УДК МИРОШНИЧЕНКО И. Д., РЕЗЕНОВА А. П., РУДНЕВА О. И.

АНАЛИЗ ЭФФЕКТИВНОСТИ СОВРЕМЕННЫХ АЛГОРИТМОВ СОРТИРОВОК БОЛЬШИХ ДАННЫХ (С УЧЕТОМ ПАРАЛЛЕЛЬНОЙ ОБРАБОТКИ РАСПРЕДЕЛЕННЫХ ДАННЫХ)

Аннотация

Статья посвящена сравнительному анализу функционирования и реализации алгоритмов сортировок как последовательного программирования, так и параллельного, в том числе применимых к обработке больших данных.

ВВЕДЕНИЕ

К 2020 году программирование глубоко внедрилось в различные сферы нашей жизни. Сотни тысяч привычных вещей не существовали бы без программирования или были бы гораздо менее удобными в использовании. В большой степени это касается и задач, решаемых Вооруженными силами Российской Федерации. Как следует из исследований Zecurion[1] Analytics, специализированные подразделения по кибербезопасности официально используют несколько десятков стран, а неофициально — более сотни. Известно, что Россия входит в топ-5 стран по численности и финансированию кибервойск.

По словам руководителя аналитического центра Zecurion Владимира Ульянова[2] "Зависимость различных устройств и оборудования от интернета будет только расти. В результате будет увеличиваться уязвимость отдельных пользователей, их гаджетов, машин, а также систем и инфраструктуры стран".

Эти исследования подтверждают важность подготовки специалистов в области программирования в различных родах войск Российской Федерации, в том числе и в ВМФ РФ.

Само по себе программирование востребовано практически во всех сферах нашей жизни, на самом же деле программирование представляет собой далеко не такой простой предмет, как многим кажется на первый взгляд. Его сложность проявляется в том, что специалисту или обучающемуся для достижения результатов требуется обеспечивать серьезную умственную отдачу, что тренирует мышление, а также способствует развитию логики.

Знание базовых алгоритмов, каковыми являются алгоритмы поиска и сортировок, необходимы, в первую очередь, для умения решать типовые задачи при разработке программного обеспечения, хотя полезны также и для саморазвития.

В статье рассмотрены особенности функционирования наиболее важных последовательных алгоритмов сортировок, сравнительная их эффективность на последовательностях различной степени упорядоченности, а также особенности представления параллельных алгоритмов сортировок в распределенных средах.

Общее представление об алгоритмах сортировки.

Сортировка данных — одна из часто встречающихся и важных задач в программировании, влияющая на существенные характеристики выполнения программ: скорость, эффективность, устойчивость. Существует огромное множество алгоритмов сортировок, предназначенных для специфических задач: одни алгоритмы хороши для больших объемов данных, другие — для почти отсортированных (почти упорядоченных) данных, третьи — для случаев, когда использование памяти не ограничено или, наоборот, при существенной ограниченности памяти; а для современных вычислительных средств также важен способ обработки данных: последовательный или параллельный.

Так, например, для решения одних задач может не быть ограничений на время или на количество потребляемой памяти при сортировке, в других ситуациях необходимо, чтобы объекты сортировались с учетом исходной последовательности (устойчивость сортировки). Иногда требуется, чтобы сортировка была максимально быстрой, при этом требуется учитывать особенности входных данных. В каждом конкретном случае только разработчик-программист может и должен определить, какой алгоритм выбрать. Поэтому и нужны знания о разных алгоритмах, их слабых и сильных сторонах, что позволит правильно выбрать необходимый алгоритм для решения конкретной задачи.

Понимание функционировании, эффективности алгоритмов, в частности, сортировок, даже пузырьковой сортировки позволяет понять типичные решения проблем, которые впоследствии смогут применить в разработке других задач.

Определение *алгоритма сортировки*[3] *(Википедия)* звучит следующим образом: “это алгоритм для упорядочивания элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма”.

Более строго, математически, сортировка обычно понимается как задача размещения элементов неупорядоченного набора значений $S\{a_1, a_2, a_3, \ldots, a_n\}$ в порядке монотонного возрастания или убывания

$$S\sim S'\{(a_{1'}, a_{2'}, a_{3'}, \ldots, a_{n'}): a_{1'}<+a_{2'}<= a_{3'}<=\ldots<=a_{n'}\}$$

и является одной из типовых проблем обработки данных.

Понятия сложности и эффективности алгоритма.

Существуют инструменты — программы - *профайлеры* (*profilers*), измеряющие, насколько быстро работает код. Эти программы определяют время выполнения в миллисекундах, выявляя узкие места и оптимизируя их. Это полезный инструмент, но он не имеет отношения к сложности алгоритмов. Так, например, может оказаться, что плохой алгоритм, написанный на низкоуровневом языке (ассемблере) выполняется намного быстрее, чем хороший алгоритм, написанный на языке программирования высокого уровня (модных сегодня Python или Ruby).

---

[1] Zecurion основана в 2001 году и занимается разработкой программного обеспечения для защиты данных от утечек (DLP). Работает в сфере информационной безопасности на рынках России, СНГ, Европы, США, Японии и Турции.

[2] Газета «Коммерсант» от 30 января 2021 г. https://www.kommersant.ru/doc/3187320

[3] https://ru.wikipedia.org/wiki/Алгоритм_сортировки

Сложность алгоритма — это способ формально измерить, насколько быстро программа или алгоритм работают. Процедура выяснения сложности алгоритма основывается *на сравнении двух алгоритмов* на идеальном уровне, когда игнорируются как выбор собственно языка программирования, на котором написана программа, так и реализация компилятора с языка программирования, «железо», на котором выполняется программа (набор команд конкретного процессора. Рассматривается исключительно *вычисление*, без других часто выполняемых компьютером вещей: сетевых задач или пользовательского ввода-вывода.

Анализ сложности позволяет узнать, насколько быстра некоторая программа, когда она выполняет вычисления, и объяснить, как будет вести себя алгоритм при возрастании входного потока данных. Например, если алгоритм выполняется одну секунду на входной последовательности из 1000 элементов, то крайне важно знать, как он себя поведёт, если удвоить длину входной последовательности: также быстро, в полтора раза быстрее или в четыре раза медленнее? Анализ сложности также дает понимание того, как долго будет выполняться конкретный код на самом длинном тесте при проверке его правильности. Так что, если определить общее поведение программы на небольшом объёме входных данных, то можно получить хорошее представление о том, как она будет вести себя при больших потоках данных.

При анализе сложности обязательно рассматривается анализ наиболее неблагоприятного случая («наихудшего варианта»), то есть рассмотрение максимально неудачного варианта.

Кроме того, при анализе сложности важность имеет только тот факт, который показывает, *что происходит с вычислениями* (функцией подсчёта команд анализируемого фрагмента) *при значительном возрастании* ***n*** (длины последовательности входных данных). Это коррелирует с рассмотрением «наихудшего варианта»: интересно поведение алгоритма, находящегося в «плохих условиях». Именно это по-настоящему полезно при сравнении алгоритмов: если один из них опережает другой при большом входном потоке данных, то велика вероятность, что он останется быстрее и на маленьких потоках. Поэтому при подсчете команд анализируемого фрагмента кода отбрасывают те элементы функции подсчета команд кода, которые при росте ***n*** возрастают медленно, и оставляют только те, которые растут значительно. Такое исследование кода представляет исследование *асимптотического поведения* кода. Если описать выше сказанное математическим языком, это означает, что при исследовании асимптотического поведения кода интересует предел функции *f* (подсчета команд кода) при *n* (количестве элементов входной последовательности), стремящемся к бесконечности. То есть асимптотическое поведение, например, для функции $f(n) = 2n + 8$, будет описываться асимптотической функцией $f(n) = n$.

Итак, с увеличения размера входных данных обычно растет сложность алгоритма, а, значит, и время его выполнения. Сложность алгоритма принято называть *порядком роста*, который чаще всего представляется в виде записи ***O***-большое (от функции длины входной последовательности). Запись $O(f(n))$ – это математическое обозначение (характеристика сложности), которая дает представление о ресурсах, которые требуются алгоритму для выполнения вычислений. Рассматривают следующие градации сложности алгоритмов.

Константная сложность — $O(1)$. Порядок роста $O(1)$ означает, что вычислительная сложность алгоритма не зависит от размера входных данных. Единица в аргументе формулы означает, что время выполнения фрагмента кода не зависит от длины входных данных. Это не значит, что алгоритм выполняется за одну операцию. Например, код может состоять из нескольких операторов присваивания, в которых нет зависимости от входной последовательности, тогда сложность фрагмента кода все равно есть $O(1)$.

Линейная сложность — $O(n)$. *Порядок* роста $O(n)$ означает, что сложность алгоритма линейно растет с увеличением *размера входной последовательности данных.* Если алгоритм обрабатывает один элемент десять миллисекунд, то можно ожидать, что тысяча элементов будет обрабатываться десять секунд. Такие алгоритмы обычно представляют цикл с выполнением простого оператора по каждому элементу входной последовательности. Простейший пример фрагмента кода алгоритма вычисления сложности $O(n)$ на алгоритмическом языке C++ (сложность которого равна $O(n)$).

```
long Summa(int Mas[])                    //O(1)+ O(n) = O(n)
    {long sum = 0;                       //O(1)
     int i, n;
     n = Mas.Length;
     for (i = 0; i < n; i++)             //O(n)
            {sum += i; }
     return sum;
   }
```

Пример вычисления функции сложности линейного алгоритма $f(n) = 5n + 12$ даст результат $f(n) = n$.

Логарифмическая сложность – $O(\log_2 n)$ или $O(\log n)$. Порядок роста $O(\log n)$ означает, что время выполнения алгоритма растет логарифмически с увеличением размера входной последовательности данных (в анализе алгоритмов по умолчанию используется логарифм по основанию 2). Большинство алгоритмов, использующих принцип дихотомии или «деления пополам», имеют логарифмическую сложность. Например, метод поиска по бинарному дереву (*binary search tree*) имеет порядок роста $O(\log n)$.

Линеарифметическая сложность — $O(n \cdot \log n)$. Порядок роста *л*инеарифметического (или линейно-логарифмического) алгоритма означает, что время выполнения алгоритма растет в соответствии с порядком сложности $O(n \cdot \log n)$. Некоторые алгоритмы такие, как, например, сортировка слиянием или быстрая сортировка, имеют именно такую сложность, они относятся к алгоритмам, построенным по принципу «разделяй и властвуй».

Квадратичная сложность — $O(n^2)$. Порядок роста $O(n^2)$ означает, что время работы алгоритма с указанным порядком роста зависит от квадрата размера входной последовательности (например, двойной цикл обработки каждого элемента входной последовательности). Хотя алгоритмы с такой сложностью встречаются весьма часто, — это повод пересмотреть используемые алгоритмы или структуры данных. К примеру, если массив из 1000 элементов потребует 1 000 000 операций, массив из 1 000 000 элементов потребует 1 000 000 000 000 операций. Если одна операция требует миллисекунду для выполнения, то *последовательный* квадратичный алгоритм будет обрабатывать 1 000 000 элементов 1 000 000 000 секунд или 1000000000/(3600x24x365) = 32 года. Пример вычисления функции квадратичной сложности алгоритма $f(n) = n^2 + 3n + 112$ даст результат $f(n) = n^2 + n + 1 = n^2$.

Экспоненциальная сложность — ***O(c^n, c>1).*** Порядок роста ***O(c^n, c>1)*** означает, что время работы алгоритма увеличивается очень быстро, с экспоненциальной скоростью. Это алгоритмы, связанные с перебором элементов некоторого множества, они относятся к классу NP-полных, иначе говоря, определяется, не являются ли два логических утверждения эквивалентными с помощью полного перебора. Например, алгоритм нахождения решения задачи коммивояжёра с помощью динамического программирования.

Эффективность алгоритма

Приведенные примеры о сложности алгоритмов выше линейной, показывают, что при исследовании сложности алгоритмов нужно учитывать

— количество операций, требуемых для завершения работы, то есть *вычислительную* сложность, и
— объем ресурсов, в частности, памяти, который необходим алгоритму, то есть *пространственную* сложность.

Теперь, анализируя сложность алгоритма, можно оценить его применимость в реальных условиях: алгоритм, который выполняется в десять раз быстрее, но использует в десять раз больше памяти, чем, грубо говоря, входная последовательность, может вполне подходить для вычислительного комплекса с большим объемом памяти, но для систем, где объём памяти ограничен, такой алгоритм использовать нельзя [2].

Эффективность алгоритма — это решение задачи за приемлемое для разработчика время. То есть, это свойство алгоритма, которое связано с вычислительными ресурсами, используемыми алгоритмом. Поэтому алгоритм должен быть проанализирован для того, чтобы определить необходимые алгоритму ресурсы. Эффективность алгоритма можно рассматривать как аналог производственной производительности повторяющихся или непрерывных процессов.

Анализ эффективности алгоритма включает изучение

— ситуаций, в которых алгоритм демонстрирует свои наилучшие свойства (*наилучший случай*),
— ситуаций, когда его эффективность минимальна (*наихудший случай*),
— а также оценку его средней производительности (*средний случай,* анализ которого обычно самый сложный).

Для достижения максимальной эффективности требуется уменьшить использование ресурсов. Однако различные ресурсы (такие как время и память) нельзя сравнивать напрямую, так что какой из двух алгоритмов считать более эффективным часто зависит от того, какой фактор более важен, например, требование высокой скорости, минимального использования памяти, сбалансированности обоих параметров или другой меры эффективности.

Приведем схему, представляющую некоторые классические типы алгоритмов сортировок (рис. 1) последовательной обработки, характеристики которых (эффективность, сложность) упоминаются в статье.

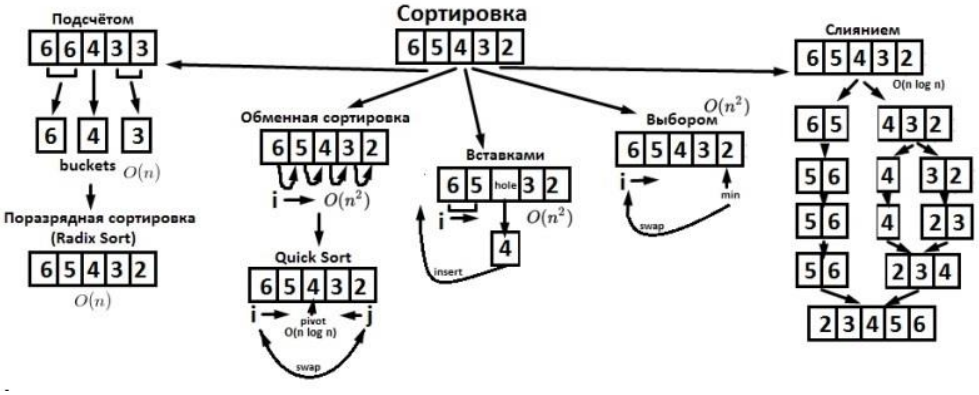


Рис.1 Типы последовательных сортировок

Алгоритмы сортировки используют разные методы обработки и обладают различными характеристиками. Представим свойства некоторых из них в следующей таблице.

Таблица 1. Свойства некоторых классических сортировок

Сортировка	Время работы, сложность	Метод	Область использования
Вставкой	***O(n^2)***	Вставка	Очень малые массивы
Выбором	***O(n^2)***	Выбор	Очень малые массивы
Пузырьковая	***O(n^2)***	Двусторонние прохождения, ограничения рассматриваемых пределов	Очень малые и частично сортированные массивы
Пирамидальная	***O(n · log n)***	Кучи, хранение полных деревьев в массиве	Крупные массивы с неизвестным распределением
Быстрая	***O(n · log n)*** ***O(n^2)*** - худший	«Разделяй и властвуй», перемещение элементов в позицию, рандомизация во избежание худшего случая	Крупные массивы без большого количества дубликатов, параллельная сортировка
Слиянием	***O(n · log n)***	«Разделяй и властвуй», объединение, внешняя сортировка	Крупные массивы с неизвестным распределением, большие объемы данных, параллельная сортировка
Подсчетом	***O(n + m)***	Счет	Крупные массивы с достаточно единообразным распределением значений

Рассмотрим сначала последовательные алгоритмы сортировок, кратко описывая метод обработки элементов последовательности; особенности некоторых из них, указывая асимптотическую сложность и эффективность функционирования.

1. *Пузырьковая сортировка (Bubble_sort)*[4]. Алгоритм считается учебным и практически не применяется вне учебной литературы, вместо него на практике применяются более эффективные алгоритмы сортировки.

[4] https://ru.wikipedia.org/wiki/Сортировка_пузырьком

Однако метод сортировки обменами лежит в основе некоторых более совершенных алгоритмов, таких как шейкерная сортировка, пирамидальная сортировка и быстрая сортировка.

Метод обработки состоит в попарном сравнении элементов последовательности при движении слева направо. Если текущий элемент больше следующего, меняем их местами. После первого прохода (первой итерации) самый большой элемент окажется на последнем месте последовательности (его больше сравнивать и перемещать не нужно). Таким образом, за одну итерацию в худшем случае будет сделан *n-1* обмен. После следующего прохода (второй итерации) на предпоследнем месте окажется следующий по величине элемент последовательности (он может быть меньше либо равен отсортированному на первой итерации последнему элементу). Очевидно, не более, чем после *n-1* итераций последовательность будет отсортирована.

Сортировка пузырьком работает медленно на тестах, в которых элементы с меньшими значениями стоят в конце исходной последовательности (их еще называют «черепахами»). На каждом шаге алгоритма такой элемент будет сдвигаться всего на одну позицию влево. Таким образом, *асимптотическая сложность* в худшем и среднем случае равна $O(n^2)$, в лучшем случае — $O(n)$ (когда входная последовательность изначально отсортирована). Сортировка не требует дополнительной памяти для размещения последовательности или ее части.

2. *Шейкерная сортировка* (сортировка *перемешиванием* или *коктейльная* сортировка *Shaker_sort*)[5] — это видоизмененная сортировка пузырьком. Выполним её оптимизацию: будем сравнивать попарно элементы последовательности, двигаясь не только слева направо, но и справа налево. Для этого объявим два указателя (для левой части последовательности - ***beg)*** и для правой части - ***end***), которые будут отмечать ещё не отсортированную часть последовательности по направлению обработки. При достижении правого указателя ***end*** вычитаем из значения указателя единицу и движемся справа налево, аналогично, при достижении левого указателя ***beg*** прибавляем единицу к значению указателя и двигаемся слева направо. При совпадении значений указателей выполнение сортировки заканчивается.

Асимптотическая сложность у алгоритма шейкерной сортировки совпадает с асимптотической сложностью пузырьковой сортировки (в худшем и среднем случае равна $O(n^2)$, в лучшем случае — $O(n)$), но реальное время (в худшем и среднем случае лучше) меньше, потому что путь передвижения меньших значений влево несколько уменьшается. Сортировка не требует дополнительной памяти для размещения последовательности или ее части.

3. *Сортировка расческой*[6] (*Comb_sort*) — также модификация пузырьковой сортировки. Основная идея (как и у шейкерной сортировки) — устранить «черепах», или маленькие значения в конце последовательности, которые крайне замедляют сортировку пузырьком («кролики», большие значения в начале последовательности, не представляют проблемы для сортировки пузырьком).

Для этого первоначально берется достаточно большое расстояние между сравниваемыми элементами и, по мере упорядочивания последовательности, это расстояние сужается до минимального[7]. Сначала выбирают расстояние между сравниваемыми элементами максимальным, то есть равным размеру последовательности минус один. Затем, пройдя последовательность с этим шагом, необходимо поделить шаг на фактор уменьшения (оптимальное значение которого равно примерно 1,247) и пройти последовательность снова с новым шагом. Продолжать процесс до тех пор, пока разность индексов сравниваемых элементов не достигнет единицы. На последнем проходе сравниваются соседние элементы как в сортировке пузырьком.

Асимптотическая сложность у алгоритма сортировки расчёской в худшем случае совпадает с асимптотической сложностью пузырьковой сортировки $O(n^2)$, в лучшем случае равна $O(n \cdot log\ n)$, среднем случае – не хуже $O(n^2/2)$ и реальное время (в худшем и среднем случае лучше) меньше, как и в шейкерной сортировке. Сортировка не требует дополнительной памяти для последовательности.

4. *Сортировка вставками (Insertion_sort).* На каждом шаге выбирается очередной элемент из последовательности и вставляется в предшествующую отсортированную часть последовательности так, чтобы она снова была отсортирована[8]. Асимптотическая сложность алгоритма сортировки в худшем и среднем случае совпадает с асимптотической сложностью пузырьковой сортировки $O(n^2)$, в лучшем случае равна (когда последовательность изначально отсортирована) $O(n)$. Сортировка в описанном варианте не требует дополнительной памяти.

5. Сортировка *Шелла* (*Shell_sort*)[9] использует ту же идею, что и сортировка расческой, только в применении к сортировке вставками. Сначала выбирается некоторое расстояние d для входной последовательности (обычно, это p, где $p=[n/2]$ близкие к степени двойки — $p=2^k$) и сравниваются и обмениваются между собой значения, стоящие один от другого на этом расстоянии d. Процедура повторяется на каждом проходе для новых значений $d_i=d_i-1/2$, а завершается сортировка Шелла упорядочиванием элементов при $d = 1$ (то есть обычной сортировкой вставками). Эффективность сортировки Шелла, как и в сортировке расческой обеспечивается тем, что элементы «быстрее» встают на свои места.

Среднее время работы (и сложность) алгоритма зависит от длин промежутков d на каждом шаге алгоритма, которые выбираются для сравнения элементов исходной последовательности длины n. Асимптотическая сложность алгоритма сортировки Шелла в худшем случае совпадает с асимптотической сложностью пузырьковой сортировки $O(n^2)$, в лучшем случае равна $O(n \cdot log^2 n)$, в среднем случае зависит от выбранного расстояния на длине последовательности.

Кроме подхода к выбору d, предложенного Шеллом, исследовались:

- последовательность Хиббарда для $d=2^i-1<=n$, тогда сложность среднего случая равна $O(n^{1,5})$;
- последовательность Седжвика для $d=9x2^i-9x2^{i/2}+1$, если i-четное и для $d=8x2^i-6x2^{(i+1)/2}+1$, если i – нечетное, тогда сложность среднего случая получилась равной $O(n^{7/6})$, худшего — $O(n^{4/3})$;

[5] https://ru.wikipedia.org/wiki/Сортировка_перемешиванием

[6] Алгоритм изначально спроектирован Влодзимежом Добосевичем в 1980 г., позднее был переоткрыт и популяризован в статье Стивена Лэйси и Ричарда Бокса в журнале Byte Magazine в апреле 1991 г.

[7] Сайт https://ru.wikipedia.org/wiki/расческой наглядно представляет работу сортировки расческой.

[8] https://ru.wikipedia.org/wiki/Сортировка_вставками

[9] https://ru.wikipedia.org/wiki/Сортировка_Шелла

— последовательность Пратта для $d=2^i x 3^j <= n/2$, тогда сложность среднего случая получилась равной $O(n \cdot log^2 n)$;

— последовательности вида $s_i = a * s_i - 1 + k * s_i - 1$ для небольшого количества элементов с коэффициентами $a = 3, k = 1/3$; $a = 4, k = ¼$ и $a = 4, k = -1/5$ также дали наилучшие результаты.

Сортировка для этих вариантов не требует дополнительной памяти для размещения последовательности или ее части.

6. *Сортировка деревом* (сортировка двоичным деревом, древесная сортировка, сортировка с помощью бинарного дерева *Tree_sort*). Алгоритм выполняется в два этапа: сначала строится двоичное дерево поиска по ключам-элементам последовательности, затем выполняется последующая сборка результирующего последовательности путём обхода узлов построенного дерева в неубывающем (невозрастающем) порядке следования ключей.

Эта сортировка оптимальна при получении данных путём непосредственного чтения из потока (например, файла или консоли). Если использовать сбалансированное дерево, например, красно-черное, асимптотическая сложность алгоритма будет равна $O(n \cdot log_2 n)$ в худшем, среднем и лучшем случае.

Однако, процедура добавления объекта в бинарное дерево имеет *среднюю* алгоритмическую сложность порядка $O(log\ n)$. Соответственно, для добавления n объектов сложность алгоритма будет составлять $O(n \cdot log\ n)$, что характеризует сортировку с помощью двоичного дерева как «быструю сортировку». Однако, сложность добавления объекта в разбалансированное дерево может достигать и $O(n)$, тогда общая сложность алгоритма бинарного дерева порядка будет равна $O(n^2)$.

При построении древовидной структуры для последовательности длины n требуется памяти не менее, чем $4n$ ячеек дополнительной памяти, так как каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист, однако, существуют способы уменьшить требуемую дополнительную память.

7. *Гномья сортировка* (*Gnome_sort*)[10]. Алгоритм сортировки похож на сортировку вставками, но в отличие от последней перед вставкой на нужное место происходит серия обменов, как в сортировке пузырьком. Название происходит от предполагаемого поведения садовых гномов при сортировке линии садовых горшков[11].

Фиксируем указатель на текущий элемент. Алгоритм находит первое место, где два соседних элемента стоят в неправильном порядке и меняет их местами, то есть, если указатель больше предыдущего или он первый — смещаем указатель на позицию вправо, иначе меняем текущий и предыдущий элементы местами и смещаемся влево Обмен порождает новую пару, стоящую в неправильном порядке, только до или после переставленных элементов. Он не допускает, чтобы элементы после текущей позиции были отсортированы, таким образом, нужно только проверить позицию до переставленных элементов.

Асимптотическая сложность алгоритма сортировки в худшем и среднем случае совпадает с асимптотической сложностью сортировки вставками $O(n^2)$, в лучшем случае (последовательность изначально отсортирована) равна $O(n)$. Сортировка в описанном варианте не требует дополнительной памяти для размещения последовательности или ее части.

Эту сортировку можно несколько улучшить. Если запоминать место, в котором встретилось неотсортированное значение элемента и сделать несколько корректирующих итераций назад, то после получения отсортированности в левой части последовательности, можно прыгнуть сразу туда, где прервались, и следовать по последовательности далее.

8. *Сортировка выбором (Selectio_sort)*. Рассмотрим следующий вариант реализации. Сначала найти минимальный элемент в последовательности (запомнив место, где он находился) и поместить его самым первым в последовательности, а всю подпоследовательность, стоявшую до позиции, в которой находился минимальный элемент последовательности, сдвинуть на один элемент вправо. Далее снова найти минимальный элемент, но уже начиная со второй позиции (так как на первом месте стоит уже элемент с самым маленьким значением) и поместить его на второе место, а подпоследовательность со второй позиции до позиции второго минимального элемента последовательности снова сдвинуть на одну позицию вправо. Выполнять до последнего элемента позиции.

Асимптотическая сложность алгоритма — $O(n^2)$ в лучшем, среднем и худшем случае. Отметим, что эту сортировку можно реализовать двумя способами – сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ немного быстрее. Сортировка не требует дополнительной памяти для размещения последовательности или ее части.

9. *Пирамидальная сортировка (Heap_sort)*[12] — это развитие идеи сортировки выбором. Для этого используется структура данных типа «куча» (или «пирамида», откуда и название алгоритма). Сортировка пирамидой использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия:

— каждый лист имеет глубину либо d, либо $d - 1$, где d — максимальная глубина дерева.

— значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

Такая структура позволяет получать минимум за $O(1)$, добавляя элементы и извлекая минимум за $O(log^2 n)$. Таким образом, асимптотическая сложность пирамидальной сортировки равна $O(n \cdot log^2 n)$ в худшем, среднем и лучшем случае. Количество применяемой служебной памяти не зависит от размера массива (то есть практически не требуется).

10. *Быстрая сортировка, сортировка Хоара*[13] *(Quicksort)* — один из самых быстрых известных универсальных алгоритмов сортировки последовательностей: в среднем $O(n \cdot log\ n)$ обменов при упорядочении n элементов. Из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками. Идея состоит в том, что выбирается некоторый опорный элемент, после этого перемещаются все элементы, меньшие опорного, налево, а большие – направо. Затем рекурсивно выполняется та же операция с каждой из частей. В итоге получаем отсортированную последовательность, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотическая сложность алгоритма $O(n \cdot log^2 n)$ в среднем и лучшем случае. Наихудшая оценка достигается при неудачном выборе опорного элемента и равна $O(n^2)$.

---

[10] https://ru.wikipedia.org/wiki/Гномья_сортировка

[11] «Гномья сортировка основана на технике, используемой обычным голландским садовым гномом (нидерл. *tuinkabouter*). Это метод, которым садовый гном сортирует линию цветочных горшков. По существу, он смотрит на текущий и предыдущий садовые горшки: если они в правильном порядке, он шагает на один горшок вперёд, иначе он меняет их местами и шагает на один горшок назад. Граничные условия: если нет предыдущего горшка, он шагает вперёд; если нет следующего горшка, он закончил.» Дик Грун.

[12] https://ru.wikipedia.org/wiki/Пирамидальная_сортировка

[13] https://ru.wikipedia.org/wiki/Быстрая_сортировка разработана английским информатиком Тони Хоаром во время его работы в МГУ в 1960 году

Помимо чистой быстрой сортировки, предлагается быстрая сортировка, переходящая при малом количестве элементов в сортировку вставками.

К сожалению, ввиду ограниченности объема статьи демонстрировать выполнение алгоритма быстрой сортировки возможности нет.

11. *Сортировка слиянием (Merge_sort).* Алгоритм сортировки, который упорядочивает последовательности (или другие структуры данных, к элементам которых доступ можно получать только последовательно, например, это могут быть потоки) в определённом порядке. Сортировка основана на парадигме «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

В частности, разделим последовательность пополам, рекурсивно отсортируем части, после этого выполним процедуру слияния: установим два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Слияние работает за ***O(n)***, уровней всего ***log n***, поэтому асимптотическая сложность алгоритма равна ***O(n·log n)*** в лучшем, среднем и худшем случае. Эффективно заранее создать временный массив размером с последовательность и передать его в качестве аргумента функции. Эта сортировка рекурсивна, как и быстрая, а потому возможен переход на квадратичную при небольшом числе элементов.

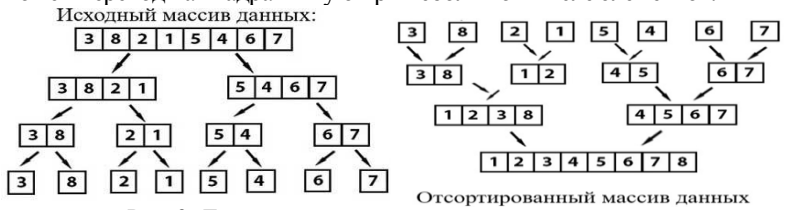


Рис. 2. Два этапа выполнения сортировки слиянием
(слева — рекурсивное разбиение на 2 части, справа — слияние в правильном порядке)

Следует отметить, что в отличие от линейных алгоритмов сортировки, сортировка слиянием делит и склеивает элементы последовательности вне зависимости от того, был он отсортирован изначально или нет. Поэтому, несмотря на то, что в худшем случае алгоритм отработает быстрее, чем линейный, в лучшем случае, производительность алгоритма будет ниже, чем у линейного. Поэтому сортировка слиянием — не самое лучшее решение, когда необходимо отсортировать частично упорядоченный массив.

Недостатки метода заключаются в том, что требуется дополнительная память по объему, равная объему сортируемого файла. Поэтому для *больших файлов* проблематично организовать сортировку слиянием в оперативной памяти. В случаях же, когда гарантированное время сортировки важно и размещение в оперативной памяти возможно, то следует предпочесть метод сортировки слиянием.

12. *Сортировка подсчетом (Counting_sort)*[14]. Алгоритм сортировки, в котором используется диапазон чисел сортируемой последовательности (списка) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000.

Существует несколько модификаций сортировки подсчётом, часто рассматривают три линейных и одна квадратичная, которая использует другой подход, но имеет то же название.

Идея алгоритма состоит в предварительном подсчете количества элементов с различными ключами в исходной последовательности и разделении результирующей последовательности на части соответствующей длины (блоки). Затем при повторном проходе исходной последовательности каждый его элемент копируется в специально отведенный его ключу блок, в первую свободную ячейку. Таким образом после завершения алгоритма в результирующей последовательности содержится исходная последовательность в отсортированном виде, так как блоки расположены по возрастанию соответствующих ключей.

Рассмотрим пример. Пусть имеем исходную последовательность ***0 4 4 3 0 1 2 2***

Рис. 3. Исходная последовательность и вспомогательный массив из 5 элементов (0-4)

Пройдя по последовательности, выясняем, что различных элементов 5 (***0 1 2 3 4***), подготовили вспомогательный массив из пяти элементов, поместив в его элементы последовательно числа от 0 до 4.

Теперь пройдем снова с начала массива и поместим в ячейки вспомогательного массива соответствующее индексу ячейки количество чисел в последовательности (рис. 4).

Рис. 4. Подсчет количества одинаковых значений

То есть, в ячейке с индексом 0 находится число 2 (в исходной последовательности есть два числа 0), в ячейке с индексом 1 находится число 1 (в исходной последовательности одно число 1), в ячейке с индексом 2 находится число 2 (в исходной последовательности есть два числа 2), аналогично в ячейке с индексом 3 имеем число 1, а с индексом 4 – число 2.

Теперь будем использовать вспомогательный массив немного по-другому (рис. 5).

[14] https://neerc.ifmo.ru/wiki/index.php?title=Сортировка_подсчетом

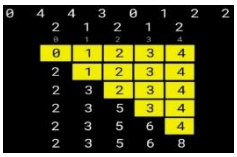


Рис. 5. Заполнение вспомогательного массива суммами

В ячейке с индексом 0 оставим то значение, которое в нем находится, то есть число 2, в ячейку с индексом 1 поместим сумму значений чисел, находящихся в 0-й ячейке и 1-й ячейке, то есть 2+1=3, в ячейку с индексом 2 поместим сумму значений, находящихся в ячейках с индексами 0,1 и 2, то есть 2+1+2=5. Аналогичными действиями получим в ячейке с индексом 3 получим 2+1+2+1=6, а в ячейке с индексом 4: 2+1+2+1+2=8 (рис. 5).

На практике для тестового массива с 1 000 000 элементов и диапазоном значений от 0 до 1000, *быстрая сортировка* заняла 4,29 секунд, а сортировка подсчетом — всего 0,03 секунды. Следует отметить, что для *быстрой сортировки* выбранный пример оказался не самым лучшим, поскольку среди значений было множество равных друг другу (приблизительно 1000), и данный алгоритм справляется с ними плохо.

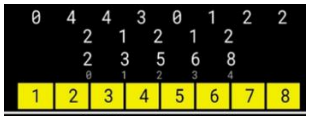
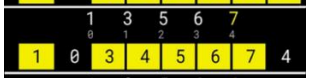
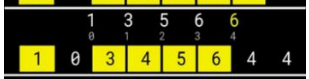
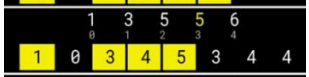
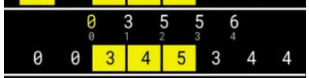
	Исходная последовательность Число одинаковых значений Последовательность сумм (позиция) Последовательность чисел Результирующая последовательность
	Берем первый элемент исходной последовательности – 0. В строке сумм с 0-м индексом стоит число 2. Вычитаем из 2-1, получаем индекс 1 для результирующей последовательности: на место с индексом 1 ставим число 0
	Берем второй элемент исходной последовательности 4. В последовательности чисел находим 4 и находим для него значение суммы 8.
	Вычитаем: 8-1=7. На 7 место результирующей последовательности ставим число 4. В исходной последовательности чисел находимт3-й элемент 4 и находим для него значение суммы 7. Вычитаем: 7-1=6.
	В результирующей последовательности на 6 место ставим число 4. Берем 4-й элемент исходной последовательности 3. Находим для него индекс размещения в массиве сумм 6. Вычитаем 6-1=5.
	На 5-е место результирующей последовательности ставим число 3. Берем 5-й элемент исходной последовательности 0. В последовательности чисел находим 0 и находим для него значение суммы 1.
	Вычитаем: 1-1=0. В результирующей последовательности на 0-е место ставим число 0. Берем 6-й элемент исходной последовательности 1. Находим для него индекс в массиве сумм – это 3. Вычитаем:3-1=2
	Помещаем 1 во вторую позицию результирующего массива. Берем 7-й элемент исходной последовательности 2. Находим для него индекс в массиве сумм 5. Вычитаем: 5-1=4.
	Помещаем 2 в четвертую позицию результирующей последовательности. Берем 8-й элемент исходной последовательности 2. Находим для него индекс в массиве сумм 4. Вычитаем: 4-1=3.
	Помещаем 2 в третью позицию результирующего массива.
	Получили результирующую последовательность

Рис. 12. Демонстрация функционирования сортировки подсчетом

В предложенном алгоритме первые два цикла работают за ***O(k)*** и ***O(n)***, соответственно; двойной цикл — за ***O(n+k)***. Алгоритм имеет линейную сложность и временную трудоёмкость ***O(n+k)***. Используемая дополнительная память требуется в размере не менее ***n+k*** ячеек. Алгоритм работает за линейное время, но является псевдополиномиальным.

13. *Блочная сортировка (корзинная* и *карманная сортировка)* (*Bucket_sort*)[15]. Алгоритм сортировки, в котором сортируемые элементы распределяются между конечным числом отдельных блоков (карманов, корзин) так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше (или меньше), чем в предыдущем. Каждый блок затем сортируется отдельно, либо рекурсивно тем же методом, либо другим. Затем элементы помещаются обратно в массив. Этот тип сортировки может обладать линейным временем исполнения.

Заметим, что алгоритм требует знаний о природе сортируемых данных, выходящих за рамки функций "сравнить" и "поменять местами", достаточных для сортировки слиянием, сортировки пирамидой, быстрой сортировки, сортировки Шелла, сортировки вставкой.

Если количество блоков равно двум, то данный алгоритм превращается в разновидность быстрой сортировки.

Идея же блочной сортировки состоит в том, что элементы последовательности разбиваются на блоки. Если *l* – минимальный, а *r* – максимальный элемент массива, то в первый блок попадут элементы от l до l + k, во втором – от l + k до l + 2k и т.д., где k = (r – l) / количество блоков.

Алгоритм относится к классу быстрых алгоритмов с линейным временем исполнения ***O(n)*** (на удачных входных данных).

На других (средних и плохих) последовательностях асимптотическая сложность алгоритма определяется сложно. Существует приблизительная оценка $O(n)+n\cdot O(2-1/n)=O(n)$

Если в качестве алгоритма сортировки блоков используется сортировка вставками, асимптотическая сложность алгоритма равна $O(n^2)$.

Алгоритм блочной сортировки сильно замедляется при большом количестве мало различающихся элементов или же при применении неудачной функции получения номера корзины по содержимому элемента. В некоторых таких случаях

15 https://ru.wikipedia.org/wiki/Блочная_сортировка

для строк, возникающих в реализациях основанного на сортировке строк алгоритма сжатия BWT, оказывается, что быстрая сортировка строк версии Седжвика значительно превосходит по скорости блочную сортировку.

В эффективной реализации рекомендуется использовать несколько идей:

1) без создания новых массивов: для этого использовать технику сортировки подсчетом – подсчитывают количество элементов в каждом блоке, префиксные суммы и, таким образом, позицию каждого элемента в массиве.

2) не запускать из пустых блоков: индексы непустых блоков заносятся в отдельный массив и блоки запускаются только от них.

3) выполнять с предварительной проверкой отсортированности последовательности. Это не ухудшит время работы, так как все равно нужно сделать проход с целью нахождения минимума и максимума, однако позволит алгоритму ускориться на частично отсортированных данных, так как элементы вставляются в новые блоки в том же порядке, что и в исходном массиве.

4) поскольку алгоритм довольно громоздкий, при небольшом количестве элементов он крайне неэффективен (до такой степени, что переход на сортировку вставками ускоряет работу примерно в 10 раз).

14. *Поразрядная сортировка, цифровая сортировка (Radix_sort)*[16]. Алгоритм сортировки выполняется за линейное время. Существует две версии этой сортировки, в которых, по-видимому, мало общего, кроме идеи воспользоваться представлением числа в какой-либо системе счисления (например, двоичной).

Изначально алгоритм предназначался для сортировки целых чисел, записанных цифрами. Но так как любая информация в памяти компьютеров записывается целыми числами, алгоритм пригоден для сортировки любых объектов, запись которых можно представить поразрядно, где «разряды» содержат сравнимые значения. Например, сортировать можно не только числа, записанные в виде набора цифр, но и строки, являющиеся набором символов, и вообще произвольные значения в памяти, представленные в виде набора байт.

Сравнение производится поразрядно: сначала сравниваются значения одного крайнего разряда, и элементы группируются по результатам этого сравнения, затем сравниваются значения следующего разряда, соседнего, и элементы либо упорядочиваются по результатам сравнения значений этого разряда внутри образованных на предыдущем проходе групп, либо переупорядочиваются в целом, но сохраняя относительный порядок, достигнутый при предыдущей сортировке. Затем аналогично делается для следующего разряда, и так до конца.

Из описания функционирования следует, что на каждом шаге достаточно стабильно сортировать элементы по новым k битам. Для этого идеально подходит сортировка подсчетом (необходимо 2^k памяти и времени, что немного при удачном выборе константы). Асимптотическая сложность $O(n)$, если считать, что числа фиксированного размера (а в противном случае нельзя было бы считать, что сравнение двух чисел выполняется за единицу времени).

Худшее время $O(w\,n)$, где w — количество бит, требуемых для хранения каждого ключа. Затраты памяти $O(w+n)$.

15. *Битонная сортировка (Bitonic_sort)*[17]. — *п*араллельный алгоритм — один из старейших параллельных алгоритмов [2] сортировки данных, метод для создания сортировочной сети. В основе алгоритма лежит понятие «битонной последовательности». Название было выбрано по аналогии с монотонной последовательностью[1]. Наряду с четно-нечетной параллельной сортировкой слиянием (которую рассмотрим в анализе параллельных сортировок), является одним из первых методов построения сортировочной сети для любого количества входов[2].

Идея данного алгоритма заключается в том, что исходный массив преобразуется в битонную последовательность — такую последовательность, которая сначала возрастает, а потом убывает. Ее можно эффективно отсортировать следующим образом: разбить последовательность на две части, создать два массива для этих частей: в первый добавим все элементы, равные минимуму из соответственных элементов каждой из двух частей, а во второй — равные максимуму.

Утверждается, что получатся две битонные последовательности, каждую из которых можно рекурсивно отсортировать тем же образом, после чего можно склеить две последовательности (так как любой элемент первой последовательности меньше или равен любому элементу второй последовательности).

Для того, чтобы преобразовать исходную последовательность в битонную последовательность требуется выполнить следующие действия:

- если последовательность состоит из двух элементов, можно завершить работу,
- иначе разделить последовательность пополам, рекурсивно вызвать от половинок алгоритм, после чего отсортировать первую часть в одном порядке, вторую в обратном порядке и склеить.

Очевидно, получится битонная последовательность.

Асимптотическая сложность описанного варианта алгоритма равна $O(n \cdot \log^2 n)$, так как при построении битонной последовательности использовали сортировку, работающую за $O(n \cdot \log n)$, а всего уровней было $\log n$. Отметим, что важно, чтобы размер массива должен быть равен степени двойки, так что, возможно, придется его дополнять фиктивными элементами (что не влияет на асимптотику). Однако, в литературе утверждается, что в худшем, среднем и лучшем случаем асимптотическая сложность равна $O(\log^2 n)$. Возможно, вариант имеет указанную асимптотику при использовании вспомогательной памяти объемом в размер не менее, чем размер последовательности.

16. *Timsort*[18] — гибридная сортировка, совмещающая сортировку вставками и сортировку слиянием. С момента появления уже стал стандартным алгоритмом сортировки в Python, OpenJDK 7 и Android JDK 1.5.

В реальном мире при решении большинства конкретных проектов (в частности, связанных с Big Data) сортируемые массивы данных часто содержат в себе упорядоченные подмассивы (последовательности). На таких данных *Timsort* существенно быстрее многих алгоритмов сортировки.

Идея алгоритма: разбить элементы последовательности на несколько подмассивов (подпоследовательностей) небольшого размера, при этом расширять подмассив (подпоследовательность), пока элементы в нем отсортированы.

---

[16] https://ru.wikipedia.org/wiki/поразрядная_сортировка

[17] https://ru.wikipedia.org/wiki/Битонная_сортировка — параллельный алгоритм, разработанный американским информатиком Кеннетом Бэтчером в 1968 году.

18 Алгоритм опубликован 2002 году Тимом Петерсом и назван его именем.

Затем отсортировать подмассивы (подпоследовательности) сортировкой вставками, пользуясь тем, что она эффективно работает на отсортированных последовательностях.

Далее сливать подмассивы (подпоследовательности) как в сортировке слиянием, беря их примерно равного размера (иначе время работы приблизится к квадратичному).

Для этого удобного хранить подмассивы (подпоследовательности) в стеке[19], поддерживая инвариант: чем дальше от вершины, тем больше размер.

Сливать подмассивы (подпоследовательности) на верхушке только тогда, когда размер третьего по отдаленности от вершины подмассива (подпоследовательности) больше или равен сумме их размеров. Асимптотическая сложность алгоритма: ***O(n)*** в лучшем случае и ***O(n·log n)*** в среднем и худшем случае. Реализация нетривиальна.

На первый взгляд, в таблице представлен огромный список сортировок, на любой выбор. Однако в этой таблице всего 7 адекватных алгоритмов (то есть с приемлемой сложностью и временем выполнения ***O(n log n)*** в *среднем* и *худшем* случае), а среди них только две могут похвастаться *стабильностью* и *сложностью* ***O(n)*** в *лучшем* случае. Один из этих двух алгоритмов — давно и хорошо всем известная *сортировка с помощью двоичного дерева*. А второй алгоритм — сортировка *Timsort*.

В заключение анализа приведенных сортировок, интересно привести сводную таблицу эффективности из Википедии:

Таблица 3. Сводная таблица эффективности современных сортировок

Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Selection sort	n^2	n^2	n^2	1	Depends
Timsort	n	$n \log n$	$n \log n$	n	Yes
Shell sort	n	$n(\log n)^2$	$O(n\log^2 n)$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes
Cycle sort	—	n^2	n^2	1	No
Library sort	—	$n \log n$	n^2	n	Yes
Patience sorting	—	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	—	$n \log n$	$n \log n$		
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	—	—	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
Bogosort	n	$n \cdot n!$	$n \cdot n! \to \infty$	1	No

Параллельные сортировки и распределенная обработка данных

Рассмотрим некоторые из параллельных сортировок, и проанализируем, чем отличается процесс разработки последовательного и параллельного алгоритма.

Целью применения параллельных вычислений во многих случаях является не только уменьшение времени вычисления расчетов, но и обеспечение возможности решения более сложных вариантов задач (решение которых не представляется возможным при использовании однопроцессорных вычислительных систем). Способность параллельного алгоритма эффективно использовать процессоры при повышенной сложности вычислений является важной характеристикой вычислительного алгоритма

Рассмотрим некоторые из параллельных сортировок, и проанализируем, чем отличается процесс разработки последовательного и параллельного алгоритма [5].

Прежде всего отметим, что, зная сложность последовательного алгоритма сортировки, знаем нижнюю оценку количества операций для упорядочения последовательности из n элементов.

Ускорение алгоритма может быть получено при использовании p (p>1) процессоров. Исходная упорядочиваемая последовательность в этом случае разделяется между p процессорами; в ходе сортировки данные пересылаются между процессорами и сравниваются меду собой. Результирующая последовательность, как правило также разделена между процессорами; при этом для устранения коллизий вводится некоторая система последовательной нумерации и обычно требуется, чтобы при завершении алгоритма сортировки значения, располагаемые на процессорах с меньшими номерами не превышали значений, располагаемых на процессорах с большими номерами.

Рассмотрим сортировки, для которых вся упорядочиваемая последовательность может быть размещена в оперативной памяти.

Как уже выясняли, основная операция любой сортировки — это операция обмена некоторых пар.

Обобщенная операция обмена параллельной сортировки

19 Стек (от англ. stack — стопка) — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (last-in, first-out — LIFO). Примером стека в реальной жизни может являться стопка тарелок: когда мы хотим вытащить тарелку, мы должны снять все тарелки выше.

Сначала рассмотрим ситуацию, когда число процессоров *p* совпадает с числом элементов *n* упорядочиваемой последовательности *p=n*, и на каждом из процессоров находится по одному элементу этой последовательности. Тогда сравнение элементов a_i и a_j, расположенных на процессорах p_i и p_j можно организовать следующим образом:

- выполнить взаимообмен значений a_i и a_j на процессорах p_j и p_i с сохранением на них исходных значений;
- сравнить на каждом процессоре p_i и p_j получившиеся пары значений (a_i , a_j); результаты сравнений используем для разделения данных между процессорами: на одном процессоре (например, p_i) остается наименьший элемент, а на другом (например, p_j) — наибольший элемент пары для дальнейшей обработки: $a_i'=\min(a_i , a_j)$, $a_j'=\max(a_i , a_j)$.

Рассмотренное параллельное обобщение параллельной операции сортировки может быть обобщено на число процессоров p<n, тогда процессор будет содержать не единственное значение, а часть (блок ***n/p***) сортируемой последовательности данных. Такая ситуация называется масштабируемостью параллельных вычислений.

Определим в качестве результата выполнения алгоритма сортировки: на каждом процессоре значения упорядочены, а порядок распределения блоков по процессорам соответствует линейной нумерации (то есть последнее значение на процессоре с меньшим значением p_i меньше либо равно первому значению на процессоре p_{i+1}, $0<i<=p-1$.

Блоки упорядочиваются обычно в самом начале сортировки на каждом процессоре в отдельности при помощи какого-либо быстрого последовательного алгоритма (начальная стадия параллельной сортировки). Далее по схеме одноэлементного сравнения, взаимодействие пары процессоров p_i и p_{i+1}для совместного упорядочения содержимого блоков A_i и A_{i+1} может быть осуществлено следующим образом:

- выполнить взаимообмен блоками между процессорами P_i и P_{i+1};
- объединить блоки A_i и A_{i+1} на каждом процессоре в один отсортированный блок двойного размера (при исходной упорядоченности блоков A_i и A_{i+1} процедура их объединения сводится к быстрой операции слияния упорядоченных наборов данных);
- разделить полученный двойной блок на две равные части и оставить одну из этих частей (например, с меньшими значениями данных) на процессоре P_i, а другую часть (с большими значениями) — на процессоре P_{i+1}

Рассмотренная процедура в литературе называется “сравнить и разделить” (compare-split). Отметим, что сформированные в результате такой процедуры блоки на процессорах P_i и P_{i+1} совпадают по размеру с исходными блоками A_i и A_{i+1}, и все значения, расположенные на процессоре P_i, не превышают значений на процессоре P_{i+1}.

Будем использовать эту процедуру “сравнить и разделить” в качестве базовой подзадачи для организации параллельных вычислений. Количество таких подзадач параметрически зависит от числа имеющихся процессоров. Блоки данных, относящиеся к подзадачам, изменяются в хоте выполнения сортировки.

В простых случаях размер блоков данных в подзадачах остается неизменным. В более сложных ситуациях (быстрая сортировка) объем располагаемых на процессорах данных может различаться — это может приводить к нарушению равномерной вычислительной загрузки процессоров.

Пузырьковая сортировка. Алгоритм чет-нечет сортировки.

Алгоритм пузырьковой сортировки в прямом виде не годится для реализации параллельного алгоритма, так как сравнение пар происходит строго последовательно. Поэтому для параллельной сортировки пузырьком используется не сам алгоритм, а его модификация — *чет-нечет перестановка*.

Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила для итерации метода: в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами.

Таким образом, на всех нечетных итерациях сравниваются пары (a_1, a_2), (a_3, a_4), …, (a_{n-1}, a_n) при четном n, а на *четных* итерациях обрабатываются элементы (a_2, a_3), (a_4, a_5), …, (a_{n-2}, a_{n-1})

После n-кратного повторения итерации сортировки исходный набор данных оказывается упорядоченным.

Получение параллельного варианта для чет-нечетной перестановки уже не представляет каких-либо затруднений: сравнение пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. В случае, когда ***p<n,*** когда количество процессоров меньше числа упорядочиваемых значений, процессоры содержат блоки данных **n/p,** и в качестве базовой подзадачи может быть использована операция “сравнить и разделить”.

Параллельный алгоритм чет-нечетной перестановки

```
ParallelOldEvenSort (double A[],int i)
  {int id=GetProcId();       //номер процесса
   int np=GetProcNum();  //количество процессов
   for(int i=0; i<np;i++)
       {if (i%2==1)           //нечетная итерация
         { if (id%2==1)       // нечетный номер процесса
//Сравнение- обмен с процессом-соседом справа
                 {if (id<np-1) compare_split_min(id+1);}
               else
//Сравнение- обмен с процессом-соседом слева
                 if (id<0  ) compare_split_max(id-1);
         }
        else   {//четная итерация
                    if (id%2==0)    // четный номер процесса
//Сравнение- обмен с процессом-соседом справа
                        {if (id<np-1) compare_split_min(id+1);}
               else
//Сравнение- обмен с процессом-соседом слева
                        compare_split_max(id-1);
       }
  }
```

}

Для пояснения такого параллельного способа сортировки в Таблице 1 приведен пример упорядочения данных при n=16, p=4 (то есть блок значений на каждом процессоре содержит n/p=4 элемента). В первом столбце приводится номер и тип итерации метода, перечислятся пары процессов, для которых параллельно выполняются операции “сравнить и разделить”. Взаимодействующие пары процессов выделены в таблице рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

В общем случае выполнение параллельного метода может быть прекращено, если в течение каких-либо двух последовательных итераций сортировки состояние упорядочиваемого набора данных не было изменено. Как результат, общее количество итераций может быть сокращено, и для фиксации таких моментов необходимо введение некоторого управляющего процессора, который определял бы состояние набора данных после выполнения каждой итерации сортировки. Однако трудоемкость такой коммуникационной операции (сборка на одном процессоре сообщений от всех процессоров) может оказаться столь значительной, что весь эффект от возможного сокращения итераций сортировки будет поглощен затратами на реализацию операций межпроцессорной передачи данных.

Таблица 4. Пример сортировки данных параллельным методом чет-нечетной перестановки

N и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 нечет (1,2),(3,4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 чет (2,3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
3 нечет (1,2),(3,4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 чет (2,3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88

Количество подзадач соответствует числу процессоров, масштабирование вычислений производить нет необходимости. Исходное распределение блоков упорядочиваемого набора данных по процессорам может быть произведено произвольным образом. Для эффективного выполнения рассмотренного параллельного алгоритма сортировки нужно, чтобы процессоры с соседними номерами имели непосредственные линии связи.

Анализ эффективности. Прежде всего, начальная последовательная сортировка (быстрая сортировка) характеризуется квадратичной сложностью этой части алгоритма. Но применение этой оценки сложности последовательного алгоритма приведет к искажению сложности параллельного алгоритма.

Как основной результат приведенных рассуждений можно сформулировать утверждение о том, что при определении показателей *ускорения* и *эффективности* параллельных вычислений в качестве оценки сложности последовательного способа решения рассматриваемой задачи следует использовать трудоемкость наилучших последовательных алгоритмов.

Предположим, что данная начальная сортировка может быть выполнена при помощи быстродействующих алгоритмов упорядочивания данных, тогда трудоемкость начальной стадии вычислений можно определить выражением вида: $T_p^1 \approx (n/p)\log_2(n/p)$.

Далее, на каждой выполняемой итерации параллельной сортировки взаимодействующие пары процессоров осуществляют передачу блоков между собой, после чего получаемые на каждом процессоре пары блоков объединяются при помощи процедуры слияния. Общее количество итераций не превышает величины p, и, как результат, общее количество операций этой части параллельных вычислений оказывается равным $T_p^2 = 2n$. С учетом полученных соотношений показатели эффективности и ускорения параллельного метода сортировки имеют вид:

$$S_p = \frac{n \times \log_2 n}{\frac{n}{p} \cdot \log_2(\frac{n}{p}) + 2n} = \frac{p\log_2 n}{\log_2(\frac{n}{p}) + 2p} \qquad E_p = \frac{n \times \log_2 n}{p\left(\frac{n}{p} \cdot \log_2(\frac{n}{p}) + 2n\right)} = \frac{\log_2 n}{\log_2(\frac{n}{p}) + 2p}$$

Если расширить приведенные выражения, учесть длительность выполняемых вычислительных операций и оценить трудоемкость операции передачи блоков между процессорами, общее время выполнения параллельного алгоритма сортировки определится следующим выражением:

$T_p = \mathrm{n}\left(\left(\frac{n}{p}\right)\log_2\left(\frac{n}{p}\right) + 2n\right)\tau + p \cdot (\alpha + \omega \cdot \frac{n}{p} / \beta)$, где τ — время выполнения базовой операции сортировки.

Таблица 4. Результаты вычислительных экспериментов для параллельного алгоритма *пузырьковой сортировки*

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10000	0,001422	0,002210	0,643439	0,003270	0,434862
20000	0,002991	0,004428	0,675474	0,004596	0,650783
30000	0,004612	0,006745	0,683766	0,006873	0,671032
40000	0,006297	0,008033	0,783891	0,009107	0,691446
50000	0,008014	0,009770	0,820266	0,010840	0,739299

Рассмотрим еще пару параллельных сортировок: Шелла и быструю сортировку.

Сортировка Шелла.

Как было выяснено для последовательных сортировок, общая идея сортировки Шелла состоит в сравнении на начальных стадиях сортировки пар значений, располагаемых достаточно далеко друг от друга

в упорядочиваемом наборе данных. Для алгоритма Шелла может быть предложен параллельный аналог, если топология коммуникационной сети может быть представлена в виде N-мерного куба, то есть количество процессоров равно $p=2^n$.

Выполнение сортировки в таком случае может быть разделено на два последовательных этапа. На первом этапе (N итераций) осуществляется взаимодействие процессоров, являющихся соседними в структуре куба (но эти процессоры могут оказаться далекими при линейной нумерации). Для нумерации в структуре куба можно применить код Грея. Формирование пар процессоров, взаимодействующих между собой при выполнении операции “сравнить и разделить”, может быть обеспечено при помощи простого правила: на каждой итерации i, $0<=i<N$, парными становятся процессоры, у которых различие в битовых представлениях их номеров имеется только в позиции $n-1$.

Второй этап состоит в реализации обычных итераций параллельного алгоритма чет-нечетной перестановки. Итерации данного этапа выполняются до прекращения фактического изменения сортируемого набора, и, тем самым, общее количество L таких итераций может быть различным – от 2 до p.

Рассмотрим схему сортировки массива из 16 элементов с помощью предложенного способа (процессоры – кружки, номера процессоров даны в битовом представлении). Данные оказываются упорядоченными уже после первого этапа и нет необходимости выполнять чет-нечетную перестановку.

С учетом представленного описания параллельного варианта алгоритма Шелла базовая подзадача может быть определена, как и в пузырьковой сортировке, на основе операции “сравнить и разделить”. При этом количество подздач всегда совпадает с числом имеющихся процессоров.

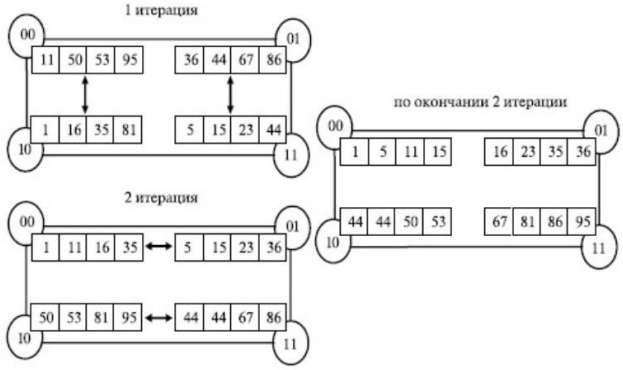


Рис. 13. Блок-схема выполнения параллельной сортировки Шелла для 4 процессоров

Распределение блоков упорядочиваемого набора данных по процессорам должно быть выбрано с учетом возможности эффективного выполнения операций “сравнить и разделить” при представлении топологии сети передачи данных в виде гиперкуба.

Анализ эффективности.

Для оценки эффективности параллельного аналога алгоритма Шелла могут быть использованы соотношения, полученные для параллельного метода пузырьковой сортировки. При этом следует только учесть двухэтапность алгоритма Шелла. С учетом данной особенности общее время выполнения этого параллельного метода может быть определено при помощи выражения

$$Tp=\left(\frac{n}{p}\right)log_2\left(\frac{n}{p}\right)\tau+(log_2 p+L)\left[\left(\frac{2n}{p}\right)\tau+\left(\alpha+\omega\cdot\left(\frac{n}{p}\right)/\beta\right)\right]$$

Эффективность параллельного варианта сортировки Шелла существенно зависит от значения L (скорость передачи данных между процессами). При малом значении величины L новый параллельный способ сортировки выполняется быстрее, чем ранее рассмотренный алгоритм чет-нечет сортировки.

Быстрая сортировка, параллельный алгоритм.

Параллельное обобщение алгоритма быстрой сортировки наиболее простым способом может быть получено, если топология коммуникационной сети может быть эффективно представлена в виде N-мерного гиперкуба ($p=2^n$). Пусть, как и раньше, исходный набор данных распределен межу процессорами блока одинакового размера n/p; результирующее расположение блоков должно соответствовать нумерации процессоров гиперкуба. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

- выбрать (каким-либо образом) ведущий элемент и разослать его по всем процессорам системы (например, в качестве ведущего элемента можно взять среднее арифметическое элементов, расположенных на выбранном ведущем процессоре);
- разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;
- образовать пары процессоров, для которых битовое представление номеров отличается только в позиции n и осуществить взаимообмен данными между этими процессорами.

В результате выполнения такой итерации сортировки исходный набор оказывается разделенным на две части, одна из которых (со значениями меньшими, чем значение ведущего элемента) располагается на процессорах, в битовом представлении номеров которых бит n равен 0. Таких процессоров всего $p/2$ и, таким образом, исходный n мерный гиперкуб тоже оказывается разделенным на два гиперкуба размерности $n-1$. К этим гиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После n-кратного повторения подобных операций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы.

На блок-схеме ниже представлен пример упорядочения данных при $n=16$, $p=4$ (то есть блок каждого процессора содержит 4 элемента). На ней процессоры изображены в виде прямоугольников, внутри которых показано содержимое упорядочиваемых блоков данных; значения блоков данных приводятся в начале и при

каждой итерации сортировки. Взаимодействующие пары процессоров соединены двунаправленными стрелками. Для разделения данных выбирались наилучшие значения ведущих элементов: на первой итерации для всех процессоров использовалось значение 0, на второй итерации для пары процессоров 0, 1 ведущий элемент равен -5, для пары процессоров 2,3 это значение было принято равным 4.

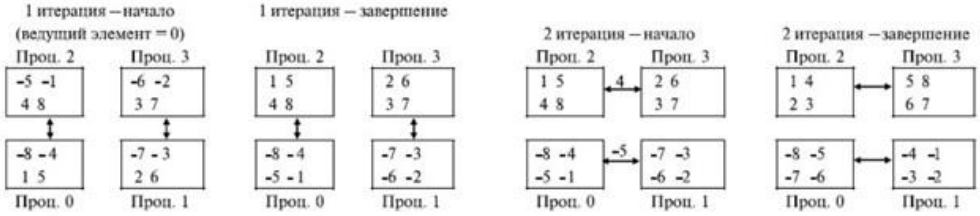


Рис. 14. Блок-схема упорядочения данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

Как и ранее, в качестве базовой подзадачи для организации параллельных вычислений может быть выбрана операция “сравнить и разделить”, а количество подзадач совпадает с числом используемых процессоров. Распределение подзадач по процессорам должно производиться с учетом возможности эффективного выполнения алгоритма при представлении топологии сети передачи данных в виде гиперкуба.

Эффективность параллельного метода быстрой сортировки, как и в последовательном варианте, во многом зависит от правильности выбора значений ведущих элементов. Определение общего правила для выбора этих значений представляется затруднительным. Сложность такого выбора может быть снижена, если выполнить упорядочение локальных блоков процессоров перед началом сортировки и обеспечить однородное распределение сортируемых данных между процессами вычислительной системы.

Определим вычислительную сложность алгоритма сортировки. На каждой из $log_2 p$ итерации сортировки каждый процессор осуществляет деление блока относительно ведущего элемента, сложность этой операции составляет ***n/p*** операций (будем предполагать, что на каждой итерации сортировки каждый блок делится на равные по размеру части).

При завершении вычислений процессор выполняет сортировку своих блоков, что может быть выполнено при использовании быстрых алгоритмов за $(n/p)log_2(n/p)$ операций.

Таким образом, общее время вычислений параллельного алгоритма быстрой сортировки составляет $T_p(\text{calc}) = \left[\left(\frac{n}{p}\right) log_2 p + \left(\frac{n}{p}\right) log_2\left(\frac{n}{p}\right)\right]\tau$, где τ есть время выполнения базовой операции перестановки.

Сложность выполнения коммуникационных операций определяется общим количеством межпроцессорных обменов для рассылки ведущего элемента на n – мерном гиперкубе и может быть ограничено оценкой

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{log_2 p(log_2 p + 1)}{2} \approx (log_2 p)^2$$

При используемых предположениях (выбор ведущих элементов осуществляется наилучшим образом) количество итераций алгоритма равно $log_2 p$, а объем передаваемых данных между процессорами всегда равен половине блока, то есть величине $(n/p)/2$. При таких условиях коммуникационная сложность параллельного алгоритма быстрой сортировки определяется при помощи соотношения

$T_p(\text{comm}) = (log_2 p)^2(\alpha + \omega/\beta) + log_2 p(\alpha + \omega(n/2p)/\beta)$, где α – латентность, β – пропускная способность сети, а ω есть размер элемента набора в байтах. С учетом всех полученных соотношений общая трудоемкость алгоритма оказывается равной $T_p = \left[\left(\frac{n}{p}\right) log_2 p + \left(\frac{n}{p}\right) log_2\left(\frac{n}{p}\right)\right]\tau + (log_2 p)^2(\alpha + \omega/3) + log_2 p(\alpha + u(n/2p)\beta)$

ЗАКЛЮЧЕНИЕ

В статье рассмотрены важные теоретические и практические вопросы реализации одной из важнейших областей обработки данных, исследован большой объем важнейших алгоритмов, используемых практически во всех областях, связанных с разработкой или сопровождением программных продуктов, в том числе и специализированных, в частности, проведен анализ эффективности современных методов последовательных и параллельных сортировок.

Подробно рассмотрены последовательные сортировки, показаны их сильные и слабые стороны, применимость и эффективность в конкретных условиях на различной длине последовательностей.

Рассмотрены также и параллельные сортировки, которые в современных условиях все больше и больше востребованы при обработке больших данных с использованием распределенных вычислений. Это значительно более сложная область исследований, поэтому в главе значительное место занял материал, касающийся как собственно понятия «больших данных», так и основных принципов параллельной обработки данных, без которого было бы сложно разобраться с особенностями разработки параллельных алгоритмов.

Результаты статьи интересны и весьма важны также с точки зрения рассмотрения методов их реализации, актуализации представления их в программах, что позволяет более ясно представлять особенности их функционирования. Статья в большой мере полезна для углубленной подготовки специалистов Военно-морского флота в области разработки общего и специального программного обеспечения и обучения курсантов высших военных учебных заведений по инженерным специальностям.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Стивенс Род. Алгоритмы теория и практическое применение, 2016. –126 с.
2. Robert Horvick. Алгоритмы сортировки данных. [Электронный ресурс], 2020. URL: https://tproger.ru/translations/sorting-for-beginners/ (дата обращения 08.03.2020).
3. Кен Браунси. Структура данных и реализация в C++, 2016. –138с.
4. Никлаус Вирт. Алгоритмы и структуры данных, 2017. –124с.
5. В.П. Гергель Теория и практика параллельных вычислений, Национальный открытый университет “Интуит”, 2016, 501с.