Санкт-Петербургский государственный университет

Группа 22.Б07-мм

Савельева Полина Андреевна

Экспериментальное исследование производительности алгоритма обхода графа в ширину

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.

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.

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 главным образом представляет собой обход всех вершин и рёбер. Два наиболее известных алгоритма поиска — это обход в ширину *Breadth-First Search* (BFS) и обход в длину Depth-First Search (DFS). В данной работе алгоритм DFS будет упомянут лишь косвенно, но нельзя не упомянуть его основное отличие от BFS — обход в ширину последовательно рассматривает смежные вершины, а обход в длину — вершины, расположенные по одному пути. Благодаря этому различию в BFS возможно использование матрично-векторных операций (SpMSpV) для эффективной работы с разреженными данными [5].

Вследствие разделения данных на части и обработки фрагментов в разных потоках, операции над векторами и матрицами могут быть эффективно *распараллелены*. Это особенно полезно в масштабных вы-

числениях, где параллельная реализация может значительно ускорить общее время работы.

Экспериментальное исследование является важным аспектом при разработке эффективного алгоритма. Во-первых, экспериментальный анализ позволяет оценить производительность метода на реальных данных и его пригодность для практического применения. Во-вторых, 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.

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 и без параллельной версии матрично-векторных операций.

2. Обзор

2.1. Терминология

Граф — это упорядоченная пара (V, E), где V множество вершин, а E множество рёбер, соединяющих вершины.

Ориентированный граф — это упорядоченная пара (V, E), где V множество вершин, а $E \subseteq V \times V$ множество упорядоченных пар вершин, называемых дугами.

Помеченный граф — это упорядоченная пара (G, μ), где $G = (V, E)$ граф с множеством вершин V и множеством рёбер E, и $\mu : E \to L$ отображение, сопоставляющее каждому ребру $e \in E$ метку из множества L.

Весовая матрица графа — это квадратная матрица $A = (a_{ij})$ порядка n, где n количество вершин, и элемент a_{ij} равен весу ребра, соединяющего вершины i и j, или значению, указывающему на отсутствие ребра между ними.

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

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

Дерево — это структура данных, состоящая из вершин (узлов), связанных между собой рёбрами (ветвями). Начальный узел структуры называется корневым и не имеет ни одного узла-родителя. Каждый узел дерева (кроме листового) может иметь неограниченное количество узлов-потомков.

Бинарное дерево — это дерево, каждый узел которого (за исключением корневого) имеет ровно два родителя.

2.2. Представление разреженных структур

Рассмотрим несколько существующих форматов хранения разреженных матриц. Большинство методов основаны на определённых свойствах конкретных структур, следовательно, степень их эффективности может разительно отличаться.

Наивный способ представления разреженных структур предполагает хранение всех элементов, включая нулевые. 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 [4]. 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 каком-либо конкретном свойстве и могут быть использованы для хранения любой разреженной матрицы [4]. 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 [2].

Дерево квадрантов — это структура данных, часто используемая для представления разреженных двумерных данных. *Q-Tree* разбивается на четыре части (реже на любое другое количество частей), называе-

мых *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. При поиске значения в Q-Tree необходимо пройти по дереву, начиная от самого корня. Этот процесс может занимать достаточно много времени, особенно в случаях, когда большое количество данных сосредоточенно в одной области. Но несмотря на другие особенности формата дерево квадрантов удобно использовать для параллельных вычислений.

2.3. Алгоритм обхода графа в ширину

Приведём неформальное описание работы алгоритма обхода графа в ширину:

1. Инициализировать очередь — структуру данных для хранения вершин, которые необходимо посетить — и пустой массив для хранения посещённых вершин. Добавить вершину(-ы), с которой(-ых) начинается поиск, в очередь.

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

3. Добавить вершины, которые соединены ребром с текущей, в очередь.

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

Как было упомянуто раннее, линейная алгебра может быть полезна при реализации некоторых этапов BFS, например, для эффективного представления графа или для выполнения операций над матрицами и векторами внутри алгоритма. Далее рассмотрим реализацию шагов 1–4 с использованием векторно-матричных операций. На рис. 1 представлен пример работы этого алгоритма:

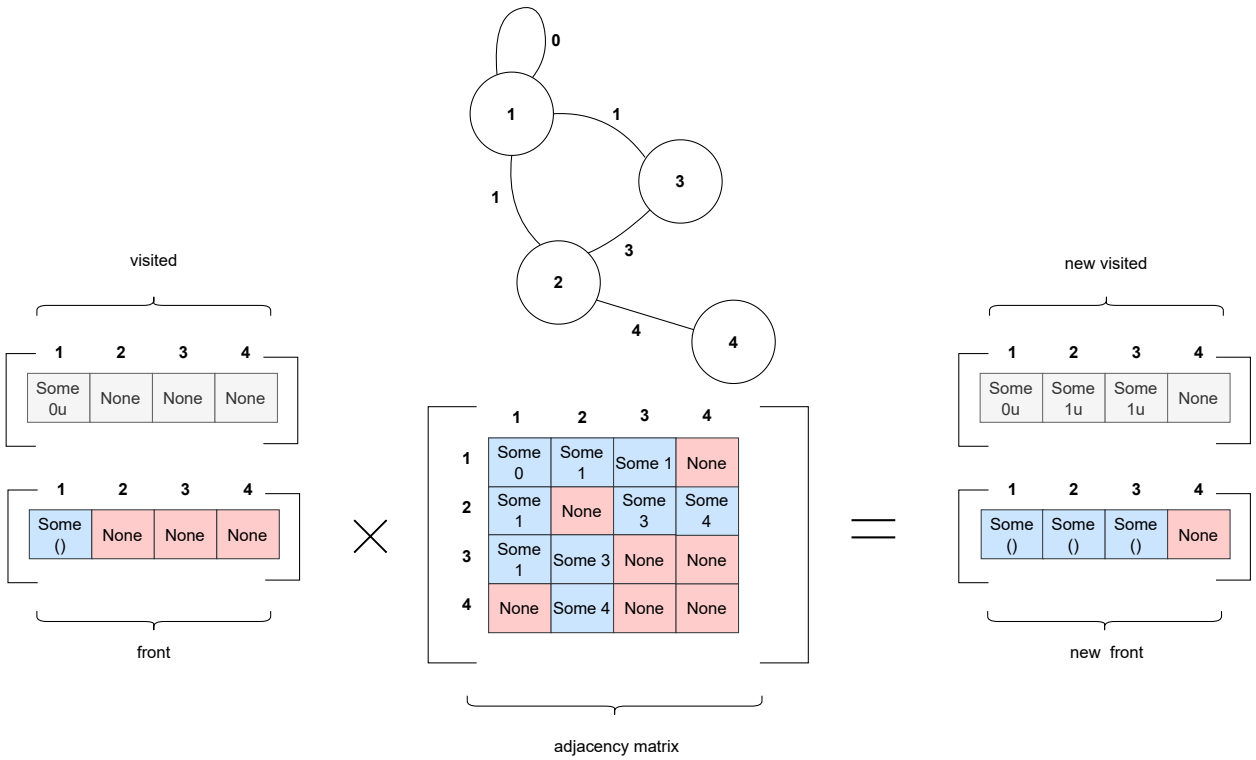

Рис. 1: Шаги 3–4 BFS с использованием линейной алгебры

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 вершину(-ы), с которой(-ых) 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.

2.4. Параллельные вычисления и многопоточность

Большинство несложных программ, с большой вероятностью выполняются в одном *потоке* (*thread*). Другими словами, в любой момент времени только один оператор находится в процессе выполнения; и если поток блокируется, то вся работа программы останавливается [3].

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», эффективно использовать больше потоков, чем имеющихся ядер [1].

При реализации многопоточности необходимо учитывать следующие аспекты:

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 дополнительные ресурсы.

4. *Зависимость от алгоритмов и данных.* Некоторые алгоритмы (данные) не могут быть эффективно распараллелены из-за своей природы (например, если операция зависит от результата предыдущего шага).

5. *Зависимость от аппаратного обеспечения.* Некоторые системы

имеют ограничения на количество доступных ядер, что ограничивает возможности параллельных вычислений.

3. Архитектура решения

3.1. Детали реализации SpMSpV

На платформе .NET существует поддержка параллельных вычислений с использованием асинхронных (т. е не блокирующих выполнение других процессов) выражений `tasks`. Они позволяют разбить работу программы на небольшие подзадачи и запустить их параллельно на доступных вычислительных ядрах.

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` (листинг. 1) формируются две задачи, исполняемые параллельно, для ненулевого `level -- 1` формируются ещё две подзадачи и т. д.

Листинг 1: Часть функции сложения векторов, отвечающая за параллельную составляющую векторной операции.

```
33 | BinTree.Node (x, y), BinTree.Node (z, w) ->
34     if parallelLevel = 0u then
35         let left = treesAddition 0u x z
36         let right = treesAddition 0u y w
37 
38         if left = BinTree.None && right = BinTree.None then
39             BinTree.None
40         else
41             BinTree.Node(left, right)
42     else
43         let tasks =
44             [| async { return treesAddition (parallelLevel - 1u) x z };
45             async { return treesAddition (parallelLevel - 1u) y w } |]
46 
47         let results = tasks |> Async.Parallel |> Async.RunSynchronously
```

Аналогично сложению используются `multiLevel`, `addLevel` в функции `multiplication`, однако в процессе формируются четыре подзадачи вместо двух (листинг. 2). Это различие обусловлено строением вектора и матрицы: узлы дерева, представляющего вектор, имеют ровно два потомка, а узлы дерева, представляющие матрицу — четыре. Примерная визуализация распределения задач между узлами вектора при сложении представлена на рис. 2, а последовательность выполнения операции — на рис. 3.

Листинг 2: Часть функции умножения вектора и матрицы, отвечающая за параллельную составляющую векторно-матричной операции.

```
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 leftTree2 = multiResults[1]
    let rightTree1 = multiResults[2]
    let rightTree2 = multiResults[3]

    let addTasks =
       [| async { return (vectorAddition addLevel plusOperation
                   leftTree1 leftTree2).Storage }
          async { return (vectorAddition addLevel plusOperation
                   rightTree1 rightTree2).Storage } |]

    let addResults = addTasks |> Async.Parallel |> Async.RunSynchronously
```

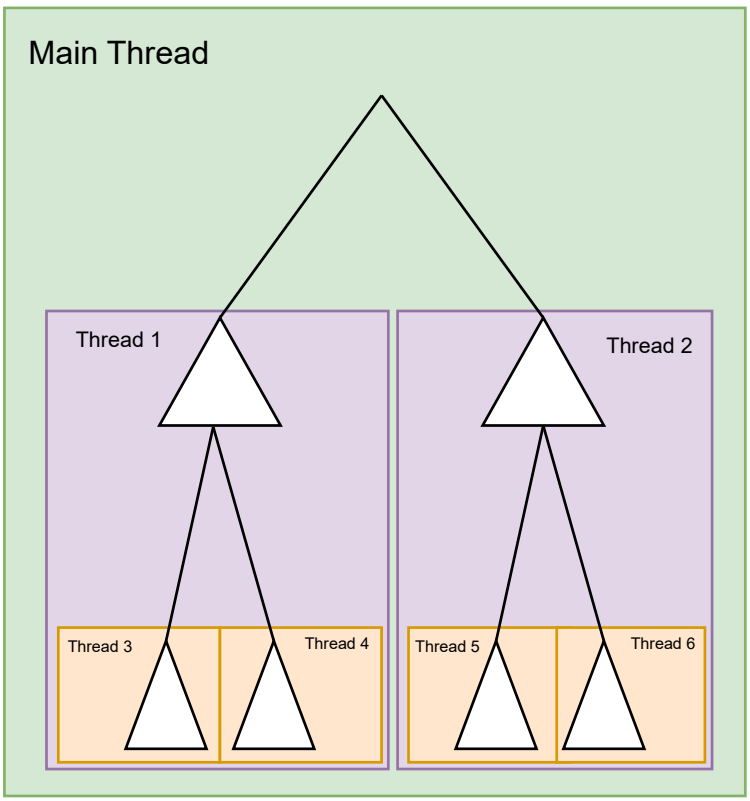

Рис. 2: Распределение потоков между узлами дерева, представляющего разреженный вектор, при сложении

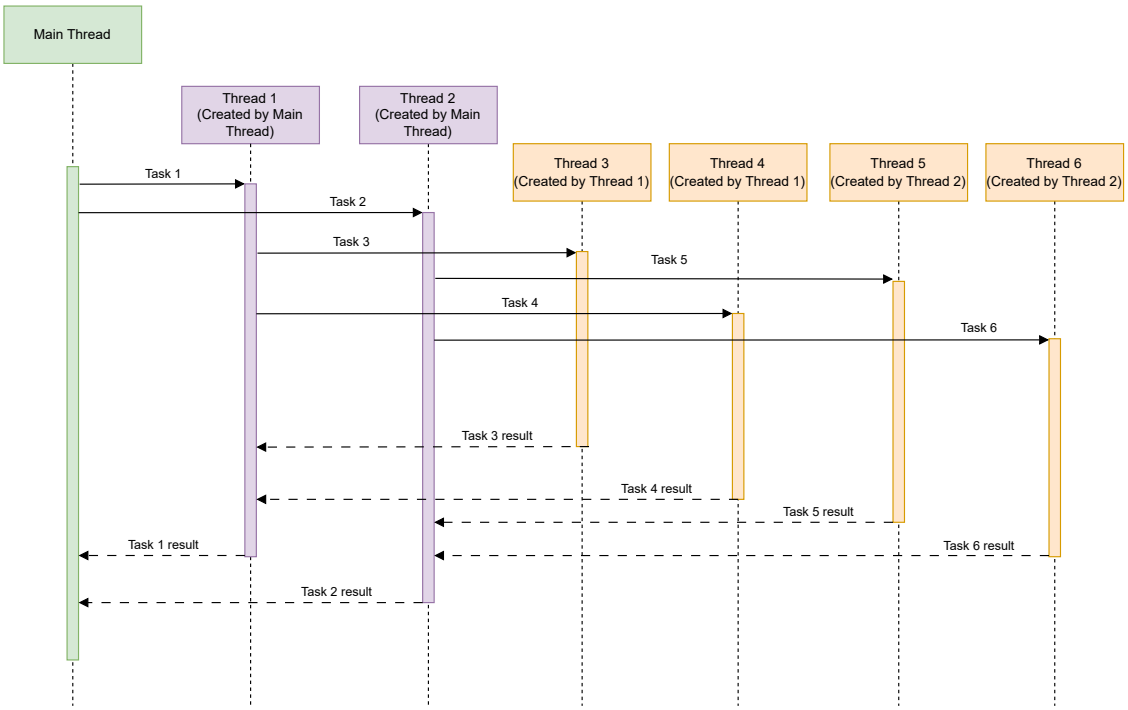

Рис. 3: Последовательность выполнения операции сложения векторов с использованием многопоточности

3.2. Реализация BFS

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

```
32 let BFS startVertexList (graph: Graph<'Value>) =
33
34    let rec inner (front: Vector<unit>) visited iterationNumber =
35       if front.IsEmpty then
36           visited
37       else
38           let newFront = multiplication 0u 0u fPlus fMulti front graph.
                AdjacencyMatrix
39
40           let front = vectorAddition 0u fPlusMask newFront visited
41
42           let visited = vectorAddition 0u (fPlusVisited iterationNumber)
                visited front
43
44           inner front visited (iterationNumber + 1u)
45
46    let front = Vector(startVertexList, graph.VerticesCount, ())
47    let visited = Vector(startVertexList, graph.VerticesCount, 0u)
48
49    inner front visited 1u
```

Алгоритм BFS на листинге. 3 в точности реализует шаги 1–4, упомянутые в обзоре. Неочевидным остаётся шаг получения маски; в действительности необходимые действия выполняются в строке 40 без инициализации самой `mask`.

Функция `fPlusMask` на листинге. 4 используется при сложении нового фронта и вектора посещённых вершин — на месте `front` получается ненулевое значение, только когда вершину предстоит посетить впервые.

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

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

4. Эксперимент

В данном разделе будут рассмотрены результаты экспериментального исследования алгоритма, приведённого в предыдущих главах. Основная задача исследования: оценить производительность BFS в условиях, близких к реальным, сравнить полученные показатели и ответить на вопросы RQ1-RQ3.

4.1. Условия эксперимента

Эксперименты проводились на рабочей станции со следующими характеристиками:

- Центральный процессор: AMD Ryzen 5 5600X
- Количество ядер ЦПУ: 6-Core Processor, 12 Threads
- Базовая тактовая частота ЦПУ: 3.70 GHz
- Объём оперативной памяти: 16.0 GB
- Операционная система: Windows 11 Pro, version 22H2

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?

RQ3: Чем можно объяснить такое количество потоков?

4.3. Метрики

Для замеров производительности используется пакет BenchmarkDotNet v0.13.4. В качестве метрик производительности выступает время, требуемое для выполнения операции.

Для всех экспериментов используются неориентированные графы — то есть графы с симметричной матрицей смежности. Примеры реальных данных взяты из MatrixMarket, разреженность этих данных не превышает 2 процента. Данные различной степени разреженности генерируются с помощью алгоритма листинг. какой-то.

В данной работе результат работы БФС не зависит от типа весов на рёбрах (в ответе учитывается только порядок посещения вершин), то элементы матриц, взятых в качестве экспериментальных данных, будут принадлежать Pattern-типу. Другими словами, элемент либо есть (Some() в реализации) или его нет (None в реализации), без привязки к конкретному значению. Необходимость использования элементов одного строения для проведения серии экспериментов обусловлено тем, что сравнения и другие арифметические операции с одними типами могут быть более ресурсоёмкий по сравнению с другими.

4.4. Результаты

4.4.1. RQ1

Пояснения

4.4.2. RQ2

Пояснения

4.4.3. RQ3

Пояснения

Заключение

Список литературы

[1] 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.

[2] 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.

[3] Sarcar Vaskaran. Threading // Test Your Skills in C# Programming: Review and Analyze Important Features of C#. — Springer, 2022. — P. 385–429.

[4] 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.

[5] 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.