Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа ТП.22Б07-мм

Экспериментальное исследование производительности алгоритма обхода графа в ширину

БУРАШНИКОВ Артем Максимович

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
доцент кафедры информатики, к. ф.-м. н., С. В. Григорьев

Санкт-Петербург
2023

Оглавление

Введение **3**

1. Постановка задачи **4**

2. Обзор предметной области **5**

2.1. Существующие исследования 5

2.2. Обход в ширину в контексте линейной алгебры 6

2.3. Представление графа в памяти компьютера 6

2.4. Деревья квадрантов 8

2.5. Языковые особенности `F#` 9

3. Детали реализации **12**

3.1. Концепция алгоритма 12

3.2. Параллельные вычисления 13

4. Подготовка тестового стенда **15**

4.1. Характеристики оборудования 15

4.2. Исследовательские гипотезы 15

4.3. Использованные метрики 16

4.4. Инструменты для измерений 17

4.5. Набор данных 17

4.6. Эксперимент 18

Заключение **28**

Список литературы **29**

Введение

Использование такой абстракции как *граф* для анализа и изучения различных форм реляционных данных имеет большое значение. Теоретические проблемы, существующие в областях применения, включают в себя определение и выявление значащих объектов, обнаружение аномалий, закономерностей или внезапных изменений, группировке тесно связанных сущностей. Поиск в ширину (англ. — *Breadth-First Search*, сокр. — *BFS*) и поиск в глубину (англ. — *Depth-First Search*, сокр. — *DFS*) являются двумя основными алгоритмами для исследование графов.

Для решения задач теоретического анализа в современных приложениях важна скорость вычислений. Одним из способов получения ускорение являются *параллельные вычисления*, дающие выигрыш в производительности на современных многоядерных системах. Благодаря природе работы, алгоритм обхода в ширину естественным образом позволяет внедрять в свою реализацию использование дополнительных *потоков (англ. — threads)* и *ядер процессора (англ. — processor cores)*. Таким образом, простота реализации, широкая область применения и возможность использования параллельных вычислений делают BFS более популярным инструментом и объектом исследований, чем DFS.

1 Постановка задачи

Целью работы является проведение экспериментального исследования производительности обхода в ширину и анализ следующих исследовательских вопросов.

- **RQ1**
  При каких параметрах графа выгоднее использовать параллельную версию алгоритма, а при каких последовательную?
- **RQ2**
  Использование какого количества потоков даёт наибольший выигрыш в производительности и почему?

Для выполнения поставленной цели были сформулированы перечисленные ниже задачи.

- Реализовать параллельную и последовательную версии алгоритма обхода в ширину с использованием структур, подходящих для хранения в памяти компьютера разреженных матриц и векторов.
- Оценить влияние конкретных характеристик графа и количества используемых потоков на итоговую производительность BFS и найти критические величины указанных параметров, при которых она максимальна.

Выполнение обозначенных задач позволило определить какая версия алгоритма (последовательная или параллельная) предпочтительнее к использованию при том или ином сценарии.

2 Обзор предметной области

Перед проведения эксперимента необходимо ознакомиться с существующими работами, затрагивающими проблемы многопоточной реализации BFS. Кроме того, требуется рассмотреть преимущества использования абстрактных операций линейной алгебры в алгоритме обхода в ширину, отметив выбор подходящей структуры данных для хранения матриц и векторов, а также обратить внимание на особенности выбранного для реализации языка программирования.

2.1 Существующие исследования

Алгоритм обхода графа в ширину ввиду своей прикладной значимости был проанализирован в разном контексте в ряде исследовательских работ. Некоторые из них рассмотрены далее.

В работе [4] отмечается, что производительность BFS в значительной мере зависит от топологии подаваемого на вход графа. Ся Инлун (Xia Yinglong) и Празанна Виктор (Prasanna Viktor) показали, что в случае большого количества итераций алгоритма при малых количествах вершин/ребер между итерациями (то есть малом количестве вершин во фронте на каждой итерации) параллельная версия терпит снижение производительности из-за накладных расходов на создание параллельных задач.

Кроме того, в исследовании [2] Вират Агарвал (Virat Agarwal) и др. продемонстрировали, что последовательная версия алгоритма в некоторых ситуациях оказывается предпочтительнее не оптимизированной параллельной ввиду большой задержки работы с памятью и высокой вычислительной стоимостью её синхронизации.

Упомянутые авторы нашли решение проблем оптимизации в тонкой настройке взаимодействия с общей памятью в используемой архитектуре или применении адаптивных алгоритмов, способных динамически контролировать количество используемых потоков во время исполнения.

Анализируя представленные работы, можно прогнозировать зависимости между параметрами входных данных, выбранной архитектурой

и ожидаемой производительностью используемой реализации обхода в ширину. Данное экспериментальное исследование посвящено выявлению таких зависимостей для алгоритма BFS, реализованного с применением методов линейной алгебры, что существенным образом влияет не только на сам алгоритм, но и на внутреннее представление графа в памяти компьютера.

2.2 Обход в ширину в контексте линейной алгебры

Для эффективного представления и манипулирования графовыми структурами можно использовать матрицы и вектора. Рассмотрим теоретические основы такой абстракции и её преимущества.

GraphBlas

Набор операций, который может быть применен к матрицам и векторам для выполнения различных графовых алгоритмов (таких как обходы, поиск путей, кластеризация и др.) заложен в *GraphBlas* [1] [1].

Это инициитива, нацеленная на создание стандартизованного интерфейса для разработки графовых алгоритмов. Её целью является обеспечение высокой производительности и переносимости для широкого спектра решений, работающих с различными графовыми структурами.

Пользователю предоставляется спецификация, определяющая абстрактное умножение матрицы на вектор, сложение векторов и умножение матрицы на матрицу. Описанная семантика и требования для интерфейса позволяют создавать собственные реализации на различных языках программирования и для различных аппаратных платформ.

2.3 Представление графа в памяти компьютера

Нужно отметить, что реальные графы сильно разреженны, но в то же время обладают сравнительно большим количеством вершин. Плотность

[1] Форум, посвещенный стандарту GraphBlas. Дата посещения: 23 мая 2023 г.

графа (англ. — *density*) можно вычислить по формуле:

$$\text{density} = \frac{2 \cdot \left|E\right|}{\left|V\right| \cdot (\left|V\right| - 1)}, \tag{1}$$

где $\left|E\right|$ — количество рёбер, а $\left|V\right|$ — количество вершин. Разреженными можно считать графы со значением плотности $< 10\%$. В то время как на практике в основном встречаются сильно разреженные графы со значением density $\ll 2\%$.

Хранить такие данные в виде двумерных таблиц не является эффективным решением, поэтому используют специальные структуры. Рассмотрим некоторые из них.

- **Список смежности**
  Это один из наиболее простых и эффективных способов хранения разреженных графов. Для каждой вершины выделяется список ее соседей. Это может быть реализовано, к примеру, с помощью массива списков, где каждый индекс массива представляет вершину, а лежищай по выбранному индексу список содержит соседей этой вершины.

- **Матрица смежности**
  В матрице смежности каждый элемент указывает наличие или отсутствие ребра между двумя вершинами. В случае разреженных графов, где большая часть элементов матрицы будет нулевыми, может быть эффективно использована разреженная матрица, где хранятся только ненулевые значения.

- **Список ребер**
  Вместо хранения информации о соседних вершинах можно хранить список всех ребер графа. Каждое ребро тогда представляется парой вершин, которые оно соединяет. Этот подход эффективен для определенных операций, таких как перебор всех ребер.

- **Компактные структуры данных**
  Например, CSR (*Compressed Sparse Row*) и CSC (*Compressed Sparse*

Column), которые оптимизируют использование памяти для разреженных графов, сохраняя при помощи массивов информацию о вершинах, ребрах и их связях, и позволяют эффективно выполнять операции над разреженными графами.

Каждый метод имеет свои преимущества и недостатки с точки зрения памяти и производительности. Для сильно разреженных графов, которые были использованы для исследования BFS, в контексте линейной алгебры их представление лучше всего подходит в виде матрицы смежности. Более того, поскольку большинство значений у таких матриц равно нулю, для их хранения целесообрано использовать дополнительную структуру данных, абстракцией над которой выступают вектора и матрицы.

2.4 Деревья квадрантов

Деревья квадрантов (англ. — Quadtree) являются рекурсивной структурой данных, широко используемой для эффективного хранения и обработки разреженных матриц. Возможность представлять их компактным образом является ценным инструментом для оптимизации использования памяти.

Quadtree строится на основе деления матрицы смежности графа на четыре равные части (квадранта) и рекурсивного применения этого процесса деления ко всем четырем дочерним квадрантам. Каждый такой элемент может быть представлен в виде узла дерева, а в листьях хранится информация об объектах или данных, находящихся в соответствующем квадранте. Это позволяет эффективно представлять разреженные данные, так как области без объектов могут быть сохранены в упрощенной форме.

Аналогично представлению матриц с помощью Quadtree, вектора можно хранить в виде двоичного дерева (*англ. — Binary tree*), применяя тот же принцип последовательного рекурсивного разделения. Представление векторов и матриц таким образом удобно не только для оптимизации памяти, но и для внедрения параллельных вычислений.

На рисунке 2 представлена четверка узлов, образующих квадрант. Нумерацию и обозначение квадрантов принято вести слева направо, сверху вниз: (I) NW — North-West, (II) NE — North-East, (III) SW — South-West, (IV) SE — South-East.

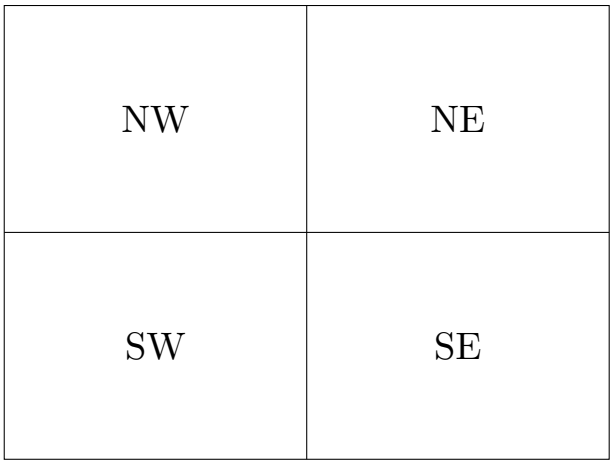

Рис. 1: Квадратная матрица, разделенная на квадранты

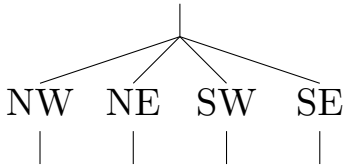

Рис. 2: Схематичное изображение одного из узлов дерева квадрантов и его потомков

На рисунке 3 продемонстрирована ситуация, в которой на некотором уровне квадрант имеет всего один значащий узел, остальные элементы матрицы являются незначащими (в нашем случае для них использовано обозначение `None`). Использование такого обозначения нулевых элементов обусловлено особенностями выбранного языка программирования, который будут рассмотрены далее.

2.5 Языковые особенности F#

`Option` — это тип-контейнер, который может либо содержать значение определенного типа, либо быть пустым (`None`). Такое представление незначащих элементов очень удобно по нескольким причинам.

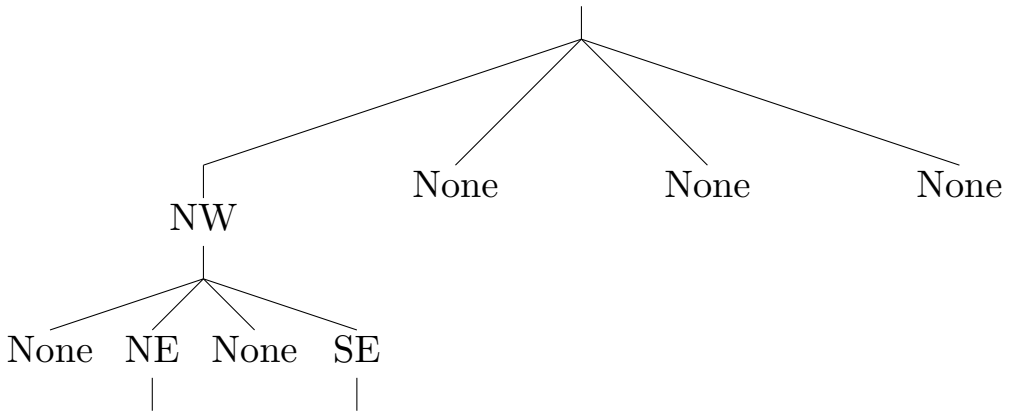

Рис. 3: Схематичное изображение нескольких уровней дерева с отсутствующими значениями в некоторых узлах

- **Естественное выражение отсутствия значения**
  Деревья квадрантов могут иметь узлы, которые не содержат никаких данных или не имеют потомков. Использование типа `Option`позволяет явно на это указать, делая код более читабельным.

- **Спосотавление по образцу (англ. — pattern-matching)**
  Эта особенность языка позволяет легко обрабатывать различные случаи, встречающиеся в узлах деревьев.

- **Безопасность типов**
  Использование `Option` обеспечивает статическую проверку типов. Например, компилятор предотвратит ошибки, связанные с попыткой обращения к значению там, где его нет.

Необходимо также затронуть такую особенность, использовавшуюся при реализации параллельной версии BFS, как *асинхронные* вычисления. Это подход к обработке данных, при котором задачи выполняются независимо друг от друга и без явной синхронизации. Вместо того чтобы ждать завершения одной задачи (англ. — Task) до перехода к следующей, асинхронные методы позволяют запускать несколько задач одновременно на разных потоках и обрабатывать результаты по мере их готовности. В `F#`такой метод вызывается с помощью ключевого слова `async`, в то время как тип `Async<'T>` представляет асинхронную

операцию.

3 Детали реализации

В этом разделе будут продемонстрированы шаги обхода в ширину с применением матрично-векторных операций и представлен пример параллельного алгоритма на языке `F#`.

3.1 Концепция алгоритма

Применение операции умножения вектора на матрицу, определенной в GraphBlas, является переходом от одних вершин графа к другим по инцидентным этим вершинам рёбрам. Как упоминалось выше, векторы представлены в виде Binary tree, а матрицы — в виде Quadtree. Рассмотрим конкретные шаги обхода в ширину в контексте линейной алгебры.

- **Шаг 1**
  Создается матрица смежности (*англ. — adjacency matrix*), соответствующая данному графу. В нашем случае по матрице смежности, представленной как список координат, строится дерево квадрантов. Значения в листьях заполняются значениями на соответствующих рёбрах графа.

- **Шаг 2**
  Задаются два вектора.

  1. **Фронт (*англ. — front*)**
     Вектор, в котором каждый индекс соответствует вершине графа, а значения указывают на текущие просматриваемые вершины.
  2. **Результирующий вектор**
     При инициализации совпадает с фронтом а в процессе работы алгоритма аккумулирует промежуточные результаты (например, длину минимального пути).

- **Шаг 3**
  Выполняется умножение вектора состояний на матрицу смежно-

сти. В качестве поэлементных операций сложения и умножения используются логические `ИЛИ` и `И`. После каждого умножения на результирующий вектор применяется маска (*англ.* — *mask*), чтобы алгоритм не обрабатывал уже посещенные вершины. Результатом итерации является новый фронт и обновленный результирующий вектор.

Шаг 3 повторяется до тех пор, пока не будут достигнуты требуемые условия остановки. Например, все вершины станут исследованы или произойдет выполнение определенного условия.

3.2 Параллельные вычисления

Благодаря рекуррентной природе деревьев, являющихся в нашем случае внутренним представлением векторов и матриц, реализованный алгоритм естественным образом позволяет внедрить дополнительные вычислительные потоки. В листинге 1 представлен псевдокод с использованием абстрактных операций линейной алгебры. *Минимальными единицами* такой операции являются вектор размера `1×2` и матрица размера `2×2`, что на каждой итерации позволяет использовать до четырех дополнительных потоков для разделения вычислений между ними. Рисунок 4 демонстрирует операции, необходимые для вычисление результирующего вектора при перемножении минимальных единиц.

$$\begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} \\ a_{11} \times b_{12} + a_{12} \times b_{22} \end{bmatrix}$$

Рис. 4: Элементарная операция умножения вектора на матрицу

Элементы матрицы — поддеревья, поэтому вычисление элемента $a_{is} \times b_{sj}$ является рекурсивным вызовом на соответствующем поддереве. В представленной работе операция умножения над *минимальными единицами* разбивается на два потока: один из потоков отвечает за вычисление значения в первой строке результирующего вектора, а другой поток — за вычисление во второй. При необходимости увеличить

Листинг 1 Псевдокод параллельного алгоритма обхода в ширину с использованием методов линейной алгебры

```
let BFS parallelLevel startingVertices graph =

  let adjacencyMatrix = graph.AdjMtx

    let frontier = SparseVector startingVertices

    let result = SparseVector startingVertices

    let rec inner frontier visited counter =

        if frontier.IsEmpty then
            result
        else

            let newFrontier =
                MatrixAlgebra.vecByMtx parallelLevel frontier adjacencyMatrix
                |> SparseVector.Map2 parallelLevel fMask result

            let newResult =
                SparseVector.Map2 parallelLevel (fUpdateCount counter) result
    newFrontier

            inner newFrontier newResult (counter + 1u)

    inner frontier result 1u
```

глубину параллелизации на вход в BFS передаётся целый положительный параметр *parallelLevel*, величина которого уменьшается с каждым вызовом тела функции, в результате чего производятся дополнительные асинхронные вычисления на следующем уровне рекурсии. При значении параметра равным 0 выполняется последовательная версия BFS.

Дополнительные детали реализацииалгоритма можно найти в репозитории [2].

[2] Репозиторий с реализацией BFS в контексте линейной алгебры. Дата посещения: 23 мая 2023 г.

4 Подготовка тестового стенда

В этом разделе указаны характеристики оборудования, на котором проводились исследования. Также были выдвинуты необходимые гипотезы, проверке которых посвещены проводимые эксперименты, и обозначены используемые метрики и инструменты для фиксирования измерений. Отдельно прокомментирован выбор набора тестируемых данных.

4.1 Характеристики оборудования

Аппаратная конфигурация обладает приведенными ниже характеристиками.

Операционная система

```
Operating System: Ubuntu 22.04.2 LTS
```

CPU

```
Architecture:        x86_64
Model name:          AMD Ryzen 5 4500U with Radeon Graphics
Thread(s) per core: 1
Core(s) per socket: 6
```

RAM

```
Total (MB): 9351
```

4.2 Исследовательские гипотезы

Анализ поставленных задач позволил выдвинуть следующие гипотезы.

Гипотеза №1

Ожидается, что в параллельной версии алгоритма обхода в ширину производительность будет значительно превышать последовательную

версию на сильно разреженных неориентированных графах, потому что в таких графах большинство вершин имеют небольшую степень, что позволит эффективно распределить работу между потоками и уменьшить накладные расходы на синхронизацию. Таким образом, параллельная версия должна продемонстрировать ощутимое ускорение.

Гипотеза №2

Предполагается существование оптимального количества потоков в параллельной версии алгоритма, которое приведет к наибольшему выигрышу в производительности за счет эффективного использования доступных ресурсов вычислительной системы. Его значение ожидается близким к доступному количеству логических ядер используемого процессора.

4.3 Использованные метрики

Для исследования *Гипотезы №1* измерено ускорение (*англ. — Speedup*) параллельной версии алгоритма относительно последовательной со следующим набором контролируемых параметров:

- количество вершин в графе;
- плотность графа;
- количество используемых потоков.

Speedup (S) вычисляется по формуле:

$$\mathrm{S} = \frac{T_{old}}{T_{new}}, \tag{2}$$

где T_{old} - время работы последовательной версии алгоритма, T_{new} - время работы параллельной версии алгоритма при заданных параметрах.

Для поиска оптимального значения, обозначенного в *Гипотезе №2*, проанализировано ускорение параллельной версии алгоритма на сильно разреженных графах с использованием разного количества потоков: 1, 2,

4, 8, 16. Выбор таких величин обусловлен ограничениями используемого оборудования.

4.4 Инструменты для измерений

Все замеры выполнены с использованием библиотеки для измерения производительности `BenchmarkDotNet v0.13.4` [3], разрабатываемой и поддерживаемой для платформы `.NET`.

4.5 Набор данных

Для фиксации исследуемых величин выбраны 11 различных разреженных квадратных матриц из коллекции университета Флориды [3]. Плотные матрицы и разреженные матрицы с крайне малым количеством вершин (< 100) созданы искусственно, потому что такие синтетические данные не нарушают чистоту эксперимента, так как плотные графы редко встречаются на практике, а очень маленькие размеры представляют лишь формальный интерес, и ручная регулировка количества вершин и плотности позволит точнее определить влияние этих параметров на производительность.

Популярный набор для тестирования плотных графов `MIVIA Large Dense Graphs` [4] является синтетическим, однако предоставляется в неудобном для данной работы формате, поэтому файлы графов с малым количеством вершин в количестве 10 штук были созданы согласно стандарту файлов `Matrix Market` [5] с помощью библиотеки `NetworkX` [6] для языка `Python`. В листинге 2 продемонстрирован использованный для генерации графов код.

Информация о разреженных матрицах представлена в таблице 1. Для обозначения числа ненулевых элементов используется аббревиатура *Nnz*. Приведено официальное (при наличии) название матрицы, количество

[3] Библиотека .NET для замеров производительности. Дата посещения: 23 мая 2023 г.

[4] Ресурс исследовательной лаборатории Mivia университета Solerno. Дата посещения: 23 мая 2023 г.

[5] Главная страница проекта Matrix Market. Дата посещения: 23 мая 2023 г.

[6] Библиотека анализа сложных систем для языка Python. Дата посещения: 23 мая 2023 г.

строк, количество ненулевых элементов, отношение ненулевых элементов к числу всех возможных элементов (плотность).

Матрица M	Количество строк R	Nnz	$Nnz/R^2, \%$
10_0.1	10	2	4.4
20_0.1	20	25	13.1
30_0.1	30	47	10.8
40_0.1	40	90	11.5
50_0.1	50	114	9.3
60_0.1	60	181	10.2
70_0.1	70	247	10.2
80_0.1	80	309	9.7
90_0.1	90	377	9.4
100_0.1	100	515	8.4
yalea_10nn	165	1,134	8.4
glass_10nn	214	1,493	6.6
usair97	332	2,126	3.9
olivetti_norm_10nn	400	2,828	3.6
spectro_10nn	531	3,711	2.6
vehicle_10nn	846	5,447	1.5
g51	1,000	5,909	1.2
yeast_30nn	1,484	31,175	2.8
dataset16mfeatkarhunen_10nn	2,000	13,834	0.7
misknowledgemap	2,427	28,511	0.9
mycielskian12	3,071	203,600	4.3

Таблица 1: Разреженные матричные данные

4.6 Эксперимент

Результаты измерений ускорения относительно уровней параллельности представлены в таблицах 4, 5, 6, 7, 8, 9. Каждый уровень соответствует 2 потокам. Так, на уровне 2 в асинхронных вычислениях задействованы 4 потока, на уровне 3 — 8 и т.д.

Проверка инструмента измерения производительности

Измерения, полученные с помощью `BenchmarkDotNet` были проверены на нормальность с помощью библиотек языка `Python`: `SciPy v1.10.1`[7], `NumPy v1.24.3`[8] и `Matplotlib v3.7.1`[9]. Для этого отдельно выполнены 100 измерений производительности алгоритма на графе `mycielskian12` с параметром `parallelLevl` = 3. Граф выбран как предположительно обладающий достаточным для эффективного применения асинхронных вычислений количества вершин. Значение `parallelLevel` выбиралось как предположительно оптимальное. Результаты измерений зафиксированы в таблице 2.

Время 1 (мс)	Время 2 (мс)	Время 3 (мс)	Время 4 (мс)	Время 5 (мс)	Время 6 (мс)	Время 7 (мс)	Время 8 (мс)	Время 9 (мс)	Время 10 (мс)
130.273	193.6342	816.4792	147.604	154.5787	144.4763	168.322	157.4245	165.3245	174.7072
130.0219	207.0966	169.0884	156.9507	143.6333	159.6349	148.3321	141.0419	123.8125	103.7556
155.7681	173.8194	141.1922	144.1792	127.7025	128.5758	221.2754	164.8696	163.6834	122.9248
192.0396	143.0681	121.7847	141.0921	138.0951	141.7283	132.8897	186.3074	186.3662	135.5439
144.3839	131.7593	128.9409	119.6813	144.5015	153.9668	231.7988	194.2572	152.5939	175.0124
213.1968	152.7859	166.767	179.1278	154.8963	136.6072	211.6087	146.9048	179.1302	178.0791
134.6695	193.0113	170.6718	158.6846	195.0692	140.8904	181.9895	203.541	139.4999	154.5966
163.0994	149.5508	139.4742	252.4698	222.6095	160.4422	160.3885	121.6504	145.7919	215.9388
151.9468	133.891	146.2433	170.738	137.8877	176.1576	159.5394	156.7406	194.014	166.329
155.0427	152.6318	124.2822	183.2932	135.5936	180.9422	159.2848	152.0111	117.932	203.1799

Таблица 2: Измерения, полученные во время работы параллельной версии алгоритма на графе mycielskian12, parallelLevl = 3

С помощью функций `scipy.stats.shapiro()` и `scipy.stats.normaltest()` проверялось соответствие полученных величин нормальному распределению. Значение `pvalue` в результате соответствовало 0.15 (Шапиро) и 0.13 (Пирсон), что удовлетворяет критерию нормальности. По данным в таблице 2 построена гистограмма 3. Вид гистограммы соответствует нормальному распределению.

Анализ результатов

Ниже даны ответы на вопросы исследования.

RQ1: Результаты ускорения, зафиксированного на графиках 4, 5 показали, что для *разреженных* графов с размером вершин менее 100

[7]Адрес в интернете https://scipy.org/ (дата доступа: 24 мая 2023 г.).

[8]Адрес в интернетеhttps://numpy.org/ (дата доступа: 24 мая 2023 г.).

[9]Адрес в интернетеhttps://matplotlib.org/ (дата доступа: 24 мая 2023 г.).

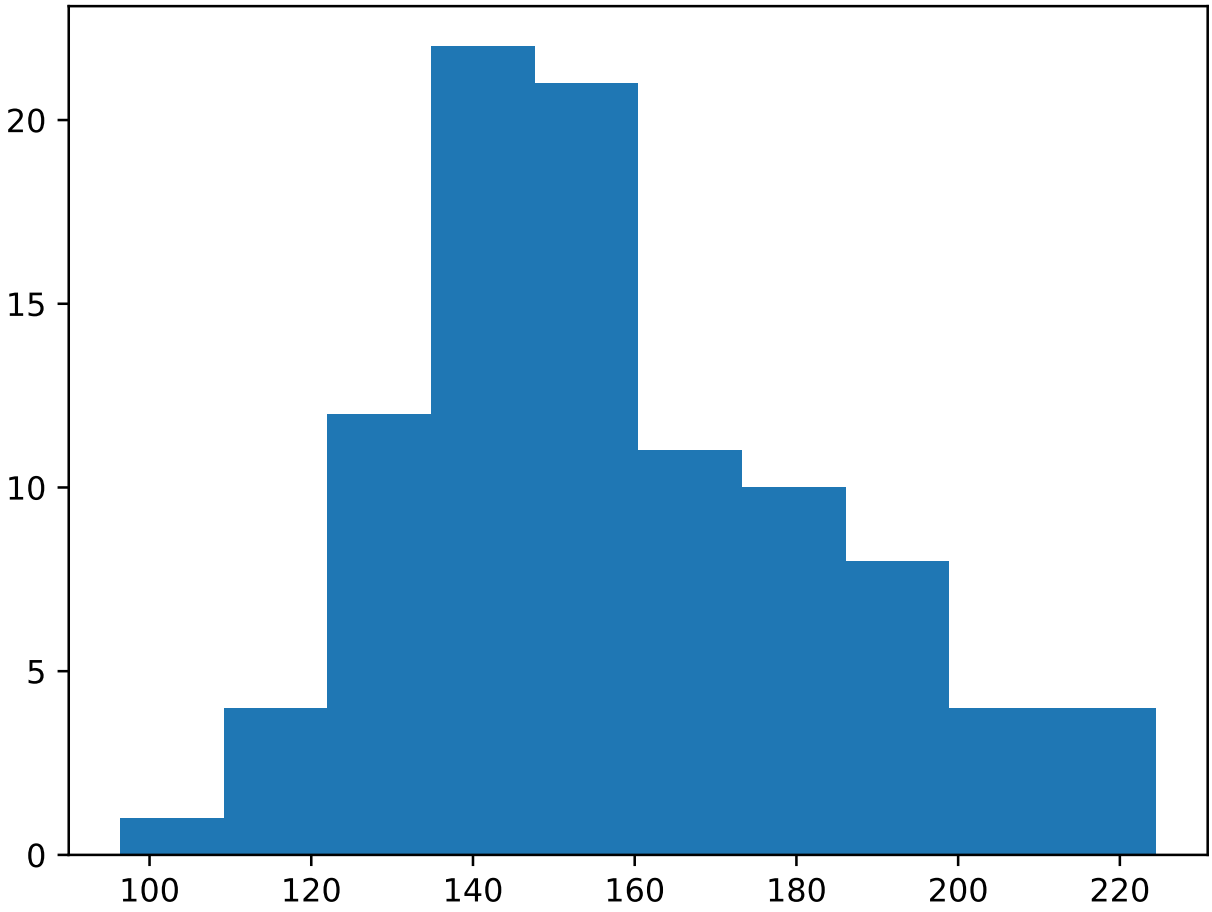

Таблица 3: Гистограмма распределения значений, полученных, при проведении проверки инструмента BenchmarkDotNet

выгоднее использовать последовательную реализацию алгоритма обхода в ширину. Причем параллельная версия показывает ускорение на 4 потоках, начиная с графа со 100 вершинами. Такие значения объясняются тем, что вершин и рёбер хоть и достаточно, чтобы увидеть рост производительности при параллельных вычислениях, их не хватает, для получения преимущества перед накладными расходами, возникающими при последующем увеличении количества используемых логических процессоров.

Результаты в 6, 7 демонстрируют рост ускорения с увеличением количества вершин, тем самым преимущество параллельной версии для разреженных графов становится очевидным.

Полученные выводы согласуются с поставленной *Гипотезой №1*.

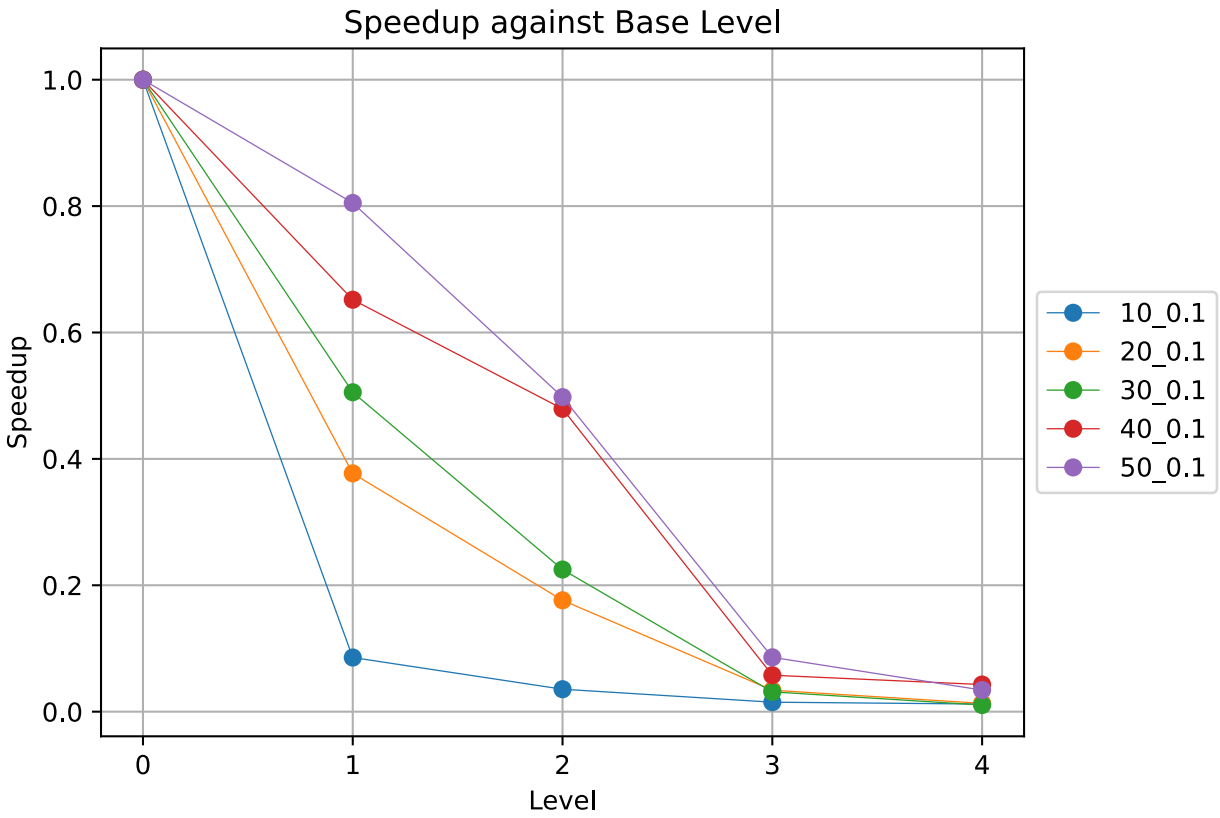

Таблица 4: Ускорение параллельной версии BFS относительно последовательной для разреженных графов с количеством вершин в диапазоне 10 — 50 из доступного набора графов

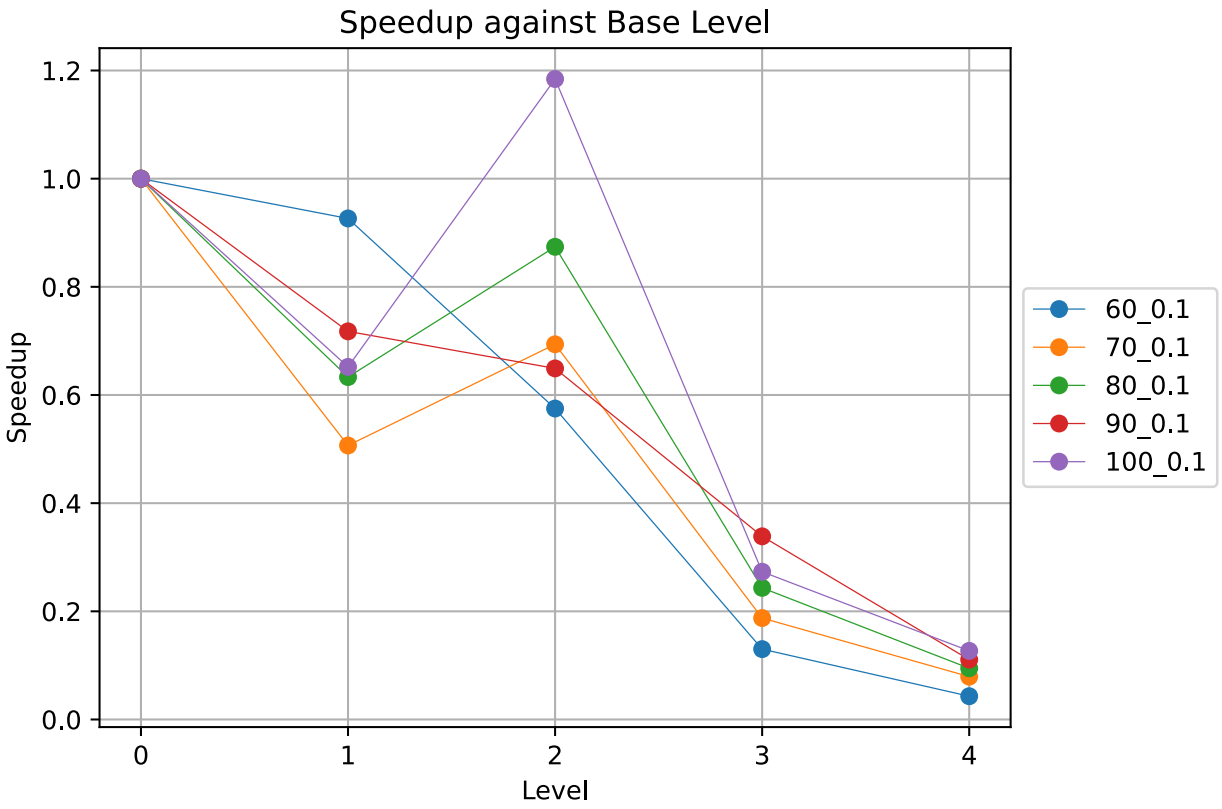

Таблица 5: Ускорение параллельной версии BFS относительно последовательной для разреженных графов с количеством вершин в диапазоне 60 — 100 из доступного набора графов

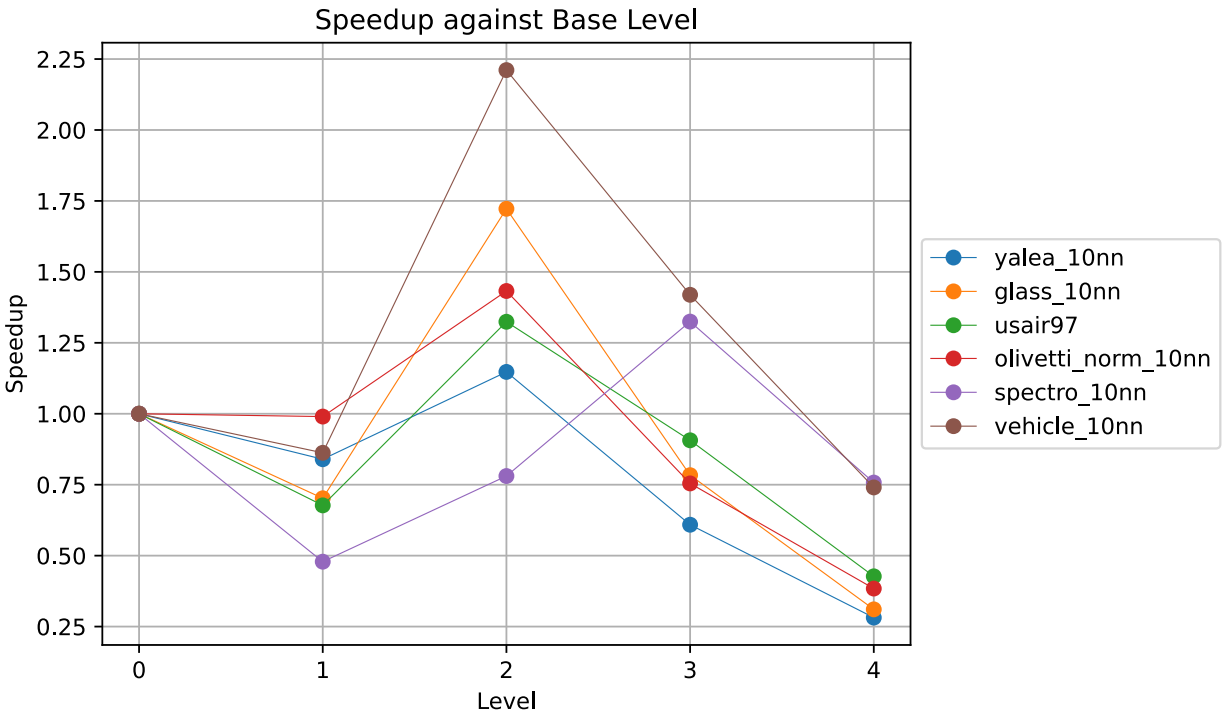

Таблица 6: Ускорение параллельной версии BFS относительно последовательной для разреженных графов с количеством вершин в диапазоне 165 — 846 из доступного набора

RQ2: На графиках 6, 7 выделяется оптимальное количество потоков, равное 8, дающее максимальный прирост производительности при использовании параллельной версии обхода в ширину. Полученное значение является верхней границей количества ядер используемого процессора. Отмечено, что при росте количества вершин, это оптимальное значение может смещаться в сторону большего количества потоков. Например, алгоритм на графе `mycielskian12` показал увеличение ускорения при использовании 16 потоков, что вызвано особенностями архитектуры процессора и оптимизацией, проводимой операционной системой. Предполагается, что для процессоров с большим количеством физических и логических ядер лучший параметр используемых потоков находится на уровнях выше (например, 4, 5 и т.д.).

Наблюдения об оптимальном значении согласуется с выставленной *Гипотезой №2*.

Отдельно измерено влияние плотности на количество вершин, при

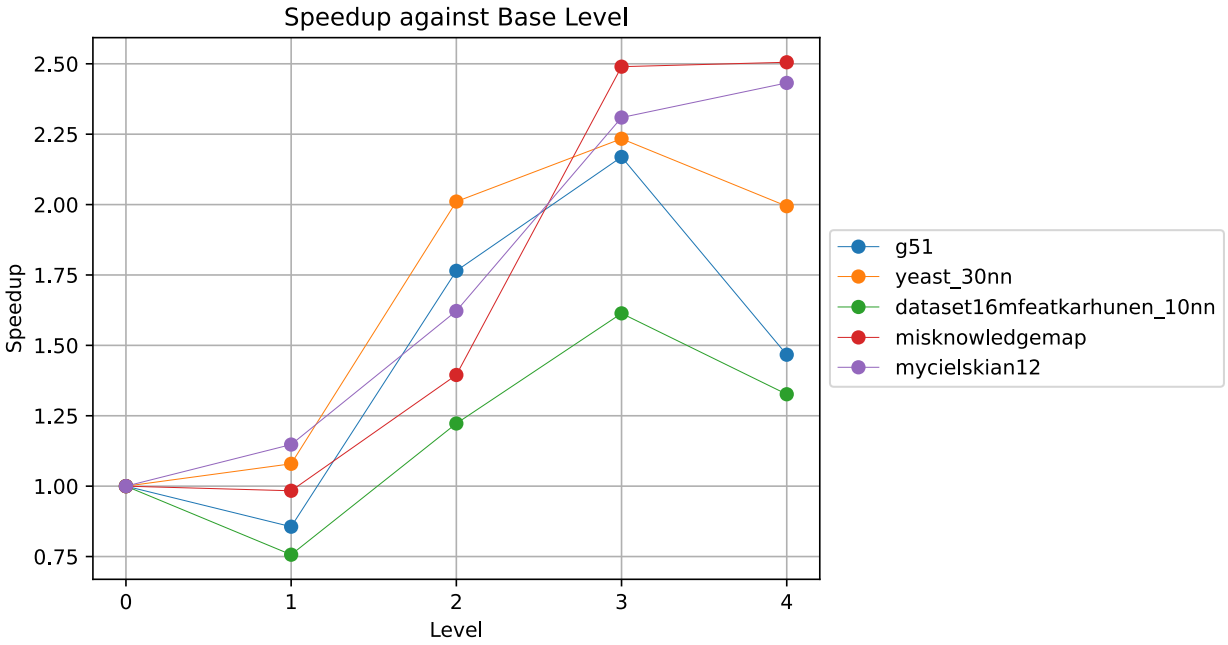

Таблица 7: Ускорение параллельной версии BFS относительно последовательной для разреженных графов с количеством вершин в диапазоне 1000 — 3071 из доступного набора

котором параллельная версия BFS становится эффективнее. Рассматривая параметр плотности в значении 0.7 и постепенно снижая его для графов с количеством вершин не более 100, мы наблюдали смещение критического значения размера в сторону уменьшения. Результаты для $density = 0.7$ представлены на графике 8. Параллельная реализация показала ускорение для размеров 100, 90 и 80, но оказалась медленнее для плотного графа с количеством вершин, равным 70.

На графике 9 параллельный алгоритм демонстрирует ускорение для графа с параметром плотности 0.6 на 80 вершинах, но работает не лучше, чем последовательная версия на 70 вершинах. Причем для 2 потоков накладные расходы на их обслуживание превышают выигрыш производительности, поэтому наблюдается общее снижение ускорения, но для 4 потоков удаётся распределить задачи так, что не происходит ни падения, ни роста эффективности.

Анализ 8 и 9 позволил заключить, что плотность графа уменьшает нижнюю границу значения, при котором параллельная версия становится выгоднее последовательной. То есть прирост ускорения происходит

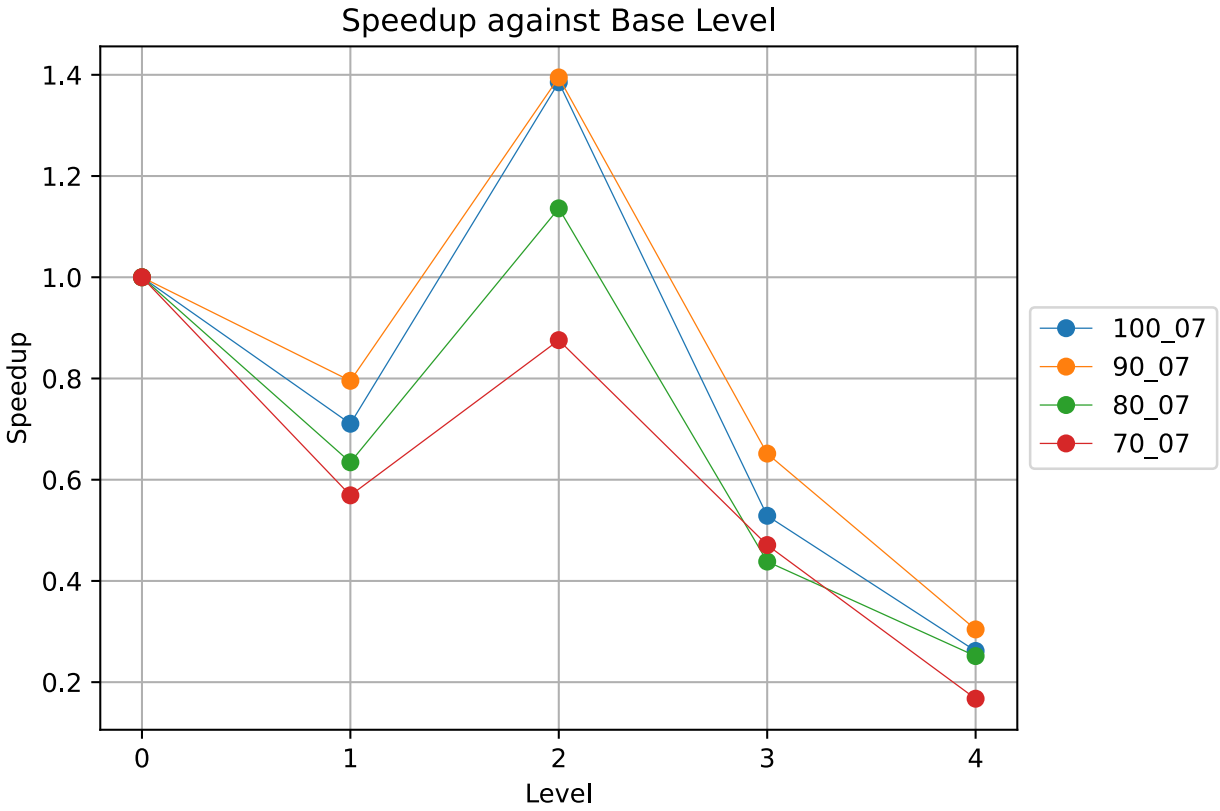

Таблица 8: Ускорение параллельной версии BFS относительно последовательной графов плотности 0.7 с количеством вершин 70, 80, 90, 100

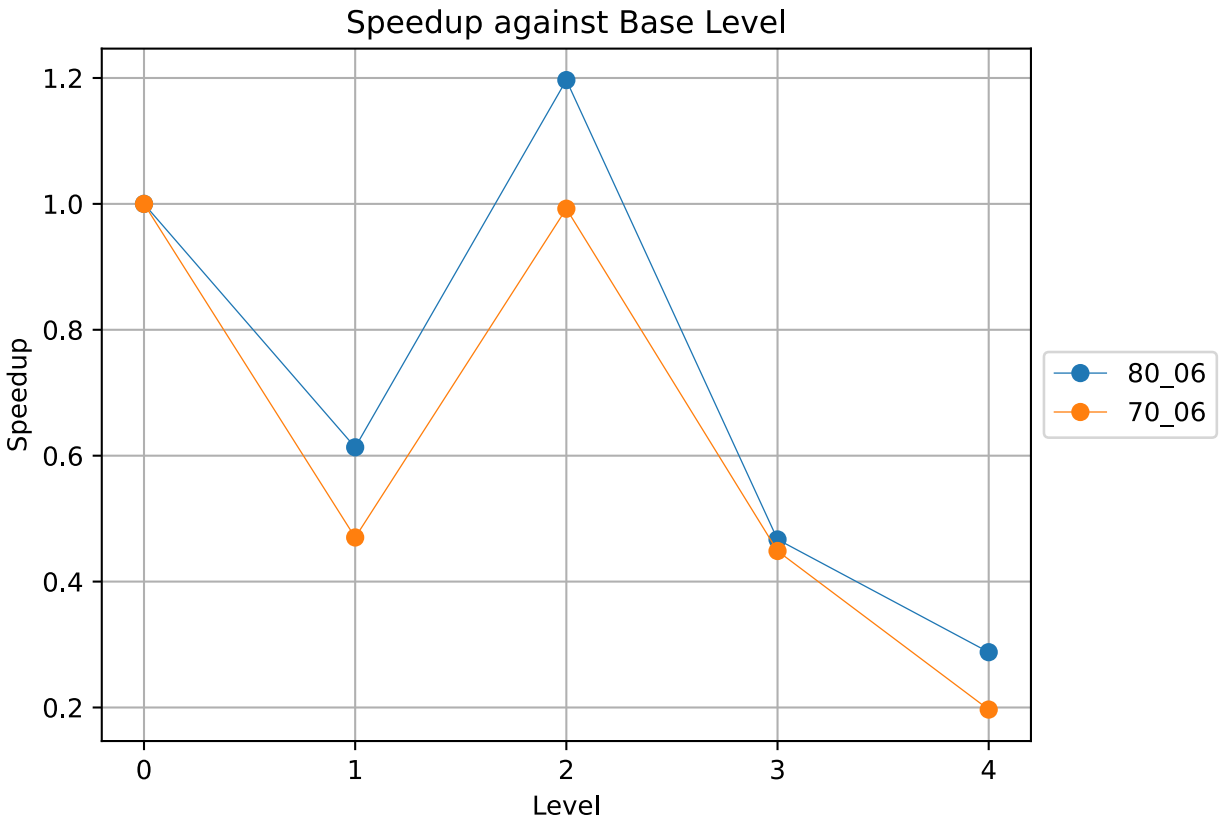

Таблица 9: Ускорение параллельной версии BFS относительно последовательной для графов плотности 0.6 с количеством вершин 70, 80

для плотных структур на меньшем количестве вершин, чем для разреженных. Это объясняется тем, что с ростом плотности увеличивается количество ребёр, что позволяет распределить вычислительные задачи между используемым потоками более эффективно, так как умножаются вектора и матрицы с достаточным для этого количеством значащих элементов. Для графов с количеством вершин более 100 и параллельная, и последовательная версии будут включать больше операций, обрабатываемых процессором, поэтому негативных последствий для эффективности параллельной версии не наблюдается.

Заключение

В рамках выполнения данной работы проведено экспериментальное исследование. Получены следующие результаты.

- Реализована параллельная и последовательная версии алгоритма обхода в ширину с использованием абстрактных операций линейной алгебры над матрицами и векторами, представленными в памяти как деревья.
- Оценено влияние плотности и количества вершин графа на эффективность параллельной версии обхода в ширину. В частности, установлены те параметы, при которых параллельная версия оказывается лучше последовательной.
- Найдено оптимально значение количества потоков, позволяющее получить максимальное ускорение.

Продолжение исследование может включать оценку ускорения параллельной версии обхода в ширину без использования линейной алгебры и представления матрицы как Quadtree. Также интерес представляет выявление зависимости связности графа и выбора стартовой вершины (или вершин) на итоговую производительность алгоритма.

Список литературы

[1] Mathematical foundations of the GraphBLAS / J. Kepner, P. Aaltonen, D. Bader et al. // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — 2016. — P. 1–9.

[2] Scalable Graph Exploration on Multicore Processors / Virat Agarwal, Fabrizio Petrini, Davide Pasetto, David A. Bader // SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. — 2010. — P. 1–11.

[3] T. Davis. The SuiteSparse Matrix Collection (the University of Florida Sparse Matrix Collection). — 2020. — URL: https://sparse.tamu.edu (дата обращения: 15.05.2023).

[4] Xia Yinglong, Prasanna V. Topologically adaptive parallel breadth-first search on multicore processors. — 2009. — 01.

Листинг 2 Пример кода для генерации графов на языке Python с использованием библиотеки `NetworkX`

```
import networkx
import numpy
import scipy.io  # for mmread() and mmwrite()
import io  # Use BytesIO as a stand-in for a Python file object

# Specify graph parameters
mininumSize = 10
maximumSize = 100+1
step = 10

# Specify density list
p_list = [0.1]

for n in range(mininumSize, maximumSize, step):
    for p in p_list:

        fh = io.BytesIO()

        # Generate random graph
        G = networkx.gnp_random_graph(n, p)

        # Choose random label on existing edge
        for u, v in G.edges:
           G[u][v]['weight'] = numpy.random.random()

        # Convert to sparse matrix representation
        sparseArray = networkx.to_scipy_sparse_array(G)

        # Write output to a file
        scipy.io.mmwrite(fh, sparseArray)
        with open("graphs/" + str(n)+"_" + str(p) + ".mtx", 'w', encoding='utf-8') as file:
            file.write(fh.getvalue().decode('utf-8'))
```