ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ СОРТИРОВКИ БОЛЬШИХ ОБЪЕМОВ ДАННЫХ

© М.В.Якобовский, 2004, 2008
mail: lira@imamod.ru
http://lira.imamod.ru

Введение

Сортировка больших объемов данных используется при численном моделировании широкого круга задач, например, при решении систем линейных уравнений большой размерности, при разбиении графов, описывающих двух- и трехмерных расчетные сетки, при сжатии сеточных функций – результатов выполненных крупномасштабных вычислительных экспериментов и так далее. Существенную роль эти алгоритмы играют при кэшировании геометрических данных, описывающих пространственные тела сложной конфигурации. Большинство алгоритмов сортировки не поддается распараллеливанию, но могут быть использованы на первом этапе рассматриваемого далее хорошо масштабируемого метода ориентированного на многопроцессорные системы с распределенной памятью.

Постановка задачи

Расположить в порядке не убывания n элементов массива чисел, равномерно размещенных на p процессорах. По окончании сортировки на процессорах с меньшими номерами должны быть размещены элементы массива с меньшими значениями.

Сортировке данных посвящена обширная литература. Монография Кнута [1] содержит подробное описание и анализ огромного количества различных алгоритмов. Далее некоторые из них (быстрая, пирамидальная сортировка, сортировка методом слияния списков, «обменной сортировки со слиянием» Бэтчера) используются в качестве базовых. На их основе конструируются последовательный и параллельный алгоритмы сортировки. Основную цель конструирования новых алгоритмов можно определить следующим образом: уменьшение общего времени сортировки массива. Кроме того, преследуется цель создания параллельных алгоритма и программы, позволяющих выполнять сортировку данных, объем которых превышает оперативную память каждого из используемых процессорных узлов.

Рассматриваемые далее параллельные алгоритмы предполагают двухэтапную сортировку:

- последовательную сортировку фрагментов массива, распределенных по процессорам системы;

- объединение упорядоченных фрагментов массива – перемещение элементов массива между процессорами.

Для уменьшения общего времени выполнения сортировки следует по возможности сократить время выполнения каждого из указанных этапов, поэтому в начале обсуждается последовательный алгоритм, показавший наименьшее время сортировки тестовых массивов. Именно относительно этого алгоритма определяется эффективность алгоритмов параллельной сортировки. Таким образом, в работе используется метод определения эффективности относительно «наилучшего» из имеющихся в распоряжении последовательных алгоритмов сортировки.

Время сортировки массива зависит от множества факторов, среди которых не последнее место занимает характер распределения чисел в сортируемом массиве (степень его упорядоченности перед началом сортировки). В качестве тестовых данных используются различные массивы целых четырехбайтовых чисел:

- псевдослучайные неупорядоченные числа;
- числа, расположенные по не возрастанию;
- числа, расположенные по не убыванию.

Указанный набор тестов, не являясь исчерпывающим, позволил создать достаточно эффективные алгоритмы и процедуры.

Далее, везде приняты следующие обозначения:

n - число элементов в сортируемом массиве;

p - число процессоров;

$T(n, p)$ - общее время сортировки массива из n элементов на p процессорах;

$M(n, p)$ - общее число условных операций, необходимых для сортировки массива из n элементов на p процессорах.

Подробное описание алгоритмов быстрой, пирамидальной и «обменной сортировки со слиянием» Бэтчера, четно-нечетного слияния, а также сортировки слиянием списков можно найти в монографии [1], там же приведен анализ соответствующих алгоритмов, необходимые доказательства и оценка объема выполняемых алгоритмами действий.

В таблице 1 приведены асимптотические оценки зависимости количества операций, необходимых для сортировки с помощью ряда алгоритмов, от размера сортируемого массива (в скобках указан первоисточник данных). Поскольку реальное время выполнения значительно зависит от конкретной реализации алгоритма и от свойств используемой вычислительной системы, приведенные оценки не являются достаточным основанием для выбора того или иного алгоритма в качестве «наилучшего». Мотивированный выбор алгоритма и его конкретной реализации описывается в следующем разделе, посвященном тестированию последовательных алгоритмов.

Таблица 1

Зависимость количества операций от размера массива

Алгоритм сортировки	Среднее время	максимальное время	эксперимент
Быстрая (***qsort***)	11.7 $n \log_2 n$ [1]	$O(n^2)$	
Пирамидальная (***hsort***)	16 $n \log_2 n$ [1]	18 $n \log_2 n$+ 38n [1]	3 $n \log_2 n$
Слияние списков (***lsort***)	10 $n \log_2 n$ [1]	$O(n \log_2 n)$ [1]	
Параллельная сортировка сдваиванием	$O\left(\frac{n}{p}\left[\log_2\left(\frac{n}{p}\right)+p\right]\right)$		
Параллельная «обменная сортировка со слиянием»	$O\left(\frac{n}{p}\left[\log_2\frac{n}{p}+\frac{\lceil\log_2 p\rceil^2}{2}\right]\right)$		

Приведенный в монографии [1] анализ дает оценку числа операций (Табл. 1) необходимых для сортировки. Именно ***числа операций***, а не ***времени выполнения*** процедуры сортировки на реальной вычислительной системе. Однако, как показывает практика, между числом операций и временем выполнения программы не всегда есть линейная зависимость. В частности, выражение вида $T(n)=O(f(n))$ не несет информации о том, начиная с какого n можно на практике при оценке величины $T(n)$ ориентироваться на значения функции $f(n)$. При одном и том же асимптотическом поведении кривых *F* и *G* (Рис. 1) их значения в точке *x*0 значительно отличаются. К реальному значению функций *F* и *G* в окрестности точки *x*0 асимптотика отношения не имеет, и ориентироваться на нее, при оценке значений функций около этой точки не следует. Как показано далее, подобная ситуация складывается при оценке времени сортировки больших объемов данных. Выяснение причин такого явления выходит за рамки рассматриваемых вопросов, однако его наличие – факт установленный и с ним следует считаться при построении эффективных последовательных и параллельных алгоритмов.

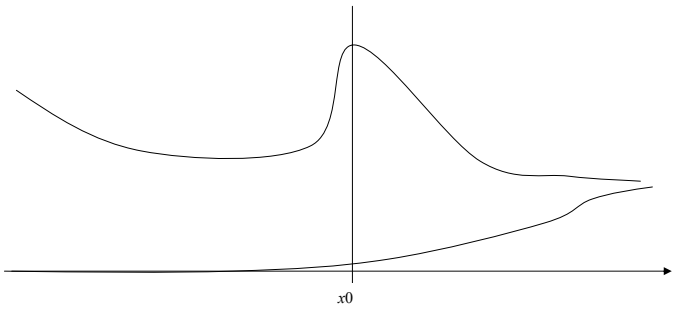

Рис. 1. Сравнение функций

Следует обратить внимание еще на одно обстоятельство. Для тестирования алгоритмов используются последовательности псевдослучайных чисел. Они могут быть получены с помощью стандартного генератора, входящего в состав библиотеки времени выполнения. Последовательности, формируемые этим генератором, вообще говоря, зависят и от разрядности используемой вычислительной системы и от особенностей реализации. В Таблице 2 приведены первые 10 чисел, полученных с

помощью функции ***rand*** при компиляции и выполнении тестовой программы на двух разных вычислительных системах.

Таблица 2.

Псевдослучайные числа, полученные функцией ***rand***()

Windows, VisualC 6.0	Linux, gcc 3.2.2
RAND_MAX = 32767	RAND_MAX = 2147483647
41	1804289383
18467	846930886
6334	1681692777
26500	1714636915
19169	1957747793
15724	424238335
11478	719885386
29358	1649760492
26962	596516649
24464	1189641421

Столь разительные отличия влекут за собой как минимум два неприятных следствия:

1. использование стандартного генератора не позволяет адекватно сравнивать работу алгоритмов сортировки на разных вычислительных системах.
2. использование генератора псевдослучайных чисел с маленьким диапазоном их значений (0 … 32767) приведет к тому, что в массиве большого размера (10^8 элементов и более) будет много одинаковых элементов, что так же не позволит оценить параметры изучаемых алгоритмов.

Во избежание указанных эффектов предлагается использовать для формирования последовательности псевдослучайных чисел алгоритм [3,4]. Его использование гарантирует воспроизводимость результатов и обеспечивает генерацию чисел в достаточном для проведения тестов диапазоне значений.

Последовательные алгоритмы сортировки

Пирамидальная сортировка

На Рис. 2 приведены результаты замера времени обработки массивов с помощью алгоритма пирамидальной сортировки ***hsort***. Число операций, необходимых для сортировки массива с его помощью можно оценить как $M(n) = R(n) n \log_2(n)$.

Численный эксперимент полностью подтверждает эту оценку, как при сортировке неупорядоченных, так и при сортировке упорядоченных массивов. При этом экспериментальное значение $R(n) \approx 2.7$. Под числом операций при экспериментальном определении $R(n)$ понимается суммарное число операций сравнения между собой и перемещения элементов массива. Таким образом, число действий, выполняемых при сортировке массива с помощью алгоритма пирамидальной сортировки, действительно составляет порядка $O(n \log_2(n))$. Совершенно иначе выглядит кривая зависимости реального времени выполнения пирамидальной сортировки от размера массива (Рис. 2).

При обработке неупорядоченных массивов, число элементов в которых не превышает 10^5, наблюдается хорошее соответствие времени выполнения ожидаемой оценке.

$$T(n) = K(n) 10^{-9} n \log_2(n),$$

$$K(n) \approx 21$$

Однако, при дальнейшем увеличении размера сортируемого массива (с 10^5 до 10^8 элементов), коэффициент $K(n)$ возрастает более чем в три раза: с 21 до 65. Несмотря на то, что число операций возрастает пропорционально $n\log_2(n)$, время выполнения растет быстрее чем $n\log_2(n)$.

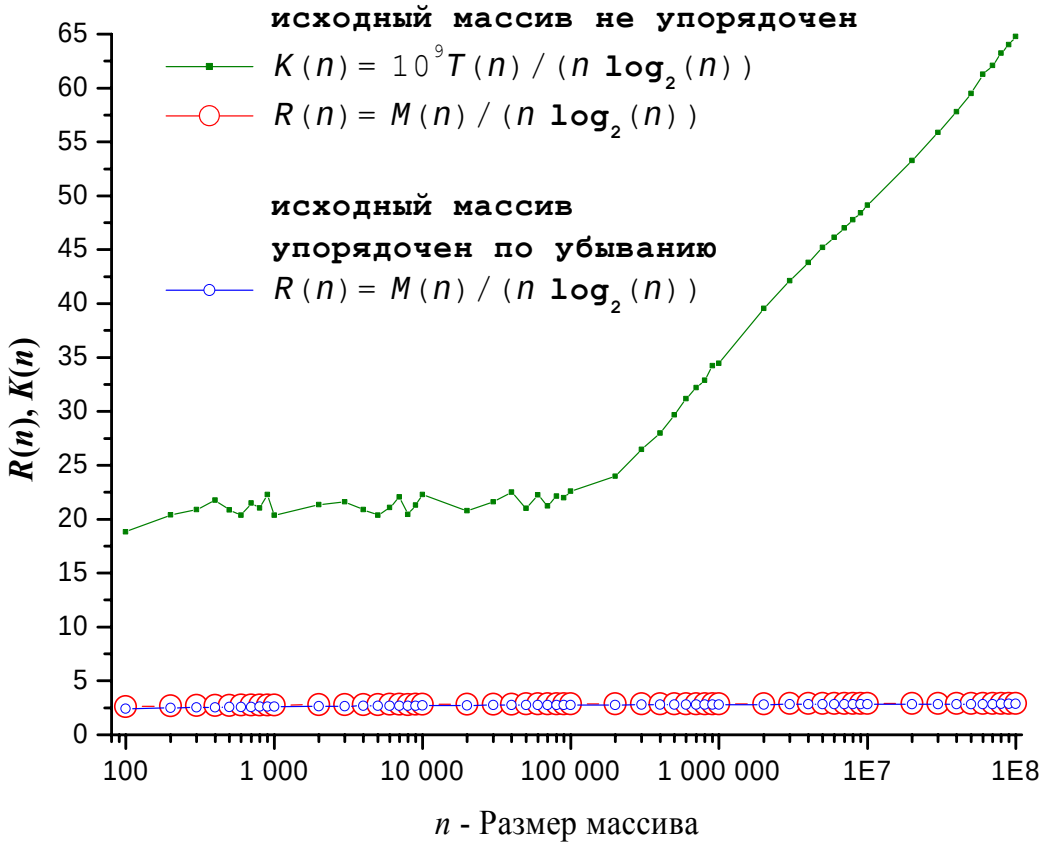

Рис. 2. Пирамидальная сортировка

Разница в числовых значениях $R(n)$, полученных в эксперименте и приведенных в монографии [1] можно объяснить тем, что, кроме операций сравнения и перемещения элементов выполняется некоторый ряд вспомогательных действий, число которых оценивается в монографии [1], но не подсчитывается в проведенном численном эксперименте.

Сортировка слиянием списков

Аналогичный результат наблюдается при использовании алгоритма сортировки слиянием списков ***lsort***. Согласно монографии [1] этот алгоритм обладает наименьшей, по сравнению с другими, мультипликативной константой при $n\log_2(n)$. Сортировка с его помощью требует, таким образом, выполнения меньшего числа операций. Однако, рост времени выполнения относительно $n\log_2(n)$ оказывается настолько значительным при больших значениях n (Рис. 3), что сводит на нет потенциальные преимущества алгоритма ***lsort***. Опять же, речь не идет о том, плох или хорош сам алгоритм сортировки слиянием списков. Нельзя так же утверждать, что неудачна конкретная программная реализация. Можно лишь утверждать, что неудачно сочетание конкретной реализации этого алгоритма, конкретной вычислительной системы,

установленной на ней операционной системы, компилятора, и опции компиляции или любого из перечисленных компонент, список которых можно продолжить.

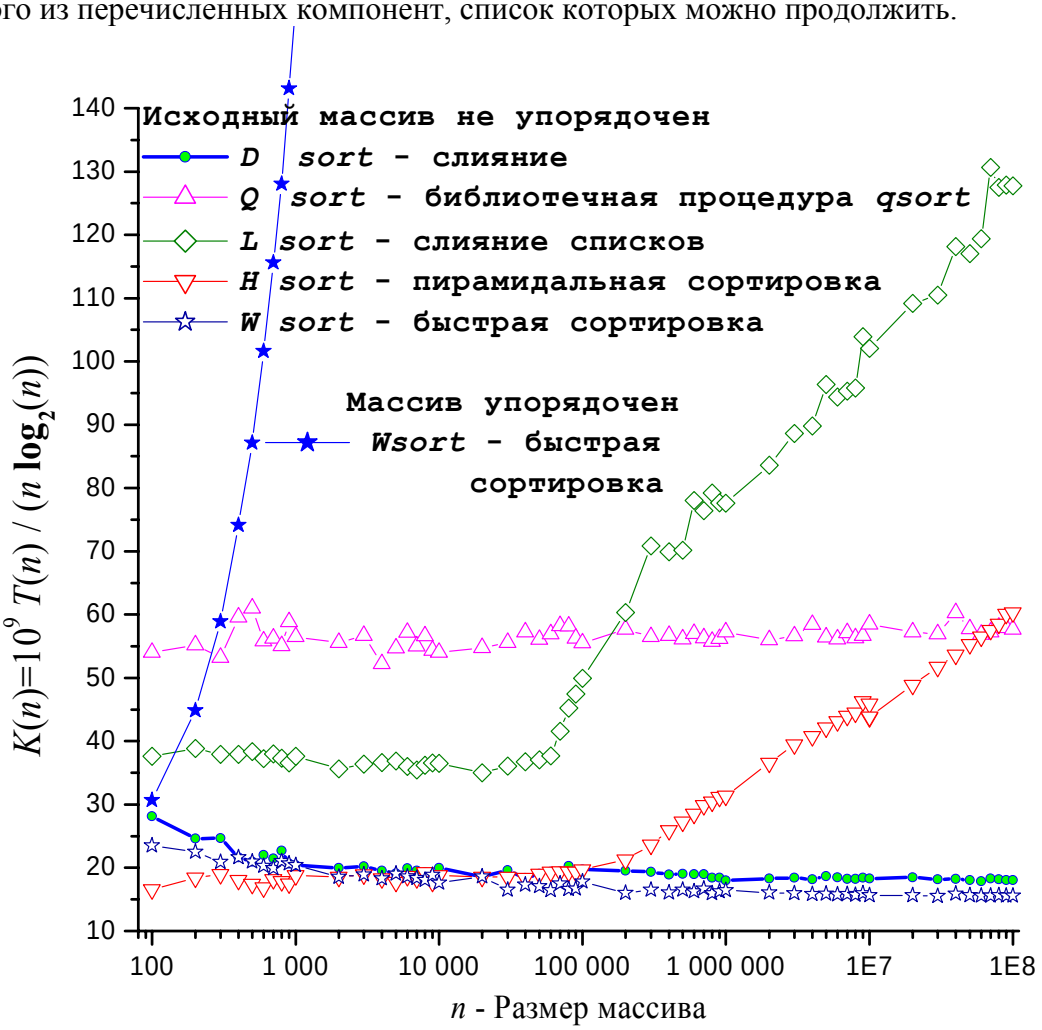

Рис. 3. Сортировки массива на IBM PC, Pentium IV, 2.4 GHz, 2 GByte ОП.

Стандартная процедура «быстрой сортировки» *qsort*, входящая в состав библиотеки времени выполнения

Использование стандартной процедуры «быстрой сортировки» ***qsort***, входящей в состав библиотеки времени выполнения компилятора, нецелесообразно по следующим причинам:

- алгоритм быстрой сортировки, он же алгоритм обменной сортировки с разделением, он же алгоритм сортировки Хоара, в среднем обладает асимптотической оценкой числа операций $O(n\log_2(n))$. Но, в наихудшем случае требуется порядка $O(n^2)$ операций, что не позволяет выполнять сортировку произвольных массивов сколько-нибудь значительного размера за приемлемое время;
- время выполнения сортировки с помощью процедуры ***qsort*** больше, чем время работы других рассматриваемых алгоритмов, даже в среднем случае (Рис. 3). Вероятно, это является платой за универсальность интерфейса (процедура ***qsort*** позволяет обрабатывать данные любого типа) и происходит за счет неэффективного доступа к элементам сортируемого массива.

Процедура «быстрой сортировки» *wsort*

Процедура «быстрой сортировки» ***wsort***, непосредственно реализованная согласно [1] действительно быстрее остальных рассмотренных программ выполняет сортировку неупорядоченных массивов больших размеров, оправдывая тем самым свое название. Однако, при сортировке массивов, элементы которых предварительно упорядочены, время выполнения ***wsort*** растет, как и следовало ожидать, пропорционально n^2 (уходящая вверх помеченная звездочками кривая на Рис. 3), что не позволяет за приемлемое время упорядочивать произвольные массивы с большим числом элементов.

Сортировка слиянием

Следующий, из обсуждаемых алгоритмов, алгоритм ***dsort*** требует выполнения $O(n \log_2 n)$ действий. Он основан на идее рекурсивного слияния упорядоченных фрагментов массива и может быть описан следующим образом:

```
сортировать ( массив mas, число элементов n)
  {
  если (n > 1)
    {
    // сортировка первой половины массива
    сортировать ( mas, n/2);
    // сортировка второй половины массива
    сортировать ( mas+n/2, n-n/2);

    // слияние отсортированных половинок массива
    слияние ( mas, n/2, mas+n/2,n-n/2);
    }
  }
```

Непосредственное использование рекурсии в алгоритме ***сортировать*** требует значительных накладных расходов. Главным образом, они вызваны необходимостью применения дополнительных массивов при слиянии упорядоченных фрагментов.

Само слияние двух упорядоченных массивов с длинами n и m требует $O(n+m)$ действий, при использовании отдельного массива для записи результата слияния. Поскольку в рекурсивном алгоритме результат должен быть размещен в том же массиве, что и исходный массив, необходимо на каждом шаге выполнить копирование результата, что значительно замедляет обработку. В связи с этим предлагается использовать аналогичный алгоритм, не использующий рекурсию.

```
Dsort(intsort *array, int n)
{
 a=array;       // сортируемый массив
 b=array_second;    // вспомогательный массив

 for(i=1;i<n;i=i*2) // размер объединяемых фрагментов
     {
     for(j=0;j<n;j=j+2*i) // начало первого из объединяемых
                          // фрагментов
          {
          r=j+i; // начало второго из объединяемых фрагментов
```

```
            n1=min(i,n-j);
            n2=min(i,n-r);

            if(n1<0)n1=0;
            if(n2<0)n2=0;

            // слияние упорядоченных фрагментов

            for(ia=0,ib=0,k=0;k<n1+n2;k++)
                 {
                 if(ia>=n1) b[j+k]=a[r+ib++];
                 else
                 if(ib>=n2) b[j+k]=a[j+ia++];
                 else
                 if(a[j+ia]<a[r+ib]) b[j+k]=a[j+ia++];
                 else                b[j+k]=a[r+ib++];
                 }
            }

      c=a;a=b;b=c;
      }

 c=a;a=b;b=c;

 // копирование, если результат
 // размещен не в основном, а во вспомогательном массиве
 if(b!=array)
     memcpy(array,b,n*sizeof(intsort));
}
```

Очевидная дальнейшая оптимизация приведенного алгоритма заключается в предварительной сортировке алгоритмом ***hsort*** коротких фрагментов массива, а затем – в применении алгоритма ***dsort*** к уже подготовленным упорядоченным фрагментам. Полученная таким образом процедура ***dhsort*** затрачивает наименьшее, или мало отличающееся от наименьшего, время (Рис. 4) по сравнению с остальными рассмотренными процедурами. В широком диапазоне объемов сортируемых данных при их различной начальной упорядоченности, она может рассматриваться в качестве «наилучшей доступной последовательной процедуры».

В заключение обсуждения последовательных алгоритмов сортировки следует отметить, что результаты, аналогичные приведенным на рис. 2-4, получены при тестировании широкого круга вычислительных систем. Ниже перечислены процессоры и операционные системы некоторых из них:

Pentium IV, Windows	Alpha21264A, Linux	P670, AIX
Pentium III, Linux	Itanium, Linux	Power PC-603, AIX

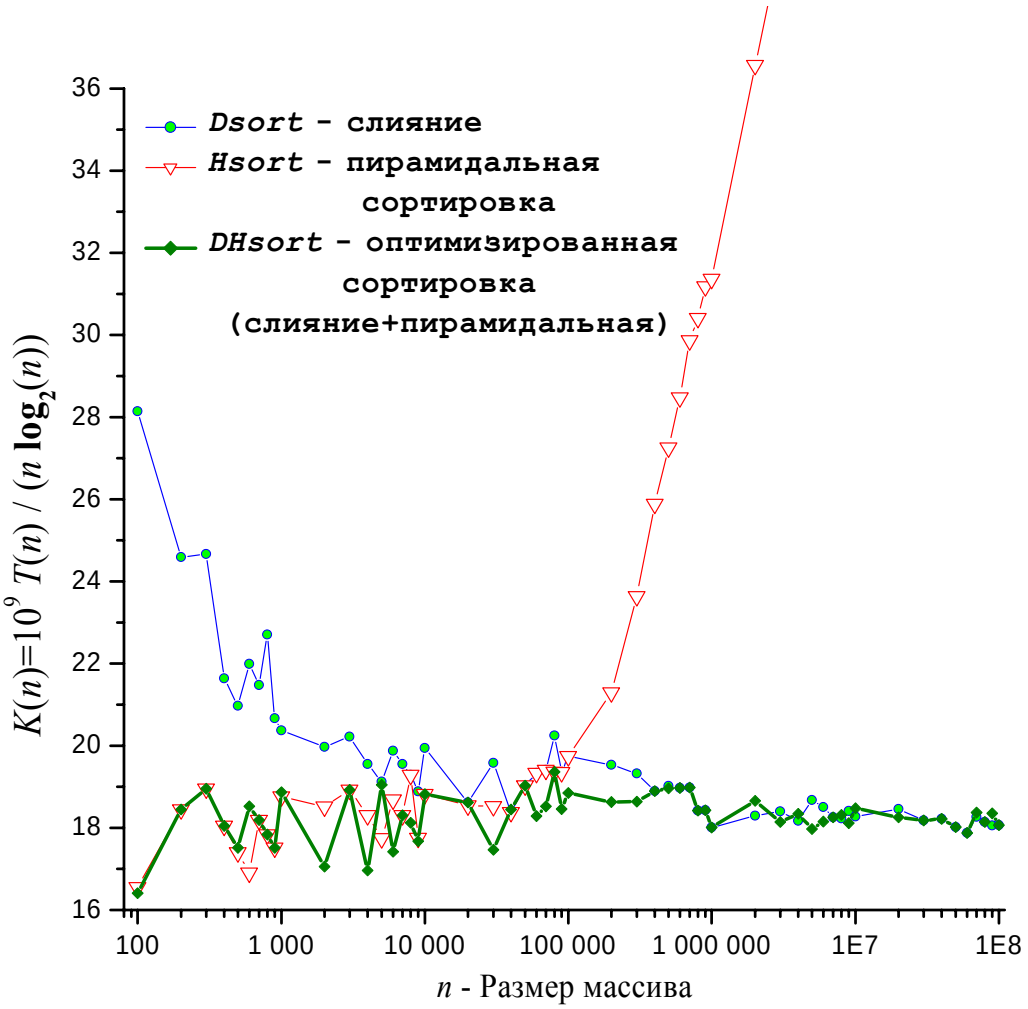

Рис. 4. Сортировки массива на IBM PC, Pentium IV, 2.4 GHz, 2 GByte ОП.

Параллельные алгоритмы сортировки

Начиная обсуждение параллельных алгоритмов сортировки, уточним, что будет пониматься под упорядоченным массивом. Будем предполагать, что p используемых процессоров имеют номера от 0 до p-1. Будем предполагать, что в начале исходный массив распределен по процессорам некоторыми порциями A_i, где i – номер процессора. По окончании сортировки элементы исходного массива распределены по процессорам некоторыми упорядоченными по не убыванию порциями B_i. В общем случае размеры исходных и отсортированных порций могут не совпадать: возможно, что $|A_i| \neq |B_i|$; возможно, что $|A_i| \neq |A_j|$; возможно, что $|B_i| \neq |B_j|$. Под B_i^k будем понимать k-ый элемент фрагмента массива, расположенного на процессоре i. При сделанных соглашениях будем считать распределенный по процессорам массив B отсортированным, если выполняются условия:

$$\begin{cases} B_i^k \le B_i^j, \text{ для любых } i, \text{ при } k<j \\ B_i^k \le B_l^j, \text{ для любых } j,k, \text{ при } i<l \end{cases}$$

Для простоты изложения будем также предполагать, что $p=2^k$, где r – натуральное число.

Рассмотрим два параллельных алгоритма сортировки массивов. Первый разработан на основе метода сдваивания, второй - на основе «обменной сортировки со слиянием» Бэтчера [1].

Метод сдваивания

Параллельный алгоритм сортировки массива на основе метода сдваивания состоит из двух этапов:

1) на каждом из процессоров с помощью алгоритма ***dhsort*** сортируется фрагмент массива A_i длинной $|A_i| = \frac{n}{p}$. При этом, для простоты анализа, будем предполагать, что $n=vp$, где v - целое неотрицательное число.

2) элементы отсортированных на первом этапе фрагментов A_i упорядочиваются с помощью слияния, причем последовательность слияний определяется методом сдваивания (Рис. 5).

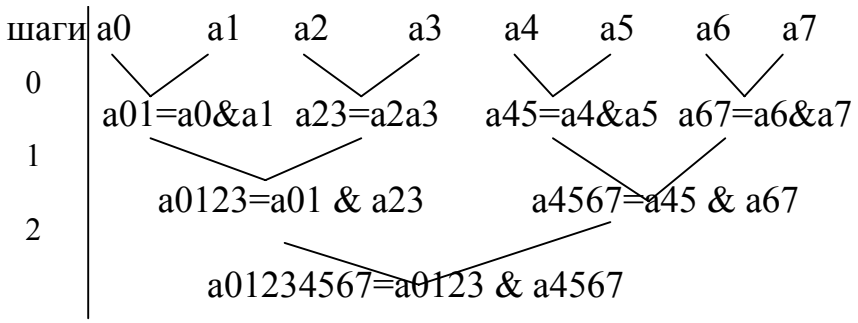

Рис. 5. Порядок выполнения слияния фрагментов массива методом сдваивания на восьми процессорах

На нулевом шаге (j=0) каждый процессор с номером $2i$, (i=0,…,p-1) получает от процессора с номером $2i$+1 отсортированный фрагмент массива A_{2i+1} длиной n/p, что требует $k_2 \frac{n}{p}$ действий, после чего процессоры с номерами $2i$ выполняют слияние пар фрагментов, что требует от каждого процессора выполнения $2k_3 \frac{n}{p}$ действий: $A_{2i} \otimes A_{2i+1} \longrightarrow A'_{2i}$.

На шаге j каждый процессор с номером $2^j \cdot 2i$ получает от процессора с номером $2^j(2i+1)$ отсортированный фрагмент массива длиной $2^j \frac{n}{p}$, что требует от каждого работающего процессора $k_2 \frac{n}{p} \cdot 2^j$ действий. Далее процессор с номером $2^j \cdot 2i$ выполняет слияние двух фрагментов $A_{2^j 2i} \otimes A_{2^j(2i+1)} \longrightarrow A'_{2^j 2i}$, что требует выполнения $k_3 \cdot \frac{n}{p} \cdot 2 \cdot 2^j$ действий на каждом работающем процессоре.

Общее время выполнения сортировки при таком подходе:

$$T(n,p) = \frac{n}{p}\left(k_1 \log_2 \frac{n}{p} + k_2(p-1) + 2k_3(p-1)\right)$$

Отметим, что $k_1 = k_3$, поскольку оба эти коэффициента определяют число действий выполняемых одним и тем же алгоритмом слияния упорядоченных массивов.

Таким образом, ожидаемое ускорение:

$$S(n,p) = \frac{T(n,1)}{T(n,p)} =$$

$$= \frac{k_1 n \log_2 n}{\frac{n}{p}\left[k_1\left(\log_2 \frac{n}{p} + 2_3(p-1)\right) + k_2(p-1)\right]} =$$

$$= \frac{p}{\left(\log_2 n - \log_2 p + 2(p-1) + \frac{k_2}{k_1}(p-1)\right)\log_n 2}$$

$$S(n,p) = \frac{p}{1 + \left(2(p-1) - \log_2 p + \frac{k_2}{k_1}(p-1)\right)\log_n 2}$$

Определим предполагаемое ускорение, достижимое на 4 и на 32 процессорах при $n = 10^9$.

$$S(10^9,4) \approx \frac{4}{1.13 + \frac{1}{30}\frac{k_2}{k_1}} < 3.5$$

$$S(10^9,32) = \frac{32}{1 + \frac{1}{30}\left(56 + 31\frac{k_2}{k_1}\right)} \approx \frac{32}{3 + \frac{k_2}{k_1}} < 10$$

При четырех процессорах, даже имеющих доступ к общей памяти (в этом случае затраты на передачу массивов от процессора к процессору можно считать равными нулю: k_2=0), ускорение не превысит 3.5. При использовании 32 процессоров на общей памяти ускорение не превысит 10. Поскольку на реальных системах с распределенной памятью $k_2 >> k_1$, можно предположить, что ускорение будет незначительным, а значит нецелесообразно использовать системы с распределенной памятью с числом процессоров большим нескольких штук. Кроме того, данный алгоритм не позволяет обрабатывать массивы, объем которых превышает оперативную память одного процессорного узла, поскольку заключительное слияние двух половин всего массива выполняется на одном процессоре.

Таким образом, метод сдваивания прост в реализации, но эффективен только при небольшом числе процессоров, объединенных общей памятью.

«Обменная сортировка со слиянием» Бэтчера

Параллельный алгоритм сортировки массива на основе метода «обменной сортировки со слиянием» Бэтчера состоит из двух этапов, причем первый из них совпадает с первым этапом рассмотренного выше алгоритма. На втором этапе так же выполняется ряд слияний упорядоченных фрагментов, но есть два существенных отличия от алгоритма сдваивания. Во-первых, порядок слияния определяется сетью сортировки. Во-вторых, и это самое важное, при слиянии фрагментов не происходит увеличение размера обрабатываемого на каждом из процессоров фрагмента. В следствии этого не происходит увеличения, к концу процесса сортировки, объема передаваемых от процессора к процессору данных, равно как не происходит и

уменьшения числа выполняющих полезную работу процессоров. В этом заключается существенное отличие от метода сдваивания, на последнем шаге которого работает только один процессор, принимающий половину всего сортируемого массива. Итак:

1) на каждом из процессоров с помощью алгоритма ***dhsort*** сортируется фрагмент массива длинной *n/p*. При этом будем полагать, что *n=rp*, где *r* - целое число.

2) отсортированные фрагменты объединяются с помощью процедуры, выполняемой модулем компаратора слияния [2], причем последовательность слияний определяется алгоритмом «обменной сортировки со слиянием» Бэтчера [1].

Процедура, выполняемая компаратором слияния заключается в преобразовании фрагментов массива, расположенных на процессорах **а** и **б**:

- процессоры **а** и **б** обмениваются хранящимися на них отсортированными фрагментами, после чего на каждом из процессоров **а** и **б** оказываются два предварительно упорядоченных фрагмента.

- процессор **а** выделяет из двух фрагментов, длины *m* каждый, *m* наименьших элементов, формируя новый отсортированный фрагмент длины *m*. Одновременно с этим, процессор **б** выделяет *m* наибольших элементов, формируя новый отсортированный фрагмент длины *m*.

Опишем теперь алгоритм сортировки предварительно упорядоченных фрагментов массива.

Алгоритм рекурсивный и предполагает при сортировке *n*+*m* фрагментов выполнение независимой сортировки *n* первых фрагментов и *m* последних фрагментов, после чего объединение двух сформированных упорядоченных массивов с помощью (*n*,*m*)-сети слияния фрагментов. Предлагаемый алгоритм практически полностью совпадает с подробно изложенным в монографии Кнута [1] методом, с той разницей, что вместо сортировки элементов выполняется сортировка фрагментов массива.

(*n*,*m*)-сеть слияния фрагментов можно описать следующим образом:

- Если *n*=0 или *m*=0, то сеть пуста.
- Если *n*=1 и *m*=1, то сеть состоит из единственного компаратора слияния.
- Если *nm*>1, то, обозначив объединяемые последовательности через $\langle A_1, A_2, \ldots, A_n \rangle$ и $\langle B_1, B_2, \ldots, B_n \rangle$, объединим последовательности фрагментов, имеющих нечетные номера $\langle A_1, A_3, \ldots, A_{2\lceil n/2 \rceil - 1} \rangle \otimes \langle B_1, B_3, \ldots, B_{2\lceil m/2 \rceil - 1} \rangle \to \langle C_1, C_2, \ldots, C_r \rangle$. Аналогично поступим с последовательностями фрагментов, имеющих четные номера $\langle A_2, A_4, \ldots, A_{2\lfloor n/2 \rfloor} \rangle \otimes \langle B_2, B_4, \ldots, B_{2\lfloor n/2 \rfloor} \rangle \to \langle D_1, D_2, \ldots, D_t \rangle$,
  где $r = \lceil n/2 \rceil + \lceil m/2 \rceil$, $t = \lfloor n/2 \rfloor + \lfloor m/2 \rfloor$

  Сформируем окончательный результат $\langle E_1, E_2, \ldots, E_{n+m} \rangle$, выполнив операции компаратора слияния над парами фрагментов:

  $C_1 \to E_1$

  $\langle D_1 \rangle \otimes \langle C_2 \rangle \to \langle E_2, E_3 \rangle$,

  $\langle D_2 \rangle \otimes \langle C_3 \rangle \to \langle E_4, E_5 \rangle$,

  $\vdots$

  если *r*=*t*+1

  $\langle D_t \rangle \otimes \langle C_r \rangle \to \langle E_{n+m-1}, E_{n+m} \rangle$,

  иначе,

$$C_r \to E_{n+m}$$

Рассмотрим в качестве примера этапы сортировки шести фрагментов массива длиной $n/6$ каждый.

1) отсортировать на каждом процессоре массивы длиной $m=n/6$.

2) выполнить слияние отсортированных фрагментов в соответствии со следующей схемой:

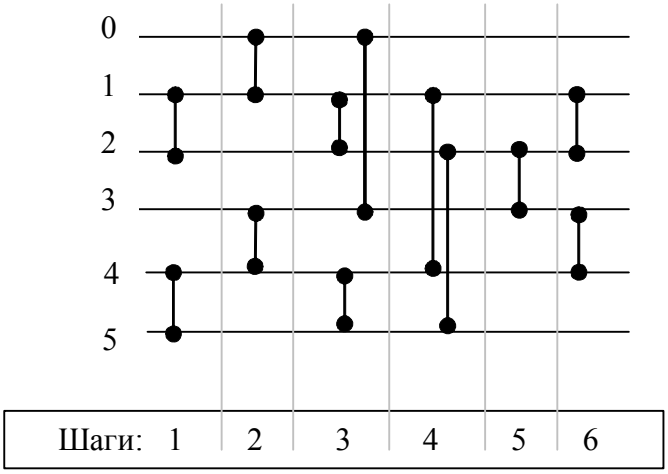

Рис. 6. Порядок выполнения слияния фрагментов массива методом сдваивания на восьми процессорах

Каждая горизонтальная черта (Рис. 6) символизирует процессор, каждая вертикальная – компаратор слияния.

Используя принцип нулей и единиц [1] несложно показать, что описанный алгоритм правильно сортирует произвольные массивы тогда и только тогда, когда длины объединяемых фрагментов *в точности равны между собой*. Таким образом, при сортировке реальных данных следует дополнять массив фиктивными элементами, с тем, что бы суммарная длина массива была кратна числу процессоров.

Для доказательства необходимости равенства размеров фрагментов достаточно убедится, что существует массив, неверно сортируемый сетью, если длины фрагментов не равны. Соответствующий пример приведен на рис. 7, на котором показана сортировка массива содержащего три единицы (черные кружки) и четыре нуля (белые кружки). После сортировки элементы массива, имеющие б**о**льшие значения должны быть размещены на процессорах с б**о**льшими номерами, однако на процессоре 6 после сортировки остался один из нулей, что является ошибкой.

Можно показать, что суммарный объем данных, передаваемых (принимаемых) каждым из процессоров не превышает n, независимо от числа используемых процессоров, что обуславливает хорошую эффективность алгоритма в целом. Кроме того, на каждом процессоре ни в какой момент времени не требуется хранить более чем $3\frac{n}{p}$ элементов сортируемых массивов (два исходных фрагмента и один фрагмент результата). Таким образом, общий объем сортируемых данных ограничен только суммарным объемом оперативной памяти всех используемых процессорных узлов (в предположении, что все процессорные узлы равноценны с точки зрения размера

доступной им оперативной памяти). Общее число сортируемых элементов не ограничивается объемом оперативной памяти каждого из процессорных узлов.

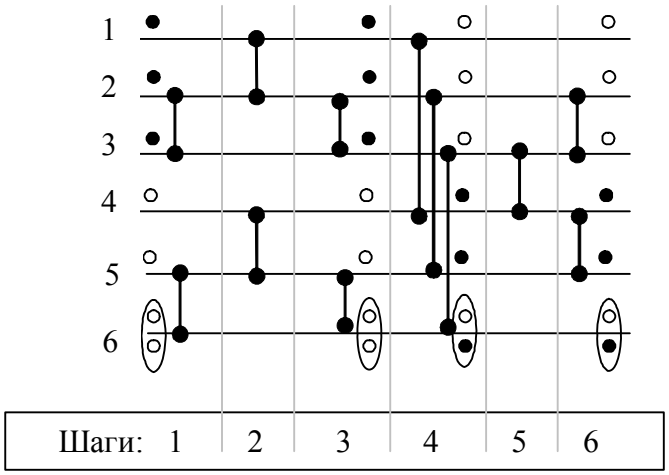

Рис. 7. Пример неправильного упорядочивания фрагментов неравной длины: на процессорах 1 … 5 размещено по одному элементу, на процессоре 6 – два элемента

Из приведенной схемы (рис.6) следует, что слияние фрагментов может быть выполнено за шесть шагов. Например, на четвертом шаге выполняется обмен данными и процедура компаратора слияния одновременно между парами процессоров (0,3), (1,4), (2,5). На этом шаге работой обеспечены все процессоры. Однако на остальных шагах это не так, например, на пятом шаге простаивают все процессоры, кроме пары (2,3). Таким образом, описанный алгоритм обладает ограниченной степенью внутреннего параллелизма, не зависимо от типа используемой вычислительной системы.

Время сортировки массива оценивается следующим выражением:

$$T(n,p) = k\frac{n}{p}\left(\log_2\left(\frac{n}{p}\right) + b\cdot s_p\right), \quad s_p \approx \frac{\lceil \log_2 p \rceil(\lceil \log_2 p \rceil + 1)}{2},$$

где s_p - число шагов слияния (точные значения для некоторых p приведены в таблице 2), $b \geq 1$ - константа, определяющая время слияния двух фрагментов массива, включая время либо на передачу данных между процессорами, либо на синхронизацию (если используется вычислительная система с общей памятью). В случае использования общей памяти при малом числе процессоров $b \sim 1$, соответственно, максимальное значение коэффициента эффективности использования вычислительной мощности дается выражением:

$$E^{\max}(n,p) = \frac{t(n,1)}{pt(n,p)} = \frac{\log_2 n}{\log_2 n + s_p - \log_2 p} \approx \frac{1}{1 + \log_n p(\log_2 p - 1)/2}$$

Таким образом, без учета накладных расходов на обмены, максимально возможная эффективность используемого алгоритма, при отсутствии накладных расходов на обмены заведомо меньше 100%. Для некоторых значений p точные значения величины s_p приведена в таблице 2.

Результаты численных экспериментов

В Таблице 3. приведены результаты, полученные при сортировке 10^8 4х байтовых целых чисел на 768 процессорной системе МВС 1000М (МСЦ РАН, www.jscc.ru), оснащенной процессорами Alpha21264A, с частотой 667 MHz. Сеть Myrinet2000 обеспечивает обмен данными между двумя процессорными узлами МВС 1000М при использовании библиотеки MPI со скоростью 110 - 150 Мбайт/сек.

Таблица 3

Сортировка 10^8 4х байтовых целых чисел на системе МВС 1000М

P	*T,сек*	*E*	*S*	E^{max}	S^{max}	s_p
1	83.51	100.00%	1.00	100%	1.0	0
2	46.40	90.00%	1.80	100%	2.0	1
3	35.93	77.48%	2.32	95%	2.8	3
4	29.68	70.35%	2.81	96%	3.9	3
5	24.45	68.33%	3.42	91%	4.5	5
6	22.16	62.80%	3.77	92%	5.5	5
7	21.82	54.67%	3.83	89%	6.2	6
8	19.95	52.32%	4.19	90%	7.2	6
16	12.36	42.22%	6.75	82%	13.1	10
27	9.32	33.20%	8.97	74%	20.0	14
32	7.85	33.24%	10.64	73%	23.3	15
48	6.45	26.97%	12.95	66%	31.9	19
64	4.92	26.53%	16.98	64%	40.9	21
128	3.19	20.47%	26.20	56%	71.5	28
192	2.52	17.29%	33.19	51%	98.2	33
256	1.99	16.41%	42.02	49%	124.6	36
384	1.63	13.33%	51.20	49%	187.0	41
512	1.29	12.64%	64.74	42%	217.4	45
640	1.21	10.78%	69.02	41%	264.7	47

Здесь ***T*** – время сортировки (сек), ***E***, ***S*** – эффективность и ускорение, полученные при численном расчете, E^{max}, S^{max} – максимально возможные эффективность и ускорение, обусловленные степенью внутреннего параллелизма используемого алгоритма. Таким образом, массив из 100 миллионов четырехбайтовых чисел отсортирован на 640 процессорах за 1.21 сек. Одному процессору этой системы для сортировки того же массива требуется 83.51сек. Максимальное ускорение, которое могло бы быть достигнуто на 640 процессорах в отсутствии потерь на передачу данных составляет 264.7. Достигнутое ускорение составило 69, что является достаточно хорошим результатом, учитывая большое число использованных процессоров. Следует подчеркнуть, что с увеличением числа используемых процессоров время решения задачи устойчиво сокращается, что вполне соответствует основной преследуемой цели - минимизации времени сортировки массива.

Дальнейшее увеличение ускорения может быть получено за счет использования более эффективной битонной сортировки [1]. В настоящее время не известно какого-либо регулярного метода построения «наиболее быстродействующих сетей» сортировки, (за исключением полного перебора), но для некоторых значений *p* такие сети построены. К примеру, для шести процессоров известна сеть сортировки, обеспечивающая обработку данных за 5 шагов (Рис. 7), в отличии от 6 шагов, при сортировке с помощью сети, указанной на Рис. 6.

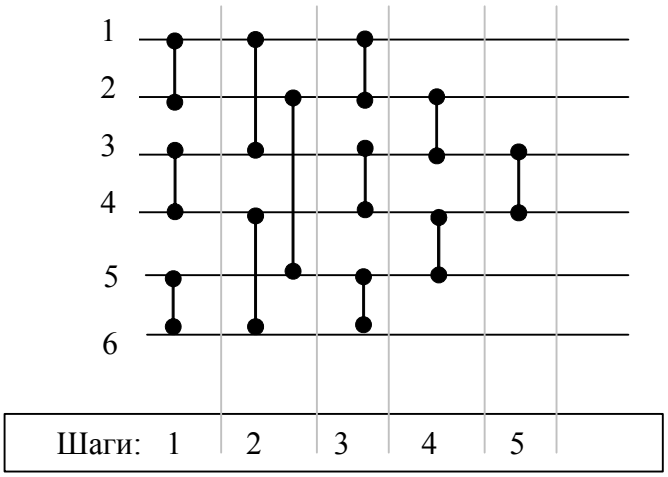

Рис. 7. Наиболее быстродействующая сеть для 6 процессоров [1]

Список литературы

1. Дональд Э.Кнут. *Искусство программирования, т.3. Сортировка и поиск* 2-е изд.: Пер. с английского – М.: Издательский дом «Вильямс», 2001.
2. Седжвик Роберт. *Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск*: Пер. с англ./Роберт Седжвик. - СПб.: ООО "ДиаСофтЮП", 2002.-688с.
3. *Якобовский М.В.* Параллельный алгоритм генерации последовательностей псевдослучайных чисел. // Математическое моделирование, в печати
4. *Якобовский М.В.* Последовательности псевдослучайных чисел для многопроцессорных приложений, 2008. http://www.imamod.ru/projects/FondProgramm/RndLib/lrnd32_v02/