Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б07-мм

Экспериментальное исследование производительности алгоритма обхода графа в ширину

Савельева Полина Андреевна

Отчёт по учебной практике
в форме «Эксперимент»

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

Санкт-Петербург
2023

Оглавление

Введение

Графы, представляющие набор объектов и их отношений, используются для моделирования сложных систем в различных сферах жизни от математики и биологии до социологии и лингвистики. При работе с графовыми моделями анализ и эффективное хранение больших объёмов данных являются серьёзными проблемами. Кроме того, во многих прикладных задачах необходимо оперировать *разреженными* структурами [2] [4] [8].

Приведём простой пример: рассмотрим граф социальной сети, пользователи которого представлены в виде вершин, а дружеские отношения между ними — в виде рёбер. В такой модели каждый пользователь имеет небольшую часть из всех возможных связей. Следовательно, матрицу смежности искомого графа можно назвать разреженной — значительная часть её ячеек будет заполнена *нулями*.

Важно учесть, что эффективность алгоритма, работающего с разреженными данными, существенно зависит от способа их хранения. Так, в рассмотренном примере представление матрицы смежности графа в виде двумерного массива будет крайне неэффективным не только с точки зрения затрат памяти, но и поиска информации.

Процесс поиска информации в графе представляет собой обход всех вершин и рёбер. Два наиболее известных алгоритма поиска — это обход в ширину *Breadth-First Search* (BFS) и обход в глубину Depth-First Search (DFS). В данной работе алгоритм DFS будет упомянут лишь косвенно, но нельзя не отметить его основное отличие от BFS — обход в ширину последовательно рассматривает смежные вершины, а обход в глубину — вершины, расположенные по одному пути. Благодаря этому различию в BFS (в отличие от DFS, имеющего немногочисленные линейно-алгебраические формулировки [10]) возможно использование матрично-векторных операций (SpMSpV) для эффективной работы с разреженными данными [12].

Операции над векторами и матрицами могут быть *распараллелены* вследствие разделения данных на части и обработки фрагментов в раз-

ных *потоках*. Это применимо, например, в масштабных вычислениях, где параллельная реализация может ускорить общее время работы [7].

Для оценки эффективности алгоритма и его соответствия ожидаемым результатам необходимо провести *экспериментальное исследование*. Более того, экспериментальный анализ позволяет оценить производительность метода на реальных данных и его пригодность для практического применения, способствует пониманию того, как различные аппаратные конфигурации влияют на производительность, и вместе с тем результаты исследования помогают при оптимизации конкретных сценариев работы алгоритма, в которых входные данные могут сильно различаться по размеру и плотности.

Поэтому необходимо не только реализовать алгоритм, но и провести экспериментальное исследование его поведения в условиях, близких к реальным.

1. Постановка задачи

Целью данной работы является анализ производительности алгоритма обхода графа в ширину с использованием линейной алгебры и многопоточности. Для её выполнения были поставлены следующие задачи.

- Реализация разреженных матриц и векторов, а также операций сложения и умножения над ними.
- Реализация алгоритма обхода графа в ширину с использованием линейной алгебры.
- Реализация параллельной версии матрично-векторных операций.
- Экспериментальное исследование алгоритма обхода в ширину с использованием параллельной версии матрично-векторных операций.

2. Обзор предметной области

В данном разделе представлен обзор некоторых существующих методов и концепций, связанных с графами. В частности, используемая терминология, способы представление разреженных структур, алгоритм обхода графа в ширину и параллельные вычисления.

2.1. Терминология

Определение 1. *Простой неориентированный граф $\mathcal{G}$ — это упорядоченная пара конечных множеств V и E таких, что $\mathcal{G} = \langle V, E \rangle$, V — непустое множество, элементы которого называются вершинами, и E — множество неупорядоченных пар вершин, называемых рёбрами.*

Определение 2. *Простой ориентированный граф $\mathcal{G}$ — это упорядоченная пара конечных множеств V и E таких, что $\mathcal{G} = \langle V, E \rangle$, V — непустое конечное множество, элементы которого называются вершинами, и $E \subseteq V \times V$ — множество упорядоченных пар вершин, называемых рёбрами.*

Определение 3. *Помеченный ориентированный граф — это упорядоченная пара $\langle \mathcal{G}, \mu \rangle$, где $\mathcal{G}$ — простой ориентированный граф, и отображение $\mu : E \to L$, сопоставляющее каждому ребру $e \in E$ метку (вес) из множества L. Помеченный неориентированный граф определяется аналогично.*

Определение 4. *Весовая матрица графа $\mathcal{G}$ — это квадратная матрица $A = (a_{ij})$ порядка n, где n — количество вершин графа $\mathcal{G}$, а элемент a_{ij} равен весу ребра, соединяющего вершины i и j, или значению, указывающему на его отсутствие.*

Определение 5. *Матрица смежности графа $\mathcal{G}$ — это квадратная матрица $A = (a_{ij})$ порядка n, где n — количество вершин графа $\mathcal{G}$, а элемент a_{ij} равен 1, если между вершинами i и j есть ребро, или 0, если ребра нет.*

Замечание. *Для удобства под матрицей смежности ориентированного графа иногда подразумевают весовую, принимая вес между вершинами за 1, а значение, указывающее на отсутствие ребра, за 0.*

Определение 6. *Структура данных дерево — это иерархическая структура данных, представляющая набор связанных элементов — узлов. В каждом дереве существует ровно один корневой узел, имеющий от 0 до n узлов-потомков (которые аналогично имеют от 0 до n потомков) и ни одного узла-предка. А также конечное число узлов, не имеющих ни одного потомка, называемых листьями.*

Определение 7. *Бинарное дерево — это дерево, каждый узел которого, за исключением листьев, имеет от 0 до 2 узлов-потомков.*

2.2. Представление разреженных структур

Ниже представлены несколько существующих форматов хранения разреженных матриц.

Наивный способ представления разреженных структур предполагает хранение всех элементов, включая нулевые, указывающие на отсутствие связей или отношений. Этот подход требует больших затрат памяти и на практике малоэффективен.

Координатное представление (*COO*) использует тройки (i, j, *value*) для представления ненулевых элементов разреженной структуры. Параметры (i, j) указывают на позицию элемента в матрице, а *value* — на его значение. На практике используют три неупорядоченных массива (один для каждой координаты тройки) и скаляр, отвечающий за общее количество непустых ячеек [11]. Такая модель используется для хранения небольших разреженных структур и может быть неэффективна при работе с большими данными. В частности, из-за необходимости перебора большого количества троек при поиске, и трудностей с добавлением новых элементов.

Форматы хранения *Compressed Sparse Row* (*CSR*) и *Compressed Sparse Column* (*CSC*) не основаны на конкретном свойстве и могут быть использованы для хранения любой разреженной матрицы [11]. Исходные

данные распределяются на несколько массивов — один, содержащий значения ненулевых элементов, один для хранения индексов строк или столбцов (в CSR по строкам, а в CSC по столбцам) соответствующих элементов и один массив-указатель на первое ненулевое вхождение каждой строки или столбца. Аналогично координатному формату вставка или удаление элементов в CSR/CSC может потребовать перестройки всей структуры. Кроме того, в них сложнее обеспечить эффективное параллельное выполнение операций [6].

Дерево квадрантов (Quadtree) — это структура данных, часто используемая для представления разреженных двумерных данных. Quadtree разбивается на четыре части (реже на любое другое количество частей), называемых *квадрантами*, где каждый квадрант может быть разбит на четыре подквадранта и т. д. Если в одной области наблюдаются только элементы одного типа (например, нули в разреженных матрицах), дерево «обрезают» до одного листа, содержащего информацию об узлах, расположенных на нижних уровнях. Из-за своей рекуррентной природы дерево квадрантов удобно использовать для параллельных вычислений.

2.3. Алгоритм обхода графа в ширину

Ниже приведено неформальное описание работы BFS.

1. Инициализировать очередь — структуру данных для хранения вершин, которые необходимо посетить — и пустой массив для хранения посещённых вершин. Добавить вершины, с которых начинается поиск, в очередь.

2. Удалить первую вершину из очереди. Пометить эту вершину как посещённую.

3. Добавить вершины, соединённые ребром с текущей, в очередь.

4. Повторить пункты 2–3, пока очередь не окажется пустой.

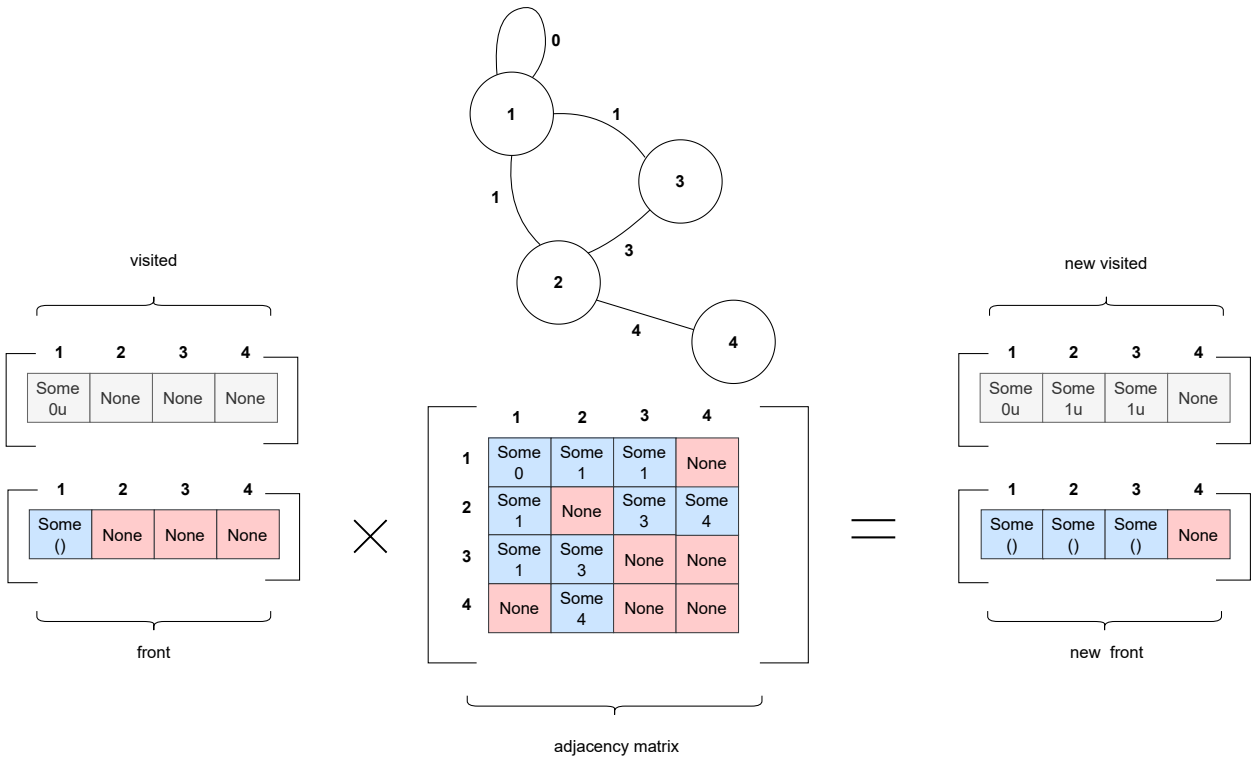

Рис. 1: Шаги 3–4 BFS с использованием линейной алгебры

Как было упомянуто раннее, линейная алгебра может быть полезна при реализации некоторых этапов BFS, например, для эффективного представления графа или выполнения операций над матрицами и векторами внутри алгоритма. Ниже приведена реализация шагов 1–4 с использованием векторно-матричных операций.

1. Инициализировать матрицу смежности графа (`adjacency matrix`), вектор для хранения посещённых вершин (`visited`) и вектор схожий с очередью, называемый фронтом (`front`), в котором хранятся вершины, находящиеся на текущем уровне обхода. Добавить вершины, с которых начинается поиск, в `visited`.

2. Умножить `front` на `adjacency matrix` и получить новый фронт (`new front`), добавив, таким образом, все вершины, соединённые с текущей.

3. Сложить `new front` с `visited` и получить новый вектор (`mask`) так, что значения из нового фронта сохраняются, только когда они не были помечены в `visited` (это действие избавит нас от

многократного посещения одной и той же вершины). Обновить `visited` (`new visited`), используя полученный вектор и векторные операции.

4. Повторить шаги 2–3, пока все вершины не будут помечены как посещённые.

На рис. 1 представлен пример работы алгоритма обхода графа в ширину с использованием линейной алгебры.

2.4. Параллельные вычисления и многопоточность

Большинство несложных программ с большой вероятностью выполняются в одном *потоке*. Другими словами, в любой момент времени только один оператор находится в процессе выполнения; и если поток блокируется, то вся работа программы останавливается [9].

Использование *ядер* центрального процессора может уменьшить общее время простоя и повысить эффективность программы. Как правило, одно ядро за раз обслуживает только один поток. Однако, даже процессор одноядерной машины, быстро переключаясь между задачами, способен создавать иллюзию их одновременного выполнения. В связи с этим, когда несколько потоков, работающих на одном ядре, могут «чередоваться», эффективнее использовать больше потоков, чем имеющихся ядер [5].

При реализации многопоточности, стоит обратить внимание на следующие особенности.

1. *Накладные расходы на синхронизацию.* Во избежание некорректной работы при использовании нескольких потоков необходимо обеспечить их синхронизацию. Координация, в свою очередь, вызывает накладные расходы, что не лучшим образом влияет на производительность.

2. *Конкуренция за ресурсы.* Когда несколько потоков пытаются получить доступ к общим ресурсам, возникает конкуренция, замедляющая работу программы.

3. *Накладные расходы на создание.* Создание потоков и управление ими требуют дополнительных ресурсов.

4. *Зависимость от алгоритмов и данных.* Некоторые алгоритмы (данные) не могут быть эффективно распараллелены из-за своей природы.

5. *Зависимость от аппаратного обеспечения.* Некоторые системы имеют ограниченное количество доступных ядер, что оказывает влияние на возможности параллельных вычислений.

Данные аспекты были учтены при проведении экспериментального исследования и анализе полученных результатов.

3. Архитектура решения

В данном разделе представлена реализация методов, упомянутых в обзорной части. В частности, алгоритм обхода в ширину с использованием параллельных вычислений и векторно-матричные операции.

3.1. Детали реализации SpMSpV

Листинг 1: Фрагмент функции сложения векторов, отвечающий за параллельную составляющую операции

```
let vectorAddition level plusOperation vector1 vector2 =
   ...
   let rec treesAddition parallelLevel tree1 tree2 =
      match tree1, tree2 with
      ...
      | BinTree.Node (x, y), BinTree.Node (z, w) ->
         if parallelLevel = 0u then
            let left = treesAddition 0u x z
            let right = treesAddition 0u y w
            if left = BinTree.None && right = BinTree.None then
               BinTree.None
            else
               BinTree.Node(left, right)
         else
            let tasks =
                [| async { return treesAddition (parallelLevel - 1u) x z };
                   async { return treesAddition (parallelLevel - 1u) y w } |]
            let results = tasks |> Async.Parallel |> Async.RunSynchronously
```

На платформе .NET существует поддержка параллельных вычислений с использованием асинхронных (т. е не блокирующих выполнение других процессов) выражений `async`, позволяющих разбить работу программы на небольшие подзадачи и запустить их параллельно на доступных вычислительных ядрах. Именно они были использованы при реализации векторно-матричных операций на языке F#[1].

[1]Полная реализация алгоритма обхода в ширину и матрично-векторных операций: https://github.com/PolinaSavelyeva/ProgrammingHomeworkFs.

Так как эффективность параллельной версии алгоритма, главным образом, зависит от количества используемых потоков, для векторно-матричных операций были добавлены параметры `level` — для сложения и `multiLevel`, `addLevel` — для умножения. Эти аргументы представляют собой *уровни распараллеливания*; к примеру, для ненулевого `level` функции `vectorAddition` (лист. 1) формируются две задачи, исполняемые параллельно, для ненулевого `level` — 1 формируются ещё две подзадачи и т. д.

Аналогично сложению используются `multiLevel`, `addLevel` в функции `multiplication`, однако в процессе формируются четыре подзадачи вместо двух (лист. 2). Это различие обусловлено строением вектора и матрицы: узлы дерева, представляющего вектор, имеют ровно два потомка, а узлы дерева, представляющие матрицу — четыре.

Листинг 2: Фрагмент функции умножения вектора и матрицы, отвечающий за параллельную составляющую операции

```
let multiplication multiLevel addLevel fPlus fMulti vector matrix =
   if parallelLevel = 0u then
   ...
   else
     let multiTasks =
       [| async { return Vector(multiTrees (parallelLevel - 1u) left first,
                   vector.Length) }
          async { return Vector(multiTrees (parallelLevel - 1u) right third,
                   vector.Length) }
          async { return Vector(multiTrees (parallelLevel - 1u) left second,
                   vector.Length) }
          async { return Vector(multiTrees (parallelLevel - 1u) right fourth,
                   vector.Length) } |]
     let multiResults = multiTasks |> Async.Parallel |> Async.RunSynchronously
     let leftTree1 = multiResults[0]
     ...
     let addTasks =
       [| async { return (vectorAddition addLevel fPlus leftTree1 leftTree2).
          Storage }; async { return (vectorAddition addLevel fPlus rightTree1
       rightTree2).Storage } |]
     let addResults = addTasks |> Async.Parallel |> Async.RunSynchronously
```

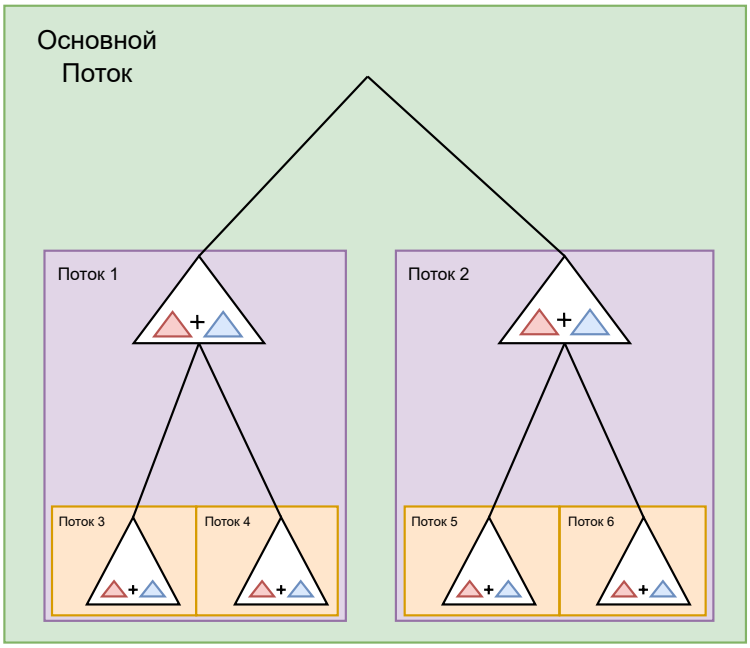

Рис. 2: Распределение потоков между узлами дерева, представляющего разреженный вектор, при сложении векторов (первый вектор отмечен синим цветом, второй красным)

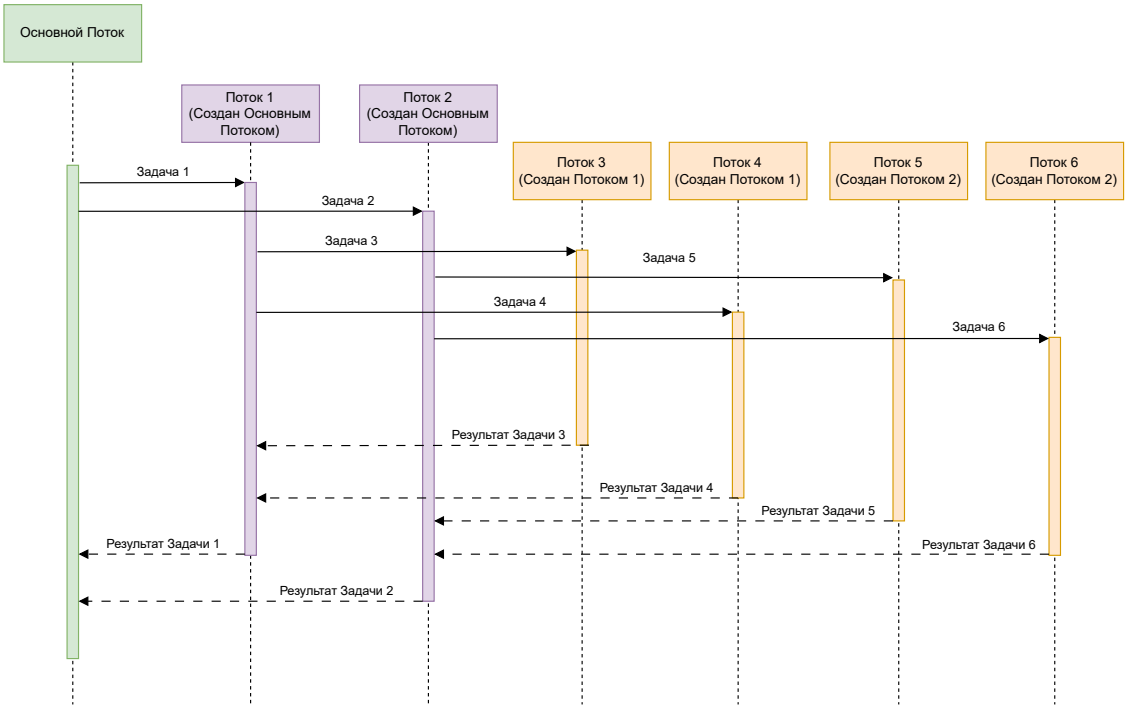

Рис. 3: Последовательность выполнения операции сложения векторов с использованием многопоточности

Визуализация распределения задач между узлами вектора при сложении представлена на рис. 2, а последовательность выполнения операции — на рис. 3.

3.2. Реализация BFS

Листинг 3: Функция, реализующая обход в ширину с применением матрично-векторных операций

```
1  let BFS multiLevel addMultiLevel addLevel startVertexList graph =
2     let rec inner (front: Vector<unit>) visited iterationNumber =
3        if front.IsEmpty then
4           visited
5        else
6           let newFront = multiplication multiLevel addMultiLevel fPlus fMulti
               front graph.AdjacencyMatrix
7           let front = vectorAddition addLevel fPlusMask newFront visited
8           let visited = vectorAddition addLevel (fPlusVisited iterationNumber)
               visited front
9           inner front visited (iterationNumber + 1u)
10    let front = Vector(startVertexList, graph.VerticesCount, ())
11    let visited = Vector(startVertexList, graph.VerticesCount, 0u)
12    inner front visited 1u
```

Алгоритм BFS на лист. 3 реализует шаги 1–4, упомянутые в обзоре. Неочевидным остаётся шаг получения маски; в действительности необходимые действия выполняются в строке 7 без инициализации самой `mask`. Функция `fPlusMask` (лист. 4) используется при сложении нового фронта и вектора посещённых вершин — на месте `front` получается ненулевое значение, только когда вершину предстоит посетить.

Листинг 4: Функция, имитирующая поведение маски для обновления вектора-фронта

```
1  let fPlusMask a b =
2      match a, b with
3      | Some _, Option.None -> Some()
4      | _ -> Option.None
```

4. Эксперимент

В данном разделе представлены результаты экспериментального исследования алгоритма, описанного в предыдущих главах. Основные задачи исследования — оценить производительность BFS с использованием линейной алгебры и многопоточности в условиях, близких к реальным, сравнить полученные показатели и ответить на вопросы, приведённые ниже.

RQ1: При каких параметрах графа выгоднее использовать параллельную версию алгоритма, а при каких последовательную?

RQ2: Использование какого количества потоков даёт наибольший выигрыш в производительности?

RQ3: Чем можно объяснить полученное количество потоков, дающих наибольший выигрыш в производительности?

RQ4: Почему в некоторых вполне разумных случаях на больших матрицах параллельность только замедляет?

4.1. Условия эксперимента

Эксперименты проводились на рабочей станции со следующими характеристиками.

- Центральный процессор: AMD Ryzen 5 5600X
- Количество ядер ЦПУ: 6 ядер, 12 потоков
- Операционная система: Windows 11 Pro 22H2

4.2. Набор данных

Для замеров производительности использовались неориентированные графы, т. е. графы с симметричной матрицей смежности. Примеры реальных данных взяты из коллекции университета Флорида [1]. Из

Таблица 1: Разреженные матричные данные

Название	Порядок	Ненулевые эл-ы	Плотность
HB/bcsstm03	112	72	0.57
HB/dwt_512	512	3,502	1.34
Gset/G52	1,000	11,832	1.18
Gset/G59	5,000	59,140	0.24
ML_Graph/kmnist_norm_10NN	10,000	156,932	0.15

коллекции было отобрано 5 матриц разного размера: маленькие, средние и большие, имеющие, кроме всего прочего, сравнительно малую плотность — не более 1.34%. В табл. 1 представлена информация об используемых данных: название, порядок, количество ненулевых элементов и плотность — отношение числа ненулевых элементов к общему числу, умноженное на 100.

В данной работе результат работы BFS не зависит от типа меток на рёбрах (в ответе учитывается только порядок посещения вершин), поэтому все матрицы, взятые в качестве экспериментальных данных, будут обрабатываться как матрицы Pattern-типа. Такой выбор обусловлен тем, что арифметические операции с одними примитивными типами могут быть более ресурсоёмкий по сравнению с другими (в данном случае в сравнении с типами Real и Integer).

4.3. Метрики

В исследовании для замеров производительности была использована библиотека BenchmarkDotNet v0.13.4[2]. В качестве метрик производительности выступает время, требуемое для выполнения операции. Предварительно совершался не учитывающийся в замерах прогревочный запуск, содержащий от 6 до 50 итераций (точное количество рассчитывается с помощью эвристики). Показатели времени усреднены по количеству целевых итераций; минимальное количество итераций 15, максимальное — 100 (точное количество рассчитывается с помощью эвристики). Стандартное отклонение (StdDev) составляет не более 10% от среднего значения (mean) для каждого отдельного эксперимента.

[2]Библиотека BenchmarkDotNet для замеров производительности: https://benchmarkdotnet.org/.

Таблица 2: Уровни, дающие наибольший выигрыш

Название	Порядок	multiLevel	addMultiLevel	addLevel	Ускорение
Gset/G52	1,000	2	0	2	3.45
Gset/G59	5,000	3	0	0	2.65
ML_Graph/kmnist_norm_10NN	10,000	2	0	4	3.21

4.4. Результаты

Столбчатые диаграммы, реализованные с помощью Matplotlib [3], на рисунках 5, 6, 7, 8 и 9 иллюстрируют результаты экспериментального исследования. На оси абсцисс отмечены уровни параллельности, представленные в формате (`multiLevel`, `addMultiLevel`, `addLevel`), на оси ординат — отношение среднего значения на базовом уровне (0,0,0) к среднему значению на текущем, далее именуемое *ускорением*. Красным цветом выделены столбцы, соответствующие наименьшему времени выполнения алгоритма. Горизонтальная пунктирная линия проведена на уровне базового значения и помогает визуально сравнить с ним другие показатели.

RQ1: При каких параметрах графа выгоднее использовать параллельную версию алгоритма, а при каких последовательную?

В трёх из пяти случаях, оказалось выгоднее использовать параллельную версию алгоритма, а именно в графах с матрицами смежности порядка 1000, 5000 и 10000. В графах с меньшими (относительно остальных) размерами 112 и 512 параллельные версии обхода в ширину не ускоряли, а, наоборот, замедляли процесс. Наилучший показатель (на уровне (0,0,1)) параллельной версии алгоритма на матрице порядка 112 примерно в 2.9 раз медленнее базового значения; на матрице порядка 512 — в 1.6 раз (на уровне (0,0,1)). Такой результат можно объяснить тем, что на накладные расходы, приведённые в обзоре, и параллельное выполнение суммарно тратиться больше времени, чем на работу однопоточной версии.

RQ2: Использование какого количества потоков даёт наибольший выигрыш в производительности?

Ответ на исследовательский вопрос RQ2 представлен в табл. 2.

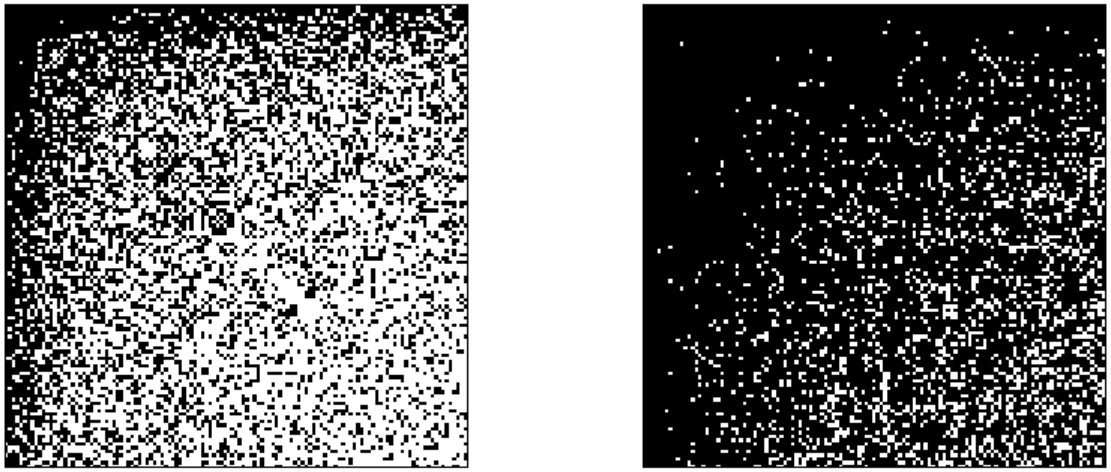

Рис. 4: Визуализация разреженности матрицы порядка 1000 (слева) и порядка 5000 (справа)

RQ3: Чем можно объяснить полученное количество потоков, дающих наибольший выигрыш в производительности?

Как было упомянуто ранее, на каждом уровне параллельности умножения создаётся по 4 задачи и по 2 — на каждом уровне сложения. Таким образом, имея параметры (2,0,2), мы получим суммарно 21 задачу для умножения и 13 задач для сложения; задачи умножения могут распределиться между 12 потоками не более чем по две на каждый, что может быть обосновано высказанным раннее утверждением об эффективности переключении процессора между задачами, находящимися в одном потоке. Действительно, вновь обращаясь к рис. 3 можно заметить, что после распределения задач основным потоком, он простаивает в ожидании ответа от потоков 1–2, которые аналогично ожидают потоки 3–6 — в это время их можно дополнительно нагрузить. Оптимальным является сценарий, при котором задачи, созданные на одном уровне параллельности (1 задача на уровне 0, 4 на уровне 1, 16 на уровне 2 и т.д.), выполнялись бы на максимально возможном количестве потоков. Таким образом, при параметре `multiLevel = 1` мы получим 4 задачи, которые можно распределить максимум между 4 потоками, а остальные 8 окажутся незанятыми, хотя потенциально на них могли выполняться вычисления.

Также отметим следующий факт — чем больше уровень параллельности, тем на более мелкие составляющие разделится основная задача. В случае, когда вычисление каждой подзадачи занимает меньше времени, чем накладные расходы (например, в области сильной разреженности), выгоднее остановить разбиение. Верно и обратное. Это утверждение может объяснить, почему в матрице размера 5000 выгоднее использовать 3, а не 2 уровня параллельности умножения —достаточно сравнить количество областей разреженности матрицы размера 1000 и матрицы размера 5000 на рис. 4.

Кроме того, на диаграммах 7, 8, 9 видно, что параллельная версия сложения не так эффективна, как параллельная версия умножения — достаточно сравнить показатели вида (0,0,x) или (0,x,0) и (x,0,0). Однако, оно может давать небольшой выигрыш, например в случае с матрицами 1000 и 10000.

RQ4: Почему в некоторых вполне разумных случаях на больших матрицах параллельность только замедляет?

Обобщив всё вышесказанное, можно объяснить факт замедления параллельной версии алгоритма на матрицах большого размера. Если попробовать разбить одну такую матрицу на меньшие составляющие, они всё ещё будут существенны по размеру, и, следовательно, синхронизировать данные будет затратно. Если же задать такой уровень параллельности, при котором матрица разделится до частей приемлемого для синхронизации размера, задачи, находящиеся на одном уровне, окажутся в положении ничуть не лучше последовательного выполнения — каждый поток будет ими перегружен.

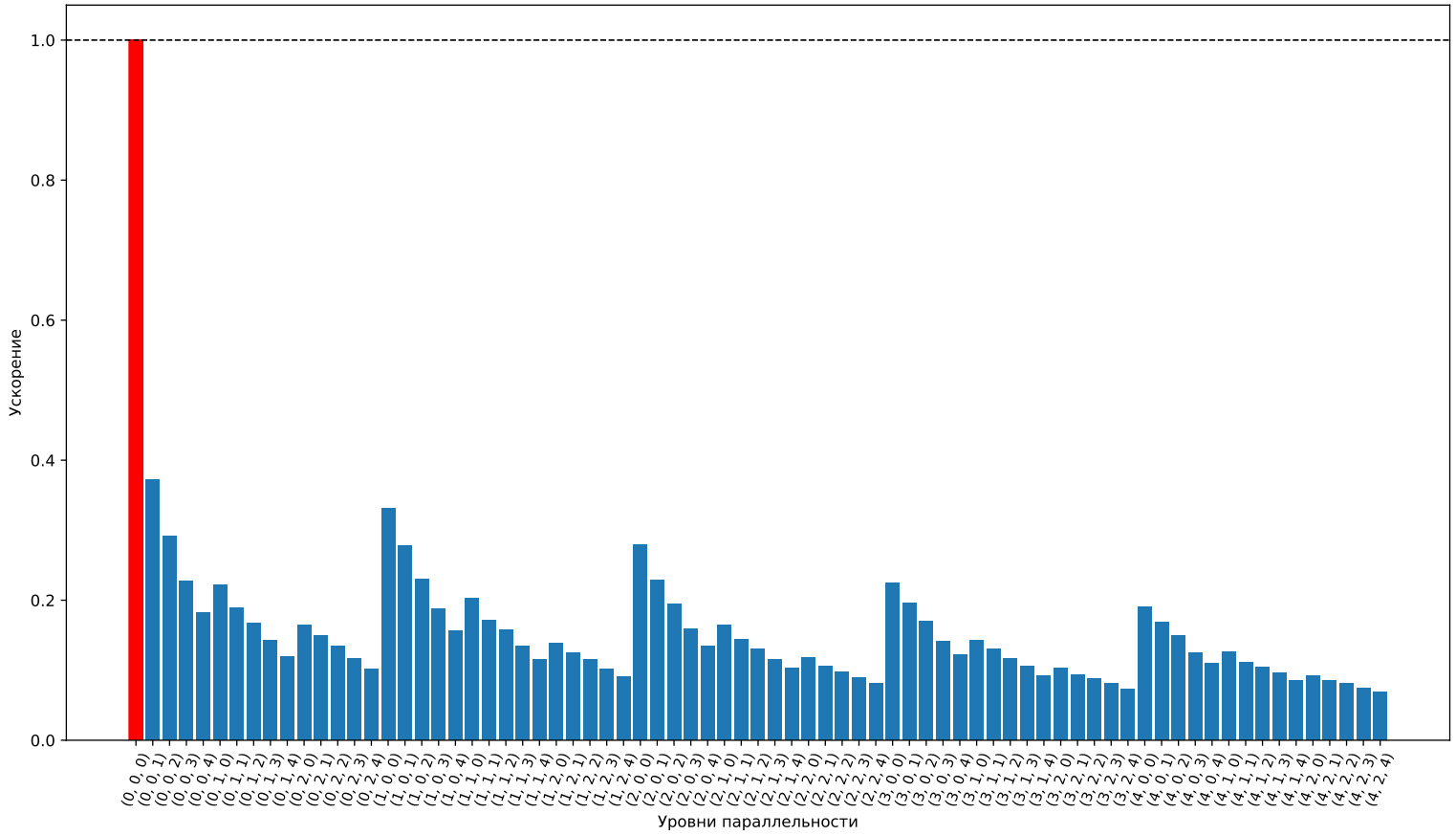

Рис. 5: Результаты измерения времени работы BFS с различными уровнями параллельности на матрице HB/bcsstm03 порядка 112

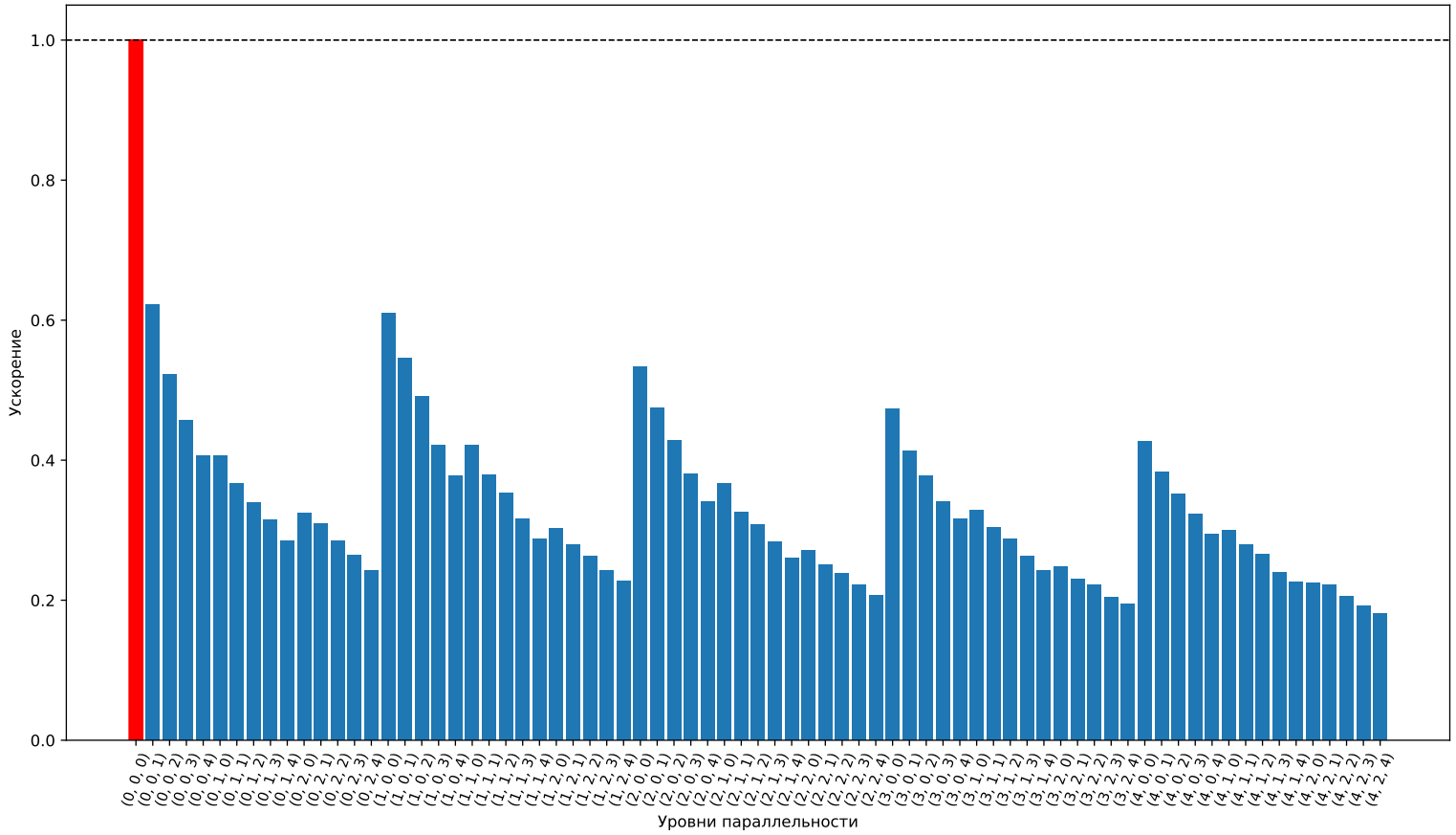

Рис. 6: Результаты измерения времени работы BFS с различными уровнями параллельности на матрице HB/dwt_512 порядка 512

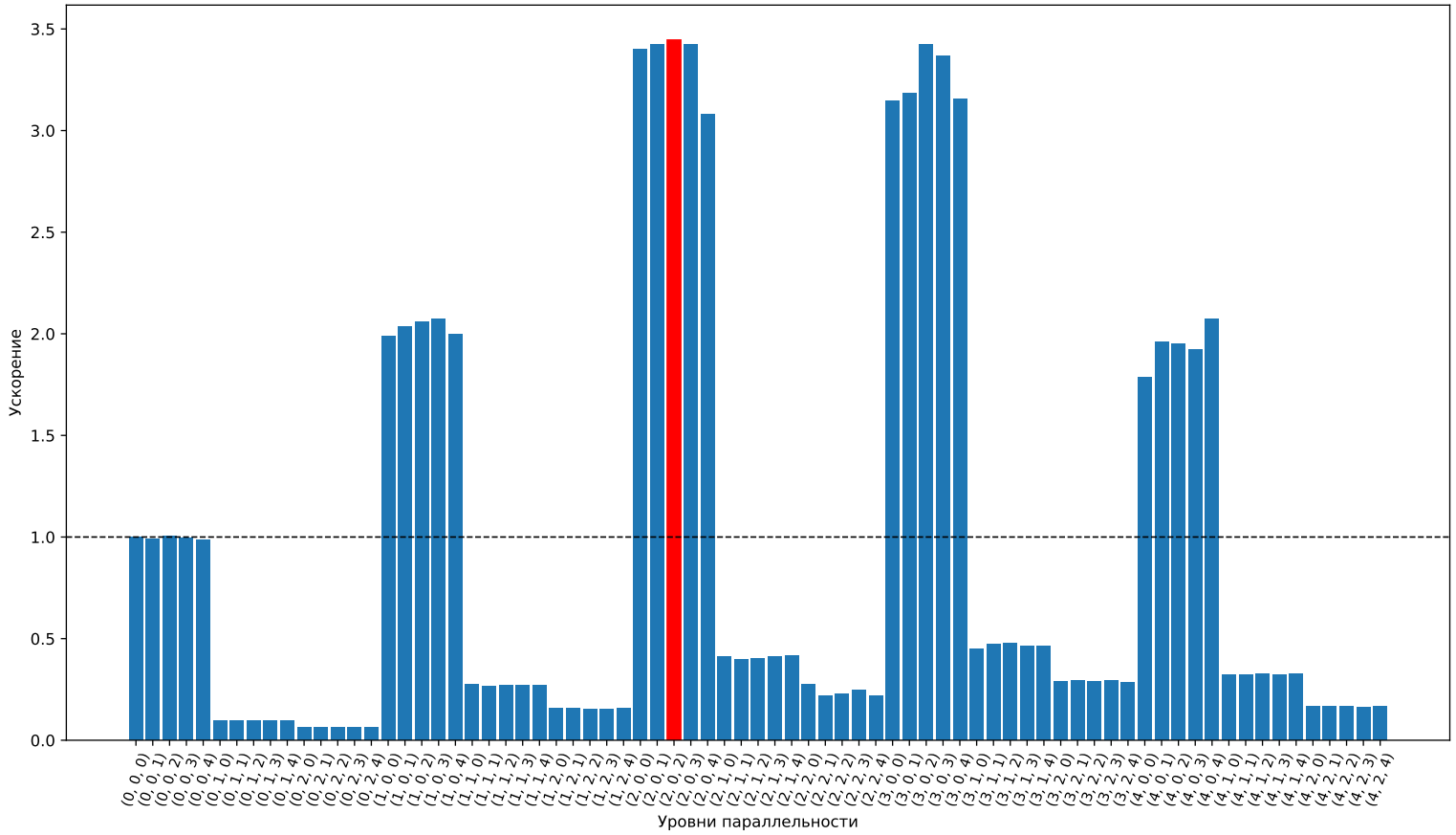

Рис. 7: Результаты измерения времени работы BFS с различными уровнями параллельности на матрице Gset/G52 порядка 1000

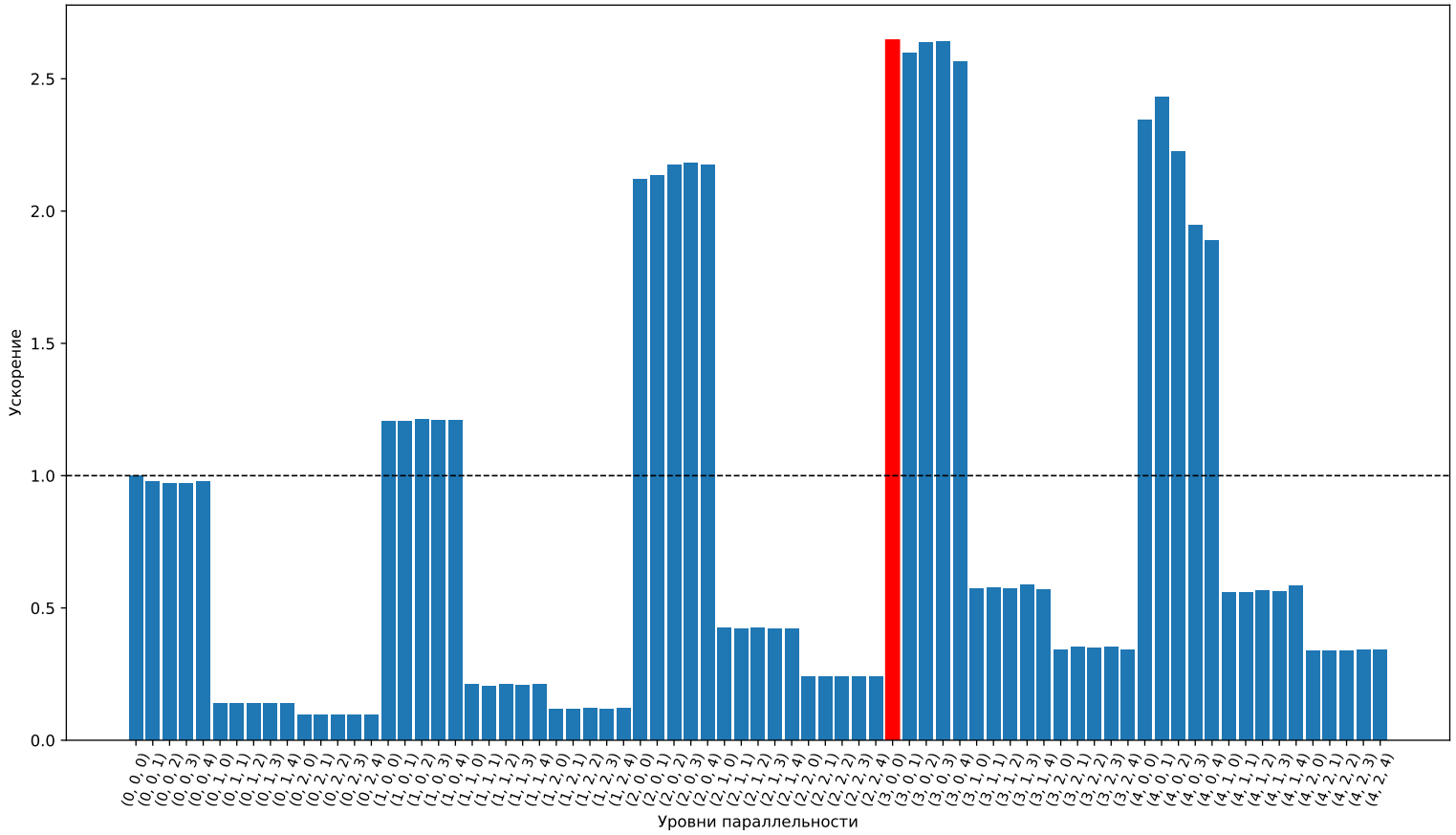

Рис. 8: Результаты измерения времени работы BFS с различными уровнями параллельности на матрице Gset/G59 порядка 5000

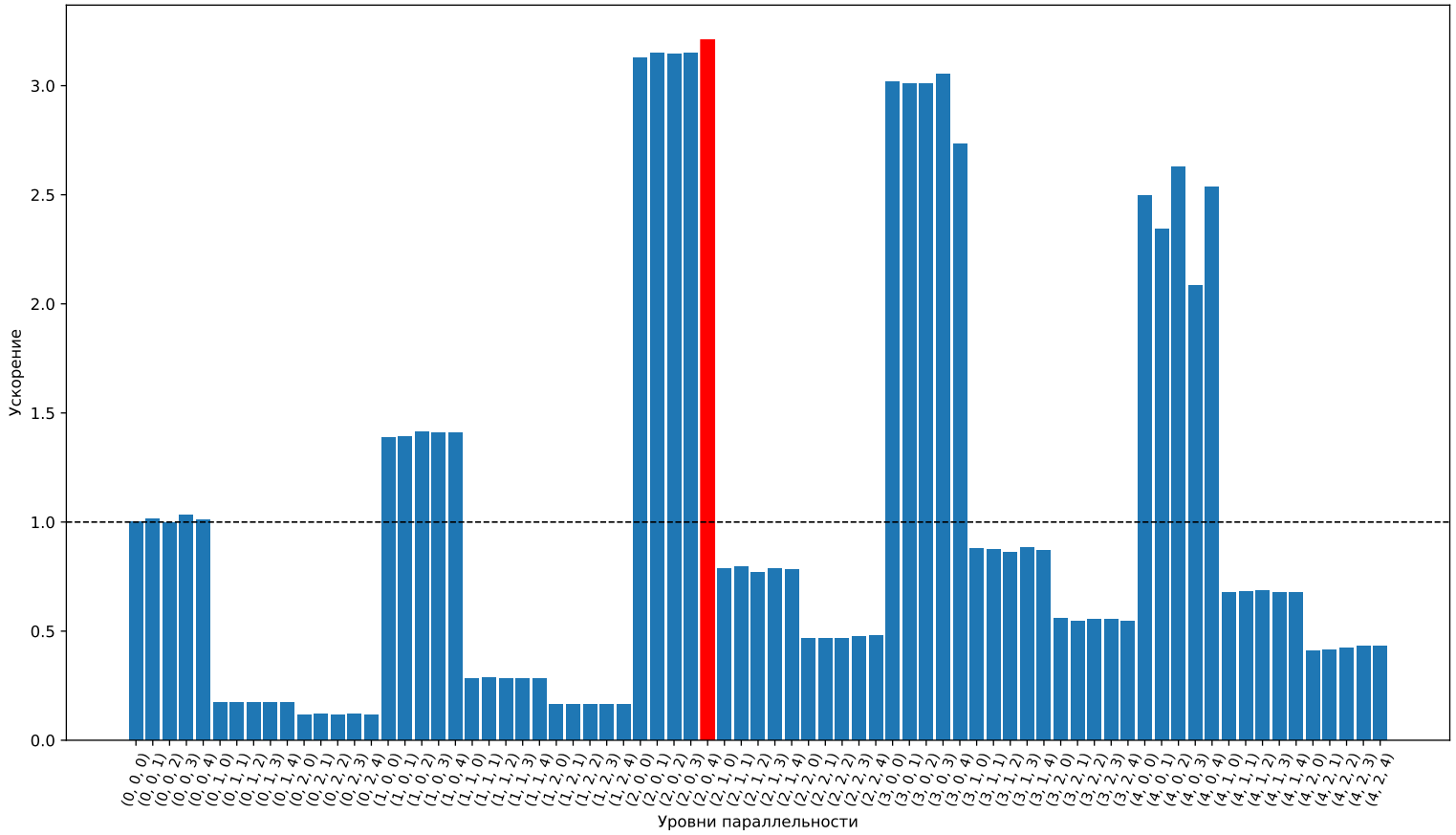

Рис. 9: Результаты измерения времени работы BFS с различными уровнями параллельности на матрице ML_Graph/kmnist_norm_10NN порядка 10000

Заключение

В рамках выполнения данной работы были получены следующие результаты.

- Реализованы тип-матрица и тип-вектор с использованием деревьев квадрантов в качестве метода хранения разреженных структур, а также векторно-матричные операции над ними.
- Реализован алгоритм обхода графа в ширину с использованием линейной алгебры.
- Реализованы параллельные версии векторно-матричных операций с помощью выражений `async` — специальной конструкции языка F#.
- Выполнено экспериментальное исследование реализованного алгоритма. Обход в ширину с использованием параллельной версии матрично-векторных операций показал ускорение до 3.45 раз на матрицах среднего размера в сравнении с однопоточной версией и не показал себя эффективно на матрицах малого и большого размеров.

Результаты исследования могут иметь практическое применение в областях, где требуется анализ разреженных графовых данных.

Список литературы

[1] Davis Timothy A., Hu Yifan. The University of Florida Sparse Matrix Collection // ACM Trans. Math. Softw. — 2011. — dec. — Vol. 38, no. 1. — 25 p. — URL: https://doi.org/10.1145/2049662.2049663.

[2] Doak Daniel F, Gross Kevin, Morris William F. Understanding and predicting the effects of sparse data on demographic analyses // Ecology. — 2005. — Vol. 86, no. 5. — P. 1154–1163.

[3] Hunter J. D. Matplotlib: A 2D graphics environment // Computing in Science & Engineering. — 2007. — Vol. 9, no. 3. — P. 90–95.

[4] John David J, Fetrow Jacquelyn S, Norris James L. Continuous cotemporal probabilistic modeling of systems biology networks from sparse data // IEEE/ACM Transactions on Computational Biology and Bioinformatics. — 2010. — Vol. 8, no. 5. — P. 1208–1222.

[5] Lee Joshua. Multi-threading // Encyclopedia of Big Data / Ed. by Laurie A. Schintler, Connie L. McNeely. — Cham : Springer International Publishing, 2020. — P. 1–4. — ISBN: 978-3-319-32001-4. — URL: https://doi.org/10.1007/978-3-319-32001-4_404-1.

[6] Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks / Aydin Buluç, Jeremy T Fineman, Matteo Frigo et al. // Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. — 2009. — P. 233–244.

[7] Parhami Behrooz. Parallel Processing with Big Data. — 2019.

[8] Rao Chengping, Sun Hao, Liu Yang. Embedding physics to learn spatiotemporal dynamics from sparse data // arXiv preprint arXiv:2106.04781. — 2021.

[9] Sarcar Vaskaran. Threading // Test Your Skills in C# Programming:

Review and Analyze Important Features of C#. — Springer, 2022. — P. 385–429.

[10] Spampinato Daniele G, Sridhar Upasana, Low Tze Meng. Linear algebraic depth-first search // Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. — 2019. — P. 93–104.

[11] Stanimirovic Ivan P, Tasic Milan B. Performance comparison of storage formats for sparse matrices // Ser. Mathematics and Informatics. — 2009. — Vol. 24, no. 1. — P. 39–51.

[12] Yang Carl, Buluç Aydın, Owens John D. GraphBLAST: A High-Performance Linear Algebra-Based Graph Framework on the GPU. — New York, NY, USA : Association for Computing Machinery, 2022. — feb. — Vol. 48. — 51 p. — URL: https://doi.org/10.1145/3466795.