------+

Usually this is due to dask's dtype inference failing, and
*may* be fixed by specifying dtypes manually by adding:

dtype={'volume': 'float64'}

to the call to `read_csv`/`read_table`.

Alternatively, provide `assume_missing=True` to interpret
all unspecified integer columns as floats.
```

Рис. 8.22. Сообщение об ошибке при импорте CSV-файла

Затем мы переписываем метод `read_csv()`, добавляя предложенные изменения:

```
ddf = dd.read_csv('data/prices.csv', dtype={'volume': 'float64'})
ddf.compute()
```

На этот раз все будет в порядке, и мы получим загрузку данных в Dask DataFrame, как показано на рис. 8.23.

	Unnamed: 0	symbol	date	open	high	low	close	volume	adjusted
0	1	FLWS	1999-08-03	21,7500	22,2500	17,875	18,1875	9543400,0	18.1875
1	2	FLWS	1999-08-04	18,9375	20,5000	16,875	17,0000	4731800,0	17.0000
2	3	FLWS	1999-08-05	17,0625	17,1250	13,500	16,0000	2851900,0	16.0000
3	4	FLWS	1999-08-06	16,0000	17,7500	16,000	17,0000	1411800,0	17.0000
4	5	FLWS	1999-08-09	17,9375	17,9375	15,875	16,0000	583000,0	16.0000
...	...	...	...	...	...	...	...	...	...
991985	12995441	ZUMZ	2022-11-07	22,4100	22,8800	22,020	22,6700	172900,0	22,6700
991986	12995442	ZUMZ	2022-11-08	22,8700	23,3600	22,220	22,4700	311800,0	22,4700
991987	12995443	ZUMZ	2022-11-09	22,3600	22,4300	20,640	20,7800	239800,0	20,7800
991988	12995444	ZUMZ	2022-11-10	21,8300	23,6900	21,780	23,5100	327600,0	23,5100
991989	12995445	ZUMZ	2022-11-11	23,7700	24,6200	23,670	24,2800	205000,0	24,2800

12995445 rows × 9 columns

Рис. 8.23. DataFrame, полученный из файла prices.csv

Теперь рассмотрим более подробно характеристики DataFrame.

```
ddf.info(memory_usage="deep")
<class 'dask.dataframe.core.DataFrame'>
Columns: 9 entries, Unnamed: 0 to adjusted
dtypes: object(2), float64(6), int64(1)
memory usage: 892.3 MB
```

Как можно видеть, тип данных столбца **Date** – не **DateTime**, а `object`. Кроме того, первый столбец, содержащий индексы строк в файле CSV, бесполезен, и его можно удалить. Поэтому мы переписываем:

```
ddf = dd.read_csv('data/prices.csv', dtype={'volume': 'float64'}, parse_
dates = ['date'])
del ddf['Unnamed: 0']
ddf
```

На этом этапе структура и типы данных корректны, как показано на рис. 8.24.

Dask DataFrame Structure:

	symbol	date	open	high	low	close	volume	adjusted
npartitions=13								
	object	datetime64[ns]	float64	float64	float64	float64	float64	float64
	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...

Dask Name: drop_by_shallow_copy, 26 tasks

Рис. 8.24. Структура данных правильного DataFrame

Также любопытно посмотреть на то, как наш Dask DataFrame был автоматически разделен на тринадцать разделов.

Нам остается только оценить его:

```
ddf.compute()
```

Результат будет таким, как показано на рис. 8.25.

На этом этапе мы можем применить операцию к DataFrame. Выберем умеренно сложную операцию, которая позволит сгруппировать результаты по значениям в определенных столбцах:

```
mean = ddf.groupby(['symbol',ddf['date'].dt.year])[['open','high','low',
'close','volume']].mean()
```

	symbol	date	open	high	low	close	volume	adjusted
0	FLWS	1999-08-03	21,7500	22,2500	17,875	18,1875	9543400,0	18,1875
1	FLWS	1999-08-04	18,9375	20,5000	16,875	17,0000	4731800,0	17,0000
2	FLWS	1999-08-05	17,0625	17,1250	13,500	16,0000	2851900,0	16,0000
3	FLWS	1999-08-06	16,0000	17,7500	16,000	17,0000	1411800,0	17,0000
4	FLWS	1999-08-09	17,9375	17,9375	15,875	16,0000	583000,0	16,0000
...	...	...	...	...	...	...	...	...
991985	ZUMZ	2022-11-07	22,4100	22,8800	22,020	22,6700	172900,0	22,6700
991986	ZUMZ	2022-11-08	22,8700	23,3600	22,220	22,4700	311800,0	22,4700
991987	ZUMZ	2022-11-09	22,3600	22,4300	20,640	20,7800	239800,0	20,7800
991988	ZUMZ	2022-11-10	21,8300	23,6900	21,780	23,5100	327600,0	23,5100
991989	ZUMZ	2022-11-11	23,7700	24,6200	23,670	24,2800	205000,0	24,2800

12995445 rows × 8 columns

Рис. 8.25. Вычисление задачи DataFrame

На этом этапе, поскольку нас также интересует аспект производительности, мы введем магический код Jupyter Notebook, `%%time`, который дает возможность измерить время выполнения инструкций в ячейке. Затем выполним операцию группировки над Dask DataFrame.

```
%%time
mean.compute()
```

В результате получаем время выполнения вычислений.

```
CPU times: user 6.23 ms, sys: 12.3 ms, total: 18.5 ms
Wall time: 2.04 s
```

В результате Dask DataFrame выглядит так, как показано на рис. 8.26.

Теперь для сравнения выполним те же операции, но на локальном уровне с Pandas DataFrame:

```
df = pd.read_csv('data/prices.csv', dtype={'volume': 'float64'}, parse_
dates = ['date'])
del df['Unnamed: 0']
df
```

А в следующей ячейке блокнота введем следующее:

```
%%time
df.groupby(['symbol',df['date'].dt.year])[['open','high','low','close',
'volume']].mean()
```

		open	high	low	close	volume
symbol	date					
A	1999	33,984561	35,447962	33,226037	34,580651	5,739997e+06
	2000	49,966151	51,826004	48,017594	49,877136	4,238259e+06
	2001	23,837856	24,570816	23,174646	23,912350	3,311148e+06
	2002	16,433482	16,814270	16,050480	16,414407	3,700359e+06
	2003	14,447368	14,686315	14,223693	14,478762	3,244672e+06
...	...	...	...	...	...	...
ZUMZ	2018	23,494422	23,925498	22,961633	23,433625	4,686191e+05
	2019	26,504286	26,986825	26,019841	26,532857	4,514512e+05
	2020	27,748182	28,426561	26,997787	27,699881	3,554577e+05
	2021	44,467183	45,372540	43,545119	44,431587	2,604456e+05
	2022	32,647477	33,432982	31,858028	32,622982	3,536784e+05

53124 rows × 5 columns

Рис. 8.26. Результат группировки по Task DataFrame

На этот раз мы получим такое время обработки:

```
CPU times: user 1.96 s, sys: 396 ms, total: 2.35 s
Wall time: 2.35 s
```

Как можно видеть, теперь время выполнения намного превышает показатели предыдущего случая. Здесь мы имеем дело с секундами, тогда как раньше с Dask речь шла о миллисекундах. И ситуация будет меняться все больше по мере увеличения размера DataFrame и сложности выполняемых над ним операций.

В качестве примера давайте увеличим размер рассматриваемого DataFrame в четыре раза, соединив четыре его копии:

```
mddf = dd.concat([ddf, ddf, ddf, ddf])
mddf.info(memory_usage="deep")
mddf
```

Тем самым мы получили значительно более объемный Dask DataFrame, как видно из результата предыдущих строк кода и на рис. 8.27.

```
<class 'dask.dataframe.core.DataFrame'>
Columns: 8 entries, symbol to adjusted
dtypes: datetime64[ns](1), object(1), float64(6)
memory usage: 3.1 GB
```

Dask DataFrame Structure:

	symbol	date	open	high	low	close	volume	adjusted
npartitions=52								
	object	datetime64[ns]	float64	float64	float64	float64	float64	float64
	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...
	...	...	...	...	...	...	...	...

Dask Name: concat, 78 tasks

Рис. 8.27. Структура DataFrame, увеличенного в четыре раза

Повторим те же операции, что и ранее:

```
mddf.compute()
mean = mddf.groupby(['symbol',mddf['date'].dt.year])[['open','high','low',
'close','volume']].mean()
%%time
mean.compute()
```

В этот раз получается следующий результат:

```
CPU times: user 15.9 ms, sys: 5.77 ms, total: 21.7 ms
Wall time: 2.92 s
```

Как можно видеть, по сравнению с предыдущим случаем ситуация практически не изменилась. Теперь попробуем сделать то же самое с DataFrame, увеличенным в четыре раза, но с использованием `pandas`:

```
mdf = pd.concat([df, df, df, df]) mdf.info(memory_usage="deep")
mdf
%%time
mdf.groupby(['symbol',mdf['date'].dt.year])[['open','high','low','close',
'volume']].mean()
```

И без параллелизации полученное время оказалось в четыре раза больше:

```
CPU times: user 7.98 s, sys: 1.39 s, total: 9.37 s
Wall time: 9.36 s
```

Заключение

В этой главе мы рассмотрели возможности расширения вычислительных мощностей для аналитики данных с помощью систем, распределенных в виде кластеров, с использованием библиотеки Dask. Вы узнали, как масштабировать

код, написанный с помощью Pandas, с системы на одной машине до распределенных систем. Вы также познакомились с кластерами в интернете, которые предлагают определенные услуги и позволяют работать через Jupyter Notebook с исходным кодом, использующим библиотеку Dask. Благодаря нескольким практическим примерам вы ближе познакомились с этими средами и с теми преимуществами, которые они дают по сравнению с использованием одного компьютера, что позволит нам также обратиться к Big Data. В следующей главе мы также расширим эти концепции на искусственный интеллект (ИИ), в частности на такие его области, как машинное обучение и глубокое обучение с использованием нейронных сетей, которые быстро набирают популярность благодаря параллельным вычислениям на распределенных системах.

Список литературы

- *https://saturncloud.io/blog/what-is-dask/*.
- *https://saturncloud.io/blog/should-i-use-dask/*.

Глава 9

Раскрытие потенциала ИИ с параллельными вычислениями

На этой главе мы завершаем раздел, посвященный приложениям параллельных вычислений. Заключительная тема – искусственный интеллект. Эта дисциплина информационных технологий занимается разработкой приложений, способных имитировать человеческий интеллект. Это сложная задача, требующая обширных исследований и мощных технологий, но в последние годы в этой области достигнуты значительные успехи. Среди всего спектра возможных приложений особое место занимает машинное обучение – направление разработки искусственного интеллекта, посвященное имитации человеческого обучения. Это направление, в частности, получило свое наиболее яркое воплощение в языке программирования Python, имеющем ряд хорошо проработанных библиотек алгоритмов и готовых моделей, таких как Scikit-learn. Еще более перспективным направлением является глубокое обучение – одно из направлений машинного обучения, для которого характерно использование нейронных сетей для создания приложений, способных распознавать информацию и анализировать данные, такие как изображения, звуки и тексты. Все эти отрасли требуют значительных вычислительных ресурсов для обработки очень сложных алгоритмов и огромных объемов данных, необходимых для их обучения. Использование параллельных вычислений с интеграцией графических процессоров (GPU) для этих целей является необходимым условием для существенного сокращения времени процессорных операций. Кроме того, в последнее время для этих систем были предложены методы обработки на распределенных системах, таких как компьютерные кластеры с графическими процессорами и с использованием фреймворка Dask. В связи с этим мы рассмотрим ряд примеров, которые охватывают все эти случаи, что позволит получить базовые представления о методах работы с этими системами, а тем, кто желает узнать больше, даст основу для старта.

Структура

В этой главе мы обсудим следующие темы:

- ИИ и параллельные вычисления;
- параллельное и распределенное машинное обучение;
- параллельное и распределенное глубокое обучение;
- PyTorch и TensorFlow;
- кластеры Dask и SaturnCloud.

Искусственный интеллект (ИИ)

Одной из самых новых областей развития, тесно связанной с компьютерами и анализом данных, является искусственный интеллект (ИИ). Под этим термином обычно подразумеваются системы или машины, которые каким-то образом способны имитировать человеческий интеллект, особенно в области обучения. Эти системы, обычно состоящие из различных очень сложных алгоритмов, способны непрерывно совершенствоваться в выполнении заданной деятельности благодаря процессам обучения, которые позволяют им накапливать опыт на основе собранной информации. Чем больше информации они собирают, тем более совершенными становятся их способности.

Хотя термин «искусственный интеллект» всегда ассоциируется с образами человекоподобных роботов, на самом деле это совершенно разные вещи. Сегодня системы искусственного интеллекта получили достаточно широкое распространение и присутствуют в нашей повседневной жизни в самых разных формах. В большинстве случаев это нематериальные, практически абстрактные сущности, которые проявляют себя совсем не так, как обычно представляют себе андроидов. Обычно они управляют поисковыми системами или работают в социальных сетях и других сервисах, анализируя наши привычки, вкусы и потребности. Эти системы исследуют варианты нашего предыдущего выбора и на их основе прогнозируют наши будущие потребности, предлагая, например, серию фильмов на YouTube или потенциально интересные нам темы. Цель этих систем (по крайней мере, наиболее очевидная) заключается в облегчении пользовательского выбора самого интересного материала без потери времени на просмотр тысяч возможных вариантов.

Еще одно современное применение систем искусственного интеллекта – это чат-боты. Эти чаты, часто присутствующие на веб-сайтах поставщиков услуг, предназначены для сбора запросов клиентов по поводу информации или возникших проблем. Они предлагают клиенту самые быстрые способы решения проблемы, например заполнение и отправку формы, номер телефона конкретного сотрудника или даже несколько шагов для решения проблемы из дома. Для работы этих чат-ботов требуется умение наилучшим образом заменить оператора-человека.

В то же время глубокое использование искусственного интеллекта менее очевидно, чем обучение с помощью параметризуемых модельных систем,

которые используют поток большого объема входящих данных для повышения точности и возможности прогнозирования сложных явлений. Этот подход привел к появлению ряда дисциплин, таких как машинное обучение (Machine Learning) и глубокое обучение (Deep Learning).

ИИ, машинное обучение и глубокое обучение

Термин «искусственный интеллект» нередко используется как синоним таких терминов, как «машинное обучение» и «глубокое обучение». Однако между этими концепциями существуют существенные различия. Машинное обучение – это методология вычислений для разработки систем, способных учиться и улучшать свои результаты в определенных видах деятельности на основе предоставленных данных. Таким образом, можно с уверенностью утверждать, что машинное обучение относится к искусственному интеллекту, но нельзя сказать обратное, то есть что искусственный интеллект эквивалентен машинному обучению. Принцип, который поможет понять различия между этими концепциями, показан на рис. 9.1.

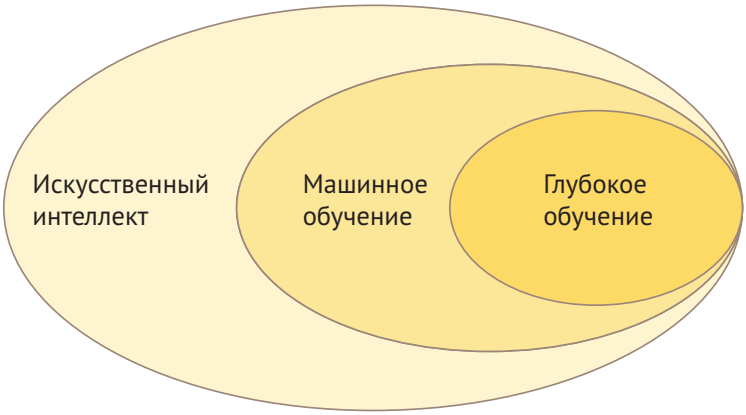


Рис. 9.1. Различия между ИИ, машинным обучением и глубоким обучением

Глубокое обучение – это еще более сложная концепция. Методология по-прежнему направлена на разработку систем, способных к самообучению, но при этом еще больше похожих на человека. Эти системы основаны на искусственных нейронных сетях, распределенных по нескольким слоям, подобно нейронам в человеческом мозге, как показано на рис. 9.2.

Это сложные системы, способные извлекать информацию и преобразовывать сложные данные, такие как человеческие голоса или изображения. Поэтому они могут не только обучаться, но также распознавать любые объекты или слова в нередко сложном контексте. Кроме того, в этом случае искусственные нейронные сети должны пройти ряд этапов обучения, чтобы быть способными максимально эффективно выполнять свои задачи. Таким образом, глубокое обучение можно считать особой методикой машинного обучения, форму искусственного интеллекта, но не наоборот (см. рис. 9.1).

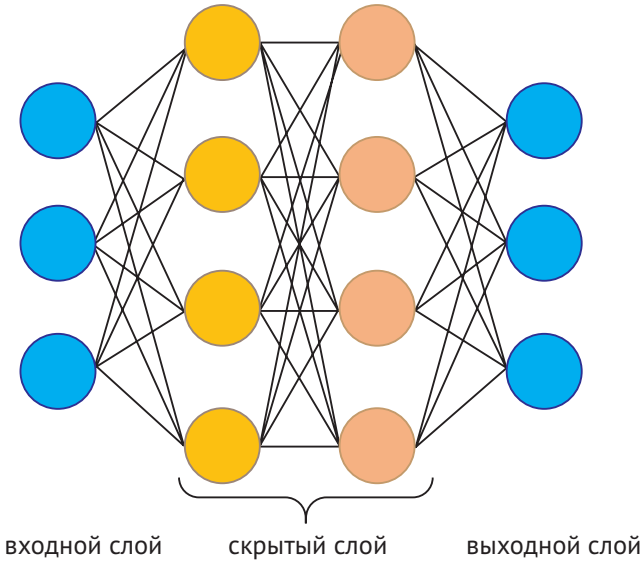


Рис. 9.2. Пример искусственной нейронной сети

Глубокое обучение имеет множество областей применения, рассмотрим некоторые из наиболее примечательных.

Обработка естественного языка (Natural Language Processing, NLP): глубокое обучение доказало свою эффективность в распознавании устной речи (распознавание речи), а также является особенно эффективным при обработке текстовой документации. В действительности оно способно искать имена, понятия и данные в тысячах публикаций и разнообразных источниках.

Компьютерное зрение (computer vision): глубокое обучение заложило основу для интерпретации изображений, особенно в области распознавания лиц, а также для решения таких задач, как определение местоположения транспортных средств и пешеходов по изображениям с веб-камер.

Фармакология: глубокое обучение способно анализировать диагностические изображения и обучаться распознаванию любых аномалий. Оно также может извлекать информацию из физиологических характеристик и обнаруживать патологии.

Биология: глубокое обучение доказало свою полезность при классификации белков, геномике, классификации клеток и анализе белково-белковых взаимодействий.

Генерация изображений: глубокое обучение дает возможность интеллектуально модифицировать изображения с помощью сложных операций, таких как раскрашивание черно-белых изображений, увеличение разрешения изображений, удаление шума из изображений, а также создание произведений искусства в стиле известных художников на основе базовых изображений.

Поиск в интернете: поисковые системы в интернете используют глубокое обучение.

Игры: все больше игровых проектов делают ставку на искусственный интеллект в целях улучшения стратегического планирования и имитации человеческого поведения у виртуальных противников.

Робототехника: глубокое обучение находит применение при расчете траекторий и динамики сложных робототехнических систем, предназначенных для управления объектами.

Искусственный интеллект, будь то машинное обучение, глубокое обучение или что-то еще, в скором времени станет одним из основополагающих элементов в корпоративных процессах принятия решений будущего.

Контролируемое и неконтролируемое обучение

После определения областей обучения в рамках концепции искусственного интеллекта очень важно ввести еще одно фундаментальное определение, прежде чем перейти к обсуждению вопроса о его применении в параллельных вычислениях.

Все методики обучения, как машинного, так и глубокого, основываются на двух подходах относительно типа данных, используемых на этапе обучения:

- контролируемое обучение;
- неконтролируемое (unsupervised) обучение.

Эти два подхода различаются по наличию или отсутствию маркировки (меток) для используемых данных. Под маркировкой подразумевается наличие или отсутствие правильного ответа для данных.

При контролируемом (supervised) обучении для тренировки моделей используются данные, помеченные правильными ответами. Знание правильного ответа используется для уточнения возможностей модели в ее прогнозах. Этот подход используется в двух конкретных методах:

- классификации;
- регрессии.

Классификация (classification) использует алгоритмы для классификации исследуемых данных. К наиболее часто используемым алгоритмам относятся:

- логистическая регрессия;
- метод k-ближайших соседей;
- случайный лес;
- наивный байесовский алгоритм;
- стохастический градиентный спуск;
- деревья решений.

Регрессия (regression) использует алгоритмы, которые измеряют взаимосвязь между зависимой переменной и одной или несколькими независимыми переменными. К наиболее известным из них относятся:

- регрессия Риджа;
- лассо;
- регрессия нейронной сети;
- логистическая регрессия.

В случае неконтролируемого (unsupervised) обучения данные не содержат никакой дополнительной информации о своей природе, кроме собственно самих данных. Поэтому алгоритмы этого типа могут работать без какого-либо контроля со стороны человека. Этот подход является предпочтительным для глубокого обучения.

Существует три различных метода, которые следуют этому подходу:

- кластеризация;
- ассоциация;
- снижение размерности.

Кластеризация (clustering) опирается на сходство или различия исследуемых данных, которые группируются по определенным характеристикам. Наиболее известный алгоритм, следующий этой методике – это метод k-средних (k-means clustering).

Ассоциация (association) позволяет выявить взаимосвязь между переменными в наборе данных. Эти взаимосвязи не определяются до начала изучения набора данных.

Уменьшение размерности (dimensionality reduction) – это техника, которая служит для уменьшения количества исследуемых характеристик в наборе данных, поскольку число этих характеристик иногда может быть слишком большим. Алгоритмы, основанные на этой технике, способны уменьшать количество характеристик, которые отличают данные в наборе, без ущерба для их целостности.

Искусственный интеллект и параллельные вычисления

Как можно легко догадаться, создание сложных компьютерных систем, способных выполнять задачи искусственного интеллекта, требует огромных вычислительных ресурсов, и по мере роста их возможностей понадобятся все более мощные вычислительные машины. Именно по этой причине системы ИИ получили широкое распространение только в последние годы. Следовательно, для работы в этом секторе и разработки систем ИИ, способных выполнять все более сложные задачи, нам придется использовать системы, основанные на параллельных вычислениях, которые в настоящее время являются единственными, способными предоставлять необходимые вычислительные ресурсы.

Например, облачные вычисления предоставляют огромные вычислительные мощности, что дает разработчикам возможность получить высокопроизводительные вычислительные ресурсы по доступной цене. До этого развития

единственными вычислительными средами, доступными для ИИ, были суперкомпьютеры, стоимость которых была непомерно высокой. При работе на отдельных компьютерах можно создавать только простые модели с длительным временем обработки, которые в любом случае требуют для работы дополнительной поддержки в виде различных графических процессоров. Такой подход может быть приемлем для исследовательской или опытно-конструкторской среды.

Параллельное и распределенное машинное обучение

Если говорить о машинном обучении, язык Python предлагает широкие возможности, и можно с уверенностью сказать, что Python является эталонным языком для этого направления. Существует множество библиотек, предоставляющих алгоритмы машинного обучения, среди которых можно выделить следующие:

- Scikit-learn;
- XGBoost;
- LightGBM;
- TensorFlow.

Кроме того, многие из этих библиотек разработаны с учетом возможности работы в параллельных системах (параллельные вычисления) или даже расширения в средах, состоящих из нескольких машин (распределенные вычисления). Это связано с тем, что машинное обучение требует больших вычислительных ресурсов, поэтому естественно использовать его алгоритмы на мощных системах с возможностью параллельных вычислений.

В частности, для машинного обучения необходимы некоторые операции, где параллельные вычисления играют основополагающую роль. Наиболее важной из них является умножение матриц. В первых главах книги мы уже убедились в том, что правильная параллельная реализация может принести огромную пользу, особенно если для выполнения этих вычислений используются графические процессоры. Умножение матриц является основой алгоритмов линейной регрессии, которые широко используются в машинном обучении. Существенное сокращение времени выполнения подобных вычислений однозначно влияет на общую производительность этапов обучения и на скорость прогнозных расчетов моделей машинного обучения.

Еще одной очень распространенной операцией в машинном обучении является вычисление расстояния между двумя точками. Эта операция широко используется во многих алгоритмах (таких как кластеризация k-средних) и выполняется многократно в рамках одного алгоритма. Благодаря тому, что вычисление отдельного расстояния в последующих итерациях не зависит от других вычислений в той же итерации, эти операции могут выполняться параллельно, что значительно сокращает время обработки.

Кроме того, в контексте операций, которые могут быть обременительными в плане выполнения машинного обучения, существует k-кратная перекрестная проверка (k-fold cross-validation), основанная на интенсивной обработке различных сегментов набора данных. Этот метод используется для проверки модели и основан на исключении из вычислений одного из k сегментов (кратности), на которые разделен набор данных. Все остальные k – 1 сегментов проходят обучение, а затем k-й элемент используется в качестве тестового набора данных. Поскольку мы имеем дело с k сегментами, эта операция выполняется k раз, и в каждом случае из обучения исключается какой-либо иной сегмент. После получения k результатов вычисляется их среднее значение. Разумеется, что при последовательном выполнении алгоритм k-кратной перекрестной проверки отнимет много времени, поскольку одни и те же операции придется повторять k раз. Если же выполнять его в параллельном режиме, когда каждое вычисление, выполняемое на одной из k проверок, выполняется одновременно, это даст значительный выигрыш в плане повышения производительности.

Все эти и другие операции, если они обрабатываются на параллельной вычислительной системе, значительно повышают эффективность машинного обучения и позволяют решать более сложные задачи и обрабатывать более крупные наборы данных, увеличивая прогнозирующую способность создаваемых моделей, а также сокращая время обработки, которое в системе с одной машиной вскоре стало бы непомерно большим.

Машинное обучение с помощью Scikit-learn

В настоящее время Scikit-learn является наиболее используемой библиотекой Python в области машинного обучения и предоставляет пользователям все основные алгоритмы для наиболее часто используемых типов. Это:

- классификация;
- кластеризация;
- регрессия;
- снижение размерности.

Scikit-learn – это хорошо документированная библиотека, и в интернете доступно множество примеров, документации и различных приложений для нее. Поэтому вполне нормально начинать работу с машинным обучением именно с этой библиотеки.

Чтобы ознакомиться с механизмом работы машинного обучения с помощью Scikit-learn, рассмотрим простой пример линейной регрессии. Среди различных примеров этот является очень простым для понимания и поэтому идеально подходит в качестве демонстрационного случая даже для людей, не знакомых с машинным обучением. Эта техника основана на построении очень простой модели системы, поведение которой предположительно является линейным, для прогнозирования результата по заданному значению переменной этой системы.

Для упрощения мы используем в качестве данных для исследования точки, сгенерированные около прямой линии, которые удаляются от правильного линейного тренда на случайное расстояние. Тем самым мы смоделируем данные из процесса, который соответствует линейному поведению, на основе которого мы должны будем получить линейный тренд (пересечение с осью и коэффициент наклона прямой линии). Для этого мы генерируем набор данных из 30 значений по уравнению прямой линии с наклоном `0,3` и коэффициентом `1,0`, к которому добавляем случайный шум, моделирующий отклонение значений от правильного тренда.

```
import numpy as np
x = 50 * np.random.random((30, 1))
y = 0.3 * x + 1.0 + np.random.normal(size=x.shape)
```

После создания набора данных `dataset(x,y)`, в котором `x` будет представлять переменные, а `y` будет представлять метки с результатами, его можно разделить на две части. Одну из них мы будем использовать в качестве обучающего набора, а другую – в качестве тестового набора. Для этого в Scikit-learn есть удобная функция `train_ test_split()`, которая позволяет разделить набор данных в соответствии с процентом, заданным параметром `test_size`. Таким образом, если мы хотим, чтобы 20 % набора данных было предназначено для тестирования, мы напишем следующий код:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

Теперь, когда у нас есть все данные для обучения и проверки модели, нам нужно создать саму модель. Поскольку это линейная регрессия, можно создать конкретную модель, просто обозначив конструктор `LinearRegression()`. После создания модели мы перейдем к этапу обучения, используя метод `fit()` и передавая его в качестве аргумента в обучающий набор.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(x_train, y_train)
```

Как можно видеть, Scikit-learn значительно упрощает реализацию машинного обучения, сводя все к двум строкам кода. После выполнения мы получим модель, которая сможет прогнозировать соответствующие значения `y`, передавая ей значения переменной `x`. Ранее уже упоминалось, что эта модель очень проста и основана на параметризации уравнения прямой линии, в которой два параметра (коэффициент пересечения и коэффициент наклона) постепенно оцениваются на основе этапа обучения. Если необходимо увидеть значения этих двух параметров, следует написать следующий код:

```
print(model.intercept_)
print(model.coef_)
```

В результате будут получены два оцененных параметра:

```
[1.41274643]
[[0.29301879]]
```

Эти значения немного отличаются от реальных: 1 и 3. Теперь перейдем к оценке этой модели посредством тестирования ее прогнозирующих возможностей. Для проведения оценочного тестирования модели используется метод `predict()`, которому в качестве аргумента передается тестовый набор:

```
y_new = model.predict(x_test)
```

Результатом будет список значений `y`, соответствующих `x` и переданным в тестовом наборе. Проверяя разницу между ожидаемыми значениями `y_new` и реальными `y_test`, можно оценить достоверность работы только что созданной модели. Поскольку это числовые значения, описывающие точки около прямой линии на графике в декартовой системе координат, лучшим способом оценки качества этой модели будет их графическое представление. Для этого можно использовать библиотеку `matplotlib`, которая очень полезна для создания графиков, отображаемых в Jupyter Notebook с помощью нескольких строк кода.

```
import matplotlib.pyplot as plt
plt.figure()
plt.scatter(x,y)
plt.scatter(x_test,y_new, c='red')
```

При выполнении ячейки Notebook, содержащей предыдущий код, мы получим в результате график, очень похожий на приведенный на рис. 9.3.

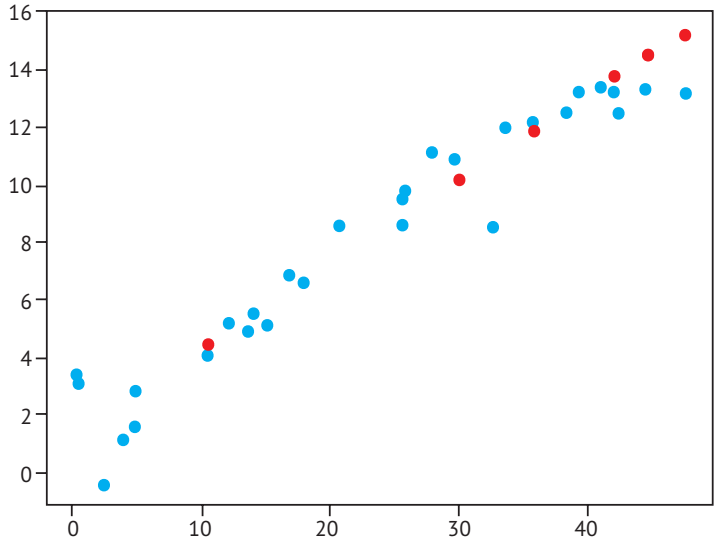


Рис. 9.3. Результаты линейной регрессии
(синие точки из обучающего набора, красные точки из тестового набора)

Для оценки качества используемой модели можно использовать метод `score()`. Этот метод позволяет рассчитать коэффициент детерминации, известный как R^2, как для обучающих, так и для тестовых данных. Максимальное значение оценки составляет 1, а минимальное составляет 0. Чем ближе это значение к 1, тем выше будет качество подгонки модели. Поэтому это значение является отличным параметром для оценки прогнозирующих способностей обученной модели.

```
model.score(x_train, y_train)
0.920774873832153

model.score(x_test, y_test)
0.9011230374736323
```

Масштабирование Scikit-learn с Dask-ML

Подобно тому, что было рассмотрено в предыдущей главе 8, Dask может быть отличным решением для масштабирования приложений машинного обучения в распределенных средах. Как и в случае с Dask DataFrame для библиотеки Pandas и Dask Array для NumPy, для машинного обучения также имеется решение: модуль Dask-ML позволяет использовать библиотеку Scikit-learn в распределенных средах, таких как кластеры.

Этот модуль использует одну из функций параллельных вычислений Scikit-learn: `joblib`. Многие алгоритмы Scikit-learn были оптимизированы для параллельного выполнения при использовании функции `joblib`, которая изначально обеспечивает параллелизм на основе потоков и процессов. В Dask-ML эта возможность используется для распределения вычислений между узлами кластера. Тот же механизм используется для реализации моделей машинного обучения других библиотек, таких как XGboost.

Для установки Dask-ML в виртуальной среде, специально созданной для машинного обучения, следует выполнить следующие действия.

1. Откройте командную оболочку и введите следующую команду:

   ```
   conda install dask-ml -c conda-forge
   conda install joblib
   conda install matplotlib -c conda-forge
   ```

2. По завершении установки откроем запись в Jupyter:

   ```
   from dask.distributed import Client
   client = Client()
   ```

3. Кроме того, Dask-ML, подобно Dask, предоставляет набор методов и функций, соответствующих исходным библиотекам, но способных

работать в распределенных средах. В этом случае функция `train_test_split()` из Scikit-learn будет заменена аналогичной функцией из библиотеки Dask-ML:

```
from dask_ml.model_selection import train_test_splitx_train, x_test,
y_train, y_test = train_test_split(x, y, test_ size=0.2)
```

4. Как и функции Scikit-learn, модели также представлены в своей собственной версии, распространяемой с Dask-ML. Заменим в коде класс `LinearRegression` из Scikit-learn на класс из Dask-ML. На этом этапе методы обучения `fit()` и оценки `predict()` также будут предоставлены Dask-ML и адаптированы для работы в распределенных средах.

```
from dask_ml.linear_model import LinearRegression
model = LinearRegression()
```

Однако прежде чем перейти к этапу обучения, нам необходимо добавить библиотеку `joblib`. Эта библиотека дает возможность библиотекам Python, таким как Scikit-learn, работать на одной машине в параллельном режиме. С помощью `parallel_backend`, предоставляемого этой библиотекой, эту задачу можно выполнить с учетом особенностей Dask. Так мы активируем выполнение кода в режиме параллельных вычислений даже в распределенной среде: еще одно улучшение производительности этой библиотеки.

```
import joblib
from joblib import parallel_backend

with parallel_backend('dask'):
  model.fit(x_train,y_train)
  y_new = model.predict(x_test)
```

5. Теперь, когда мы обучили и оценили модель в параллельном режиме, мы можем проверить результаты графически, как это делали ранее.

```
import matplotlib.pyplot as plt

plt.figure()
plt.scatter(x_train,y_train)
plt.scatter(x_test,y_new, c='red')
```

6. Запустив код, мы получим график, подобный приведенному на рис 9.4.

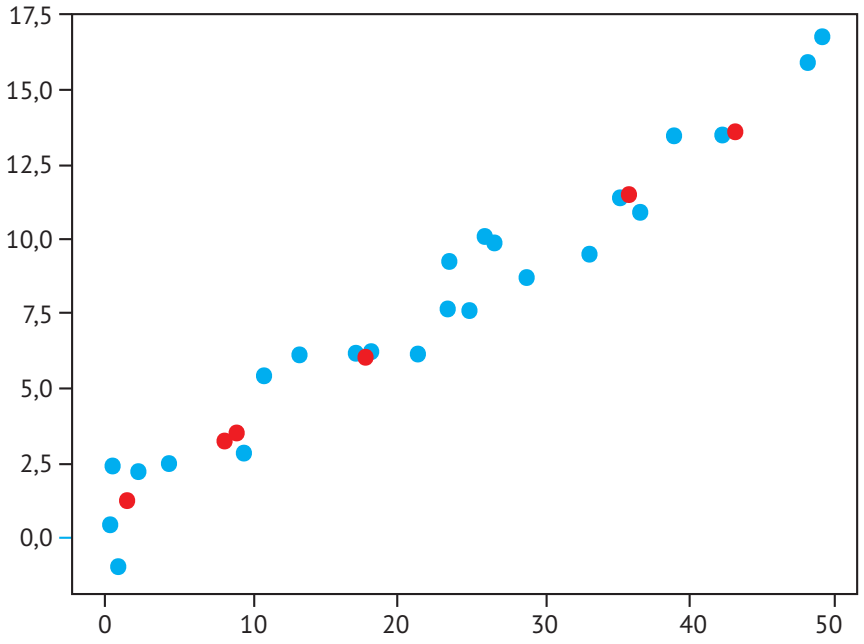


Рис. 9.4. Результаты, полученные моделью с Dask и параллельными вычислениями

Параллельное и распределенное глубокое обучение

В основе глубокого обучения лежат нейронные сети, которые благодаря своей особой архитектуре отлично подходят для обнаружения взаимосвязей и структур, скрытых за данными без предварительной подготовки. Их способность к обучению и анализу, которая в определенной степени имитирует процесс обучения человеческого мозга, позволяет использовать их для работы с данными, не имеющими строго числового характера, такими как изображения и звуки. Именно по этой причине глубокое обучение особенно активно используется в области обработки естественного языка, а также компьютерного зрения.

Искусственные нейронные сети – это сложные структуры, для эффективной работы которых требуются огромные вычислительные ресурсы. Типичные нейронные сети, используемые в глубоком обучении, включают в свои модели миллионы параметров. Кроме того, для обучения эти системы должны использовать большие объемы данных. В результате этот процесс требует значительных вычислительных ресурсов и занимает огромное количество времени при использовании систем с одной машиной – зачастую несколько дней. Кроме того, необходимо учитывать, что многие наборы данных слишком велики, чтобы поместиться в памяти одной машины. К счастью, особая многослойная структура нейронов в нейронной сети идеально подходит для распределенных

систем и поэтому позволяет в полной мере использовать преимущества параллельных вычислений. Модели, используемые для глубокого обучения, могут быть распределены по разным слоям нейронной сети, а их параметры (веса) – по разным нейронам, присутствующим в каждом слое. Таким образом, можно использовать эти особенности для распределения нагрузки и информации используемых моделей, а также выполнения соответствующих вычислений на разных машинах в распределенной системе.

Существует несколько методов распределения и, следовательно, параллелизации вычислений при обучении на кластерах машин. Эти методы можно использовать по отдельности или в сочетании.

- Параллельная обработка:
  - многоядерная обработка;
  - обработка на GPU.
- Распределенное обучение:
  - параллелизм данных;
  - параллелизм моделей.

С помощью этих методов можно преобразовать алгоритмы глубокого обучения в параллельные алгоритмы, где отдельные задачи будут распределены между разными ядрами (многоядерная обработка) или, что еще эффективнее, между разными графическими процессорами, установленными на этих машинах (обработка на GPU). Что касается наборов данных, то, поскольку они, как правило, имеют огромный размер, их можно разделить и затем распределить по разным машинам (параллелизм данных), тем самым снижая перегрузку на память. В свою очередь, даже сами модели глубокого обучения можно разделить на более мелкие фрагменты, присвоив набор параметров для каждой машины (параллелизм моделей). Оба этих метода позволяют значительно сократить время обучения. Однако параллелизм данных зачастую предпочтительнее параллелизма моделей. В последнем случае некоторые операции синхронизации требуют большего времени обработки, поэтому параллелизм данных является более быстрым из них. С другой стороны, параллелизм моделей дает более точные результаты прогнозирования.

PyTorch и TensorFlow

Из множества возможных вариантов, предлагаемых языком Python для работы с глубоким обучением, в последнее время особой популярностью пользуются две платформы с открытым исходным кодом:

- PyTorch;
- TensorFlow.

PyTorch – это относительно новая среда для глубокого обучения. Эта библиотека была разработана исследовательской группой Facebook AI Lab и в 2017 году

стала открытой на GitHub. Ее основной целью является реализация приложений для обработки естественного языка (NLP). Благодаря таким характеристикам, как простота, эффективное использование памяти и в значительной степени нативность, PyTorch быстро завоевала успех, поскольку она генерирует код с более высокой скоростью обработки.

TensorFlow – это также фреймворк с открытым исходным кодом, созданный специально для глубокого обучения. Эта библиотека была разработана Google и впервые была представлена в 2015 году. По сравнению с PyTorch, TensorFlow является гораздо более популярным и широко используемым фреймворком. Это связано не только с тем, что он существует дольше, но и прежде всего с большим количеством доступной документации и широкой поддержкой со стороны других технологий, которые облегчают масштабируемость в распределенных системах и на различных платформах, включая Android. Кроме того, с 2017 года в TensorFlow интегрирован Keras, еще один фреймворк для глубокого обучения, который существует уже несколько лет и располагает обширной документацией по этой дисциплине. В настоящее время TensorFlow является предпочтительным инструментом для многих профессионалов и исследователей, а также располагает множеством уже обученных моделей.

Пример глубокого обучения с PyTorch

Переходя к практической части, рассмотрим одну из этих двух библиотек. Выберем PyTorch для реализации примера глубокого обучения. Мы будем использовать нейронную сеть для анализа набора данных изображений и распознавания их содержания.

Установка PyTorch

Чтобы установить PyTorch с помощью Anaconda, перейдите в **Anaconda Navigator** и на панели **Environments** (Среды) выберите виртуальную среду, предназначенную для глубокого обучения. Если вы еще этого не сделали, вы можете создать новую, нажав кнопку **Create** (Создать), как показано на рис. 9.5.

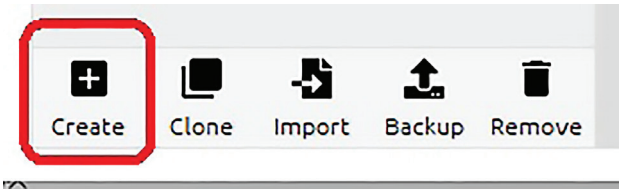


Рис. 9.5. При нажатии кнопки **Create** создается новая виртуальная среда

На этом этапе появится диалоговое окно, в котором необходимо ввести название создаваемой виртуальной среды и версию Python. Выберем версию 3.10.9, как показано на рис. 9.6.

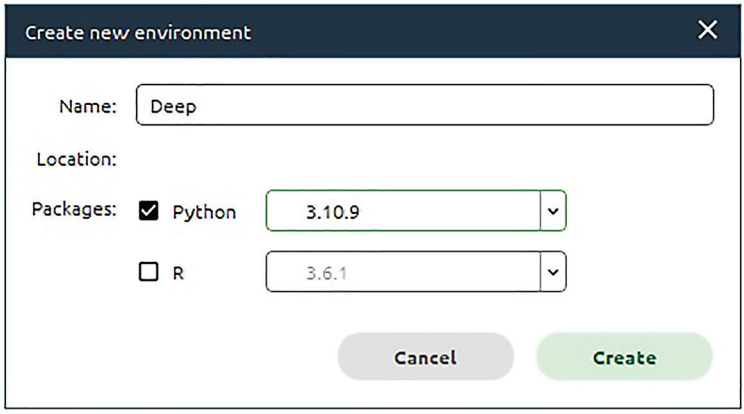


Рис. 9.6. Диалоговое окно для создания новой виртуальной среды

После создания и активации виртуальной среды, предназначенной для глубокого обучения, можно приступить к установке пакетов, необходимых для работы с PyTorch. Для этого посетите веб-сайт PyTorch по адресу *https://pytorch.org/get-started/ locally/* и сделайте соответствующий выбор в зависимости от характеристик вашей рабочей платформы в таблице системных характеристик, отображаемой в центре страницы, как показано на рис. 9.7.

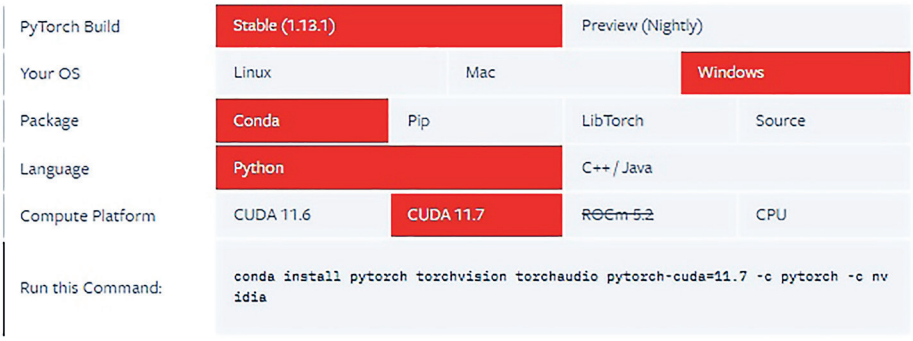


Рис. 9.7. Панель для выбора пакетов для установки с сайта PyTorch

После выбора актуальной конфигурации используемой платформы будет отображаться команда conda со списком всех необходимых пакетов:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch
-c nvidia
```

Далее откройте командную оболочку виртуальной среды и запустите предыдущую команду conda. При запросе на установку пакета введите Y для подтверждения и дождитесь завершения загрузки и установки. На этом этапе мы откроем новый блокнот в Jupyter и протестируем установку с помощью следующих строк:

```
import torch
torch.__version__
```

Как результат, вы должны получить версию PyTorch:

```
'1.13.1'
```

Затем мы проверим взаимодействие с CUDA с помощью следующего кода:

```
torch.cuda.is_available()
True

torch.version.cuda
'11.7'
```

Если вы получите эти результаты без каких-либо сообщений об ошибках, значит, установка этих пакетов прошла успешно, и они без проблем интегрируются с CUDA и драйверами NVIDIA для параллельной обработки данных с помощью графических процессоров.

Пример с набором данных Fashion-MNIST

Для решения подобных задач глубокого обучения рекомендуется брать за основу уже существующие модели и затем вносить в них некоторые изменения. Проведя поиск в Сети, в частности на Kaggle (сайт сообщества Data Scientist), мы смогли найти полезный для нашего случая пример: исследование набора данных Fashion MNIST с помощью PyTorch по адресу *https://www. kaggle.com/code/pankajj/fashion-mnist-with-pytorch-93-accuracy*.

В PyTorch есть несколько предварительно загруженных наборов данных, которые можно использовать для тестирования моделей. В библиотеке вы найдете следующее:

- наборы данных изображений;
- наборы данных текста;
- наборы данных аудио.

Для нашего примера, а именно распознавания объектов на изображениях, можно использовать набор данных Fashion-MNIST. Этот набор данных включает примерно 60 000 небольших (28×28 пикселей) черно-белых (в оттенках серого) изображений одежды, взятых с сайта Zalando. Именно такой набор данных нам и нужен.

Что касается используемой модели, на Kaggle можно найти готовую сверточную (convolutional) нейронную сеть `FashionCNN()`, которую можно использовать в нашем примере. Итак, откроем новый блокнот в Jupyter и импортируем все необходимые модули в первую ячейку.

```
import torch
import torch.nn as nn
```

```
from torch.autograd import Variable

import torchvision
from torchvision import datasets
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader

import datetime
```

На этом этапе мы приступаем к извлечению данных из набора данных и распределению их по обучающему и тестовому наборам, необходимым для этапа обучения и тестирования.

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Для ознакомления с типом изображений, использованных как в обучающем, так и в тестовом наборах, визуализируем первые девять изображений обучающего набора.

```
import matplotlib.pyplot as plt
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
```

```
        img, label = training_data[sample_idx]
        figure.add_subplot(rows, cols, i)
        plt.title(labels_map[label])
        plt.axis("off")
        plt.imshow(img.squeeze(), cmap="gray")
    plt.show()
```

Если выполнить приведенный выше код, мы получим таблицу из девяти черно-белых изображений различных предметов одежды, каждый из которых помечен одним из десяти предметов одежды, описанных в `labels_map`, как показано на рис. 9.8.

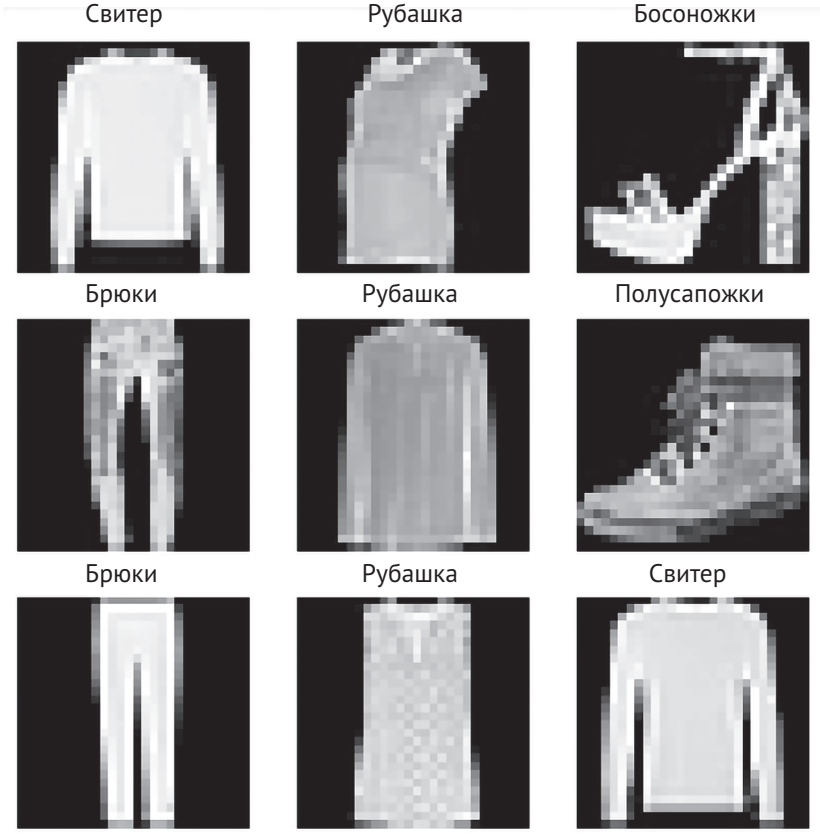


Рис. 9.8. Некоторые изображения одежды в обучающем наборе

Теперь в следующей ячейке Jupyter Notebook запишем код, связанный со сверточной нейронной сетью FashionCNN.

```
class FashionCNN(nn.Module):

    def  init (self):
        super(FashionCNN, self). init ()
```

```
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.fc1 = nn.Linear(in_features=64*6*6, out_features=600)
        self.drop = nn.Dropout2d(0.25)
        self.fc2 = nn.Linear(in_features=600, out_features=120)
        self.fc3 = nn.Linear(in_features=120, out_features=10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.drop(out)
        out = self.fc2(out)
        out = self.fc3(out)
        return out
```

Как только наборы данных и модель нейронной сети будут готовы, можно переходить к реализации этапов обучения модели и одновременному расчету точности (accuracy) и потерь (loss) модели на этапе обучения. Эти значения понадобятся позже для оценки качества полученной модели.

```
def train(bs, lr):

    batch_size = bs
    learning_rate = lr
    train_loader = torch.utils.data.DataLoader(training_data,batch_size)
    test_loader = torch.utils.data.DataLoader(test_data,batch_size)

    #device = torch.device('cuda:0')
    device = torch.device('cpu')
    num_epochs = 5
    count = 0
    model = FashionCNN()
    model.to(device)
```

```
    error = nn.CrossEntropyLoss()
    learning_rate = 0.001
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    training_start_time = datetime.datetime.now()
    results = []

    for epoch in range(num_epochs):
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            train = Variable(images.view(100, 1, 28, 28))
            labels = Variable(labels) outputs = model(train)
            loss = error(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            count += 1

        # Тестирование модели
            if not (count % 50): # То же самое что и "if count % 50 == 0"
                total = 0
                correct = 0
                for images, labels in test_loader:
                    images, labels = images.to(device), labels.to(device)
                    test = Variable(images.view(100, 1, 28, 28))
                    outputs = model(test)
                    predictions = torch.max(outputs, 1)[1].to(device)
                    correct += (predictions == labels).sum()
                    total += len(labels)

               accuracy = correct * 100 / total
               new_results = {
                  "batch_size": batch_size,
                  "loss": loss.data.item(),
                  "iteration": count,
                  "accuracy": accuracy.item(),
                  "elapsed_time_sec": (datetime.datetime.now() - training_
start_time).total_seconds(),
                }
                results.append(new_results)
    return results
```

Теперь мы выполняем процесс обучения нейронной сети, вызывая только что определенную функцию и передавая `batch_size` и `learning_rate` в качестве аргументов:

```
# batch_size = 100, learning_rate = 0.001
results = train(100, 0.001)
```

Выполнение предыдущего кода займет некоторое время. Примерно через минуту модель завершит этап обучения. На этом этапе мы рассчитали и собрали некоторые параметры, которые учитывают качество этапа обучения, такие как потери и точность. Эти значения, которые мы собрали в переменной `results`, передаем в Pandas DataFrame, чтобы их было проще просматривать.

```
import pandas as pd

results_df = pd.DataFrame.from_dict(results)
results_df.head(10)
```

Выполнив приведенный выше код, мы отобразим первые десять строк DataFrame с помощью метода `head()`, как показано на рис. 9.9.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
0	100	0,503404	50	79,540001	10,698972
1	100	0,563143	100	81,220001	20,112245
2	100	0,452134	150	84,059998	29,749219
3	100	0,395133	200	85,709999	39,393635
4	100	0,441687	250	85,050003	48,891572
5	100	0,422865	300	84,529999	58,477025
6	100	0,347971	350	86,680000	68,099902
7	100	0,439388	400	86,589996	77,511428
8	100	0,620000	450	84,849998	87,049638
9	100	0,501260	500	87,669998	96,446948

Рис. 9.9. Первые 10 строк DataFrame с результатами

Как видно из DataFrame, мы собрали значения потерь и точности, а также времени, затраченного на выполнение каждых 50 итераций, которое в данном случае составляет около 10 с. Чтобы увидеть время, затраченное на весь процесс обучения (3000 итераций), можно просмотреть последние строки DataFrame с помощью метода `tail()`.

```
results_df.tail()
```

Выполнив этот метод, мы получаем таблицу с последними пятью результатами, которые соответствуют последним пяти шагам этапа обучения, как показано на рис. 9.10.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
55	100	0,250286	2800	90,449997	533,636908
56	100	0,484450	2850	89,900002	543,004209
57	100	0,292580	2900	90,570000	552,542618
58	100	0,195649	2950	90,129997	562,113212
59	100	0,211632	3000	89,910004	571,689600

Рис. 9.10. Последние пять строк DataFrame с результатами

После размещения всех данных, собранных из результатов, в Pandas DataFrame мы также пользуемся простотой отображения данных в этих объектах с помощью `matplotlib`. Давайте посмотрим на динамику потерь на протяжении фазы обучения, для чего в следующей ячейке Notebook напишем следующий код:

```
import matplotlib.pyplot as plt

plt.plot(results_df['iteration'], results_df['loss'])
plt.show()
```

При выполнении кода получается линейный график, на котором представлена динамика потерь на протяжении всей фазы обучения (3000 итераций), как показано на рис. 9.11.

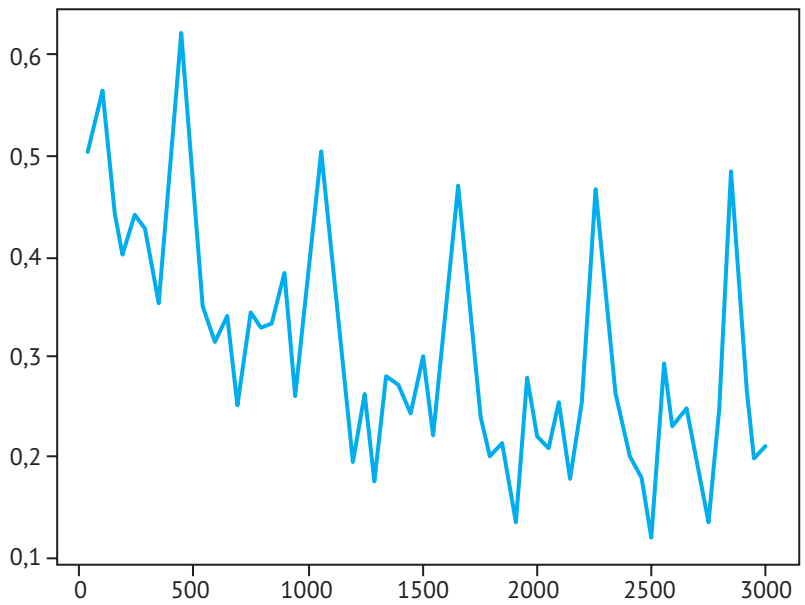


Рис. 9.11. Динамика потерь на этапе обучения

Как видно из данных графика, динамика потерь очень изменчива, и для каждой эпохи наблюдается схожая тенденция с высокими и низкими значениями. При этом для каждой эпохи наблюдается постепенное улучшение показателя потерь, который изначально составлял 0,5, но после пяти эпох обучения стабилизировался на уровне 0,2.

То же самое можно сделать с показателем точности. Для построения графика тенденции точности можно написать следующий код:

```
plt.plot(results_df['iteration'], results_df['accuracy'])
plt.show()
```

Выполнив приведенный выше код, мы получим график, аналогичный представленному на рис. 9.12.

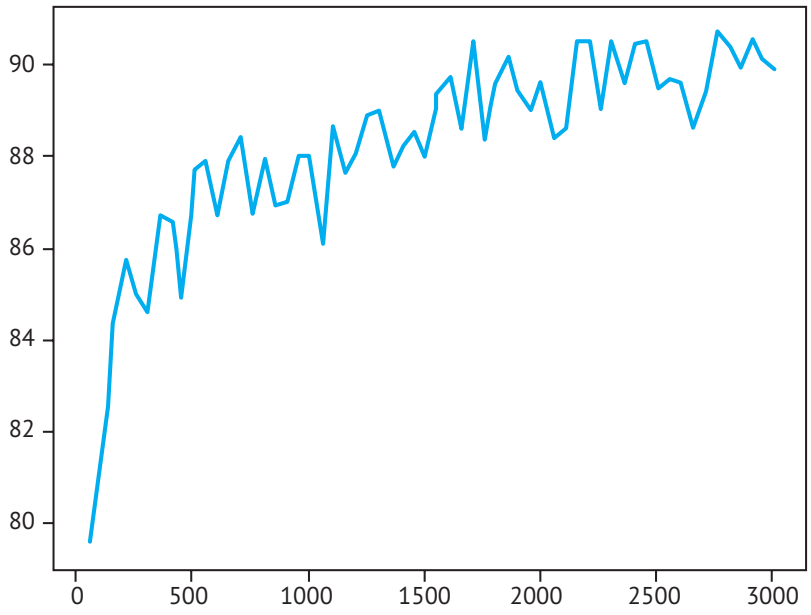


Рис. 9.12. Тенденция изменения точности обучения нейронной сети

На этом графике также можно увидеть чередующиеся пики высокой и низкой точности, что свидетельствует о ее значительной изменчивости от одной итерации к другой. Однако общая тенденция точности всегда улучшается, увеличиваясь с начального значения 80 % до значения около 90 % в конце фазы обучения.

Еще один интересный график может отражать время, затраченное на выполнение итераций в течение всего процесса обучения.

```
plt.plot(results_df['iteration'], results_df['elapsed_time_sec'])
plt.show()
```

В результате выполнения этого кода мы получим график как на рис. 9.13.

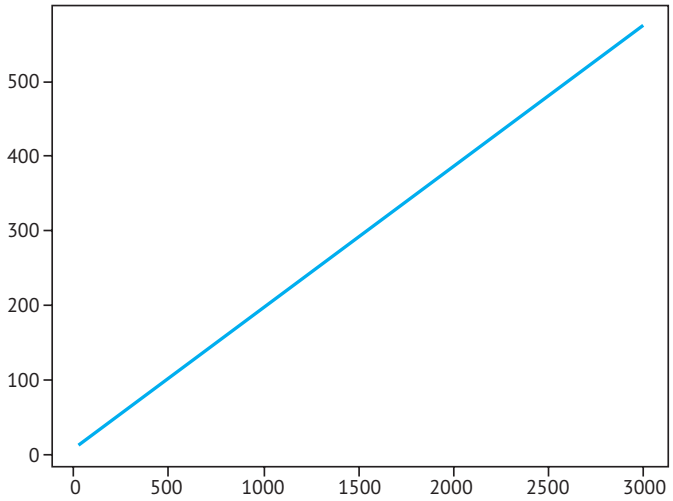


Рис. 9.13. Динамика затрат времени на обучение нейронной сети

Как видно из графика, время выполнения задачи остается неизменным на протяжении всего процесса обучения.

Пример глубокого обучения с использованием PyTorch и GPU

Теперь, когда у нас есть полноценный пример обучения нейронной сети на наборе изображений, выполненный на одной машине, интересно было бы посмотреть, каким образом можно использовать доступные нам технологии параллельных вычислений. Например, с помощью библиотеки PyTorch можно выполнить вычисления с использованием графической карты нашего компьютера. Исходя из уже написанного кода, мы можем внести в него некоторые изменения и сравнить результаты его выполнения.

В функции обучения нейронной сети, которую мы определили как `train()`, мы изменим вычислительное устройство, заменив `cpu` на `cuda:0`. Теперь после повторного запуска всего кода мы сможем получить поддержку параллелизма с помощью графической карты NVIDIA благодаря интеграции CUDA.

```
def train(bs, lr):

    batch_size = bs
    learning_rate = lr
    train_loader = torch.utils.data.DataLoader(training_data,batch_size)
    test_loader = torch.utils.data.DataLoader(test_data,batch_size)

    device = torch.device('cuda:0')

    num_epochs = 5
```

```
count = 0
model = MyCNN()
. . .
```

Очищаем Jupyter Notebook, повторно запустив ядро и удалив все полученные результаты. Для этого выберите в меню **Kernel** (Ядро) **> Restart & Clear Output** (Перезапустить и очистить результаты), как показано на рис. 9.14, и дождитесь перезапуска системы Jupyter.

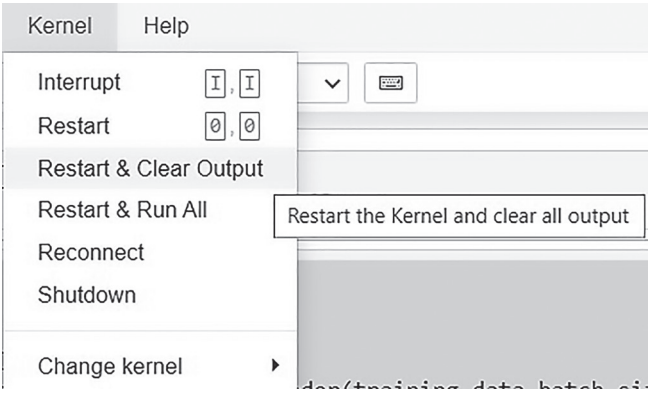


Рис. 9.14. Перезапуск Jupyter Notebook

После сброса Notebook и изменения устройства в коде мы повторно выполним все ячейки от начала до конца и посмотрим на разницу в полученных результатах. Проверим время выполнения из DataFrame, содержащего параметры вычислений каждой итерации, как показано на рис. 9.15.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
0	100	0,549456	50	78,680000	4,770011
1	100	0,449008	100	81,519997	7,666718
2	100	0,513790	150	84,449997	10,254896
3	100	0,434090	200	85,599998	12,682994
4	100	0,424354	250	86,070000	15,455611
5	100	0,348596	300	86,839996	17,886880
6	100	0,338119	350	87,209999	20,646353
7	100	0,411364	400	84,680000	23,106270
8	100	0,621500	450	86,419998	25,801914
9	100	0,430898	500	87,680000	28,419534

Рис. 9.15. DataFrame с результатами, полученными при добавлении GPU

Как видно на приведенном выше рис. 9.15 и на следующем рис. 9.16, время сократилось более чем вдвое. Время выполнения около 50 итераций, которое раньше составляло 10 с, при использовании GPU сократилось до менее чем четырех секунд.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
55	100	0,259191	2800	89,930000	147,375217
56	100	0,433528	2850	89,629997	150,086596
57	100	0,217143	2900	90,930000	152,501081
58	100	0,207694	2950	89,529999	155,274581
59	100	0,198218	3000	90,290001	157,656414

Рис. 9.16. Последние строки DataFrame с результатами, полученными после добавления GPU

Давайте также посмотрим на потери и точность на двух графиках, показанных на рис. 9.17.

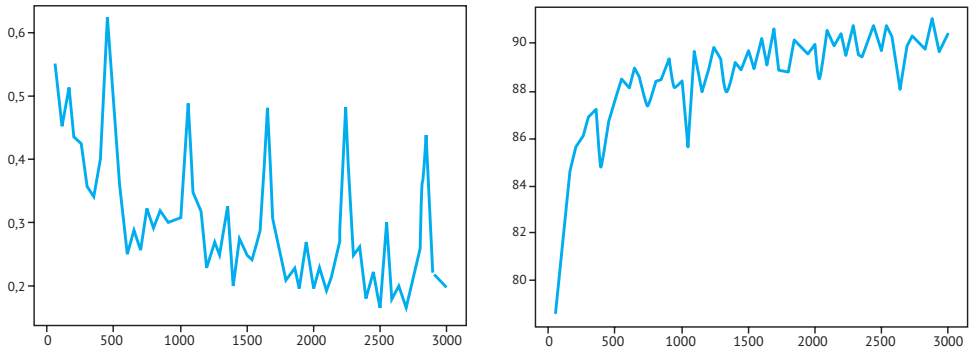


Рис. 9.17. Динамика потерь и точности во время обучения с использованием GPU

В данном случае добавление параллельных вычислений с помощью графической карты не оказывает никакого влияния (ни улучшения, ни ухудшения) на возможности обучения нейронной сети. Просто обучение происходит гораздо быстрее.

Масштабирование PyTorch с помощью Dask

До сих пор мы рассматривали пример глубокого обучения и поведение нейронной сети в системе с одним компьютером, с GPU или без. Теперь давайте масштабируем тот же пример в распределенную систему на базе компьютерного кластера. Как и в предыдущей главе, мы воспользуемся услугой облачного провайдера SaturnCloud.io, который предоставляет бесплатный доступ к кла-

стеру компьютеров на 30-часовой пробный период. Этого времени более чем достаточно для проведения наших тестов.

Для масштабирования нашего кода на основе PyTorch в распределенной системе мы вновь воспользуемся Dask и предоставляемыми им технологиями. Как было рассмотрено в предыдущих главах, выполнить этот шаг довольно просто, внеся лишь минимальные изменения в только что использованный код. Далее Dask самостоятельно выполнит необходимые преобразования во время выполнения.

Мы переходим на SaturnCloud.io и, войдя в систему с помощью учетной записи, активируем кластер машин с помощью уже подготовленного шаблона, созданного специально для использования PyTorch. Перейдите в раздел **Resources** (Ресурсы) и нажмите кнопку **New Resource from a Template** (Новый ресурс из шаблона), как показано на рис. 9.18.

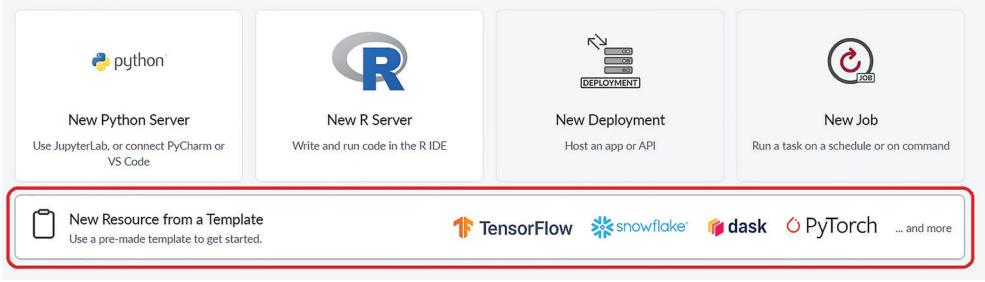


Рис. 9.18. Выбор возможных типов ресурсов, доступных на SaturnCloud.io

При нажатии кнопки появится ряд возможных вариантов с множеством шаблонов, каждый из которых предназначен для конкретного приложения, включая PyTorch. Нажмите соответствующую кнопку, чтобы активировать кластер, уже подготовленный для PyTorch со всеми необходимыми библиотеками, как на рис. 9.19.

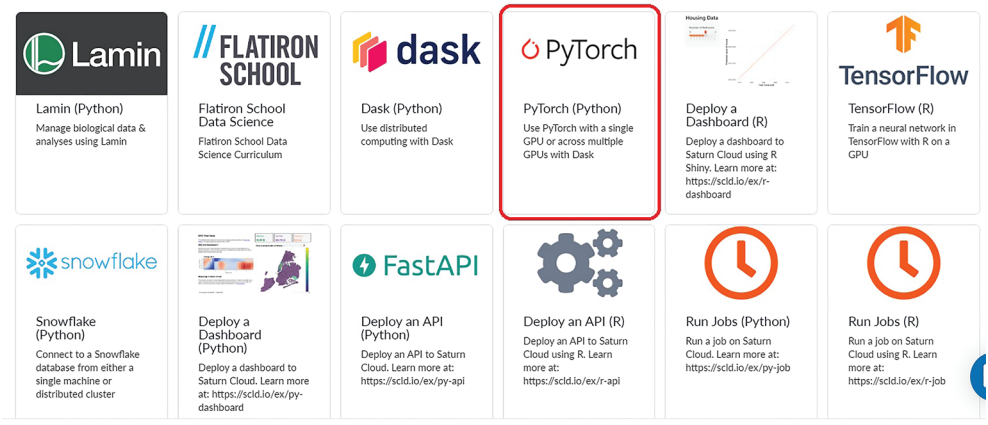


Рис. 9.19. Выбор из различных предустановленных шаблонов кластеров на SaturnCloud.io

Сразу после этого появится диалоговое окно, в котором нужно ввести имя создаваемого ресурса. Вы можете оставить имя по умолчанию, `example-pytorch`, или ввести новое по своему усмотрению, как показано на рис. 9.20.

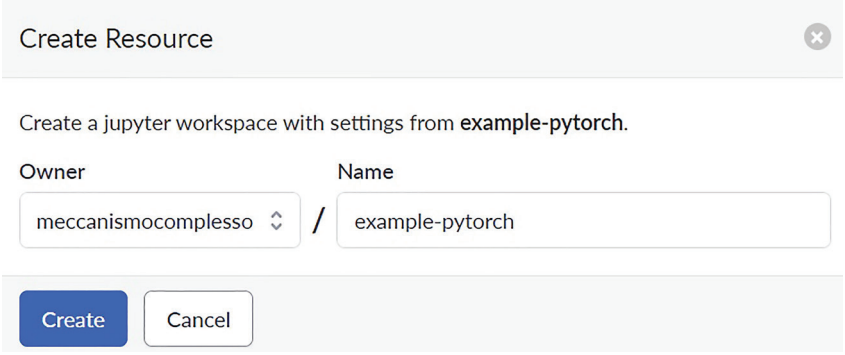


Рис. 9.20. Выбор имени ресурса, который будет создан на SaturnCloud.io

После завершения процесса создания ресурса все входящие в него машины начнут активироваться, включая сервер Jupyter, на который вы будете вставлять код, и четыре другие машины, в том числе три рабочих процесса (кластер Dask) и планировщик. Если все пройдет правильно, в конце вы сможете найти вновь созданный ресурс среди других ресурсов с зелеными индикаторами, обозначающими рабочий статус running, как показано на рис. 9.21.

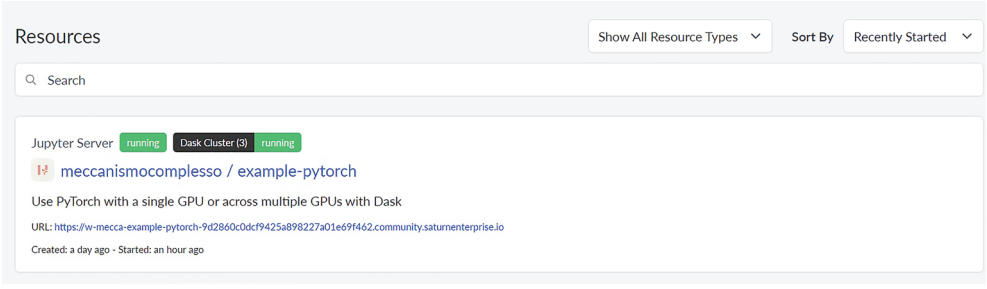


Рис. 9.21. Название вновь созданного ресурса Dask Cluster на SaturnCloud.io

Щелкните по нему для доступа к ресурсу. На странице вы увидите две панели: одна посвящена серверу Jupyter, а другая – кластеру Dask. В нижней части панели **Server Jupyter** выберите кнопку **Jupyter Notebook**, как показано на рис. 9.22, и щелкните по ней для доступа к приложению.

После открытия Jupyter создайте новый блокнот и начните вводить код, который мы использовали ранее, но с некоторыми изменениями. Что касается импорта различных необходимых модулей и библиотек, здесь все остается без изменений.

```
import torch
import torch.nn as nn
from torch.autograd import Variable
```

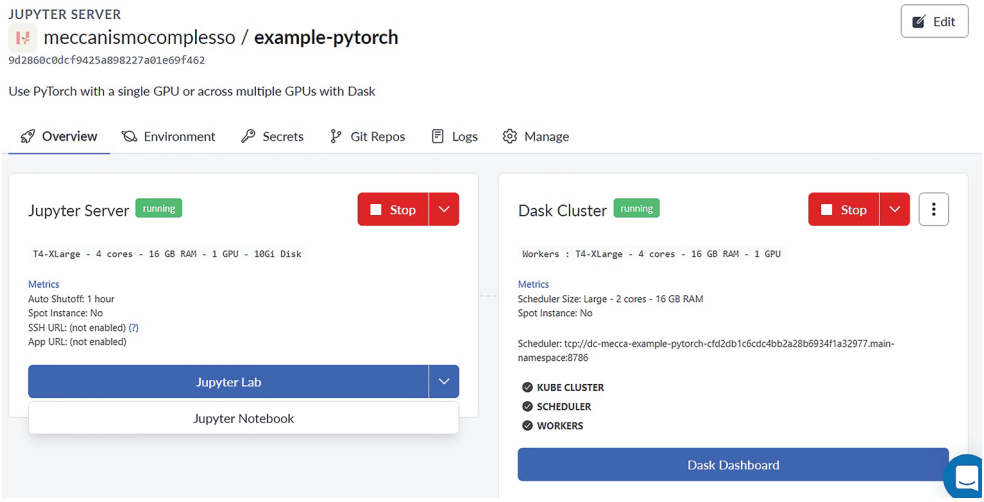


Рис. 9.22. Панель управления вновь созданного ресурса на SaturnCloud.io

```
import torchvision
from torchvision import datasets
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader

import datetime
```

Но для импорта Dask и использования этого кода в распределенной системе, такой как SaturnCloud, необходимо добавить новые библиотеки. Далее мы импортируем несколько проприетарных библиотек, которые позволят вам интегрировать Dask в кластер SaturnCloud.

```
import dask
from dask_saturn import SaturnCluster
from dask.distributed import Client
from distributed.worker import logger
```

Теперь в следующей ячейке блокнота мы сигнализируем Dask о наличии кластера и количестве присутствующих рабочих процессов (workers).

```
n_workers = 3
cluster = SaturnCluster(n_workers=n_workers) client = Client(cluster)
client.wait_for_workers(n_workers)
```

Запустив предыдущую ячейку, вы получите следующий вывод, который сообщит о готовности кластера к запуску кода, распределенного через Dask:

```
INFO:dask-saturn:Cluster is ready
INFO:dask-saturn:Registering default plugins
INFO:dask-saturn:Success!
```

После этого следует вставить код, относящийся к импорту данных для обучения и тестирования нейронной сети. Этот фрагмент кода практически не изменился.

```
training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Выполнив эту часть кода, вы увидите, как в выводе начинается загрузка данных, как показано на рис. 9.23.



Рис. 9.23. Загрузка данных на сервер Jupyter на SaturnCloud.io

Класс, описывающий нейронную сеть CNN, также остается неизменным.

```
class FashionCNN(nn.Module):

    def  init (self):
        super(FashionCNN, self). init ()

        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3),
```

```
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.fc1 = nn.Linear(in_features=64*6*6, out_features=600)
        self.drop = nn.Dropout2d(0.25)
        self.fc2 = nn.Linear(in_features=600, out_features=120)
        self.fc3 = nn.Linear(in_features=120, out_features=10)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.view(out.size(0), -1)
        out = self.fc1(out)
        out = self.drop(out)
        out = self.fc2(out)
        out = self.fc3(out)
        return out
```

Теперь нам необходимо вставить код обучения нейронной сети, который мы поместили в функцию `train()`. Выполнение этого шага значительно упрощает интеграцию этого кода с Dask. В библиотеке Dask есть декоратор `@dask.delayed`, который позволяет распределить код, заключенный в связанную с ним функцию, по разным задачам, которые будут назначены нескольким рабочим процессам для параллельного выполнения. Все это прозрачно для разработчика, чьей единственный заботой будет добавление декоратора.

```
@dask.delayed
def train(bs, lr):

    batch_size = bs
    learning_rate = lr
    train_loader = torch.utils.data.DataLoader(training_data,batch_size)
    test_loader = torch.utils.data.DataLoader(test_data,batch_size)

    device = torch.device('cuda:0')
    num_epochs = 5
    count = 0
    model = FashionCNN() model.to(device)
    error = nn.CrossEntropyLoss()
    learning_rate = 0.001
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    training_start_time = datetime.datetime.now()
    results = []

    for epoch in range(num_epochs):
```

```
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            train = Variable(images.view(100, 1, 28, 28))
            labels = Variable(labels)
            outputs = model(train)
            loss = error(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            count += 1

        # Тестирование модели
            if not (count % 50):
                total = 0
                correct = 0
                for images, labels in test_loader:
                    images, labels = images.to(device), labels.to(device)
                    test = Variable(images.view(100, 1, 28, 28))
                    outputs = model(test)
                    predictions = torch.max(outputs, 1)[1].to(device)
                    correct += (predictions == labels).sum()
                    total += len(labels)

                accuracy = correct * 100 / total
                new_results = {
                     "batch_size": batch_size,
                     "loss": loss.data.item(),
                     "iteration": count,
                     "accuracy": accuracy.item(),
                  "elapsed_time_sec": (datetime.datetime.now() - training_
start_time).total_seconds(),
                }
                results.append(new_results)

    return results
```

До сих пор использовался код, практически идентичный используемому в случае одной машины, за исключением некоторых дополнений. Теперь начинается другая часть, характерная для особенностей поведения Dask. В следующей ячейке мы запускаем выполнение обучения с помощью функции отображения `client.map()`.

```
experiment = [(100, 0.001)]
train_future = client.map(train, experiment)
```

При выполнении предыдущей ячейки мы увидим, что никакого выполнения не произойдет, а вывод будет немедленным. Это происходит потому, что

Dask является ленивым, и вызов функции не приводит к какому-либо выполнению. Параллельное выполнение кода произойдет только при вызове `client.compute()`.

```
futures_gathered = client.gather(train_future)
futures_computed = client.compute(futures_gathered)
results = [x.result() for x in futures_computed]
```

На этот раз при выполнении предыдущей ячейки мы получим длительное время выполнения, и результаты для каждой итерации будут собраны в results, по-другому, но в той же форме, что и в предыдущих примерах. После завершения выполнения мы можем посмотреть результаты.

```
import pandas as pd

results_concatenated = [item for sublist in results for item in sublist]
results_df = pd.DataFrame.from_dict(results_concatenated)
results_df.head(10)
```

Вы получите результат как на рис. 9.24.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
0	100	1,008428	50	61,660000	2,228519
1	100	1,139894	100	61,430000	3,704949
2	100	0,940989	150	65,250000	5,168618
3	100	0,805514	200	66,369995	6,641557
4	100	0,796870	250	65,830002	8,113434
5	100	0,962739	300	66,309998	9,618787
6	100	0,915788	350	64,669998	11,094542
7	100	1,013723	400	66,220001	12,570212
8	100	1,099278	450	65,309998	14,060075
9	100	0,881033	500	65,409996	15,532078

Рис. 9.24. DataFrame с результатами обучения

То же самое для последних пяти значений DataFrame.

```
results_df.tail()
```

Вы получите финальную часть DataFrame, см. рис. 9.25.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
55	100	0,721782	2800	69,250000	83,680797
56	100	0,987083	2850	69,549995	85,286323
57	100	0,778295	2900	70,400002	86,749804
58	100	0,662645	2950	68,739998	88,234482
59	100	0,748363	3000	70,479996	89,732871

Рис. 9.25. Последние пять строк DataFrame с результатами обучения

Как видно из рис. 9.24 и 9.25, выполнение 3000 итераций было даже быстрее, чем в случае параллельного выполнения с помощью GPU на одной машине. Время выполнения снова сократилось вдвое, в среднем до 1,5 с на каждые 50 итераций. Это примерно в два раза меньше, чем в предыдущем случае на одной машине с GPU (около 3 с).

```
import matplotlib.pyplot as plt

plt.plot(results_df['iteration'], results_df['loss'])
plt.show()
```

Давайте рассмотрим значения потерь и точности во время итераций на графиках, показанных на рис. 9.26.

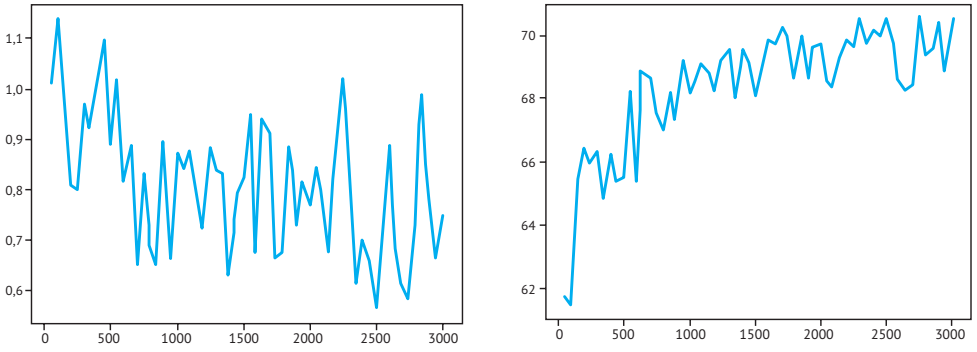


Рис. 9.26. Тенденции потерь и точности, полученные с помощью Dask на SaturnCloud.io

Как можно видеть, на этот раз система ухудшила свои показатели, и теперь потери и точность не так хороши, как в случае с одиночной машиной. Это означает, что, несмотря на повышение производительности при выполнении кода

глубокого обучения в распределенной системе, качество обучения используемой нейронной сети не сохранилось. Это связано с тем, что мы, скорее всего, интегрировали существующий код в Dask в распределенной системе с минимальными изменениями и, следовательно, пренебрегли некоторыми механизмами масштабирования, что привело к потере качества обучения нейронной сети.

Давайте более подробно рассмотрим возможные причины упомянутых выше проблем. Для этого проанализируйте только что использованный код на отдельном компьютере. В числе многих других функций Dask предоставляет возможность выполнять распределенный код даже на архитектурах с одним компьютером.

Для этого достаточно изменить настройки Dask Client. Откройте на своем ПК Jupyter Notebook в виртуальной среде Anaconda, предназначенной для PyTorch, и скопируйте в него весь предыдущий код. Затем измените ячейку, относящуюся к Dask Client, следующим образом:

```
import dask
#from dask_saturn import SaturnCluster
from dask.distributed import Client
#from distributed.worker import logger
#n_workers = 3
#cluster = SaturnCluster(n_workers=n_workers)
#client = Client(cluster)
#client.wait_for_workers(n_workers)

client = Client()
```

Запустив все ячейки, посмотрим на полученные результаты на рис. 9.27.

	batch_size	loss	iteration	accuracy	elapsed_time_sec
0	100	0,446040	50	80,779999	4,762428
1	100	0,545907	100	82,389999	8,147475
2	100	0,513924	150	85,379997	11,207155
3	100	0,406646	200	86,110001	14,164868
4	100	0,388509	250	84,759995	16,760471
5	100	0,372058	300	86,860001	19,718247
6	100	0,360087	350	86,970001	22,391748
7	100	0,461946	400	83,979996	25,316237
8	100	0,668416	450	84,930000	28,051320
9	100	0,486805	500	87,669998	30,677633

	batch_size	loss	iteration	accuracy	elapsed_time_sec
55	100	0,294967	2800	90,099998	158,686704
56	100	0,458460	2850	89,939995	161,465027
57	100	0,235022	2900	90,409996	164,436711
58	100	0,153714	2950	90,029999	167,063556
59	100	0,156465	3000	90,570000	169,751957

Рис. 9.27. DataFrame с результатами интеграции с Dask, но на отдельном компьютере

Точно так же тенденции потерь и точности будут представлены на графиках, сгенерированных `matplotlib`, как показано на рис. 9.28.

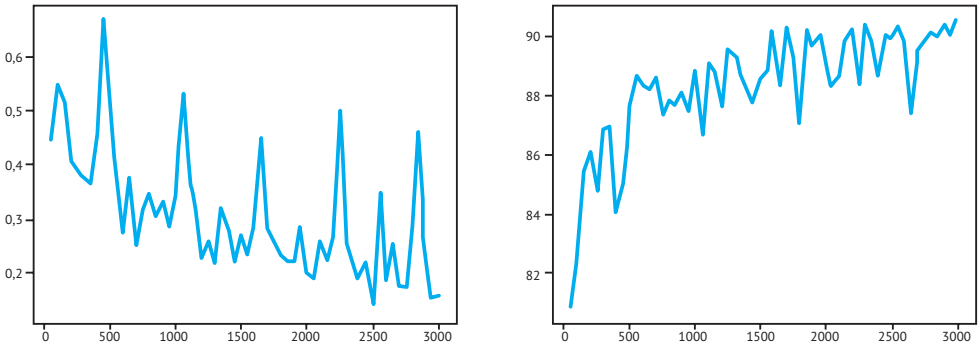


Рис. 9.28. Графики тенденций потерь и точности обучения с Dask, но на отдельном компьютере

Как можно видеть в данном случае, потери и точность вернулись к правильной динамике и приблизились к тем же значениям, что и в предыдущих примерах на отдельной машине. Таким образом, скорее всего, проблема заключается не в интеграции Dask в код, а в распределении задач между различными задачами или даже в распределении данных, которые влияют на механизмы обучения используемой нейронной сети.

Соответственно, требуется более точный анализ используемого кода и внесение необходимых изменений. Очевидно, что масштабирование сложного кода, такого как Deep Learning, на распределенных системах еще не является простым процессом, как это может быть в случае Pandas или Scikit-learn с их моделями машинного обучения. Однако существуют и другие подходы к интеграции моделей PyTorch с Dask в распределенной системе, например путем обертывания моделями, которые в большей степени интегрированы с фреймворком Dask, что, вероятно, будет обеспечивать более эффективную масштабируемость. Однако такие аналитические исследования выходят за рамки данной книги, так что всем заинтересованным предлагается поискать инновационные решения этой проблемы самостоятельно.

С другой стороны, несмотря на низкую эффективность обучения, обусловленную механизмами используемого кода, становится очевидным, что использование распределенной системы ведет к значительному увеличению скорости вычислений, что значительно ускоряет этапы обучения нейронных сетей.

Заключение

В последнее десятилетие искусственный интеллект (ИИ) получил мощный импульс развития в мире информационных технологий, особенно в области разработки и изучения самообучающихся систем. Именно с этой целью была разработана дисциплина «машинное обучение», которая настолько глубоко укоренилась в сфере искусственного интеллекта, что практически стала синонимом этого концепта. Машинное обучение наилучшим образом пред-

ставлено в языке Python, который предоставляет исследователям и разработчикам мощные библиотеки со всеми необходимыми инструментами для этой дисциплины. Используемые модели, объемы данных и алгоритмы обучения становятся все более сложными, и переход к параллельным вычислениям стал необходимым условием.

Что касается глубокого обучения, оно является самым последним рубежом в области применения распределенных систем. Искусственные нейронные сети и соответствующие системы обучения представляют собой достаточно сложные структуры, потребляющие все больше вычислительных ресурсов. Этот вопрос настолько сложен, что результаты зачастую непредсказуемы и требуют дальнейшего изучения. Тем не менее глубокое обучение является чрезвычайно перспективной дисциплиной, в которой еще многое предстоит изучить.

В следующей главе мы завершим тему параллельных вычислений обзором перспективных тенденций и технологических инноваций, которые будут способствовать значительному развитию описанных в книге концепций. Машинное обучение и глубокое обучение, о которых идет речь в этой главе, лишь авангард того, что нас ждет в ближайшем будущем. Новые технологии принесут с собой новые дисциплины, и потребность в разработчиках параллельного программирования, которые будут нести на себе бремя необходимости идти в ногу со временем, станет все более острой.

Список литературы

- *https://www.oracle.com/it/artificial-intelligence/what-is-ai/*.
- *https://www.meccanismocomplesso.org/deep-learning/*.
- *https://medium.com/@gaurav2proud/what-is-the-need-of-parallel-processing-for-machine-learning-in-real-time-7bfc9b66e40c*.
- *https://web.stanford.edu/~rezab/classes/cme323/S16/projects_reports/ hedge_usmani.pdf*.
- *https://www.alteryx.com/glossary/supervised-vs-unsupervised-learning*.
- *https://analyticsindiamag.com/a-guide-to-parallel-and-distributed-deep-learning-for-beginners/*.
- *https://www.e2enetworks.com/blog/demystifying-parallel-and-distributed-deep-learning*.
- *https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article*.
- *https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-frameworks*.
- *https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-algorithm*.
- *https://pytorch.org/docs/stable/notes/multiprocessing. html#asynchronous-multiprocess-training-e-g-hogwild*.
- *https://pytorch.org/tutorials/intermediate/ddp_tutorial.html*.
- *https://www.tensorflow.org/guide/distributed_training*.

Глава 10

Приложения параллельных вычислений

В предыдущей 9 главе мы подробно рассмотрели процесс развития искусственного интеллекта в контексте параллельных вычислений. Успех этого подхода и стремление создать интеллектуальную систему для совместной работы с человеком способствуют стремительному развитию этой отрасли. Этому способствует и растущий технологический потенциал, на котором построены эти системы.

В этой главе вы узнаете о следующих технологических разработках, в которых будут использоваться параллельные вычисления и искусственный интеллект. Эти технологические инновации повлекут за собой не только эволюцию процессов и концепций, обсуждаемых в этой книге, но также будут стимулировать создание новых концепций и парадигм, способных идти в ногу с технологическим прогрессом. Такой прогресс имеет отношение не только к строго академической и профессиональной деятельности, но также постепенно становится неотъемлемой частью повседневной жизни и затрагивает наши бытовые привычки и устоявшиеся традиции. Параллельные вычисления играют все более важную роль и в конечном итоге станут стандартом обработки данных. Работающие в этой области профессионалы станут специалистами будущего, перед которыми открываются многочисленные возможности для интересной и перспективной работы.

Структура

В этой главе мы обсудим следующие темы:

- массивно-параллельный искусственный интеллект;
- пограничные вычисления;
- киберфизические системы;
- кибербезопасность;
- экзафлопсные вычисления;
- квантовые вычисления;
- новые профессиональные возможности;
- достижения в области библиотек Python.

Массивно-параллельный искусственный интеллект

Среди новых разработок в области искусственного интеллекта наиболее близкой к реализации является концепция массивно-параллельного искусственного интеллекта (Massively Parallel Artificial Intelligence), которая, как следует из названия, будет основана на постоянно растущей доступности массивно-параллельных систем, см. рис. 10.1.

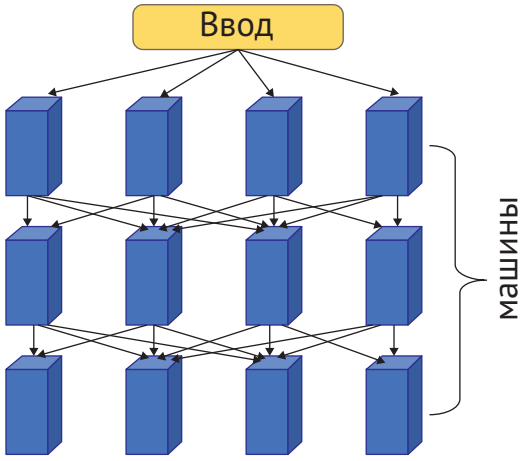


Рис. 10.1. Модель массивно-параллельной системы

Термин «массивно-параллельная система» относится к набору множества процессоров, структурированных для выполнения определенных операций в режиме параллельной работы.

Подобные системы существовали и до 1970-х годов, но с развитием технологий они претерпели существенные изменения. Как правило, такие системы относятся к определенному типу суперкомпьютеров, в которых вычислительные блоки, которыми могут быть центральные процессоры, графические процессоры или отдельные устройства (массивно-параллельные машины), скомпонованы в параллельную структуру, и каждый из них одновременно выполняет определенную часть задачи.

Сегодня массивно-параллельные системы становятся все более многочисленными и мощными. Такие системы используют Facebook и Google, поскольку только они способны обрабатывать огромные объемы данных из поисковых систем и социальных сетей.

Особая параллельная структура, состоящая из однородных и независимых блоков, также позволяет масштабировать эти системы в соответствии с потребностями вычислений.

Например, если одной из этих систем требуется увеличить ресурсы для выполнения задач, к ней можно с легкостью добавить дополнительные процес-

сорные блоки, при необходимости подключив новые машины. Если же вычислительные ресурсы окажутся избыточными, их можно задействовать по мере необходимости или перенаправить на другие задачи, что в данном случае помогает сократить излишнее расходование ресурсов.

Именно поэтому массивно-параллельные системы находят широкое применение в научных исследованиях и прикладных областях, а работа с ними (например, настройка производительности) может стать хорошей возможностью для трудоустройства.

Особенности этих систем, их постоянно растущая мощность и гибкость делают их подходящими для использования с технологиями искусственного интеллекта. Именно искусственный интеллект используется для настройки работы этих систем, что обеспечивает их динамическую адаптацию к вычислительным потребностям, делая их все более автономными и эффективными.

Эта новая область исследований в области искусственного интеллекта обладает невообразимой производительностью, непосредственным результатом которой будет постоянно растущая эффективность вычислений. Но конечная цель заключается не в этом. На самом деле с переходом от машинного обучения к глубокому обучению доступность ресурсов и вычислительная мощность, как показывает опыт прошлого, несомненно приведут к появлению нового подхода к созданию интеллектуальных систем. Например, уже сейчас ведется разработка новых методов, позволяющих внедрять в приложения формы логического мышления с использованием памяти вместо простого параметрического обучения. Еще одна технология, о которой в последнее время много говорят, заключается в параллельной передаче маркеров. В рамках этой технологии разрабатываются новые алгоритмы высокой степени параллелизма, способные трансформировать принципы использования нейронных сетей. Эти и многие другие инновации, подкрепленные использованием массивно-параллельных машин, открывают перед искусственным интеллектом новые возможности для решения сложных задач, связанных с необъятностью и разнообразием реальных данных. Кроме того, в области обработки данных внедряются такие новые технологии, как очень большие базы знаний (Very Large Knowledge Bases, VLKB).

Пограничные вычисления

В последних главах книги, 8 и 9, мы рассмотрели примеры того, как кластеры машин и облачные интернет-сервисы могут служить мощными инструментами, позволяющими создавать распределенные системы и, следовательно, масштабировать наши приложения для параллельных вычислений от отдельного компьютера до кластеров. В настоящее время именно такой подход является обычно используемым для тех, кто хочет разрабатывать и изучать приложения для машинного обучения и глубокого обучения. Но это еще не все: другие дисциплины, которые в настоящее время используют параллельные вычисления, также вскоре перейдут на эту форму вычислений благодаря постоянно растущей доступности этих систем.

В ближайшем будущем ожидается значительный прогресс в развитии таких распределенных систем. Совсем недавно в качестве новой вычислительной парадигмы, дополняющей рассмотренные в этой книге, была предложена технология пограничных вычислений (edge computing). В системах такого типа вычислительные ресурсы и ресурсы памяти расположены непосредственно рядом с данными и, как правило, поблизости от всех источников информации. С внедрением пограничных вычислений к существующим распределенным вычислениям (или облачным вычислениям) добавятся новые парадигмы и реализации, которые будут учитывать задержку и пропускную способность коммуникационных каналов при передаче информации между различными компонентами системы. Эти характеристики будут все больше взаимодействовать с системами безопасности и политик, которые по мере развития распределенных систем постепенно будут интегрированы в процесс обработки данных и повседневную деятельность.

Таким образом, обработка данных будет разделена на два уровня: с одной стороны, пограничный, а с другой – облачный, как показано на рис. 10.2.

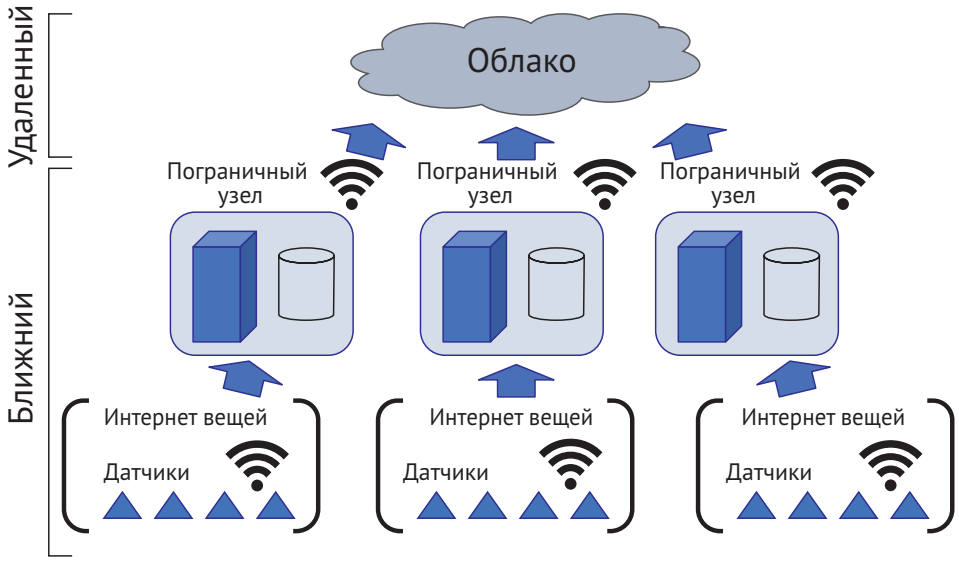


Рис. 10.2. Пограничные вычисления в системе Интернета вещей

Справедливости ради стоит добавить, что типы входных данных, подлежащих обработке, будут самыми разнообразными и все чаще будут поступать из объектов повседневного применения. Даже форма данных во время их обработки от источника через периферию до облака, где в конечном итоге из них будет извлечена информация на основе соответствующих моделей, будет эволюционировать по мере все более тесной интеграции интеллектуальных систем. На практике эти системы будут преобразовывать числовые формы, изображения, звуки и все другие знакомые нам виды информации в абстрактные

и обобщенные данные, в непривычные для нас формы, но более подходящие для искусственного интеллекта, которые затем будут использоваться этими системами для генерации знаний и построения логических суждений (а не просто для обучения).

Такая картина является лишь неточным представлением того, что ожидается в будущем, поскольку для этого необходимо пройти еще много этапов и решить множество технологических проблем. Однако именно к такой тенденции в будущем стремятся все приложения параллельных вычислений.

С практической точки зрения у нас уже есть несколько примеров, которые позволяют составить представление о перспективах. Например, в Google создан сервис Edge-TPU (*https://cloud.google.com/edge-tpu*), доступный для использования онлайн по подписке. Этот сервис использует ML-выводы с рядом многоуровневых технологий, из которых можно выбрать подходящую для вашей среды, включая TensorFlow в качестве фреймворка, см. табл. 10.1.

Таблица 10.1. Сервисы, доступные в Google Edge-TPU

	Периферия	**Облако**
Задача	ML-вывод	ML-вывод и обучение
Сервис	Linux, Windows	ИИ-платформа, механизм Kubernetes, вычислительный механизм, облачное ядро Интернета вещей
Фреймворк ML	TensorFlow Lite, NN API	TensorFlow, Scikit-learn, XGBoost, Keras
Аппаратные акселераторы	Периферийный CPU, GPU, CPU	Облачный TPU, GPU, CPU

Распределенная вычислительная инфраструктура с киберфизическими системами

Предыдущий раздел сразу же наводит на мысль о других отраслях и связанных с ними технологиях, которые имеют отношение к инновациям в области искусственного интеллекта и интеграции все более интеллектуальных систем в наше общество.

Мы упомянули источники данных, будущее пограничных вычислений и растущее использование датчиков и других форм сбора данных из реального мира. В последние годы технический прогресс в области коммуникационных технологий и возможность покрытия системами Wi-Fi привели к появлению и развитию Интернета вещей (Internet of Things, IoT). Киберфизические системы (Cyber-Physical Systems, CPS) играют решающую роль в этой новой распределенной системе сбора данных в реальном мире.

Это компьютерные системы, предназначенные для непрерывного взаимодействия с окружающей их реальной средой. Системы CPS должны отвечать

трем основным требованиям – так называемым свойствам 3 C (3 C properties), чтобы считаться таковыми:

- вычисления (computing);
- коммуникации (communication);
- управление (control).

Таким образом, системы с этими тремя свойствами должны быть способны взаимодействовать с окружающим миром, собирая данные через сеть датчиков или других человеко-машинных интерфейсов, обрабатывая их все более интеллектуальным образом и передавая обработанную информацию другим подключенным к ним компьютерным системам (пограничные вычисления). На рис. 10.3 показана эволюция схемы, представленной на рис. 10.2, с внедрением интеллектуальных систем, киберфизических систем, в пограничные вычисления, которые способны обрабатывать входные данные и экстраполировать абстрактную информацию уже на этом уровне, а также взаимодействовать с внешней средой через исполнительные механизмы или человеко-машинные интерфейсы (Human-Machine interfaces, HMI).

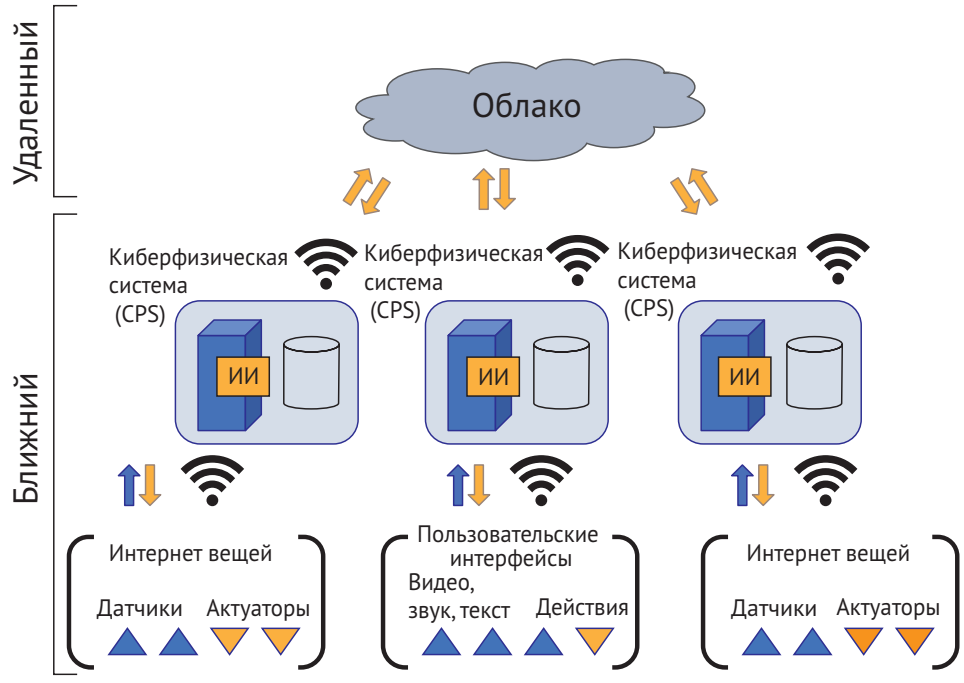


Рис. 10.3. Эволюция пограничных вычислений с помощью интеллектуальных (киберфизических) систем

Кроме того, эту новую интеллектуальную систему, киберфизическую систему, также можно будет использовать для управления физической средой,

в которой она функционирует, с помощью исполнительных механизмов или других систем физического взаимодействия на основе обработанной или полученной информации. На будущее ставится цель сделать эти системы все более интеллектуальными, интегрировав их с приложениями, способными учиться, анализировать и действовать подобно человеку. Таким образом, мы будем развивать новые технологии, подходы и методы для внедрения этих новых интеллектуальных систем, которые, безусловно, будут использовать параллельные вычисления.

Искусственный интеллект в кибербезопасности

Прямым следствием всех упомянутых в прежних разделах факторов является все более широкое внедрение интеллектуальных систем во все сферы человеческой деятельности, как профессиональной, так и социальной. Возложение всех этих операций и контроля на интеллектуальные системы с возможностью самоконтроля требует параллельного развития новых систем безопасности, способных гарантировать отсутствие каких-либо проблем при их использовании.

Такая реальность не за горами. Недавние исследования показывают, что искусственный интеллект уже охватывает многие области применения, некоторые из которых имеют важное значение для безопасности общества. Такие секторы, как финансы, военная промышленность, транспорт и здравоохранение, уже используют интеллектуальные системы, и в ближайшие годы их использование будет расширяться. Сбой в их работе или вторжение злоумышленников с целью захвата контроля над ними может нанести огромный ущерб обществу. Поэтому очевидно, что одновременно с этим необходимо разработать специальные системы безопасности, способные обеспечивать контроль, немедленно и корректно реагировать на возникающие ситуации.

В настоящее время кибербезопасность охватывает современные компьютеры с функциональными и неинтеллектуальными операционными системами и уже достаточно загружена решением таких проблем, как противодействие вторжениям, защита конфиденциальности и социальная инженерия. Распространение этих проблем на системы с искусственным интеллектом, которые все больше интегрируются в повседневную жизнь, является новым вызовом для завтрашнего дня.

Даже в этом случае искусственный интеллект может быть использован для решения вопросов безопасности. Рассмотрение взаимосвязи между ИИ и кибербезопасностью является ключевым фактором для будущего общества и использования информационных систем.

В связи с этим Агентство Евросоюза по кибербезопасности (European Union Agency for Cybersecurity, ENISA) выпустило интересный документ в формате PDF, который можно бесплатно скачать в интернете по адресу *https://www.enisa.europa.eu/publications/artificial-intelligence-cybersecurity-challenges*. В нем описываются проблемы, с которыми кибербезопасность столкнется в ближайшем будущем. Краткое изложение этих проблем представлено на рис. 10.4.

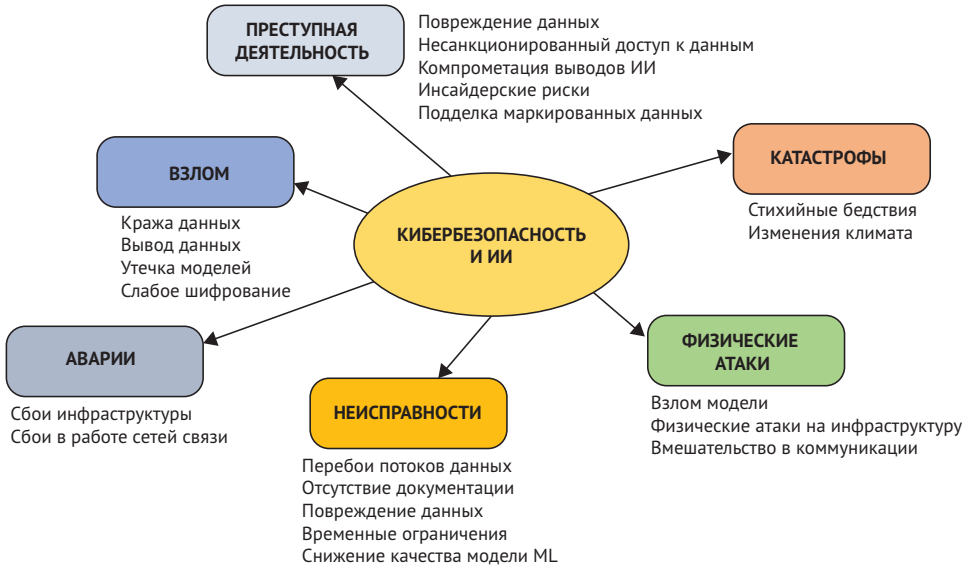


Рис. 10.4. Некоторые из угроз в сфере кибербезопасности в ближайшие годы

Вступление в эпоху Web 5.0

Даже во Всемирной паутине нас ждут большие перемены. Появление Web 5.0, также называемой Brain-Net,[1] кардинально изменит взаимодействие между пользователями. Эта новая система добавит к потоку обмена данными эмоциональность и ощущения, а также повысит степень взаимодействия между человеком и машиной.

Такая инновационная система повлечет за собой появление совершенно нового набора технологий, которые будут генерировать данные в новых формах, не учитываемых современными информационными системами. Эти данные будут принимать все более сложные и абстрактные формы, далекие от привычных нам сегодня простых цифр и текстов.

Поскольку наука о данных (и в первую очередь большие данные) основана исключительно на информации из интернета, нам придется адаптироваться к этим изменениям. Кроме того, если учесть, что для обработки текущих ежедневных запросов на услуги (например, Google обрабатывает около 3,5 млрд запросов ежедневно) требуются чрезвычайно мощные вычислительные системы, которые уже используют искусственный интеллект, становится ясно, что инновации в этой области должны развиваться быстрыми темпами, чтобы идти в ногу со временем.

Эти новые формы данных потребуют не только специальных алгоритмов, которые еще не разработаны, но и систем искусственного интеллекта, способных их понимать. Поэтому даже в этой области потребуются значительные

[1] Буквально «мозговая сеть». – *Прим. перев.*

усилия для внедрения необходимых инноваций. Все это будет интегрировано с концепциями, изложенными ранее в этой главе, такими как интеллектуальные вычисления на границе Сети (Edge Computing), которые позволят обрабатывать входные данные непосредственно «близко» к источнику, тем самым сокращая затраты на централизованную обработку и трафик данных.

Таким образом, Web 5.0 будет использовать децентрализованные веб-узлы (Decentralized Web Nodes, DWN), которые будут выполнять функции хранения данных и позволять субъектам любого типа (людям, организациям и т. д.) отправлять и хранить зашифрованные или общедоступные сообщения с данными. Это будет способствовать созданию широкого спектра децентрализованных приложений и протоколов, как показано на рис. 10.5.

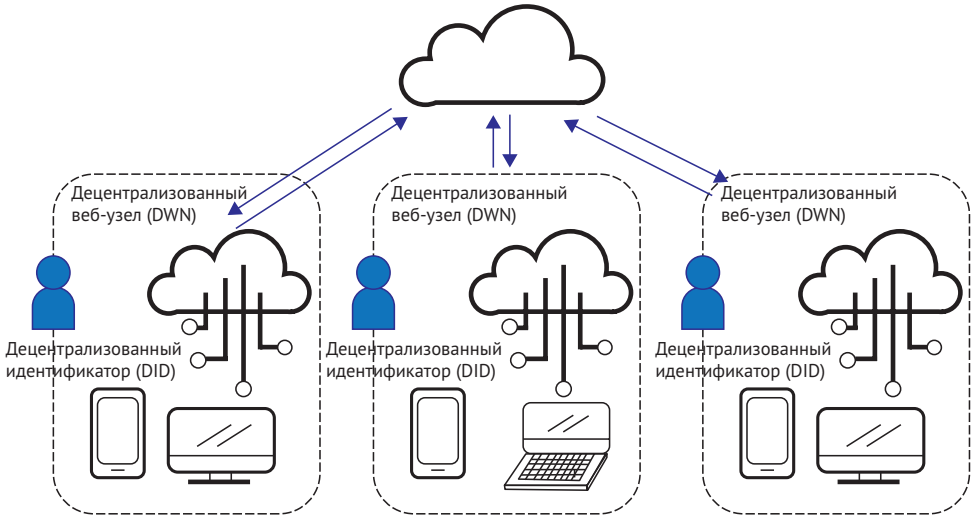


Рис. 10.5. Децентрализованные веб-узлы (DWN) в Web 5.0

Благодаря внедрению децентрализованных веб-узлов еще одним целевым направлением Web 5.0 станет децентрализация идентификации пользователей и хранения их данных. Управление учетными записями также будет децентрализовано за счет использования децентрализованных идентификаторов (Decentralized Identifiers, DID) – самостоятельных идентификаторов, которые позволят перенести аутентификацию и маршрутизацию из центра в периферию. Приложения, разработанные в этой области, вместо отправки данных идентификации в центральный модуль будут хранить их локально, отправляя только их обработанные версии. Даже если эта функция кажется тривиальной, она отвечает многим современным потребностям в области кибербезопасности и конфиденциальности данных. Она также значительно революционизирует формы обработки данных, что потребует новых систем обработки. Станет ли управление бесчисленными учетными записями и паролями просто плохим воспоминанием? Посмотрим.

Экзафлопсные вычисления

Возвращаясь к более технологической тематике, следует упомянуть, что одной из самых модных технологических инноваций будущего являются вычисления с производительностью на уровне экзафлопса (exascale computing). Этот термин обозначает новый уровень супервычислений, способный выполнять не менее 1 000 000 000 000 000 000 (экзафлопс, то есть 10^{18}, или 1 квинтильон) операций с плавающей запятой в секунду. Такая вычислительная мощность может существенно повысить производительность приложений, использующих параллельные вычисления.

Создание суперкомпьютеров на основе подобных систем обеспечит огромные преимущества для использования имитационных моделей, развития искусственного интеллекта и анализа данных благодаря возможности решения невероятно сложных задач, с которыми не справляются даже самые совершенные современные компьютеры.

Эта реальность уже наступает. В США калифорнийская компания HPE построила первый суперкомпьютер экзафлопсного уровня под названием Frontier, который будет установлен в Национальной лаборатории Ок-Ридж (2023), междисциплинарной научно-технической лаборатории при Министерстве энергетики США. Этот суперкомпьютер продемонстрировал производительность 1,1 экзафлопса, преодолев до сих пор недостижимый предел и тем самым заняв позицию самого мощного суперкомпьютера в мире.

Как и другие суперкомпьютеры экзафлопсного уровня в стадии разработки, Frontier был построен с использованием комбинаций процессоров и графических ускорителей, многопроцессорных узлов и других специально разработанных устройств, основанных исключительно на параллельных вычислениях. Несомненно, с распространением таких систем появится мощный стимул для создания новых подходов и методов в контексте параллельного программирования. Очень интересно, каким будет будущее развития технологий программирования.

Квантовые вычисления

Даже наше нынешнее представление о параллельных вычислениях в скором времени изменится с появлением новых вычислительных технологий, которые будут полностью отличаться от привычных нам. Одной из них являются квантовые вычисления (Quantum Computing). Эта вычислительная технология основана на квантовых свойствах материи, таких как суперпозиция состояний и квантовая запутанность. Благодаря этим свойствам такие системы способны выполнять операции с данными не с помощью транзисторов с двоичным состоянием (0 или 1), а с помощью квантовых битов, называемых кубитами, которые могут одновременно принимать несколько значений, что полностью противоречит основам двоичного кода.

Компьютеры на основе квантовых вычислений уже являются по сути параллельными на очень низком уровне, в отличие от того, к чему мы привыкли, и эти характеристики обещают революционные изменения в производительно-

сти вычислений. В силу своей природы эти технологии, несомненно, требуют коренного пересмотра основ параллельных вычислений, что приведет к появлению новых концепций и подходов.

Потенциал компьютеров такой мощности привел к настоящей гонке между различными странами мира за создание подобных вычислительных систем. В настоящее время подобные компьютеры уже существуют, хотя это и очень новая технология. Первый квантовый компьютер был выпущен на рынок компанией IBM в 2019 году и получил название IBM Q System One. Впоследствии США создали серию квантовых компьютеров под названием Eagle (127 кубитов) в виде чипов, которые будут адаптированы к новой системе: IBM System Two, с соответствующим программным обеспечением.

В Канаде стартап Xanadu присоединился к квантовым сверхдержавам, создав квантовый компьютер на основе запутанности фотонов под названием Borealis с вычислительной мощностью 216 кубитов. Эта система использовалась для вычисления гауссовой выборки бозонов из 216 фотонов всего за 36 мкс. Для решения такой задачи на самых быстрых современных суперкомпьютерах потребовалось бы более 9000 лет вычислений.

Что касается Европы, то из 6 строящихся квантовых компьютеров (один из которых находится в Италии) работают уже два (в Париже, Франция, и в Юлихе, Германия). Они сформируют первую в мире сеть квантовых компьютеров. Эти машины будут использоваться для исследований, но также будут доступны европейским стартапам и компаниям для стимулирования дальнейшего технологического прогресса. Еще более свежей (2022 год) является новость из Китая о создании первого в стране квантового компьютера Wuyan, оснащенного 24 кубитами и построенного компанией Origin Quantum Computing.

Квантовая технология, интегрированная в существующие системы HPC, позволит добиться значительных успехов во всех секторах промышленности.

Новые профессиональные возможности

Хорошей новостью для тех, кто выбирает профессию программиста, является то, что было сказано на одном из последних Всемирных экономических форумов (ВЭФ). Среди новых профессий, которые будут определять работу будущего, есть специалисты по искусственному интеллекту и данным, которые будут все больше пересекаться с человеческими навыками. От специалиста по данным до инженера по облачным технологиям, от эксперта по искусственному интеллекту до эксперта по кибербезопасности – все эти специальности присутствуют в списке 96 профессий будущего, приведенном в исследовании «Работа завтрашнего дня: картирование возможностей в новой экономике» (*Jobs of Tomorrow*: *Mapping Opportunity in the New Economy*), опубликованном Всемирным экономическим форумом.

Среди групп новых профессий есть «наука о данных и искусственный интеллект», что подтверждает высказанные в этой книге ожидания и концентрирует большую часть рассмотренных здесь тем. В то же время, согласно исследованию ВЭФ, в ближайшем будущем ожидается рост спроса на специалистов по

искусственному интеллекту, даже если количество открытых вакансий не будет высоким, и эта профессия по-прежнему будет оставаться нишевой для экспертов в данной области. С другой стороны, количество вакансий для специалистов по данным значительно вырастет (это третья по количеству открытых вакансий новая профессия).

Что касается навыков, необходимых в этом секторе, в исследовании предлагается термин «навыки, связанные с технологическими прорывами», под которым подразумеваются более продвинутые цифровые навыки, позволяющие их обладателям использовать и разрабатывать бизнес-модели, тем самым оказывая значительное и долгосрочное влияние на рынок труда. Фактически под этим термином мы находим следующие компетенции:

- науку о данных;
- обработку естественного языка;
- робототехнику и автоматизацию;
- кибербезопасность.

Все навыки, упомянутые в этой книге, связаны с искусственным интеллектом и – косвенно – с параллельными вычислениями.

Необходимые улучшения в параллельных вычислениях для Python

Сегодня все вычислительное оборудование является многоядерным, поэтому «Python должен все в большей степени обеспечивать параллелизм высокой производительности».

Именно об этом говорилось на EuroPython 2019. Эта цель поставлена на ближайшее будущее для всех разработчиков библиотек Python, которые столкнутся с необходимостью инноваций в своих библиотеках для адаптации к этим требованиям.

Впереди еще долгий путь. Несмотря на все, что мы видели в этой книге, потребуется много изменений для адаптации API этих библиотек к параллельным системам, которые скоро станут обычным явлением.

Учитывая природу Python, в этом процессе будут участвовать и пользователи этих библиотек, которые должны будут сигнализировать о важных потребностях своей профессиональной деятельности и продвигать изменения библиотек в соответствии с этими запросами.

Также и в этой области появится новое поколение разработчиков, для которых параллельное программирование станет основной формой мышления, и они смогут создавать новые конструкции и шаблоны с большей эффективностью и удобством в использовании. Например, эти новые разработчики смогут создавать новые библиотеки, интегрируя в них самые современные технологии параллельных вычислений. По сути, Python также подходит для обертывания нативных реализаций, зачастую разработанных на C (например, BLAS), что позволяет этому языку использовать преимущества инноваций из других

языков. Примером этого является библиотека MPIRE, которая предлагается в качестве эволюции модуля multiprocessing. В своей статье об этой библиотеке С. Янсен (S. Jansen) проводит ряд интересных тестов производительности наиболее распространенных библиотек, где показана производительность наиболее часто используемых для параллельных вычислений из числа рассмотренных в этой книге. На рис. 10.6 показаны его окончательные результаты, где выделена производительность различных библиотек и отражены тенденции времени выполнения по мере увеличения числа потоков.

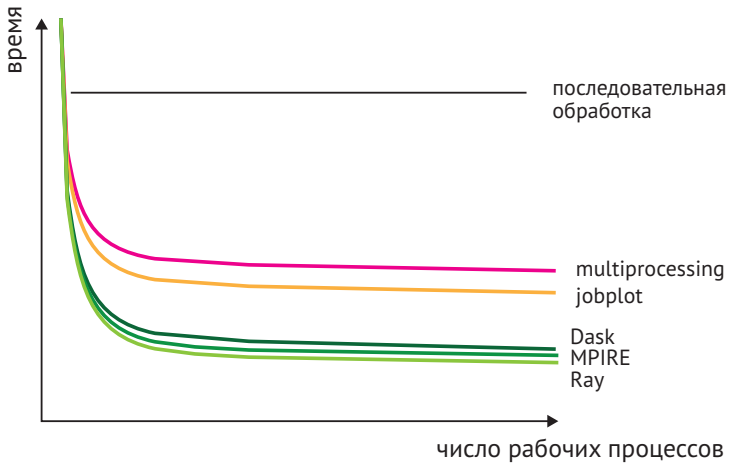


Рис. 10.6. Производительность некоторых библиотек для параллельных вычислений

Как сообщается в статье, библиотека MPIRE позволяет достичь той же производительности, что и распределенные системы с такими библиотеками, как Dask и Ray, но при этом остается в пределах одной машины. В этом случае производительность будет повышена без необходимости масштабирования вычислений на кластерах машин, что позволит снизить затраты и упростить реализацию кода.

Библиотека MPIRE, безусловно, является только началом. Другие библиотеки будут использовать другие механизмы для параллельных вычислений, скорее всего, обертывая другие нативные библиотеки (такие как CUDA для графических процессоров).

Что касается библиотек машинного обучения и глубокого обучения, то здесь улучшения и инновации, безусловно, будут происходить не только с появлением новых библиотек, но и с улучшением существующих, таких как PyTorch и TensorFlow.

Будущее PyTorch и TensorFlow

В отношении этих двух библиотек интересным фактором, касающимся их будущего, будет не только их способность внедрять современные технологии для

улучшения своей производительности, но также и их потенциал по предоставлению услуг и моделей сообществу пользователей.

В интересной статье от AssemblyAI «PyTorch и TensorFlow в 2023 году» (*PyTorch vs TensorFlow in 2023*) раскрывается интересная динамика между возможным будущим этих библиотек, которые, несомненно, будут конкурировать друг с другом. Например, в статье утверждается, что важным фактором является то, что в настоящее время создание новых моделей с нуля может быть слишком затратным для небольших компаний, и поэтому им придется использовать предварительно обученные модели. Таким образом, именно количество и качество предварительно обученных моделей, предлагаемых двумя библиотеками, будет определять их будущую судьбу.

В подтверждение этой гипотезы в статье иллюстрируется тенденция успеха этих двух фреймворков путем измерения количества посвященных им публикаций с течением времени. Ориентировочная реконструкция этого графика показана на рис. 10.7.

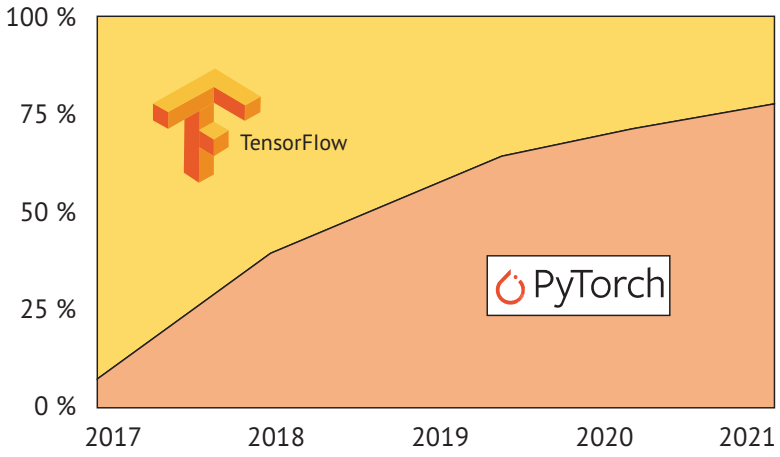


Рис. 10.7. Доля публикаций об использовании PyTorch или TensorFlow

Как легко заметить, в последние годы PyTorch набирает популярность в академических и исследовательских кругах и вытесняет TensorFlow. Это происходит вопреки преимуществам и отличной производительности TensorFlow. Из этого можно сделать вывод, что библиотеки (такие как языки программирования) являются инструментами, и простота и доступность их применения, например за счет более качественных предварительно обученных моделей, делает их более предпочтительными по сравнению с другими, которые могут быть более эффективными, но более сложными в использовании или требующими больше ресурсов.

Если хочется развеять все сомнения, в статье предлагается еще один параметр для подтверждения этой теории: рассмотрим количество репозиториев, в которых используются различные фреймворки глубокого обучения, включая PyTorch и TensorFlow. Ориентировочные данные представлены на рис. 10.8.

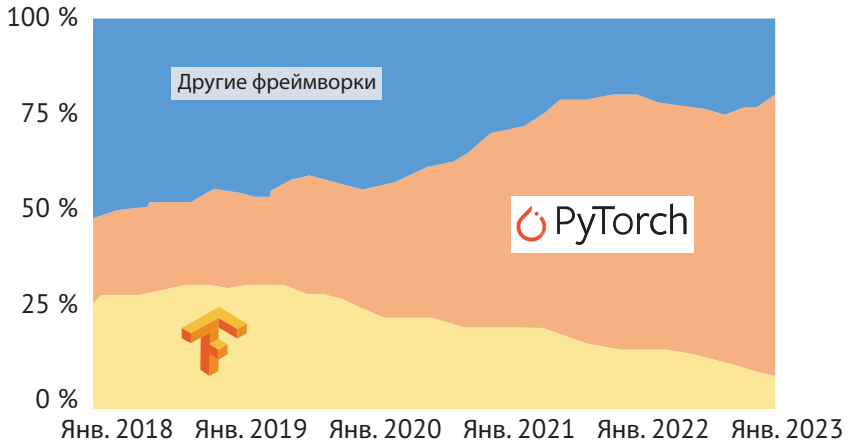


Рис. 10.8. Процент репозиториев по фреймворкам глубокого обучения

И в этом контексте мы также можем видеть более широкое распространение PyTorch по сравнению с TensorFlow за последние три года. Разумеется, эти тенденции отражают только динамику за последние 5 лет и, конечно, не могут гарантировать, что PyTorch станет эталонной библиотекой для глубокого обучения в ближайшем будущем. Внедрение инноваций в TensorFlow, изменение потребностей пользователей и появление новой библиотеки могут полностью изменить сложившуюся ситуацию.

Заключение

В этой завершающей главе представлен обзор современных технологий и инноваций, которые окажут влияние на мир параллельных вычислений. Прочитав все главы книги, вы, несомненно, получите представление о мире параллельного программирования и параллельных вычислений в настоящее время, а также о тенденциях его развития в ближайшем будущем. Вы убедитесь в значительной роли, которую эти темы будут играть в будущем не только в научной сфере, но и прежде всего в профессиональной и социальной сферах. Те, кто освоит эти системы, безусловно, получат в свои руки ключи к работе будущего, но не только это: они смогут лучше понять мир, который уже скоро будет нас окружать и который находится в постоянной эволюции.

Список литературы

- *https://ai.googleblog.com/2021/03/massively-parallel-graph-computation. html*.
- *https://www.ijcai.org/Proceedings/91-1/Papers/087.pdf*.
- *https://ieeeaccess.ieee.org/closed-special-sections/artificial-intelligence-in-parallel-and-distributed-computing/*.
- *https://ieeeaccess.ieee.org/closed-special-sections/distributed-computing-infrastructure-for-cyber-physical-systems/*.

- *https://ieeeaccess.ieee.org/closed-special-sections/artificial-intelligence-in-cybersecurity/*.
- *https://www.hpe.com/it/it/what-is/exascale.html*.
- *https://www.enisa.europa.eu/publications/artificial-intelligence-cybersecurity-challenges*.
- *https://insidehpc.com/2019/10/parallel-computing-in-python-current-state-and-recent-advances/*.
- *https://towardsdatascience.com/mpire-for-python-multiprocessing-is-really-easy-d2ae7999a3e9*.
- *https://ethz.ch/en/news-and-events/eth-news/news/2020/06/the-first-intuitive-programming-language-for-quantum-computers.html*.

Предметный указатель

Символы

2D массив 187
3D анимация 236
3D рендеринг 233

A

Anaconda, установка 148

B

bioconda, ссылка 231
Borealis (216 кубитов) 316

C

Celery, архитектура систем 145
Celery, определение 144
Celery, установка системы 148
CPU-связанный 27
CUDA 178
CUDA, установка 181

D

Dask Delayed 255
Dask-ML, установка 278
Docker, установка 151
Dramatiq, установка 162

E

Eagle (127 кубитов) 316

I

IBM Q System One 316
I/O-связанный 27
IoT *см.* Интернет вещей

K

k-кратная перекрестная проверка 275

N

Numba для CUDA 179

O

OpenCL 178, 195

P

Pipe в сравнении с Queue 95
pyOpenCL, установка 196
PyTorch, установка 282

Q

Queue 74

S

SaturnCloud.io 295
SCOOP, установка 170
Semaphore 63

W

Web 5.0 313
Wuyan (24 кубита) 316

А

Актор 163
Анализ методом конечных элементов (FEA) 225
Архитектура GPU 176
Асинхронная итерация 135
Асинхронная модель программирования 122
Асинхронное программирование 25, 121, 122
Ассоциация 273
Астрофизика 220

Б

Библиотека
  AsyncIO 123
  BioPython 231
  Curio, ссылка 140
  Dask 242
  Dramatiq 162
  mpi4py 103
  MPIRE 318
  Numba 179
  Obs-Py 219
  Pandas 241
  PolyFem 226
  Pyfolio 224
  PyGame 234
  PyOpenCL 195
  PyTorch 281
  Quandl 224
  QuantLib 224
  scikit-learn 275

SCOOP 169
SfePy 226
TA-Lib 224
TensorFlow 282
Trio, ссылка 140
Zipline 224
Большие данные 213
Брокер 146
Брокер, установка 152
Будущее параллелизма на Python 317
Будущее PyTorch и TensorFlow 318
Бэкенд результатов 147

В

Визуальные эффекты (VFX) 236
Высокопроизводительные вычисления (HPC) 212

Г

Геномика 230
Геометрия потока 187
Глобальная блокировка интерпретатора (GIL) 22
Глубокое обучение, определение 270
Глубокое обучение с PyTorch 282
Глубокое обучение с PyTorch и GPU 292
Горутины *см.* Сопрограммы
Граф задач 254

Д

Данные CTD 218
Двухмерная функция ядра 188
Двухточечная коммуникация 108
Декартова топология 116
Децентрализованный веб-узел 314
Децентрализованный идентификатор 314
Динамическое планирование задач 243

Ж

Жизненный цикл процесса 80

З

Загрузка данных в кластер 259
Задача 127, 147, 156
Закон Амдаля 35
Закон Густафсона 37

И

Игровой движок 232
Игровой движок Unity 233
Идентификационный номер процесса (PID) 18
ИИ в кибербезопасности 312
ИИ и параллельные вычисления 273
ИИ, определение 269
Интернета вещей 213, 310
Интерфейс передачи сообщений (MPI) 103
Искусственные нейронные сети 280
Искусственный интеллект 304
Использование идентификатора процесса 83
Использование подклассов Thread 48

К

Канал 93
Канал коммуникации 93
Квантовые вычисления 315
Киберфизическая система 310
Классификация 272
Класс
Condition 70
Lock 51
ProcessPoolExecutor 100
RLock 60
Thread 40
ThreadPoolExecutor 45
Кластеризация 273
Кластеризация k-средних 274
Кластер Saturn.io 257
Клиент 146
Ключевое слово
async 124
await 124
Коллективная коммуникация 109
Коллективная коммуникация
в режиме AlltoAll 113
со сбором данных 111
с распределением данных 110
с трансляцией данных 109
Коллекция больших данных 243
Коллекция
Dask 247
Dask Array 247
Dask DataFrame 247
Конкуренция потоков 47
Конструкция async / await 134
Контролируемое и неконтролируемое обучение 272

Л

Логическая иерархия программной модели GPU 180

М

Магнитно-резонансная томография (МРТ) 226
Массивно-параллельный ИИ 307

Масштабирование
PyTorch с Dask 294
scikit-learn с Dask-ML 278
Масштабируемость 36
Матрица смежности 116
Машинное обучение, определение 270
Метеорология 215
Метод коллекции 252
Метод
chain() 160
chord() 160
chunks() 162
group() 159
join() 42
mapReduce() 173
Механизм синхронизации на основе Event 72
Многозадачное программирование 27
Многозадачность 20, 23
Многозадачность и параллелизм в Python 23
Многозадачность и параллелизм, отличие 20
Многопоточное (конкурентное) программирование 122
Модель программирования PyOpenCL 197
Модуль concurrent.futures 45
Модуль Dask-ML 278
Модуль multiprocessing 29, 80
Модуль threading 29
Молекулярный докинг 229

Н

Набор данных Fashion-MNIST 284

О

Облегченная многозадачность с гринлетами 24
Обработка естественного языка (NLP) 271
Общая память 31
Объект Event 72
Объявление и вызов ядра 182
Ожидающий выполнения (awaitable) объект 124, 129
Океанография 217
Операция свертки (редукция) 115
Оптимизация коммуникации при помощи топологии 116
Отображение функции через пул процессов 96
Оценка эффективности параллельного программирования 34
Очередь 90
Очередь в асинхронной модели 138
Очередь задач 147

П

Память устройства 180
Память хоста 180
Параллелизм 16, 21
Параллелизм процессов 104
Параллельное вычисление, определение 213
Параллельное вычисление, преимущество 214
Параллельное и распределенное глубокое обучение 280
Параллельное и распределенное машинное обучение 274
Параллельное отображение с chunksize 99
Плюсы и минусы асинхронного и синхронного программирования 121
Пограничные вычисления 309
Подпроцесс 19
Поток 19
Поток и процесс в Python 23
Потоковый мультипроцессор (SM) 177
Потоковый процессор (SP) 177
Поэлементные вычисления с PyOpenCL 208
Предварительная обработка 26
Применение mpi4py 107
Пример глубокого обучения с набором данных Fashion-MNIST 284
Пример программирования с Numba 184
Примитив 159
Проблема потоков в Python (GIL) 22
Программирование кластера 260
Программирование GPU на Python 178
Протокол управления контекстом для Lock 55
Профессия будущего 316
Профилирование 38
Процесс 19
Пул процессов 85

Р

Рабочий процесс 146
Разведка ресурсов 222
Разработка лекарственных препаратов 228
Разработка параллельного игрового движка 234
Разработка программы с pyOpenCL 198
Распределение памяти между процессорами 31
Распределенная память 32
Распределенная система, определение 142
Распределенное программирование 33
Регрессия 272
Режим работы канала коммуникации 90

С

Сейсмология 219
Сервис Google Edge-TPU 310
Сигнатура 158
Сильная масштабируемость 36
Синтаксис async/await 123
Синхронизация потоков 44, 50
Синхронное программирование 25, 121
Слабая масштабируемость 36
События 133
Сообщество Kaggle 256
Сопрограмма 124
Суммирование двух матриц 189

Т

Тестирование производительности в Python 37
Трансляция 109
Трассировка лучей 237

У

Уменьшение размерности 273
Умножение матриц 192, 201
Уравнение в частных производных 225
Ускорение 34

Ф

Финансовая математика 223
Футура 130

Ц

Центральный процессор (CPU) 19
Цикл событий 134

Ч

Человеко-машинный интерфейс 311
Численное прогнозирование погоды (NWP) 215

Ш

Шаблон синхронизации потоков 44

Э

Экзафлопс 315

Я

Ядро 19

Книги издательства «ДМК Пресс» можно купить оптом и в розницу
на складе издательства по адресу:
г. Москва, ул. Электродная, д. 2, стр. 12, офис 7,
тел. +7(499)322-19-38,
а также заказать на сайте **www.dmkpress.com**
с доставкой в любой регион РФ

Фабио Нелли

Параллельное программирование на Python

Код на Python для ПК и кластеров
с использованием CUDA, PyTorch и Dask

Главный редактор *Яценков В. С.*
dmkpress@gmail.com
Перевод с английского *Бахур В. И.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 26,49.
Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**