Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б07-мм

Исследование производительности алгоритма обхода графов в ширину

Лодыгин Леонид Александрович

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 **5**

2.1. Терминология 5

2.2. Представление графа в памяти 6

2.3. Breadth-first search 7

2.4. Многопоточность 10

3. Детали реализации **11**

3.1. SparseMatrix и SparseVector 11

3.2. Breadth-first search 13

4. Эксперимент **16**

4.1. Условия эксперимента 16

4.2. Исследовательские вопросы 16

4.3. Метрики 16

4.4. Результаты 17

Заключение **24**

Список литературы **25**

Введение

В настоящее время использование такой структуры данных как *граф* имеет широкое применение для абстракции реальных систем с целью их дальнейшего анализа и обработки. Граф представляет из себя совокупность двух множеств — объектов, называемых вершинами графа, и рёбер, обозначающих попарные связи между объектами. Такое представление используется в социальных науках, физике, химии, но чаще всего в информатике и сетевых технологиях, например, для хранения моделей машинного обучения, для работы с картами или же для хранения и обработки баз данных.

Для хранения графа в памяти используются различные представления, зависящие от соотношения количества вершин и рёбер в графе. К примеру, граф с относительно большим количеством рёбер часто представляется как *матрица смежности*, где каждый столбец и строка обозначают вершину, а в ячейках хранится информация о наличии связи между данными вершинами. Становится очевидно, что в случае малого количества связей между вершинами, то есть когда исследуется *разреженный граф*, такой подход для хранения не является рациональным, поэтому для оптимизации занимаемой памяти граф можно представить в виде *дерева квадрантов*.

Графовая модель позволяет решать внушительный спектр задач, связанный с отношением между вершинами в графе. Для этого разработано большое количество алгоритмов, одним из которых является алгоритм обхода графа в ширину, иначе *Breadth-first search*. При реализации такого алгоритма возможно применение *векторно-матричных* операций, что позволяет естественным образом использовать *многопоточное программирование* для оптимизации скорости работы алгоритма.

Общее количество вершин и *плотность графа* непосредственно влияют на скорость работы алгоритма обхода в ширину, поэтому важной задачей является не только разработка параллельной версии такого алгоритма, но и исследование влияния этих параметров на его работу.

1. Постановка задачи

Целью данной работы является исследование производительности алгоритма обхода графа в ширину в зависимости от входных параметров графа и количества параллельных потоков, используемых для работы алгоритма. Для её выполнения были поставлены следующие задачи:

1. реализовать типы `SparseMatrix` и `SparseVector`;
2. реализовать параллельные версии векторно-матричных операций;
3. реализовать параллельную версию алгоритма обхода графа в ширину;
4. провести эксперименты над графами с различными параметрами и ответить на следующие вопросы:
   - при каких параметрах графа выгоднее использовать параллельную версию алгоритма, а при каких последовательную;
   - использование какого количества потоков даёт наибольший выигрыш в производительности и почему.

2. Обзор

Для реализации алгоритма и необходимых для его работы компонентов не лишним будет для начала рассмотреть базовую теорию графов, ознакомиться с существующими методами хранения больших данных в памяти и иметь представление о многопоточном программировании.

2.1. Терминология

Граф $\mathcal{G} = \langle V, E \rangle$ — упорядоченная пара множеств, где V конечное непустое множество вершин графа, а $E \subseteq V \times V$ множество рёбер. Рёбра графа могут обладать дополнительной информацией, в таком случае граф называется *помеченным* и определятся тройкой $\langle V, E, L \rangle$, где L это конечное множество меток графа. Под меткой стоит понимать данные, которые может содержать ребро в конкретной задаче. Например, если с помощью графа представлена карта, где города это вершины графа, а рёбра — дороги между городами, то в качестве меток могут быть взяты длины этих дорог.

Разреженный граф — граф с малым количеством рёбер по отношению к количеству вершин.

Граф можно представлять в памяти как *матрицу смежности* — квадратная матрица M размеров $n \times n$, где n это количество вершин в графе. Элементы такой матрицы $M[i, j]$ соответствуют единице, если между вершинами i и j существует ребро и нулю, если такое ребро отсутствует. В случае помеченного графа, элементы матрицы принимают вместо единицы значение метки, соответствующее данному ребру, а вместо нуля в матрице для обозначения отсутствия ребра, как правило, вводится специальное обозначение, например `None`.

Список рёбер — другой подход к представлению графа в памяти. В списке рёбер в каждом узле записываются две смежные вершины и вес соединяющего их ребра (для помеченного графа).

Путь — последовательность вершин, в которой каждая вершина соединена со следующей ребром.

Две вершины в графе считаются *связными*, если между ними существует путь.

Дерево — структура данных, состоящая из узлов, связанных между собой в виде иерархической структуры без циклов. Один из узлов в дереве называется корневым. Каждый узел может иметь ноль или более дочерних узлов. Узел, не имеющий дочерних узлов называется *листом*.

Бинарное дерево — это особый тип дерева, где каждый узел может иметь не более двух дочерних узлов.

Дерево квадрантов — это дерево, в котором у каждого внутреннего узла ровно 4 потомка. Деревья квадрантов часто используются для рекурсивного разбиения двухмерного пространства по 4 квадранта (области).

2.2. Представление графа в памяти

Для хранения графа в памяти обычно используют матрицу смежности или список рёбер. При использовании матриц смежности в памяти создается двумерный массив, каждая ячейка которого соответствует элементу матрицы. Однако в случае малого количества рёбер в графе, такой подход нельзя считать рациональным, так как выделенная под несуществующие рёбра память расходуется впустую. При использовании списка рёбер проблема с избыточным использованием памяти решается, но возникают другие трудности, например, с добавлением новых рёбер или же с исследованием вершин на связность. Существует схожий со списком рёбер способ хранения в формате сжатой разреженной строки (`Compressed sparse row`), где матрица представляется тремя одномерными массивами, содержащими ненулевые значения, экстенты строк и индексы столбцов. Такой формат позволяет быстро обращаться к элементу по индексу и производить векторно-матричные операции, однако любое изменение графа в таком формате понесёт за собой большие накладные расходы, о чём говорится в статье Аапо Кироло [4].

Одним из решений данной проблем с избыточным использованием памяти является представление такой матрицы в виде дерева квадран-

тов (рисунок 1), о чём в своей работе упоминает Гилберт Дж. Р. [2]. При таком подходе матрица рекурсивно разбивается на 4 части. В тот момент, когда в одной из частей отсутствуют значащие ячейки, разбиение останавливается и вся часть обозначается как незначащая. Преимуществом такой абстракции является не только сокращение объема памяти, требуемое для хранения графа, но и сохранение возможности применения векторно-матричных операций для реализации необходимых алгоритмов.

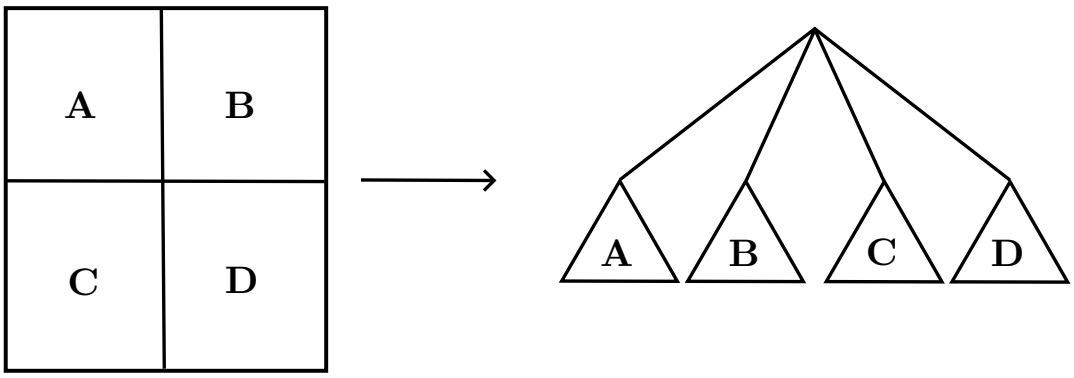

Рис. 1: Представление матрицы в виде дерева квадрантов

2.3. Breadth-first search

Обход в ширину — один из простейших алгоритмов обхода графа, являющийся основой для многих важных алгоритмов для работы с графами. Он подразумевает поуровневое исследование графа:

1. посещение одной или нескольких произвольно выбранных вершин;
2. рекурсивное посещение всех смежных вершин для данной или данных.

Вершины просматриваются в порядке возрастания кратчайшего пути до заданной. Алгоритм прекращает свою работу в случае обхода всех

возможных вершин, либо в случае выполнения требуемого условия. Например, если задачей был поиск кратчайшего пути до определённой вершины.

Данный алгоритм применяется для решения многих задач, одни из которых:

1. поиск кратчайшего пути;
2. поиск компонент связности;
3. поиск всех вершин/рёбер, лежащих на кратчайшем пути.

Простейшей реализацией такого алгоритма является применение *очереди* — абстрактного типа данных с дисциплиной доступа к элементам «первый пришёл — первый вышел». Первая вершина помещается в очередь с меткой 0. Далее, рассматриваются все не посещенные ранее вершины, смежные с ней и добавляются в очередь с меткой данного шага алгоритма. Первая вершина удаляется из очереди и отмечается посещенной. Алгоритм продолжает работу, пока очередь не окажется пустой. Однако, существует другой подход к реализации данного алгоритма. Если граф представлен в виде матрицы смежности, то такой алгоритм возможно реализовать с помощью векторно-матричных операций линейной алгебры [3]. Тогда алгоритм будет выглядеть следующим образом.

1. На вход поступает матрица смежности графа и вектор, в ячейках которого отмечены стартовые вершины.
2. Вектор (фронт) умножается на матрицу и на выходе получается вектор, в ячейках которого отмечены вершины, смежные с изначальными. Посещенные вершины заносятся в результирующий вектор.
3. Из фронта убираются все вершины, посещенные ранее. Предыдущий шаг повторяется.
4. Алгоритм завершает работу, когда вектор становится пустым.

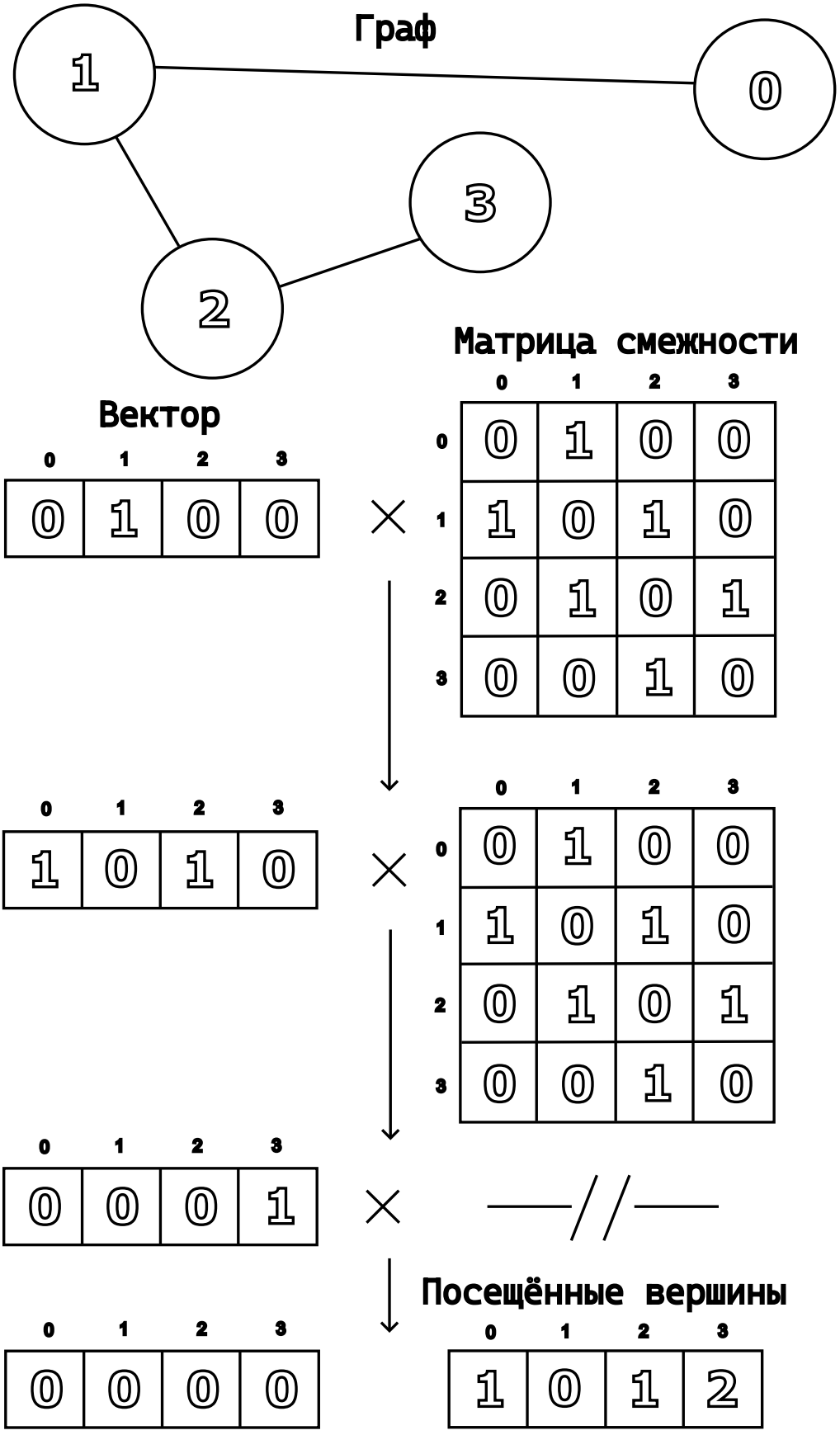

Рис. 2: Алгоритм обхода в ширину с применением линейной алгебры

2.4. Многопоточность

Многопоточность — это концепция, связанная с одновременным выполнением нескольких *потоков* или нитей внутри одной программы. Потоки представляют собой последовательности инструкций, выполняющихся параллельно и независимо друг от друга, имея доступ к общим ресурсам программы. Многопоточные программы могут эффективно использовать многоядерные и многопроцессорные системы, что позволяет выполнять задачи параллельно и ускоряет выполнение программы. Однако такая концепция влечёт за собой ряд проблем. Во-первых, создание отдельного потока требует определённых ресурсов системы. При неумелом выделении количества потоков на решение задачи возможна потеря производительности, если накладные расходы на создание и *синхронизацию* потоков превысят полученную выгоду от многопоточности. Во-вторых, при использовании общих ресурсов возникает риск *состояния гонки* (race condition), когда несколько потоков пытаются одновременно изменять одни и те же данные. Это может привести к непредсказуемым результатам и ошибкам в программе. Наконец, многопоточность напрямую зависит от системы и её аппаратных возможностей. Количество *ядер* и *логических потоков* процессора однозначно определяют максимальное количество нитей, на которые следует разбивать нашу задачу во избежание потери производительности [1].

3. Детали реализации

В данном разделе предлагается рассмотреть основные детали реализации типов `SparseMatrix`, `SparseVector` и `Graph` для представления графов, а также реализацию векторно-матричных операций над этими типами и алгоритма `Breadth-first search` на основе этих операций. Разработка осуществлена на языке F♯. Полный код реализации доступен в публичном репозитории[1].

3.1. SparseMatrix и SparseVector

Тип `SparseVector` (листинг 1) представляет из себя структуру данных, с которой возможны операции как с обычным массивом, например, обращение к элементу, но в то же время хранящуюся в памяти как тип `BinaryTree` (бинарное дерево, листинг 2). Такая реализация обусловлена оптимизацией выделения памяти под хранение экземпляров данного типа. Элементы вектора, обёрнутые в тип **Option** из библиотеки **Microsoft.FSharp.Core**, хранятся в листьях дерева. В том случае, если оба листа одного родителя пусты и обозначены как `None`, то листья «отрезаются» и родитель заменятся на `None`. Таким образом, векторы малой плотности будут занимать меньше памяти для хранения, чем такие же векторы, представленные в памяти в виде обычного массива, но в то же время будут сохранять все свойства векторов.

Листинг 1: Тип SparseVector

```
type SparseVector<'Value when 'Value: equality> =
    val Memory: BinaryTree<'Value>
    val Length: uint
```

Листинг 2: Тип BinaryTree

```
type BinaryTree<'Value> =
    | Node of BinaryTree<'Value> * BinaryTree<'Value>
    | Leaf of 'Value
    | None
```

[1]Репозиторий с реализацией алгоритма: https://github.com/LeonidLodygin/SPBU.2022.FunctionalProgramming.Homeworks (дата доступа: 21 мая 2023 г.).

Для сложения экземпляров типа `SparseVector` реализована функция `ParallelFAddVector`, принимающая два вектора одинакового размера и возвращающая результирующий вектор, в ячейках которого лежат результаты поэлементной операции, переданной в качестве вспомогательной функции. `ParallelFAddVector` с помощью рекурсивной функции `helper` обходит оба дерева и, доходя до листьев, применяет необходимую операцию к значениям в листьях. Пример вспомогательной функции для математического сложения векторов в листинге 3

Листинг 3: Пример вспомогательной поэлементной функции

```
let fAdd a b =
    match a, b with
    | Some x, Some y ->
        if x + y <> 0 then
            Some(x + y)
        else
            Option.None
    | Option.None, Some x -> Some x
    | Some x, Option.None -> Some x
    | Option.None, Option.None -> Option.None
```

Представление в виде двоичных деревьев сохраняет возможность использования многопоточности при реализации такой функции. Для сложения векторов необходимо рекурсивно обходить оба дерева, представляющие данные векторы, и, доходя до листьев, соответствующих ячейкам в векторах с одинаковым индексом, применять к ним поэлементную операцию. Функция для обхода деревьев на каждом шаге сопоставляет узлы деревьев. В том случае, если оба узла имеют по два дочерних, не отмеченных как `None` узла, происходит два рекурсивных вызова функции от дочерних узлов, следовательно, в данном месте имеется возможность для использования параллельных вычислений (листинг 4). Для контроля количества потоков в функцию `ParallelFAddVector` передается параметр типа `int`, определяющий количество потоков в степени двойки, которое мы хотим выделить для работы функции.

Листинг 4: Разделение задачи сложения двух деревьев на две подзадачи и использование потоков для получения результата

```
| Node (left, right), Node (left2, right2) ->
    if level = 0 then
        Node(helper left left2 level, helper right right2 level)
        |> NoneDestroyer
    else
        let tasks =
            [| async { return helper left left2 (level - 1) }
               async { return helper right right2 (level - 1) } |]

        let results = tasks |> Async.Parallel |> Async.RunSynchronously
        Node(results[0], results[1]) |> NoneDestroyer
```

Объекты типа `SparseMatrix` аналагично объектам типа `SparseVector` хранятся в памяти в виде дерева квадрантов. Абстракция позволяет взаимодействовать с такими объектами как с двумерными массивами и, соответственно, применять к ним векторно-матричные операции. Для умножения вектора типа `SparseVector` на матрицу типа `SparseMatrix` была реализована функция `ParallelMultiplyVecMat`, принимающая вектор, матрицу и возращающая результирующий вектор. Аналогично функции сложения, данная функция легко поддается разделению на подзадачи для параллельных вычислений. Входной параметр `parallelLevel` определяет количество подзадач в четвёртой степени.

3.2. Breadth-first search

Функция `Bfs` принимает на вход объект типа `Graph` (листинг 5), хранящийся в памяти как `SparseMatrix`, список стартовых вершин и два числовых параметра, отвечающих за многопоточность — `fAddLevel`, который затем передаётся в функцию сложения векторов и `multLevel`, передающийся функции умножения (листинг 6).

Листинг 5: Тип Graph

```
type Graph<'Value when 'Value: equality> =
    val Memory: SparseMatrix<'Value>
    val Vertices: uint
    val Edges: uint
```

Листинг 6: Алгоритм Bfs с использованием векторно-матричных операций

```
let Bfs (graph: Graph<'Value>) (apexes: List<uint>) fAddLevel multLevel =
    let apexes = List.map (fun x -> (x, ())) apexes
    let front = SparseVector(apexes, graph.Vertices)

    let visited =
        ParallelFAddVector(SuperSum 0u)
            front
            (SparseVector(BinaryTree.None, graph.Vertices)) fAddLevel

    let rec helper (front: SparseVector<'A>) visited iter =
        if front.IsEmpty then
            visited
        else
            let newFront =
                ParallelFAddVector
                    Mask
                    (ParallelMultiplyVecMat front graph.Memory FrontAdd FrontMult multLevel)
                    visited
                    fAddLevel

            let visited = ParallelFAddVector(SuperSum iter) newFront visited fAddLevel
            helper newFront visited (iter + 1u)

    helper front visited 1u
```

По данном списку создается вектор `front`, который на каждом шаге алгоритма будет содержать актуальные вершины, в которых мы находимся. Далее, создается вектор посещенных уже ранее вершин с помощью функции сложения векторов и вспомогательной поэлементной функции `SuperSum` (листинг 7).

Далее, запускается вспомогательная рекурсивная функция, которая будет умножать `front` на матрицу, убирать из него уже посещенные ранее вершины с помощью поэлементной функции `Mask` (листинг 8) и складывать новые посещенные вершины в вектор посещенных вершин.

Алгоритм завершает свою работу, когда `front` становится пустым.

Листинг 7: Вспомогательная поэлементная функция для определения посещенных вершин

```
let SuperSum iter value1 value2 =
    match value1, value2 with
    | Option.None, Option.None -> Option.None
    | Option.None, value2 -> value2
    | Some _, Option.None -> Some iter
```

Листинг 8: Вспомогательная поэлементная функция для удаления из ветора front посещенных ранее вершин

```
let Mask value1 value2 =
    match value1, value2 with
    | Option.None, _ -> Option.None
    | Some value1, Option.None -> Some value1
    | Some _, _ -> Option.None
```

4. Эксперимент

В данном разделе предлагается рассмотреть результаты экспериментального исследования реализованного алгоритма обхода графа в ширину. Основная задача: выявить зависимость производительности алгоритма от входных параметров графа и количества асинхронных потоков, обрабатывающих данный алгоритм.

4.1. Условия эксперимента

Для экспериментов использовалась рабочая станция с процессором Intel Core i5-10300H с тактовой частотой 2.50GHz, RAM DDR4 объемом 8гб под управлением ОС Windows 10.

4.2. Исследовательские вопросы

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?

4.3. Метрики

В качестве метрик производительности используется время, требуемое на завершение алгоритма. Показатели времени получены с помощью библиотеки `BenchmarkDotNet v0.13.5`[2]. Для замеров были выбраны стандартные настройки по «прогреву» и количеству итераций для измерений.

Для анализа алгоритма было решено воспользоваться собственным генератором графов нужного размера с нужной плотностью. Данный подход позволяет минимизировать влияние разницы прочих параметров на исследуемый, следовательно результаты, полученные в таком ис-

[2]Репозиторий библиотеки `BenchmarkDotNet`: https://github.com/dotnet/BenchmarkDotNet (дата доступа: 21 мая 2023 г.).

следовании, позволят точнее выявить необходимые зависимости, нежели при использовании уже существующих графов. Генератор принимает на вход количество вершин в графе и параметр `density` типа `float`, отвечающий за плотность графа. По количеству заданных вершин создаётся двумерный массив, ячейки которого заполняются следующим образом. Если параметр плотности графа меньше 0.5, все ячейки массива заполняются `Option.None`, затем с помощью цикла в случайных ячейки массива заносятся сгенерированные случайные числа `int`, обёрнутые в тип `Option`, пока не будет достигнута необходимая плотность. В противном случае, массив полностью заполняется случайными числами типа `int`, обёрнутыми в тип `Option`, а в случайные ячейки заносятся `Option.None`, пока опять же таки не будет достигнута необходимая плотность.

4.4. Результаты

Входные параметры:

1. `Vertices` — количество вершин в графе;
2. `Density` — плотность графа;
3. `parallelMult` — уровень распараллеливания функции умножения, каждый уровень увеличивает число подзадач в 4 раза;
4. `parallelAdd` — уровень распараллеливания функции сложения, каждый уровень увеличивает число подзадач в 2 раза.

Результаты замеров:

1. `Time` — результат измерений в мс (за исключением таблицы 5, где время измерений указано в микросекундах и таблицы 7, где время указано в секундах);

Замеры проводились начиная от очень малых графов (10 вершин), заканчивая графами с 5000 вершинами. Дополнительно были исследованы графы с 10000 вершинами, но очень малой плотности. Выбор мак-

симального размера графа связан с ограничением оперативной памяти системы, на которой производились вычисления.

В таблице 1 представлены результаты замеров времени работы последовательного алгоритма при различных входных параметрах графа. Из данной таблицы видно, что время работы алгоритма возрастает с увеличением плотности графа, а затем, когда граф достигает плотности 0.5 начинает снижаться. Не трудно догадаться, что при высокой плотности графа уменьшается количество шагов алгоритма, необходимых для его завершения, так как за каждый шаг будет осуществляться переход в большее количество смежных вершин, следовательно общее время работы уменьшается.

Таблица 1: Производительность последовательного алгоритма обхода графов в ширину.

Vertices	Density	Time	Vertices	Density	Time
500	0.1	17 ± 0.1	2500	0.1	429 ± 2
500	0.3	33 ± 0.3	2500	0.3	860 ± 3
500	0.5	44 ± 0.2	2500	0.5	1091 ± 8
500	0.7	42 ± 0.2	2500	0.7	1095 ± 3
500	0.9	36 ± 0.2	2500	0.9	980 ± 4
1000	0.1	68 ± 0.4	5000	0.1	1717 ± 3
1000	0.3	137 ± 0.6	5000	0.3	3531 ± 20
1000	0.5	182 ± 1.1	5000	0.5	4426 ± 23
1000	0.7	179 ± 1.2	5000	0.7	4421 ± 9
1000	0.9	158 ± 0.3	5000	0.9	3984 ± 14

В таблице 2 приведены результаты замеров времени работы параллельного алгоритма обхода графа при различных параметрах. Для удобства, при каждом параметре плотности графа взята наилучшая скорость среди всех параллельных версий алгоритма. Из полученных результатов можно заметить, что при увеличении размеров графа, наиболее оптимальное количество подзадач для параллельных вычислений выравнивается к 16 для функции умножения. Стоить отметить, что на достаточно больших графах, при одном и том же уровне распараллеливания функции умножения, влияние распараллеливания функции сложения становится не столь существенным, любая из комбинаций даёт один и тот же результат по времени в пределах погрешности (таблица 3). Связано это с тем, что наибольший процент времени выделяется под функцию умножения, как наиболее затратную по ресурсам.

Таблица 2: Производительность параллельного алгоритма обхода графов в ширину (лучшие данные по метрике). Графа `SpeedUp` показывает отношение скорости работы параллельной версии алгоритма к скорости работы последовательной версии.

Vertices	Density	parallelMult	parallelAdd	SpeedUp
500	0.1	2	1	0.53
500	0.3	1	1	0.57
500	0.5	1	1	0.56
500	0.7	1	2	0.62
500	0.9	2	2	0.64
1000	0.1	1	1	0.54
1000	0.3	1	1	0.53
1000	0.5	1	1	0.5
1000	0.7	1	2	0.55
1000	0.9	1	3	0.57
2500	0.1	2	3	0.56
2500	0.3	2	3	0.54
2500	0.5	2	1	0.53
2500	0.7	2	3	0.59
2500	0.9	2	1	0.6
5000	0.1	2	1	0.56
5000	0.3	2	3	0.51
5000	0.5	2	1	0.53
5000	0.7	2	3	0.59
5000	0.9	2	1	0.58

Таблица 3: Сравнение влияния уровня распараллеливания функции умножения и уровня распараллеливания функции сложения на графе с 5000 вершинами.

Vertices	Density	parallelMult	parallelAdd	Time
5000	0.1	1	1	1411 ± 10
5000	0.1	1	2	1401 ± 12
5000	0.1	1	3	1409 ± 9
5000	0.1	2	1	970 ± 19
5000	0.1	2	2	998 ± 19
5000	0.1	2	3	980 ± 19
5000	0.1	3	1	1126 ± 21
5000	0.1	3	2	1138 ± 23
5000	0.1	3	3	1151 ± 18

Так как на всех параметрах графа параллельная версия алгоритма обходила по производительности последовательную, было решено произвести дополнительные замеры при графах с меньшим количеством вершин (таблица 5). Выяснилось, что при таких размерах графа последовательная версия работает значительно быстрее, чем параллельная.

Однако, уже на графах с 50 вершинами параллельная версия догоняет и начинает опережать последовательную версию по производительности.

Также, чтобы исследовать оптимальное разделение на подзадачи для функции умножения, как наиболее значимой для распараллели-

Таблица 4: Сравнение производительности параллельного и обычного Bfs на графах с 10 вершинами. Графа `SpeedUp` показывает отношение скорости работы параллельной версии алгоритма к скорости работы последовательной версии.

Vertices	Density	TimeBfs	TimeParallelBfs	SpeedUp
10	0.1	15.00 ± 0.03	53 ± 0.2	3.5
10	0.3	20 ± 0.4	78 ± 0.3	3.9
10	0.5	32 ± 0.3	124 ± 1.3	3.8
10	0.7	27 ± 0.5	60 ± 0.4	2.2
10	0.9	25 ± 0.6	44 ± 0.4	1.76

Таблица 5: Сравнение производительности параллельного и обычного Bfs на графах с 50 вершинами. Графа `SpeedUp` показывает отношение скорости работы параллельной версии алгоритма к скорости работы последовательной версии.

Vertices	Density	TimeBfs	TimeParallelBfs	SpeedUp
50	0.1	293 ± 6	284 ± 3	0.97
50	0.3	372 ± 7	278 ± 5	0.74
50	0.5	572 ± 11	614 ± 9	1.07
50	0.7	438 ± 7	325 ± 2	0.74
50	0.9	416 ± 8	312 ± 2	0.75

вания, были проведены замеры с теми же размерами и плотностями графа, но с использованием последовательной функции сложения. Результаты приведены в таблице 6.

Таблица 6: Производительность параллельного алгоритма обхода графов в ширину с использованием последовательной функции сложения веторов.

Vertices	Density	parallelMult	Time	Vertices	Density	parallelMult	Time
500	0.1	1	$9,8 \pm 0.1$	2500	0.1	2	240 ± 3
500	0.3	1	19 ± 0.4	2500	0.3	2	470 ± 9
500	0.5	2	44 ± 0.9	2500	0.5	2	1004 ± 19
500	0.7	1	26 ± 0.5	2500	0.7	2	661 ± 13
500	0.9	2	24 ± 0.4	2500	0.9	2	620 ± 12
1000	0.1	1	36 ± 0.5	5000	0.1	2	923 ± 18
1000	0.3	1	72 ± 1.4	5000	0.3	2	1843 ± 36
1000	0.5	2	153 ± 1.4	5000	0.5	2	4143 ± 78
1000	0.7	1	98 ± 1.6	5000	0.7	2	2571 ± 39
1000	0.9	1	90 ± 1.1	5000	0.9	2	2372 ± 41

В ходе замеров было установлено, что при увеличении размеров графа, вне зависимости от плотности, время работы алгоритма при разделении на 16 подзадач для функции умножения хоть и больше, чем при работе с разделением на 64 подзадачи, но разрыв становится все меньше и меньше. Вследствие этого наблюдения, было решено произ-

вести замеры для параллельных версий алгоритма для графов с 10000 вершинами, что являлось пределом допустимого значения количества вершин для обработки графа системой. Исследовались графы маленькой плотности с целью выяснить, сможет ли версия с 64 подзадачами обогнать по производительности версию с 16 подзадачами. Результаты приведены в таблице 7.

Таблица 7: Сравнение производительности алгоритма обхода в ширину при распараллеливании на 16 и 64 подзадачи для функции умножения. Графа `SpeedUp` показывает отношение скорости работы алгоритма при 64 подзадачах к скорости работы при 16 подзадачах.

Vertices	Density	parallelMult	Time	SpeedUp
1000	0.1	2	0.0400 ± 0.0003	1.25
2500	0.1	2	0.240 ± 0.002	1.22
5000	0.1	2	0.9 ± 0.03	1.2
10000	0.1	2	4 ± 0.07	1.25

Однако, как показало дополнительное измерение при графе с 10000 вершинами, по производительности всё так же лидирует распараллеливание на 16 подзадач, причём разрыв между скоростями вырос.

Результаты, полученные с помощью `BenchmarkDotNet` были проверены на адекватность с помощью библиотек `SciPy v1.10.1`[3], `NumPy v1.24.3`[4] и `Matplotlib v3.7.1`[5] на языке `Python`. Например, рассмотрим данные, полученные при измерении производительности параллельной версии алгоритма на графе с 5000 вершинами плотности 0.9 и уровнями распараллеливания parallelMult, равным двум, и parallelAdd, равным 1. Результаты замеров приведены в таблице 8.

Таблица 8: Замеры времени работы параллельной версии алгоритма на графе с 5000 вершинами плотности 0.9. Функция умножения разделена на 16 подзадач, функция сложения на две. Время измерено в секундах.

N	1	2	3	4	5	6	7	8	9	10
Time	2.4081	2.3506	2.3857	2.3383	2.2634	2.2931	2.2984	2.3895	2.2800	2.2899
N	11	12	13	14	15	16	17	18	19	20
Time	2.3293	2.2977	2.2358	2.3113	2.4044	2.3317	2.3054	2.2614	2.4011	2.3421

С помощью функций `scipy.stats.shapiro()` (тест Шапиро-Уилка)

[3]Сайт библиотеки `SciPy`: https://scipy.org/ (дата доступа: 21 мая 2023 г.).

[4]Сайт библиотеки `NumPy`: https://numpy.org/ (дата доступа:21 мая 2023 г.).

[5]Сайт библиотеки `MatPlotLib`: https://matplotlib.org/ (дата доступа: 21 мая 2023 г.).

и `scipy.stats.normaltest()` (критерий согласия Пирсона) было проверено соответствие выборки нормальному распределению. Значение `pvalue` в результате проверок равнялось 0.36 (Шапиро) и 0.5 (Пирсон), что позволяет нам не отклонять «нулевую гипотезу» и считать полученные данные как нормальное распределение. Далее, с помощью функций `np.mean()` и `np.std()` были посчитаны среднее значение $Mean = 2.32586$ и стандартное отклонение $StdDev = 0.051$. Аналогичные результаты были получены и библиотекой `BenchmarkDotNet`: $Mean = 2.326$, $StdDev = 0.051$. При помощи `matplotlib.pyplot.hist()` была построена гистограмма (рисунок 3) из полученной выборки.

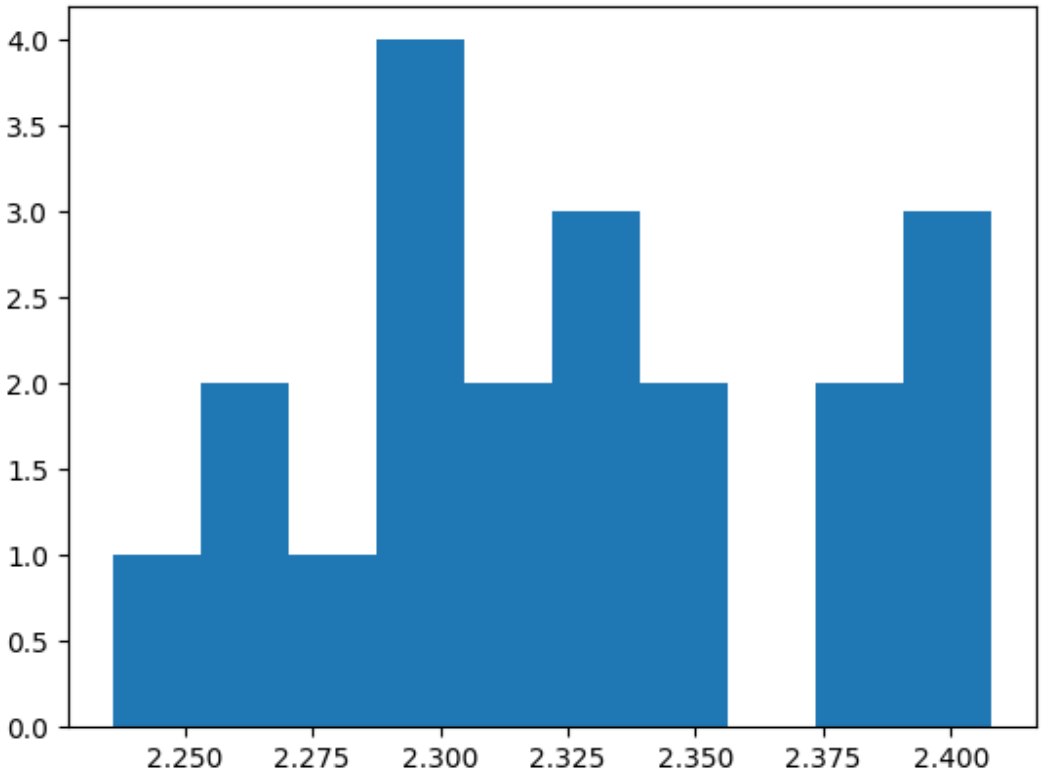

Рис. 3: Гистограмма распределения данных

4.4.1. RQ1

Исходя из полученных данных, последовательная версия алгоритма работает быстрее параллельной только на очень малых графах (10

вершин), когда накладные расходы на выделение дополнительных потоков и их синхронизацию влияют сильнее, чем ускорение относительно последовательной версии. При входных графах от 50 вершин и более, параллельная версия даёт прирост к производительности вплоть до увеличения скорости работы алгоритма в 2 раза при любых параметрах плотности.

4.4.2. RQ2

В ходе экспериментов было установлено, что использование разделения на 16 подзадач, начиная с определённых размеров графа (2500 вершин для системы, используемой при замерах), даёт максимальный прирост по производительности алгоритма обхода в ширину. Такой результат можно объяснить количеством ядер процессора. Система, на которой производились замеры имеет 4-х ядерный процессор `Intel Core i5-10300H`, вследствие чего количество независимых вычислителей равняется четырём. С другой стороны, данный процессор поддерживает технологию `Hyper-Threading`, позволяющую разделить одно ядро процессора на 2 отдельных логических процессора, работающих независимо. Таким образом наилучшая производительность наблюдается на 8 потоках, каждый из которых обрабатывает по 2 подзадачи. Польза от такого разделения влияет на конечное время работы алгоритма сильнее, чем затраты времени на выделение необходимых потоков и синхронизацию их работы.

Заключение

В рамках проведения работы были получены следующие результаты.

- Реазлизованы типы `SparseMatrix` и `SparseVector`.
- Реализованы параллельные версии векторно-матричных операций — `ParallelMultiplyVecMat` и `ParallelFAddVector`.
- Реализована параллельная версия обхода графа в ширину.
- Проведено экспериментальное исследование реализованного алгоритма. Параллельная версия работает быстрее последовательной на любых графах с количеством вершин больше 10. Наибольшую выгоду для производительности при обработке графов относительно крупных размеров даёт использование количества подзадач, равное удвоенному числу логических процессоров в системе.

В качестве будущих задач для исследований можно выделить следующие пункты.

- Исследование производительности алгоритма обхода графа в ширину в зависимости от типа данных, содержащихся на рёбрах.
- Выявление оптимального количества подзадач при многопоточном программировании в зависимости от количества оперативной памяти системы.

Список литературы

[1] Akhter Shameem, Jason Roberts. Multi-Core Programming Increasing Performance through Software Multi-threading. — 2006.

[2] Buluç Aydin, Gilbert John R. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments // SIAM Journal on Scientific Computing. — 2012. — Vol. 34, no. 4. — P. C170–C191.

[3] Davis Timothy. Algorithm 9xx: SuiteSparse: GraphBLAS: graph algorithms in the language of sparse linear algebra // Submitted to ACM TOMS. — 2018.

[4] Kyrola Aapo, Blelloch Guy, Guestrin Carlos. GraphChi: Large-Scale Graph Computation on Just a PC // Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. — OSDI'12. — USA : USENIX Association, 2012. — P. 31–46.