МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Тульский государственный университет»

Интернет-институт

ПРАКТИЧЕСКАЯ РАБОТА

по дисциплине

«Технологии программирования 2»

на тему

«Оценка сложности алгоритмов.
Показатели эффективности параллельного алгоритма»

Семестр 5

Вариант 3

Выполнил: студент гр. ИБ262521-ф

Артемов Александр Евгеньевич

Проверил: канд. техн. наук, доц.

Сафронова Марина Алексеевна

Тула 2024

Содержание

Вводная часть

Существует несколько способов измерения сложности алгоритма. Программисты обычно сосредотачивают внимание на скорости алгоритма, но не менее важны и другие показатели – требования к объёму памяти, свободному месте на диске. Использование быстрого алгоритма не приведёт к ожидаемым результатам, если для его работы понадобится больше памяти, чем есть у компьютера.

Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т.е. сложность при стремлении размера входных данных к бесконечности.

Допустим, некоторому алгоритму нужно выполнить 4n3 + 7n условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна O(n3), т.е. зависит от размера входных данных кубически.

Использование заглавной буквы O (или так называемая O-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).

Сложность алгоритмов

Асимптотический анализ

Когда мы говорим об измерении сложности алгоритмов, мы подразумеваем анализ времени, которое потребуется для обработки очень большого набора данных. Такой анализ называют асимптотическим. Сколько времени потребуется на обработку массива из десяти элементов? Тысячи? Десяти миллионов? Если алгоритм обрабатывает тысячу элементов за пять миллисекунд, что случится, если мы передадим в него миллион? Будет ли он выполняться пять минут или пять лет?

Порядок роста

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего он представлен в виде O-нотации (от нем. «Ordnung» — порядок): O(f(x)), где f(x) — формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n, представляющая размер входных данных. Ниже приводится список наиболее часто встречающихся порядков роста, но он ни в коем случае не полный.

Константный — O(1)

Порядок роста O(1) означает, что вычислительная сложность алгоритма не зависит от размера входных данных. Следует помнить, однако, что единица в формуле не значит, что алгоритм выполняется за одну операцию или требует очень мало времени. Он может потребовать и микросекунду, и год. Важно то, что это время не зависит от входных данных.

O(n) — линейная сложность

Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.

Порядок роста O(n) означает, что сложность алгоритма линейно растет с увеличением входного массива. Если линейный алгоритм обрабатывает один элемент пять миллисекунд, то мы можем ожидать, что тысячу элементов он обработает за пять секунд.

Такие алгоритмы легко узнать по наличию цикла по каждому элементу входного массива.

O(log n) — логарифмическая сложность

Порядок роста O(log n) означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива. (Прим. пер.: в анализе алгоритмов по умолчанию используется логарифм по основанию 2). Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность.

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.

Линеарифметическая или линеаризованная — O(n·log n)

Линеарифметический (или линейно-логарифмический) алгоритм имеет порядок роста *O(n*·log *n)*. Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию. Означает, что удвоение размера входных данных увеличит время выполнения чуть более, чем вдвое. Примеры алгоритмов с такой сложностью: Сортировка слиянием или множеством n элементов.

$O(n^2)$ — квадратичная сложность

Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n * n, т.е. n2.

Бывают и другие оценки по сложности, но все они основаны на том же принципе.

Время работы алгоритма с порядком роста $O(n^2)$ зависит от квадрата размера входного массива. Несмотря на то, что такой ситуации иногда не избежать, квадратичная сложность — повод пересмотреть используемые алгоритмы или структуры данных. Проблема в том, что они плохо масштабируются. Например, если массив из тысячи элементов потребует 1000 000 операций, массив из миллиона элементов потребует 1 000 000 000 000 операций. Если одна операция требует миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов 32 года. Даже если он будет в сто раз быстрее, работа займет 84 дня.

Наилучший, средний и наихудший случаи

Что мы имеем в виду, когда говорим, что порядок роста сложности алгоритма — O(n)? Это усредненный случай? Или наихудший? А может быть, наилучший?

Обычно имеется в виду наихудший случай, за исключением тех случаев, когда наихудший и средний сильно отличаются. К примеру, мы увидим примеры алгоритмов, которые в среднем имеют порядок роста O(1), но периодически могут становиться O(n) (например, ArrayList.add). В этом случае мы будем указывать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает. Самое важное здесь то, что O(n) означает, что алгоритм потребует не более n шагов.

Параллельный алгоритм

В информатике параллельный алгоритм, противопоставляемый традиционным последовательным алгоритмам, — алгоритм, который может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением полученных результатов и получением корректного результата.

Некоторые алгоритмы достаточно просто поддаются разбиению на независимо выполняемые фрагменты. Например, распределение работы по проверке всех чисел от 1 до 100000 на предмет того, какие из них являются простыми, может быть выполнено путём назначения каждому доступному процессору некоторого подмножества чисел с последующим объединением полученных множеств простых чисел (похожим образом реализован, например, проект GIMPS).

С другой стороны, большинство известных алгоритмов вычисления значения числа π не допускают разбиения на параллельно выполняемые части, так как требуют результата предыдущей итерации выполнения алгоритма. Итеративные численные методы, такие как, например, метод Ньютона или задача трёх тел, также являются сугубо последовательными алгоритмами. Некоторые примеры рекурсивных алгоритмов достаточно сложно поддаются распараллеливанию. Одним из примеров является поиск в глубину на графах.

Параллельные алгоритмы весьма важны ввиду постоянного совершенствования многопроцессорных систем и увеличения числа ядер в современных процессорах. Обычно проще сконструировать компьютер с одним быстрым процессором, чем с множеством медленных процессоров (при условии достижения одинаковой производительности). Однако производительность процессоров увеличивается главным образом за счёт совершенствования техпроцесса (уменьшения норм производства), чему мешают физические ограничения на размер элементов микросхем и

тепловыделение. Указанные ограничения могут быть преодолены путём перехода к многопроцессорной обработке, что оказывается эффективным даже для малых вычислительных систем.

Сложность последовательных алгоритмов выражается в объёме используемой памяти и времени (числе тактов процессора), необходимых для выполнения алгоритма. Параллельные алгоритмы требуют учёта использования ещё одного ресурса: подсистемы связей между различными процессорами. Существует два способа обмена между процессорами: использование общей памяти и системы передачи сообщений.

Системы с общей памятью требуют введения дополнительных блокировок для обрабатываемых данных, налагая определённые ограничения при использовании дополнительных процессоров.

Системы передачи сообщений используют понятия каналов и блоков сообщений, что создаёт дополнительный трафик на шине и требует дополнительных затрат памяти для организации очередей сообщений. В дизайне современных процессоров могут быть предусмотрены специальные коммутаторы (кроссбары) с целью уменьшения влияния обмена сообщениями на время выполнения задачи.

Ещё одной проблемой, связанной с использованием параллельных алгоритмов, является балансировка нагрузки. Например, поиск простых чисел в диапазоне от 1 до 100000 легко распределить между имеющимися процессорами, однако некоторые процессоры могут получить больший объём работы, в то время как другие закончат обработку раньше и будут простаивать. Проблемы балансировки нагрузки ещё больше усугубляется при использовании гетерогенных вычислительных сред, в которых вычислительные элементы существенно отличаются по производительности и доступности (например, в грид-системах).

Разновидность параллельных алгоритмов, называемая распределёнными алгоритмами, специально разрабатываются для

применения на кластерах и в распределённых вычислительных системах с учётом ряда особенностей подобной обработки.

Оценка параллельных алгоритмов

При разработке параллельных алгоритмов решения задач вычислительной математики принципиальным моментом является анализ эффективности использования параллелизма, состоящий обычно в оценке получаемого ускорения процесса вычисления (сокращения времени решения задачи).

Для упрощения оценки алгоритмов вводится ряд допущений:

параллельная вычислительная система, на которой будет выполняться алгоритм, состоит из любого нужного числа процессоров и произвольно большой памяти, одновременно доступной всем процессорам.

каждый процессор за единицу времени может выполнить любую унарную или бинарную операцию.

время выполнения всех вспомогательных операций, а также время взаимодействия с памятью и время, затрачиваемое на управление процессом, считаются пренебрежимо малыми.

все входные данные перед началом вычислений записаны в памяти. Каждый процессор считывает свои операнды из памяти и после выполнения операции записывает результат в память.

после окончания вычислительного процесса все результаты остаются в памяти.

все процессоры и устройство памяти объединяются в единую систему, связанную каналами передачи информации.

Показатели эффективности параллельного алгоритма

Принципиальный момент при разработке алгоритмов — анализ эффективности использования параллелизма:

оценка максимально возможного ускорения процесса решения рассматриваемой задачи (анализ всех возможных способов выполнения вычислений);

оценка эффективности распараллеливания конкретных выбранных методов выполнения вычислений.

Ускорение (speedup) получаемое при использовании параллельного алгоритма для p процессоров, по сравнению с последовательным вариантом выполнения вычислений, определяется величиной

$$S_p(n) = T_1(n) / T_p(n)$$

т.е. как отношение времени решения задач на скалярной ЭВМ к времени выполнения параллельного алгоритма (величина n используется для параметризации вычислительной сложности решаемой задачи и может пониматься, например, как количество входных данных задачи).

Эффективность (efficiency) использования параллельным алгоритмом процессоров при решении задачи определяется соотношением:

$$E_p(n) = T_1(n) / (pT_p(n)) = S_p(n)/p$$

Величина эффективности определяет среднюю долю времени выполнения параллельного алгоритма, в течение которого процессоры реально используются для решения задачи.

Из приведенных соотношений можно показать, что в наилучшем случае $S_p(n) = p$ и $E_p(n) = 1$. При практическом применении данных показателей для оценки эффективности параллельных вычислений следует учитывать два важных момента:

1. При определенных обстоятельствах ускорение может оказаться больше числа используемых процессоров ускорение $S_p(n) > p$ может иметь место в силу следующего ряда причин: в этом случае говорят о существовании сверхлинейного ускорения. Несмотря на парадоксальность таких ситуаций (ускорение превышает число процессоров), на практике сверхлинейное ускорение может иметь место. Одной из причин такого явления может быть неодинаковость условий выполнения последовательной и параллельной программ. Например, при решении задачи на одном процессоре оказывается недостаточно оперативной памяти для хранения всех обрабатываемых данных и тогда становится необходимым использование более медленной внешней памяти (в случае же использования нескольких процессоров оперативной памяти может оказаться достаточно за счет разделения данных между процессорами). Еще одной причиной сверхлинейного ускорения может быть нелинейный характер зависимости сложности решения задачи от объема обрабатываемых данных. Так, например, известный алгоритм пузырьковой сортировки характеризуется квадратичной зависимостью количества необходимых операций от числа упорядочиваемых данных. Как результат, при распределении сортируемого массива между процессорами может быть получено ускорение, превышающее число процессоров. Источником сверхлинейного ускорения может быть и различие вычислительных схем последовательного и параллельного методов.

2. При внимательном рассмотрении можно обратить внимание, что попытки повышения качества параллельных вычислений по одному из показателей (ускорению или эффективности) могут привести к ухудшению ситуации по другому показателю, ибо показатели качества параллельных вычислений являются часто противоречивыми. Так, например, повышение ускорения обычно может быть обеспечено за счет увеличения числа процессоров, что приводит, как правило, к падению эффективности. И наоборот, повышение эффективности достигается во многих случаях при

уменьшении числа процессоров (в предельном случае идеальная эффективность $E_p(n) = 1$ легко обеспечивается при использовании одного процессора). Как результат, разработка методов параллельных вычислений часто предполагает выбор некоторого компромиссного варианта с учетом желаемых показателей ускорения и эффективности.

Показатели качества параллельных вычислений являются противоречивыми: попытки повышения качества параллельных вычислений по одному из показателей может привести к ухудшению ситуации по другому показателю.

При выборе надлежащего параллельного способа решения задачи может оказаться полезной оценка стоимости вычислений, определяемой как произведение времени параллельного решения задачи и числа используемых процессоров $C_p = pT_p$.

Стоимостно-оптимальный (cost-optimal) параллельный алгоритм - метод, стоимость которого является пропорциональной времени выполнения наилучшего последовательного алгоритма.

Заключение

Можно выделить следующий ряд общих проблем, возникающих при использовании параллельных вычислительных систем:

высокая стоимость параллельных систем - в соответствии с подтверждаемым на практике законом Гроша, производительность компьютера возрастает пропорционально квадрату его стоимости и, как результат, гораздо выгоднее получить требуемую вычислительную мощность приобретением одного производительного процессора, чем использование нескольких менее быстродействующих процессоров;

потери производительности для организации параллелизма - согласно гипотезе Минского, ускорение, достигаемое при использовании параллельной системы, пропорционально двоичному логарифму от числа процессоров;

постоянное совершенствование последовательных компьютеров. В соответствии с законом Мура мощность последовательных процессоров возрастает практически в 2 раза каждые 18-24 месяцев и, как результат, необходимая производительность может быть достигнута и на "обычных" последовательных компьютерах;

существование последовательных вычислений - в соответствии с законом Амдаля, ускорение процесса вычислений при использовании нескольких процессоров ограничивается, т.е., например, при наличии всего 10% последовательных команд в выполняемых вычислениях эффект использования параллелизма не может превышать 10-кратного ускорения обработки данных;

зависимость эффективности параллелизма от учета характерных свойств параллельных систем - в отличие от единственности классической схемы фон Неймана последовательных ЭВМ параллельные системы отличаются существенным разнообразием архитектурных принципов построения, и максимальный эффект от использования параллелизма может

быть получен только при полном использовании всех особенностей аппаратуры;

существующее программное обеспечение ориентировано в основном на последовательные ЭВМ - данное возражение состоит в том, что для большого количества задач уже имеется подготовленное программное обеспечение и все эти программы ориентированы главным образом на последовательные ЭВМ - как результат, переработка такого количества программ для параллельных систем вряд ли окажется возможным.

Используемые источники

1. https://ru.wikipedia.org/wiki/Параллельный_алгоритм
   Википедия: Параллельный алгоритм

2. https://tproger.ru/articles/computational-complexity-explained
   Tproger: Оценка сложности алгоритмов, или Что такое O(log n)

3. https://habr.com/ru/articles/104219/ Хабр: Оценка сложности алгоритмов

4. https://it.kgsu.ru/ParalAlg/palg012.html
   Информатика и программирование: Шаг за шагом.