В.И.Шелехов

ТРАНСФОРМАЦИЯ ПРЕДИКАТНОЙ ПРОГРАММЫ СОРТИРОВКИ СЛИЯНИЕМ В ЭФФЕКТИВНУЮ ПАРАЛЛЕЛЬНУЮ ПРОГРАММУ

В данной работе описывается техника предикатного программирования, используемая для разработки параллельного алгоритма сортировки. Применяется нетривиальный способ склеивания массивов, являющихся параметрами предиката. Обычно рекурсивное определение предиката конструируется через рекурсивный вызов на ветви условного оператора или оператора расщепления. Здесь же рекурсивное определение реализуется через вызов, являющийся составной частью параллельного оператора. Новая форма рекурсии определяет также новую форму параллелизма, требующую введения в язык P потоков (**threads**) и боксов (**boxes**), в которых эти потоки локализуются. Читателю предварительно следует ознакомиться с работой [2] настоящего сборника.

Допустим, имеется вещественный массив элементов a[m..n], где $m \le n$. Необходимо реализовать сортировку массива методом слияния [1]. В этом алгоритме массив a делится на две части. В каждой части сортировка проводится независимо. Два отсортированных массива объединяются (сливаются) в один отсортированный массив просмотром по обоим массивам и сравнением текущих элементов. Для каждой из двух частей массива сортировка реализуется по той же схеме: массив делится на два и т.д.

Тип массива a имеет следующее описание:

```
type Ar(int x, y) = array x..y of real;
```

Для глобальных переменных m и n будем использовать описание:

```
int m, n; // m <= n
```

Спецификация программы сортировки слиянием представляется предикатом sort(Ar(n,m) a*:). Предполагается, что результат сортировки должен быть в том же массиве a, поэтому в реализации массивы a и a' должны быть склеены.

Непосредственная реализация сортировки слиянием может быть дана следующим определением:

```
sort(Ar(m,n) a* : ) // m <= n, a' — упорядочена, a' есть перестановка a
{     nat d = (n-m+2)/2;
      if d = 1 then a' = a
      else   sort(a[m..m+d-1] : Ar(m, m+d-1) b);
             sort(a[m+d..n]: Ar(m+d..n) c);
             merge(b, c: a')
      end
}
```

В теле определения предикат merge(b, c: a') реализует отсортированное объединение двух отсортированных массивов b и c.

Определение sort является рекурсивным, причем рекурсия является бинарной. Чтобы преобразовать бинарную рекурсию к хвостовому виду, необходимо сначала преобразовать ее к унарной рекурсии. Обычно применяемый метод — это обобщение исходной задачи для сортировки набора массивов, являющихся подмассивами одного массива. Для этой цели будем рассматривать массив a с разбиением на куски (подмассивы) из d элементов: a[m+d*i..m+d*i+d-1], где i = 0,1,2,...

Новый алгоритм сортировки определяется следующим образом. Сначала мы сортируем массив кусками из двух элементов, затем — из четырех, далее — из восьми. Последовательно удваивая куски, мы в конце концов накроем массив a одним отсортированным куском, и таким образом решим исходную задачу.

Определение исходного предиката sort сводится к более общей задаче sortMulti:

```
sort(Ar(m,n) a* : ) // m <= n, a' — упорядочена, a' есть перестановка a
{     sortMulti(1, a: a')     }
```

Массив a в предикате sortMulti является отсортированным кусками длиной d: каждая из вырезок a[m+d*i.. m+d*(i+1)], (i=0,1,2,...) является упорядоченной. Это условие будем обозначать предикатом sortP(a, d, m, n). Очевидно, что куски длиной d=1 всегда являются отсортированными.

```
sortMulti(nat d, Ar(m,n) a* : )
        // m <= n, sortP(a, d, m, n), a' — упорядочена, a' есть перестановка a
{     if n-m+1 <= d then a
      else sortMulti(d*2, sortDouble(d, m, a))
      end
}
```

Определение sortMulti дано в функциональном стиле. В нем сортировка в целом рекурсивно определяется через сортировку вдвое большими кусками. Предикат sortDouble имеет на входе массив a[p..n], отсортированный кусками длиной d, и сортирует его кусками по 2*d.

```
sortDouble(nat d, int p, Ar(p,n) a* : )
      // sortP(a, d, p, n), sortP(a', 2*d, p, n)
{     if p+d-1 >= n then a
      else int q = p+2*d;
            if q-1 >=n then sortPiece(d, p, n, a)
            else a'[p..q-1] = sortPiece(d, p, q-1, a[p..q-1]) ||
                  a'[q..n] = sortDouble(d, q, a[q..n])
            end
      end
}
```

Предикат sortPiece сортирует один кусок a[p..r] длиной не более 2*d при условии, что две его половины отсортированы. Сначала в первой половине a[p..p+d-1] с помощью предиката findUnsorted находится наименьший индекс t, для которого a[t] > a[p+d]. Если условие ложно на всем куске a[p..p+d-1], то реализуется первая ветвь предиката findUnsorted. Кусок a[p..t-1] является отсортированной частью в массиве a[p..r].

После определения индекса t оставшийся кусок a[t..p+d-1] переписывается в массив b[t..p+d-1]. Далее куски b[t..p+d-1] и a[p+d..r] сливаются в отсортированном виде в массив a[t..r]. Это слияние реализуются предикатом merge.

```
sortPiece(nat d, int p, r, Ar(p,r) a* : )
      // sortP(a, d, p, r), a' — упорядочена, a' есть перестановка a
{     int s = p+d;
      real e = a[s];
      split findUnsorted(p, r, s, a, e: | int t)
      do  a
      do a'[p..t-1] = a[p..t-1] ||
         a'[t..r] = merge(t, t, t, s-1, s+1, r, (t: e), a[t..s-1], a[s+1..r])
      end
}
```

Предикат merge сливает в отсортированном виде массивы b[j..k] и c[l..r] в массив a′[t..r], причем начальная часть a′[t..i]=a[t..i] — есть отсортированная часть, полученная ранее путем слияния.

В реализации императивной программы массив a[t..i] склеивается с начальной частью массива a'[t..r], а массив c[l..r] — с конечной частью a'[t..r]. Массив b не склеивается с куском a[t..s-1], а получается в результате его копирования.

```
merge(int t, i, j, k, l, r, Ar(t,i) a, Ar(j,k) b, Ar(l,r) c : Ar(t,r) a')
      // a, b, c — отсортированы, a′[t..i]=a[t..i], a[i]<=b[i], a[i]<=c[l],
      // a′ - отсортирован  и является перестановкой a+b+c
{   if j > k then a' = a + c
    elsif l > r then a' = a + b
    else
        int i1 = i+1;
        if b[j] < c[l] then
                merge(t, i1, j+1, k, l, r, a + (i1: b[j]), b[j+1..k], c : a')
        else    merge(t, i1, j, k, l+1, r, a + (i1: c[l]), b, c[l+1..r] : a')
        end
    end
}
```

Оператор **int** i1 = i+1 не является необходимым. Однако в случае его отсутствия при замене в определении merge хвостовой рекурсии циклом, на каждой из ветвей условного оператора появится оператор i := i+1, который далее следовало бы выносить из обоих ветвей перед условным оператором. Понятно, что подобное вынесение проще сделать с помощью оператора **int** i1 = i+1 в рамках предикатной программы. Отметим, что в императивной программе переменные i и i1 будут склеены.

Предикат findUnsorted определяет минимальный индекс t, для которого a[t] > e. Если условие a[t] > e ложно на всем куске a[p..r], реализуется первая ветвь предиката findUnsorted

```
findUnsorted(int p, r, s, Ar(p,r) a, real e : | t)
      // t=min{x | a[x]>e}
{     if p > s then #1
      elsif a[p] > e then t = p #2
      else findUnsorted(p+1, r, s, a[p+1..r], e : #1 | t #2)
      end
}
```

Приведенные выше определения предикатов представляют полную предикатную программу. Преобразуем ее в эффективную императивную программу.

На первом этапе реализуется склеивание переменных и замена хвостовой рекурсии циклом в каждом из определений предикатов. Предварительно рассмотрим реализацию параллельного оператора в определении sortDouble:

```
a'[p..q-1] = sortPiece(d, p, q-1, a[p..q-1]) || a'[q..n] = sortDouble(d, q, a[q..n])
```

Вызов предиката sortPiece может быть запущен параллельным процессом (потоком), выполняемым одновременно со вторым оператором. Будем использовать конструкцию **thread** <блок>. При ее исполнении создается и запускается новый параллельный процесс, тело которого есть <блок>. Далее исполнение программы, породившей новый процесс, передается конструкции, следующей за **thread** <блок>. Будем также использовать конструкцию **box** <блок> — ее исполнение завершается при завершении исполнения всех параллельных процессов, запущенных при исполнении <блока>.

Таким образом, приведенный параллельный оператор реализуется в императивной программе следующим образом:

```
box { thread { sortPiece(d, p, q-1, a[p..q-1]: a'[p..q-1]) };
      sortDouble(d, q, a[q..n]: a'[q..n])
    }
```

Рекурсия в преобразованном определении предиката sortDouble не является хвостовой. Для приведения рекурсии к хвостовому виду вынесем конструкцию **box** из определения и применим ее для вызова sortDouble в определении sortMulti. Далее проведем склеивание переменных и замену хвостовой рекурсии циклов во всех определениях. Во всех определениях переменная a' склеивается с a. При раскрытии возникающих групповых операторов иногда требуется поменять порядок присваивания переменных. Предварительно функциональный стиль записи заменяется операторным.

```
sort(Ar(m,n) a* : )
{     sortMulti(1, a: a)
}
sortMulti(nat d, Ar(m,n) a* : )
{     loop
            if n-m+1 <= d then exit
            else    box { sortDouble(d, m, a: a) };
                    d := d*2
            end
      end
}
```

```
sortDouble(nat d, int p, Ar(p,n) a* : )
{   loop
        if p+d-1 >= n then exit
        else   int q = p+2*d;
               if q-1 >=n then sortPiece(d, p, n, a: a); exit
               else    thread {sortPiece(d, p, q-1, a[p..q-1]: a[p..q-1]) };
                       p:=q
               end
        end
   end
}
sortPiece(nat d, int p, r, Ar(p,r) a* : )
{       int s = p+d;
        real e = a[s];
        findUnsorted(p, r, s, a, e: #1 | int t);
        merge(t, t, t, s-1, s+1, r, (t: e), a[t..s-1], a[s+1..r]: a[t..r])
}
findUnsorted(int p, r, s, Ar(p,r) a, real e : | t)
{       loop   if p > s then #1
               elsif a[p] > e then t = p #2
               else p:=p+1
               end
        end
}
```

В определении merge начальный кусок a'[t..i] склеивается с a[t..i], а переменная — c[l..r] склеивается с конечным куском a'[l..r]. Массив b не склеивается со средним куском массива a, а копируется из него. Переменная i1склеивается с i.

```
merge(int t, i, j, k, l, r, Ar(t,i) a, Ar(j,k) b, Ar(l,r) a : Ar(t,r) a)
{     loop
              if j > k then a[t..r] = a[t..i] + a[l..r]; exit
              elsif l > r then a[t..r] = a[t..i] + b; exit
              else
                       i := i+1;
                       if b[j] < c[l] then     a[i] := b[j]; j:=j+1
                       else                    a[i] := c[l]; l:=l+1
                       end
              end
      end
}
```

В операторе a[t..r] = a[t..i] + a[l..r] выполняется равенство l = i+1, поскольку j = k+1 и l-i = k-j+2. Данный оператор является тождественной пересылкой и поэтому удаляется из программы.

Подставим определения findUnsorted и merge на место вызовов в определение sortPiece. Подставим определение sortDouble в sortMulti, а затем sortMulti в sorti. Приведем текст итоговой императивной программы.

```
type Ar(int x, y) = array x..y of real;
int m, n; // m <= n
sort(Ar(m,n) a* : )
{     int d:=1;
      loop
            if n-m+1 <= d then exit end;
            box {     int p:=m;
                      loop
                            if p+d-1 >= n then exit end;
                            int q = p+2*d;
                            if q-1 >=n then sortPiece(d, p, n, a: a); exit end;
                            thread {sortPiece(d, p, q-1, a[p..q-1]: a[p..q-1])};
                            p:=q
                      end
                };
            d := d*2
      end
}
sortPiece(nat d, int p, r, Ar(p,r) a* : )
{     int s = p+d; real e = a[s];
      loop if p > s then return
            elsif a[p] > e then t = p; exit
            else p:=p+1
            end
      end;
      int i:=t; int j:=t; int k:=s-1; int l:=s+1;
      Ar(j,k) b := a[t..s-1]; a[t]:=e;
      loop
            if j > k then exit
            elsif l > r then a[i+1..r] = b[j..k]; exit
            else i := i+1;
                  if b[j] < c[l] then     a[i] := b[j]; j:=j+1
                  else                    a[i] := c[l]; l:=l+1
                  end
            end
      end
}
```

Очевидно, что массив b[m..n] можно сделать глобальным по отношению к определению sortPiece, используя в теле sortPiece его единственную копию. Кроме того, следует удалить переменные d и a из списка параметров определения sortPiece, трактуя их в контексте определения sort.

СПИСОК ЛИТЕРАТУРЫ

1. **Кнут Д.** Искусство программирования для ЭВМ. Т.3. Сортировка и поиск. Пер. с англ. — М.: Мир, 1978. — 843с.
2. **Шелехов В.И.** Предикатное программирование: основы, язык, технология // Наст. сб. — С.7–15.