ISSN:2231-0711
www.ijcset.net

From Permutations to Iterative Permutations

Hoang Chi Thanh [#1], Nguyen Thi Thuy Loan [*2], Nguyen Duy Ham [^3]

[#] *Department of Computer Science, VNU University of Science, Hanoi, Vietnam*
[1] thanhhc@vnu.vn
[*] *Department of Computer Science, College of Broadcasting II, HoChiMinh City, Vietnam*
[2] nguyenthithuyloan@vov.org.vn
[^] *Department of Computer Science, University of People's Security, HoChiMinh City, Vietnam*
[3] duyhaman@yahoo.com

***Abstract*-- In this paper we construct a new efficient simple algorithm to generate all permutations of a finite set. And then we extend the algorithm for generating all iterative permutations of a multi-set. Applying the parallelizing method based on output decomposition we parallelize this algorithm. Further, we use the parallel algorithm to solve an optimal problem of task arrangement.**

***Keywords*-- complexity, multi-set, optimal arrangement, parallel algorithm, permutation**

I. INTRODUCTION

Permutations of a set are frequently used in many areas of computer science such as scheduling problems, system controls and data mining… There are some good algorithms for generating permutations of a set such as the (adjacent) transposition algorithm [4], the algorithm determining a permutation from its set of reductions [1], the algorithm generating permutations by factorial digits [3], the algorithm reconstructing permutations from cycle minors [5], the algorithm reconstructing permutations from ascending and descending blocks [7], the algorithm generating permutations by inversion vectors [9]… Construction of a fast, simple algorithm for generating permutations of a set and its applications attract many researchers.

The notion of set was extended to multi-set. It is a set to which elements may belong more than once. This extension sets up many new researches. The notion of multi-set becomes a good manner for proving program correctness. It is often used in information processing [6] and many other problems. Significant notions of combinatorics such as permutations, combinations and partitions of a (standard) set are being transformed for multi-sets.

In this paper we first construct a new algorithm for generating all permutations of a standard set. Then we extend the notion of permutation into iterative permutation, investigate its properties and construct an efficient algorithm to generate all iterative permutations of a multi-set. Applying the parallelizing technique based on the output decomposition presented in [8,9] we parallelize this algorithm. To do that, the sequence of desirable iterative permutations of a multi-set is divided into subsequences with 'nearly' equal lengths by appropriate choice of pivots. Using a common algorithm (program) with corresponding input and termination condition, processors will execute in parallel to generate iterative permutations of these subsequences. Furthermore, we apply the algorithm above presented to finding an optimal solution of a task arrangement problem.

The rest of this paper is organized as follows. In part 2 we propose a new algorithm based on the lexicographical order for generating all permutations of a standard set. Part 3 presents notions of multi-set and iterative permutation and a new algorithm for generating all iterative permutations of a multi-set by lexicographical order. Part 4 is devoted to the parallelism of this algorithm. Part 5 presents its application in solving an optimal problem of task arrangement. Some developing directions are proposed in conclusion.

II. SET PERMUTATIONS

Let X be an n-element set. Each permutation of the set X is a checklist of X. It is indeed a bijection from X to itself.

Identify the set $X \equiv \{1, 2, ..., n\}$. Thus, a permutation of X is an integer sequence of the length n, consisting of all integers in X. Each integer sequence may be considered as a word on the alphabet X. Thus, we sort the words increasingly by the lexicographical order.

- The first word (the least) is: 1 2 ... n-1 n. It is an increasing sequence.
- The last word (the most) is: n n-1 ... 2 1. It is an decreasing sequence and is the reverse of the first one.

Starting with the first word, our algorithm repeats a loop to find remaining words. To do so, we use the inheritance principle: the next word is inherited a left part as long as possible of the preceding one.

Assume that $t = a_1a_2 \ldots a_{n-1}a_n$ is a just found word. We have to find a word $t' = a'_1a'_2 \ldots a'_{n-1}a'_n$ next to t in the sorted sequence.

By the lexicographical order, the changing position p is the maximal index i, where $a_i < a_{i+1}$. Thus:

$$p = \max \{ i \mid 1 \le i \le n\text{-}1 \wedge a_i < a_{i+1} \}$$

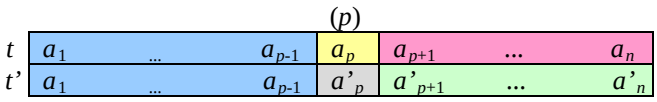

Fig. 2.1. Inheritance of permutations

The desirable word t' inherits a left part of the word t from a_1 to a_{p-1}. It is easy to see that the subsequence $< a_{p+1}, a_{p+2}, ..., a_n >$ is a decreasing sequence.

The changing part in the word t' from the position p to the last is determined as follows:

1) a'_p is the least among all elements in the subsequence $< a_{p+1}, a_{p+2}, \dots , a_n >$ but greater than a_p. We swap a_p for this element:

$$a_p \leftrightarrow \min\{ a_i \mid p+1 \le i \le n \ \wedge \ a_i > a_p \}$$

It ensures that after swapping the subsequence $< a_{p+1}, a_{p+2}, \dots , a_n >$ still is a decreasing sequence.

2) Reverse the subsequence $< a_{p+1}, a_{p+2}, \dots , a_n >$. The result is an increasing sequence and is the least among all permutations of the set $\{a_{p+1}, a_{p+2}, \dots , a_n\}$. It is indeed the subsequence $< a'_{p+1}, a'_{p+2}, \dots , a'_n >$ in the word t'.

Our algorithm terminates when the last permutation has been generated. At that time we have:

$$p = 0$$

This is a termination condition for our algorithm. As the above analysis we have the following simple algorithm to generate all permutations of a set.

Algorithm 2.1

Input: An integer n

Output: A sequence of all permutations of the set {1, 2, ..., n}, sorted increasingly by the lexicographical order.

```
1  Begin
2    input an integer n ;
3    for i ← 1 to n do A[i] ← i ;
4    repeat
5        print the array A[1..n] ;
6        p ← n - 1 ;
7          while A[p] > A[p+1] do p ← p - 1;
8        if p > 0 then
9                { i ← p +1 ;
10                 while A[i] > A[p] do i ← i +1 ;
11                 swap A[p] for A[i-1] ;
12                 reverse the subarray A[p+1..n] ; }
13   until p = 0 ;
14 End .
```

Complexity of the algorithm:

As above presented, generating and printing a permutation with the linear complexity O(n). So the total complexity of Algorithm 2.1 is O($n!.n$). The complexity is least. Hence, this algorithm becomes the best amongst algorithms for generating permutations of a set.

III. Multi-set and Iterative Permutation

3.1. Multi-set: Multi-set is an extended notion of set [4,6] and defined as follows.

Definition 3.1: *A multi-set* is an unordered collection of elements in which elements are allowed to repeat.

For example, a set of values of all variables in a program, a marking of a net system… are typical illustrations of multi-set.

A multi-set is written as follows:

$X = (k_1 * x_1, k_2 * x_2, ..., k_n * x_n)$, where $k_i \ge 1$ with i = 1,2, ..., n.

It means, there are k_1 elements x_1, k_2 elements x_2, … and k_n elements x_n in the multi-set X.

Elements $x_1, x_2, \dots x_n$ are called *basic elements* of the multi-set X, and $k_1, k_2, ..., k_n$ as the *multiplicity* of the corresponding element.

The cardinality of a multi-set is the sum of of its all basic element's multiplicity,

$$|X| = \sum_{i=1}^{n} k_i$$

3.2. Iterative permutation

We extend the notion of permutation of a multi-set as follows.

Definition 3.2: An iterative permutation of a multi-set is a checklist of its all elements.

Example 3.3: Given a multi-set X = (1**a* , 2**b*, 1**c*).

Iterative permutations of the multi-set X are sorted by lexicographical order in the following table.

No	Repesentative integer sequences	Iterative permutations
1	1 2 2 3	*a b b c*
2	1 2 3 2	*a b c b*
3	1 3 2 2	*a c b b*
4	2 1 2 3	*b a b c*
5	2 1 3 2	*b a c b*
6	2 2 1 3	*b b a c*
7	2 2 3 1	*b b c a*
8	2 3 1 2	*b c a b*
9	2 3 2 1	*b c b a*
10	3 1 2 2	*c a b b*
11	3 2 1 2	*c b a b*
12	3 2 2 1	*c b b a*

It is easy to show that the number of all iterative permutations of a multi-set $X = (k_1 * x_1, k_2 * x_2, ..., k_n * x_n)$ is $c_m = \frac{m!}{k_1!k_2!...k_n!}$, where m is the cardinality of the multi-set X.

3.3. Iterative permutation generation algorithm

Given a multi-set $X = (k_1 * x_1, k_2 * x_2, ..., k_n * x_n)$.
Problem: Find all iterative permutations of X.

In other words, we have to construct an efficient algorithm to generate all iterative permutations of this multi-set.

It is easy to see that bijection is not suitable to represent iterative permutation. So we have to find another representation.

Identify the element $x_1 \equiv 1$, element $x_2 \equiv 2$, ..., element $x_n \equiv n$. Each iterative permutation of the multi-set X is represented by an integer sequence of the length m, where there are k_1 integers 1, k_2 integers 2, ... and k_n integers n. Now we find all the integer sequences.

Each integer sequence may be considered as a word on the alphabet {1, 2, ... , n}. So we sort these words increasingly by the lexicographical order.

- The first word (the least) is $\underbrace{1\,1...1}_{k_1}\ \underbrace{2\,2...2}_{k_2}\ ...\ \underbrace{n\,n...n}_{k_n}$. It is a nondecreasing sequence.

- The last word (the most) is $\underbrace{n\,n...n}_{k_n}\ \underbrace{n-1\,n-1...n-1}_{k_{n-1}}\ ...\ \underbrace{1\,1...1}_{k_1}$. It is a nonincreasing sequence and is indeed the reverse of the first one.

Starting with the first iterative permutation, our algorithm repeats a loop to find remaining iterative permutations.

As in Part 2, we assume that $t = a_1a_2 ... a_{m-1}a_m$ is a just found iterative permutation. We have to find an iterative permutation $t' = a'_1a'_2 ... a'_{m-1}a'_m$ next to t in the sorted sequence. By the lexicographical order, the changing position p is the maximal index i, where $a_i < a_{i+1}$. Thus:

$$p = \max \{ i \mid 1 \le i \le m\text{-}1 \ \wedge \ a_i < a_{i+1} \}.$$

The desirable iterative permutation t' inherits a left part of the iterative permutation t from a_1 to a_{p-1}. It is easy to see that the subsequence $< a_{p+1}, a_{p+2}, ... , a_m >$ is a nonincreasing sequence. The changing part in the word t' from the position p to the last is determined as follows:

1) a'_p is the least among all elements in the subsequence $< a_{p+1}, a_{p+2}, ... , a_m >$ but greater than a_p. We swap a_p for this element:

$$a_p \leftrightarrow \min\{ a_i \mid p+1 \le i \le m \ \wedge \ a_i > a_p \}$$

Note that, if there are several least elements we choose the element with the greatest index. It ensures that after swapping the subsequence $< a_{p+1}, a_{p+2}, ... , a_m >$ still is a nonincreasing sequence.

2) Reverse the subsequence $< a_{p+1}, a_{p+2}, ... , a_m >$. The result is a nondecreasing sequence. It is indeed the subsequence $< a'_{p+1}, a'_{p+2}, ... , a'_m >$ in the iterative permutation t'.

Our algorithm terminates when the last iterative permutation was generated. At that time the changing position $p = 0$.

We have the following detail algorithm.

Algorithm 3.1
Input: An integer n and multiplicities $k_1, k_2, ..., k_n$
Output: A sorted sequence of all iterative permutations of the multi-set $(k_1 * x_1, k_2 * x_2, ... , k_n * x_n)$

```
1  Begin
2    input the number of basic elements n ;
3    input multiplicities k_i , i = 1, 2, ..., n ;
4    m ← Σ_{i=1}^{n} k_i ;
5    A[1..m] ← 1 1...1 (k_1) 2 2...2 (k_2) ... n n...n (k_n) ;
6    repeat
7        print the array A[1..m] ;
8        p ← m - 1 ;
9        while A[p] ≥ A[p+1] do p ← p - 1;
10       if p > 0 then
11            { i ← p + 1 ;
12              while A[i] > A[p] do i ← i + 1 ;
13              swap A[p] for A[i-1] ;
14              reverse the subsequence A[p+1..m] ; }
15   until p = 0 ;
16 End .
```

Complexity of the algorithm:

- Instructions 2-3 input data with the complexity O(n).
- Instructions 4-5 calculate the number of elements and assign the first iterative permutation with the complexity O(m).
- The loop 7-14 computes and prints an interative permutation, where:

Instruction 7 prints an interative permutation with the complexity O(m).

Loop 9 locates the changing position p with the complexity O(m).

Instructions 11-13 find the element a'_p and swap it for a_p with the complexity O(m).

Instruction 14 reverses the subsequence $< a_{p+1}, a_{p+2}, ... , a_m >$ with the complexity O(m).

So the complexity of generating an iterative permutation is O(m).

The total complexity of Algorithm 3.1 is O($c_m.m$).

IV. PARALLEL ALGORITHM GENERATING ITERATIVE PERMUTATIONS

Applying the parallelizing technique based on the output decomposition presented in [8,9] we parallelize Algorithm 3.1. To do so, we split the sequence of all desirable iterative permutations of a multi-set into k subsequences ($k \geq 2$), such that lengths of the subsequences are 'nearly' equal. The number k depends on the number of processors the computing system devotes to computing. Each subsequence will be generated by one computing process executed on one processor. So input and termination condition for each computing process must be already determined. The input of the first process is just the input of the problem and the termination condition of the last process is the one of the algorithm. These k processes use a common algorithm (program) to generate concurrently iterative permutations of the subsequences as the following scheme.

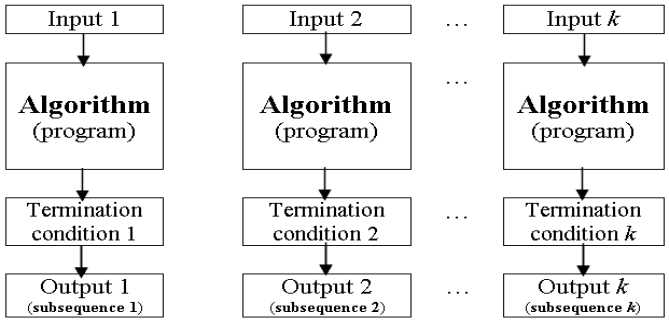

Fig. 4.1. The scheme of a parallel computing organization for generating all iterative permutations

This parallelizing method is an illustration of the output decomposition technique in parallel computing [2].

For simplicity of presentation we perform with $k = 3$. With k greater, one can do analogously.

Assume that $n \geq 3$. This assumption makes our splitting realistic. We split the sequence of all sorted diserable iterative permutations into three subsequences with 'nearly' equal lengths by appropriate choice of the two following basic elements:

$$r = \max\left(1, \max\{\, q \mid \sum_{i=1}^{q} k_i \leq \frac{m}{3} \,\}\right) \text{ and}$$

$$s = \max\left(r+1, \max\{\, q \mid \sum_{i=r+1}^{q} k_i \leq \frac{m}{3} \,\}\right).$$

The first pivot chosen is the first iterative permutation of a block, whose first element is r+1. It is:

$$r+1\ \underbrace{1\,1\ldots1}_{k_1}\ \underbrace{2\ 2\ldots2}_{k_2}\ \ldots\underbrace{r\ r\ldots r}_{k_r}\ \underbrace{r+1\ r+1\ldots r+1}_{k_{r+1}-1}\ \underbrace{r+2\ r+2\ r+2}_{k_{r+2}}\ \ldots\ \underbrace{n\ n\ldots n}_{k_n}$$

It becomes the input of the second computing process. The second pivot chosen is the first iterative permutation of a block, whose first element is s+1. It is:

$$s+1\ \underbrace{1\,1\ldots1}_{k_1}\ \underbrace{2\ 2\ldots2}_{k_2}\ \ldots\underbrace{s\ s\ldots s}_{k_s}\ \underbrace{s+1\ s+1\ldots s+1}_{k_{s+1}-1}\ \underbrace{s+2\ s+2\ s+2}_{k_{s+2}}\ \ldots\ \underbrace{n\ n\ldots n}_{k_n}$$

And it becomes the input of the third computing process. The last iterative permutation of the first subsequence is indeed the last in a block, whose first element is r. Then it is the following iterative permutation:

$$r\ \underbrace{n\ n\ldots n}_{k_n}\ \underbrace{n-1\ n-1\ldots n-1}_{k_{n-1}}\ \ldots\underbrace{r+1\ r+1\ldots r+1}_{k_{r+1}}\ \underbrace{r\ r\ldots r}_{k_r-1}\ \underbrace{r-1\ r-1\ldots r-1}_{k_{r-1}}\ \ldots\ \underbrace{1\,1\ldots1}_{k_1}$$

So the termination for the first computing process is: $A[1] = r \wedge p = 1$.

The last iterative permutation of the first subsequence is indeed the last in a block, whose first element is s. It is just the following iterative permutation:

$$s\ \underbrace{n\ n\ldots n}_{k_n}\ \underbrace{n-1\ n-1\ldots n-1}_{k_{n-1}}\ \ldots\underbrace{s+1\ s+1\ldots s+1}_{k_{s+1}}\ \underbrace{s\ s\ldots s}_{k_s-1}\ \underbrace{s-1\ s-1\ldots s-1}_{k_{s-1}}\ \ldots\ \underbrace{1\,1\ldots1}_{k_1}$$

Thus, the termination for the second computing process is: $A[1] = s \wedge p = 1$.

The sequence of all iterative permutations of the multi-set X is divided into three subsequences as in the following table.

Subsequence	Iterative permutations
1	$\underbrace{1\,1\ldots1}_{k_1}\ \underbrace{2\ 2\ldots2}_{k_2}\ \ldots\ \underbrace{i\ i\ldots i}_{k_i}\ \ldots\ \underbrace{n\ n\ldots n}_{k_n}$ . . . . . $r\ \underbrace{n\ n\ldots n}_{k_n}\ \underbrace{n-1\ n-1\ldots n-1}_{k_{n-1}}\ \ldots\underbrace{r+1\ r+1\ldots r+1}_{k_{r+1}}\ \underbrace{r\ r\ldots r}_{k_r-1}\ \underbrace{r-1\ r-1\ldots r-1}_{k_{r-1}}\ \ldots\ \underbrace{1\,1\ldots1}_{k_1}$
2	$r+1\ \underbrace{1\,1\ldots1}_{k_1}\ \underbrace{2\ 2\ldots2}_{k_2}\ \ldots\underbrace{r\ r\ldots r}_{k_r}\ \underbrace{r+1\ r+1\ldots r+1}_{k_{r+1}-1}\ \underbrace{r+2\ r+2\ r+2}_{k_{r+2}}\ \ldots\ \underbrace{n\ n\ldots n}_{k_n}$ . . . . . $s\ \underbrace{n\ n\ldots n}_{k_n}\ \underbrace{n-1\ n-1\ldots n-1}_{k_{n-1}}\ \ldots\underbrace{s+1\ s+1\ldots s+1}_{k_{s+1}}\ \underbrace{s\ s\ldots s}_{k_s-1}\ \underbrace{s-1\ s-1\ldots s-1}_{k_{s-1}}\ \ldots\ \underbrace{1\,1\ldots1}_{k_1}$
3	$s+1\ \underbrace{1\,1\ldots1}_{k_1}\ \underbrace{2\ 2\ldots2}_{k_2}\ \ldots\underbrace{s\ s\ldots s}_{k_s}\ \underbrace{s+1\ s+1\ldots s+1}_{k_{s+1}-1}\ \underbrace{s+2\ s+2\ s+2}_{k_{s+2}}\ \ldots\ \underbrace{n\ n\ldots n}_{k_n}$ . . . . . $\underbrace{n\ n\ldots n}_{k_n}\ \underbrace{n-1\ n-1\ldots n-1}_{k_{n-1}}\ \ldots\ \underbrace{i\ i\ldots i}_{k_i}\ \ldots\ \underbrace{1\,1\ldots1}_{k_1}$

These three subsequences are generated concurrently by three parallel computing processes, with corresponding input and termination condition, executed concurrently on three processors P_1, P_2 and P_3 as follows.

P_1

```
Begin
  input the number of basic elements n ;
  input multiplicities k_i , i = 1, 2, ..., n ;
  $m \leftarrow \sum_{i=1}^{n} k_i$ ;
  $r \leftarrow \max(1, \max\{ q \mid \sum_{i=1}^{q} k_i \le \frac{m}{3} \})$ ;
  $s \leftarrow \max(r+1, \max\{ q \mid \sum_{i=r+1}^{q} k_i \le \frac{m}{3} \})$ ;
  $A[1..m] \leftarrow \underbrace{1\ 1 \dots 1}_{k_1}\ \underbrace{2\ 2 \dots\ 2}_{k_2} \dots \underbrace{i\ i \dots\ i}_{k_i} \dots \underbrace{n\ n \dots\ n}_{k_n}$ ;
  repeat
      print the array A[1..m] ;
      p ← m - 1 ;
      while A[p] >= A[p+1] do p ← p - 1;
      if p > 0 then
          { i ← p +1 ;
            while A[i] > A[p] do i ← i +1 ;
            swap A[p] for A[i-1] ;
            reverse the subsequence A[p+1..m] ; }
   until A[1] = r ∧ p = 1 ;
 End .
```

Generating iterative permutations of the first subsequence

P_2

```
Begin
  input the number of basic elements n ;
  input multiplicities k_i , i = 1, 2, ..., n ;
  $m \leftarrow \sum_{i=1}^{n} k_i$ ;
  $r \leftarrow \max(1, \max\{ q \mid \sum_{i=1}^{q} k_i \le \frac{m}{3} \})$ ;
  $s \leftarrow \max(r+1, \max\{ q \mid \sum_{i=r+1}^{q} k_i \le \frac{m}{3} \})$ ;
  $A[1..m] \leftarrow r+1\ \underbrace{1\ 1 \dots 1}_{k_1}\ \underbrace{2\ 2 \dots 2}_{k_2} \dots \underbrace{r\ r \dots r}_{k_r}\ \underbrace{r+1\ r+1 \dots r+1}_{k_{r+1}-1}\ \underbrace{r+2\ r+2\ r+2}_{k_{r+2}} \dots \underbrace{n\ n \dots n}_{k_n}$ ;
  repeat
      print the array A[1..m] ;
      p ← m - 1 ;
      while A[p] >= A[p+1] do p ← p - 1;
      if p > 0 then
          { i ← p +1 ;
            while A[i] > A[p] do i ← i +1 ;
            swap A[p] for A[i-1] ;
            reverse the subsequence A[p+1..m] ; }
   until A[1] = s ∧ p = 1 ;
 End .
```

Generating iterative permutations of the second subsequence

P_3

```
Begin
  input the number of basic elements n ;
  input multiplicities k_i , i = 1, 2, ..., n ;
  $m \leftarrow \sum_{i=1}^{n} k_i$ ;
  $r \leftarrow \max(1, \max\{ q \mid \sum_{i=1}^{q} k_i \le \frac{m}{3} \})$ ;
  $s \leftarrow \max(r+1, \max\{ q \mid \sum_{i=r+1}^{q} k_i \le \frac{m}{3} \})$ ;
  $A[1..m] \leftarrow s+1\ \underbrace{1\ 1 \dots 1}_{k_1}\ \underbrace{2\ 2 \dots 2}_{k_2} \dots \underbrace{s\ s \dots s}_{k_s}\ \underbrace{s+1\ s+1 \dots s+1}_{k_{s+1}-1}\ \underbrace{s+2\ s+2\ s+2}_{k_{s+2}} \dots \underbrace{n\ n \dots n}_{k_n}$ ;
  repeat
      print the array A[1..m] ;
      p ← m - 1 ;
      while A[p] >= A[p+1] do p ← p - 1;
      if p > 0 then
          { i ← p +1 ;
            while A[i] > A[p] do i ← i +1 ;
            swap A[p] for A[i-1] ;
            reverse the subsequence A[p+1..m] ; }
   until p = 0 ;
 End .
```

Generating iterative permutations of the third subsequence

So the time for generating all iterative permutations of a multi-set reduces to one third.

V. AN APPLICATION: OPTIMAL ARRANGEMENT OF TASKS

Given k_1 tasks e_1, k_2 tasks e_2, ... and k_n tasks e_n with the same time for performance, e.g. one day. The tasks are performed uninterruptedly. At a moment only one task is being performed. For each task e_i (i = 1,2,...,n), d_i is the deadline, x_i is the award money per day as early finish and y_i is the mandate money per day as late finish. Arrange these tasks, such that the money obtained is as most as possible.

This problem is an extention of the job arrangement problem presented in [4], which is solved by a matroid. For our problem, we can not create any matroid.

Denote $m = \sum_{i=1}^{m} k_i$. We have m tasks. The set of tasks may be represented by a multi-set:

$$X = (k_1 * e_1, k_2 * e_2, ..., k_n * e_n).$$

Identify the task $e_1 \equiv 1$, task $e_2 \equiv 2$, ..., task $e_n \equiv n$. Each arrangement of the tasks is indeed an iterative permutation of X. It is represented by an integer sequence $< a_1\ a_2\ ...\ a_{m\text{-}1}\ a_m >$ of the length m, with k_1 integers 1, k_2 integers 2, ... and k_n integers n.

Construct an award/mandate function for the task a_j as follows:

$$tp(a_j) = \begin{cases} (d_i - j)x_i & \text{, if } a_j = e_i \text{ and } j < d_i \\ (d_i - j)y_i & \text{, if } a_j = e_i \text{ and } j > d_i \end{cases}$$

With an arrangement as the iterative permutation $< a_1 a_2 \ldots a_{m-1} a_m >$ the total money obtained is:

$$tg = \sum_{j=1}^{m} tp(a_j)$$

The optimal problem of task arrangement is derived to the problem of finding an iterative permutation with the most *tg*.

Combining Algorithm 3.1 and the greatest number finding algorithm we have an algorithm for optimal arrangement of tasks.

We may use the parallel algorithm presented in Part 3 to get result more quickly, if the computing system allows.

Example 5.1: Given tasks as in the following table:

Task e_i	Quantity k_i	Deadline d_i	Award x_i	Mandate y_i
1	2	5	3	2
2	3	4	5	3
3	2	2	10	4

We have a multi-set X = (2 * e_1, 3 * e_2, 2 * e_3). The set has 210 iterative permutations. It is all possible variants to arrange the tasks.

Computing by the above algorithm we get the following optimal variant: 3 2 2 3 2 1 1, after which the greatest money obtained is 8.

VI. Conclusion

In this paper we proposed a new algorithm to generate all permutations of a standard set. Then we extended the notion of permutation to iterative permutation and have constructed a new simple algorithm to generate all iterative permutations of a multi-set.

Applying the parallelizing method based on output decomposition, we parallelized this new algorithm by splitting the sequence of all iterative permutations into subsequences with 'nearly' equal lengths as our approach presented in [8,9]. Therefore, the amount of time required for finding all the iterative permutations of a multi-set will be drastically decreased by the number of subsequences. The computing organization is an association of the bottom-up design and the divide and conquer one. We applied this algorithm to solve an optimal problem of task arrangement.

We keep investigating combinatories on multi-sets and apply them to huge computing problems in data mining, time-series data matching as well as in system controls.

Acknowledgement

The authors would like to acknowledge Vietnam National University, Hanoi as the sponsors for the research project.

References

[1] J. Ginsburg, *Determining a permutation from its set of reductions*, Ars Combinatoria, No. 82, 2007, pp. 55-57

[2] A. Grama, A. Gupta and G. Karypis, V. Kumar, *Introduction to Parallel Computing*, Addison-Wesley, 2003

[3] T. Kuo, *A new method for generating permutations in lexicographic order*, Journal of Science and Engineering Technology, Vol. 5, No. 4, 2009, pp. 21-20

[4] W. Lipski, *Kombinatoryka dla programistów*, WNT, Warszawa, 1982

[5] M. Monks, *Reconstructing permutations from cycle minors*, The Electronic Journal of Combinatorics, No. 16, 2009, #R19

[6] D. Singh, A.M. Ibrahim, T. Yohanna and J.N. Singh, *An Overview of the applications of Multisets*, Novi Sad Journal of Mathematics, Vol. 37, No. 2, 2007, pp. 73-92

[7] J. Steinhardt, *Permutations with ascending and descending blocks*, The Electronic Journal of Combinatorics, No. 17, 2010, #R14

[8] H.C. Thanh and N.Q. Thanh, *An Efficient Parallel Algorithm for the Set Partition Problem*, Studies in Computational Intelligence, Springer, Vol. 351, 2011, pp. 25-32

[9] H.C. Thanh, *Parallel Generation of Permutations by Inversion Vectors*, Proceedings of IEEE-RIVF International Conference on Computing and Communication Technologies, IEEE, 2012, pp. 129-132