Branches in the Dataflow Metalanguage UPL (METAL) and Methods of their Implementation in the PDCS “Buran”

A.V. Klimov, N.N. Levchenko

Institute for Design Problems in Microelectronics (IPPM) of RAS, Moscow,
klimov@ippm.ru, nick@ippm.ru

Abstract — Branches in the dataflow language for parallel programming UPL (Universal Parallel Language), earlier called METAL, are considered as well as methods of their implementation for Parallel Dataflow Computing System (PDCS) “Buran”. A node with P input ports can have several branches, each of which is activated by a subset of input *tokens* sent on the node’s ports. This feature allows for flexible controlled nondeterminism of algorithms. The use of branches is demonstrated by five dining philosophers problem as an example. Also, the necessary information on the semantics of UPL is presented.

For implementation of branches the matching device must be able to detect whether tokens are present on all the required (for a particular branch) input ports. To provide this ability, tokens should contain some extra bits of information (in addition to the port number and the usage bit indicating that the port is used by a branch). Three methods to encode this information into tokens are considered.

The first two methods are simple but expensive regarding the token size. The last method requires just k bits for each branch provided that the maximum number of ports per branch p_m is bounded by $A(2^k, 0)$, where $A(m, n)$ is very rapidly growing Ackerman function. To compare: the first two ways require that $p_m \leq 2^k$. For each method, an algorithm is described to determine whether a set of accepted tokens is sufficient to activate a particular branch. A valuable result of the article is the formulation of the very problem, in the solution of which the Ackermann function is involved.

Keywords — dataflow computation model, dataflow language, nodes with branches, activation on subset of tokens, dining philosophers, Join calculus, token structure, Ackerman function.

I. INTRODUCTION

Branches is a feature of the dataflow meta-language UPL [1] (Universal Parallel Language) that we design for parallel programming. This mechanism allows node to fire not only when data is available at all its inputs, but also when data tokens are present on a subset of inputs. There can be several such subsets specified, each calling its proper *branch* of the node’s program. The article discusses both the mechanism of the branches in the UPL language and the methods of its implementation in the Buran, a parallel dataflow computing system (PDCS) [2]. The use of branches allows one to introduce flexible, controlled non-determinism into the algorithms.

The UPL language has both text and graphic forms. This article uses the text one. The graphic form has been

presented in [1], [3]. The whole program is a set of descriptions of nodes, together constituting the dataflow graph of the algorithm. The node description contains a node name, a list of port names with data types on them, a list of index component names and a node program. Usually index is a tuple of integers. The links between nodes in the algorithm graph are presented by the token sending operators encountered in the node programs.

When executed, instances of nodes, also called virtual nodes, are generated, run and disappear, each with its own index value. At the same time, we can assume that they all initially “virtually” exist, without consuming (unlike arrays) any resources.

When activated, virtual nodes perform some calculations and send the results in the form of tokens to each other’s ports. The virtual node is activated when tokens with input values have come to all its ports. As a result of the activation, a node program is executed, which may contain, among the usual programming facilities, also token send statements. Such statement specifies a *node name*, a *port name*, an *index value*, and a *data item* of the required type as a “payload”. The same information is placed into the generated token. The node name and index together form a *key* that plays the role of the virtual node address.

A virtual node is created “really” at the moment the first token comes to one of its ports. The key is placed into the key association memory (KAM) with reference to the descriptor (see Fig. 1), which is created in the descriptor memory (DM). For each port there is a room for a list of incoming tokens in the descriptor. Normally it is either empty or contains a single element.

When a next token arrives, the descriptor is found or created in the matching memory (MM), the token is inserted into proper list, after which the activation decision is made. If there is at least one token on each port, then a token is withdrawn from each port list and their data items are passed as actual parameters to the node program call. The node program will be separately executed later without additional delays and waitings on a executing unit (EU). The node index is also passed as a parameter.

No other “global” or “side” data are available to the node program (except for global constants). The node program cannot produce any side effect other than the non-blocking token sends. New tokens are sent to the token

switch (TS), that serves as the first level of search in the global KAM (the token switch unites many such cores in the PDCS). The result tokens are sent via TS to the host that has launched initial tokens for the current task.

Fig. 1. Logical structure of PDCS core (within dotted rectangle)

The activation decision algorithm for plain nodes (without branches) is simple: there is a bit scale in the descriptor, where the ports with non-empty lists are marked. When a token arrives, the corresponding bit is set. When all bits are set, the node is activated. It is important to note that AKM does not “know” how many ports are at which nodes – therefore, their total number must be transmitted in each token. After node is activated, the bit is cleared for those ports where the list of tokens gets empty.

II. NODES WITH BRANCHES

Unlike plain nodes, nodes with branches have several definitions of node program, called *branches*, each with its own subset of inputs used for the branch activation. Upon receipt of next token, a readiness condition may arise for several branches, at least one of which will fire, usually the one written earlier. Several consecutive branches can be combined into a priority group, of which any one will fire. Hence, the non-determinism of execution may occur. The choice of a branch for activation also depends on the order in which tokens arrive at different inputs of the same node. Thus, generally, the result may not be determinate.

A similar facility can be found in Join Calculus [4] within a functional programming language (JoCaml). Another analogue is Petri nets [5], in which positions represent ports and transitions represent branches.

Fig. 2 shows an example of a simple node with two ports and two branches of different priority. The first branch has higher priority (1). When there are tokens on both ports it fires and executes statement S1. The second branch (statement S2) fires when there is only one token on port a. It has a lower priority (2), which implies that it won't be activated as long as token is present on port b.

As a whole, the behavior of node **C** will be as follows. If token **b** comes first, it will simply be placed in the new descriptor. And when the token **a** comes later, it “sees” the token **b** and starts the program S1. But if the token **a** comes first, it will not “see” the token **b**, and thus the program S2 will be launched. This behavior almost exactly mimics the

execution of a so-called **check-node**, which was available in the DFL language of the PDCS without branches. Token **a** was implemented as “PP-request”, whose arrival caused activation regardless of the presence of token **b**. And within a single node program one was able to detect whether the token **b** was used by means of the predicate **given(b)**.

```
node C(a,b);
  (prio=1)
  case (a,b): S1; end
  (prio=2)
  case (a): S2; end
end;
```

Fig. 2. The model for check-node

With the help of branches, one can also model some other special DFL tokens, e.g. “indirection” and “erasure”. With the advent of the branches, there is no more need to support them.

The mechanism of branches creates much wider opportunities that were previously impossible. As an illustration, consider the classical problem of “five dining philosophers,” proposed by E.W. Dijkstra [6]. Five philosophers (A, B, C, D, E) sit around the table, 5 forks lie between them (f1, f2, f3, f4, f5). Each philosopher at each moment of time either “thinks” or “eats”. He can start “eating” only when there are two forks on both the left and right sides of him – then he takes them and uses simultaneously. “After eating,” he puts both forks back (in any order) and starts to “think.” When he gets hungry, he may attempt to start eating again. Initially, all forks “lie”, and all philosophers “think.” It is clear that no two neighboring philosophers can “eat” simultaneously. It is required to write in any form an algorithm that implements (non-deterministic) any feasible order of philosophers’ actions (the objection of preventing an endless “starvation” is not imposed now).

The solution is a program (see Fig. 3), in which the main node **T** (the table) has 10 ports and 5 branches. The ports are the philosophers being “hungry” and forks “lying” on the table. 5 other plain single-port nodes fire when “the philosopher has finished eating and begins to think.”

The token send statement has the form $x \rightarrow N.p(i)$. It sends value **x** to port **p** of node **N** with index **i**. The port name may be omitted if it is a single port node. The node name or the index is omitted if it is the same as the current one. When a common value is sent to several places, the latter are listed comma-separated. The value is omitted when it is empty (of type **void**). The syntax is free.

Whereas the whole process proceeds in time, the activation of the a node occurs instantly. Then, in some time, the node program is executed and issues a certain number of new tokens. No more interactions with environment are there within this execution. Token transfer takes indefinite, but finite time. It is the transfer of token

Xe or **T.hX** that represents the philosopher **X** as “eating” or “thinking”.

```

node T(f1,hA,f2,hB,f3,hC,f4,hD,f5,hE);
  case (f1,hA,f2): → Ae; end
  case (f2,hB,f3): → Be; end
  case (f3,hC,f4): → Ce; end
  case (f4,hD,f5): → De; end
  case (f5,hE,f1): → Ee; end
end;
node Ae; → T.f1, T.f2, T.hA; end
node Be; → T.f2, T.f3, T.hB; end
node Ce; → T.f3, T.f4, T.hC; end
node De; → T.f4, T.f5, T.hD; end
node Ee; → T.f5, T.f1, T.hE; end

```

Fig. 3. The program for five philosophers problem

The process is initiated by sending 10 tokens – one for each T’s port. Further, the tokens waiting at these ports symbolize “forks on the table” and “hungry philosophers”.

This solution looks like a specification. The work in accordance with its requirements is ensured by the mechanism of the branches. All 5 branches are considered equal priority. Therefore, when there is an opportunity to activate several branches, anyone can be activated. When a branch is activated, all the participating tokens are simultaneously (atomically!) removed from their port lists, therefore two branches with a common port on which one token “sits” cannot be activated together. The activation of the branch for the philosopher **X** implies that both “forks” to the right and to the left of it are instantly removed together with the token **T.hX**, which prevents the activation of neighboring philosophers. The activated branch sends a token to the **Xe** node. This means that the philosopher **X** starts eating. When the **Xe** token reaches its target port, the corresponding single-port node will be unconditionally activated with the result that the same forks are sent back along with **T.hX**. Now **X** is “thinking.”

The structure of our solution in the UPL is similar to that in JoCAML [7]. In particular both are not distributed. This is due to the entire selection being made within a single node T.

III. IMPLEMENTATIONS

It is assumed that upon receipt of a token, the KAM must search through the headers of all branches and activate the first (or anyone) of those for which all necessary tokens are present. However, given that KAM “does not know” the program code, the branch headers are also unavailable. Thus all this information should be transmitted with tokens. (Of course, it must be the responsibility of the compiler rather than the programmer.) However, to transmit information about all ports of all branches in each token is too expensive (for B branches and P ports, at least $B \cdot P$ additional bits are needed). Therefore, it is desirable to include in each token only a certain part of this information, so that in total, upon receipt of only a part of the tokens, there would be enough

information to make the decision on the activation of a ready branch.

Fixing the representation of a token implies some restrictions on its components’ value ranges. We assume that the number of ports P (of any node) does not exceed 16 (in token bit format, the port number will take 4 bits), with the same value we limit the number of branches B . Perhaps, additional limitation will be imposed on the maximum number of ports involved in a branch and the maximum number of branches that use given port.

Since each port can participate in several branches, say b , the token sent to it must contain a number of additional bits for each of b branches ($b \leq b_m \leq B$). We assume that first comes the bitmask of length b_m with b ones, describing the subset of branches that use the token’s target port (these branches will be called *relevant*). It is clear that only these bits are not enough to make an activation decision, hence we need to add some k bits to the token for each relevant branch — totally kb bits. We call this value the *fullness code*. Note that now there is no need to include the total number of ports into the token.

Now, we describe three schemes for filling each k -bit element of the fullness code. Let’s start with the simplest.

A. Scheme “ $b \cdot \log$ ”

The first and simplest solution is to record the number of ports of a given branch in the element (may be reduced by 1). If $p_m = 16$ this requires $k = 4$ bits for each relevant branch, $k = 3$ bits are required if $p_m = 8$, and, in general, there are $b \lceil \log(p_m) \rceil$ bits for all branches, which is also reflected in the name of the scheme.

Consider the algorithm of the activation block for this scheme. As usual, when the first token arrives, a descriptor is created. In addition, an array of B cells **NTOK** should be created, in which for each branch the number of its ports that have received at least one token will be accumulated. When the token arrives (and if it is first for its port), we add 1 to the **NTOK** cells for all relevant branches in which the token participates. Along the way, check whether the result matches the corresponding element of the fullness code. If so, the branch is ready and can be activated. If several branches are ready, one of them will be selected (taking into account priorities, if they have been set). When a branch fires, one token is removed from each used port, and if the port becomes empty, then the corresponding elements of the **NTOK** array are reduced by 1.

B. Scheme “ $b + \log$ ”

The previous scheme can be improved by reducing the bit consumption if the number of ports of this branch is not transmitted in all participating tokens, but only directed to one selected port. In each branch, choose one port as the main one. The first bit of the fullness code element (generated by the compiler) tells whether the target port is main for the given branch, and, if so, the number of

branch' ports p is in the other $\lceil \log(p_m) \rceil$ bits. Thus, the total size of the fullness code will be $b + \lceil \log(p_m) \rceil$.

In the descriptor, we must have another array **NREQ** of the same size B — the number of tokens required for each branch, initially zero. If the new token is the main for the branch j , the corresponding value from the fullness code is entered into the j -th element of the **NREQ**. Now, after adding 1 to the **NTOK** element, the result is compared with the corresponding **NREQ** element. If it matches, this branch can be activated.

C. Scheme "2·b"

In the first scheme, information is distributed equally among all tokens of the branch, but duplicated, and therefore the costs (in bits) are large. In the second, there is no duplication and, therefore, on average, costs are lower, but they are not uniform. The question is whether it is possible to combine the merits of both solutions, so that the costs are less and uniform. It turns out you can.

In the 3d scheme with symbolic name "2·b", the fullness code contains exactly 2 bits for each branch. All information about the number of branch' ports is evenly distributed among the tokens going to these ports.

So, $k = 2$ and the contents of these k bits will be called "color". Now the task is, using K colors (4 in this case) and distributing them between branch tokens, to provide the KAM with the opportunity to decide whether all of the branch' tokens are present, or some are missed (and so for each branch). This decision can rely only on the number of received tokens of each color. Thus, for each number of inputs p from 1 to some maximum $N = p_m$, we need to select the number of tokens of each color in the full set of tokens. In other words, it is necessary for each $p \in \{1, \dots, N\}$ to choose a partition of the number p into a sum of K non-negative terms. Such a partition can be specified in the form of a vector $\mathbf{v} = \mathbf{v}(p)$ of length K with nonnegative integer elements, giving a total of p . The compiler will assign a color to each token so that the number of tokens of each color in a full set of p tokens be equal to the corresponding element of the vector $\mathbf{v}(p)$. This assignment must allow KAM to always take correct decision on the fullness of the set of tokens received so far.

Definition 1. A method to assign for each $p \in \{1, \dots, N\}$ the number of tokens of each color is called *suitable*, if it always allows, having received any part (or all) of the p necessary tokens, to determine whether all the p tokens were received (without knowing this p).

A method to choose the number of tokens of each color for each $p \in \{1, \dots, N\}$ can be specified as a table of size $K \times N$ (see Table 1 as an example).

Definition 2. We call a table *well-formed* if (a) the sum of the components of each column vector $\mathbf{v}_i$ is i and

(b) neither two different columns $\mathbf{v}_i$ and $\mathbf{v}_j$ satisfy the relation $\mathbf{v}_m \mathbf{v}_j$ (which means $\mathbf{v}_i[k] \leq \mathbf{v}_j[k]$ for each k).

The following theorem makes it possible to reduce the problem of choosing a suitable method to the task of building a well-formed table of a given size.

Theorem 1. A method is suitable if and only if it is specified by a well-formed table.

Proof. Condition (a) is evident. Consider (b). We'll prove contrapositive assertion for each part.

Part $\Rightarrow$. Suppose the table (that specifies the given method) is not well-formed. Thus, there are two columns $\mathbf{v}_i$ и $\mathbf{v}_j$ in it, $i < j$, such that $\mathbf{v}_m \mathbf{v}_j$. Consider the situation in which exactly $\mathbf{v}_i[k]$ tokens of each color k has been received. It can occur when either $p = i$ in which case the full token set has been received, or when $p = j$ in which case only proper subset is present. Thus, in this situation, it is not possible to decide if the set of present tokens is full, hence the method is not suitable.

Part $\Leftarrow$. Suppose that given method is not suitable. It means that a situation is possible with numbers of tokens of each color represented by vector $\mathbf{v}$, such that, on the one hand, this set is full for $p=i$ and $\mathbf{v}=\mathbf{v}_i$, and, on the other hand, this set is partial for $p=j>i$ and $\mathbf{v}<\mathbf{v}_j$. Therefore $\mathbf{v}_i < \mathbf{v}_j$, that is the table is not well formed. $\square$

When $K = 4$, the assignment can be made according to a well-formed Table 1, containing $N = 12$ vectors. For larger N , 5 or more colors are required.

Table 1

Numbers of tokens of each color for $p \leq 12$ ports

p :	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	2	4	0	2	4	6	8	10	12
1	0	0	2	1	0	6	5	4	3	2	1	0
2	0	2	1	1	1	0	0	0	0	0	0	0
3	1	0	0	0	0	0	0	0	0	0	0	0

The table is built according to the following rule. The top row lists the column vectors numbers. The sum of the elements of column i is equal to i . The first column has 1 in the leading position and 0 in the others. Each next column vector is chosen so that it is lexicographically¹ smaller than the previous one, but is the lexicographically largest of all such. The first condition guarantees the absence of comparable pairs, that is, well-formedness. The second maximizes the number of vectors. It can be proved

¹ To compare vectors lexicographically, we compare first the elements in the leading position (lowest in the table), if they are equal, then in the next position, etc.

that with $K = 4$ colors, a maximum of $N = 12$ is reached for the number of columns. Yet with $K = 5$ colors you can reach as much as $N = 65532$. For an arbitrary K the well-formed table of size $K \times N$ can be constructed for each $N < A(K, 0)$, where $A(m, n)$ is the well-known Ackermann function [8], defined by the following recursive relations:

$$\begin{aligned} A(0, n) &= n + 1, \\ A(m, 0) &= A(m - 1, 1), \\ A(m, n) &= A(m - 1, A(m, n - 1)). \end{aligned}$$

Now consider how the fullness is determined in KAM. In each descriptor it is necessary to have a two-dimensional array of $K \cdot B$ cells in which the numbers of received tokens of each color are maintained for each branch (as before, only one token is taken into account for each input if there are several of them; accordingly, the sum is reduced only for the last token removed from the port). Having the numbers of available tokens of each color (v_3, v_2, v_1, v_0) known, one can write the criterion of fullness as the following equation (for $K = 4$):

$$(2^-)^{v_3} ((2^{v_2} (2v_1 + v_0 + 4))) = 16, \quad (1)$$

where 2^- is the function symbol for “raising two to the power of the argument” ($2^-(n) = 2^n$), and the power over the function symbol is understood as the composition of the specified number of function instances ($f^0 = I$ is the identity function, $f^1 = f$, $f^2 = f \circ f$, ...). And generally (for arbitrary K) the fullness criterion is expressed by the formula:

$$f_{K-1}^{v_{K-1}} (\dots f_1^{v_1} (f_0^{v_0} (1)) \dots) = f_K(0),$$

where

$$f_m(n) = A(m, n).$$

Some initial values of the function $A(m, n)$ are shown in Table 2.

Table 2

Initial values of functions f_m

	0	1	n
f_0	1	2	$n+1$
f_1	2	3	$n+2$
f_2	3	5	$2n+3$
f_3	5	13	$2^{n+3}-3$
f_4	13	65533	$\underbrace{2^{2^{n+3}}}_{n+3} - 3$
f_5	65533	$2^{65536}-3$	...

When checking equality (1), its left side can be calculated step by step, checking the following conditions as we proceed:

$$\begin{aligned} \text{step 1: } v_0 + 4 &= 2i \quad (\text{even}) \\ \text{step 2: } 2v_1 + 2i &= 2^j \quad (\text{power of 2}) \end{aligned}$$

Next, pass to logarithm:

$$\begin{aligned} \text{step 3: } v_2 + j &= l \quad (2 \text{ or } 4) \\ \text{step 4: } (2^-)^{v_3} (l) &= 4 \end{aligned}$$

After the first violation we can stop. In the last equality, since $v_3 \leq 1$, the expression in the left side can be rewritten as $2^2 (v_3 = 0 ? l : 2^l)$.

Returning to our philosophers, we see that each branch has three inputs out of ten. The constraint $p_n = 12$ is satisfied. For $p = 3$, the assignment is given by the vector $(0, 1, 2, 0)$ with the least color on the right. Therefore, for each branch, it is necessary to assign color 1 for some two tokens and color 2 to a third one. The compiler will do this. Here it is convenient to mark philosopher tokens with color 2, and fork tokens with color 1 (but every which way is also possible, moreover, a token can have different colors in different branches).

Consider the operation of the activation block by example. Suppose one token of color 1 and one of color 2 come to some branch. Having a vector $v = (0, 1, 1, 0)$ we perform checks.

Step 1: $0 + 4 = 4$ is even, ok.

Step 2: $2 \cdot 1 + 4 = 6$ is not a degree of 2, so the set is not complete.

Now let the token of color 1 come, which yields the vector $(0, 1, 2, 0)$. Then step 1 is the same.

Step 2: $2 \cdot 2 + 4 = 8 = 2^3$, ok, $j = 3$.

Step 3: $1 + 3 = 4$, ok.

Step 4: $(2^-)^0 (4) = I(4) = 4$, ok. The set is full, and the branch can be activated.

IV. CONCLUSION

To enable the determination of branch' readiness for activation, three solution schemes have been proposed. The most effective is the third method, in which two bits (four colors) are sufficient for each (relevant) branch, provided that each branch involves maximum 12 ports. For five colors, at a cost of $\log_2(5) \approx 2^{1/3}$ bits for each branch (or a byte for each 3 branches), we can have almost unlimited number of ports in a branch – up to 65532.

Consider a more general problem: for a given set of colors $\{1, \dots, K\}$, to maximize (for a given n) the range $[n..N]$ of feasible numbers of ports in each branch. Our 3rd

² Conditional expression in C syntax.

method gives the best solution $N = f_K(n-1)-1$, in which $f_K(n) = A(K, n)$ is Ackerman's function of level K , and there is reason to believe that it cannot be improved. On the other hand, if the maximum number of inputs is given, then the inverse Ackerman function, properly defined, will show a minimum sufficient number of colors.

Perhaps the most interesting and significant contribution of the paper is comprised of the very formulation of a new practical problem, the solution of which is described exactly by the Ackermann function.

REFERENCES

- [1] Klimov A.V., Okunev A.S. Graficheskij potokovyj metazyk dlya asinkhronnogo raspredelenogo programmirovaniya (A graphical dataflow meta-language for asynchronous distributed programming) // Problemy razrabotki perspektivnykh mikro- i nanoelektronnykh sistem — 2016 (MES-2016). Sb. trudov. M.: IPPM RAN, 2016. Ch. II. S. 151–158.
- [2] Stempkovskij A.L., Levchenko N.N., Okunev A.S., Cvetkov V.V. Parallelnaya potokovaya vychislitel'naya sistema — dal'nejshee razvitie arkhitektury i strukturnoj organizacii vychislitel'noj sistemy s avtomaticheskim raspredeleniem resursov (Parallel dataflow computing system — further development of the architecture and structural organization of the computing system with automatic allocation of resources). Zhurnal «INFORMACIONNYYE TEHNOLOGII». 2008, № 10, pp. 2–7 (In Russian).
- [3] Klimov A.V. O paradigme universal'nogo yazyka parallel'nogo programmirovaniya (On the paradigm of the universal language for parallel programming) // Yazyki programmirovaniya i kompilyatory — 2017. Proc. of the conference. Rostov-na-Donu: YuFU, 2017. S. 141–146.
- [4] Fournet C., Gonthier G. The Join Calculus: A Language for Distributed Mobile Programming // In: Barthe G., Dybjer P., Pinto L., Saraiva J. (eds) Applied Semantics. APPSEM 2000. Lecture Notes in Computer Science. Vol 2395. Springer, Berlin, Heidelberg, 2000. P. 268–332.
- [5] Kotov V.E. Seti Petri (Petri nets). — M: Nauka, 1984. — 160 p.
- [6] Dijkstra, E.W. Hierarchical Ordering of Sequential Processes // Acta Informatica. 1971. V. 1(2). P. 115–138.
- [7] URL: https://rosettacode.org/wiki/Dining_philosophers#JoCaml (access date: 11.04.2018).
- [8] Sundblad, Y. The Ackermann function. A theoretical, computational, and formula manipulative study // BIT Numerical Mathematics. 1971. V. 11. P. 107–119.