10. Сортировка данных

Сортировка является одной из типовых проблем обработки данных и обычно понимается как задача размещения элементов неупорядоченного набора значений

$$S = \{a_1, a_2, ..., a_n\}$$

в порядке монотонного возрастания или убывания

$$S \sim S' = \{(a_1', a_2', ..., a_n') : a_1' \le a_2' \le ... \le a_n'\}$$

(здесь и далее все пояснения для краткости будут даваться только на примере упорядочивания данных по возрастанию).

Возможные способы решения этой задачи широко обсуждаются в литературе; один из наиболее полных обзоров *алгоритмов сортировки* содержится в работе Кнута (1981), среди последних изданий может быть рекомендована работа Кормена, Лейзерсона и Ривеста (1999).

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

$$T \sim n^2 .$$

Для более эффективных алгоритмов (сортировка слиянием, сортировка Шелла, быстрая сортировка) трудоемкость определяется величиной

$$T \sim n \log_2 n .$$

Данное выражение дает также нижнюю оценку необходимого количества операций для упорядочивания набора из n значений; алгоритмы с меньшей трудоемкостью могут быть получены только для частных вариантов задачи.

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, здесь основное внимание мы уделим изучению параллельных способов выполнения для ряда широко известных *методов внутренней сортировки*, когда все упорядочиваемые данные могут быть размещены полностью в оперативной памяти ЭВМ.

10.1. Принципы распараллеливания

При внимательном рассмотрении способов упорядочивания данных, применяемых в алгоритмах сортировки, можно обратить внимание, 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 перестановки этих значений, если их порядок не соответствует условиям сортировки.

```
// базовая операция сортировки
if ( A[i] > A[j] ) {
  temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

Пример 10.1. Операция "сравнить и переставить"

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_i, можно организовать следующим образом (*параллельное обобщение базовой операции сортировки*):

- выполнить взаимообмен имеющихся на процессорах P_i и P_i значений (с сохранением на этих процессорах исходных элементов),
- сравнить на каждом процессоре P_i и P_i 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

$$a_i' = \min(a_i, a_j) , \; a_j' = \max(a_i, a_j) .$$

10.2. Масштабирование параллельных вычислений

Рассмотренное параллельное обобщение базовой операции сортировки может быть надлежащим образом адаптировано и для случая $p < n$, когда количество процессоров является меньшим 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 или равно значения первого элемента на процессоре P_{i+1}, где $0 \le i < p-1$).

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 как *операция "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 неизмененным. В более сложных ситуациях (как, например, в алгоритме быстрой сортировки – см. подраздел 10.5) объем располагаемых на процессорах данных может различаться, что может приводить к нарушению равномерной вычислительной загрузки процессоров.

10.3. Пузырьковая сортировка

10.3.1. Последовательный алгоритм

Последовательный алгоритм *пузырьковой сортировки* (см., например, Кнут (1981), Кормен, Лейзерсон и Ривест (1999)) сравнивает и обменивает соседние элементы в последовательности, которую нужно отсортировать. Для последовательности

$(a_1, a_2, \ldots, a_n)$

алгоритм сначала выполняет $n-1$ базовых операций "сравнения-обмена" для последовательных пар элементов

$(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$.

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

$(a'_1, a'_2, \ldots, a'_{n-1})$.

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 10.1.
// Последовательный алгоритм пузырьковой сортировки
BubbleSort(double A[], int n) {
  for (i=0; i<n-1; i++)
    for (j=0; j<n-i; j++)
      compare_exchange(A[j], A[j+1]);
}
```

Алгоритм 10.1. Последовательный алгоритм пузырьковой сортировки

10.3.2. 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* (*odd-even transposition*) – см., например, Kumar et al. (2003). 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

$(a_1, a_2), (a_3, a_3), ..., (a_{n-1},a_n)$ (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.

```
// Алгоритм 10.2
// Последовательный алгоритм чет-нечетной перестановки
OddEvenSort ( double A[], int n ) {
  for ( i=1; i<n; i++ ) {
    if ( i%2==1 ) { // нечетная итерация
      for ( j=0; j<n/2-2; j++ )
        compare_exchange(A[2j+1],A[2j+2]);
      if ( n%2==1 ) // сравнение последней пары при нечетном n
        compare_exchange(A[n-2],A[n-1]);
     }
     if ( i%2==0 )    // четная итерация
       for ( j=1; j<n/2-1; j++ )
         compare_exchange(A[2j],A[2j+1]);
  }
}
```

Алгоритм 10.2. Последовательный алгоритм чет-нечетной перестановки

10.3.3. 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 является меньшим числа упорядочиваемых значений, процессоры содержат блоки данных размера n/p, и в качестве *базовой подзадачи* может быть использована операция "сравнить и разделить" (см. подраздел 10.2).

```
// Алгоритм 10.3
// 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
        // сравнение-обмен с процессом - соседом слева
        if ( id > 0 ) compare_ split_max(id-1);
    }
    if ( i%2 == 0 ) { // 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);
    }
```

```
    }
}
```

Алгоритм 10.3. Параллельный алгоритм чет-нечетной перестановки

Для пояснения такого параллельного способа сортировки в табл. 10.1 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 двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

Таблица 10.1. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ 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

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.

10.3.4. Масштабирование и распределение подзадач по процессорам

Как отмечалось ранее, количество подзадач соответствует числу имеющихся процессоров и, как результат, 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 необходимым является, чтобы процессоры с соседними номерами имели прямые линии связи.

10.3.5. 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 данного раздела, 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

$$T_1 = n\log_2 n, \qquad (10.1)$$

и для сравнения, насколько быстрее могут быть упорядочены данные при использовании параллельных вычислений, в обязательном порядке должна использоваться именно данная оценка сложности. Как основной результат выполненных 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 является равным 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:

$$T_p^1 = (n/p)\log_2(n/p). \qquad (10.2)$$

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

$$T_p^2 = 2p(n/p) = 2n. \qquad (10.3)$$

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

$$S_p = \frac{n\log_2 n}{(n/p)\log_2(n/p)+2n} = \frac{p\log_2 n}{\log_2(n/p)+2p}, \quad E_p = \frac{n\log_2 n}{p((n/p)\log_2(n/p)+2n)} = \frac{\log_2 n}{\log_2(n/p)+2p}. \qquad (10.4)$$

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:

$$T_p(comm) = p\cdot(\alpha + w\cdot(n/p)/\beta), \qquad (10.5)$$

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:

$$T_p = ((n/p)\log_2(n/p)+2n)\tau + p\cdot(\alpha + w\cdot(n/p)/\beta), \qquad (10.6)$$

где τ есть время выполнения базовой операции сортировки.

10.3.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма пузырьковой сортировки проводились при условиях, указанных в п. 7.6.5, и состоят в следующем.

The experiments were carried out on the computational cluster on the basis of the processor Intel XEON 4 EM64T 3000 Mhz and Gigabit Ethernet under OS Microsoft Windows Server 2003 Standart x64 Edition (см. п. 1.2.3).

Для оценки длительности τ базовой скалярной операции алгоритма сортировки проводилось решение задачи упорядочивания при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 10.41 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 130 мкс и 53,29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, т. е. величина w равна 8 байт.

Результаты вычислительных экспериментов приведены в табл. 10.1. Эксперименты выполнялись с использованием двух и четырех процессоров.

Таблица 10.1. Результаты вычислительных экспериментов по исследованию параллельного алгоритма пузырьковой сортировки

Количество	Последовательный	Параллельный алгоритм

элементов	алгоритм	2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,002210	0,643439	0,003270	0,434862
20 000	0,002991	0,004428	0,675474	0,004596	0,650783
30 000	0,004612	0,006745	0,683766	0,006873	0,671032
40 000	0,006297	0,008033	0,783891	0,009107	0,691446
50 000	0,008014	0,009770	0,820266	0,010840	0,739299

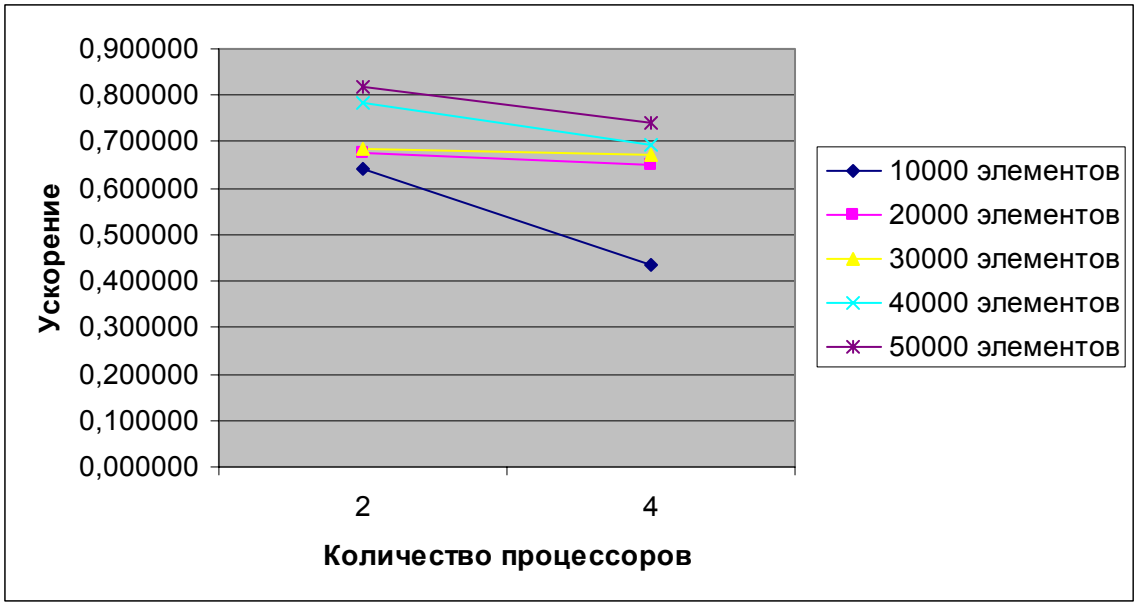

Рис. 10.1. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма пузырьковой сортировки

Как можно заметить из приведенных результатов вычислительных экспериментов, параллельный вариант алгоритма сортировки работает медленнее исходного последовательного метода пузырьковой сортировки, т. к объем передаваемых данных между процессорами является достаточно большим и сопоставим с количеством выполняемых вычислительных операций (и этот дисбаланс объема вычислений и сложности операций передачи данных увеличивается с ростом числа процессоров).

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.5) приведено в таблице 10.2 и на рис. 10.2.

Таблица 10.2. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма пузырьковой сортировки

Размер данных	Параллельный алгоритм			
	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,002003	0,002210	0,002057	0,003270
20 000	0,003709	0,004428	0,003366	0,004596
30 000	0,005455	0,006745	0,004694	0,006873
40 000	0,007227	0,008033	0,006035	0,009107
50 000	0,009018	0,009770	0,007386	0,010840

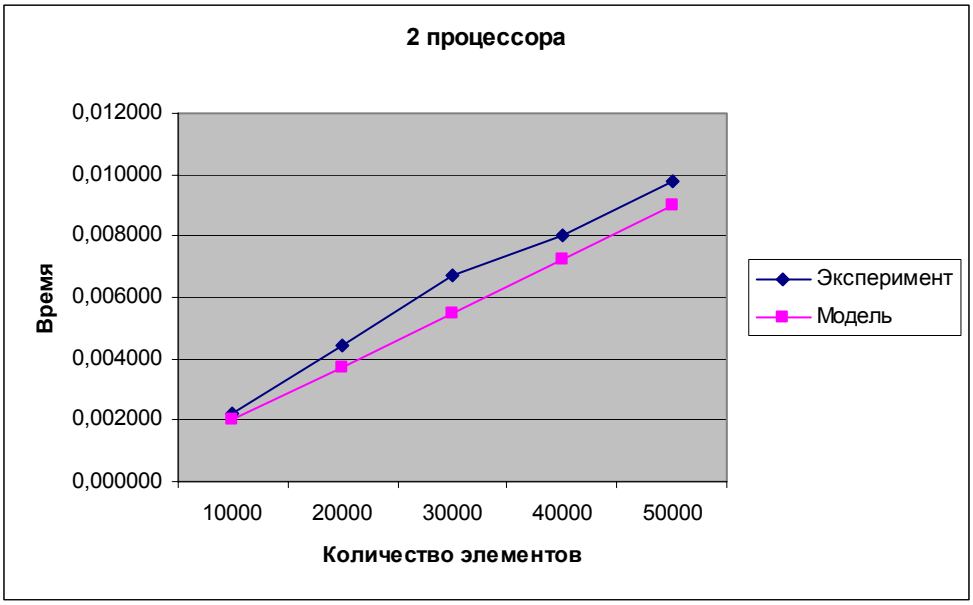

Рис. 10.2. График зависимости экспериментального и теоретического времени проведения эксперимента на двух процессорах от объема исходных данных

10.4. Сортировка Шелла

10.4.1. Последовательный алгоритм

Общая идея *сортировки Шелла* (см., например, Кнут (1981), Кормен, Лейзерсон и Ривест (1999)) состоит в сравнении на начальных стадиях сортировки пар значений, располагаемых достаточно далеко друг от друга в упорядочиваемом наборе данных. Такая модификация метода сортировки позволяет быстро переставлять далекие неупорядоченные пары значений (сортировка таких пар обычно требует большого количества перестановок, если используется сравнение только соседних элементов).

Общая схема метода состоит в следующем. На первом шаге алгоритма происходит упорядочивание элементов $n/2$ пар $(a_i, a_{n/2+i})$ для $1 \le i \le n/2$. Далее на втором шаге упорядочиваются элементы в $n/4$ группах из четырех элементов $(a_i, a_{n/4+1}, a_{n/2+1}, a_{3n/4+1})$ для $1 \le i \le n/4$. На третьем шаге упорядочиваются элементы уже в $n/4$ группах из восьми элементов и т.д. На последнем шаге упорядочиваются элементы сразу во всем массиве $(a_1, a_2, \ldots, a_n)$. На каждом шаге для упорядочивания элементов в группах используется метод сортировки вставками. Как можно заметить, общее количество итераций алгоритма Шелла является равным $log_2 n$.

В более полном виде алгоритм Шелла может быть представлен следующим образом.

```
// Алгоритм 10.4
// Последовательный алгоритм сортировки Шелла
ShellSort(double A[], int n){
  int incr = n/2;
  while( incr > 0 ) {
    for ( int i=incr+1; i<n; i++  ) {
      j = i-incr;
      while ( j > 0 )
        if ( A[j] > A[j+incr] ){
          swap(A[j], A[j+incr]);
          j = j - incr;
        }
        else j = 0;
    }
    incr = incr/2;
  }
}
```

Алгоритм 10.4. Последовательный алгоритм сортировки Шелла

10.4.2. Организация параллельных вычислений

Для алгоритма Шелла может быть предложен параллельный аналог метода (см., например, Kumar et al. (2003)), если топология коммуникационный сети может быть представлена в виде N-мерного гиперкуба (т.е. 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 гиперкуба (но эти процессоры могут оказаться далекими при линейной нумерации; для установления соответствия двух систем нумерации процессоров может быть использован, как и ранее, код Грея – см. раздел 3). Формирование пар процессоров, взаимодействующих между собой при выполнении операции "сравнить и разделить", может быть обеспечено при помощи следующего простого правила – на каждой итерации i, $0 \le i < N$, парными становятся процессоры, у которых различие в битовых представлении их номеров имеется только в позиции N-i.

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.

На рис. 10.3 показан пример сортировки массива из 16 элементов с помощью рассмотренного способа. Нужно заметить, что данные оказываются упорядоченными уже после первого этапа и нет необходимости выполнять чет-нечетную перестановку.

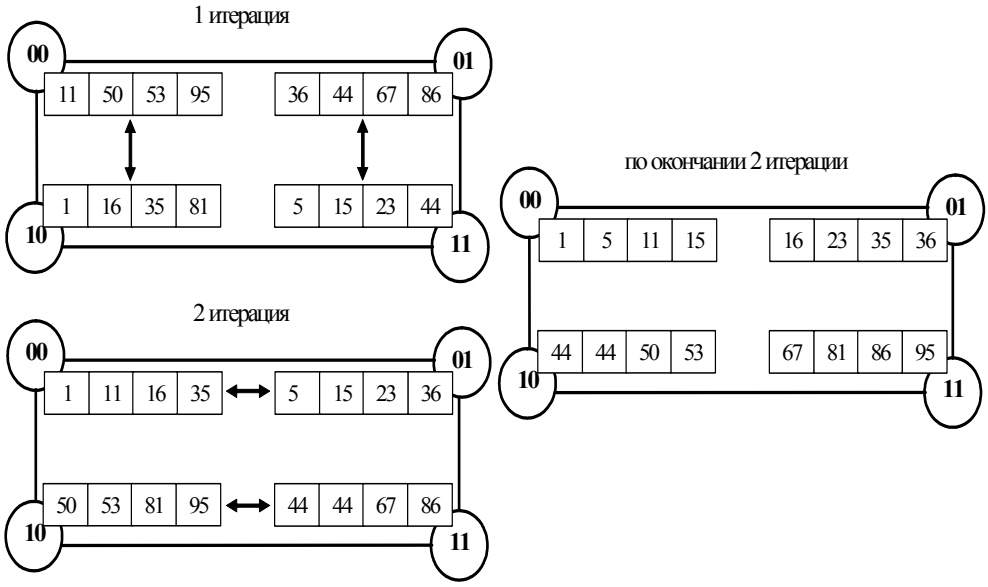

Рис. 10.3. Пример работы алгоритма Шелла для 4 процессоров (процессоры показаны кружками, номера процессоров даны в битовом представлении)

С учетом представленного описания параллельного варианта алгоритма Шелла *базовая подзадача* для организации параллельных вычислений, как и ранее (см. п. 10.3.3), может быть определена на основе операции "сравнить и разделить". Как результат, количество подзадач всегда совпадает с числом имеющихся процессоров (размер блоков данных в подзадачах равен n/p) и не возникает проблемы масштабирования. 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.

10.4.3. Анализ эффективности

Для оценки эффективности параллельного аналога алгоритма Шелла могут быть использованы соотношения, полученные для параллельного метода пузырьковой сортировки (см. п. 10.3.5). При этом следует только учесть двухэтапность алгоритма Шелла – с учетом данной особенности общее время выполнения нового параллельного метода может быть определено при помощи выражения

$$T_p = (n/p)\log_2(n/p)\tau + (\log_2 p + L)[(2n/p)\tau + (\alpha + w\cdot(n/p)/\beta)]. \qquad (10.7)$$

Как можно заметить, эффективность параллельного варианта сортировки Шелла существенно зависит от значения L – при малом значении величины L новый параллельный способ сортировки выполняется быстрее, чем ранее рассмотренный алгоритм чет-нечетной перестановки.

10.4.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта сортировки Шелла проводились при тех же условиях, что и ранее выполненные эксперименты (см. п. 10.3.4).

Результаты вычислительных экспериментов приведены в табл. 10.4. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 10.4. Результаты вычислительных экспериментов по исследованию параллельного алгоритма сортировки Шелла

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Time	Speedup	Time	Speedup
10 000	0,001422	0,002959	0,480568	0,007509	0,189373
20 000	0,002991	0,004557	0,656353	0,009826	0,304396
30 000	0,004612	0,006118	0,753841	0,012431	0,371008
40 000	0,006297	0,008461	0,744238	0,017009	0,370216
50 000	0,008014	0,009920	0,807863	0,019419	0,412689

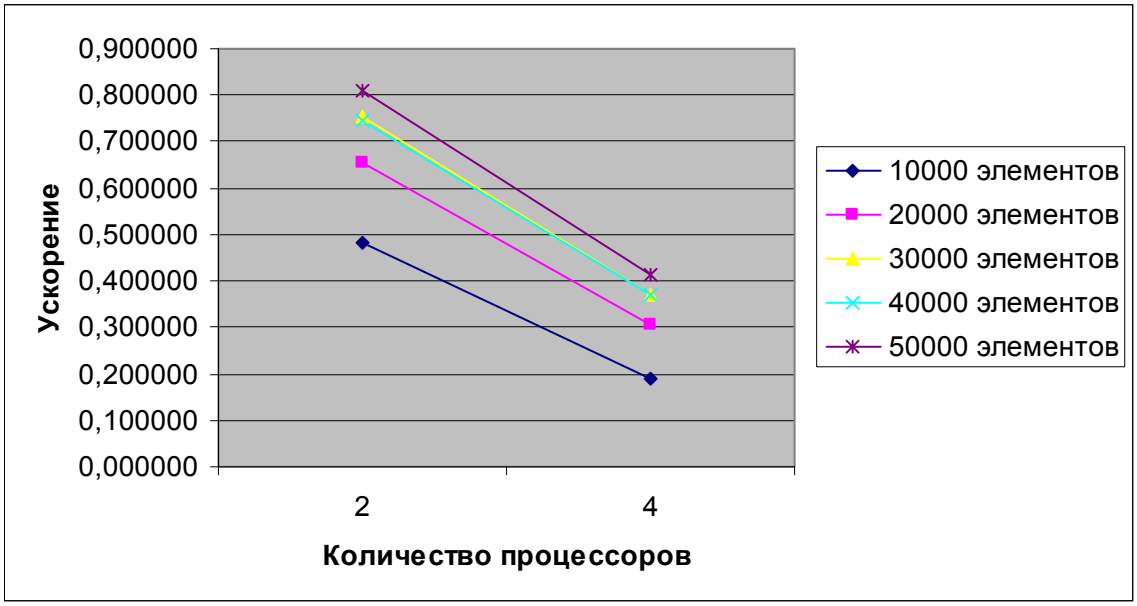

Рис. 10.4. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма сортировки Шелла

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.7) приведено в таблице 10.5 и на рис. 10.5.

Таблица 10.5. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сортировки Шелла

Размер данных	Параллельный алгоритм			
	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,002684	0,002959	0,002938	0,007509
20 000	0,004872	0,004557	0,004729	0,009826
30 000	0,007100	0,006118	0,006538	0,012431
40 000	0,009353	0,008461	0,008361	0,017009
50 000	0,011625	0,009920	0,010193	0,019419

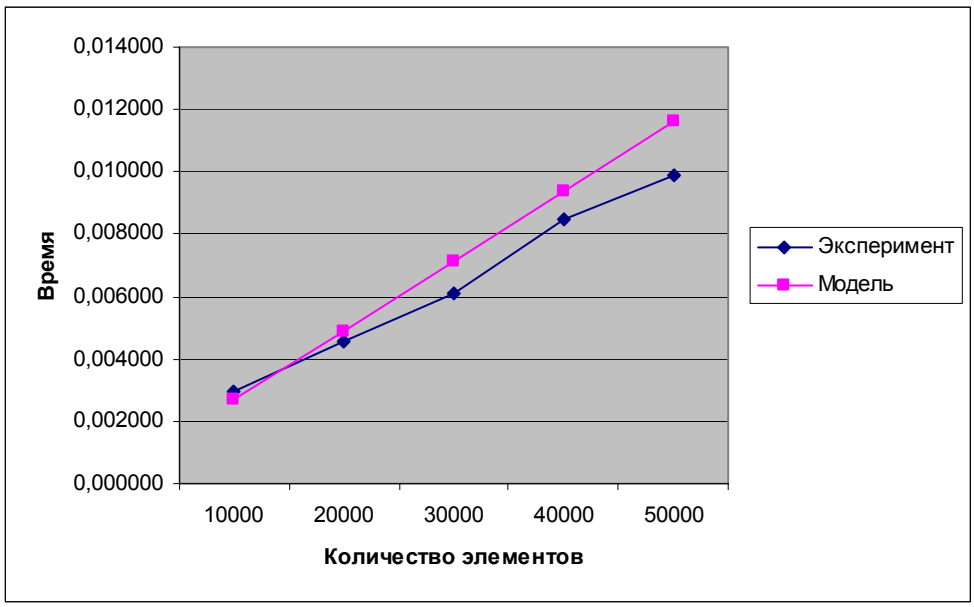

Рис. 10.5. График зависимости от объема исходных данных экспериментального и теоретического времени проведения эксперимента на двух процессорах

10.5. Быстрая сортировка

10.5.1. Последовательный алгоритм

При общем рассмотрении *алгоритма быстрой сортировки*, предложенной Хоаром (*Hoare C.A.R.*), прежде всего следует отметить, что этот метод основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности (для любой пары блоков все значения одного из этих блоков не превышают значений другого блока). На первой итерации метода осуществляется деление исходного набора данных на первые две части – для организации такого деления выбирается некоторый *ведущий элемент* и все значения набора, меньшие ведущего элемента, переносятся в первый формируемый блок, все остальные значения образуют второй блок набора. На второй итерации сортировки описанные правила применяются рекурсивно для обоих сформированных блоков и т.д. При надлежащем выборе ведущих элементов после выполнения $log_2 n$ итераций исходный массив данных оказывается упорядоченным. Более подробное изложение метода может быть получено, например, в Кнут (1981), Кормен, Лейзерсон и Ривест (1999).

Эффективность быстрой сортировки в значительной степени определяется правильностью выбора ведущих элементов при формировании блоков. В худшем случае трудоемкость метода имеет тот же порядок сложности, что и пузырьковая сортировка (т.е. $T_1 \sim n^2$). При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки ($T_1 \sim n \log_2 n$). В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением (см., например, Кнут (1981), Кормен, Лейзерсон и Ривест (1999)):

$T_1 = 1.4 n \log_2 n$.

Общая схема алгоритма быстрой сортировки может быть представлена в следующем виде (в качестве ведущего элемента выбирается первый элемент упорядочиваемого набора данных).

```
// Алгоритм 10.5
// Последовательный алгоритм быстрой сортировки
QuickSort(double A[], int i1, int i2) {
  if ( i1 < i2 ){
    double pivot = A[i1];
    int is = i1;
    for ( int i = i1+1; i<i2; i++ )
      if ( A[i] ≤ pivot ) {
        is = is + 1;
        swap(A[is], A[i]);
      }
```

```
    swap(A[i1], A[is]);
    QuickSort (A, i1, is);
    QuickSort (A, is+1, i2);
  }
}
```

Алгоритм 10.5. Последовательный алгоритм быстрой сортировки

10.5.2. Параллельный алгоритм быстрой сортировки

10.5.2.1. Организация параллельных вычислений

Параллельное обобщение алгоритма быстрой сортировки (см., например, Quinn (2003)) наиболее простым способом может быть получено для вычислительной системы с топологией в виде N-мерного гиперкуба (т.е. $p=2^N$). Пусть, как и ранее, исходный набор данных распределен между процессорами блоками 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 таких пересылок данных на процессорах, для которых в битовом представлении номера бит позиции N равен 0, должны оказаться части блоков со значениями, меньшими ведущего элемента; процессоры с номерами, в которых бит N равен 1, должны собрать, соответственно, все значения данных, превышающие значение ведущего элемента.

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 также оказывается разделенным на два гиперкуба размерности $N-1$. К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После N -кратного повторения подобных итераций для завершения сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре вычислительной системы.

Для пояснения на рис.10.6 представлен пример упорядочивания данных при $n=16$, $p=4$ (т.е. блок каждого процессора содержит 4 элемента). На этом рисунке процессоры изображены в виде прямоугольников, внутри которых показано содержимое упорядочиваемых блоков данных; значения блоков приводятся в начале и при завершении 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 4, для пары процессоров 2, 3 это значение было принято равным –5.

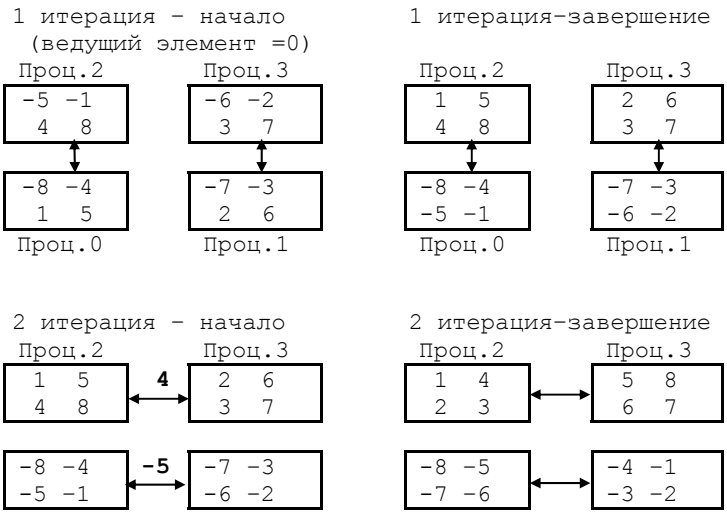

Рис. 10.6. Пример упорядочивания данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

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.

10.5.2.2. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Пусть у нас имеется N -мерный гиперкуб состоящий из $p = 2^N$ процессоров, где $p < n$.

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 процессорами вычислительной системы.

Определим вначале вычислительную сложность алгоритма сортировки. На каждой из $log_2 p$ итераций 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 процессоры 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

$$T_p(calc) = [(n/p)\log_2 p + (n/p)\log_2(n/p)]\tau, \qquad (10.8)$$

где τ есть время выполнения базовой операции перестановки.

Рассмотрим теперь сложность выполняемых коммуникационных операций. Общее количество межпроцессорных обменов для рассылки ведущего элемента на N-мерном гиперкубе может быть ограничено оценкой

$$\sum_{i=1}^{N} i = N(N+1)/2 = \log_2 p\,(\log_2 p + 1)/2 \sim (\log_2 p)^2. \qquad (10.9)$$

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

$$T_p(comm) = (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta), \qquad (10.10)$$

где α - латентность, β - пропускная способность сети, а w есть размер элемента набора в байтах.

С учетом всех полученных соотношений общая трудоемкость алгоритма оказывается равной

$$T_p = [(n/p)\log_2 p + (n/p)\log_2(n/p)]\tau + (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta). \qquad (10.11)$$

10.5.2.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта быстрой сортировки проводились при тех же условиях, что и ранее выполненные эксперименты (см. п. 10.3.4).

Результаты вычислительных экспериментов приведены в табл. 10.6. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 10.6. Результаты вычислительных экспериментов по исследованию параллельного алгоритма быстрой сортировки

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,001521	0,934911	0,003434	0,414094
20 000	0,002991	0,002234	1,338854	0,004094	0,730581

30 000	0,004612	0,003080	1,497403	0,005088	0,906447
40 000	0,006297	0,004363	1,443273	0,005906	1,066204
50 000	0,008014	0,005486	1,460809	0,006635	1,207837

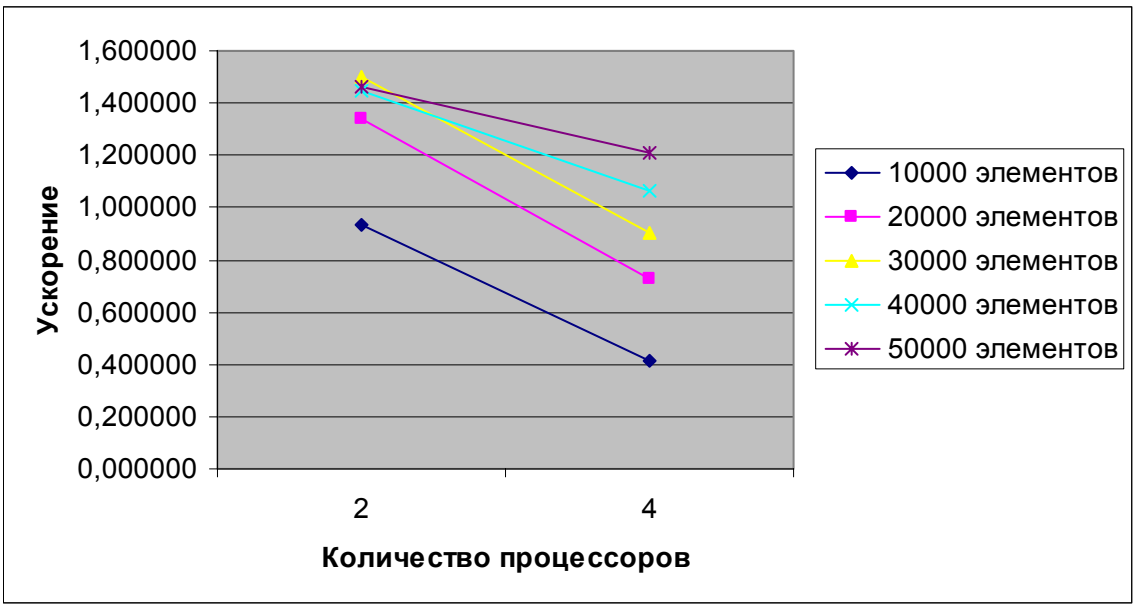

Рис. 10.7. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма быстрой сортировки

Как можно заметить по результатам вычислительных экспериментов, параллельный алгоритм быстрой сортировки уже позволяет получить ускорение при решении задачи упорядочивания данных.

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.11) приведено в таблице 10.7 и на рис. 10.8.

Таблица 10.7. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма быстрой сортировки

Размер данных	Параллельный алгоритм			
	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,001280	0,001521	0,001735	0,003434
20 000	0,002265	0,002234	0,002321	0,004094
30 000	0,003289	0,003080	0,002928	0,005088
40 000	0,004338	0,004363	0,003547	0,005906
50 000	0,005407	0,005486	0,004175	0,006635

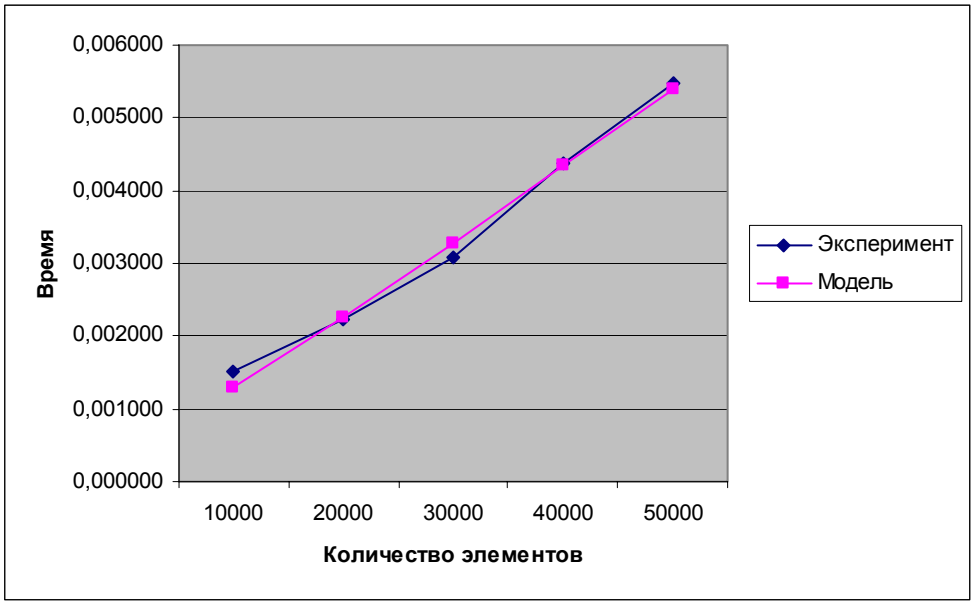

Рис. 10.8. График зависимости экспериментального и теоретического времени проведения эксперимента на двух процессорах от объема исходных данных

10.5.3. Обобщенный алгоритм быстрой сортировки

В *обобщенном алгоритме быстрой сортировки* (*HyperQuickSort algorithm*) в дополнение к обычному методу быстрой сортировки предлагается конкретный способ выбора ведущих элементов. Суть предложения состоит в том, что сортировка располагаемых на процессорах блоков выполняется в самом начале выполнения вычислений. Кроме того, для поддержки упорядоченности в ходе вычислений процессоры должны выполнять операция слияния частей блоков, получаемых после разделения. Как результат, в силу упорядоченности блоков, при выполнении алгоритма быстрой сортировки в качестве ведущего элемента целесообразнее будет выбирать средний элемент какого-либо блока (например, на первом процессоре вычислительной системы). Выбираемый подобным образом ведущий элемент в отдельных случаях может оказаться более близок к реальному среднему значению всего сортируемого набора, чем какое-либо другое произвольно выбранное значение.

Все остальные действия в новом рассматриваемом алгоритме выполняются в соответствии с обычным методом быстрой сортировки. Более подробное описание данного способа распараллеливания быстрой сортировки может быть получено, например, в Quinn (2003).

При анализе эффективности обобщенного алгоритма можно воспользоваться соотношением (10.11). Следует только учесть, что на каждой итерации метода теперь выполняется операция слияния частей блоков (будем, как и ранее, предполагать, что их размер одинаков и равен (*n/p)/2*). Кроме того, в силу упорядоченности блоков может быть усовершенствована процедура деления – вместо перебора всех элементов блока теперь достаточно будет выполнить для ведущего элемента бинарный поиск в блоке. С учетом всех высказанных замечаний трудоемкость обобщенного алгоритма быстрой сортировки может быть выражена при помощи выражения следующего вида:

$$T_p = [(n/p)\log_2(n/p) + (\log_2(n/p) + (n/p))\log_2 p]\tau + (\log_2 p)^2(\alpha + w/\beta) + \log_2 p\,(\alpha + w(n/2p)/\beta). \quad (10.12)$$

10.5.3.1. Программная реализация

Представим возможный вариант параллельной программы обобщенной быстрой сортировки. При этом реализация отдельных модулей не приводится, если их отсутствие не оказывает влияния на понимание общей схемы параллельных вычислений.

1. Главная функция программы. Реализует логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 10.1.
// Обобщенная быстрая сортировка
  int ProcRank;       // Rank of current process
  int ProcNum;        // Number of processes
int main(int argc, char *argv[]) {
  double *pProcData;     // Блок данных для процесса
```

```
  int     ProcDataSize;  // Размер блока данных

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);

  // Инициализация данных и их распределение между процессами
  ProcessInitialization ( &pProcData, &ProcDataSize);

  // параллельная сортировка
  ParallelHyperQuickSort ( pProcData, ProcDataSize );

  // Завершение вычислений процесса
  ProcessTermination ( pProcData, ProcDataSize );

  MPI_Finalize();
}
```

Функция *ProcessInitialization* определяет исходные данные решаемой задачи (размер сортируемого массива), выделяет память для хранения данных, осуществляет генерацию сортируемого массива (например, при помощи датчика случайных чисел) и распределяет его между процессами.

Функция *ProcessTermination* выполняет необходимый вывод результатов решения задачи и освобождает всю ранее выделенную память для хранения данных.

Реализация всех перечисленных функций может быть выполнена по аналогии с ранее рассмотренными примерами и предоставляется читателю в качестве самостоятельного упражнения.

2. Функция ParallelHyperQuickSort. Функция выполняет параллельную быструю сортировку согласно рассмотренному алгоритму.

```
void ParallelHyperQuickSort ( double *pProcData, int ProcDataSize) {
  MPI_Status status;
  int CommProcRank; // ранг процессора, с которым выполняется взаимодействие
  double *pData,      // часть блока, остающаяся на процессоре
         *pSendData,  // часть блока, передаваемая процессору CommProcRank
         *pRecvData,  // часть блока, получаемая от процессора CommProcRank
         *pMergeData; // блок данных, получаемый после слияния
  int     DataSize, SendDataSize, RecvDataSize, MergeDataSize;
  int HypercubeDim = (int)ceil(log(ProcNum)/log(2)); // размерность гиперкуба
  int Mask = ProcNum;
  double Pivot;

  // первоначальная сортировка блоков данных на каждом процессоре
  LocalDataSort ( pProcData, ProcDataSize );

  // итерации обобщенной быстрой сортировки
  for ( int i = HypercubeDim; i > 0; i-- ) {

    // определение ведущего значения и его рассылка всем процессорам
    PivotDistribution (pProcData,ProcDataSize,HypercubeDim,Mask,i,&Pivot);
    Mask = Mask >> 1;

    // определение границы разделения блока
    int pos = GetProcDataDivisionPos (pProcData, ProcDataSize, Pivot);

    // разделение блока на части
    if ( ( (rank&Mask) >> (i-1) ) == 0 ) { // старший бит = 0
      pSendData = & pProcData[pos+1];
      SendDataSize = ProcDataSize - pos - 1;
      if ( SendDataSize < 0 ) SendDataSize = 0;
      CommProcRank = ProcRank + Mask
      pData = & pProcData[0];
      DataSize = pos + 1;
    }
    else { // старший бит = 1
      pSendData = & pProcData[0];
```

```
      SendDataSize = pos + 1;
      if ( SendDataSize > ProcDataSize ) SendDataSize = pos;
      CommProcRank = ProcRank – Mask
      pData = & pProcData[pos+1];
      DataSize = ProcDataSize - pos - 1;
      if ( DataSize < 0 ) DataSize = 0;
    }
    // пересылка размеров частей блоков данных
    MPI_Sendrecv(&SendDataSize, 1, MPI_INT, CommProcRank, 0,
      &RecvDataSize, 1, MPI_INT, CommProcRank, 0, MPI_COMM_WORLD, &status);

    // пересылка частей блоков данных
    pRecvData = new double[RecvDataSize];
    MPI_Sendrecv(pSendData, SendDataSize, MPI_DOUBLE,
      CommProcRank, 0, pRecvData, RecvDataSize, MPI_DOUBLE,
      CommProcRank, 0, MPI_COMM_WORLD, &status);

    // слияние частей
    MergeDataSize = DataSize + RecvDataSize;
    pMergeData = new double[MergeDataSize];
    DataMerge(pMergeData, pMergeData, pData, DataSize,
              pRecvData, RecvDataSize);
    delete [] pProcData;
    delete [] pRecvData;
    pProcData = pMergeData;
    ProcDataSize = MergeDataSize;
  }
}
```

Функция *LocalDataSort* выполняет сортировку блока данных на каждом процессоре, используя последовательный алгоритм быстрой сортировки.

Функция *PivotDistribution* определяет ведущий элемент и рассылает его значение всем процессорам.

Функция *GetProcDataDivisionPos* выполняет разделение блока данных относительно ведущего элемента. Ее результатом является целое число, обозначающее позицию элемента на границе двух блоков.

Функция *DataMerge* выполняет слияние частей в один упорядоченный блок данных.

3. Функция PivotDistribution. Функция выбирает ведущий элемент и рассылает его все процессорам гиперкуба. Так как данные на процессорах отсортированы с самого начала, ведущий элемент выбирается как средний элемент блока данных.

```
void PivotDistribution (double *pProcData, int ProcDataSize, int Dim,
  int Mask, int Iter, double *pPivot) {
  MPI_Group WorldGroup;
  MPI_Group SubcubeGroup; // группа процессов - подгиперкуб
  MPI_Comm  SubcubeComm;  // коммуникатор подгиперкуба
  int j = 0;

  int  GroupNum = ProcNum /(int)pow(2, Dim-Iter);
  int *ProcRanks = new int [GroupNum];

  // формирование списка рангов процессов для гиперкуба
  int StartProc = ProcRank – GroupNum;
  if (StartProc < 0 ) StartProc = 0;
  int EndProc = (ProcRank + GroupNum;
  if (EndProc > ProcNum ) EndProc = ProcNum;
  for (int proc = StartProc; proc < EndProc; proc++) {
    if ((ProcRank & Mask)>>(Iter) == (proc & Mask)>>(Iter)) {
      ProcRanks[j++] = proc;
    }
  }
  //объединение процессов подгиперкуба в одну группу
  MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
  MPI_Group_incl(WorldGroup, GroupNum, ProcRanks, &SubcubeGroup);
  MPI_Comm_create(MPI_COMM_WORLD, SubcubeGroup, &SubcubeComm);
```

```
  // поиск и рассылка ведущего элемента всем процессам подгиперкуба
  if (ProcRank == ProcRanks[0])
    *pPivot = pProcData[(ProcDataSize)/2];

  MPI_Bcast ( pPivot, 1, MPI_DOUBLE, 0, SubcubeComm );
  MPI_Group_free(&SubcubeGroup);
  MPI_Comm_free(&SubcubeComm);
  delete [] ProcRanks;
}
```

10.5.3.2. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта обобщенной быстрой сортировки проводились при тех же условиях, что и ранее выполненные эксперименты (см. п. 10.3.4).

Результаты вычислительных экспериментов приведены в табл. 10.8. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 10.8. Результаты вычислительных экспериментов по исследованию параллельного алгоритма обобщенной быстрой сортировки

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10 000	0,001422	0,001485	0,957576	0,002898	0,490683
20 000	0,002991	0,002180	1,372018	0,003770	0,793369
30 000	0,004612	0,003077	1,498863	0,004451	1,036172
40 000	0,006297	0,003859	1,631770	0,004721	1,333828
50 000	0,008014	0,005041	1,589764	0,005242	1,528806

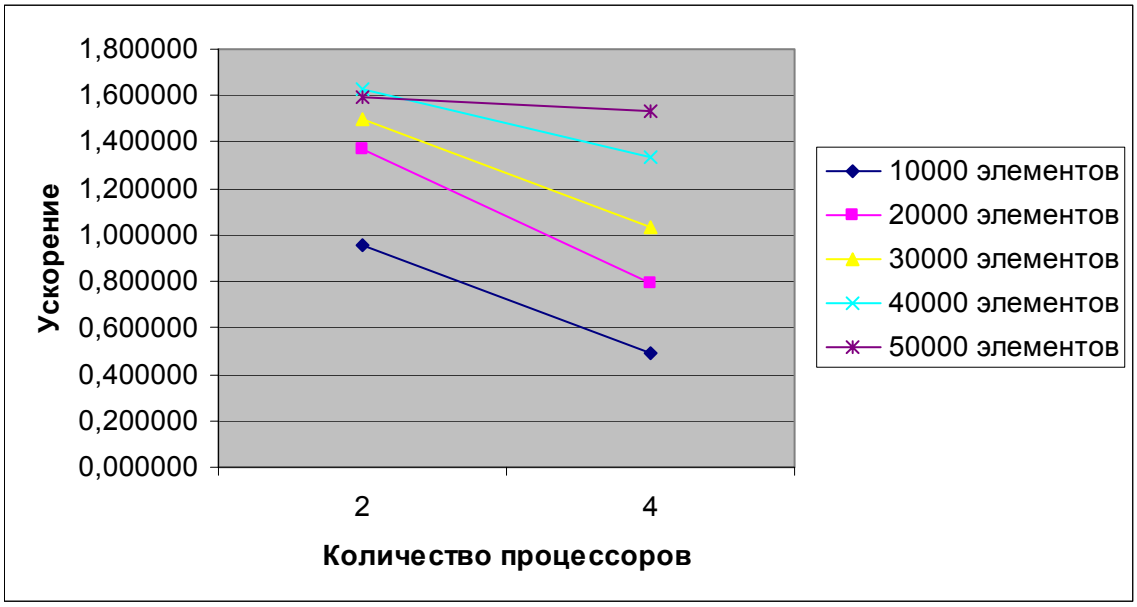

Рис. 10.9. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма обощенной быстрой сортировки

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.12) приведено в таблице 10.9 и на рис. 10. 10.

Таблица 10.9. Сравнение экспериментального и теоретического времен выполнения параллельного алгоритма обобщенной быстрой сортировки

Размер данных	Параллельный алгоритм	
	2 процессора	4 процессора

	T_2	T_2^*	T_4	T_4^*
10 000	0,001281	0,001485	0,001735	0,002898
20 000	0,002265	0,002180	0,002322	0,003770
30 000	0,003289	0,003077	0,002928	0,004451
40 000	0,004338	0,003859	0,003547	0,004721
50 000	0,005407	0,005041	0,004176	0,005242

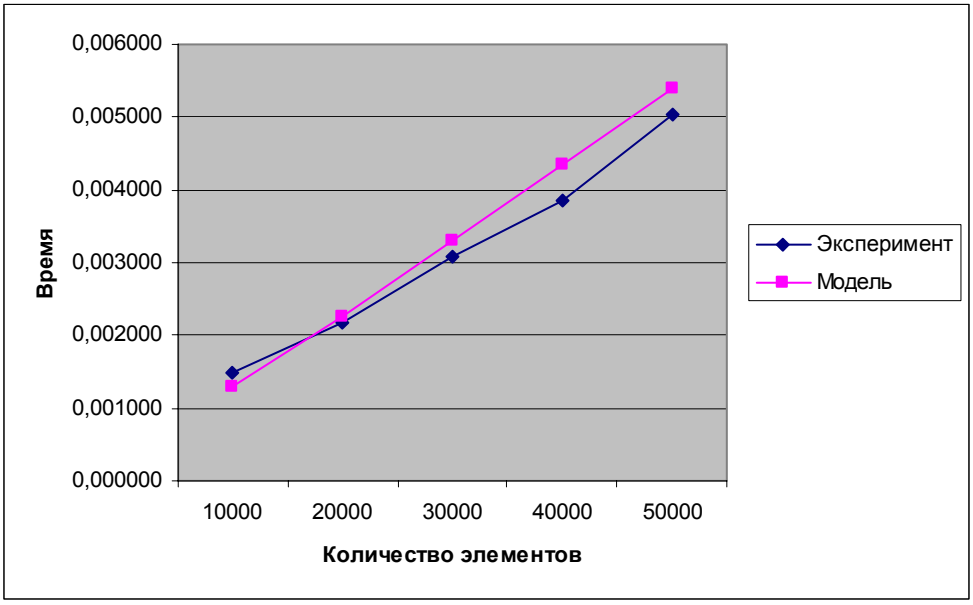

Рис. 10.10. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных

10.5.4. Сортировка с использованием регулярного набора образцов

10.5.4.1. Организация параллельных вычислений

Алгоритм сортировки с использованием регулярного набора образцов (*Parallel Sorting by regular sampling*) также является обобщением метода быстрой сортировки (см., например, в Quinn (2003)).

Упорядочивание данных в соответствии с данным вариантом алгоритма быстрой сортировки осуществляется в ходе выполнения следующих четырех этапов:

- на *первом этапе* сортировки производится упорядочивание имеющихся на процессорах блоков; данная операция может быть выполнена каждым процессором независимо друг от друга при помощи обычного алгоритма быстрой сортировки; далее каждый процессор формирует набор из элементов своих блоков с индексами $0,\ m,\ 2m, \ldots, (p-1)m$, где $m = n/p^2$;
- на *втором этапе* выполнения алгоритма все сформированные на процессорах наборы данных собираются на одном из процессоров системы и объединяются в ходе последовательного сливания в одно упорядоченное множество; далее из полученного множества значений из элементов с индексами

$$p + \lfloor p/2 \rfloor - 1,\quad 2p + \lfloor p/2 \rfloor - 1, \ldots, (p-1)p + \lfloor p/2 \rfloor$$

формируется новый набор ведущих элементов, который передается всем используемым процессорам; в завершение этапа каждый процессор выполняет разделение своего блока на p частей с использованием полученного набора ведущих значений;

- на *третьем этапе* сортировки каждый процессор осуществляет рассылку выделенных ранее частей своего блока всем остальным процессорам системы; рассылка выполняется в соответствии с порядком нумерации - часть j, $0 \le j < p$, каждого блока пересылается процессору с номером j;
- на *четвертом этапе* выполнения алгоритма каждый процессор выполняет слияние p полученных частей в один отсортированный блок.

По завершении четвертого этапа исходный набор данных становится отсортированным.

На рис.10.11. приведен пример сортировки массива данных с помощью алгоритма, описанного выше. Следует отметить, что число процессоров для данного алгоритма может быть произвольным, в данном примере оно равно 3.

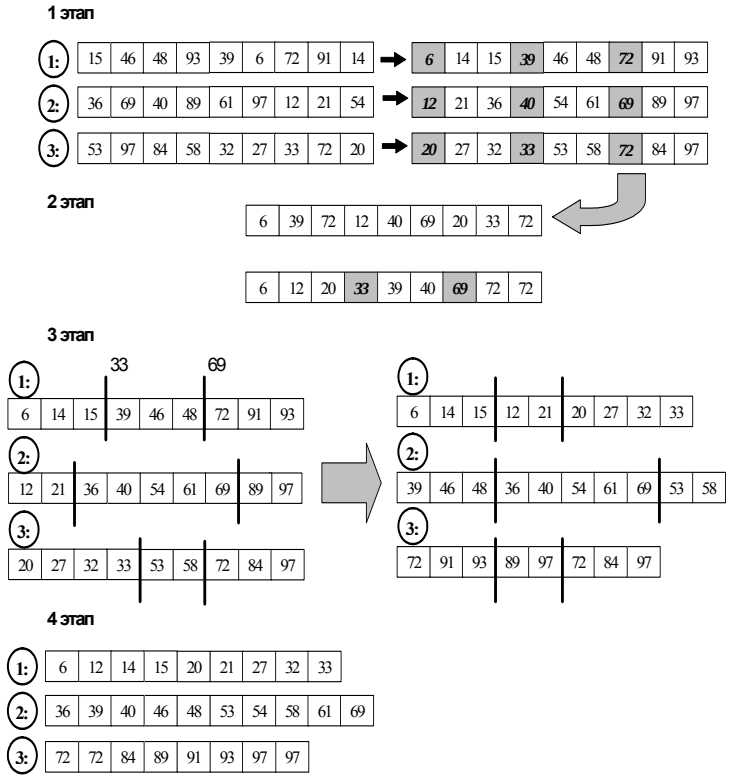

Рис. 10.11. Пример работы алгоритма сортировки с использованием регулярного набора образцов для 3 процессоров

10.5.4.2. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Пусть, как и ранее, n есть количество сортируемых данных, p, $p<n$, обозначает число используемых процессоров и, соответственно, n/p есть размер блоков данных на процессорах.

В течение первого этапа алгоритма каждый процессор сортирует свой блок данных с помощью быстрой сортировки, тем самым, длительность выполняемых при этом операций является равной

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

где τ есть время выполнения базовой операции сортировки.

На втором этапе алгоритма один из процессоров собирает наборы из p элементов со всех остальных процессоров, выполняет слияние всех полученных данных (общее количество элементов составляет p^2), формирует набор из $p-1$ ведущих элементов и рассылает полученный набор всем остальным процессорам. С учетом всех перечисленных действий общая длительность второго этапа составляет

$$T_p^2 = [\alpha \log_2 p + wp(p-1)/\beta] + [p^2 \log_2 p\,\tau] + [p\,\tau] + [\log_2 p\,(\alpha + wp/\beta)]$$

(в приведенном соотношении выделенные подвыражения соответствуют четырем перечисленным действиям алгоритма); здесь, как и ранее, α – латентность, β – пропускная способность сети передачи данных, а w есть размер элемента упорядочиваемых данных в байтах.

В ходе выполнения третьего этапа алгоритма каждый процессор разделяет свои элементы относительно ведущих элементов на p частей (общее количество операций для этого может быть ограничено величиной n/p). Далее все процессоры выполняют рассылку сформированных частей блоков между собой – оценка трудоемкости такой коммуникационной операции рассмотрена в разделе 3 при представлении топологии вычислительной сети в виде гиперкуба. Как было показано, выполнение такой операции может быть осуществлено за $log_2 p$ шагов, на каждом из которых каждый процессор передает и получает сообщение из $(n/p)/2$ элементов. Как результат, общая трудоемкость третьего этапа алгоритма может быть оценена как

$$T_p^3 = (n/p)\tau + \log_2 p(\alpha + w(n/2p)/\beta)\,.$$

На четвертом этапе алгоритма каждый процессор выполняет слияние p отсортированных частей в один объединенный блок. Оценка трудоемкости такой операции уже проводилась при рассмотрении второго этапа и, тем самым, длительность выполнения процедуры слияния составляет

$$T_p^4 = (n/p)\log_2 p\tau \ .$$

С учетом всех полученных соотношений, общее время выполнения алгоритма сортировки с использованием регулярного набора образцов составляет

$$\begin{aligned} T_p \ = \ & (n/p)\log_2(n/p)\ \tau + (\alpha\ \log_2\ p + wp(p-1)/\beta) + p^2\ \log_2\ p\tau + (n/p)\tau + \\ & + \log_2\ p(\alpha + wp/\beta) + p\tau + \log_2\ p(\alpha + w(n/2p)/\beta) + (n/p)\log_2\ p\tau \end{aligned} \ . \qquad (10.13)$$

10.5.4.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта сортировки с использованием регулярного набора образцов проводились при тех же условиях, что и ранее выполненные эксперименты (см. п. 10.3.4).

Результаты вычислительных экспериментов приведены в табл. 10.10. Эксперименты проводились с использованием двух и четырех процессоров. Время указано в секундах.

Таблица 10.10. Результаты вычислительных экспериментов по исследованию параллельного алгоритма сортировки с использованием регулярного набора образцов

Количество элементов	Последовательный алгоритм	Параллельный алгоритм			
		2 процессора		4 процессора	
		Время	Ускорение	Время	Ускорение
10,000	0,001422	0,001513	0,939855	0,001166	1,219554
20,000	0,002991	0,002307	1,296489	0,002081	1,437290
30,000	0,004612	0,003168	1,455808	0,003099	1,488222
40,000	0,006297	0,004542	1,386394	0,003819	1,648861
50,000	0,008014	0,005503	1,456297	0,004370	1,833867

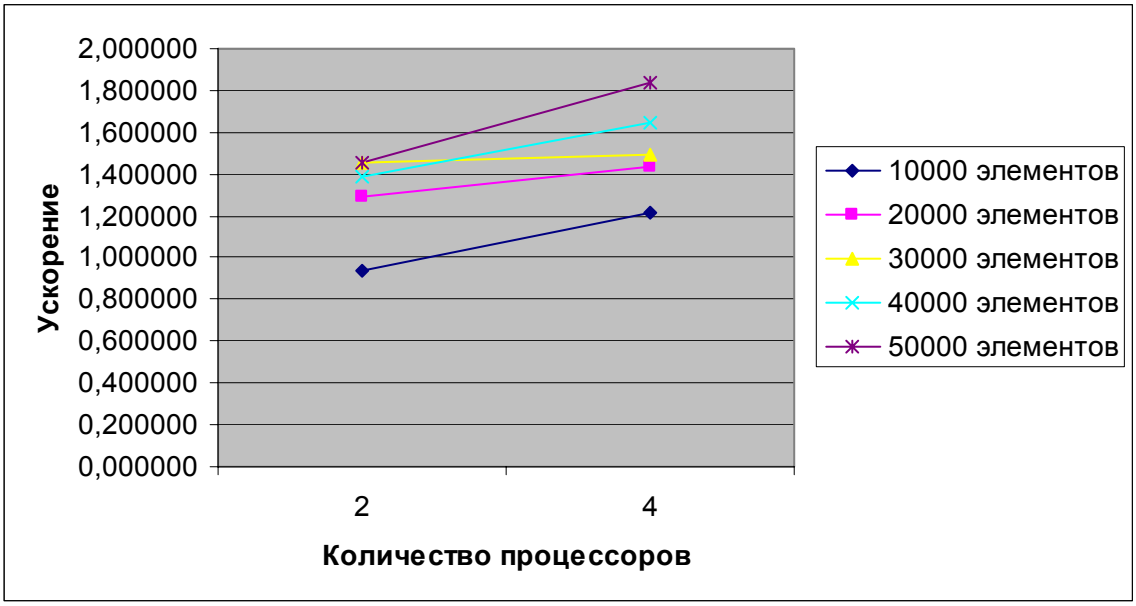

Рис. 10.12. Зависимость ускорения от количества процессоров при выполнении параллельного алгоритма сортировки с использованием регулярного набора образцов

Сравнение времени выполнения эксперимента T_p^* и теоретической оценки T_p из (10.13) приведено в таблице 10.11 и на рис. 10.13.

Таблица 10.11. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма сортировки с использованием регулярного набора образцов

Размер	Параллельный алгоритм

данных	2 процессора		4 процессора	
	T_2	T_2^*	T_4	T_4^*
10 000	0,001533	0,001513	0,001762	0,001166
20 000	0,002569	0,002307	0,002375	0,002081
30 000	0,003645	0,003168	0,003007	0,003099
40 000	0,004747	0,004542	0,003652	0,003819
50 000	0,005867	0,005503	0,004307	0,004370

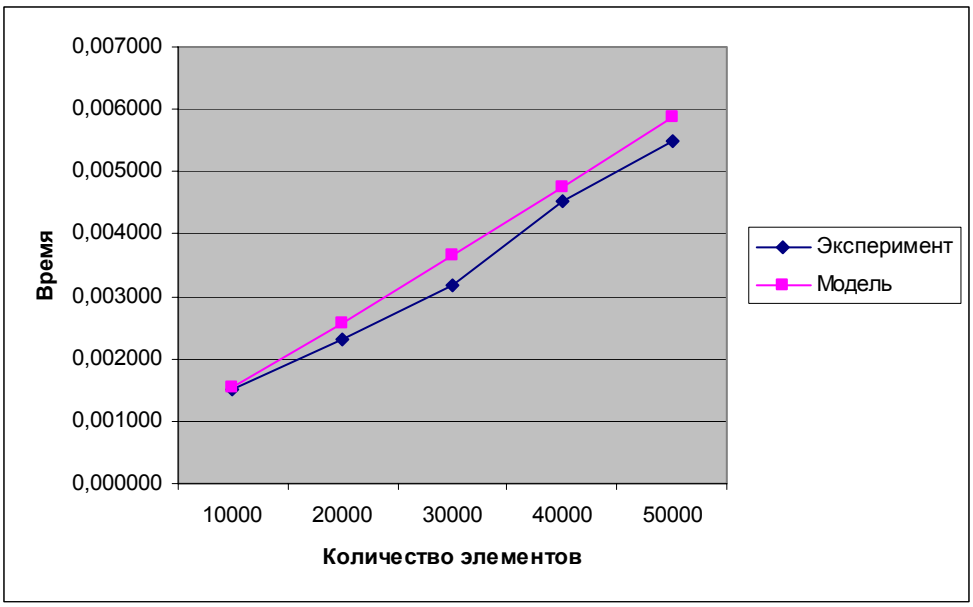

Рис. 10.13. График зависимости экспериментального и теоретического времени проведения эксперимента на четырех процессорах от объема исходных данных.

10.6. Краткий обзор раздела

В разделе рассматривается часто встречающаяся в приложениях *задача упорядочения данных*, для решения которой в рамках данного учебного материала выбраны широко известные алгоритмы пузырьковой сортировки, сортировки Шелла и быстрой сортировки. При изложении методов сортировки основное внимание уделяется возможным способам распараллеливания алгоритмов, анализу эффективности и сравнению получаемых теоретических оценок с результатами выполненных вычислительных экспериментов.

Алгоритм пузырьковой сортировки (подраздел 10.3) в исходном виде практически не поддается распараллеливанию в силу последовательного выполнения основных итераций метода. Для введения необходимого параллелизма рассматривается обобщенный вариант алгоритма - *метод чет-нечетной перестановки*. Суть обобщения состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода в зависимости от четности номера итерации сортировки. Сравнения пар значений упорядочиваемого набора данных на итерациях метода чет-нечетной перестановки являются независимыми и могут быть выполнены параллельно.

Для *алгоритма Шелла* (подраздел 10.4) рассматривается схема распараллеливания при представлении топологии сети в виде гиперкуба. При таком представлении топологии оказывается возможным организация взаимодействия процессоров, расположенных далеко друг от друга при линейной нумерации. Как правило, такая организация вычислений позволяет уменьшить количество выполняемых итераций алгоритма сортировки.

Для *алгоритма быстрой сортировки* (подраздел 10.5) приводятся три схемы распараллеливания. Первые две схемы также основываются на представлении топологии сети в виде гиперкуба. Основная итерация вычислений состоит в выборе одним из процессоров ведущего элемента, который далее рассылается всем остальным процессорам гиперкуба. После получения ведущего элемента процессоры проводят разделение своих блоков, и получаемые части блоков передаются между попарно связанными процессорами. В результате выполнения подобной итерации исходный гиперкуб оказывается разделенным на 2 гиперкуба меньшей размерности, к которым, в свою очередь, может быть применена приведенная выше схема вычислений.

При применении алгоритма быстрой сортировки одним из основных моментов является правильность выбора ведущего элемента. Оптимальная ситуация состоит в выборе такого значения ведущего элемента, при котором данные на процессорах разделяются на части одинакового размера. В общем случае, при произвольно сгенерированных исходных данных достижение такой ситуации является достаточно сложной задачей. В первой схеме предлагается выбирать ведущий элемент, например, как среднее арифметическое элементов на управляющем процессоре. Во второй схеме данные на каждом процессоре предварительно упорядочиваются с тем, чтобы взять средний элемент блока данных как ведущее значение.

Третья схема распараллеливания алгоритма быстрой сортировки основывается на многоуровневой схеме формирования множества ведущих элементов. Такой подход может быть применен для произвольного количества процессоров, позволяет избежать множества пересылок данных и приводит, как правило, к лучшей балансировке распределения данных между процессорами.

10.7. Обзор литературы

Возможные способы решения задачи упорядочения данных широко обсуждаются в литературе; один из наиболее полных обзоров *алгоритмов сортировки* содержится в работе Кнута (1981), среди последних изданий может быть рекомендована работа Кормена, Лейзерсона и Ривеста (1999).

Параллельные варианты *алгоритма пузырьковой сортировки* и *сортировки Шелла* рассматриваются в Kumar (1994).

Схемы распараллеливания *быстрой сортировки* при представлении топологии сети передачи данных в виде гиперкуба описаны в Kumar (1994) и Quinn (2003). *Сортировка с использованием регулярного набора образцов* (*parallel sorting by regular sampling*) представлена в работе Quinn (2003).

Полезной при рассмотрении вопросов параллельных вычислений для сортировки данных может оказаться работа Akl (1985).

10.8. 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?
6. В чем состоит параллельный вариант алгоритма Шелла? Какие основные отличия этого варианта параллельного алгоритма сортировки от метода чет-нечетной перестановки?
7. Что представляет собой параллельный вариант алгоритма быстрой сортировки?
8. Что зависит от правильного выбора ведущего элемента для параллельного алгоритма быстрой сортировки?
9. Какие способы выбора ведущего элемента могут быть предложены?
10. Для каких топологий могут применяться рассмотренные алгоритмы сортировки?
11. В чем состоит алгоритм сортировки с использованием регулярного набора образцов?

10.9. Задачи и упражнения

1. Выполните реализацию параллельного алгоритма пузырьковой сортировки. Проведите эксперименты. Постройте теоретические оценки с учетом тех операций пересылок данных, которые использовались при реализации, и параметров вычислительной системы. Сравните получаемые теоретические оценки с результатами экспериментов.

2. Выполните реализацию параллельного алгоритма быстрой сортировки по одной из приведенных схем. Определите значения параметров латентности, пропускной способности и времени выполнения базовой операции для используемой вычислительной системы и получите оценки показателей ускорения и эффективности для реализованного метода параллельных вычислений.

3. Разработайте параллельную схему вычислений для широко известного алгоритма сортировки слиянием (подробное описание метода может быть получено, например, в работах Кнута (1981) или Кормена, Лейзерсона и Ривеста (1999)). Выполните реализацию разработанного алгоритма и постройте все необходимые теоретические оценки сложности метода.

Литература

Knuth, D. E. (1997). The Art of Computer Programming. Volume 3: Sorting and Searching, second edition. – Reading, MA: Addison-Wesley. (русский перевод Кнут Д. (2000). Искусство программирования для ЭВМ. Т. 3. Сортировка и поиск. 2 издание - М.: Издательский дом "Вильямс")