УДК 004

Применение многопоточности в реализации стандартных алгоритмов сортировки и поиска

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье рассмотрены реализации стандартных алгоритмов сортировки слиянием и быстрой сортировки, а также бинарного поиска с внедрением принципов многопоточности. Описана использованная библиотека в Python, а также проведено сравнение между однопоточным и многопоточным алгоритмами.
Ключевые слова: алгоритмизация, программирование, процессы, многопоточность, сортировка, поиск, Python

Application of multithreading in the implementation of standard sorting and search algorithms

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
The paper considers implementations of standard algorithms of merge sorting and quick sorting, as well as binary search with implementation of multithreading principles. The library used in Python is described, and a comparison between single-threaded and multithreaded algorithms is made.
Keywords: algorithmisation, programming, processes, multithreading, sorting, searching, Python

1. Введение

1.1 Актуальность

В современном мире объемы данных, которые необходимо обрабатывать, постоянно растут. Традиционные однопоточные алгоритмы сортировки и поиска зачастую не справляются с такими объемами в приемлемое время. Многопоточные алгоритмы позволяют значительно ускорить процесс обработки данных за счет параллельного выполнения задач.

Многопоточные алгоритмы сортировки и поиска могут значительно сократить время выполнения за счет параллельной обработки данных. В связи с этим появляется необходимость создания новых версий стандартных

алгоритмов, в которых будет использоваться многопоточность для улучшения работы программ.

1.2 Обзор исследований

С. А. Раздобудов и И. И. Сальников провели исследование и анализ возможностей реализации многопоточности при разработке программ на языке C++ [1]. Для выбора оптимального метода распараллеливания были рассмотрены две модели: параллельного выполнения задач и параллельного использования данных.

Е. В. Фешина, Д. А. Омельченко и Р. Г. Гонатаев описали механизмы работы многопоточности и асинхронности в объектно-ориентированном языке программирования Python [2]. Также в статье отображена работа Global Interpreter Lock и проблемы, с которыми можно столкнуться при разработке многопоточных алгоритмов. В статье также рассмотрены возможности языка программирования, его особенности и преимущества.

К. С. Новиков, А. В. Григорьев, А. Г. Избасов, И. И. Кочегаров и Р. К.Валеев рассмотрели способы реализации многопоточности на примере программных систем, написанных на языке программирования Python [3]. В статье описываются функции стандартных библиотек treading и multiprocessing. Также демонстрируются сценарии и возможности многопоточного программирования в задачах обработки визуальных данных.

1.3 Цель исследования

Цель – написать стандартные алгоритмы сортировки (сортировка слиянием и быстрая сортировка) и алгоритм бинарного поиска с учётом многопоточности.

2 Материалы и методы

Для реализации поставленной цели используется язык программирования Python и модули для организации многопоточности.

3 Результаты и обсуждения

Перед началом реализации многопоточных алгоритмов сортировки и поиска стоит определить – какие алгоритмы можно изменить, внедрив в них многопоточность. Можно условно разделить алгоритмы сортировки и поиска на 2 категории: реализованные без использования рекурсия и с использованием рекурсии.

К рекурсивным сортировкам относятся, в первую очередь, 2 одни из самых распространённых алгоритма: сортировка слиянием и быстрая сортировка.

Начать стоит с сортировки слиянием, так как быстрая сортировка является улучшением данного алгоритма, соответственно их принцип работы схож (за исключением подхода к разделению массива).

Перед добавлением многопоточности в исходный алгоритм стоит понять принцип работы этого алгоритма. Алгоритм сортировки слиянием

можно разделить на несколько этапов, которые необходимо выполнить для достижения результата:

1. Сортируемый массив разбивается на две части примерно одинакового размера.
2. Каждая из получившихся частей сортируется отдельно, например – тем же самым алгоритмом.
3. Затем происходит слияние отсортированных частей.

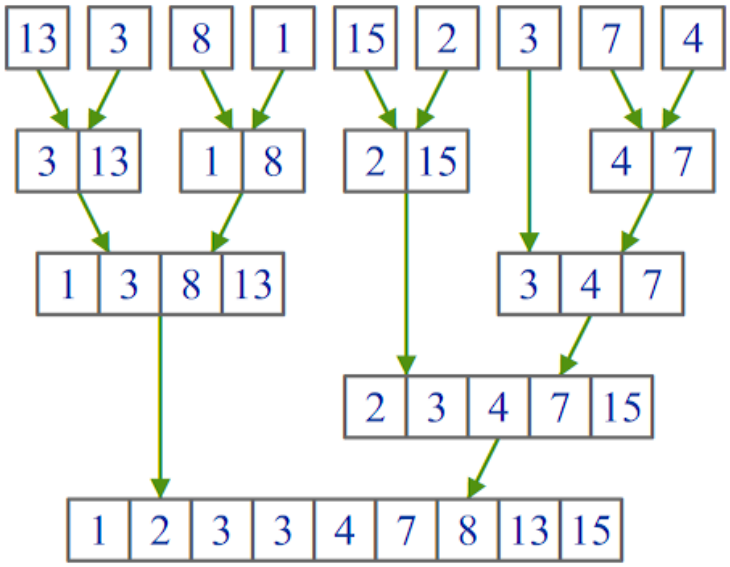

Рисунок 1. Пример выполнения процедуры слияния элементов массива

Пункт 1 повторяется до тех пор, пока части массивов не будут составлять 1 или 2 элемента. Деление происходит по определённому элементу, который должен всегда выбираться по его расположению в массиве – в сортировке слиянием это середина массива. Таким образом можно реализовать данный алгоритм следующим образом: сначала создаётся функция, в которой будет происходить и деление, и слияние массивов (можно создать 2 отдельные функции для проведения этих операций, но в стандартной реализации алгоритма такое не предусмотрено). На первом этапе выполнения данной функции необходимо разделить массив на 2 примерно равные части и для этих частей вызвать функцию сортировки слиянием (тем самым образуя рекурсию):

```
def merge_sort(arr):
  if len(arr) > 1:
     mid = len(arr) // 2
     left_half = arr[:mid]
     right_half = arr[mid:]

     merge_sort(left_half)
     merge_sort(right_half)
```

Рекурсивные вызовы будут происходить до тех пор, пока длина массива, поступающего на вход функции, не будет равна 1.

После завершения работы рекурсии можно начать слияние двух массивов в один общий. Сделать это можно через стандартный цикл while с проверкой условий на позиции указателей в сливаемых массивах:

```
i = j = k = 0

while i < len(left_half) and j < len(right_half):
  if left_half[i] < right_half[j]:
     arr[k] = left_half[i]
     i += 1
  else:
     arr[k] = right_half[j]
     j += 1
  k += 1
```

Также стоит учитывать, что указатель одного из массивов может не достигнуть до конца во время выполнения данного цикла, поэтому нужно дозаполнить результирующий массив элементами, которые остались после выполнения цикла:

```
while i < len(left_half):
  arr[k] = left_half[i]
  i += 1
  k += 1

while j < len(right_half):
  arr[k] = right_half[j]
  j += 1
  k += 1
```

Таким образом работает сортировка слиянием. Асимптотика данного алгоритма в лучшем, среднем и худшем случаях O(n*logn), что делает такую сортировку устойчивой к различным входным данным и одной из лучших среди алгоритмов сортировки.

Данная сортировка может быть изменена путём добавления многопоточности с помощью модуля threading. Данный модуль является стандартным в Python и установлен в ЯП по умолчанию. Threading предоставляет возможность работы с потоками, что позволяет выполнять несколько операций одновременно в одном процессе. Этот модуль полезен для задач, которые могут выполняться параллельно, например, для выполнения ввода-вывода или работы с многопоточными алгоритмами.

Рассмотрим простой пример, где создается несколько потоков для выполнения различных задач:

```
import threading
import time
```

```
    def worker(name):
        print(f"Поток {name} начал работу")
        time.sleep(2)
        print(f"Поток {name} закончил работу")

    threads = []
    for i in range(5):
        thread = threading.Thread(target=worker, args=(f'ПТК-
{i}',))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    print("Все потоки завершили свою работу!")
```

Через цикл for создаются объекты класса Thread, которые выполняют функцию worker. В данной функции просто стоит ожидание на завершение работы через метод sleep() из модуля time. В функцию worker передается имя потока, которое используется для вывода сообщений. Далее каждый поток засыпает на 2 секунды, симулируя выполнение задачи.

Метод join() в данном примере используется для блокировки основного потока до завершения всех потоков, чтобы гарантировать, что все задачи выполнены до завершения программы.

Теперь возникает вопрос – как можно изменить исходный алгоритм сортировки слиянием, чтобы он работал с учётом многопоточности?

Благодаря тому, что сортировка слиянием построена на принципе "Разделяй и властвуй", выполнение данного алгоритма можно весьма эффективно распараллелить. При оценке асимптотики допускается, что возможен запуск неограниченного количества независимых процессов, т.е. процессов с вычислительными ресурсами, не зависящими от других процессов, что на практике не достижимо. Более того, при реализации имеет смысл ограничить количество параллельных потоков.

Внесем в алгоритм сортировки слиянием следующую модификацию: будем сортировать левую и правую части массива параллельно.

```
    import threading

    def merge_sort_multithreaded(arr, thread_name="Main
Thread"):
        if len(arr) > 1:
            mid = len(arr) // 2
            left_half = arr[:mid]
            right_half = arr[mid:]

            print(f"{thread_name} сортирует следующую часть
массива: {arr}")
```

```
            left_thread                                                 =
threading.Thread(target=merge_sort_multithreaded,
args=(left_half, f"ПТК-{threading.get_ident()}-L"))
            right_thread                                                =
threading.Thread(target=merge_sort_multithreaded,
args=(right_half, f"ПТК-{threading.get_ident()}-R"))

            left_thread.start()
            right_thread.start()

            left_thread.join()
            right_thread.join()

            i = j = k = 0

            while i < len(left_half) and j < len(right_half):
                if left_half[i] < right_half[j]:
                    arr[k] = left_half[i]
                    i += 1
                else:
                    arr[k] = right_half[j]
                    j += 1
                k += 1

            while i < len(left_half):
                arr[k] = left_half[i]
                i += 1
                k += 1

            while j < len(right_half):
                arr[k] = right_half[j]
                j += 1
                k += 1

            print(f"{thread_name} Завершил сортировку следующей
части массива: {arr}")
```

Main Thread сортирует следующую часть массива: [12, 11, 13, 5, 6, 7]
ПТК-12164-L сортирует следующую часть массива: [12, 11, 13]ПТК-12164-R сортирует следующую часть массива: [5, 6, 7]

ПТК-13060-R сортирует следующую часть массива: [11, 13]ПТК-9100-R сортирует следующую часть массива: [6, 7]

ПТК-13060-R Завершил сортировку следующей части массива: [11, 13]ПТК-9100-R Завершил сортировку следующей части массива: [6, 7]

ПТК-12164-L Завершил сортировку следующей части массива: [11, 12, 13]ПТК-12164-R Завершил сортировку следующей части массива: [5, 6, 7]

Main Thread Завершил сортировку следующей части массива: [5, 6, 7, 11, 12, 13]
Результат: [5, 6, 7, 11, 12, 13]

Рисунок 2. Результат выполнения многопоточной сортировки слиянием

Асимптотика такого алгоритма будет отличаться от однопоточной. Количество элементов n будет делиться примерно равномерно между допустимым количеством потоков N, а также нужно к данной асимптотике прибавить ещё раз n, которое будет обозначать необходимое количество

шагов для выполнения слияния подмассивов что приведёт к следующей асимптотике: $O(\frac{n}{N} log \frac{n}{N} + n)$. В числах разница описана ниже (см. таблицу 1), при этом количество потоков всегда фиксированное, а именно 8.

Таблица 1. Сравнение асимптотик алгоритмов для различного количества потоков

Количество элементов n	Однопоточная сортировка	Многопоточная сортировка
10	33.2	10.4
100	664.4	145.5
10000	132877.1	22859.6
1000000	$1.9 * 10^7$	$3.12 * 10^6$

Можно сделать вывод, что многопоточная реализация алгоритма сортировки слиянием работает значительно быстрее, чем однопоточная. Но стоит учитывать, что такой алгоритм может не подойти для работы со слишком большими данными, а также при слишком большом количестве доступных потоков.

Улучшить работу с помощью внедрения многопоточности можно также и быструю сортировку, так как она работает по схожему принципу. Только отличается подход к делению массива на подмассивы: как правило выбирается средний элемент по значению и на основе этого элемента формируются 2 новых подмассива: в одном подмассиве записаны элементы, меньшие выбранного среднего, во втором наоборот – большие. Многопоточность можно реализовать на уровне разделения массивов, как и в случае с сортировкой слиянием:

```
import threading

def quicksort_multithreaded(arr, low, high, thread_name="Main Thread"):
    if low < high:
        pi = partition(arr, low, high)

        print(f"{thread_name} начал сортировку подмассива {arr[low:high+1]}")

        left_thread = threading.Thread(target=quicksort_multithreaded, args=(arr, low, pi - 1, f"ПТК-{threading.get_ident()}-L"))
        right_thread = threading.Thread(target=quicksort_multithreaded, args=(arr, pi + 1, high, f"ПТК-{threading.get_ident()}-R"))

        left_thread.start()
        right_thread.start()
```

```
            left_thread.join()
            right_thread.join()

            print(f"{thread_name} завершил сортировку
подмассива: {arr[low:high+1]}")

    def partition(arr, low, high):
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            if arr[j] < pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        return i + 1
```

```
Main Thread начал сортировку подмассива [1, 5, 8, 9, 10, 7]
ПТК-141076-R начал сортировку подмассива [7, 9, 10, 8]
ПТК-189900-R начал сортировку подмассива [8, 10, 9]
ПТК-189868-R начал сортировку подмассива [9, 10]
ПТК-189868-R завершил сортировку подмассива: [9, 10]
ПТК-189900-R завершил сортировку подмассива: [8, 9, 10]
ПТК-141076-R завершил сортировку подмассива: [7, 8, 9, 10]
Main Thread завершил сортировку подмассива: [1, 5, 7, 8, 9, 10]
```

Рисунок 3. Пример выполнения многопоточной быстрой сортировки

Стоит также обратить внимание и на алгоритмы поиска, которые так или иначе связаны с сортировками. Одним из самых популярных алгоритмов поиска является двоичный поиск, который также в своей классической реализации является рекурсивным алгоритмом.

Двоичный поиск заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект. Или же, в зависимости от постановки задачи, мы можем остановить процесс, когда мы получим первый или же последний индекс вхождения элемента. Последнее условие – это левосторонний/правосторонний двоичный поиск.

Идея поиска заключается в том, чтобы брать элемент посередине, между границами, и сравнивать его с искомым. Если искомое больше(в случае правостороннего — не меньше), чем элемент сравнения, то сужаем область поиска так, чтобы новая левая граница была равна индексу середины предыдущей области. В противном случае присваиваем это значение правой границе. Проделываем эту процедуру до тех пор, пока правая граница больше левой более чем на 1.

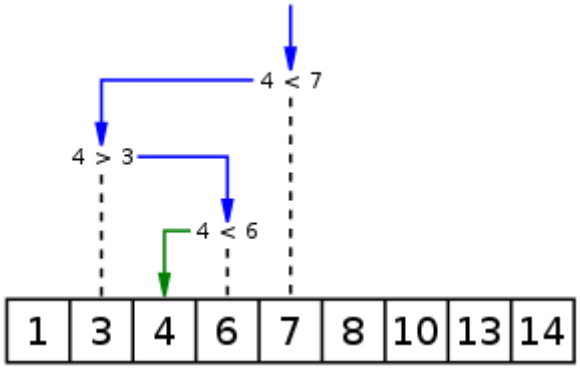

Рисунок 4. Пример работы двоичного поиска

Стандартная реализация алгоритма выглядит следующим образом и работает за O(logn):

```
def binary_search(arr, target, left, right):
    if left > right:
        return -1

    mid = left + (right - left) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, target, left, mid - 1)
    else:
        return binary_search(arr, target, mid + 1, right)
```

Можно оптимизировать данный алгоритм и избавиться от рекурсии. Тогда в многопоточной версии поиска достаточно запустить процессы на каждом потоке и привязать к этим потокам работу двоичного итеративного поиска. Такой алгоритм будет выглядеть следующим образом:

```
import threading
def multithreaded_binary_search(arr, target):
    num_threads = 8
    thread_list = []
    result = []
    segment_length = len(arr) // num_threads

    for i in range(num_threads):
        low = i * segment_length
        high = (i + 1) * segment_length - 1 if i != num_threads - 1 else len(arr) - 1
        thread_name = f"ПТК-{i+1}"
        thread = threading.Thread(target=binary_search, args=(arr, low, high, target, result, thread_name))
        thread_list.append(thread)
        thread.start()
```

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

    if result:
        return result[0]
    else:
        return -1
```

ПТК-1 не нашёл X ПТК-2 не нашёл X ПТК-3 не нашёл X ПТК-4 не нашёл X ПТК-5 не нашёл X ПТК-6 не нашёл X ПТК-7 нашёл X на позиции 6 ПТК-8 не нашёл X

X найдено на позиции 6

Рисунок 5. Результат работы многопоточного двоичного поиска

4 Выводы

Можно сделать вывод, что стандартные алгоритмы сортировок и поиска могут быть реализованы с внедрением многопоточности, что может позволить ускорить работу этих же алгоритмов в значительной мере. Но при этом нужно учитывать, что неправильная работа с потоками может привести к нежелательным последствиям (в лучшем случае к понижению быстродействия работы алгоритма). В рамках данной статьи были рассмотрены многопоточные реализации таких стандартных алгоритмов как: алгоритм сортировки слиянием, быстрая сортировка и двоичный поиск.

Библиографический список

1. Раздобудов С. А., Сальников И. И. Исследование и анализ возможности реализации многопоточности при разработке по на языке C++ // Международный студенческий научный вестник. 2018. № 3-2. С. 334-337.
2. Фешина Е. В., Омельченко Д. А., Гонатаев Р. Г. Многопоточность и асинхронность в языке программирования Python // Инновации. Наука. Образование. 2021. № 28. С. 988-992.
3. Новиков К. С. и др. Реализация многопоточности в программных системах на языке Python // Труды международного симпозиума "Надежность и качество". 2023. Т. 1. С. 309-311.
4. Целочисленный двоичный поиск - викиконспекты URL: https://neerc.ifmo.ru/wiki/index.php?title=Целочисленный_двоичный_поиск. (Дата обращения: 29.06.2024).
5. PSRS-сортировка - викиконспекты URL: https://neerc.ifmo.ru/wiki/index.php?title=PSRS-сортировка. (Дата

обращения: 29.06.2024).
6. Многопоточная сортировка слиянием URL: https://neerc.ifmo.ru/wiki/index.php?title=Многопоточная_сортировка_слиянием. (Дата обращения: 29.06.2024).
7. Быстрая сортировка URL: https://neerc.ifmo.ru/wiki/index.php?title=Быстрая_сортировка. (Дата обращения: 29.06.2024).