Санкт-Петербургский государственный университет
Математико-механический факультет
Искусственный интеллект и наука о данных

Доклад на тему

Граф алгоритма и параллелизм: как структура зависимостей определяет скорость вычислений

Автор: Мурадян Денис Степанович

Уровень: Бакалавриат, 3 курс

Дисциплина: Параллельное программирование

2025

1. Аннотация

Доклад посвящён рассмотрению алгоритмов параллельных вычислений с точки зрения их графовой структуры. Вместо того чтобы исходить только из архитектуры вычислительной системы или конкретных технологий распараллеливания, анализ проводится на уровне *графа алгоритма*: ориентированного ациклического графа зависимостей между элементарными операциями.

В работе обсуждаются основные понятия, связанные с таким представлением: граф зависимостей, ориентированные ациклические графы (DAG), топологический порядок выполнения, параллельные формы графа, ярусы, высота и ширина. На этой основе вводится понятие *критического пути* и формулируется концепция *неограниченного параллелизма*, в которой время выполнения алгоритма определяется исключительно структурой его графа.

Для иллюстрации используются два показательных примера: вычисление произведения n чисел (сравнение последовательной схемы и процесса сдваивания) и сортировка слиянием, естественным образом задающая дерево рекурсий и граф операций слияния. На этих примерах показано, как изменение структуры алгоритма может радикально изменить высоту графа и, соответственно, потенциальное ускорение при распараллеливании.

В заключительной части кратко обсуждаются связи между теоретическими конструкциями и практическими системами: DAG-планировщиками в фреймворках обработки данных (Spark, Dask) и вычислительными графами в современных системах машинного обучения (TensorFlow, JAX). Проводится параллель между классическими результатами, изложенными в учебниках И. Г. Буровой, Ю. К. Демьяновича и В. В., Вл. В. Воеводиных, и современными реализациями параллелизма.

Задачи работы

1. Описать представление алгоритма в виде ориентированного ациклического графа зависимостей и выделить основные виды зависимостей (по данным и по управлению).

2. Ввести понятия параллельной формы графа, ярусов, высоты и ширины, интерпретировать их в терминах времени выполнения и степени параллелизма.

3. На примере вычисления произведения n чисел и сортировки слиянием показать влияние структуры алгоритма на высоту графа и возможное ускорение.

4. Сформулировать понятие критического пути и объяснить концепцию неограниченного параллелизма, в которой длина критического пути задаёт нижнюю границу времени выполнения.

5. Показать связь теоретических конструкций с практическими системами параллельных вычислений, использующими DAG в качестве основной модели исполнения.

Содержание

2. Введение

Параллельное программирование традиционно рассматривается в двух взаимосвязанных плоскостях. С одной стороны, исследуется архитектура вычислительных систем: многопроцессорные и многоядерные машины, системы с общей и распределённой памятью, графы связности, особенности подсистемы памяти. С другой стороны, анализируется структура самих алгоритмов и то, каким образом они могут быть адаптированы или переосмыслены для эффективной работы на таких архитектурах.

На практике часто возникает соблазн связывать возможное ускорение в первую очередь с количеством доступных процессоров. Однако классические результаты теории параллельных вычислений, подробно изложенные, например, в работах Буровой и Демьяновича, Воеводиных, показывают, что фундаментальные ограничения задаются не архитектурой как таковой, а *структурой алгоритма*. Даже при наличии огромного числа процессоров нельзя превысить теоретический предел ускорения, если в алгоритме присутствует значительная строго последовательная часть.

Для исследование этой структуры удобно представлять алгоритм в виде *графа зависимостей*: вершины соответствуют элементарным операциям или фрагментам вычислений, а рёбра фиксируют отношения предшествования, обусловленные зависимостями по данным и по управлению. Такой подход позволяет формализовать понятия *внутреннего параллелизма*, *критического пути*, *параллельной формы* и других характеристик, непосредственно связанных с временем выполнения алгоритма в различных моделях вычислений.

Цель данного доклада — изложить ключевые идеи графового подхода к анализу параллелизма алгоритмов на доступном примере. В центре внимания находятся следующие вопросы:

- как из алгоритма строится граф зависимостей;
- как по графу определить возможный параллелизм и нижнюю границу времени выполнения;
- как изменение алгоритма (при сохранении решаемой задачи) влияет на высоту графа;
- каким образом эти идеи реализуются в современных системах параллельной обработки данных.

В последующих разделах последовательно вводятся необходимые определения, рассматриваются простые примеры и формулируются выводы, которые затем соотносятся с практическими реализациями.

3. Алгоритм как граф зависимостей

3.1. Ориентированный граф операций

Рассмотрим алгоритм, состоящий из конечного числа элементарных операций (шагов). Под элементарной операцией в дальнейшем будем понимать такой шаг, который в выбранной модели вычислений выполняется неделимо и в течение одного такта логического времени.

Будем представлять алгоритм в виде ориентированного графа

$$G = (V, E),$$

где множество вершин V соответствует множеству операций, а множество рёбер E описывает зависимости между ними.

Если результат операции $u \in V$ используется в операции $v \in V$, то между вершинами u и v проводится ориентированное ребро $(u, v) \in E$. Это означает, что операция v не может быть начата до завершения операции u. Такого рода зависимости называют *зависимостями по данным*. Они автоматически задают частичный порядок выполнения операций.

Помимо зависимостей по данным, в алгоритме присутствуют и *зависимости по управлению*. Они возникают в результате наличия условных операторов и циклов: выбор ветви исполнения или решение о продолжении цикла зависят от уже вычисленных значений и, следовательно, накладывают ограничения на порядок последующих операций.

В совокупности зависимости по данным и по управлению образуют ориентированный граф, который будем называть *графом зависимостей* алгоритма. Именно этот граф определяет, какие операции обязаны выполняться раньше других, а какие операции могут быть выполнены независимо и, в принципе, параллельно.

3.2. Ацикличность и ориентированные ациклические графы

Структура зависимостей, возникающая при корректном определении алгоритма, обладает важным свойством: в ней не должно быть ориентированных циклов.

Предположим, что в графе зависимостей существует цикл

$$v_1 \to v_2 \to \cdots \to v_k \to v_1.$$

Это означает, что операция v_1 (через цепочку других операций) зависит от самой себя. Выполнить такой алгоритм невозможно: для начала выполнения v_1 необходимо заранее знать её же результат.

Поэтому граф корректного алгоритма должен быть *ориентированным ациклическим*, то есть не содержать ориентированных циклов. Такие графы называются *DAG* (от англ.

directed acyclic graph).

В дальнейшем будем исходить из предположения, что граф алгоритма всегда является DAG. Все последующие определения — топологический порядок, параллельные формы, критический путь — формулируются именно для ориентированных ациклических графов.

Для иллюстрации приведём небольшой пример DAG (формально он будет использован ниже при объяснении высоты и ширины графа).

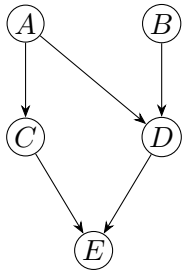

Рис. 1: Пример ориентированного ациклического графа зависимостей

3.3. Топологический порядок и план выполнения

Ориентированный ациклический граф задаёт *частичный* порядок на множестве операций: далеко не все вершины сравнимы между собой по отношению «должна быть выполнена раньше». Для практической реализации алгоритма требуется выбрать *конкретный* порядок исполнения.

Топологический порядок (или топологическая сортировка) ориентированного ациклического графа $G = (V, E)$ — это такой линейный порядок вершин $v_{i_1}, v_{i_2}, \ldots, v_{i_{|V|}}$, что для любого ребра $(u, v) \in E$ вершина u встречается в этом порядке раньше вершины v. Известно, что ориентированный граф допускает топологический порядок тогда и только тогда, когда он является ацикличным.

Топологическая сортировка позволяет перейти от частичного порядка, заданного зависимостями, к конкретному плану выполнения операций. Любой топологический порядок задаёт допустимую последовательность исполнения алгоритма, совместимую со всеми зависимостями по данным и по управлению.

В системах параллельных вычислений и в компиляторах именно идеи топологической сортировки лежат в основе построения расписаний, планов выполнения и определения точек синхронизации.

4. Граф алгоритма и параллельные формы

4.1. Граф алгоритма

Переходя от общего графа зависимостей к более структурированному объекту, будем использовать понятие *графа алгоритма* в духе классических работ по параллельным вычислениям.

Граф алгоритма определяется как ориентированный ациклический граф

$$G = (V, E),$$

где:

- вершины V соответствуют элементарным операциям или шагам алгоритма на выбранном уровне абстракции;
- рёбра E фиксируют отношения предшествования, порождённые зависимостями по данным и по управлению;
- на множестве вершин задан частичный порядок, согласованный с ориентацией рёбер.

Таким образом, граф алгоритма не просто фиксирует набор зависимостей, но и служит удобной моделью для анализа структуры вычислений: по нему можно судить о наличии независимых ветвей, последовательных цепочек, о «ширине» фронта параллельной работы и о минимально возможном времени выполнения при заданной модели вычислений.

4.2. Примеры зависимостей

Рассмотрим простейшие иллюстрации.

Арифметическое выражение. Пусть требуется вычислить выражение $(a + b) \cdot (c + d)$. Естественное дерево вычислений имеет вид:

$$a,\, b \xrightarrow{+} t_1, \quad c,\, d \xrightarrow{+} t_2, \quad t_1,\, t_2 \xrightarrow{\cdot} t_3.$$

Сложения $(a + b)$ и $(c + d)$ независимы и могут быть выполнены параллельно, однако умножение возможно только после того, как оба промежуточных результата будут получены. Граф алгоритма в этом случае представляет собой небольшое дерево, в котором чётко выделяются параллельный и последовательный фрагменты.

Цикл без и с рекуррентной зависимостью. Рассмотрим два варианта цикла:

$$x_i = f(i), \quad i = 1, \ldots, n,$$

и

$$x_i = x_{i-1} + g(i), \quad i = 1, \ldots, n.$$

В первом случае каждая итерация использует только значение индекса i, и между итерациями нет зависимостей: все n итераций в принципе могут выполняться параллельно. Во втором случае каждая итерация опирается на результат предыдущей, и значения x_i выстраиваются в цепочку; в графе алгоритма возникает длинный путь длины n, который задаёт строго последовательный фрагмент.

Подобные цепочки зависимостей являются основными источниками ограничений на ускорение: даже при большом числе процессоров операции, лежащие на такой цепочке, не могут быть распараллелены.

4.3. Ярусы и параллельная форма графа

Чтобы формально описать возможный параллелизм в графе алгоритма, вводится понятие *яруса* и *параллельной формы*.

Пусть $G = (V, E)$ — ориентированный ациклический граф. Под *ярусом* (или *уровнем*) будем понимать подмножество вершин $L \subseteq V$, которое может быть выполнено в один логический шаг времени при условии, что все предшественники вершин из L уже завершены.

Параллельной формой графа называем разбиение множества вершин на упорядоченную последовательность ярусов:

$$V = L_1 \cup L_2 \cup \cdots \cup L_k,$$

такое, что:

- множества $L_1, \ldots, L_k$ попарно не пересекаются;
- для любого ребра $(u, v) \in E$ вершина u принадлежит некоторому L_i, а вершина v — некоторому L_j с $i < j$.

Интуитивно это означает, что:

- все вершины первого яруса L_1 не имеют предшественников и могут быть выполнены одновременно;
- вершины яруса L_2 могут быть выполнены после завершения всех операций из L_1 и т.д.

Параллельная форма описывает один из возможных вариантов группировки операций по «шагам времени» в модели с неограниченным числом процессоров: на каждом шаге выполняются все операции соответствующего яруса.

4.4. Высота и ширина графа

Для параллельной формы вводятся две числовые характеристики, играющие ключевую роль в оценке времени выполнения и степени параллелизма.

- *Высота* графа G (обозначим её $h(G)$) определяется как минимальное число ярусов в параллельной форме графа:

  $$h(G) = \min\{k \mid \exists \text{ параллельная форма } V = L_1 \cup \cdots \cup L_k\}.$$

  В идеализированной модели с неограниченным числом процессоров высота задаёт минимальное число шагов логического времени, за которое в принципе может быть выполнен алгоритм.

- *Ширина* графа G (обозначим $w(G)$) — это максимальное число вершин в одном ярусе:

  $$w(G) = \max_j |L_j|.$$

  Ширина оценочно характеризует максимальное количество независимых операций, которые могут выполняться одновременно на каком-либо шаге.

Для графа на рис. 1, если разбить вершины на ярусы

$$L_1 = \{A, B\}, \quad L_2 = \{C, D\}, \quad L_3 = \{E\},$$

то высота $h(G)$ равна трём, а ширина $w(G)$ — двум.

5. Примеры алгоритмов и структура параллелизма

В этом разделе рассмотрим два простых, но показательных примера, в которых одна и та же вычислительная задача может решаться алгоритмами с существенно различной структурой графа, а значит — с различной потенциальной степенью параллелизма.

5.1. Произведение n чисел: последовательный алгоритм

Пусть требуется вычислить произведение n чисел:

$$P = a_1 a_2 \cdots a_n.$$

Самый простой способ — *последовательное* умножение слева направо:

$$t_1 = a_1 a_2, \quad t_2 = t_1 a_3, \quad \ldots, \quad t_{n-1} = t_{n-2} a_n.$$

Каждое последующее умножение использует результат предыдущего, поэтому в графе алгоритма возникает цепочка из $n-1$ операций. При $n = 8$ схема выполнения имеет вид:

Данные:	$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8$
Шаг 1:	$a_1 a_2$
Шаг 2:	$(a_1 a_2) a_3$
Шаг 3:	$(a_1 a_2 a_3) a_4$
Шаг 4:	$(a_1 a_2 a_3 a_4) a_5$
Шаг 5:	$(a_1 a_2 a_3 a_4 a_5) a_6$
Шаг 6:	$(a_1 a_2 a_3 a_4 a_5 a_6) a_7$
Шаг 7:	$(a_1 a_2 a_3 a_4 a_5 a_6 a_7) a_8$

Если интерпретировать каждый шаг как отдельный ярус параллельной формы, то высота графа равна $n-1$ (для $n = 8$ — 7), а ширина на каждом шаге равна единице: в каждый момент времени активна ровно одна операция умножения. Даже при наличии большого числа процессоров этот алгоритм использует только один из них, а остальные простаивают. В терминах параллелизма граф имеет одну длинную последовательную цепочку и ширину, равную единице.

5.2. Произведение n чисел: процесс сдваивания

Рассмотрим альтернативный алгоритм, использующий *процесс сдваивания* (дерево умножений). Идея состоит в том, чтобы вначале выполнять попарные произведения, затем попарно перемножать полученные результаты и так далее, пока не останется одно значение.

Для $n = 8$ схема имеет вид:

Данные:	$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8$
Шаг 1:	$(a_1 a_2)\ (a_3 a_4)\ (a_5 a_6)\ (a_7 a_8)$
Шаг 2:	$(a_1 a_2 a_3 a_4)\ (a_5 a_6 a_7 a_8)$
Шаг 3:	$(a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8)$

На первом шаге выполняются четыре независимых умножения; на втором — два умножения результатов; на третьем — одно итоговое умножение.

Граф алгоритма в этом случае представляет собой *дерево* высоты $\lceil \log_2 n \rceil$. В первой

параллельной форме высота равна трём, а ширина первого яруса равна четырём.

В общем случае при n множителях процесс сдваивания даёт:

- высоту порядка $\lceil \log_2 n \rceil$;
- ширину порядка $n/2$ на первом ярусе.

Сравнивая два алгоритма, видим, что:

- в последовательной схеме высота графа растёт линейно, $h(G) = O(n)$;
- в схеме сдваивания высота растёт логарифмически, $h(G) = O(\log n)$.

Оба алгоритма выполняют примерно одинаковое число операций умножения (на практике даже одинаковое — $n - 1$), но структура графа радикально различается. Во втором случае последовательная часть алгоритма существенно короче, что при наличии достаточного числа процессоров даёт принципиально иные возможности для ускорения.

5.3. Сортировка слиянием

В качестве более содержательного примера рассмотрим сортировку слиянием (merge sort) на массиве из n элементов.

Алгоритм сортировки слиянием можно описать следующим образом:

1. Если длина массива равна 1, он уже отсортирован.
2. Иначе массив делится пополам на левую и правую части.
3. Рекурсивно сортируются левая и правая части.
4. Два отсортированных подмассива сливаются в один отсортированный массив.

Эта схема естественным образом задаёт *дерево рекурсий*: в корне дерева находится задача сортировки всего массива, на следующем уровне — сортировка левой и правой половин, затем каждая половина снова делится пополам и так далее до подмассивов длины один.

Глубина дерева рекурсий пропорциональна $\log_2 n$, поскольку на каждом уровне длина подмассива уменьшается вдвое. На k-м уровне дерева одновременно присутствует 2^k независимых подзадач сортировки, каждая из которых работает с массивом длины $n/2^k$.

Операции слияния также организуются в уровни: сначала сливаются пары элементов, затем пары массивов длины 2, затем длины 4 и т.д.

Граф алгоритма сортировки слиянием можно условно разделить на две части:

- дерево рекурсивных вызовов сортировки подмассивов;
- DAG операций слияния.

Каждый уровень слияния содержит несколько независимых операций, которые могут выполняться параллельно. Структура такого графа иллюстрирует типичную картину для *divide-and-conquer*-алгоритмов: имеется логарифмическая по n глубина разбиения и широкая «крона» независимых задач на каждом уровне.

Точные оценки времени выполнения в различных моделях параллельных вычислений зависят от того, каким образом реализуются операции слияния и какие ресурсы считаются ограниченными. Для целей данного доклада важно подчеркнуть, что сам подход к оценке опирается на анализ высоты и ширины графа алгоритма, а также на выделение критического пути.

6. Критический путь и концепция неограниченного параллелизма

6.1. Критический путь

Пусть $G = (V, E)$ — граф алгоритма. *Путь* в графе — это последовательность вершин

$$v_0, v_1, \ldots, v_k,$$

такая, что для каждого $i = 0, \ldots, k-1$ существует ребро $(v_i, v_{i+1}) \in E$. Каждое ребро отражает отношение предшествования, поэтому путь соответствует цепочке операций, которые должны выполняться строго последовательно: выполнение v_{i+1} возможно только после завершения v_i.

Под *критическим путём* понимают путь максимальной длины (по числу операций или по суммарному времени, если операции имеют различную длительность) от некоторого «истока» к некоторому «стоку» графа. В простейшем случае, когда все операции имеют одинаковую длительность, критическим является путь с максимальным числом рёбер.

Длина критического пути совпадает с минимально возможной высотой параллельной формы:

$$h(G) = \text{длина критического пути}.$$

Интуитивно это означает, что как бы мы ни распараллеливали алгоритм, цепочку зависимостей, лежащую на критическом пути, всё равно придётся проходить последовательно.

В терминах моделей параллельных вычислений, где допускается неограниченное число процессоров, длина критического пути задаёт *нижнюю границу* времени выполнения алгоритма.

6.2. Концепция неограниченного параллелизма

Классический подход, развиваемый в учебнике Буровой и Демьяновича, опирается на *концепцию неограниченного параллелизма*. В рамках этой концепции рассматривается идеализированная вычислительная модель, в которой:

- процессоров может быть сколь угодно много;
- все процессоры универсальны и работают синхронно;
- доступ к памяти и передача данных происходят мгновенно и без конфликтов;
- накладные расходы на создание задач и синхронизацию не учитываются.

В такой модели единственным существенным ограничением остаются зависимости, закодированные в графе алгоритма. Подразумевается, что:

- на каждом шаге логического времени можно выполнить все операции соответствующего яруса;
- минимальное время выполнения алгоритма равно высоте графа $h(G)$, то есть длине критического пути.

Основной задачей в этой концепции становится построение алгоритмов минимальной высоты: чем короче критический путь, тем меньше теоретическое время выполнения в идеализированной модели. При этом общее количество операций может оставаться тем же, как в примере с последовательным умножением и процессом сдваивания, но их организация во времени меняется.

Разумеется, реальные системы накладывают дополнительные ограничения: число процессоров конечно, существуют задержки доступа к памяти, конфликты при передаче данных и т.д. Тем не менее концепция неограниченного параллелизма играет важную роль, поскольку задаёт *фундаментальную нижнюю границу*: реальное время выполнения не может быть меньше длины критического пути.

7. Внутренний параллелизм и практические системы

7.1. Внутренний параллелизм алгоритма

Под *внутренним параллелизмом* будем понимать параллелизм, который заложен в самой структуре алгоритма и отображается в его графе зависимостей.

Он определяется:

- длиной критического пути (минимально возможным числом шагов логического времени);

- шириной ярусов (максимальным числом независимых операций, которые могут выполняться одновременно).

Важно подчеркнуть, что внутренний параллелизм не появляется за счёт изменения архитектуры или технологий — он появляется, когда меняется *структура вычислений*. Пример с произведением n чисел наглядно демонстрирует это:

- при последовательной схеме критический путь имеет длину $O(n)$, ширина графа равна единице;
- при переходе к дереву сдваивания критический путь сокращается до $O(\log n)$, ширина первых ярусов становится порядка n.

Такие преобразования алгоритмов направлены именно на уменьшение высоты графа и увеличение числа независимых операций на каждом шаге. Именно выбор структуры алгоритма в конечном счёте определяет, какого ускорения можно добиться на практике даже при весьма совершенной вычислительной архитектуре.

7.2. Графы алгоритмов в современных фреймворках

Графовый подход к описанию вычислений широко используется в современных системах параллельной обработки данных и машинного обучения.

DAG-планировщики в системах обработки данных. Во фреймворке Apache Spark последовательность преобразований над распределёнными коллекциями данных (RDD или DataFrame) сначала описывается на высоком уровне (как набор трансформаций), после чего преобразуется в *граф задач* (job DAG). Планировщик анализирует этот DAG, разбивает его на стадии, выявляет независимые ветви и распределяет задачи по узлам кластера.

В системе Dask используется схожий подход, но граф формируется более динамично. Вычисления описываются в виде ленивого графа задач, где каждая вершина соответствует операции, а рёбра задают зависимости. Планировщик Dask анализирует этот граф, определяет, какие задачи могут запускаться параллельно, и управляет их выполнением на доступных ресурсах.

Вычислительные графы в системах машинного обучения. В фреймворках TensorFlow и JAX вычисления над тензорами часто представляются в виде *вычислительного графа*. Узлы такого графа соответствуют операциям (линейные преобразования, поэлементные функции, свёртки и т.д.), а рёбра — передаче тензоров между операциями.

Наличие явного графа позволяет:

- выполнять оптимизации порядка вычислений (в том числе фьюзинг операций);
- автоматически распределять операции по устройствам (CPU, GPU, TPU);

- использовать различные стратегии распараллеливания.

Во всех этих системах по сути реализуются идеи графа алгоритма и параллельных форм: исполнитель получает на вход DAG, учитывает зависимости и выбирает, какие операции запускать параллельно, а какие должны ждать результатов предшественников. Таким образом, теоретические конструкции, обсуждаемые в рамках концепции неограниченного параллелизма, находят прямое отражение в практических инструментах.

8. Заключение

В данной работе алгоритмы параллельных вычислений рассматривались с точки зрения их графовой структуры. Алгоритм моделировался ориентированным ациклическим графом зависимостей, где вершины соответствуют элементарным операциям, а рёбра фиксируют отношения предшествования, обусловленные зависимостями по данным и по управлению.

Были введены понятия параллельной формы графа, ярусов, высоты и ширины. В идеализированной модели с неограниченным числом процессоров высота графа совпадает с длиной критического пути и задаёт нижнюю границу времени выполнения алгоритма. Ширина характеризует потенциальный фронт параллельной работы на каждом шаге.

На примерах вычисления произведения n чисел и сортировки слиянием показано, что одна и та же вычислительная задача может решаться алгоритмами с радикально различной структурой графа: в одном случае последовательная часть выражена длинной цепочкой зависимостей, в другом — организована в виде логарифмически глубокого дерева с широкой «кроной» независимых операций. Это наглядно демонстрирует, что реальный предел ускорения определяется не только архитектурой вычислительной системы, но и тем, насколько удачно выбран алгоритм с точки зрения внутреннего параллелизма.

Обсуждена концепция неограниченного параллелизма, в рамках которой время выполнения определяется исключительно зависимостями в графе алгоритма, а основная задача сводится к построению алгоритмов минимальной высоты. Хотя эта модель идеализирована и не учитывает ряд практических ограничений, она задаёт фундаментальную нижнюю границу и служит удобным теоретическим ориентиром.

Наконец, проведены параллели с современными практическими системами: DAG-планировщиками в фреймворках обработки данных и вычислительными графами в системах машинного обучения. Во всех этих системах в явном или неявном виде используется тот же самый подход: алгоритм рассматривается как граф зависимостей, и планировщик стремится максимально использовать заложенный в нём внутренний параллелизм.

Литература

1. Бурова И. Г., Демьянович Ю. К. *Алгоритмы параллельных вычислений и программирование*. СПб.: Изд-во СПбГУ, 2007.

2. Воеводин В. В., Воеводин Вл. В. *Параллельные вычисления*. СПб.: БХВ-Петербург, 2004.

3. Grama A., Gupta A., Karypis G., Kumar V. *Introduction to Parallel Computing*. Addison–Wesley, 2003.