

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	23
Список литературы	24

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 *взвешенным* и определяется 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 взвешенного графа, элементы матрицы принимают вместо единицы значение метки, соответствующее данному ребру.

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

Путь — это последовательность вершин и рёбер, в которой каждый элемент инцидентен предыдущему и последующему.

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

Дерево — ациклический связный граф. Множество вершин в данном

случае принято называть узлами, а множество рёбер — ветвями. Один узел дерева является *корнем*, и он не имеет родителей. Все остальные узлы являются потомками корня. *Листья* — это узлы дерева, которые не имеют потомков. Они находятся на самом нижнем уровне дерева.

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 двумерного 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, экстенды 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 квадрантов (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.

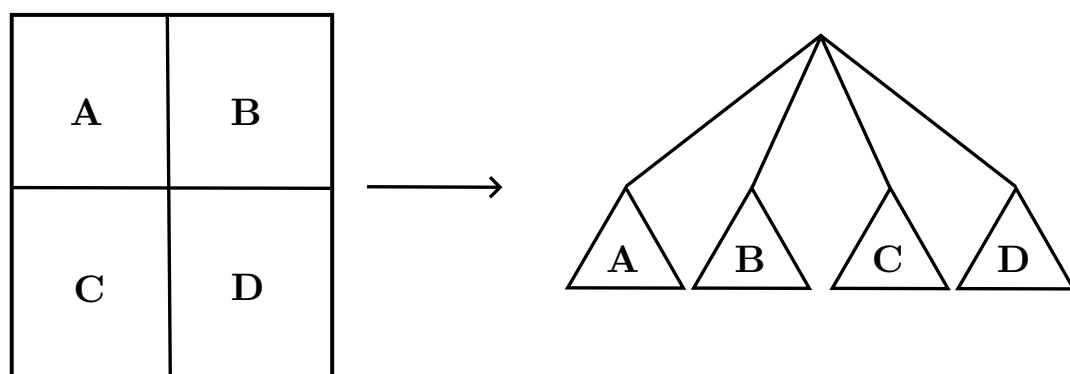


Рис. 1: Представление матрицы в виде дерева квадрантов

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 или же, в данном случае, в виде дерева квадрантов, 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 с помощью поэлементных операций 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.

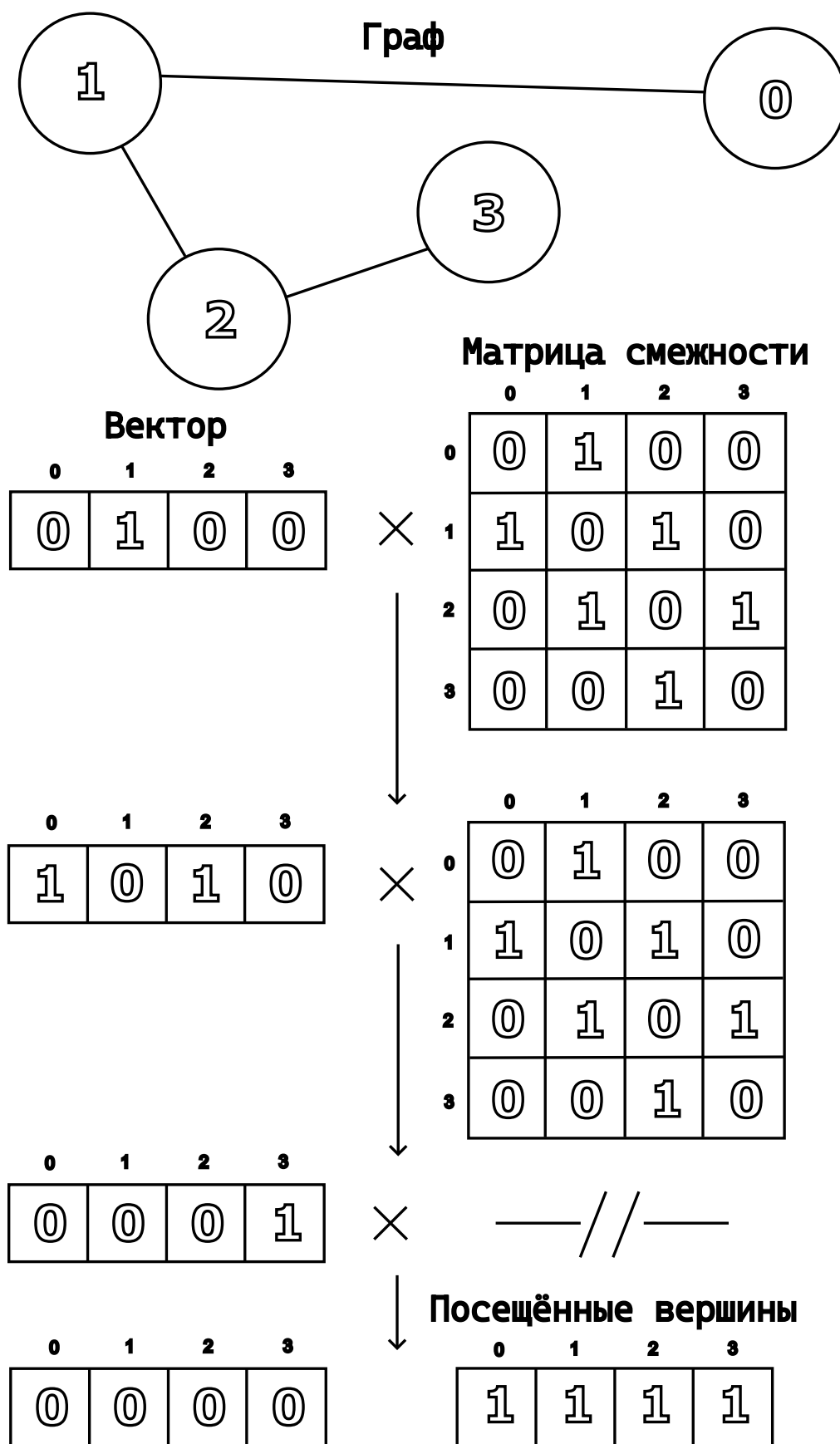


Рис. 2: Алгоритм обхода в ширину с применением линейно алгебры

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 репозитории¹.

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 заменяется 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`

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

Листинг 2: Тип `BinaryTree`

```
1 type BinaryTree<'Value> =  
2     | Node of BinaryTree<'Value> * BinaryTree<'Value>  
3     | Leaf of 'Value  
4     | None
```

¹<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

```
1 let fAdd a b =  
2   match a, b with  
3   | Some x, Some y ->  
4     if x + y <> 0 then  
5       Some(x + y)  
6     else  
7       Option.None  
8   | Option.None, Some x -> Some x  
9   | Some x, Option.None -> Some x  
10  | Option.None, Option.None -> Option.None
```

Представление в виде двоичных деревьев сохраняет возможность использования многопоточности при реализации такой функции. При обходе деревьев, если оба дерева имеют не `None` потомков, происходят два рекурсивных вызова функции `helper` от потомков, 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

```
1 | Node (left, right), Node (left2, right2) ->  
2   if level = 0 then  
3     Node(helper left left2 level, helper right right2 level)  
4     |> NoneDestroyer
```

```

5   else
6       let tasks =
7           [| async { return helper left left2 (level - 1) }
8             async { return helper right right2 (level - 1) } |]
9
10      let results = tasks |> Async.Parallel |> Async.RunSynchronously
11      Node(results[0], results[1]) |> NoneDestroyer

```

Объекты типа **SparseMatrix** аналогично 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 возвращающая результирующий вектор. Аналогично функции сложения, данная функция легко поддается разделению на подзадачи, каждая из которых выполняется на своем потоке. Количество потоков определяется входным параметром **parallelLevel**.

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 (листинг 6).

Листинг 5: Тип Graph

```

1 type Graph<'Value when 'Value: equality> =
2     val Memory: SparseMatrix<'Value>
3     val Vertices: uint
4     val Edges: uint

```

Листинг 6: Алгоритм Bfs с использованием векторно-матричных операций

```

1 let Bfs (graph: Graph<'Value>) (apexes: List<uint>) fAddLevel multLevel =
2     let apexes = List.map (fun x -> (x, ())) apexes
3     let front = SparseVector(apexes, graph.Vertices)
4
5     let visited =
6         ParallelFAddVector(SuperSum 0u) front (SparseVector(BinaryTree.None, graph.Vertices))
7         fAddLevel

```

```

7
8 let rec helper (front: SparseVector<'A>) visited iter =
9     if front.IsEmpty then
10         visited
11     else
12         let newFront =
13             ParallelFAddVector
14                 Mask
15                 (ParallelMultiplyVecMat front graph.Memory FrontAdd FrontMult multLevel)
16                 visited
17                 fAddLevel
18
19         let visited = ParallelFAddVector(SuperSum iter) newFront visited fAddLevel
20         helper newFront visited (iter + 1u)
21
22 helper front visited 1u

```

По данному 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).

Листинг 7: Вспомогательная поэлементная функция для определения посещенных вершин

```

1 let SuperSum iter value1 value2 =
2     match value1, value2 with
3     | Option.None, Option.None -> Option.None
4     | Option.None, value2 -> value2
5     | Some _, Option.None -> Some iter

```

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 8: Вспомогательная поэлементная функция для удаления из вектора **front** посещенных ранее вершин

```

1 let Mask value1 value2 =
2     match value1, value2 with
3     | Option.None, _ -> Option.None

```

```
4 | Some value1, Option.None -> Some value1
5 | 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 Pro версии 21H2.

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 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 исследовании, позволят точнее выявить необходимые зависимости, нежели при подборе базы графов из реальных существующих. Генератор

²<https://github.com/dotnet/BenchmarkDotNet> (дата доступа: 21 мая 2023 г.).

принимает на вход количество вершин в графе и параметр `density` типа `float`, отвечающий за плотность графа. По заданным вершинам создается двумерный массив, заполненный либо `Option.None`, если плотность графа менее 50%, либо случайным числом типа `int`, обёрнутым в тип `Option`. По параметру плотности графа вычисляется количество рёбер, необходимое для достижения такой плотности. Затем, запускается цикл, в котором генерируются два числа от нуля до количества вершин и, в зависимости от того, является граф плотным или нет, ячейки массива, соответствующие сгенерированным числам, заполняются либо случайными числами типа `int`, обёрнутым в тип `Option`, либо `Option.None`.

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 3, где время измерений указано в микросекундах и таблицы 5, где время указано в секундах);

Замеры проводились начиная от сравнительно малых графов (500 вершин), заканчивая графами с 5000 вершинами. Выбор максимального 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 0.7 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 ± 0.06	2500	0.1	429 ± 2
500	0.3	33 ± 0.3	2500	0.3	860 ± 3
500	0.7	42 ± 0.2	2500	0.7	1095 ± 3
500	0.9	36 ± 0.17	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	0.7	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. Стоит 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. 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	TIME	SPEEDUP
500	0.1	2	1	9 ± 0.2	0.53
500	0.3	1	1	19 ± 0.2	0.57
500	0.7	1	2	26 ± 0.2	0.62
500	0.9	2	2	23 ± 0.3	0.64
1000	0.1	1	1	37 ± 0.7	0.54
1000	0.3	1	1	73 ± 0.9	0.53
1000	0.7	1	2	99 ± 1.4	0.55
1000	0.9	1	3	91 ± 1.5	0.57
2500	0.1	2	3	241 ± 5	0.56
2500	0.3	2	3	464 ± 9	0.54
2500	0.7	2	3	650 ± 11	0.59
2500	0.9	2	1	585 ± 12	0.6
5000	0.1	2	1	970 ± 19	0.56
5000	0.3	2	3	1814 ± 36	0.51
5000	0.7	2	3	2606 ± 51	0.59
5000	0.9	2	1	2326 ± 45	0.58

Для более точного анализа были произведены дополнительные замеры при очень малых размерах входного графа (таблица 3). Выяснилось, что при малых размерах графа последовательная версия работает значительно быстрее, чем параллельная.

Таблица 3: Сравнение производительности параллельного и обычного Vfs на графах с малым количеством вершин. Графа **SpeedUp** показывает отношение скорости работы параллельной версии алгоритма к скорости работы последовательной версии.

VERTICES	DENSITY	TIMEBFS	TIMEPARALLELBFS	SPEEDUP
10	0.1	15 ± 0.03	53 ± 0.2	3.5
10	0.3	20 ± 0.4	78 ± 0.3	3.9
10	0.7	27 ± 0.5	60 ± 0.4	2.2
10	0.9	25 ± 0.6	44 ± 0.4	1.76

Аналогично, чтобы исследовать оптимальное разделение на подзадачи для функции умножения, были проведены замеры с теми же размерами и плотностями графа, но с использованием последовательной функции сложения. Результаты приведены в таблице 4.

В ходе замеров было установлено, что при увеличении размеров графа, вне зависимости от плотности, время работы алгоритма при 16 подзадачах для функции умножения хоть и больше, чем при работе с 64 подзадачами, но разрыв становится все меньше и меньше. Вследствие этого наблюдения, было решено произвести замеры для параллельных версий алгоритма для графов гораздо большего размера, но малень-

Таблица 4: Производительность параллельного алгоритма обхода графов в ширину с использованием последовательной функции сложения векторов.

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	0.7	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	0.7	1	98 ± 1.6	5000	0.7	2	2571 ± 39
1000	0.9	1	90 ± 1.1	5000	0.9	2	2372 ± 41

кой плотности с целью выяснить, сможет ли версия с 64 подзадачами обогнать по производительности версию с 16 подзадачами. Результаты приведены в таблице 5.

Таблица 5: Сравнение производительности алгоритма обхода в ширину при использовании 16 и 64 подзадач 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	0.04 ± 0.0003	1.25
2500	0.1	2	0.24 ± 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 v1.10.1³, NumPy v1.24.3⁴ и Matplotlib v3.7.1⁵ 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 6.

С помощью функций `scipy.stats.shapiro()` (тест Шапиро-Уилка) и `scipy.stats.normaltest()` (критерий согласия Пирсона) было про-

³<https://scipy.org/> (дата доступа: 21 мая 2023 г.).

⁴<https://numpy.org/> (дата доступа: 21 мая 2023 г.).

⁵<https://matplotlib.org/> (дата доступа: 21 мая 2023 г.).

Таблица 6: 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

верено 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.

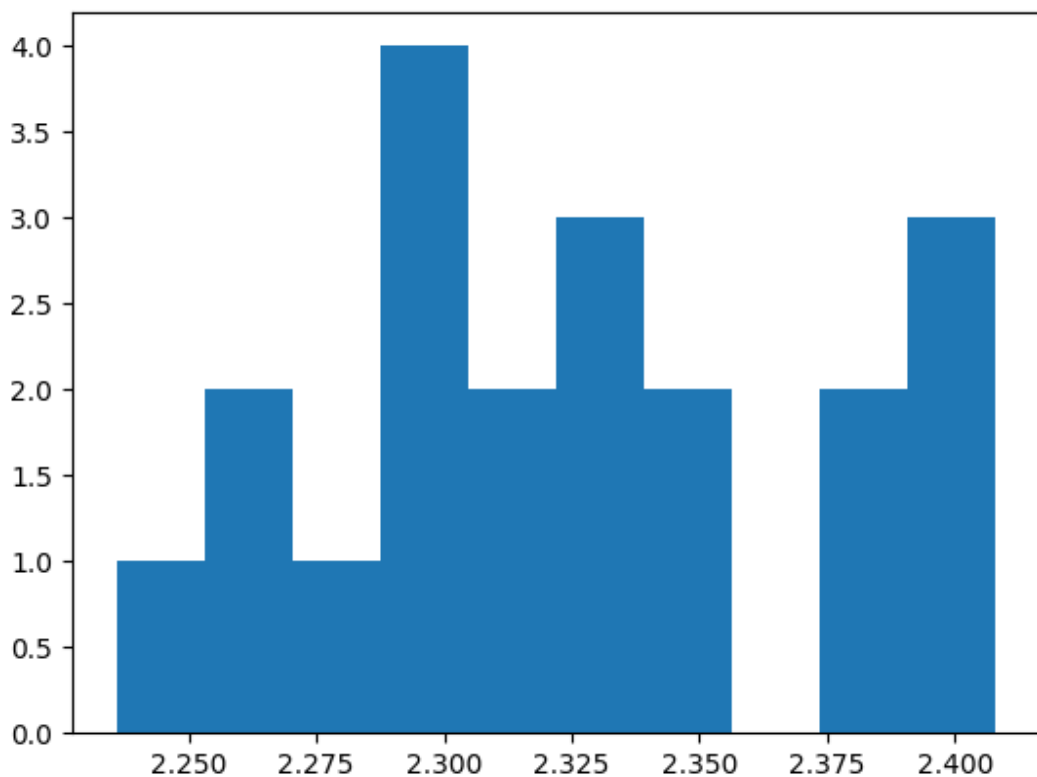


Рис. 3: Гистограмма распределения данных

4.4.1. RQ1

Исходя из полученных данных, последовательная версия алгоритма работает быстрее параллельной только на очень малых графах, когда накладные расходы на выделение дополнительных потоков и их синхронизацию влияют сильнее, чем ускорение относительно последовательной версии. При входных графах от 100 вершин и более, параллельная версия даёт внушительный прирост к производительности 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.

- Реализованы 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.