Нижегородский государственный университет им. Н.И.Лобачевского

Межфакультетская магистратура по системному и прикладному программированию для многоядерных компьютерных систем

Учебный курс "Введение в методы параллельного программирования"

Раздел "Параллельные методы сортировки данных"

Разработчик: А.А.Лабутина

Нижний Новгород
2007

Содержание

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 ($p > 1$) *вычислительных элементов* (процессоров или ядер). Исходный упорядочиваемый набор в этом случае «разделяется» на блоки; которые могут обрабатываться вычислительными элементами параллельно.

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. Основы сортировки и 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.

Рассмотренная выше базовая операция сортировки может быть надлежащим образом обобщена для случая, когда упорядочиваемые данные разделены на p, $p>1$, частей (*блоков*). Выделяемые при этом блоки имеют, как правило, одинаковый размер, и содержат в этом случае n/p элементов.

Блоки обычно упорядочиваются в самом начале сортировки - как можно заметить, блоки могут упорядочиваться независимо друг от друга, т.е. параллельно. Далее, следуя схеме одноэлементного сравнения, упорядочение содержимого блоков A_i и A_{i+1} может быть осуществлено следующим образом:

- объединить блоки A_i и A_{i+1} 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:

$[A_i \cup A_{i+1}]_{copm} = A_i^{'} \cup A_{i+1}^{'} : \forall a_i^{'} \in A_i^{'}, \forall a_j^{'} \in A_{i+1}^{'} \Rightarrow a_i^{'} \le a_j^{'}$.

Рассмотренная процедура обычно именуется в литературе как *операция "сравнить и разделить"* (*compare-split*). Следует отметить, что сформированные в результате такой процедуры блоки совпадают по размеру с исходными блоками A_i и A_{i+1}, являются упорядоченными и все значения, расположенные в блоке A'_i, не превышают значений в блоке A'_{i+1}.

Трудоемкость рассмотренной операции при использовании быстрых алгоритмов сортировки является равной:

$$T' = 2(n/p)\log_2(n/p) + 2(n/p),$$

в то время как обычное упорядочивание данных, располагаемых в двух блоках, требует выполнения

$$T'' = (2n/p)\log_2(2n/p)$$

операций. Приведенные оценки показывают, что вычислительная сложность операции "сравнить и разделить" при прочих равных условий является меньшей. Кроме того, "блочность" данной операции может привести к более эффективному использованию кэш-памяти процессоров, что позволит еще больше повысить эффективность выполнения алгоритмов сортировки.

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 блоков 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 10.4) объем блоков может различаться, что может приводить к нарушению равномерной вычислительной загрузки вычислительных элементов.

10.2. Пузырьковая сортировка

Алгоритм пузырьковой сортировки (см., например, Кнут (1981), Кормен, Лейзерсон и Ривест (1999)) является одним из наиболее широко известных методов упорядочивания данных, однако в силу своей низкой эффективности в основном используется только в учебных целях.

10.2.1. Последовательный алгоритм пузырьковой сортировки

10.2.1.1. Общая схема метода

Последовательный алгоритм пузырьковой сортировки 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

10.2.1.2. Анализ эффективности

При анализе эффективности последовательного алгоритма пузырьковой сортировки снова используем подход, примененный в п. 7.5.4.

Итак, время выполнения алгоритма складывается из времени, которое тратится непосредственно на вычисления, и времени, необходимого на чтение данных из оперативной памяти в кэш процессора. Оценим количество вычислительных операций, выполняемых в ходе пузырьковой сортировки. На первой итерации выполняется *n-1* операций сравнения-обмена, на второй итерации – *n-2* операции и т.д. Для сортировки всего набора данных необходимо выполнить *n-1* итерацию. Следовательно, общее время выполнения вычислений составляет:

$$T_1(calc) = \sum_{i=1}^{n-1}(n-i)\cdot\tau = \frac{n^2-n}{2}\cdot\tau\,, \qquad (10.1)$$

где τ есть время выполнения базовой операции сортировки.

Если размер сортируемого набора данных настолько велик, что не может полностью поместиться в кэш, то каждый проход по массиву вызывает повторное считывание значений. Действительно, при движении по массиву и сравнении пар значений со все возрастающими индексами необходимо снова и снова считывать необходимые данные из оперативной памяти в кэш. Поскольку размер кэша ограничен, то считывание новых данных приведет к вытеснению данных, прочитанных ранее. Таким образом, время, которое требуется на чтение необходимых данных в кэш, может быть ограничено сверху величиной:

$$T_1(mem) = (n-1)\cdot\frac{8n}{\beta} \qquad (10.2)$$

где β – эффективная скорость доступа к оперативной памяти.

С учетом полученных соотношений, общее время выполнения последовательного алгоритма пузырьковой сортировки может быть вычислено по формуле:

$$T_1 = \frac{n^2-n}{2}\cdot\tau + (n-1)\cdot\frac{8n}{\beta} \qquad (10.3)$$

10.2.1.3. Программная реализация

Представим возможный вариант последовательной программы, выполняющей алгоритм пузырьковой сортировки.

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

```
void main(int argc, char* argv[]) {
  double *pData;     // Data to be sorted
  int Size;          // Size of data to be sorted

  // Data initialization
  ProcessInitialization(pData, Size);

  // Serial bubble sort
  SerialBubbleSort(pData, Size);

  // Program termination
  ProcessTermination(pData);
}
```

2. Функция ProcessInitialization. Эта функция предназначена для инициализации всех переменных, используемых в программе, в частности, для ввода количества сортируемых данных, выделения памяти для сортируемых данных и для заполнения этой памяти начальными, неотсортированными, значениями. Начальные неотсортированные значения задаются в функции *RandomDataInitialization.*

```
// Function for memory allocation and data initialization
void ProcessInitialization(double* &pData,  int &Size) {
  do {
```

```
    printf ("Enter the array size:\n");
    scanf("%d", &Size);
  } while (Size<=1);
  printf ("Chosen array size = %d\n", Size);

  pData = new double [Size];
  RandomDataInitialization(pData, Size);
}
```

Реализация функции *RandomDataInitialization* предлагается для самостоятельного выполнения. Исходные данные могут быть введены с клавиатуры, прочитаны из файла или сгенерированы при помощи датчика случайных чисел.

3. Функция SerialBubbleSort. Данная функция выполняет последовательный алгоритм пузырьковой сортировки.

```
// Function for serial bubble sorting
void SerialBubbleSort(double* pData, int Size) {
  double temp;
  for (int i=0; i<Size-1; i++)
    for (int j=1; j<Size-i; j++)
      if (pData[j-1]>pData[j]) {
        temp = pData[j];
        pData[j] = pData[j-1];
        pData[j-1] = temp;
      }
}
```

Разработку функции *ProcessTermination* также предлагается выполнить самостоятельно.

10.2.1.4. Результаты вычислительных экспериментов

Эксперименты проводились на вычислительном узле на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Для снижения сложности построения теоретических оценок времени выполнения алгоритмов при компиляции и построении программ для проведения вычислительных экспериментов функция оптимизации кода компилятором была отключена (результаты оценки влияния компиляторной оптимизации на эффективность программного кода приведены в п. 7.5.4).

Для того, чтобы оценить время одной операции сравнения-обмена τ, измерим время выполнения последовательного алгоритма пузырьковой сортировки при малых объемах данных, таких, чтобы весь массив, подлежащий сортировке, полностью поместился в кэш вычислительного элемента (процессора или его ядра). Чтобы исключить необходимость выборки данных из оперативной памяти, перед началом вычислений заполним массив случайными числами. Выполнение этого действия гарантирует предварительное перемещение данных в кэш. Далее при решении задачи все время будет тратиться непосредственно на вычисления, так как нет необходимости загружать данные из оперативной памяти. Поделив полученное время на количество выполненных операций, получим время выполнения одной операции. Для вычислительной системы, которая использовалась для проведения экспериментов, было получено значение τ, равное 13,570 нс.

Алгоритм пузырьковой сортировки обладает достаточно большой трудоемкостью, и уже при сравнительно небольшом размере массива время, необходимое на выполнение сортировки, становится значительным. При проведении вычислительных экспериментов ограничимся рассмотрением ситуации, когда все данные для сортировки могут быть полностью помещены в кэш вычислительного элемента. При этом время на чтение необходимых данных из оперативной памяти в кэш (второе слагаемое в модели (10.3)) будет равно 0.

В таблице 10.1 и на рис. 10.1 представлены результаты сравнения времени выполнения последовательного алгоритма пузырьковой сортировки со временем, полученным при помощи модели (10.3).

Таблица 10.1. Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма пузырьковой сортировки

Размер массива	Эксперимент	Модель
10000	0,6764	0,6784
20000	2,7155	2,7138

30000	6,1073	6,1062
40000	10,8649	10,8556
50000	16,9752	16,9620
60000	24,4170	24,4253
70000	33,2741	33,2456
80000	43,4184	43,4229
90000	54,9581	54,9572
100000	67,8573	67,8485

Рис. 10.1. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма от объема исходных данных

10.2.2. Метод 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 обычно 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

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

а на четных итерациях обрабатываются элементы

$(a_2, a_3), (a_4, a_5), \ldots, (a_{n-2},a_{n-1})$.

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

```
// Function for serial odd-even transposition
void OddEvenSort ( double* pData, int Size ) {
  double temp;
  int upper_bound;
  if (Size%2==0)
    upper_bound = Size/2-1;
  else
    upper_bound = Size/2;

  for (int i=0; i<Size; i++) {
    if (i%2 == 0) // even iteration
      for (int j=0; j<Size/2; j++)
        compare_exchange(pData[2*j], pData[2*j+1]);
    else          // odd iteration
```

```
      for (int j=0; j<upper_bound; j++)
        compare_exchange(pData[2*j+1], pData[2*j+2]);
  }
}
```

10.2.3. Базовый параллельный алгоритм пузырьковой сортировки

Получение параллельного варианта для метода чет-нечетной перестановки уже не представляет каких-либо затруднений – несмотря на то, что четные и нечетные итерации должны выполняться строго последовательно, сравнения пар значений на итерациях сортировки являются независимыми и могут быть выполнены параллельно. Поскольку все вычислительные элементы имеют прямой доступ к каждому значению в сортируемом массиве, сравнение значений a_i и a_j может быть выполнено любым вычислительным элементом. При наличии нескольких вычислительных элементов появляется возможность одновременно выполнять операцию «сравнить и переставить» над несколькими парами значений.

10.2.3.1. Анализ эффективности

Как и ранее, при анализе эффективности базового параллельного алгоритма пузырьковой сортировки будем предполагать, что время выполнения складывается из времени вычислений (которые могут быть выполнены вычислительными элементами параллельно) и времени, необходимого на загрузку необходимых данных из оперативной памяти в кэш. Доступ к памяти осуществляется строго последовательно.

Для сортировки массива данных методом чет-нечетной перестановки требуется выполнение n итераций алгоритма, на каждой из которых параллельно выполняется сравнение $n/2$ пар значений. Значит, время выполнения вычислений составляет:

$$T_p(calc) = \frac{n^2}{2p} \cdot \tau \qquad (10.4)$$

Если объем сортируемых данных настолько велик, что не может быть полностью помещен в кэш вычислительного элемента, то на каждой итерации методы выполняется постепенное вытеснение значений, располагаемых в начале массива для того, чтобы записать на их место значения, располагаемые в конце массива. Таким образом, для перехода к выполнению следующей итерации необходимо будет снова считывать значения из начала сортируемого набора. Таким образом, на каждой итерации происходит повторное считывание всего массива данных из оперативной памяти в кэш, и затраты на доступ к памяти составляют:

$$T_p(mem) = n \cdot \frac{8n}{\beta} \qquad (10.5)$$

При получении итоговой оценки времени выполнения базового параллельного алгоритма пузырьковой сортировки необходимо также учитывать затраты на организацию и закрытие параллельных секций:

$$T_p = \frac{n^2}{2p} \cdot \tau + n \cdot \frac{8n}{\beta} + n\delta, \qquad (10.6)$$

где δ есть накладные расходы на организацию параллельности на каждой итерации алгоритма

10.2.3.2. Программная реализация

Рассмотрим возможный вариант реализации параллельного варианта метода пузырьковой сортировки. Используя OpenMP, получение параллельного алгоритма из последовательного достигается путем добавления двух директив препроцессора, при помощи которых распараллеливаются циклы сравнения пар значений на четных и нечетных итерациях метода чет-нечетной перестановки.

```
// Function for parallel odd-even transposotion
void ParallelOddEvenSort ( double* pData, int Size ) {
  double temp;
  int upper_bound;
  if (Size%2==0)
    upper_bound = Size/2-1;
  else
    upper_bound = Size/2;

  for (int i=0; i<Size; i++) {
    if (i%2 == 0) // Even iteration
#pragma omp parallel for
      for (int j=0; j<Size/2; j++)
```

```
        compare_exchange(pData[2*j], pData[2*j+1]);
    else           // Odd iteration
#pragma omp parallel for
      for (int j=0; j<upper_bound; j++)
        compare_exchange(pData[2*j+1], pData[2*j+2]);
  }
}
```

10.2.3.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода пузырьковой сортировки проводились при условиях, указанных в п. 10.2.1.4. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 10.2. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.2. Результаты вычислительных экспериментов для параллельного метода пузырьковой сортировки (при использовании двух вычислительных ядер)

Размер массива	Последовательный алгоритм пузырьковой сортировки (T_1)	Последовательный алгоритм быстрой сортировки (T_1')	Параллельный алгоритм		
			T_p	T_1 / T_p	T_1' / T_p
10000	0,6764	0,0029	0,3736	1,8104	0,0078
20000	2,7155	0,0062	1,4667	1,8514	0,0042
30000	6,1073	0,0096	3,2777	1,8633	0,0029
40000	10,8649	0,0131	5,8133	1,8690	0,0022
50000	16,9752	0,0167	9,0719	1,8712	0,0018
60000	24,4170	0,0205	13,0361	1,8730	0,0016
70000	33,2741	0,0243	17,7571	1,8738	0,0014
80000	43,4184	0,0284	23,1374	1,8765	0,0012
90000	54,9581	0,0329	29,2510	1,8788	0,0011
100000	67,8573	0,0367	36,0884	1,8803	0,0010

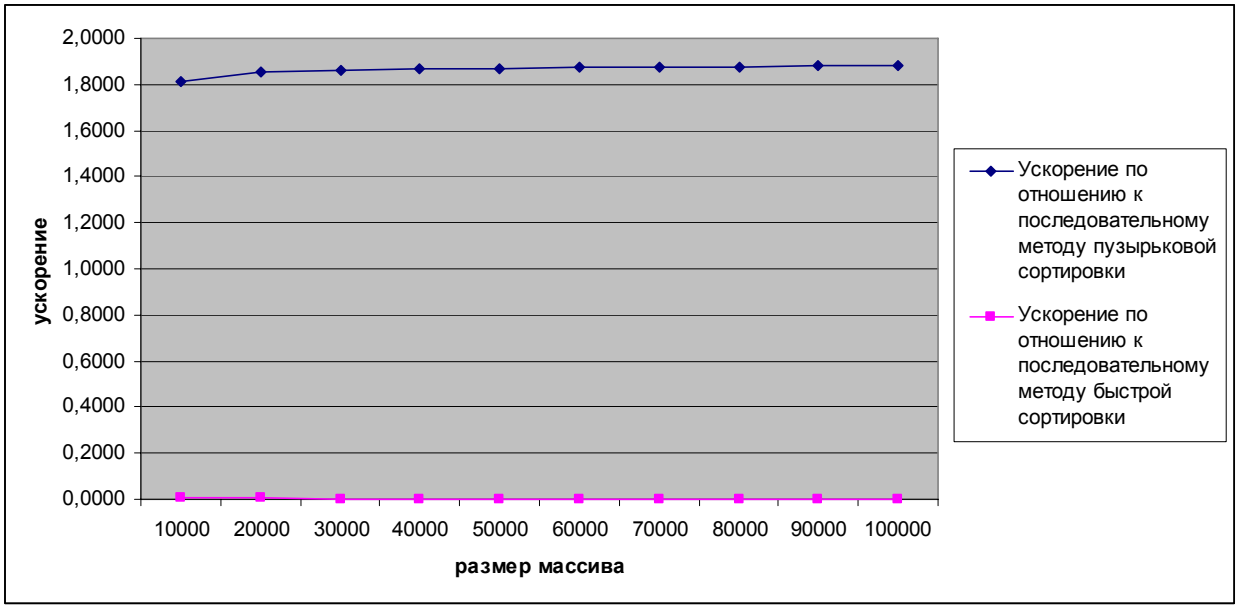


Рис. 10.2. Зависимость ускорения от количества исходных данных при выполнении параллельного метода пузырьковой сортировки

При проведении анализа эффективности 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

$$T_1 = n \log_2 n ,$$

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!

Как следует из данных, приведенных в таблице 10.2 и на рис. 10.2 ускорение полученного параллельного метода по отношению к наиболее эффективному последовательному совершенно неудовлетворительно. Поэтому при разработке параллельных методов сортировки следует использовать более сложные подходы, использующие, например, блочное разделение массива данных.

В таблице 10.3 и на рис. 10.3 представлены результаты сравнения времени выполнения базового параллельного метода пузырьковой сортировки с использованием двух потоков со временем, полученным при помощи модели (10.6). Следует отметить, однако, что базовый параллельный алгоритм пузырьковой сортировки обладает очень высокой трудоемкостью, и уже при небольших объемах сортируемых данных время на его выполнение весьма значительно. Поэтому вычислительные эксперименты выполнялись со сравнительно небольшими массивами, которые полностью могут быть помещены в кэш вычислительных элементов. Следовательно, при построении оценки времени выполнения метода время на загрузку данных из оперативной памяти не учитывалось. Величина накладных расходов δ на параллельность была оценена в разделе 7 и составляет $1 \cdot 10^{-5}$ секунд.

Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах сортируемого массива составляет не более 3%.

Таблица 10.3. Сравнение экспериментального и теоретического времени выполнения параллельного метода пузырьковой сортировки с использованием двух потоков

Размер массива	Эксперимент	Модель
10000	0,3736	0,4392
20000	1,4667	1,5570
30000	3,2777	3,3532
40000	5,8133	5,8279
50000	9,0719	8,9811
60000	13,0361	12,8128
70000	17,7571	17,3230
80000	23,1374	22,5117
90000	29,2510	28,3788
100000	36,0884	34,9245

Рис. 10.3. График зависимости экспериментального и теоретического времени выполнения базового параллельного метода пузырьковой сортировки от объема исходных данных при использовании двух потоков

10.2.4. Блочный параллельный алгоритм пузырьковой сортировки

10.2.4.1. Принципы распараллеливания

Рассмотрим ситуацию, когда количество вычислительных элементов является меньшим числа упорядочиваемых значений ($p < n$). Разделим сортируемый массив на блоки данных размера n/p. На первом этапе параллельного метода пузырьковой сортировки каждый вычислительный элемент выполняет сортировку одного из блоков данных при помощи какого-либо быстрого алгоритма (например, при помощи алгоритма быстрой сортировки – см. раздел 10.3.1); эти действия могут быть выполнены параллельно. На следующем этапе выполняется параллельный алгоритм чет-нечетной перестановки над блоками данных:

- На четных итерациях алгоритма вычислительные элементы с четными индексами $2i$ выполняют операцию "сравнить и разделить" для двух упорядоченных блоков с номерами $2i$ и $2i+1$,
- На нечетных итерациях алгоритма вычислительные элементы с нечетными индексами $2i+1$ выполняют операцию "сравнить и разделить" для двух упорядоченных блоков с номерами $2i+1$ и $2i+2$.

После выполнения p итераций чет-нечетной перестановки исходный массив оказывается упорядоченным.

Для пояснения такого параллельного способа сортировки в табл. 10.4 приведен пример упорядочения данных при $n=16$, $p=4$ (т.е. блок значений на каждом вычислительном элементе содержит $n/p=4$ элемента). В первом столбце таблицы приводится номер и тип итерации метода, перечисляются пары номеров блоков данных, для которых параллельно выполняются операции слияния. Взаимодействующие пары блоков выделены в таблице двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

Таблица 10.4. Пример сортировки данных параллельным методом чет-нечетной перестановки

№ и тип итерации	Вычислительные элементы			
	0	1	2	3
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
0 чет (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

1 нечет (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
2 чет (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
3 нечет (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

Заметим, однако, что при выполнении каждой итерации чет-нечетной перестановки блоков, задействованными оказываются только *p/2* вычислительных элементов, в то время как остальные вычислительные элементы простаивают. Это приводит к снижению общей эффективности параллельного алгоритма. Для повышения эффективности параллельного метода пузырьковой сортировки разделим исходный массив не на *p*, а на *2p* блоков, каждый из которых содержит *n/2p* значений. На первом этапе сортировки каждый вычислительный элемент выполняет сортировку двух последовательных блоков данных при помощи какого-либо быстрого алгоритма. На следующем этапе выполняется параллельный алгоритм чет-нечетной перестановки над блоками данных меньшего размера:

- На четных итерациях алгоритма каждый вычислительный элемент с индексом *i* выполняет операцию "сравнить и разделить" для двух упорядоченных блоков с номерами *2i* и *2i+1*,
- На нечетных итерациях алгоритма каждый вычислительный элемент с индексом *i* выполняет операцию "сравнить и разделить" двух упорядоченных блоков с номерами *2i+1* и *2i+2*.

После выполнения 2*p* итераций чет-нечетной перестановки исходный массив оказывается упорядоченным.

В общем случае выполнение параллельного метода может быть прекращено, если после выполнения очередной итерации массив оказался отсортированным. Как результат, общее количество итераций может быть сокращено, и для фиксации таких моментов необходимо, чтобы ведущий поток (*master thread*) определял состояние набора данных после выполнения каждой итерации сортировки.

10.2.4.2. Анализ эффективности

При построении оценок эффективности блочного параллельного алгоритма пузырьковой сортировки будут использованы соотношения, описывающие трудоемкость последовательного алгоритма быстрой сортировки (см. раздел 10.4.1). Это объясняется тем, что для первоначальной сортировки блоков данных каждым вычислительным элементом использовался именно алгоритм быстрой сортировки.

Определим теперь сложность рассмотренного параллельного алгоритма упорядочивания данных. Как отмечалось ранее, на начальной стадии работы метода каждый вычислительный элемент проводит упорядочивание двух блоков данных (размер каждого блока при равномерном распределении данных является равным *n/2p*). Предположим, что данная начальная сортировка может быть выполнена при помощи быстродействующих алгоритмов упорядочивания данных, тогда трудоемкость начальной стадии вычислений можно определить выражением вида (см. модель (10.13)):

$$T_p^1 = 2 \cdot 1{,}4 \cdot (n/2p)\log_2(n/2p). \qquad (10.7)$$

Далее на каждой выполняемой итерации параллельной сортировки каждый вычислительный элемент осуществляет объединение пары блоков при помощи процедуры слияния и затем разделения объединенного блока на две равные по размеру части. Общее количество итераций не превышает величины *2p,* и, как результат, общее количество операций этой части параллельных вычислений оказывается равным

$$T_p^2 = (2p)\cdot\left(2\frac{n}{2p}\right) = 2n. \qquad (10.8)$$

Итак, время выполнения вычислений может быть получено при помощи выражения:

$$T_p(calc) = (2 \cdot 1.4 \cdot (n/2p)\log_2(n/2p) + 2n)\cdot\tau \qquad (10.9)$$

где τ есть время выполнения базовой операции сортировки.

Теперь оценим время, необходимое на чтение данных из оперативной памяти в кэш. При выполнении первого этапа алгоритма, который состоит в «локальной» сортировке блоков, при выполнении сортировки каждого блока вычислительный элемент считывает из оперативной памяти количество значений, определяемое оценкой (10.14) с поправкой на размер блока, который обрабатывается вычислительным элементом. Поскольку доступ к памяти осуществляется строго последовательно, то время, которое необходимо одному вычислительному элементу на считывание данных, нужно умножить на количество вычислительных элементов. Далее, на каждой итерации чет-нечетной перестановки блоков вычислительные

элементы также считывают весь сортируемый набор данных снова. Таким образом, время, необходимое на загрузку необходимых данных из памяти составляет:

$$T_p(mem) = p \cdot \left(2 \cdot 1{,}4 \log_2(n/2p) \frac{8(n/2p)}{\beta} \right) + 2p \cdot \frac{8n}{\beta} = \left(1{,}4 \log_2(n/2p) + 2p\right) \cdot \frac{8n}{\beta} . \quad (10.10)$$

Кроме того, необходимо учитывать накладные расходы на организацию и закрытие параллельных секций: одна параллельная секция создается для выполнения локальной сортировки блоков, далее параллельные секции создаются для каждой итерации алгоритма чет-нечетной перестановки (2*p* итераций).

Итак, общее время выполнения параллельного алгоритма пузырьковой сортировки может быть вычислено в соответствии с выражением:

$$T_p = \left(2 \cdot 1.4 \cdot (n/2p) \log_2(n/2p) + 2n\right) \cdot \tau + \left(1{,}4 \log_2(n/2p) + 2p\right) \cdot \frac{8n}{\beta} + (2p+1) \cdot \delta . \quad (10.11)$$

10.2.4.3. Программная реализация

Рассмотрим возможный вариант реализации блочного параллельного варианта метода пузырьковой сортировки.

Для упорядочения данных необходимо организовать синхронизированное выполнение отдельных этапов блочного параллельного метода пузырьковой сортировки между потоками параллельной программы. Так, нельзя приступить к выполнению операции слияния упорядоченных блоков до тех пор, пока все вычислительные элементы не выполнят локальную сортировку блоков. Наиболее простой способ организации синхронизации – выделить каждый этап в отдельную параллельную секцию при помощи директивы **parallel**. При закрытии параллельной секции автоматически выполняется синхронизация потоков. Таким образом, можно гарантировать, что к началу выполнения очередного этапа алгоритма сортировки все потоки завершат выполнение предыдущего этапа.

Практически в каждой параллельной секции необходимо иметь доступ к переменной, которая содержит число параллельных потоков, и к переменной-идентификатору текущего потока. Значение переменной *ThreadNum*, содержащей количество потоков, одинаково во всех потоках параллельной программы, используется потоками только для чтения, и, следовательно, эта переменная может быть общей для всех потоков. Переменная-идентификатор потока *ThreadID* имеет различное значение в различных потоках. Чтобы избежать многократного вызова функций библиотеки OpenMP для определения количества потоков и идентификаторов потоков объявим соответствующие переменные как глобальные. Переменную для хранения идентификатора потока определим как *threadprivate* – значение такой переменной, будучи однажды определено для каждого потока внутри параллельной секции, сохраняется во всех последующих параллельных секциях. Функция *InitializeParallelSections* служит для инициализации переменных *ThreadNum* и *ThreadID*.

```
int ThreadNum;    // Number of threads
int ThreadID;     // Thread identifier
#pragma omp threadprivate (ThreadID)

void InitializeParallelSections() {
#pragma omp parallel
  {
    ThreadID = omp_get_thread_num();
#pragma omp single
    {
      ThreadNum = omp_get_num_threads();
    }
  }
}
```

Функция *ParallelBubbleSort* выполняет блочный параллельный алгоритм пузырьковой сортировки. Дадим пояснения об использовании дополнительных структур данных. Массив *Index* хранит индексы первых элементов блоков данных. В массиве *BlockSize* хранятся размеры блоков данных. Таким образом, *i*-ый блок данных начинается с элемента *Index[i]* исходного массива и содержит *BlockSize[i]* элементов. Использование таких дополнительных массивов позволяет работать с блоками данных разного размера, например в случае, когда размер исходного массива некратен количеству вычислительных элементов.

```
// Function for parallel bubble sorting
void ParallelBubbleSort(double* pData, int Size) {
  InitializeParallelSections();
  int* Index = new int [ThreadNum*2];
```

```
  int* BlockSize = new int [ThreadNum*2];
  for (int i=0; i<2*ThreadNum; i++) {
    Index[i] = int((i*Size)/double(2*ThreadNum));
    if (i<2*ThreadNum-1)
      BlockSize[i] = int (((i+1)*Size)/double(2*ThreadNum)) - Index[i];
    else
      BlockSize[i] = Size-Index[i];
  }

  // Local sorting with quick algorithm
#pragma omp parallel
  {
    LocalQuickSort(pData, Index[2*ThreadID],
      Index[2*ThreadID]+BlockSize[2*ThreadID] -1);
    LocalQuickSort(pData, Index[2*ThreadID+1],
      Index[2*ThreadID+1]+BlockSize[2*ThreadID+1]-1);
  }

  // Odd-even transposition of data blocks
  int Iter = 0;
  do {
#pragma omp parallel
    {
      if (Iter%2 == 0) { // Even iteration
        MergeBlocks(pData, Index [2*ThreadID], BlockSize[2*ThreadID],
          Index[2*ThreadID+1], BlockSize[2*ThreadID+1]);
      }
      else { // Odd iteration
        if (ThreadID<ThreadNum-1)
          MergeBlocks(pData, Index[2*ThreadID+1], BlockSize[2*ThreadID+1],
            Index[2*ThreadID+2], BlockSize[2*ThreadID+2]);
      }
    }// pragma omp parallel
    Iter++;
  } while (!IsSorted(pData, Size));
}
```

Функция *MergeBlocks* выполняет слияние двух подряд идущих упорядоченных блоков заданного размера.

```
// Function for merging of two sorted blocks
void MergeBlocks(double* pData, int Index1, int BlockSize1, int Index2,
  int BlockSize2) {
  double* pTempArray = new double [BlockSize1 + BlockSize2];
  int i1 = Index1, i2 = Index2, curr=0;
  while ((i1<Index1+BlockSize1) && (i2<Index2+BlockSize2)) {
    if (pData[i1] < pData[i2])
      pTempArray[curr++] = pData[i1++];
    else {
      pTempArray[curr++] = pData[i2++];
  }
  while (i1<Index1+BlockSize1)
    pTempArray[curr++] = pData[i1++];
  while (i2<Index2+BlockSize2)
    pTempArray[curr++] = pData[i2++];
  for (int i=0; i<BlockSize1+BlockSize2; i++)
    pData[Index1+i] = pTempArray[i];
  delete [] pTempArray;
}
```

Обратим внимание на следующий момент, связанный с реализацией метода. На каждой итерации параллельного алгоритма пузырьковой сортировки происходит слияние упорядоченных блоков данных, которое осуществляется при помощи функции *MergeBlocks*. Данная операция может быть выполнена более эффективно. Так, вместо определения нового массива *pTempArray* для выполнения слияния каждой пары блоков можно определить второй массив из *Size* элементов, в который будут записываться блоки данных

при выполнении слияния. Тогда при переходе к следующей итерации алгоритма исходный и вновь определенный массив можно просто поменять ролями, избежав, тем самым, трудоемкой процедуры копирования данных. С другой стороны, такая реализация усложнило бы понимание программы и по этой причине в данном случае не используется.

Функция *IsSorted* проверяет, является ли массив отсортированным.

```
// Function for checking if the array is sorted
bool IsSorted(double* pData, int Size) {
  bool res = true;
  for (int i=1; (i<Size)&&(res); i++) {
    if (pData[i]<pData[i-1])
      res=false;
  }
  return res;
}
```

10.2.4.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности блочного параллельного варианта метода пузырьковой сортировки проводились при условиях, указанных в п. 10.2.1.4. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 10.5. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.5. Результаты вычислительных экспериментов для блочного параллельного метода пузырьковой сортировки (при использовании двух вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
100000	0,0374	0,0219	1,7079
200000	0,0778	0,0476	1,6342
300000	0,1218	0,0729	1,6714
400000	0,1674	0,0974	1,7182
500000	0,2158	0,1259	1,7141
600000	0,2716	0,1557	1,7441
700000	0,3100	0,1823	1,7009
800000	0,3679	0,2095	1,7560
900000	0,4175	0,2358	1,7710
1000000	0,4718	0,2664	1,7709

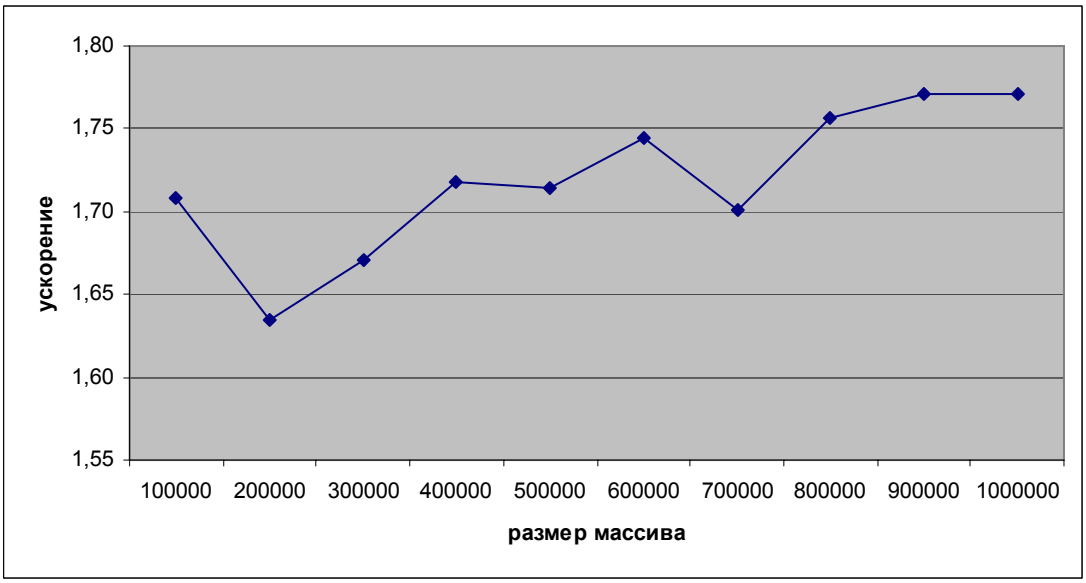


Рис. 10.4. Зависимость ускорения от количества исходных данных при выполнении параллельного метода пузырьковой сортировки

В таблице 10.6 и на рис. 10.5 представлены результаты сравнения времени выполнения параллельного метода пузырьковой сортировки с использованием двух потоков со временем, полученным при помощи модели (10.11).

Оценка величины пропускной способности канала доступа к оперативной памяти β проводилась в п. 7.5.4 и определена для используемого вычислительного узла как 5,5 Гб/с.

Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах сортируемого массива составляет не более 10%.

Таблица 10.6. Сравнение экспериментального и теоретического времени выполнения параллельного метода пузырьковой сортировки с использованием двух потоков

Размер массива	Эксперимент	Время счета ($T_p(calc)$)	Время доступа к памяти ($T_p(mem)$)	Модель (T_p)
100000	0,0219	0,0166	0,0036	0,0202
200000	0,0476	0,0351	0,0075	0,0427
300000	0,0729	0,0543	0,0116	0,0660
400000	0,0974	0,0740	0,0159	0,0899
500000	0,1259	0,0940	0,0201	0,1142
600000	0,1557	0,1143	0,0245	0,1388
700000	0,1823	0,1348	0,0289	0,1638
800000	0,2095	0,1555	0,0333	0,1889
900000	0,2358	0,1764	0,0378	0,2143
1000000	0,2664	0,1975	0,0423	0,2399

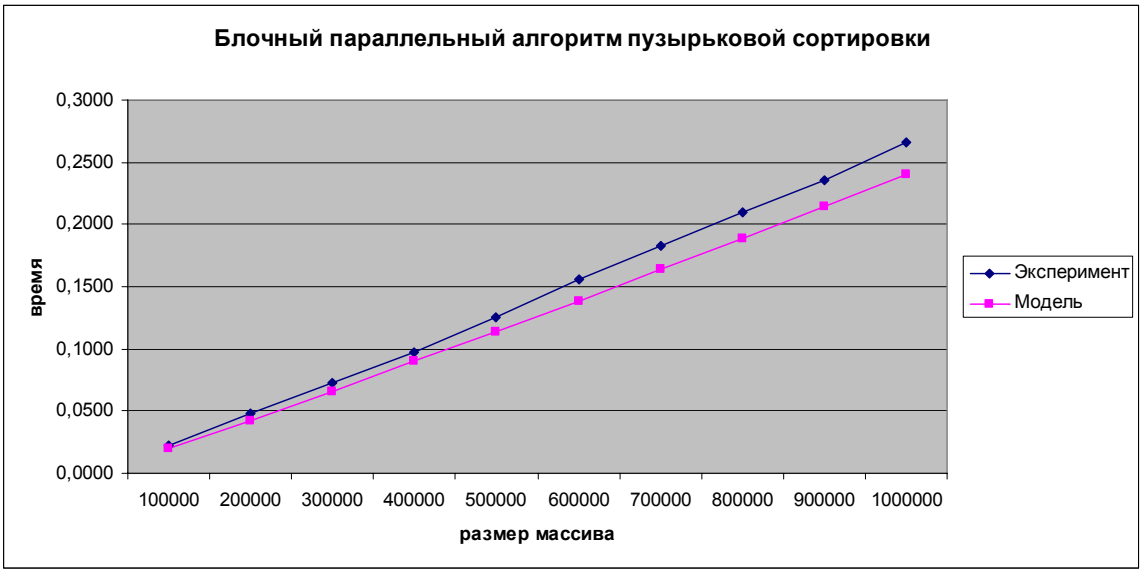


Рис. 10.5. График зависимости экспериментального и теоретического времени выполнения блочного параллельного метода пузырьковой сортировки от объема исходных данных при использовании двух потоков

10.3. Сортировка Шелла

10.3.1. 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.

```
// 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 10.2. Последовательный алгоритм сортировки Шелла

10.3.2. Организация параллельных вычислений

Для алгоритма Шелла может быть предложен параллельный аналог метода (см., например, Kumar et al. (2003)), если количество вычислительных элементов равно $p=2^N$. Разделим сортируемый набор данных на $2p$ блоков равного размера. Таким образом, количество блоков данных является равным $q=2^{N+1}$. Образуем из блоков гиперкуб размерности N+1. Выполнение сортировки в этом случае может быть разделено на два последовательных этапа. На первом этапе (N+1 итерация) осуществляется взаимодействие блоков данных, являющихся соседними в структуре гиперкуба (но эти блоки могут оказаться далекими при линейной нумерации; для установления соответствия двух систем нумерации процессоров может быть использован код Грея – см. раздел 3). На каждой итерации множество блоков разделяется на пары и для каждой пары блоков выполняется операция "сравнить и разделить". Формирование пар блоков, взаимодействующих между собой, может быть обеспечено при помощи следующего простого правила – на каждой итерации i, $0 \le i < N$+1, парными становятся блоки, у которых различие в битовых представлении их номеров имеется только в позиции (N+1)-i. Отметим, что обработка различных пар взаимодействующих блоков может выполняться параллельно.

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 $2p$.

На рис. 10.6 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.

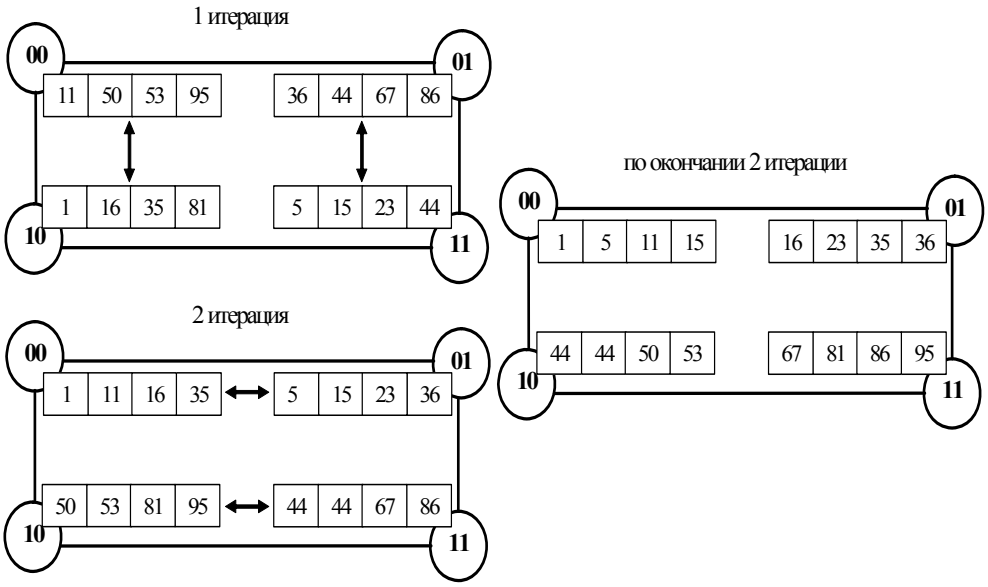


Рис. 10.6. Пример работы алгоритма Шелла для 2 вычислительных элементов (4 блока данных), номера блоков данных даны в битовом представлении

10.3.3. Анализ эффективности

Для оценки эффективности параллельного аналога алгоритма Шелла могут быть использованы соотношения, полученные для блочного параллельного метода пузырьковой сортировки (см. п. 10.2.4.2). При этом следует только учесть двухэтапность алгоритма Шелла – с учетом данной особенности общее время выполнения нового параллельного метода может быть определено при помощи выражения:

$$T_p = (2 \cdot 1.4 \cdot (n/2p)\log_2(n/2p) + (\log_2(2p) + L) \cdot n/p) \cdot \tau + ((\log_2(2p) + L) + 1{,}4\log_2(n/2p)) \cdot \frac{8n}{\beta} + (\log_2(2p) + L + 1) \cdot \delta \quad (10.12)$$

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.

10.3.4. Программная реализация

Рассмотрим возможный вариант реализации параллельного варианта метода сортировки Шелла. Как и при разработке программ, реализующих алгоритм пузырьковой сортировки, объявим несколько глобальных переменных: *ThreadNum* для определения количества потоков в параллельной программе, *DimSize* для определения размерности гиперкуба, который может быть составлен из заданного количества блоков данных, *ThreadID* для определения номера текущего потока, Создадим локальные копии переменной *ThreadID* при помощи директивы *threadprivate*.

Функция *InitializeParallelSections* определяет количество потоков *ThreadNum* и размерность виртуального гиперкуба *DimSize*, а также идентификатор потока *ThreadID*.

```
int ThreadNum;  // Number of threads
int ThreadID;   // Thread identifier
int DimSize;    // Number of dimension in hypercube, assembled of data blocks

#pragma omp threadprivate(ThreadID)

void InitializeParallelSections() {
#pragma omp parallel
  {
    ThreadID = omp_get_thread_num();
#pragma omp single
      ThreadNum = omp_get_num_threads();
  }
  DimSize = int(log10(double(ThreadNum))/log10(2.0))+1;
}
```

Для установления соответствия между номером блока данных в линейной нумерации и его номером в структуре гиперкуба используется код Грея (функция *GrayCode*). Для выполнения обратной операции

(операции получения номера блока данных в линейной нумерации по его рангу в структуре гиперкуба) используется функция *ReverseGrayCode*:

```
// Function for calculation of the data block number in hypercube
int GrayCode (int RingID, int DimSize) {
  if ((RingID==0) && (DimSize==1))
    return 0;
  if ((RingID==1) && (DimSize==1))
    return 1;
  int res;
  if (RingID < (1<<(DimSize-1)))
    res = GrayCode(RingID, DimSize-1);
  else
    res = (1<<(DimSize-1))+GrayCode((1<<DimSize)-1-RingID, DimSize-1);
  return res;
}

// Function for calculation of the data block number in linear sequence
int ReverseGrayCode (int CubeID, int DimSize) {
  for (int i=0; i<(1<<DimSize); i++) {
    if (CubeID == GrayCode(i, DimSize))
      return i;
  }
}
```

Выполнение алгоритма может быть разделено на три этапа. На *первом этапе* вычислительные элементы параллельно выполняют сортировку блоков данных, при этом каждый вычислительный элемент сортирует 2 блока при помощи последовательного алгоритма быстрой сортировки. На *втором этапе* выполняются итерации алгоритма Шелла. И, наконец, на *третьем этапе* выполняются итерации метода чет-нечетной перестановки блоков до окончания фактического изменения сортируемого массива.

Процедура выполнения первого и третьего этапов является достаточно понятной, реализация же итераций алгоритма Шелла (*второй этап*) требует дополнительных пояснений. На каждой итерации алгоритма формируется массив пар номеров блоков в структуре гиперкуба, над которыми необходимо выполнить операцию «сравнить и разделить». Число таких пар совпадает с числом вычислительных элементов. Пара с номером i, $0 \le i < p$, хранится в массиве *BlockPairs* в элементах с индексами $2i$ и $2i+1$. Формирование массива пар осуществляется в функции *SetBlockPairs*:

```
// Function for block pairs determination
// "Compare-split" operation will be carried out for that pairs
void SetBlockPairs (int* BlockPairs, int Iter) {
  int PairNum = 0, FirstValue, SecondValue;
  bool Exist;
  for (int i=0; i<2*ThreadNum; i++) {
    FirstValue = GrayCode(i, DimSize);
    Exist = false;
    for (int j=0; (j<PairNum)&&(!Exist); j++)
      if (BlockPairs[2*j+1] == FirstValue)
        Exist = true;
    if (!Exist) {
      SecondValue = FirstValue^(1<<(DimSize-Iter-1));
      BlockPairs[2*PairNum] = FirstValue;
      BlockPairs[2*PairNum+1] = SecondValue;
      PairNum++;
    } // if
  } // for
}
```

Далее для каждой пары блоков необходимо определить вычислительный элемент, который будет выполнять операцию «сравнить и разделить». Возможный способ определения номера вычислительного элемента, который выполняет операцию над данной парой блоков, состоит в следующем - необходимо из битового представления номера одного из блоков, составляющих пару, вычеркнуть бит, расположенный в позиции с номером, равным номеру итерации. Для пояснения описанного алгоритма на рис 10.7 приведен пример разделения блоков данных на пары и представлен механизм выбора вычислительного элемента в случае, когда p=4 (т.е., количество блоков данных равно $2p$=8).

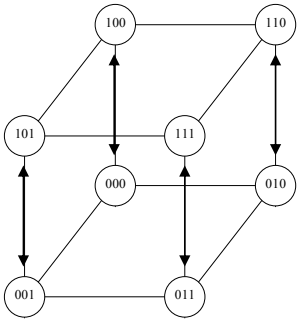
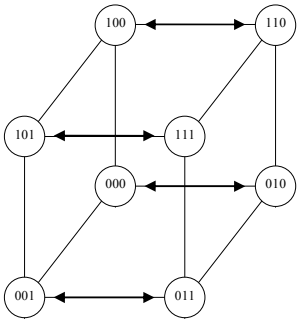
Направление обмена	Пары блоков данных	Номер вычислительного элемента
Итерация 0		
	0 (0~~0~~00) ↔ 4 (1~~0~~0)	0
	1 (0~~0~~1) ↔ 5 (1~~0~~1)	1
	2 (0~~1~~0) ↔ 6 (1~~1~~0)	2
	3 (0~~1~~1) ↔ 7 (1~~1~~1)	3
Итерация 1		
	0 (0~~0~~0) ↔ 2 (0~~1~~0)	0
	1 (0~~0~~1) ↔ 3 (0~~1~~1)	1
	4 (1~~0~~0) ↔ 6 (1~~1~~0)	2
	5 (1~~0~~1) ↔ 7 (1~~1~~1)	3
…		

Рис. 10.7. Разделение блоков данных на пары и определение номера вычислительного элемента, который должен выполнить операцию «сравнить и разделить» для пары блоков

Следуя предложенной схеме определим функцию *FindMyPair*, которая для каждого вычислительного элемента с номером *ThreadID* определяет номер пары в массиве *BlockPairs*, над которой данный вычислительный элемент должен выполнить операцию «сравнить и разделить» на данной итерации *Iter* алгоритма Шелла:

```
// Function for determination of the block pair
// Current thread will perform "compare-split" operation for this block pair
int FindMyPair (int* BlockPairs, int ThreadID, int Iter) {
  int BlockID=0, index, result;
  for (int i=0; i<ThreadNum; i++) {
    BlockID = BlockPairs[2*i];
    if (Iter == 0)
      index = BlockID%(1<<DimSize-Iter-1);
    if ((Iter>0)&&(Iter<DimSize-1))
      index = ((BlockID>>(DimSize-Iter))<<(DimSize-Iter-1)) |
        (BlockID%(1<<(DimSize-Iter-1)));
    if (Iter == DimSize-1)
      index = BlockID>>1;
    if (index == ThreadID) {
      result = i;
      break;
    }
  }
  return result;
}
```

Необходимо отметить, что предложенный способ определения номера вычислительного элемента по индексам блоков, составляющих пару, обладает важным положительным свойством: на соседних итерациях алгоритма Шелла для каждого вычислительного элемента сохраняется один из обрабатываемых блоков. Таким образом, при переходе от одной итерации к другой, нет необходимости считывать из оперативной памяти в кэш вычислительного элемента оба блока, предназначенные к обработке, – достаточно прочитать только один, «новый» для вычислительного элемента блок. Выигрыш от подобного распределения пар по

вычислительным элементам можно получить лишь в том случае, когда вычислительная система такова, что процессоры (ядра) имеют раздельный кэш и размер кэша является достаточным для размещения двух блоков одновременно.

Для того, чтобы обеспечить удачное расположение блоков данных в кэш-памяти вычислительных элементов перед началом выполнения итераций алгоритма Шелла, применим обратную циклическую схему распределения блоков между вычислительными элементами при выполнении локальной сортировки блоков. Это значит, что каждый вычислительный элемент выполняет сортировку блоков данных, которые в структуре гиперкуба имеют номера (*ThreadNum*+*ThreadID*) и *ThreadID* в указанной последовательности.

Итак, функция *ParallelShellSort* выполняет параллельный алгоритм сортировки Шелла.

```
// Function for parallel Shell sorting
void ParallelShellSort(double* pData, int Size) {
  InitializeParallelSections();
  int* Index = new int [2*ThreadNum];
  int* BlockSize = new int [2*ThreadNum];
  int * BlockPairs = new int [2*ThreadNum];

  for (int i=0; i<2*ThreadNum; i++) {
    Index[i] = int((i*Size)/double(2*ThreadNum));
    if (i<2*ThreadNum-1)
      BlockSize[i] = int (((i+1)*Size)/double(2*ThreadNum)) - Index[i];
    else
      BlockSize[i] = Size-Index[i];
  }

  // Local sorting of data blocks (reverse cycle scheme)
#pragma omp parallel
  {
    int BlockID = ReverseGrayCode(ThreadNum+ThreadID, DimSize);
    QuickSorter(pData, Index[BlockID], Index[BlockID]+BlockSize[BlockID]-1);
    BlockID = ReverseGrayCode(ThreadID, DimSize);
    QuickSorter(pData, Index[BlockID], Index[BlockID]+BlockSize[BlockID]-1);
  }

  // Iterations of the Shell method
  for (int Iter=0; (Iter<DimSize) && (!IsSorted(pData, Size)); Iter++) {
    // Block pairs determination
    SetBlockPairs(BlockPairs, Iter);

    // ”Compare-split” operation for data blocks
#pragma omp parallel
    {
      int MyPairNum = FindMyPair(BlockPairs, ThreadID, Iter);
      int FirstBlock = ReverseGrayCode(BlockPairs[2*MyPairNum], DimSize);
      int SecondBlock = ReverseGrayCode(BlockPairs[2*MyPairNum+1], DimSize);
      CompareSplitBlocks(pData, Index[FirstBlock], BlockSize[FirstBlock],
        Index[SecondBlock], BlockSize[SecondBlock]);
    } // pragma omp parallel
  } // for

  // Odd-even blocks’ transposition
  int Iter = 1;
  while (!IsSorted(pData, Size)) {
#pragma omp parallel
    {
      if (Iter%2 == 0) // Even iteration
        MergeBlocks(pData, Index[2*ThreadID], BlockSize[2*ThreadID],
          Index[2*ThreadID+1], BlockSize[2*ThreadID+1]);
      else              //  Odd iteration
        if (ThreadID<ThreadNum-1)
          MergeBlocks(pData, Index[2*ThreadID+1], BlockSize[2*ThreadID+1],
           Index[2*ThreadID+2], BlockSize[2*ThreadID+2]);
    } // pragma omp parallel
    Iter++;
```

```
  } // while

  delete [] Index;
  delete [] BlockSize;
  delete [] BlockPairs;
}
```

10.3.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода сортировки Шелла проводились при условиях, указанных в п. 10.2.1.4. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 10.7. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.7. Результаты вычислительных экспериментов для параллельного метода сортировки Шелла (при использовании двух вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
100000	0,0374	0,0227	1,6461
200000	0,0778	0,0468	1,6613
300000	0,1218	0,0720	1,6925
400000	0,1674	0,0986	1,6976
500000	0,2158	0,1237	1,7444
600000	0,2716	0,1518	1,7898
700000	0,3100	0,1789	1,7330
800000	0,3679	0,2045	1,7990
900000	0,4175	0,2326	1,7950
1000000	0,4718	0,2559	1,8441

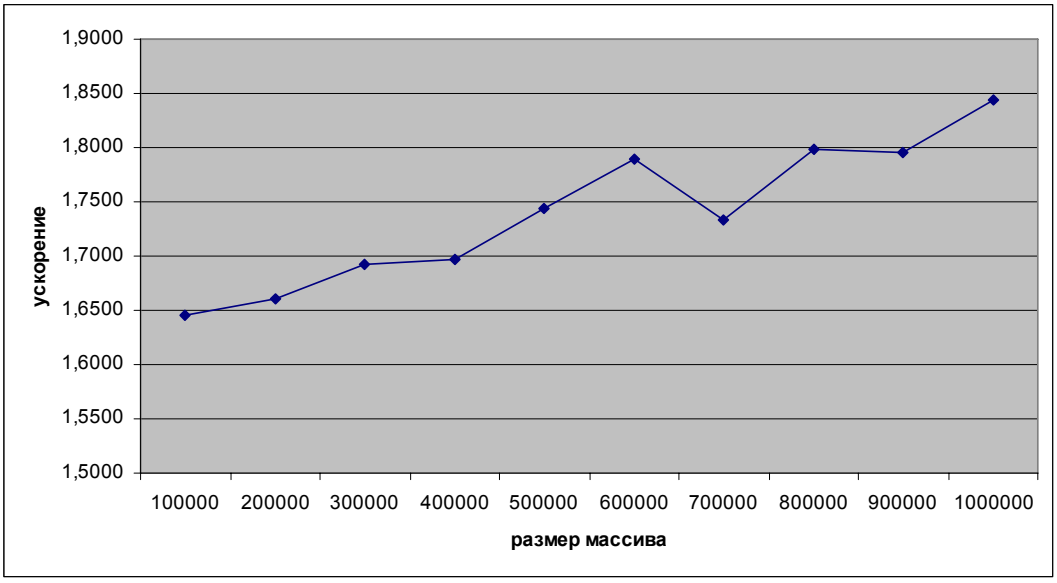


Рис. 10.8. Зависимость ускорения от количества исходных данных при выполнении параллельного метода сортировки Шелла

В таблице 10.8 и на рис. 10.9 представлены результаты сравнения времени выполнения параллельного метода сортировки Шелла с использованием двух потоков со временем, полученным при помощи модели (10.12). Как видно из приведенных данных, относительная погрешность оценки составляет не более 7%.

Таблица 10.8. Сравнение экспериментального и теоретического времени выполнения параллельного метода сортировки Шелла с использованием двух потоков

Размер массива	Эксперимент	Время счета ($T_p(calc)$)	Время доступа к памяти ($T_p(mem)$)	Модель (T_p)

100000	0,0227	0,0166	0,0036	0,0202
200000	0,0468	0,0351	0,0075	0,0427
300000	0,0720	0,0543	0,0116	0,0660
400000	0,0986	0,0740	0,0159	0,0899
500000	0,1237	0,0940	0,0201	0,1142
600000	0,1518	0,1143	0,0245	0,1388
700000	0,1789	0,1348	0,0289	0,1638
800000	0,2045	0,1555	0,0333	0,1889
900000	0,2326	0,1764	0,0378	0,2143
1000000	0,2559	0,1975	0,0423	0,2399

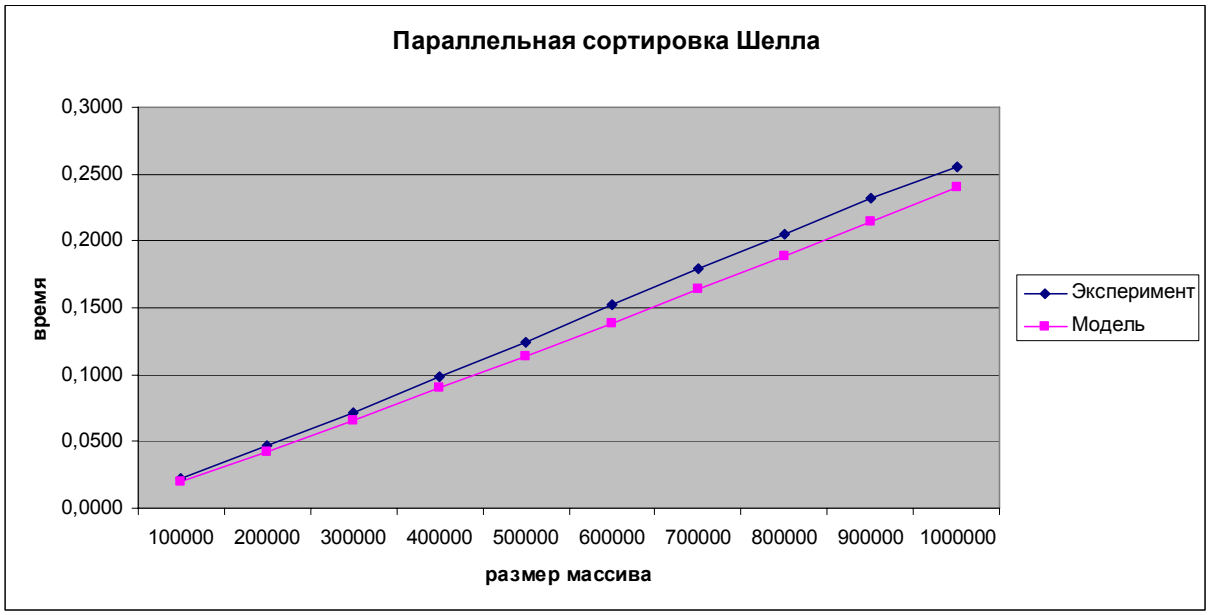


Рис. 10.9. График зависимости экспериментального и теоретического времени выполнения параллельного метода сортировки Шелла от объема исходных данных при использовании двух потоков

10.4. Быстрая сортировка

Алгоритм быстрой сортировки, предложенной Хоаром (*Hoare C.A.R.*), относится к числу эффективных методов упорядочивания данных и широко используется в практических приложениях.

10.4.1. Последовательный алгоритм быстрой сортировки

10.4.1.1. Общая схема метода

Алгоритма быстрой сортировки 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).

10.4.1.2. Анализ эффективности

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 время выполнения алгоритма быстрой сортировки, определяется выражением (см., например, Кнут (1981), Кормен, Лейзерсон и Ривест (1999)):

$$T_{calc} = 1.4 \cdot n \log_2 n \cdot \tau . \quad (10.13)$$

Если размер сортируемого массива настолько велик, что массив не может быть полностью помещен в кэш вычислительного элемента, то по мере выполнения последовательного алгоритма быстрой сортировки будет происходить чтение данных из оперативной памяти в кэш. Количество чтений определяется порядком выполнения итераций алгоритма и разницей в объеме данных для сортировки и объеме кэш. При построении оценки сверху будем считать, что необходимо выполнить чтение всего сортируемого массива из оперативной памяти в кэш на каждой итерации алгоритма быстрой сортировки. Таким образом, время на обращение к оперативной памяти составляет:

$$T_{mem} = 1{,}4 \cdot \log_2 n \cdot (8n / \beta) . \quad (10.14)$$

Таким образом, общее время выполнения последовательного алгоритма быстрой сортировки может быть определено при помощи выражения:

$$T_1 = 1.4 \cdot n \log_2 n \cdot \tau + 1{,}4 \cdot \log_2 n \cdot \frac{8 \cdot n}{\beta} = 1.4 \cdot n \log_2 n \cdot (\tau + 8n / \beta) \quad (10.15)$$

10.4.1.3. Программная реализация

Приведем код рекурсивной функции, выполняющей последовательный алгоритм быстрой сортировки. В качестве ведущего элемента выбирается первый элемент упорядочиваемого набора данных.

```
// Function for serial quick sorting
void SerialQuickSort (double* pData, int first, int last) {
  if (first >= last)
    return;
  int PivotPos = first;
  double Pivot = pData[first];
  for (int i=first+1; i<=last; i++) {
    if (pData[i] < Pivot) {
      if (i != PivotPos+1)
        swap(pData[i], pData[PivotPos+1]);
      PivotPos++;
    }
  }
  swap (pData[first], pData[PivotPos]);
  QuickSorter(pData, first, PivotPos-1);
  QuickSorter(pData, PivotPos+1, last);
}
```

10.4.1.4. Результаты вычислительных экспериментов

Эксперименты проводились при условиях, указанных в пункте 10.2.1.4. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. В таблице 10.9 и на рис. 10.10 представлены результаты сравнения времени выполнения последовательного алгоритма быстрой сортировки со временем, полученным при помощи модели (10.15). Как следует из приведенных данных, погрешность аналитической оценки трудоемкости алгоритма быстрой сортировки составляет не более 10%.

Таблица 10.9. Сравнение экспериментального и теоретического времени выполнения последовательного алгоритма быстрой сортировки

Размер массива	Эксперимент	Время вычислений (T_{calc})	Время доступа к памяти (T_{mem})	Модель (T_1)
100000	0,0374	0,0316	0,0034	0,0349
200000	0,0778	0,0669	0,0072	0,0741
300000	0,1218	0,1037	0,0111	0,1148
400000	0,1674	0,1414	0,0152	0,1566
500000	0,2158	0,1798	0,0193	0,1991

600000	0,2716	0,2188	0,0235	0,2422
700000	0,3100	0,2582	0,0277	0,2859
800000	0,3679	0,2980	0,0319	0,3300
900000	0,4175	0,3382	0,0363	0,3744
1000000	0,4718	0,3787	0,0406	0,4192

Рис. 10.10. График зависимости экспериментального и теоретического времени выполнения последовательного алгоритма быстрой сортировки от объема исходных данных

10.4.2. Параллельный алгоритм быстрой сортировки

10.4.2.1. Организация параллельных вычислений

Параллельное обобщение алгоритма быстрой сортировки (см., например, Quinn (2003)) наиболее простым способом может быть получено для случая, когда потоки параллельной программы могут быть организованы в виде N-мерного гиперкуба (т.е. количество вычислительных элементов $p=2^N$). Пусть, как и ранее, исходный набор данных логически разделен на $2p$ блоков одинакового размера $n/2p$. Тогда блоки данных образуют $(N+1)$-мерный гиперкуб. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

- выбрать каким-либо образом ведущий элемент (например, в качестве ведущего элемента можно взять среднее арифметическое элементов, расположенных на выбранном ведущем блоке);
- сформировать пары блоков, для которых необходимо выполнить взаимообмен данными на данной итерации алгоритма: пары образуют блоки, для которых битовое представление номеров отличается только в позиции $(N+1)$;
- для каждой пары блоков определить вычислительный элемент, который будет выполнять необходимые операции (для определения номера вычислительного элемента по индексам блоков, составляющих пару, можно воспользоваться алгоритмом, предложенным при рассмотрении параллельного варианта метода Шелла);
- параллельно выполнить операцию «сравнить и разделить» над всеми парами блоков, в результате такого обмена в блоках, для которых в битовом представлении номера бит позиции $N+1$ равен 0, должны оказаться части блоков со значениями, меньшими ведущего элемента; блоки с номерами, в которых бит $N+1$ 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 в блоках данных, в битовом представлении номеров которых бит $N+1$ равен 0. Таких блоков всего p и, таким образом, исходный $(N+1)$ -мерный гиперкуб также оказывается разделенным на два гиперкуба размерности N. К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После $(N+1)$-кратного повторения подобных итераций для завершения сортировки достаточно упорядочить полученные блоки данных, каждый вычислительный элемент упорядочивает 2 блока.

Для пояснения на рис.10.11 представлен пример упорядочивания данных при $n=16$, $p=2$ (т.е. исходный массив разбит на $2p = 4$ блока, каждый блок данных содержит 4 значения). На этом рисунке блоки данных изображены в виде прямоугольников; значения блоков приводятся в начале и при завершении каждой итерации сортировки. Взаимодействующие пары блоков соединены двунаправленными стрелками. Для разделения данных выбирались наилучшие значения ведущих элементов: на первой итерации для всех блоков использовалось значение 0, на второй итерации для пары блоков 0, 1 ведущий элемент равен -5, для пары блоков 2, 3 это значение было принято равным 4.

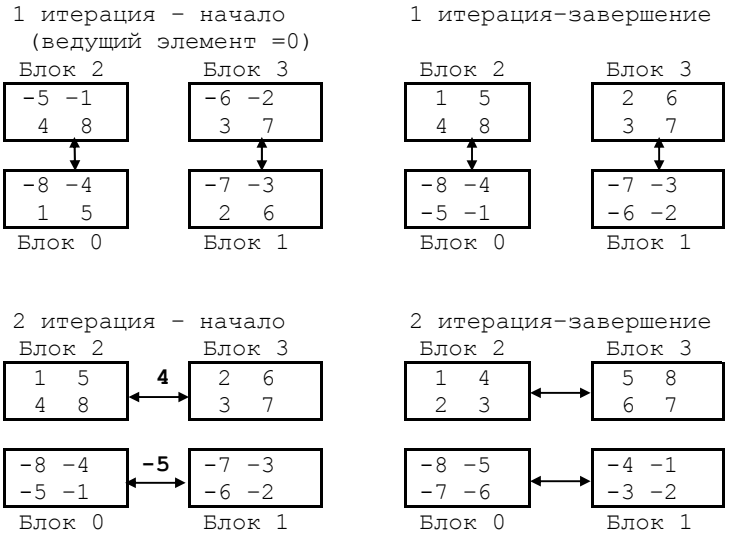


Рис. 10.11. Пример упорядочивания данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

10.4.2.2. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Пусть у нас имеется $(N+1)$-мерный гиперкуб, состоящий из $2p=2^{N+1}$ блоков данных, где $2p<n$.

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 потоками параллельной программы.

Определим вначале вычислительную сложность алгоритма сортировки. На каждой из $log_2(2p)$ итераций сортировки каждый поток осуществляет операцию «сравнить и разделить» над парой блоков в соответствии со значением ведущего элемента. Сложность этой операции составляет n/p операций (будем предполагать, что на каждой итерации блок данных делится на равные по размеру части относительно ведущего элемента и, следовательно, размер блоков данных в процессе сортировки остается постоянным).

При завершении вычислений потоки выполняют сортировку двух блоков, что может быть выполнено при использовании быстрых алгоритмов за $2 \cdot 1{,}4(n/2p)\log_2(n/2p)$ операций.

Таким образом, общее время вычислений параллельного алгоритма быстрой сортировки составляет

$$T_p(calc) = [(n/p)\log_2(2p) + 2 \cdot 1{,}4(n/2p)\log_2(n/2p)] \cdot \tau, \qquad (10.16)$$

где τ есть время выполнения базовой операции перестановки.

Предположим, что выбор ведущих элементов осуществляется самым наилучшим образом, количество итераций алгоритма равно $log_2(2p)$, и все блоки данных сохраняют постоянный размер $(n/2p)$. При таких условиях, на каждой итерации сравнения блоков выполняется считывание в кэш из оперативной памяти всего сортируемого массива. Далее на этапе локальной сортировки блоков каждый вычислительный элемент считывает упорядочиваемые блоки данных на каждой итерации повторно (см. оценку (10.14)). Таким образом, затраты на считывание необходимых данных из оперативной памяти в кэш составляют:

$$T_p(mem) = \log_2(2p) \cdot \frac{8n}{\beta} + p \cdot 2 \cdot 1{,}4\log_2(n/2p) \cdot \frac{8(n/2p)}{\beta} = (\log_2(2p) + 1.4\log_2(n/2p)) \cdot \frac{8n}{\beta}, \qquad (10.17)$$

С учетом всех полученных соотношений общая трудоемкость алгоритма оказывается равной:

$$T_p = [(n/p)\log_2(2p) + 2 \cdot 1{,}4(n/2p)\log_2(n/2p)]\tau + (\log_2(2p) + 1.4\log_2(n/2p)) \cdot \frac{8n}{\beta} \cdot \qquad (10.18)$$

Кроме того, необходимо учесть накладные расходы на организацию параллельности. На каждой итерации алгоритма создается параллельная секция для выполнения операции «сравнить и разделить». Еще одна параллельная секция создается для выполнения финальной локальной сортировки блоков на всех потоках параллельной программы, т.е.:

$$T_p = [(n/p)\log_2(2p) + 2\cdot 1{,}4(n/2p)\log_2(n/2p)]\tau + (\log_2(2p) + 1.4\log_2(n/2p))\cdot\frac{8n}{\beta} + (\log_2(2p)+1)\cdot\delta \qquad (10.19)$$

Однако следует отметить, что при построении этой оценки предполагалось, что выбор ведущих элементов осуществляется наилучшим образом. На практике обеспечить такой выбор ведущих элементов, который приводил бы к равному разделению блоков потоков и, соответственно, к равномерному распределению вычислительной нагрузки достаточно сложно. В среднем случае время выполнения параллельного алгоритма быстрой сортировки может быть определено аналогично времени выполнения последовательного алгоритма:

$$T_p = 1.4\cdot\left[[(n/p)\log_2(2p) + 2\cdot 1{,}4(n/2p)\log_2(n/2p)]\tau + (\log_2(2p) + 1.4\log_2(n/2p))\cdot\frac{8n}{\beta}\right] + (\log_2(2p)+1)\cdot\delta \qquad (10.20)$$

10.4.2.3. Программная реализация

Рассмотрим возможный вариант реализации параллельного варианта метода быстрой сортировки. Как и ранее, объявим несколько глобальных переменных: *ThreadNum* для определения количества потоков в параллельной программе, *DimSize* для определения размерности гиперкуба, который может быть составлен из заданного количества блоков данных, *ThreadID* для определения номера текущего потока. Создадим локальные копии переменной *ThreadID* при помощи директивы *threadprivate*. Функция *InitializeParallelSections* определяет количество потоков *ThreadNum* и размерность виртуального гиперкуба *DimSize*, а также идентификатор потока *ThreadID*.

Алгоритм выбора ведущих элементов для выполнения операции сравнения и разделения блоков основан на информации о распределении сортируемых значений. Если минимальное возможное значения элемента сортируемого набора равно *MIN*, а максимальное – *MAX*, то в качестве ведущего элемента на первой итерации алгоритма выбирается среднее арифметическое значение *Pivot = (MIN+MAX)/2*. Далее, предполагая равномерное изменение значений упорядочиваемых данных, на второй итерации алгоритма для подгиперкуба с меньшими номерами блоков будем использовать значение *Pivot = (3MIN+MAX)/4*, а для подгиперкуба с большими номерами блоков – *Pivot = (MIN+3MAX)/4*, и т.д.

Поскольку при выполнении параллельного алгоритма быстрой сортировки достаточно сложно обеспечить идеальный выбор ведущего элемента, блоки данных различных вычислительных элементов могут иметь разный размер. В худшем случае, в какой-то момент времени все данные могут быть сосредоточены в блоке одного вычислительного элемента. Это делает невозможным хранение блоков в рамках исходного массива. Поэтому для раздельного хранения блоков данных разных вычислительных элементов заведем систему буферов *pTempData,* каждый из которых может вместить *Size* элементов. В ходе сортировки упорядочиваемые данные размещаются в этих буферах; количество элементов, размещаемых в *i*-ом блоке *pTempData[i]*, определяется значением переменной *BlockSize[i]*.

После выполнения *DimSize* итераций сравнения и разделения блоков, выполняется локальная сортировка блоков. Далее данные из буферов *pTempData* собираются в исходный массив *pData*. После выполнения указанных действий массив оказывается отсортированным.

Функция *ParallelQuickSort* выполняет параллельный алгоритм быстрой сортировки:

```
// Function for parallel quick sorting
void ParallelQuickSort (double* pData, int Size) {
  InitializeParallelSections();
  double ** pTempData = new double * [2*ThreadNum];
  int * BlockSize = new int [2*ThreadNum];
  double* Pivots = new double [ThreadNum];
  int * BlockPairs = new int [2*ThreadNum];
  for (int i=0; i<2*ThreadNum; i++) {
    pTempData[i] = new double [Size];
    BlockSize[i] = Size/(2*ThreadNum);
  }
  for (int j=0; j<Size; j++)
    pTempData[2*j*ThreadNum/Size][j%(Size/(2*ThreadNum))] = pData[j];

  // Iterations of quick sorting
  for (int i=0; i<DimSize parallel; i++) {
    // Determination of pivot values
    for (int j=0; j<ThreadNum; j++)
```

```
      Pivots[j] = (MAX_VALUE + MIN_VALUE)/2;
    for (int iter=1; iter<=i; iter++)
      for (int j=0; j<ThreadNum; j++)
        Pivots[j] = Pivots[j] - pow(-1.0f, j/((2*ThreadNum)>>(iter+1))) *
          (MAX_VALUE-MIN_VALUE)/(2<<iter);

    // Determination of data block pairs
    SetBlockPairs(BlockPairs, i);

#pragma omp parallel
    {
      int MyPair = FindMyPair(BlockPairs, ThreadID, i);
      int FirstBlock = BlockPairs[2*MyPair];
      int SecondBlock = BlockPairs[2*MyPair+1];
      CompareSplitBlocks(pTempData[FirstBlock], BlockSize[FirstBlock],
        pTempData[SecondBlock], BlockSize[SecondBlock], Pivots[ThreadID]);
    }// pragma omp parallel
  } // for

// Local sorting
#pragma omp parallel
  {
    if (BlockSizes[2*ThreadID]>0)
      SerialQuickSort(pTempData[2*ThreadID], BlockSize[2*ThreadID]);
    if (BlockSizes[2*ThreadID+1]>0)
      SerialQuickSort(pTempData[2*ThreadID+1], BlockSize[2*ThreadID+1]);
  }

  int curr = 0;
  for (int i=0; i<2*ThreadNum; i++)
    for (int j=0; (j<BlockSize[i])&&(curr<Size); j++)
      pData[curr++] = pTempData[i][j];

  for (int i=0; i<ThreadNum; i++)
    delete [] pTempData[i];
  delete [] pTempData;
  delete [] BlockSize;
  delete [] Pivots;
  delete [] BlockPairs;
}
```

Следует отметить, что при реализации параллельного алгоритма быстрой сортировки пары блоков данных формируются непосредственно на основании индексов этих блоков, в отличие от сортировки Шелла, где пары блоков формировались на основе кодов Грея индексов блоков:

```
// Function for block pairs determination.
// "Compare-split" operation will be carried out for that pairs.
void SetBlockPairs (int* BlockPairs, int Iter) {
  int PairNum = 0, FirstValue, SecondValue;
  bool Exist;
  for (int i=0; i<2*ThreadNum; i++) {
    FirstValue = i;
    Exist = false;
    for (int j=0; (j<PairNum)&&(!Exist); j++)
      if (BlockPairs[2*j+1] == FirstValue)
        Exist = true;
    if (!Exist) {
      SecondValue = FirstValue^(1<<(DimSize-Iter-1));
      BlockPairs[2*PairNum] = FirstValue;
      BlockPairs[2*PairNum+1] = SecondValue;
      PairNum++;
    } // if
  } // for
}
```

Функция *CompareSplitBlocks* выполняет операцию «сравнить и разделить» для блоков *FirstBlock* и *SecondBlock* указанных размеров в соответствии со значением элемента *Pivot*. После выполнения операции в блоке *FirstBlock* оказываются значения из обоих блоков, меньшие ведущего элемента, а в блоке *SecondBlock* – значения, большие ведущего элемента:

```
// Function for carrying out the "compare-split" operation
// for two non-sorted data blocks according to the pivot value
void CompareSplitBlocks (double* pFirstBlock, int &FirstBlockSize,
  double* pSecondBlock, int &SecondBlockSize, double Pivot) {
  int TotalSize = FirstBlockSize + SecondBlockSize;
  double* pTempBlock = new double [TotalSize];
  int LastMin = 0, FirstMax = TotalSize - 1;
  for (int i=0; i<FirstBlockSize; i++) {
    if (pFirstBlock[i]<Pivot)
      pTempBlock[LastMin++] = pFirstBlock[i];
    else
      pTempBlock[FirstMax--] = pFirstBlock[i];
  }
  for (int i=0; i<SecondBlockSize; i++) {
    if (pSecondBlock[i]<Pivot)
      pTempBlock[LastMin++] = pSecondBlock[i];
    else
      pTempBlock[FirstMax--] = pSecondBlock[i];
  }
  FirstBlockSize = LastMin;
  SecondBlockSize = TotalSize - LastMin;
  for (int i=0; i<FirstBlockSize; i++)
    pFirstBlock[i] = pTempBlock[i];
  for (int i=0; i<SecondBlockSize; i++)
    pSecondBlock[i] = pTempBlock[FirstBlockSize+i];
  delete [] pTempBlock;
}
```

10.4.2.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода быстрой сортировки проводились при условиях, указанных в п. 10.2.1.4. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 10.10. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.10. Результаты вычислительных экспериментов для параллельного метода быстрой сортировки (при использовании двух вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
100000	0,0374	0,0281	1,3329
200000	0,0778	0,0601	1,2956
300000	0,1218	0,0907	1,3427
400000	0,1674	0,1232	1,3589
500000	0,2158	0,1566	1,3775
600000	0,2716	0,1911	1,4214
700000	0,3100	0,2248	1,3788
800000	0,3679	0,2624	1,4023
900000	0,4175	0,2993	1,3950
1000000	0,4718	0,3347	1,4098

Рис. 10.12. Зависимость ускорения от количества исходных данных при выполнении параллельного метода быстрой сортировки

В таблице 10.11 и на рис. 10.13 представлены результаты сравнения времени выполнения параллельного метода быстрой сортировки с использованием двух потоков со временем, полученным при помощи модели (10.20). Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах исходного массива составляет не более 6%.

Таблица 10.11. Сравнение экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки с использованием двух потоков

Размер массива	Эксперимент	Время счета ($T_p(calc)$)	Время доступа к памяти ($T_p(mem)$)	Модель (T_p)
100000	0,0281	0,0213	0,0046	0,0259
200000	0,0601	0,0453	0,0097	0,0551
300000	0,0907	0,0703	0,0151	0,0854
400000	0,1232	0,0960	0,0206	0,1166
500000	0,1566	0,1221	0,0262	0,1483
600000	0,1911	0,1486	0,0319	0,1805
700000	0,2248	0,1754	0,0376	0,2131
800000	0,2624	0,2025	0,0434	0,2460
900000	0,2993	0,2299	0,0493	0,2792
1000000	0,3347	0,2575	0,0552	0,3127

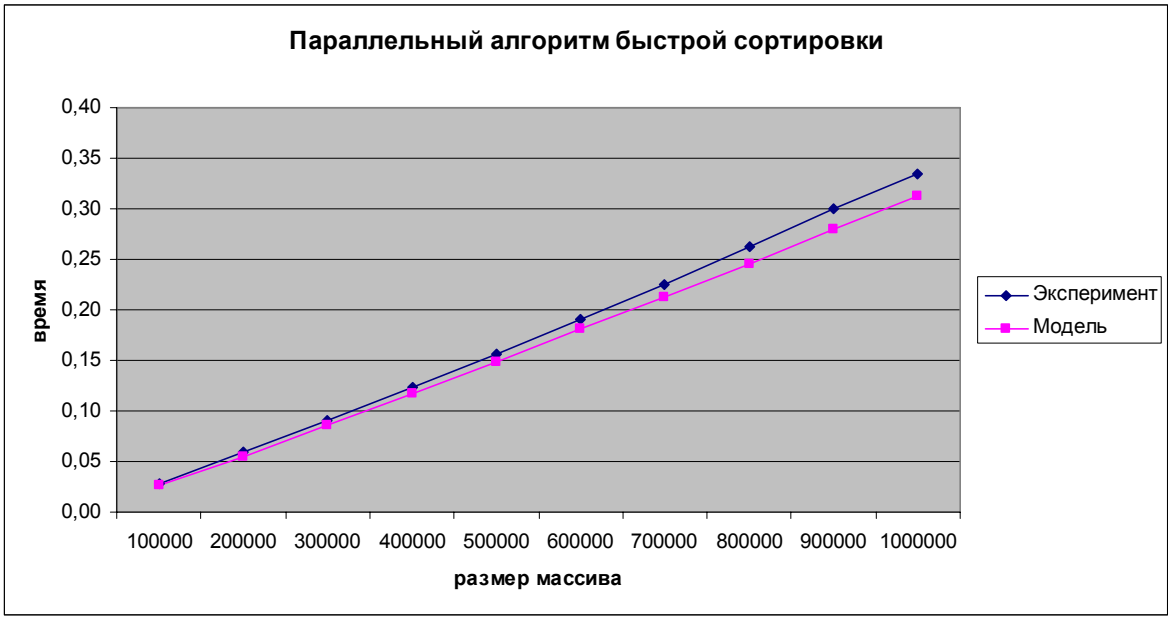


Рис. 10.13. График зависимости экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки от объема исходных данных при использовании двух потоков

10.4.3. 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 блоков данных выполняется в самом начале выполнения вычислений. Кроме того, для поддержки упорядоченности в ходе вычислений над блоками данных выполняется операция слияния, а затем деление полученного блока двойного размера согласно ведущему элементу. Как результат, в силу упорядоченности блоков, при выполнении алгоритма быстрой сортировки в качестве ведущего элемента целесообразнее будет выбирать средний элемент какого-либо блока. 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).

10.4.3.1. Анализ эффективности

При анализе эффективности обобщенного алгоритма можно воспользоваться соотношением (10.20). Следует только учесть, что на каждой итерации метода теперь выполняется операция слияния блоков (будем, как и ранее, предполагать, что их размер одинаков и равен $(n/2p)$). Кроме того, на каждой итерации метода создаются две параллельные секции: одна для выбора ведущих элементов, вторая для выполнения слияния блоков. С учетом всех высказанных замечаний трудоемкость обобщенного алгоритма быстрой сортировки может быть выражена при помощи соотношения следующего вида:

$$T_p = 1.4 \cdot \left[[(n/p)\log_2(2p) + 2 \cdot 1{,}4(n/2p)\log_2(n/2p)]\tau + \left(\log_2(2p) + 1.4\log_2(n/2p)\right) \cdot \frac{8n}{\beta} \right] + \left(2\left(\log_2 p + 1\right) + 1\right) \cdot \delta \cdot \quad (10.21)$$

10.4.3.2. 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.

Как и ранее, введем глобальные переменные: *ThreadNum* для определения количества потоков в параллельной программе, *DimSize* для определения размерности гиперкуба, который может быть составлен из заданного количества блоков данных, *ThreadID* для определения номера текущего потока, *PairNum*, *FirstBlock* и *SecondBlock* для определения номера пары блоков, которую данный поток должен обработать на данной итерации, а также конкретных номеров блоков данных потока в структуре виртуального гиперкуба, *GroupID* для определения номера группы блоков, к которой принадлежит данный данная пара на текущей

итерации обмена данными (номер группы блоков позволяет определить индекс ведущего элемента, который данный поток должен использовать для операции сравнения и разделения блоков на текущей итерации алгоритма). На первой итерации все блоки входят в одну группу, номер этой группы равен 0, на второй итерации происходит разделение исходного виртуального гиперкуба на два подгиперкуба меньшей размерности. При этом половина блоков (блоки, в битовом представлении номеров которых старший бит равен 0) формируют группу с номером 0, а другая половина блоков – группу с номером 1, и т.д. Создадим локальные копии переменных, значения которых различаются в разных потоках, при помощи директивы *threadprivate*.

```
int ThreadNum;  // Number of threads
int ThreadID;   // Thread identifier
int DimSize;    // Number of dimension in hypercube, assembled of data blocks
int MyPair;     // Number of the block pair, that should be processed
                // by current thread
int FirstBlock; // Number of the first data block in pair
int SecondBlock;// Number of the second data block in pair
int GroupID;    // Number of the group current thread belongs to

#pragma omp threadprivate (ThreadID, MyPair, FirstBlock, \
  SecondBlock, GroupID)
```

Необходимо пояснить схему выбора ведущих элементов. Один из блоков каждой группы должен задать значение ведущего элемента, который будет использоваться для обработки всех блоков данной группы. Как описано выше, в качестве ведущего значения выбирается средний элемент блока. Для выбора ведущих элементов на каждой итерации алгоритма организуется цикл по количеству групп (на первой итерации алгоритма все блоки данных входят в одну группу, на второй итерации общее количество блоков разделяется на 2 группы, на третьей итерации – 4 группы и так далее). На каждой итерации этого цикла задается значение ведущего элемента для одной группы – в качестве такого элемента выбирается среднее значение блока с минимальным номером, входящего в группу.

```
// Function for parallel hyperquick sorting
void ParallelHyperQuickSort (double* pData, int Size) {
  InitializeParallelSections();
  double ** pTempData = new double * [2*ThreadNum];
  int * BlockSizes = new int [2*ThreadNum];
  double* Pivots = new double [ThreadNum];
  int * BlockPairs = new int [2*ThreadNum];
  for (int i=0; i<2*ThreadNum; i++) {
    pTempData[i] = new double [Size];
    for (int j=0; j<Size; j++)
      pTempData[i][j] = DUMMY_VALUE;
    BlockSizes[i] = Size/(2*ThreadNum);
  }
  for (int j=0; j<Size; j++)
    pTempData[2*j*ThreadNum/Size][j%(Size/(2*ThreadNum))] = pData[j];

  // Local sorting of data blocks
#pragma omp parallel
  {
    LocalQuickSort(pTempData[ThreadNum+ThreadID], 0,
      BlockSizes[ThreadNum+ThreadID]-1);
    LocalQuickSort(pTempData[ThreadID], 0, BlockSizes[ThreadID]-1);
  }

  // Iterations of parallel hyperquick sorting algorithm
  for (int i=0; i<DimSize; i++) {
    // Determination of data block pairs
    SetBlockPairs(BlockPairs, i);

#pragma omp parallel for
      for (int j=0; j<(1<<i); j++) {
        int BlockID = (2*ThreadNum*j)/(1<<i);
        Pivots[j] = pTempData[BlockID][BlockSizes[BlockID]/2];
      } // pragma omp parallel for
```

```
    // Carrying out the "compare-split" operation for sorted data blocks
#pragma omp parallel
    {
      MyPair = FindMyPair(BlockPairs, ThreadID, i);
      FirstBlock = BlockPairs[2*MyPair];
      SecondBlock = BlockPairs[2*MyPair+1];
      GroupID = FirstBlock/(1<<(DimSize-i));
      CompareSplitBlocks(pTempData[FirstBlock], BlockSizes[FirstBlock],
        pTempData[SecondBlock], BlockSizes[SecondBlock], Pivots[GroupID]);
    }
  } // for

  int curr = 0;
  for (int i=0; i<2*ThreadNum; i++)
    for (int j=0; j<BlockSizes[i]; j++)
      pData[curr++] = pTempData[i][j];

  for (int i=0; i<2*ThreadNum; i++)
    delete [] pTempData[i];

  delete [] pTempData;
  delete [] BlockSizes;
  delete [] Pivots;
  delete [] BlockPairs;
}
```

Функция *CompareSplitBlocks* в данном случае выполняет слияние и разделение двух упорядоченных блоков:

```
// Function for carrying out the "compare-split" operation
// for two sorted data blocks according to the pivot value
void CompareSplitBlocks(double* pFirstBlock, int &FirstBlockSize,
  double* pSecondBlock, int &SecondBlockSize, double Pivot) {
  int TotalSize = FirstBlockSize + SecondBlockSize;
  double* TempBlock = new double [TotalSize];
  int i=0, j=0, curr=0;
  while ((i<FirstBlockSize)&&(j<SecondBlockSize)) {
    if (pFirstBlock[i]<pSecondBlock[j])
      TempBlock[curr++] = pFirstBlock[i++];
    else
      TempBlock[curr++] = pSecondBlock[j++];
  }
  while (i<FirstBlockSize)
    TempBlock[curr++] = pFirstBlock[i++];
  while (j<SecondBlockSize)
    TempBlock[curr++] = pSecondBlock[j++];
  curr = 0;
  while ((curr<TotalSize) && (TempBlock[curr]<Pivot))
    pFirstBlock[curr] = TempBlock[curr++];
  FirstBlockSize = curr;
  SecondBlockSize = TotalSize - curr;
  while (curr<TotalSize)
    pSecondBlock[curr-FirstBlockSize] = TempBlock[curr++];
  delete [] TempBlock;
}
```

10.4.3.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода обобщенной быстрой сортировки проводились при условиях, указанных в п. 10.2.1.4. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 10.12. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.12. Результаты вычислительных экспериментов для параллельного метода обобщенной быстрой сортировки (при использовании двух вычислительных ядер)

Размер массива	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
100000	0,0374	0,0292	1,2821
200000	0,0778	0,0589	1,3219
300000	0,1218	0,0876	1,3908
400000	0,1674	0,1188	1,4094
500000	0,2158	0,1514	1,4247
600000	0,2716	0,1860	1,4607
700000	0,3100	0,2215	1,3996
800000	0,3679	0,2510	1,4661
900000	0,4175	0,2851	1,4643
1000000	0,4718	0,3205	1,4723

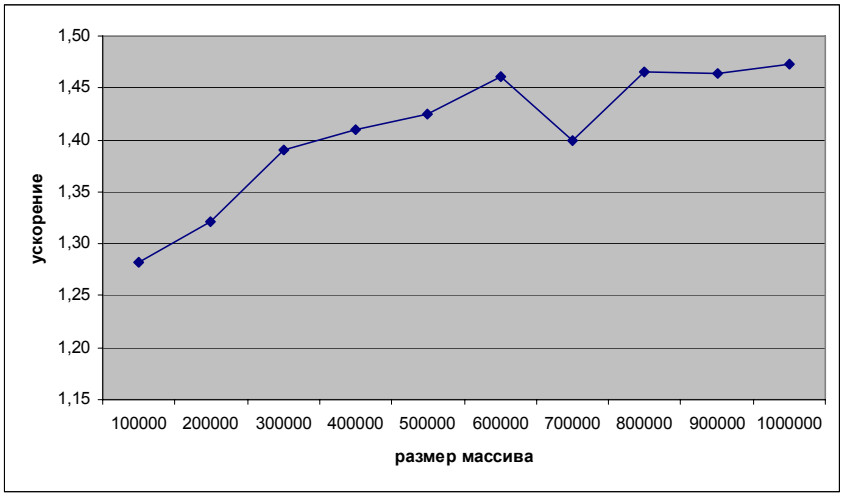


Рис. 10.14. Зависимость ускорения от количества исходных данных при выполнении параллельного метода обобщенной быстрой сортировки

В таблице 10.13 и на рис. 10.15 представлены результаты сравнения времени выполнения параллельного метода обобщенной быстрой сортировки с использованием двух потоков со временем, полученным при помощи модели (10.21). Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах исходного массива составляет не более 2%.

Таблица 10.13. Сравнение экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки с использованием двух потоков

Размер массива	Эксперимент	Время счета ($T_p(calc)$)	Время доступа к памяти ($T_p(mem)$)	Модель (T_p)
100000	0,0292	0,0213	0,0046	0,0260
200000	0,0589	0,0453	0,0097	0,0551
300000	0,0876	0,0703	0,0151	0,0854
400000	0,1188	0,0960	0,0206	0,1166
500000	0,1514	0,1221	0,0262	0,1483
600000	0,1860	0,1486	0,0319	0,1805
700000	0,2215	0,1754	0,0376	0,2131
800000	0,2510	0,2025	0,0434	0,2460
900000	0,2851	0,2299	0,0493	0,2792
1000000	0,3205	0,2575	0,0552	0,3127

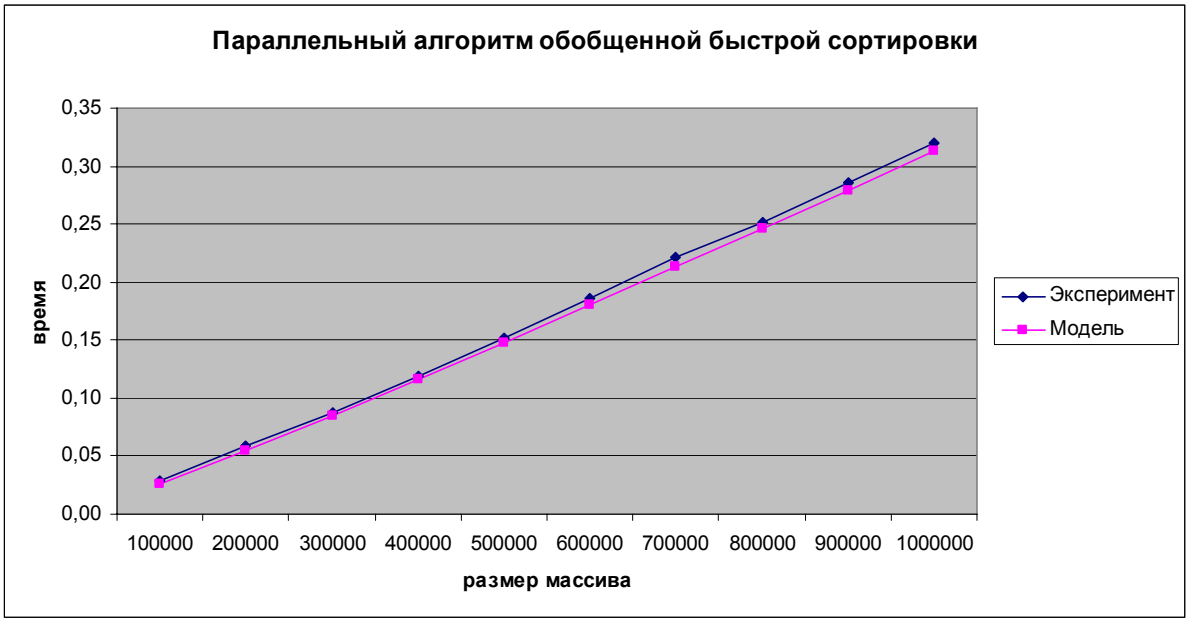


Рис. 10.15. График зависимости экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки от объема исходных данных при использовании двух потоков

10.4.4. Сортировка с использованием регулярного набора образцов

10.4.4.1. 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 блоков данных; данная операция может быть выполнена каждым потоком независимо друг от друга при помощи обычного алгоритма быстрой сортировки; далее каждый поток 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 потоками наборы данных собираются на одном из потоков (*master thread*) системы и сортируются при помощи быстрого алгоритма, таким образом они формируют 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 далее используется всеми потоками; в завершение этапа каждый поток выполняет разделение своего блока на p частей с использованием полученного набора ведущих значений;

- на *третьем этапе* сортировки каждый поток осуществляет «передачу» выделенных ранее частей своего блока всем остальным потокам; «передача» выполняется в соответствии с порядком нумерации - часть $j,\ 0 \le j < p$, каждого блока передается потоку с номером j;
- на *четвертом этапе* выполнения алгоритма каждый поток выполняет слияние p полученных частей в один отсортированный блок.

По завершении четвертого этапа исходный набор данных становится отсортированным.

На рис.10.16 приведен пример сортировки массива данных с помощью алгоритма, описанного выше. Следует отметить, что число потоков для данного алгоритма может быть произвольным, в данном примере оно равно 3.

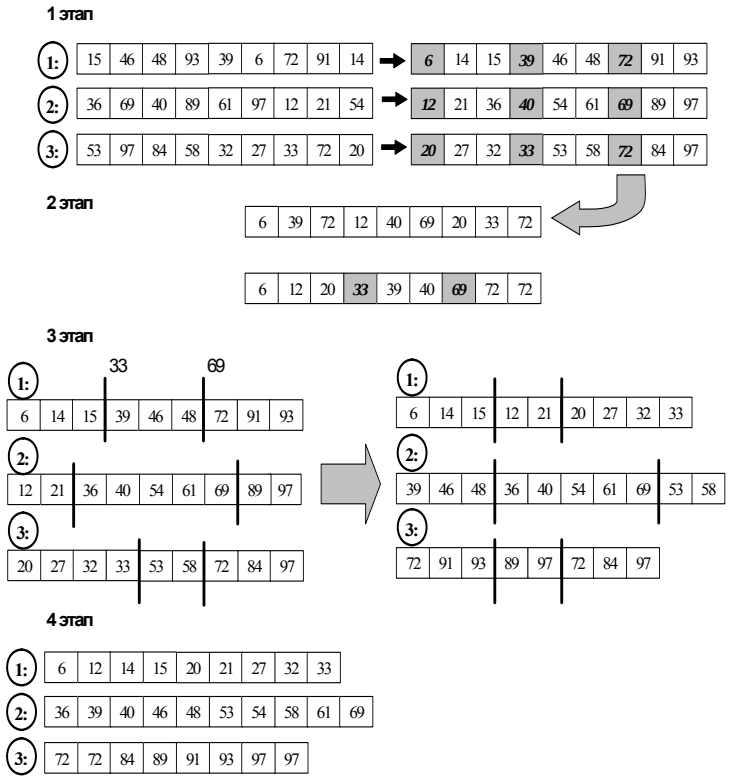


Рис. 10.16. Пример работы алгоритма сортировки с использованием регулярного набора образцов для 3 вычислительных элементов

10.4.4.2. Анализ эффективности

Оценим трудоемкость рассмотренного параллельного метода. Пусть, как и ранее, n есть количество сортируемых данных, p, $p<n$, обозначает число используемых вычислительных элементов и, соответственно, n/p есть размер блоков данных, которые обрабатываются параллельными потоками.

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

$$T_p^1 = 1{,}4\cdot(n/p)\log_2(n/p)\,\tau + 1{,}4\cdot\log_2(n/p)\cdot\frac{8n}{\beta}, \qquad (10.22)$$

где τ есть время выполнения базовой операции сортировки, β – эффективная скорость доступа к оперативной памяти.

На *втором этапе* алгоритма один из потоков (*master thread*) собирает наборы из p элементов со всех остальных процессоров, выполняет сортировку всех полученных данных (общее количество элементов составляет p^2), формирует набор из $p-1$ ведущих элементов. Поскольку в общем случае число вычислительных элементов в системе, а следовательно и число потоков невелико, но массив из p^2 элементов может быть полностью размещен в кэш, и, следовательно, дополнительных затрат на чтение данных из оперативной памяти не требуется. С учетом всех перечисленных действий общая длительность второго этапа составляет

$$T_p^2 = 1{,}4\cdot p^2\cdot\log_2\left(p^2\right). \qquad (10.23)$$

В ходе выполнения *третьего этапа* алгоритма каждый процессор разделяет свои элементы относительно ведущих элементов на p частей (поиск очередного места разбиения можно осуществить при помощи алгоритма бинарного поиска)

$$T_p^3 = p\log_2(n/p)\cdot\tau + \frac{8n}{\beta}. \qquad (10.24)$$

На *четвертом этапе* алгоритма каждый процессор выполняет слияние p отсортированных частей в один объединенный блок. Оценка трудоемкости такой операции составляет:

$$T_p^4 = p\cdot\left(1{,}4\frac{n}{p^2}\right)\cdot\tau + 2\cdot\frac{8n}{\beta} = 1{,}4\frac{n}{p}\cdot\tau + 2\cdot\frac{8n}{\beta}. \qquad (10.25)$$

Для выполнения каждого этапа создается параллельная секция; следует учитывать накладные расходы на их организацию и закрытие. С учетом всех полученных соотношений, общее время выполнения алгоритма сортировки с использованием регулярного набора образцов составляет

$$T_p = \left[(1{,}4(n/p)+p)\cdot\log_2(n/p)+1{,}4p^2\log_2(p^2)+1{,}4(n/p)\right]\cdot\tau+\left[1{,}4\log_2(n/p)+3\right]\cdot\frac{8n}{\beta}+4\delta. \quad (10.26)$$

10.4.4.3. Программная реализация

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

Как и ранее, переменные *ThreadNum* и *ThreadID* отвечают за хранение количества потоков в параллельной программе и идентификатора потока соответственно. Эти переменные объявлены как глобальные, для переменной *ThreadID* созданы локальные копии при помощи директивы *threadprivate*. Переменные проинициализированы в функции *InitializeParallelSections*.

Поясним применение дополнительных структур данных. Массив указателей *pDataBlock* хранит указатели на начала блоков в массиве *pData*, которые обрабатываются параллельными потоками. Размеры блоков всех потоков хранятся в массиве *BlockSize*. Таким образом, начало блока, который обрабатывается потоком с номером *i*, расположено по адресу *pDataBlock[i]*, и количество элементов в этом блоке равно *BlockSize[i]*.

Массив *LocalSamples* хранит набор из p^2 "локальных" ведущих элементов, выбранных потоками. Поток с номером *i* осуществляет запись своих локальных ведущих элементов в ячейки с номерами от *i·p* до *(i+1)·p-1*. После сортировки из массива *LocalSamples* выбираются глобальные ведущие элементы и сохраняются в массиве *GlobalSamples*.

Двумерный массив указателей *pDataSubBlock* служит для разделения блоков каждого процесса на подблоки согласно глобальному набору образцов. Указатели на начала подблоков в блоке, за обработку которого отвечает процесс с номером *i*, хранятся в переменных *pDataSubBlock [i][0]*, …, *pDataSubBlock [i][p-1]*. Размеры подблоков хранятся в двумерном массиве *SubBlockSize*.

Массив *pMergeDataBlock* служит для выполнения операции слияния упорядоченных подблоков.

При выполнении слияния упорядоченных подблоков в каждом потоке используется дополнительный массив *MergeSubBlockSizes*, количество элементов в котором равно количеству параллельных потоков. Перед началом операции слияния в массив заносятся размеры подблоков, предназначенных для слияния на данном потоке. Всякий раз, когда в новый упорядоченный массив добавляется новое значение из подблока какого-либо потока, соответствующий элемент массива *MergeSubBlockSizes* уменьшается на 1. Для поиска наименьшего текущего элемента во всех подблоках, над которыми выполняется операция слияния, используется функция *FindMin*.

```
// Function for parallel quick sorting with regular samples
void ParallelRegularSamplesQuickSort (double* pData, int Size)
{
  InitializeParallelSections();

  double** pDataBlock = new double * [ThreadNum];
  int* BlockSize = new int [ThreadNum];
  double* LocalSamples = new double [ThreadNum*ThreadNum];
  double* GlobalSamples = new double [ThreadNum-1];
  double*** pDataSubBlock = new double ** [ThreadNum];
  int** SubBlockSize = new int* [ThreadNum];
  double** pMergeDataBlock = new double* [ThreadNum];

  for (int i=0; i<ThreadNum; i++) {
    BlockSize[i] = Size/ThreadNum;
    pDataBlock[i] = &pData[i*Size/ThreadNum];
    pMergeDataBlock[i] = new double [Size];
    pDataSubBlock [i] = new double* [ThreadNum];
    SubBlockSize[i] = new int [ThreadNum];
  }

// Local sorting of data blocks
#pragma omp parallel
  {
    SerialQuickSort(pDataBlock[ThreadID], BlockSize[ThreadID]);
```

```
  }

  // Samples determination
#pragma omp parallel
  {
    for (int i=0; i<ThreadNum; i++)
      LocalSamples[ThreadID*ThreadNum+i] =
        pDataBlock[ThreadID][i*Size/(ThreadNum*ThreadNum)];
  }

  // Sorting of local samples set
  SerialQuickSort(LocalSamples, ThreadNum*ThreadNum);

  // Global samples determination
  for (int i=1; i<ThreadNum-1; i++)
    GlobalSamples[i-1] = LocalSamples[i*ThreadNum + (ThreadNum/2)-1];
  GlobalSamples[ThreadNum-2] =
    LocalSamples[(ThreadNum-1)*ThreadNum + (ThreadNum/2)];

  // Splitting of data blocks in accordance with global samples
#pragma omp parallel
  {
    pDataSubBlock[ThreadID][0] = pDataBlock[ThreadID];
    int Pos = 0, OldPos = 0;
    for (int i=0; i<ThreadNum-1; i++) {
      OldPos = Pos;
      Pos = BinaryFindPos(pDataBlock[ThreadID], Pos,
        Size/ThreadNum-1, GlobalSamples[i]);
      pDataSubBlock[ThreadID][i+1] = &pDataBlock[ThreadID][Pos];
      SubBlockSize[ThreadID][i] = Pos-OldPos;
    }
    SubBlockSize[ThreadID][ThreadNum-1] = Size/ThreadNum - Pos;
  }

  // Each thread performs the merging of corresponding subblocks
#pragma omp parallel
  {
    int curr = 0;
    double ** pCurr = new double* [ThreadNum];
    int* MergeSubBlockSizes = new int [ThreadNum];
    for (int i=0; i<ThreadNum; i++) {
      pCurr[i] = pDataSubBlock[i][ThreadID];
      MergeSubBlockSizes [i] = SubBlockSize[i][ThreadID];
    }
    while (!IsMergeEnded(MergeSubBlockSizes, ThreadNum)) {
      int MinPos = FindMin(pCurr, ThreadNum, MergeSubBlockSizes);
      pMergeDataBlock[ThreadID][curr] = *(pCurr[MinPos]);
      MergeSubBlockSizes[MinPos]--;
      if (MergeSubBlockSizes[MinPos] != 0)
        pCurr[MinPos]++;
      curr++;
    } // while
    BlockSize[ThreadID] = curr;
    delete [] pCurr;
    delete [] MergeSubBlockSizes;
  } // pragma omp parallel

  // Copying the data from the pMergeDataBlock arrays to the initial array
  int NewCurr = 0;
  for (int i=0; i<ThreadNum; i++)
    for (int j=0; j<BlockSize[i]; j++)
      pData[NewCurr++] = pMergeDataBlock[i][j];

  for (int i=0; i<ThreadNum; i++) {
```

```
    delete [] pDataSubBlock [i];
    delete [] pMergeDataBlock[i];
    delete [] SubBlockSize[i];
  }
  delete [] pDataBlock;
  delete [] pDataSubBlock;
  delete [] pMergeDataBlock;
  delete [] SubBlockSize;
  delete [] BlockSize;
  delete [] LocalSamples;
  delete [] GlobalSamples;
}
```

Для разделения блоков потоков на подблоки согласно глобальному регулярному набору образцов используется функция *BinaryFindPos*, которая осуществляет бинарный поиск заданного элемента *Elem* в упорядоченном массиве *Array*, начиная с элемента с индексом *first* и заканчивая элементом с индексом *last*. В качестве возвращаемого значения выступает номер позиции, на которой должен быть расположен искомый элемент с тем, чтобы массив *Array* остался упорядоченным.

```
// Function for binary searching
int BinaryFindPos (double* Array, int first, int last, double Elem) {
  if (Elem<Array[first]) return first;
  if (Elem>Array[last]) return (last);
  int middle;
  while (last-first > 1) {
    middle = (first+last)/2;
    if (Array[middle] == Elem)
      return middle;
    if (Array[middle]>Elem) last = middle;
    if (Array[middle]<Elem) first = middle;
  }
  return last;
}
```

10.4.4.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода быстрой сортировки с использованием регулярного набора образцов проводились при условиях, указанных в п. 10.2.1.4. Результаты вычислительных экспериментов приведены в таблице 10.14. Использовался вычислительный узел на базе процессора Intel Core 2 6300, 1.87 ГГц, кэш L2 2 Мб, 2 Гб RAM под управлением операционной системы Microsoft Windows XP Professional. Разработка программ проводилась в среде Microsoft Visual Studio 2005, для компиляции использовался Intel C++ Compiler 10.0 for Windows. Результаты вычислительных экспериментов приведены в таблице 10.12. Времена выполнения алгоритмов указаны в секундах.

Таблица 10.14. Результаты вычислительных экспериментов для параллельного метода быстрой сортировки с использованием регулярного набора образцов (p=2)

Размер массива	Последовательный алгоритм	Параллельный алгоритм	
		Время	Ускорение
100000	0,0374	0,0267	1,4004
200000	0,0778	0,0554	1,4047
300000	0,1218	0,0800	1,5222
400000	0,1674	0,1138	1,4712
500000	0,2158	0,1403	1,5374
600000	0,2716	0,1500	1,8112
700000	0,3100	0,1818	1,7053
800000	0,3679	0,2099	1,7531
900000	0,4175	0,2320	1,7993
1000000	0,4718	0,2619	1,8018

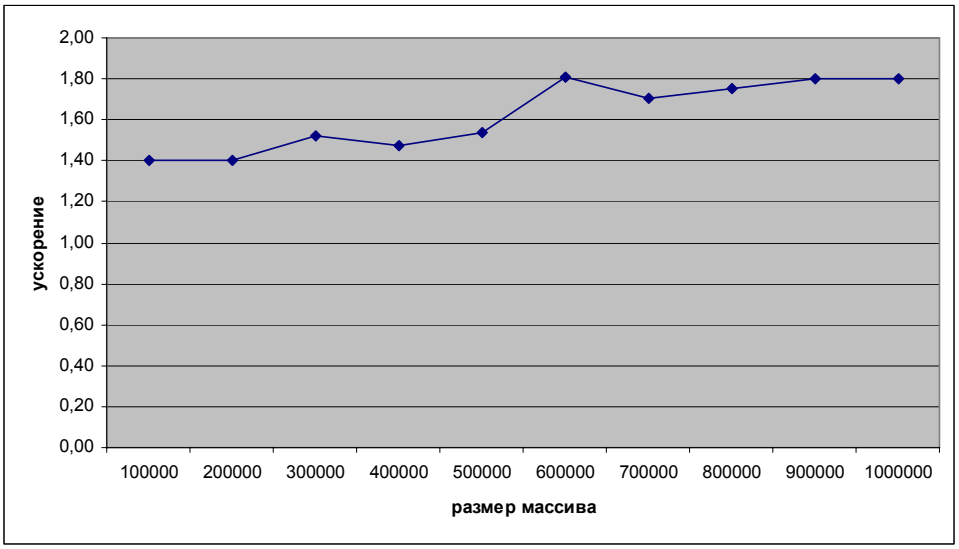


Рис. 10.17. Зависимость ускорения от количества исходных данных при выполнении параллельного метода быстрой сортировки с использованием регулярного набора образцов

В таблице 10.15 и на рис. 10.18 представлены результаты сравнения времени выполнения параллельного метода обобщенной быстрой сортировки с использованием двух потоков со временем, полученным при помощи модели (10.26). Как видно из приведенных данных, относительная погрешность оценки убывает с ростом объема сортируемых данных и при достаточно больших размерах исходного массива составляет около 10%. Такое существенное расхождение экспериментальных данных и теоретических оценок объясняется особенностями реализации алгоритма.

Таблица 10.15. Сравнение экспериментального и теоретического времени выполнения параллельного метода обобщенной быстрой сортировки с использованием двух потоков

Размер массива	Эксперимент	Время счета ($T_p(calc)$)	Время доступа к памяти ($T_p(mem)$)	Модель (T_p)
100000	0,0267	0,0158	0,0036	0,0194
200000	0,0554	0,0335	0,0076	0,0411
300000	0,0800	0,0518	0,0118	0,0637
400000	0,1138	0,0707	0,0161	0,0868
500000	0,1403	0,0899	0,0204	0,1104
600000	0,1500	0,1094	0,0248	0,1343
700000	0,1818	0,1291	0,0293	0,1585
800000	0,2099	0,1490	0,0338	0,1829
900000	0,2320	0,1691	0,0383	0,2075
1000000	0,2619	0,1893	0,0429	0,2323

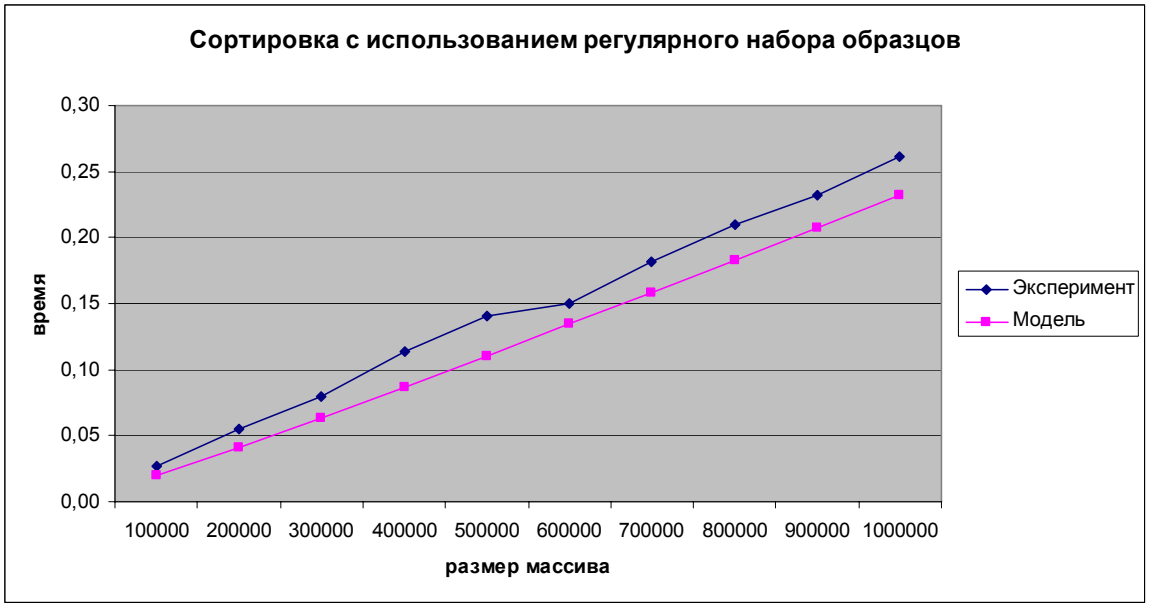


Рис. 10.18. График зависимости экспериментального и теоретического времени выполнения параллельного метода быстрой сортировки с использованием регулярного набора образцов от объема исходных данных при использовании двух потоков

10.5. 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 10.2) 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 10.3) рассматривается схема распараллеливания при представлении множества потоков параллельной программы в виде гиперкуба. При таком представлении оказывается возможным организация взаимодействия потоков и выполнить операции сравнения и разделения подблоков, 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 10.4) приводятся три схемы распараллеливания. Первые две схемы также основываются на представлении множества потоков параллельной программы в виде гиперкуба. Основная итерация вычислений состоит в выборе одним из потоков ведущего элемента. После получения ведущего элемента потоки проводят разделение своих блоков, и получаемые части блоков передаются между попарно связанными потоками. 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 блоки данных 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, на основании информации об максимальном и минимальном значениях в сортируемом наборе. Во второй схеме блоки данных предварительно упорядочиваются с тем, чтобы взять средний элемент блока 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 потоков и приводит, как правило, к лучшей балансировке распределения данных между процессорами.

Сравнение показателей ускорения различных параллельных алгоритмов сортировки в зависимости от объема исходных данных представлено на рис. 10.19.

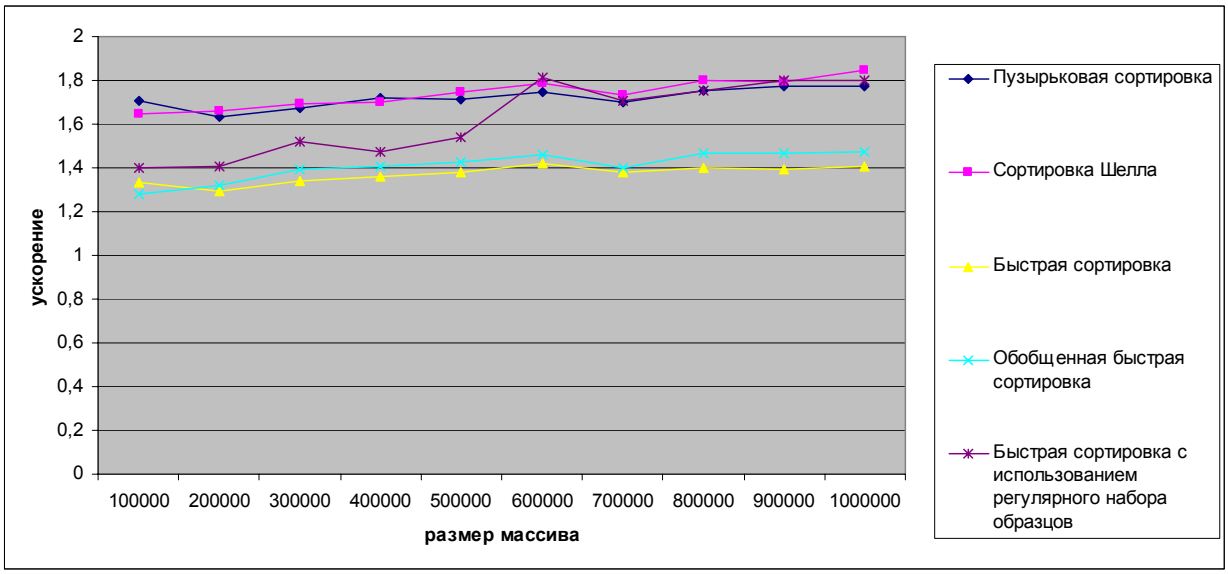


Рис. 10.19. Ускорение параллельных алгоритмов сортировки данных

10.6. 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).

10.7. 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. В чем состоит алгоритм сортировки с использованием регулярного набора образцов?

10.8. 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")