Высокопроизводительные параллельные вычисления

Лекция №7
Тема: Параллельные методы сортировки

Козбакова Айнур Холдасовна– PhD, профессор, кафедра «Информационные системы» Факультет «Инжиниринг и информационных технологии» 01.09.2025 г.

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений $S = a_1, a_2, \ldots, a_n$ в порядке монотонного возрастания или убывания

$$S \sim S' = (a'_1, a'_2, \ldots, a'_n) : a'_1 \le a'_2 \le \ldots \le a'_n$$

Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой. Так, для ряда известных простых методов (*пузырьковая сортировка*, *сортировка* включением и др.) количество необходимых операций определяется квадратичной зависимостью от числа упорядочиваемых данных $T \sim n^2$.

Ускорение сортировки может быть обеспечено при использовании нескольких (p>1) процессоров. Исходный упорядочиваемый набор в этом случае разделяется между процессорами; в ходе *сортировки* данные пересылаются между процессорами и сравниваются между собой. Результирующий (упорядоченный) набор, как правило, также разделен между процессорами; при этом для систематизации такого разделения для процессоров вводится та или иная система последовательной нумерации и обычно требуется, чтобы при завершении *сортировки* значения, располагаемые на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

Оставляя подробный анализ проблемы *сортировки* для отдельного рассмотрения, основное внимание уделим изучению параллельных способов выполнения для ряда широко известных методов внутренней *сортировки*, когда все упорядочиваемые данные могут быть размещены полностью в оперативной памяти ЭВМ.

Принципы распараллеливания

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах *сортировки*, можно заметить, что многие методы основаны на применении одной и той же *базовой операции "сравнить и переставить"* (*compare-exchange*), состоящей в сравнении той или иной пары значений из сортируемого набора данных и перестановке этих значений, если их порядок не соответствует условиям *сортировки*.Целенаправленное применение данной операции позволяет упорядочить данные; в способах выбора пар значений для сравнения, собственно, и проявляется различие алгоритмов *сортировки*.

Для параллельного обобщения выделенной базовой операции *сортировки* рассмотрим первоначально ситуацию, когда количество процессоров совпадает с числом сортируемых значений (т. е. p=n) и на каждом из процессоров содержится только по одному значению исходного набора данных. Тогда сравнение значений a_i и a_j, располагаемых соответственно на процессорах P_i и P_j, можно организовать следующим образом (параллельное обобщение базовой операции *сортировки*):

- выполнить взаимообмен имеющихся на процессорах P_i и P_j значений (с сохранением на этих процессорах исходных элементов);
- сравнить на каждом процессоре P_i и P_j получившиеся одинаковые пары значений (a_i, a_j); результаты сравнения используются для разделения данных между процессорами – на одном процессоре (например, P_i) остается меньший элемент, другой процессор (т. е. P_j) запоминает для дальнейшей обработки большее значение пары

$$a'_i = \min(a_i, a_j), \quad a'_j = \max(a_i, a_j)$$

Масштабирование параллельных вычислений

Рассмотренное параллельное обобщение базовой операции *сортировки* может быть надлежащим образом адаптировано и для случая p<n, когда количество процессоров

меньше числа упорядочиваемых значений. В данной ситуации каждый процессор будет содержать уже не единственное значение, а часть (блок размера n/p) сортируемого набора данных.

Определим в качестве *результата выполнения параллельного алгоритма сортировки* такое состояние упорядочиваемого набора данных, при котором имеющиеся на процессорах данные упорядочены, а порядок распределения блоков по процессорам соответствует линейному порядку нумерации (т. е. значение последнего элемента на процессоре P_i меньше значения первого элемента на процессоре P_{i+1}, где $0 \le i < p-1$ или равно ему). Блоки обычно упорядочиваются в самом начале *сортировки* на каждом процессоре в отдельности при помощи какого-либо быстрого алгоритма (начальная стадия параллельной *сортировки*). Далее, следуя 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 блоков 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, соответственно) – на процессоре P_{i+1}

$$[A_i \cup A_{i+1}]_{copm} = A'_i \cup a'_{i+1} : \forall a'_i, \in A'_i,\ \forall a'_j \in A'_{i+1} \Rightarrow a'_i \le a'_j.$$

Рассмотренная процедура обычно именуется в литературе *операцией "сравнить и разделить"* (*compare-split*). Следует отметить, 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}.

Определенная выше операция "сравнить и разделить" может быть использована в качестве *базовой подзадачи* для организации параллельных вычислений. Как следует из построения, количество таких подзадач параметрически зависит от числа имеющихся процессоров, и, таким образом, проблема масштабирования вычислений для параллельных алгоритмов *сортировки* практически отсутствует. Вместе с тем следует отметить, что относящиеся к подзадачам блоки данных изменяются в ходе выполнения *сортировки*. В простых случаях размер блоков данных в подзадачах остается неизменным. В более сложных ситуациях (как, например, в алгоритме *быстрой сортировки*) объем располагаемых на процессорах данных может различаться, что может приводить к нарушению равномерной вычислительной загрузки процессоров.

Пузырьковая сортировка

Последовательный алгоритм

Последовательный *алгоритм пузырьковой сортировки* (*the bubble sort algorithm*) сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности $(a_1, a_2, ..., a_n)$ алгоритм сначала выполняет n-1 базовых операций "сравнения-обмена" для последовательных пар элементов (a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n).

В результате после первой итерации алгоритма самый большой элемент перемещается ("всплывает") в конец последовательности. Далее последний элемент в преобразованной

последовательности может быть исключен из рассмотрения, и описанная выше процедура применяется к оставшейся части последовательности (a'_1, a'_2, ..., a'_{n-1}).

Как можно увидеть, последовательность будет отсортирована после n-1 итерации. *Эффективность пузырьковой сортировки* может быть улучшена, если завершать алгоритм в случае отсутствия каких- либо изменений сортируемой последовательности данных в ходе какой-либо итерации *сортировки*.

Алгоритм чет-нечетной перестановки

Алгоритм *пузырьковой сортировки* в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как *метод чет-нечетной перестановки* (*the odd-even transposition method*) Суть модификации состоит в том, что в алгоритм *сортировки* вводятся два разных правила выполнения итераций 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 итераций *сортировки* исходный набор данных оказывается упорядоченным.

Последовательный алгоритм чет-нечетной перестановки

```
// Последовательный алгоритм чет-нечетной перестановки
void OddEvenSort(double A[], int n) {
  for (int i = 1; i < n; i++) {
    if (i % 2 == 1) {    // нечетная итерация
      for (int j = 0; j < n/2 - 2; j++)
        compare_exchange(A[2*j + 1], A[2*j + 2]);
      if (n % 2 == 1) // сравнение последней пары при нечетном n
        compare_exchange(A[n - 2], A[n - 1]);
    }
    else    // четная итерация
      for (int j = 1; j < n/2 - 1; j++)
        compare_exchange(A[2*j], A[2*j + 1]);
  }
}
```

Определение подзадач и выделение информационных зависимостей

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

```
// Параллельный алгоритм чет-нечетной перестановки
ParallelOddEvenSort(double A[], int n) {
  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
    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 табл.6.1 приведен пример упорядочения данных при n=16, p=4 (т.е. блок значений на каждом процессоре содержит n/p=4 элемента). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются 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 6.1. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итераци	**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

Масштабирование и распределение подзадач по процессорам

Количество подзадач соответствует числу имеющихся процессоров, и поэтому необходимости в проведении масштабирования вычислений не возникает. Исходное распределение блоков упорядочиваемого набора данных по процессорам может быть выбрано совершенно произвольным образом. Для эффективного выполнения рассмотренного параллельного алгоритма *сортировки* нужно, чтобы процессоры с соседними номерами имели прямые линии связи.

Анализ эффективности

При анализе *эффективности*, как и ранее, вначале проведем общую оценку сложности рассмотренного параллельного алгоритма *сортировки*, а затем дополним полученные соотношения показателями трудоемкости выполняемых коммуникационных операций.

Определим первоначально трудоемкость последовательных вычислений. При рассмотрении данного вопроса алгоритм *пузырьковой сортировки* позволяет продемонстрировать следующий важный момент. Как уже отмечалось в начале этой лекции, использованный для распараллеливания последовательный метод упорядочивания данных характеризуется квадратичной зависимостью сложности от числа упорядочиваемых данных, т.е. $T_1 \sim n^2$. Однако применение подобной оценки сложности последовательного алгоритма приведет к искажению исходного целевого назначения критериев качества параллельных вычислений – показатели *эффективности* в этом случае будут характеризовать используемый способ параллельного выполнения данного конкретного метода *сортировки*, а не результативность использования параллелизма для задачи упорядочивания данных в целом как таковой. Различие состоит в том, что для *сортировки* могут быть применены более эффективные последовательные алгоритмы, трудоемкость которых имеет порядок

$$T_1 \sim n \log_2 n,$$

и чтобы сравнить, насколько быстрее могут быть упорядочены данные при использовании параллельных вычислений, в обязательном порядке должна применяться именно эта оценка сложности. 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. Параллельные методы решения

задач должны сравниваться с наиболее быстродействующими последовательными способами вычислений!

Определим теперь сложность рассмотренного параллельного алгоритма упорядочивания данных. Как отмечалось ранее, на начальной стадии работы метода каждый процессор проводит упорядочивание своих блоков данных (размер блоков при равномерном распределении данных равен n/p). Предположим, что данная начальная *сортировка* может быть выполнена при помощи быстродействующих алгоритмов упорядочивания данных, тогда трудоемкость начальной стадии вычислений можно определить выражением вида:

$$T_p^1 = (n/p)\log_2(n/p)$$

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

$$T_p^2 = 2p(n/p) = 2n$$

С учетом полученных соотношений показатели *эффективности* и *ускорения* параллельного метода *сортировки* имеют вид:

$$S_p = \frac{n\log_2 n}{(n/p)\log_2(n/p)+2n} = \frac{p\log_2 n}{\log_2(n/p)+2p},$$
$$E_p = \frac{n\log_2 n}{p((n/p)\log_2(n/p)+2n)} = \frac{\log_2 n}{\log_2(n/p)+2p}.$$

Расширим приведенные выражения – учтем длительность выполняемых вычислительных операций и оценим трудоемкость операции передачи блоков между процессорами. При использовании модели Хокни общее время всех выполняемых в ходе *сортировки* операций передачи блоков можно оценить при помощи соотношения вида:

$$T_p(comm) = p\cdot(\alpha + w\cdot(n/p)/\beta),$$

где α– латентность, β – пропускная способность сети передачи данных, а w есть размер элемента упорядочиваемых данных в байтах. С учетом трудоемкости коммуникационных действий общее время выполнения параллельного алгоритма *сортировки* определяется следующим выражением:

$$T_p = ((n/p)\log_2(n/p) + 2n)\tau + p\cdot(\alpha + w\cdot(n/p)/\beta),$$

где τ есть время выполнения базовой операции *сортировки*.

Основная литература: [93-106]; 3[135-182].

Дополнительная литература: 11[72-90] .

Контрольные вопросы:

1. В чем состоит постановка задачи *сортировки* данных?
2. Приведите несколько примеров алгоритмов *сортировки*. Какова вычислительная сложность приведенных алгоритмов?
3. Какая операция является базовой для задачи *сортировки* данных?

4. В чем суть параллельного обобщения базовой операции задачи *сортировки* данных?
5. Что представляет собой алгоритм чет-нечетной перестановки?