Annales UMCS Informatica AI X, 2 (2010) 21-28
DOI: 10.2478/v10065-010-0049-1

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Parallelizing a new algorithm for the set partition problem

Hoang Chi Thanh[1*]

Department of Informatics, Hanoi University of Science, VNUH,
334 - Nguyen Trai Rd., Thanh Xuan, Hanoi, Vietnam.

Abstract – In this paper we propose a new approach to organizing parallel computing to find a sequence of all solutions to a problem. We split the sequence into subsequences and then execute concurrently the processes to find these subsequences. We propose a new simple algorithm for the set partition problem and apply the above technique for this algorithm.

1 Introduction

When solving a computer science problem we have to construct a proper algorithm with the deterministic input and output. The algorithm is programmed. The input is put into a computer. The computer performs the corresponding program to hold all the solutions to the problem after a period of time.

The scheme for computing problem solutions is as presenting in the Fig. 1.

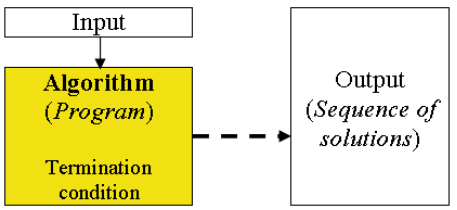

Fig. 1. The scheme for computing problem solutions.

*thanhhc@vnu.vn

The practical meaning of a problem and an algorithm is better if the time period of computing is shorter. One of good methods to reduce computing time is to organize parallel computing where the computation environment allows. There are some methods of organizing parallel computing to find quickly all solutions of a problem, for example, constructing a sequential algorithm first and then transforming it into the concurrent one [6], splitting data into separate blocks and then computing concurrently using the blocks [7]. Such parallel computing is an illustration of the top-down design.

For many problems, we can get to know quite well a number of all desirable solutions and their arrangement in a sequence. The set partition problem [2, 3, 5] is a typical example. This problem is broadly used in the graph theory [1], in the concurrent systems [4]. So we can split the sequence of all desirable solutions into subsequences and use a common program (algorithm) in the parallel computing environment to find these subsequences concurrently. Therefore, the amount of time required for finding all the solutions will be drastically decreased by the number of subsequences. This computing organization is combination of the bottom-up as well as divide and conquer designs.

2 Parallel computing of problem solutions by partitioning

To perform the above parallel computing we split the sequence of all desirable solutions of a problem into subsequences. The number of subsequences depends on the number of calculating processors. Let us split the sequence of all solutions of the problem into m subsequences ($m \geq 2$). The scheme of the parallel computing organization to find all solutions of a problem is illustrated in the Fig. 2.

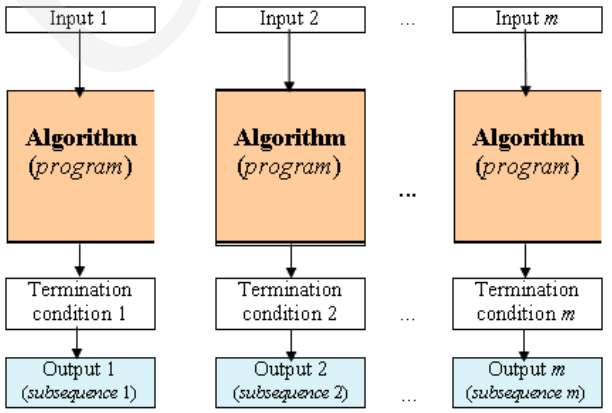

Fig. 2. The scheme of the parallel computing organization to find problem solutions.

In order to make the parallel computing organization realistic and optimal, the subsequences of solutions should satisfy the two following criteria:

(1) It is easy to determine the input and the termination condition for the computing process of each subsequence.

(2) The smaller is the difference between the subsequences length the better.

Of course, input 1 is the input of the problem and the last termination condition is that of the algorithm.

The first criterion ensures that the partition of solutions is realistic. In many cases, we can use (a part of) the input of the computing process for the next subsequence as the termination condition of the computing process for the previous subsequence. The second criterion implements the balance of computing to the processors. Then the parallel computing processes become optimal.

If the problem has many algorithms, we should choose the algorithm with the least number of variables. Just then, the input and the termination condition of computing processes of subsequences become simple.

3 Application to the set partition problem

We apply the above presented technique to the set partition problem. For simplicity of presentation, we split the sequence of all desirable solutions into two subsequences. If we want to split the sequence into many subsequences then we apply this technique to each subsequence.

3.1 Set partition problem

Let X be a given set.

Definition 1. A partition of the set X is a family $A_1, A_2, \ldots, A_k$ of subsets of X, satisfying the following properties:

(1) $A_i \neq \varnothing, 1 \leq i \leq k$;
(2) $A_i \cap A_j = \varnothing, 1 \leq i \leq j \leq k$;
(3) $A_1 \cup A_2 \cup \ldots \cup A_k = X$.

Problem: Find all partitions of a given set X.

The set partition problem is broadly used in the graph theory [1], in the concurrent systems [4]s. The number of all partitions of an n element set is denoted by the Bell number B_n, calculated by the following recursive formula [3, 5]:

$$B_n = \sum_{i=0}^{n-1} \binom{n-1}{i} B_i, \quad where \quad B_0 = 1$$

The number of all solutions of the problem grows up as quickly as the factorial function does (see Table 1).

Let identify the set $X = \{1, 2, 3, \ldots, n\}$. Let $\pi = \{A_1, A_2, \ldots, A_k\}$. Each subset A_i is called a block of the partition π. To ensure the uniqueness of representation, blocks in a partition are sorted in the ascending order from the smallest element in the block. In a partition,

Table 1. The number of the problem solutions.

n	B_n
1	1
3	5
5	52
8	4.140
10	115.975
15	1.382.958.545
20	51.724.158.235.372

the block $A_i (i = 1, 2, 3, \ldots)$ has the index i and element 1 always belongs to the block A_1. Each element $j \in X$, belonging to a block A_i has also the index i. It means, every element of X can be represented by the index of a block that includes the element. Of course, the index of element j is not greater than j. Each partition can be represented by a sequence of n indices. The sequence can be considered as a word with the length of n on the alphabet X. So we can sort these words in the ascending order. Then:

- The smallest word is 1 1 1 . . . 1. It corresponds to the partition {1,2,3, . . ., n}. This partition consists of one block only.
- The largest word is 1 2 3 . . . n. It corresponds to the partition {1}, {2}, {3}, . . ., {n}. This partition consists of n blocks, each block has only one element. This is an unique partition that has a block with the index n.

Theorem 1. For every positive integer $n, B_n \leq n!$. It means, the number of all n element set partitions is not larger than the number of all permutations on the same set.

PROOF. It follows from the index sequence representation of partitions. □

We use an integer array A*I*[1..n] to represent a partition, where A*I*[i] stores the index of the block that includes element i. Element 1 always belongs to the first block, element 2 may belong to the first or second block. If element 2 belongs to the first block then element 3 may belong to the first or second block only. If element 2 belongs to the second block then element 3 may belong to the first, second or third block. Hence, the element i may only belong to the blocks:

$$1, 2, 3, \ldots, max(AI[1], AI[2], \ldots, AI[i-1]) + 1.$$

It means, for every partition:

$$AI[i] \leq max(AI[1], AI[2], \ldots, AI[i-1]) + 1 \leq i, i = 2, 3, \ldots, n.$$

This is an invariant for all partitions of the set X. We use it to find the partitions.

Example 1. We use it to find the partitions presented in Table 2.

Table 2. Partitions of 3-element set.

No	Partition	A*I*[1..3]
1	{1, 2, 3}	1 1 1
2	{1, 2}, {3}	1 1 2
3	{1, 3}, {2}	1 2 1
4	{1}, {2, 3}	1 2 2
5	{1}, {2}, {3}	1 2 3

3.2 A new algorithm for partition generation

It is easy to determine a partition from its index array representation. So, instead of finding all partitions of the set X we find all index arrays AI[1..n], each of them can represent a partition of X. These index arrays will be sorted in the ascending order.

The first index array is 1 1 1 . . . 1 and the last index array is 1 2 3 . . . n. So the termination condition of the algorithm is A*I*[n] = n .

Let A*I*[1..n] be an index array representing a partition of X and let A*I*'[1..n] denote the index array next to A*I* in the ascending order.

To find the index array AI' we use an integer array Max[1..n], where Max[i] stores max(A*I*[1], A*I*[2], . . ., A*I*[i-1]). The array Max gives us possibilities to increase indices of the array A*I*. Of course,

$$Max[1] = 0 \text{ and } Max[i] = max(Max[i-1], AI[i-1]), i = 2, 3, \ldots, n.$$

Then,

$$AI'[i] = AI[i], i = 1, 2, \ldots, p-1, \text{ where } p = min\{q | AI[j] = Max[j] + 1, q \leq j \leq n\},$$
$$AI'[p] = AI[p] + 1 \text{ and } AI'[j] = 1, j = p+1, p+2, \ldots, n.$$

Based on the above properties of the index arrays, we construct the following algorithm for finding all partitions of a set.

Algorithm 1. (Generation of set partitions)

Input: A positive integer n.

Output: A sequence of an n element of set partitions, whose index representations are sorted by ascending.

Computation:

```
1  Begin
2        AI[1..n] ← 1 ;
3        Max[1] ← 0 ;
```

```
4       Print the partition ;
5       repeat
6            for  i ← 2 to n  do
7                 if  Max[i−1] < AI[i−1]  then  Max[i] ← AI[i−1]
                     else  Max[i] ← Max[i−1] ;
8            p ← n ;
9            while  AI[p] = Max[p] + 1  do  p ← p − 1 ;
10           AI[p] ← AI[p] + 1 ;
11           for  i ← p + 1 to  n  do  AI[i] ← 1 ;
12           Print the partition ;
13      until  AI[n] = n ;
14  End.
```

The algorithm complexity: The algorithm finds an index array and prints the corresponding partition with the complexity of O(n).

Therefore, the total complexity of the algorithm is O(B_n, n). It approximates O(nn!).
Algorithm 3.2 is much simpler and better than pointer-based algorithm 1.19 presented in [3].

3.3 Parallel generation of partitions

To paralellize the above presented sequential algorithm, we split the sequence of desirable all partitions of the set X into two subsequences. The pivot is chosen as a partition represented by the index array:

$$123...[n/2]-1[n/2]11\ldots112$$

So, the last partition of the first subsequence corresponds to the following index array:

$$123...[n/2]-1[n/2]11\ldots111$$

The chosen pivot and the last index array of the first subsequence are illustrated in the Fig. 3.

We have to determine the termination condition for the first computing process and the input of the second one.

The termination condition for the first computing process in instruction 13 is replaced by:

$$AI[i] = i, i = 2, 3, \ldots, [n/2]-1, [n/2].$$

The input of the second computing process in instruction 2 will be:

$$AI[i] \leftarrow i, i = 2, 3, \ldots, [n/2]-1, [n/2];$$
$$AI[j] \leftarrow 1, j = [n/2]+1, [n/2]+2, \ldots, n-1;$$
$$AI[n] \leftarrow 2;$$

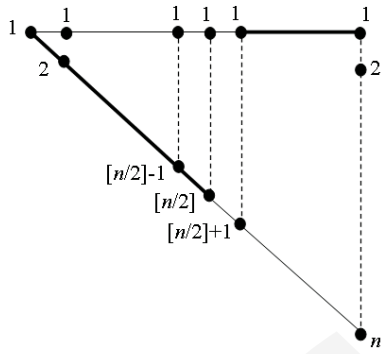

Fig. 3. The pivot and the last index array of the first subsequence.

The experimental results indicate that the pivot is nearly in the middle of the sequence of all index arrays. So the above splitting is appropriate.

4 Conclusions

In this paper we propose a new approach to organize parallel computing for finding all solutions of a problem, whose sequential algorithm takes too long to find them. The parallel computing organization presented above is combination of the bottom-up as well as divide and conquer designs. We also propose a new efficient and simple algorithm for the set partition problem and paralellizing the algorithm.

In future, we will apply this technique to other problems, namely time-series processing, time-series matching, scheduling problem and system control.

5 Acknowledgement

This work was supported by the Asia Research Center, Vietnam National University, Hanoi.

References

[1] Cameron K., Eschen E. M., Chinh T. Hoang, Sritharan R., The list partition problem for graphs, Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (New Orleans, 2004): 391–399.

[2] Cornen T. H., Leiserson C. E., Rivest R. L., Stein C., Introduction to Algorithms (The MIT Press, 2001).

[3] Lipski W., Kombinatoryka dla Programistów (WNT, Warszawa, 1982).

[4] Rajasekaran S., Lee I., Parallel algorithms for relational coarsest partition problems, IEEE Trans. Parallel Distributed Systems 9(7) (1998): 687–699.

[5] Thanh H. C., Combinatorics (in Vietnamese) (VNUH Press, 1999).

[6] Thanh H. C., Transforming sequential processes of a net system into concurrent ones, International Journal of Knowledge-based and Intelligent Engineering Systems 11(6) (2007): 391–397.

[7] Thanh H. C., Parallel dimensionality reduction transformation for time-series data, Proceedings of the 1st Asian Conference on Intelligent Information and Database Systems, IEEE Computer Society (Dong Hoi, 2009): 104–108.