УДК 519.163
MSC 05A18

Вестник СПбГУ. Прикладная математика. Информатика... 2020. Т. 16. Вып. 1

Разделяемый алгоритм перебора разбиений конечного множества на подмножества заданной мощности

А. М. Ковшов

Санкт-Петербургский государственный университет, Российская Федерация,
199034, Санкт-Петербург, Университетская наб., 7–9

Для цитирования: *Ковшов А. М.* Разделяемый алгоритм перебора разбиений конечного множества на подмножества заданной мощности // Вестник Санкт-Петербургского университета. Прикладная математика. Информатика. Процессы управления. 2020. Т. 16. Вып. 1. С. 50–61. https://doi.org/10.21638/11702/spbu10.2020.105

Представлен итерационный алгоритм перебора разбиений конечного множества, состоящего из различимых элементов, на подмножества заданной мощности. Мощности некоторых подмножеств могут совпадать. Весь алгоритм состоит из двух независимых алгоритмов. Первый алгоритм определяет, в подмножество какой мощности при разбиении попадет каждый элемент исходного множества. Для этого подмножества одинаковой мощности объединяются в составные подмножества. Производится распределение элементов исходного множества по составным подмножествам. Разбиения описываются индексным массивом, указывающим, в какое составное подмножество распределяется каждый элемент исходного множества. Перебор разбиений исходного множества по составным подмножествам сводится к перебору всех перестановок индексов в массиве. Второй алгоритм распределяет элементы внутри каждого составного подмножества по равномощным подмножествам. При этом для каждого составного подмножества создается свой индексный массив, описывающий, в какое подмножество попадает каждый элемент составного подмножества. Перебор всех разбиений составного подмножества по равномощным подмножествам сводится к частичному перебору перестановок индексов. Перестановки всех индексных массивов перебираются в лексикографическом порядке. Такое построение алгоритма позволяет разбить весь перебор на независимые части и использовать параллельные вычисления. Рассмотрен пример, показывающий состоятельность алгоритма и ускорение получения результата при применении параллельных вычислений.

Ключевые слова: параллельные вычисления, алгоритмы перебора, разделяемые алгоритмы.

1. Введение. Когда решение какой-либо задачи требует полного перебора элементов некоторого множества, встает вопрос о выборе алгоритма, который можно воплотить в компьютерную программу, способную справиться с поставленной задачей за приемлемое время. Рекурсивные алгоритмы просты в программировании, но требуют значительной компьютерной памяти, к тому же их довольно сложно приспособить для параллельных вычислений. Итерационные алгоритмы гораздо меньше расходуют память, поэтому их использование во многих случаях оказывается предпочтительнее [1]. Если к тому же удается построить алгоритм таким образом, что вычисления можно производить частями параллельно, то можно существенно ускорить получение результата. Очевидно, для того, чтобы можно было определять элементы целевого множества хотя бы в двух параллельных процессах, нужно в одном процессе запустить перебор от начала до середины, а на другом — от середины до

конца. В этом случае при независимости вычислительных процессов можно ожидать двукратной экономии времени.

Для осуществления такого подхода необходимо ввести отношение порядка на целевом множестве, чтобы для любых двух разных элементов множества можно было достаточно быстро определить, какой из них является «меньшим», а какой — «бóльшим». Слова *достаточно быстро* здесь имеют решающее значение, поскольку любой алгоритм, осуществляющий перебор, задает отношение порядка на целевом множестве естественным образом, где самым маленьким будет начальный элемент перебора, а самым большим — конечный. Однако сравнить два произвольных элемента целевого множества можно, только запустив сам алгоритм и дождавшись выбора одного из двух элементов, который и будет меньшим. Понятно, что столь продолжительное ожидание итогов сравнения никак не позволит ускорить перебор. Другое дело, если каждому элементу целевого множества сопоставить число, строку или вектор. В данном случае сравнить любые два элемента не представляет труда. Остается найти такой алгоритм, который перебирает элементы целевого множества строго в порядке возрастания, и при этом ему не требуется никакой дополнительной информации. Другими словами, такой алгоритм должен для любого произвольного элемента целевого множества находить ближайший следующий за ним элемент. Тогда, зная наименьший и наибольший элементы целевого множества, а также, умея находить средний элемент, можно запустить алгоритм параллельно на двух процессорах, на одном из которых перебор будет идти от наименьшего элемента до среднего, а на другом — от среднего до наибольшего. Как правило, определить наименьший и наибольший элементы можно достаточно легко, чего нельзя сказать о нахождении точной середины. Но даже если середина устанавливается с большой погрешностью, этого достаточно для распараллеливания вычислений. И если на одном из вычислительных устройств процесс уже завершится, а на другом до конца еще далеко, можно остановить работающий алгоритм и, разделив остаток целевого множества между двумя устройствами, снова продолжить вычисления. Понятно, что алгоритм, обладающий такими свойствами, годится для распараллеливания вычислений между любым числом устройств. К данным алгоритмам, например, относится алгоритм перебора всех подмножеств исходного множества. Поскольку каждый элемент исходного множества либо входит в какое-либо подмножество, либо не входит, то каждому подмножеству может быть сопоставлено n-значное двоичное число, где n — число элементов исходного множества. Такими же свойствами обладают алгоритм разбиения целого положительного числа на целые положительные слагаемые [1], алгоритм перебора путей на графе [2], алгоритм перебора всех разбиений множества на подмножества [3, 4], алгоритмы, перебирающие перестановки и сочетания [5, 6]. В последнее время интерес к комбинаторным параллельным алгоритмам связан с исследованиями в области теории графов [7, 8], генетики [9–14], биохимии [15, 16].

В настоящей работе предлагается подобный алгоритм для разбиения множества на подмножества заданных мощностей, т. е. разбиение на заданное число подмножеств с постоянным числом элементов в каждом. Поскольку этот алгоритм позволяет разделить вычисления на независимые части, то задача обмена данными и синхронизация параллельных вычислительных процессов [17, 18] заключается лишь в объединении результатов, полученных каждой частью отдельно.

2. Постановка задачи. Пусть есть исходное конечное множество S, состоящее из n элементов s_i, $i = 0, \ldots, n-1$. Будем считать, что все элементы s_i разные, т. е. их можно различить. Множества с различимыми элементами рассмотрены, например,

Стенли [19]. Не умаляя общности, можно считать, что множество S состоит из натуральных чисел от 0 до $n-1$. Нужно найти все разбиения S на k подмножеств, из которых k_0 подмножеств содержат ровно m_0 элементов, k_1 подмножеств — ровно m_1 элементов и т. д. При этом

$$\sum_{i=0}^{r-1} k_i = k, \quad \sum_{i=0}^{r-1} m_i k_i = n.$$

Итого в разбиении присутствуют подмножества r разных мощностей $m_0, \ldots, m_{r-1}$ при числе подмножеств каждой мощности $k_0, \ldots, k_{r-1}$ соответственно. Обозначим подмножества разбиения B_i^j, где j — номер мощности m_j этого множества, а i — порядковый номер этого подмножества среди подмножеств одинаковой мощности. Пусть объединение всех подмножеств одной и той же мощности m_j будет обозначаться A_j: $A_j = \bigcup\limits_{i=0}^{k_j-1} B_i^j$, тогда $\bigcup\limits_{j=0}^{r-1} A_j = S$ и $A_\alpha \cap A_\beta = \emptyset$, $B_\gamma^j \cap B_\delta^j = \emptyset$, где $\alpha \neq \beta$, $\gamma \neq \delta$.

Будем считать два разбиения разными, если найдутся два таких элемента s_p, s_q исходного множества S, которые при одном из двух разбиений оказались в разных подмножествах, а при другом разбиении — в одном.

В каждом подмножестве элементы упорядочиваются в порядке возрастания. Пусть при некотором разбиении подмножество B_i^j состоит из элементов s_α исходного множества S, где $\alpha = 0, \ldots, m_j - 1$. Обозначим их $b_0^{i,j}, b_1^{i,j}, \ldots, b_{m_j-1}^{i,j}$, при этом $b_\alpha^{i,j} < b_\beta^{i,j}$, если $\alpha < \beta$. Равномощные подмножества упорядочиваются в порядке возрастания их наименьшего элемента. Считается, что $B_\alpha^j < B_\beta^j$, если $b_0^{\alpha,j} < b_0^{\beta,j}$.

Например, одно из разбиений множества $S = \{0,1,2,3,4,5,6,7,8,9\}$ из десяти элементов на четыре подмножества, два из которых имеют мощность, равную двум, а два других — трем, запишется так: $\{5,9\}$, $\{7,8\}$, $\{0,1,4\}$, $\{2,3,6\}$. Здесь $m_0 = 2$, $m_1 = 3$, $k_0 = 2$, $k_1 = 2$, $B_0^0 = \{5,9\}$, $B_1^0 = \{7,8\}$, $B_0^1 = \{0,1,4\}$, $B_1^1 = \{2,3,6\}$, $A_0 = B_0^0 \cup B_1^0$, $A_1 = B_0^1 \cup B_1^1$.

Алгоритм состоит из двух переборов: внешнего и внутреннего. Внешний заключается в перечислении всех возможных распределений элементов исходного множества между подмножествами A_j, $j = 0, \ldots, r-1$. Внутренний перебирает распределения элементов каждого подмножества A_j между подмножествами B_i^j, $i = 1, \ldots, k_j - 1$.

3. Внешний алгоритм. Каждое подмножество A_j, $j = 0, \ldots, r-1$, имеет мощность n_j, равную $k_j m_j$. При этом $\sum_{j=0}^{r-1} n_j = n$. Распределение элементов исходного множества S по подмножествам A_j можно описать целочисленным одномерным массивом $\mathbf{x}$ длины n, совпадающей с мощностью исходного множества, $\mathbf{x} = (x_0, \ldots, x_{n-1})$. Каждому элементу s_α исходного множества S соответствует элемент x_α массива $\mathbf{x}$, который определяет, к какому подмножеству A_j принадлежит в данном распределении s_α. Значения x_α равны индексу j подмножества A_j, который может быть от 0 до $r-1$. Таким образом, в массиве $\mathbf{x}$ всегда будет n_α элементов со значением α, где $\alpha = 0, \ldots, r-1$. Так, для вышеприведенного примера $\mathbf{x} = (1,1,1,1,1,0,1,0,0,0)$, поскольку элементы исходного множества $0,1,2,3,4,6$ оказались в A_1, а $5,7,8,9$ — в A_0, т. е. $A_0 = \{5,7,8,9\}$, $A_1 = \{0,1,2,3,4,6\}$.

Таким образом, перебор всех распределений элементов исходного множества S по подмножествам A_j, $j = 0, \ldots, r-1$, сводится к перебору всех перестановок внутри массива $\mathbf{x}$. Например, для массива из двух нулей и двух единиц все перестановки будут такими: $(0,0,1,1)$; $(0,1,0,1)$; $(0,1,1,0)$; $(1,0,0,1)$; $(1,0,1,0)$; $(1,1,0,0)$. Начальным значением массива $\mathbf{x}$ этого итерационного алгоритма является лексикографически

наименьшее значение, при котором $x_\alpha \leqslant x_\beta$, если $\alpha < \beta$. Каждая следующая итерация массива $\mathbf{x}$ вычисляется таким образом. Ищется наибольшее значение индекса α, для которого $x_\alpha < x_{\alpha+1}$. Если оказалось, что таких α не существует, то это значит, что все перестановки перечислены. Если же такое α нашлось, то из всех x_β, для которых $\beta > \alpha$ и $x_\beta > x_\alpha$, выбирается наименьшее значение. Оно будет перемещено на место x_α, а все остальные элементы x_β, для которых $\beta > \alpha$, вместе с прежним x_α будут расставлены на места от $\alpha + 1$ до $n - 1$ в порядке возрастания: $x_\beta \leqslant x_{\beta+1}$ при $\beta > \alpha$. Так, если $\mathbf{x} = (2, 1, \mathbf{1}, 3, 3, 3, 3, 2, 2, 0)$, то следующей итерацией будет $(2, 1, \mathbf{2}, 0, 1, 2, 3, 3, 3, 3)$, а если $\mathbf{x} = (3, 1, 2, 1, 2, \mathbf{0}, 3, 3, 3, 2)$, — то $(3, 1, 2, 1, 2, \mathbf{2}, 0, 3, 3, 3)$. Здесь полужирным шрифтом выделены элементы x_α, для которых выполнено условие $\max(\alpha)$: $x_\alpha < x_{\alpha+1}$. На место данных элементов при следующей итерации помещаются наименьшие из стоящих справа элементов, которые больше этих x_α. Оставшиеся справа элементы вместе с замещаемым элементом переставляются справа в порядке возрастания.

Представим исходный текст на языке Java этого алгоритма:

```
/**
  * Вычисление следующей перестановки в массиве
  * с повторяющимися элементами.
  * @param r     Число разных элементов в перестановках.
  * @param x     Сама расстановка переставляемых элементов.
  * @return      true, если еще не все расстановки перечислены;
  *              else --- в противном случае.
  */
 public static boolean nextX(int r, int[] x) {
     int[] right = new int[r];
     right[x[x.length - 1]] = 1;
     for (int i = x.length - 2; i >= 0; i--) {
         right[x[i]]++;
         if (x[i] < x[i + 1]) {
             do {
                 x[i]++;
             } while (right[x[i]] < 1);
             right[x[i]]--;
             int l = i + 1;
             for (int j = 0; j < right.length; j++) {
                 while (right[j] > 0) {
                     x[l++] = j;
                     right[j]--;
                 }
             }
             return true;
         }
     }
     return false;
 }
```

4. Внутренний алгоритм. Каждое подмножество A_j, $j = 0, \ldots, r - 1$, содержит n_j элементов исходного множества S и $n_j = k_j m_j$. Пусть a_α^j — элемен-

ты, входящие в подмножество A_j, где $\alpha = 0, \ldots, n_j - 1$. При этом элементы упорядочены в порядке возрастания: $a^j_\alpha < a^j_\beta$, если $\alpha < \beta$. Распределение данных элементов по k_j равномощным мощности m_j подмножествам B^j_i, $i = 0, \ldots, k_j - 1$, можно описать целочисленным одномерным массивом $\mathbf{y_j}$ длины n_j. Значения каждого элемента y^j_α массива $\mathbf{y_j}$ указывают номер i подмножества B^j_i, которому принадлежит при данном распределении соответствующий элемент a^j_α. Таким образом, в массиве $\mathbf{y_j}$ присутствуют элементы, имеющие значения от 0 до $k_j - 1$, и элементов с каждым значением ровно m_j. Подмножества B^j_i упорядочены в порядке возрастания наименьших элементов, поэтому элемент a^j_0 всегда принадлежит множеству B^j_0. Для $\alpha > 0$, если $a^j_\alpha \in B^j_i$, то $\alpha \geqslant i$. Чтобы перебрать все разбиения подмножества A_j на k_j равномощных подмножеств, достаточно перебрать все перестановки элементов массива $\mathbf{y_j} = (y^j_0, \ldots, y^j_{n_j-1})$, для которых $y^j_\alpha \leqslant \alpha$. Например, если $A_j = \{0, 1, 2, 3\}$, $k_j = 2$, $m_j = 2$, то всего различных перестановок элементов массива $\mathbf{y_j}$, удовлетворяющих заданным условиям, будет три: $(0, 0, 1, 1)$, $(0, 1, 0, 1)$, $(0, 1, 1, 0)$, которым будут соответствовать следующие разбиения на два подмножества по два элемента в каждом: $\{0, 1\}$,$\{2, 3\}$; $\{0, 2\}$,$\{1, 3\}$; $\{0, 3\}$,$\{1, 2\}$.

Алгоритм нахождения всех перестановок, удовлетворяюших условиям порядка, отличается от алгоритма нахождения всех перестановок без ограничения только дополнительным условием на завершение перебора. Так, если начальной перестановкой является $(0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3)$, то при переборе всех возможных перестановок без дополнительных ограничений конечная перестановка будет $(3, 3, 3, 2, 2, 2, 1, 1, 1, 0, 0, 0)$, а при ограничении на положение элемента в расстановке перебор окончится на перестановке $(0, 1, 2, 3, 3, 3, 2, 2, 1, 1, 0, 0)$.

Исходный текст на языке Java этого алгоритма запишется так:

```
/**
  * Вычисление следующей перестановки в множестве с повторяющимися
  * элементами, при условии того, что номер места каждого элемента
  * в расстановке не может превышать его значения.
  * @param m Число совпадающих элементов каждого значения в перестановке.
  * @param k Число разных значений элементов в перестановке.
  * @param y Текущая расстановка переставляемых элементов.
  * @return  true, если еще не все расстановки перечислены;
  *          else --- в противном случае.
  */
 public static boolean nextY(int m, int k, int[] y) {
     int[] right = new int[k];
     right[y[y.length - 1]] = 1;
     for (int i = y.length - 2; i >= 0; i--) {
         right[y[i]]++;
         if (y[i] < y[i + 1] && right[y[i]] < m) {
             do {
                 y[i]++;
             } while (right[y[i]] < 1);
             right[y[i]]--;
             int l = i + 1;
             for (int j = 0; j < right.length; j++) {
                 while (right[j] > 0) {
```

```
                    y[l++] = j;
                    right[j]--;
                }
            }
            return true;
        }
    }
    return false;
}
```

Ограничивающее условие `right[y[i]] < m` можно записать проще: `y[i] < i`. В этом случае можно было бы убрать первый параметр из заголовка метода, поскольку, кроме данного условия, параметр `m` больше нигде не используется. Но была оставлена более сложная запись для того, чтобы показать, что в таком случае метод `nextY` может заменить вышеописанный метод `nextX`, если вместо числа совпадающих элементов через параметр `m` передавать в метод длину массива `y`. Тогда дополнительное условие будет выполняться в любом случае, и алгоритм будет перебирать все перестановки без ограничений.

5. Полный алгоритм. Каждое разбиение исходного множества S на подмножества заданной мощности описывается массивами $\mathbf{x}, \mathbf{y_0}, \ldots, \mathbf{y_{r-1}}$. Полный перебор всех разбиений представляет собой $r+1$ вложенных переборов. Сначала перебираются все перестановки массива $\mathbf{y_{r-1}}$, затем вычисляется следующая итерация для массива $\mathbf{y_{r-2}}$ и снова перебираются все перестановки массива $\mathbf{y_{r-1}}$. Когда будут пройдены все итерации для массива $\mathbf{y_{r-2}}$, вычислится следующая итерация для массива $\mathbf{y_{r-3}}$, а массивы $\mathbf{y_{r-2}}$ и $\mathbf{y_{r-1}}$ будут возвращены в начальное положение. И так далее, пока не будут перечислены все перестановки массива $\mathbf{x}$. На вход метода `nextZ`, вычисляющего следующую перестановку, передаются такие параметры: массив $\mathbf{m} = (m_0, \ldots, m_{r-1})$ различных мощностей; массив $\mathbf{k} = (k_0, \ldots, k_{r-1})$, описывающий, сколько подмножеств каждой мощности присутствует в разбиении; массив $\mathbf{n} = (n_0, \ldots, n_{r-1})$, показывающий, сколько элементов исходного множества распределено в подмножества каждой мощности; массив $\mathbf{x} = (x_0, \ldots, x_{n-1})$, определяющий, в подмножество какой мощности попадает каждый элемент исходного множества S; массив $\mathbf{Y} = (\mathbf{y_0}, \ldots, \mathbf{y_{r-1}})$ векторов, каждый из которых $\mathbf{y}_\alpha = (y_0^\alpha, \ldots, y_{n_\alpha-1}^\alpha)$ определяет распределение элементов исходного множества S между равномощными подмножествами мощности m_α. Метод вычисляет новые значения массивов $\mathbf{x}$ и $\mathbf{Y}$, определяющих следующее разбиение.

Представим на языке Java этот метод и метод `initialPartition`, устанавливающий начальное распределение элементов исходного множества между равномощными подмножествами:

```
/**
 * Вычисление следующей итерации разбиения множества различимых элементов
 * на заданное число подмножеств заданной мощности.
 *
 * @param m Перечень мощностей подмножеств разбиения.
 * @param k Сколько множеств каждой мощности в разбиении.
 * @param n Перечень мощностей объединений равномощных подмножеств.
 * @param x Распределение элементов по мощностям подмножеств.
 * @param y Распределения элементов по подмножествам.
 * @return  false --- если найдено последнее разбиение;
```

```
     *          true --- в противном случае.
     */
    public static boolean nextZ(int[] m, int[] k, int[] n, int[] x, int[][] y)
    {
        for (int i = y.length - 1; i >= 0; i--) {
            if (nextY(m[i], k[i], y[i])) {
                for (int j = i + 1; j < y.length; j++) {
                    initialPartition(m[j], y[j]);
                }
                return true;
            }
        }
        if (nextX(n.length, x)) {
            for (int j = 0; j < y.length; j++) {
                initialPartition(m[j], y[j]);
            }
            return true;
        }
        return false;
    }

    /**
     * Назначение начальной перестановки. Массив переставляемых
     * элементов заполняется в порядке возрастания повторяющимися
     * числами от 0 до наибольшего значения, равного частному от деления
     * длины массива переставляемых элементов на число повторений
     * каждого значения.
     * @param m Число совпадений каждого значения переставляемых элементов.
     * @param y Массив, заполняемый переставляемыми элементами.
     */
    public static void initialPartition(int m, int[] y) {
        int l = 0;
        for (int i = 0; i < y.length; l++) {
            for (int j = 0; j < m; j++) {
                y[i++] = l;
            }
        }
    }
```

Разбиения перебираются в лексикографическом порядке возрастания индексных массивов $\mathbf{x}, \mathbf{y_0}, \ldots, \mathbf{y_{r-1}}$. В рассмотренном выше примере разбиения исходного множества $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ на четыре подмножества, два из которых содержат два элемента, а два других — три, распечатка нескольких разбиений из середины перечня выглядит следующим образом:

(1,1,1,1,0,1,1,0,0,0);((0,1,1,0),(0,1,1,0,0,1));(2,3,3,2,0,2,3,1,1,0);{4,9},{7,8},{0,3,5},{1,2,6};
(1,1,1,1,0,1,1,0,0,0);((0,1,1,0),(0,1,1,0,1,0));(2,3,3,2,0,3,2,1,1,0);{4,9},{7,8},{0,3,6},{1,2,5};
(1,1,1,1,0,1,1,0,0,0);((0,1,1,0),(0,1,1,1,0,0));(2,3,3,3,0,2,2,1,1,0);{4,9},{7,8},{0,5,6},{1,2,3};
(1,1,1,1,1,0,0,0,0,1);((0,0,1,1),(0,0,0,1,1,1));(2,2,2,3,3,0,0,1,1,3);{5,6},{7,8},{0,1,2},{3,4,9};

(1,1,1,1,1,0,0,0,0,1);((0,0,1,1),(0,0,1,0,1,1));(2,2,3,2,3,0,0,1,1,3);{5,6},{7,8},{0,1,3},{2,4,9};
(1,1,1,1,1,0,0,0,0,1);((0,0,1,1),(0,0,1,1,0,1));(2,2,3,3,2,0,0,1,1,3);{5,6},{7,8},{0,1,4},{2,3,9}.

Слева напечатаны массивы $\mathbf{x}$ и $(\mathbf{y_0}, \mathbf{y_1})$; правее по этим трем массивам вычислен массив, ставящий в соответствие каждому элементу исходного множества S номер его подмножества разбиения B_i^j, которые нумеруются в порядке $B_0^0, B_1^0, B_0^1, B_1^1$ номерами $0, 1, 2, 3$ соответственно. В таком же порядке эти подмножества отображены справа в фигурных скобках. Данный вид отображения результата со сквозной нумерацией подмножеств разбиения выбран для того, чтобы сделать его более удобным для восприятия, как это сделано в работе [3].

6. Использование параллельных вычислений. Преимуществом описанного алгоритма является то, что элементы целевого множества, в нашем случае — разбиения, упорядочены при помощи определяющего массива, здесь — массива, составленного из массивов $\mathbf{x}, \mathbf{y_0}, \ldots, \mathbf{y_{r-1}}$. Поскольку все элементы целевого множества упорядочены, в рассматриваемом случае — в лексикографическом порядке, то начать перебор можно с любого элемента, задав начало перебора соответствующим значением определяющего массива. Так, если задать начальный массив $\mathbf{x} = (1, 0, 0, 0, 0, 1, 1, 1, 1, 1)$, а массивам $\mathbf{y_0}$ и $\mathbf{y_1}$ назначить лексикографически минимальные значения $(0, 0, 1, 1)$ и $(0, 0, 0, 1, 1, 1)$ соответственно, то перебор разбиений исходного множества из десяти элементов на подмножества, имеющих мощности $2, 2, 3, 3$, начнем примерно с середины. Вообще, для того, чтобы точно определить середину или любую другую заданную часть перебора, требуются дополнительные вычисления, чем на практике можно пренебречь, задав разделение алгоритма приблизительно. Для приведенного примера, если мы хотим разделить вычисления на четыре части, это можно сделать приблизительно, задав первые две цифры в стартовых значениях массива $\mathbf{x}$ в следующих сочетаниях: $(0, 0, \ldots)$; $(0, 1, \ldots)$; $(1, 0, \ldots)$; $(1, 1, \ldots)$. Остальные цифры расставляются в лексикографически наименьшем сочетании. Такое приблизительное, но не требующее дополнительных расчетов разделение вычислений на четыре части, потребует в первой части перечислить 840 разбиений, во второй — 1680, в третьей — 1680, в четвертой — 2100. От общего числа разбиений, равного 6300, первая часть составляет немногим более 13 %, вторая и третья — чуть меньше 27 %, а четвертая — ровно одну треть. Такая неравномерность обусловлена тем, что присутствующие в массиве $\mathbf{x}$ две цифры повторяются по-разному, одна из них — четыре раза, а другая — шесть. Тем не менее даже самая длинная часть в три раза короче целого процесса, что должно ускорить получение конечного результата.

Поскольку при таком грубом делении алгоритма в данном примере первая часть короче последней более чем в 2 раза, было бы естественно обсудить задачу более рационального распределения вычислений. Например, при завершении вычислений первым устройством можно запустить алгоритм, который мог бы перераспределить вычисления, загрузив высвободившиеся ресурсы, однако решение проблемы эффективного использования вычислительных мощностей не входит в цели данной работы.

7. Применение алгоритма на примере решения отдельной задачи. Для оценки возможности алгоритма ускорить получение конечного результата была рассмотрена следующая задача. Дано конечное множество S натуральных чисел от 1 до 20. Их нужно распределить по подмножествам $C_1, C_2, C_3, C_4, C_5, C_6$, имеющим мощности $8, 3, 3, 2, 2, 2$. Применительно к введенным ранее обозначениям $m_0 = 8$, $m_1 = 3$, $m_2 = 2$; $k_0 = 1$, $k_1 = 2$, $k_2 = 3$; $A_0 = B_0^0$, $A_1 = B_0^1 \cup B_1^1$, $A_2 = B_0^2 \cup B_1^2 \cup B_2^2$; $C_1 = B_0^0$, $C_2 = B_0^1$, $C_3 = B_1^1$, $C_4 = B_0^2$, $C_5 = B_1^2$, $C_6 = B_2^2$.

В каждом подмножестве берется сумма всех входящих в него чисел $\sigma_i = \Sigma_{\alpha=1}^{p_i} c_\alpha^i$, где $c_\alpha^i \in C_i$, а p_i — число элементов в подмножестве C_i, $i = 1, 2, 3, 4, 5, 6$. Для каждой полученной суммы σ_i вычисляются простые множители $\sigma_i = \prod_\beta^{g_i} \mu_\beta^i$, здесь μ_β^i — суть простые числа. Из всех разбиений требуется выбрать такие, для которых достигается наибольшее значение суммы всех простых множителей

$$u = \max_{\mathbf{x},\mathbf{Y}} \sum_{i=1}^{6} \sum_{\beta=1}^{g_i} \mu_\beta^i.$$

Из данных разбиений выбираются такие, у которых наименьший разброс значений σ_i (дисперсия):

$$v(u) = \min_{\mathbf{x},\mathbf{Y}} \sum_{i=1}^{6} \left(\sigma_i - \bar{\sigma}\right)^2,$$

где $\bar{\sigma} = \frac{1}{6} \sum_{i=1}^{6} \sigma_i$.

Из отобранных таким образом разбиений ищутся разбиения с минимальной суммой дисперсий внутри каждого подмножества:

$$w(u, v) = \min_{\mathbf{x},\mathbf{Y}} \sum_{i=1}^{6} \left(\frac{1}{p_i} \sum_{\alpha=1}^{p_i} \left(c_\alpha^i - \overline{c^i}\right)^2 \right),$$

где $\overline{c^i} = \frac{1}{p_i} \sum_{\alpha=1}^{p_i} c_\alpha^i$.

Эта задача была решена предложенным алгоритмом. Поскольку в описании алгоритма элементы исходного множества начинались с нуля, при решении данной задачи нулевому элементу было присвоено число 20. Для сравнения вычисления были проведены как целиком, так и с разделением на три независимые части. Разделение было сделано по векторам $\mathbf{x_I}$ и $\mathbf{x_{II}}$. Таким образом, в первой части вычисления проводились от начала до $\mathbf{x_I}$, во второй — от $\mathbf{x_I}$ до $\mathbf{x_{II}}$, в третьей — от $\mathbf{x_{II}}$ до конца. При этом

$$\begin{aligned}
\mathbf{x_0} &= (0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2,2),\\
\mathbf{x_I} &= (1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,2,2,2,2,2,2),\\
\mathbf{x_{II}} &= (2,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2,2).
\end{aligned}$$

Всего для решения данной задачи нужно перебрать примерно 17.5 млрд разбиений, а именно — 17 459 442 000. Из них на первую часть пришлось 6 983 776 800 разбиений, на вторую и третью — по 5 237 832 600.

В итоге были получены следующие разбиения:
из первой части — $\{20, 1, 2, 3, 4, 5, 6, 12\}, \{7, 9, 13\}, \{8, 10, 11\}, \{14, 17\}, \{15, 16\}, \{18, 19\}$;
из второй части — $\{1, 2, 3, 4, 5, 6, 7, 9\}, \{20, 11, 12\}, \{8, 10, 13\}, \{14, 17\}, \{15, 16\}, \{18, 19\}$;
из третьей части — $\{1, 2, 3, 4, 5, 6, 7, 9\}, \{8, 10, 13\}, \{11, 12, 14\}, \{20, 17\}, \{15, 16\}, \{18, 19\}$;
со следующими числами:

$$u_1 = u_2 = u_3 = 210;\ \ v_1 = 72, w_1 = 46.0;\ \ v_2 = 20, w_2 = 29.4;\ \ v_3 = 8, w_3 = 14.7.$$

Сравнив три результата, видно, что наилучшим из них является третий. Такой же результат был получен без разделения вычислений.

В сравнении с последовательным вычислением для данной задачи вычисления по частям заняли соответственно 40, 30 и 30 %. Накладные расходы, связанные с вводом

данных для запуска вычислений по частям, составили около 5 %. Таким образом, при вычислении по частям время получения конечного результата составило 45 % от времени вычислений без разделения, что более чем в два раза быстрее.

8. Заключение. Была сделана попытка показать алгоритм перебора разбиений конечного множества различимых элементов на заданное число подмножеств определенной мощности. Особенностью этого алгорима является возможность разделить вычисления на части, что, как показано на примере решения частной задачи, дает существенный выигрыш во времени, требуемом для получения конечного результата.

Литература

1. *Липский В.* Комбинаторика для программистов / пер. с польск. В. А. Евстигнеева, О. А. Логиновой; под ред. А. П. Ершова. М.: Мир, 1988. 213 с.

2. *Kovshov A. M.* The creating of shared search algorithms on the example of graph path searching // Modern science. 2018. N 12-1. P. 48–51.

3. *Djocić B., Miyakawa M., Sekiguchi S., Semba I., Stojmenović I.* A fast iterative algorithm for generating set partitions // The Computer Journal. 1989. Vol. 32. N 3. P. 281–282.

4. *Djokić B., Miyakawa M., Sekiguchi S., Semba I., Stojmenović I.* Parallel algorithms for generating subsets and set partitions // International Symposium on Algorithms. SIGAL. 1990. P. 76–85.

5. *Chen G. H., Chern M. S.* Parallel generation of permutations and combinations // BIT Numerical Mathematics. 1986. Vol. 26. P. 277–283.

6. *Kokosiński Z.* On generation of permutations through decomposition of symmetric groups into cosets // BIT Numerical Mathematics. 1990. Vol. 30. P. 583–591.

7. *Uçar B.* Partitioning, matching, and ordering: Combinatorial scientific computing with matrices and tensors. Computer Science. Lyon: ENS de Lyon, 2019. 191 p.

8. *Deveci M., Kaya K., Uçar B., Çatalyürek Ü. V.* Hypergraph partitioning for multiple communication cost metrics: Model and methods // Journal of Parallel and Distributed Computing. 2015. Vol. 77. N 3. P. 69–83.

9. *Jiang H., Feng H., Zhu D.* An 5/4-approximation algorithm for sorting permutations by short block moves // Proceedings of the 25th International Symposium on Algorithms and Computation. 2014. P. 491–503.

10. *Alexandrino A., Miranda G., Lintzmayer C., Dias Z.* Approximation algorithms for sorting permutations by length-weighted short rearrangements // Electronic Notes in Theoretical Computer Science. 2019. Vol. 346. P. 29–40.

11. *Galvão G. R., Lee O., Dias Z.* Sorting signed permutations by short operations // Algorithms for Molecular Biology. 2015. N 10. P. 1–17.

12. *Crespelle C., Perez A., Todinca I.* An $O(n^2)$ time algorithm for the minimal permutation completion problem // Discrete Applied Mathematics. 2019. Vol. 254. P. 80–95.

13. *Albert M. H., Lackner M.-L., Lackner M., Vatter V.* The complexity of pattern matching for 321-avoiding and skew-merged permutations // Discrete Mathematics & Theoretical Computer Science. 2016. Vol. 18. N 2. P. 1–17.

14. *Lackner M.-L., Lackner M.* A fast algorithm for permutation pattern matching based on alternating runs // Algorithmica. 2016. Vol. 75. N 1. P. 84–117.

15. *Estaña A., Molloy K., Vaisset M., Sibille N., Siméon T., Bernadó P., Juan Cortés J.* Hybrid parallelization of a multi-tree path search algorithm: Application to highly-flexible biomolecules // Parallel Computing. 2018. Vol. 77. P. 84–100.

16. *Oliveira A., Fertin G., Dias U., Dias Z.* Sorting signed circular permutations by super short operations // Algorithms for Molecular Biology. 2018. Vol. 13. N 12. P. 13–18.

17. *Гергель В. П., Стронгин Р. Г.* Основы параллельных вычислений для многопроцессорных вычислительных систем. Н. Новгород: Изд-во ННГУ им. Н. И. Лобачевского, 2003. 184 с.

18. *Воеводин В. В., Воеводин Вл. В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.

19. *Стенли Р.* Перечислительная комбинаторика / пер. с англ. А. И. Барвинка, А. А. Лодкина; под ред. А. М. Вершика. М.: Мир, 1990. 440 с.

Статья поступила в редакцию 2 февраля 2020 г.

Статья принята к печати 13 февраля 2020 г.

Контактная информация:

Ковшов Александр Михайлович — канд. физ.-мат. наук, доц.; a.kovshov@spbu.ru

A parallel algorithm for iterating partitions of a finite set into subsets of a given cardinality

A. M. Kovshov

St. Petersburg State University, 7–9, Universitetskaya nab., St. Petersburg, 199034, Russian Federation

For citation: Kovshov A. M. A parallel algorithm for iterating partitions of a finite set into subsets of a given cardinality. *Vestnik of Saint Petersburg University. Applied Mathematics. Computer Science. Control Processes*, 2020, vol. 16, iss. 1, pp. 50–61.
https://doi.org/10.21638/11702/spbu10.2020.105 (In Russian)

The article describes an iterative algorithm for searching partitions of a finite set consisting of distinguishable elements into subsets of a given cardinality. The cardinalities of some subsets may be the same. The entire algorithm consists of two independent algorithms. The first algorithm for each element of the original set determines the cardinality of the subset that it will fall into when partitioning. To do this, subsets of the same cardinality are united into composite subsets. The elements of the original set are distributed among composite subsets. An index array is created to describe partitions. This array of indexes indicates which composite subset each element falls into. The length of the index array is equal to the cardinality of the original set. Each composite subset has its own index in the index array. Iterating over partitions of the original set into composite subsets is reduced to iterating over all index permutations in the index array. The second algorithm distributes elements within each composite subset over subsets of equal cardinalities. For each composite subset, an index array is created that describes which subset each element of the composite subset falls into. Iterating over all partitions of a composite subset over equally powerful subsets is reduced to iterating over-index permutations. Permutations must meet the following condition: the index value must not exceed the ordinal number of its place in the permutation. This avoids generating the same permutations. Permutations of all index arrays are iterated in lexicographic order. This construction of the algorithm allows to split the entire search into independent parts and use parallel calculations. An example is considered that shows the consistency of the algorithm and the acceleration of obtaining the result when using parallel calculations.

Keywords: exhaustive search algorithms, parallel computing.

References

1. Lipsky V. *Kombinatorica dlya programmistov* [*Combinatorics for programmers*]. Moscow, Mir Publ., 1988, 213 p. (In Russian)

2. Kovshov A. M. The creating of shared search algorithms on the example of graph path searching. *Modern science*, 2018, no. 12-1, pp. 48–51.

3. Djocić B., Miyakawa M., Sekiguchi S., Semba I., Stojmenović I. A fast iterative algorithm for generating set partitions. *The Computer Journal*, 1989, vol. 32, no. 3, pp. 281–282.

4. Djokić B., Miyakawa M., Sekiguchi S., Semba I., Stojmenović I. Parallel algorithms for generating subsets and set partitions. *International Symposium on Algorithms*. SIGAL, 1990, pp. 76–85.

5. Chen G. H., Chern M. S. Parallel generation of permutations and combinations. *BIT Numerical Mathematics*, 1986, vol. 26, pp. 277–283.

6. Kokosiński Z. On generation of permutations through decomposition of symmetric groups into cosets. *BIT Numerical Mathematics*, 1990, vol. 30, pp. 583–591.

7. Uçar B. *Partitioning, matching, and ordering: Combinatorial scientific computing with matrices and tensors. Computer Science*. Lyon, ENS de Lyon Press, 2019, 191 p.

8. Deveci M., Kaya K., Uçar B., Çatalyürek Ü. V. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 2015, vol. 77, no. 3, pp. 69–83.

9. Jiang H., Feng H., Zhu D. An 5/4-approximation algorithm for sorting permutations by short block moves. *Proceedings of the 25th International Symposium on Algorithms and Computation*, 2014, pp. 491–503.

10. Alexandrino A., Miranda G., Lintzmayer C., Dias Z. Approximation algorithms for sorting permutations by length-weighted short rearrangements. *Electronic Notes in Theoretical Computer Science*, 2019, vol. 346, pp. 29–40.

11. Galvão G. R., Lee O., Dias Z. Sorting signed permutations by short operations. *Algorithms for Molecular Biology*, 2015, no. 10, pp. 1–17.

12. Crespelle C., Perez A., Todinca I. An $O(n^2)$ time algorithm for the minimal permutation completion problem. *Discrete Applied Mathematics*, 2019, vol. 254, pp. 80–95.

13. Albert M. H., Lackner M.-L., Lackner M., Vatter V. The complexity of pattern matching for 321-avoiding and skew-merged permutations. *Discrete Mathematics & Theoretical Computer Science*, 2016, vol. 18, no. 2, pp. 1–17.

14. Lackner M.-L., Lackner M. A fast algorithm for permutation pattern matching based on alternating runs. *Algorithmica*, 2016, vol. 75, no. 1, pp. 84–117.

15. Estaña A., Molloy K., Vaisset M., Sibille N., Siméon T., Bernadó P, Juan Cortés J. Hybrid parallelization of a multi-tree path search algorithm: Application to highly-flexible biomolecules. *Parallel Computing*, 2018, vol. 77, pp. 84–100.

16. Oliveira A., Fertin G., Dias, U., Dias Z. Sorting signed circular permutations by super short operations. *Algorithms for Molecular Biology*, 2018, vol. 13, no. 12, pp. 13–18.

17. Gergel V. P., Strongin R. G. *Osnovy parallelnykh vychisleniy dlya mnogoprocessornykh vychislitelnykh sistem* [*Fundamentals of parallel computing for multiprocessor computing systems*]. Nizhniy Novgorod, Publ. of Lobachevskiy NNGU, 2003, 184 p. (In Russian)

18. Voevodin V. V., Voevodin Vl. V. *Parallelnye vychisleniya* [*Parallel computing*]. Saint Petersburg, BHV-Petersburg Publ., 2002, 608 p. (In Russian)

19. Stanley R. *Perechislitelnaya kombinatorica* [*Combinatorics for programmers*]. Moscow, Mir Publ., 1990, 440 p. (In Russian)

Received: February 02, 2020.
Accepted: February 13, 2020.

Authors' information:

Alexander M. Kovshov — PhD in Physics and Mathematics, Associate Professor; a.kovshov@spbu.ru