Associative Processor as Means of Disjoint Sets Representation and Dynamic Connectivity Problem Solving

Nikolai L. Verenik
and Mikhail M. Tatur
Belarusian State University of Informatics and Radioelectronics
Minsk, Belarus
nick.verenik@gmail.com
tatur@i-proc.com

***Abstract*—The article briefly describes data structures and algorithms for working with disjoint sets. The use of an associative processor with an original architecture for representing disjoint sets and solving the dynamic connectivity problem with the help of adapted quick-find algorithm is proposed. The presented approach allows to achieve constant running time of basic union-find operations and linear complexity for processing sequence of union-find operations in any order.**

***Keywords*—disjoint set, union-find, quick-find, parallel computing, vector processor, associative memory**

I. Dynamic connectivity problem

We are given sequence of pairs of integers where every integer represents an object of some type and the pair p-q means "p is connected to q". Assume that the relation p-q is *equivalent*, i.e. has the following properties:

- *reflexivity* (p is connected to p);
- *symmetry* (if p is connected to q, then q is connected to p);
- *transitivity* (if p is connected to q and q is connected to r, then p is connected to r).

The problem is to develop an efficient data structure for storing sufficient information about the input pairs to be able to decide at any time whether an arbitrary pair of objects p and q is connected. The example of processing a sequence of pairs is presented in Fig. 1.

When processing each new pair of objects from the input, we need to determine whether it represents a new connection, and then integrate the information about the detected connection into the data structure to be able to test connections in the future.

This problem is actually a fundamental computational task that arises in a variety of applications, from percolation in physical chemistry to connectivity in large communications networks. These are few of subject areas where the data structure of such kind can serve to represent objects [1]:

- computers in a network;
- friends in a social network;
- transistors in a chip;
- pixels in a digital image;
- variable-name-equivalence problem in some programming languages;
- metallic sites in a composite system;
- elements in a mathematical set etc.

The difficulty with these applications is a possible need to process millions of objects and billions or more of connections. Thus, the following properties are common for the given problem:

- number of objects N can be huge;
- number of operations M can be huge;
- find and union commands are called in random order.

The dynamic connectivity problem can be reduced to the following two abstract operations that we need to implement:

1) CONNECTED(p, q) - determines whether the pair of objects p and q is connected.
2) UNION(p, q) - replaces the sets containing the objects p and q by their union. We define *connected components* as maximal set of objects connected with each other.

Thus, the procedure for processing the input sequence of pairs of integers can look like this:

```
1: for all (p-q) do
2:     if not CONNECTED(p, q) then
3:         UNION(p, q)
4:     end if
5: end for
```

II. Disjoint sets and operations on them

The generalization of the connectivity problem is *disjoint-set data structure* (or union-find data structure). Such a structure maintains a collection $S = \{S_1, S_2, ..., S_k\}$ of disjoint dynamic sets, where each set is identified by a *representative*, which is a certain element of the set. The element that used as a representative depends on the applied problem and doesn't always matter.

The data structure for disjoint sets should support the following operations [2]:

- MAKE-SET(p) creates a new set consisting of one element (and thus representative) that is p. Since sets are

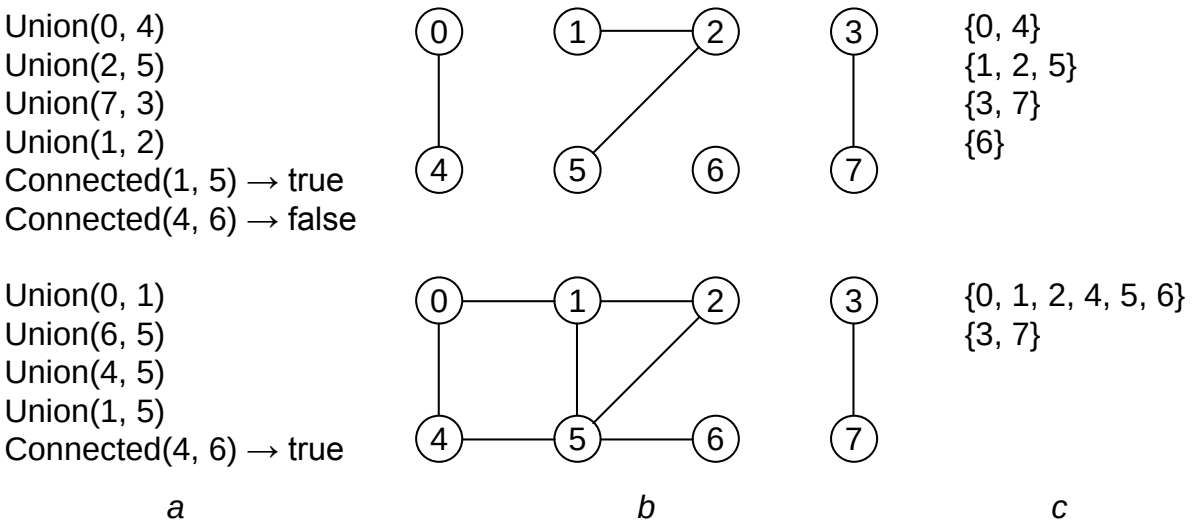

Figure 1. An example of processing a sequence of pairs of integers. The sequence of the executed commands (a); graphical representation of the data structure (b); connected components (c)

disjoint, it is required that p is not already in some other set.

- UNION(p, q) unites dynamic sets that contain p and q (denoted by S_p and S_q) into a new set. Assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any element of $S_p \cup S_q$, although many implementations of UNION choose the representative of either S_p or S_q as the new representative. Since we require all sets to be disjoint, UNION operation must conceptually destroy the sets S_p and S_q, removing them from the collection S. In practice, the elements of one of the sets are absorbed into the other set.
- FIND-SET(p) returns a pointer to the representative of the (unique) set which contains the element p.

The problem of implementing disjoint-sets data structure has been well developed by a number of researchers. The convention that both elements and sets will be identified by integer values between 0 and $N-1$, so a simple linear array $id[]$ is used as basic data structure to represent the sets. Initially, we start with N sets, each element in its own set, so we initialize $id[i]$ to i for all i from 0 to $N-1$. For each element i, we keep the information needed by FIND-SET(p) method to determine the set containing element i using various algorithm-dependent strategies. As a result CONNECTED are reduced to simple check FIND-SET(p)== FIND-SET(q).

The running time of the disjoint-set data structures is usually analyzed in terms of two parameters: N, the number of MAKE-SET operations, and M, the total number of MAKE-SET, UNION and FIND-SET operations. Assume that N MAKE-SET operations are the first N operations performed (during the initialization process), and the number of UNION operations cannot exceed $N-1$ (since the sets are disjoint by definition). Table I provides a summary of the most widely known algorithms with their worst-case running time cost [1].

III. QUICK-FIND ALGORITHM IMPLEMENTATION USING ASSOCIATIVE PROCESSOR

Consider the *quick-find* algorithm. The basis of this algorithm is an array of integers $id[]$ with the property that pair of elements p and q are connected if and only if the $id[p]$ and $id[q]$ values of array are equal. This method is called quick-find because FIND-SET(p) just needs to return $id[p]$ to complete the operation. Then to implement UNION(p, q) it is enough to replace all entries in the array corresponding to both sets by the same value, yet for that we need to go through the whole array.

Fig. 2 shows an example of processing the sequence of pairs of integers used in Fig. 1. The code is quite straightforward:

```
UNION(p, q)
1: r ← id[p]
2: for i = 0 to N − 1 do
3:     if id[i] = r then
4:         id[i] ← id[q]
5:     end if
6: end for
```

```
MAKE-SET(p)
1: id[p] ← p

FIND-SET(p)
1: return id[p]
```

As far as we can see the main disadvantage of the quick-find algorithm is UNION operation which requires seeing the entire data array $id[]$. It will take N^2 accesses to the array in order to

Table I
M UNION-FIND OPERATIONS ON A SET OF N OBJECTS

Algorithm	MAKE-SET	UNION	FIND-SET	Worst-case time
Quick-find (QF)	N	N	1	MN
Quick-union (QU)	N	N	N	MN
Weighted QU (WQU)	N	$\lg N$	$\lg N$	$N + M \lg N$
QU + path compression (QUPC)	N	$\lg N$	$\lg N$	$N + M \lg N$
WQU + path compression (WQUPC)	N	$\lg N$	$\lg N$	$N + M \lg^* N$

*Iterated logarithm

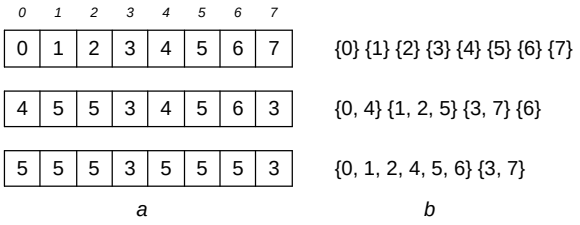

Figure 2. Representation of data in the quick-find algorithm. Elements in data array (a); connected components (b)

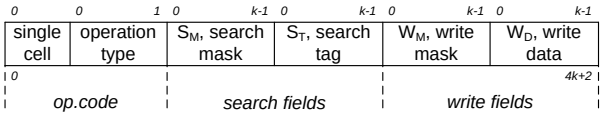

Figure 3. The associative processor command

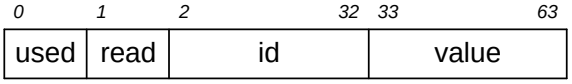

Figure 4. Interpretation of the memory cell data (bitwidth $k = 64$ bits)

process a sequence of N union operations for a structure with N objects, what is prohibitively expensive. That is the reason why quick-find can not be used to solve real-life problems with a huge number of objects and better algorithms (WQUPC, see Table I) are used. However, if we assume that the operation of writing to the data array can be performed synchronously for all elements in constant time, then the efficiency of this simple algorithm will drastically change.

In the articles [3] the architecture of the developed associative processor was considered. In essence, it is a SIMD class processor in which all memory is evenly distributed between a set of PE. Each PE is responsible for accessing its memory cells, implemented as a function of comparing the contents of the cell with the input word. As a result, the entire sequence of simple PEs can synchronously perform operations on all memory cells or over a selected set of associative memory words. We can consider the processor of this type as an intelligent memory with a specific interface for data access. Like an ordinary memory, all the functionality that the processor provides is reading and writing data.

Fig. 3 shows the only command of the associative processor. To address memory cells instead of a fixed address a search operation is used where the input parameters are the S_M and S_T fields, the mask and the search tag respectively. The PE compares the contents of each cell with the value of the search tag S_T according to the bit mask S_M. In case of equality, the cell is considered active and the selected binary operation (field *"operation type"*) is applied to it, in the simplest case corresponding to assignment . The write operation is also not applied to all bits in the memory cell, but only to the marked by W_M mask. The bit values are taken from the W_D field. The *"single cell"* command bit is used when it is necessary to guarantee the activation of only one memory cell, which is necessary, for example, to implement sequential readout of data from the processor memory.

It is proposed to use the associative processor of this type as a basis data structure when solving the dynamic connectivity problem by the quick-find algorithm.

The data format of the processor's memory cell is shown in Fig. 4. It is not used by associative processor itself as processor operates only at bit-level without resorting to upper-level abstractions. Thus, this is only one of the options for interpreting the cell data by a programmer, chosen for the convenience of solving a particular problem. The *"used"* bit is used to identify free memory cells so we can initialize them one by one. The *"read"* bit for a given problem is not used, but is reserved for implementing data readout from the processor. The *"id"* field corresponds to the element identifier or array index in the basic implementation of the quick-find algorithm. The *"value"* field contains the representative of the set that contains the element. With the 64-bit memory cell specified in the example, it is possible to address $2^{31} = 2$ billion objects.

All necessary operations for working with disjoint sets data structure can be implemented at the application level. Table II presents an example of such an implementation for the selected data format (Fig. 4). For example, INIT reset S_M mask field to zero so it won't be used (hence it doesn't matter what we send in S_T field) and all memory cells will be addressed. Similarly, W_M mask field is set to 0xFF to fill these addressed cells

Table II
OPERATIONS FOR WORKING WITH A DISJOINT SETS DATA STRUCTURE IMPLEMENTED USING THE ASSOCIATIVE PROCESSOR

	s.cell	op.type	S_M				S_T				W_M				W_D			
			used	read	id	value	used	read	id	value	used	read	id	value	used	read	id	value
1	0	00	0	0	0x00	0x00	-	-	-	-	1	1	0xFF	0xFF	0	0	0x00	0x00
2	1	00	1	0	0x00	0x00	0	-	-	-	1	0	0xFF	0xFF	1	-	p	p
3	1	00	1	0	0xFF	0x00	1	-	p	-	0	0	0x00	0x00	-	-	-	-
4	0	00	1	0	0x00	0xFF	1	-	-	id[p]	0	0	0x00	0xFF	-	-	-	id[q]

[1] INIT() fills memory cells with zeros.
[2] MAKE-SET(p) writes the element p to the first available memory cell.
[3] FIND-SET(p) returns the last addressed memory cell that is $id[p]$.
[4] UNION2($id[p], id[q]$) writes $id[q]$ value to all memory cells where $id[p]$ is stored.

with zeros (see W_D). The other thing to mention is that union method is a bit different from the initial one (here it is called UNION2). The method takes the two representatives of the sets to unite instead of the original pair of elements identifiers.

Listing 1. Quick-find algorithm using the associative processor (C++)

```
void main() {
  Init();
  for (int i = 0; i < n; ++i) MakeSet(i);
  while (std::cin >> p >> q) {
    int r1 = FindSet(p);
    int r2 = FindSet(q);
    if (r1 != r2) Union2(r1, r2);
  }
}
```

CONCLUSION

A weighted quick-union algorithm is generally used for solving disjoint-set problems. It has number of various modifications (such as path compression) and as a result worst-case running time $O(N + M \lg N)$ is achieved. Analysis can be improved to $O(N + M\alpha(M, N))$ [4] [5], where $\alpha(M, N)$ – a very slowly growing function. In practice, the running time is almost linear.

On the other hand the presented associative processor allows us to achieve constant running time for basic operations (see Table I). As a result, the cost of performing M consecutive operations is $\Theta(M)$. Basically, the running time depends solely on the number of processed M operations. Thus, the most important characteristic is the command processing time which depends on actual hardware implementation.

The running time to process single operation, or the time to process N k-bit words in the associative computing system is given by [6]:

$$T = k \times t \times (\frac{N}{n} + K),$$

where t - time of associative memory cycle; n - number of PE; K - factor of complexity of performing a single elementary operation (the number of sequential steps accessing memory). Thus, the time of processing T is constant and depends on the value of N/n, i.e. the number of memory cells per PE. After fixing the value of N/n (GPU threads configuration or scheme adjustment for PE on FPGA) the time of processing remains constant regardless of the total amount of memory being processed.

In the developed software simulation model based on the GPU cluster, a considerable amount of time is spent not so much on the processor but on the exchange of messages between the processor and the host program. The next stage of development is planned to move to a hardware prototype of the processor and possible modification of the architecture in order to reduce the costs associated with the data transmitting.

REFERENCES

[1] R. Sedgewick and K. Wayne, Algorithms. 4th ed. Addison-Wesley, 2011. 992 p.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to algorithms. 3rd ed. London, England: The MIT Press, 2009. 1312 p.
[3] N. L. Verenik, A. I. Girel, Y. N. Seitkulov, M. M. Tatur and H. P. Razhkova, "Cognitive information processing based on a parallel processor," 10th International Conference on Digital Technologies (DT '2014), pp.356–360, 2014.
[4] R. E. Tarjan, Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics. CBMS-NSF Regional Conference Series in Applied Mathematics (Book 44), 1987. 140 p.
[5] A. V. Aho, J. E. Hopcroft and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974. 470 p.
[6] B. Tsilker and S. Orlov, Organizatsiya EVM i sistem: Uchebnik dlya vuzov [Organization of computers and systems: Students textbook], St. Petersburg, Piter, 2007. 668 p.

ПРЕДСТАВЛЕНИЕ СИСТЕМЫ НЕПЕРЕСЕКАЮЩИХСЯ МНОЖЕСТВ И РЕШЕНИЕ ЗАДАЧИ СВЯЗНОСТИ СРЕДСТВАМИ АССОЦИАТИВНОГО ПРОЦЕССОРА

Вереник Н. Л., Татур М. М.
Белорусский государственный университет информатики и радиоэлектроники,
г. Минск, Республика Беларусь

В статье кратко рассмотрены структуры данных и алгоритмы для работы с системой непересекающихся множеств. Предлагается использовать ассоциативный процессор с оригинальной архитектурой для представления непересекающихся множеств и решения задачи связности с помощью адаптированного алгоритма быстрого поиска. Представленный подход позволяет достичь константного времени выполнения базовых операция поиска и объединения и линейной сложности для обработки произвольной последовательности операций поиска и объединения.