Механизм ветвей в потоковом метаязыке UPL (METAL) и методы его реализации в ППВС «Буран»

А.В. Климов, Н.Н. Левченко

Институт проблем проектирования в микроэлектронике (ИППМ) РАН, г. Москва,
klimov@ippm.ru, nick@ippm.ru

Аннотация — В статье рассматривается механизм ветвей в потоковом языке параллельного программирования UPL — Universal Parallel Language (другое название — METAL) и методы его реализации в архитектуре ППВС «Буран». В языке UPL узел с P входами может иметь несколько ветвей, каждая из которых активируется своим подмножеством входных токенов. Использование ветвей демонстрируется на примере задачи о пяти «обедающих философах». Приводятся необходимые сведения о семантике языка UPL.

Задача реализация механизма ветвей требует, чтобы в токенах содержалась дополнительная информация (помимо номера входа и его нужности для каждой ветви) для решения вопроса о полноте набора нужных входов для каждой ветви. Рассматриваются три способа внесения этой информации в токены: два простых и третий оптимальный. Последний требует только k бит в каждом токене на каждую ветвь при максимальном числе входов (в каждой ветви) $N < A(2k, 0)$, где $A(m, n)$ — очень быстро растущая функция Аккермана, тогда как первые годятся только при $N \leq 2k$. Для каждого способа описан алгоритм работы системы по определению достаточности набора принятых токенов для активации той или иной ветви. Важным результатом статьи является сама постановка задачи, в построении решения которой участвует функция Аккермана.

Ключевые слова — потоковая модель вычислений, потоковый язык, узлы с ветвями, активация по подмножеству входов, «обедающие философы», Join calculus, структура токена, функция Аккермана.

I. ВВЕДЕНИЕ

Одной из особенностей проектируемого потокового метаязыка параллельного программирования UPL [1] (Universal Parallel Language) являются *ветви*. Этот механизм позволяет узлам начинать выполнение программы узла не только по готовности данных на всех входах узла, но и при наличии входных данных на определенной части входов. Таких частей может быть определено несколько, и для каждой из них задается своя ветвь программы узла. В статье рассматриваются как сам механизм ветвей в языке UPL, так и методы его реализации в проектируемой параллельной потоковой вычислительной системе (ППВС) «Буран» [2]. Использование ветвей позволяет вводить в алгоритмы гибкий управляемый недетерминизм.

Язык UPL имеет текстовую и графическую формы. В данной статье используется текстовая. Графическая форма была представлена в работах [1],[3]. Программа строится как набор узлов, вместе составляющих граф алгоритма. Описание узла содержит имя узла, список именованных входов (портов) с типами данных на них, список полей контекста (индекса) и программу узла. Обычно индекс — это набор целочисленных величин.

При исполнении порождаются, отрабатывают и исчезают экземпляры узлов, также называемые виртуальными узлами, каждый со своим значением индекса. Они так называются, поскольку заранее их множество не известно, каких-либо ресурсов за ними не закрепляется, то есть они существуют как бы «виртуально».

Виртуальные узлы при активации производят некоторые вычисления и посылают результаты в виде *токенов* на входы друг друга. Программа узла активируется, когда на все входы данного виртуального узла придут токены со значениями входных данных. В результате активации выполняется программа узла, в которой, наряду с обычными средствами программирования, могут содержаться операторы отправки токенов на другие виртуальные узлы. В таком операторе указываются *имя узла*, *имя входа*, значение *индекса* и элемент данных требуемого типа в качестве «полезной нагрузки». Та же информация содержится в порожденном токене. Имя узла и индекс вместе образуют *ключ*, который играет роль адреса виртуального узла.

Виртуальный узел создается «реально» в тот момент, когда на один из его входов приходит первый токен. В ассоциативную память ключей (АПК) заносится ключ со ссылкой на дескриптор (см. рис. 1), который создается в памяти дескрипторов узлов (ПДУ). Для каждого входа узла в дескрипторе имеется место для списка поступивших токенов. Обычно он либо пуст, либо содержит один элемент.

Когда приходит очередной токен в сопоставляющей памяти (СП) отыскивается или создается дескриптор, в него заносится токен, после чего принимается решение об активации. Если на всех входах имеются токены, то из каждого входа изымается по одному токenu, данные которых подаются в качестве фактических параметров в вызов программы узла. Активированная программа узла

позднее будет автономно выполнена без дополнительных задержек и ожиданий на отдельном исполняющем устройстве (ИУ). Индекс также передается ей в качестве параметров.

Рис. 1. Логическая структура ядра ППВС (обведено пунктиром)

Никакие иные «глобальные» или «побочные» данные программе узла недоступны (не считая глобальных констант). Программа узла также не может произвести никакого побочного эффекта, отличного от неблокирующих посылок новых токенов. Новые токены направляются в коммутатор токенов, который служит как бы первым уровнем поиска в объединенной АПК (в ППВС коммутатор токенов объединяет много таких ядер). Некоторые токены-результаты могут быть направлены на хост-машину, запустившую начальные токены для данной задачи.

Схема реализации механизма принятия решения об активации для обычных узлов (без ветвей) проста: в дескрипторе поддерживается битовая шкала, где помечены входы с токенами. Когда приходит очередной токен, в ней взводится соответствующий бит. Важно учесть, что СП «не знает» сколько входов у каких узлов — поэтому их общее количество передается в каждом токене. При активации шкала обнуляется на тех входах, где список токенов стал пустым.

II. Узлы с ветвями

В отличие от обычных, узлы с ветвями имеют несколько определений программы узла, где в каждом определении задается свой поднабор входов, используемый при активации данной ветви. При поступлении очередного токена может возникнуть условие готовности для нескольких ветвей, хотя бы одна из которых будет активирована. Обычно это первая в порядке написания. Несколько подряд идущих ветвей могут быть объединены в группу с одним приоритетом, из которых активируется любая. Использование ветвей создает недетерминизм выполнения, поскольку выбор ветви для активации зависит от порядка прихода токенов на разные входы одного узла. В общем случае это может приводить к недетерминированному результату.

Данная конструкция имеет аналог в Join Calculus [4], где похожее средство вводится в контексте

функционального языка программирования (JoCaml). Другой аналог — сети Петри [5], где входам соответствуют позиции, а ветвям — переходы.

На рис. 2 показан пример простейшего узла с двумя входами и двумя ветвями разного приоритета. Первая ветвь имеет более высокий приоритет. Она активируется при наличии токенов на обоих входах. При этом выполняется оператор **S1**. Вторая ветвь (оператор **S2**) активируется при наличии токена на одном входе **a**. Она имеет более низкий приоритет (2), а значит при наличии токена **b** уже не может быть активирована.

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

Рис. 2. Узел типа check-node

В целом поведение узла **C** будет следующим. Если первым придет токен **b**, то он будет просто помещен в новый дескриптор. И когда затем придет токен **a**, он «увидит» токен **b** и запустит программу **S1**. А если первым придет токен **a**, то он «не увидит» токена **b**, и потому будет запущена программа **S2**. Такое поведение почти точно соответствует выполнению узла типа **check-node**, который был возможен в языке DFL системы ППВС до введения ветвей. В его реализации токен **a** помечался как «ПП-запрос», который порождает активацию независимо от наличия токена на другом входе **b**. При этом внутри единой программы узла можно было узнать, был ли токен **b**, при помощи предиката **given(b)**.

При помощи ветвей можно также смоделировать и некоторые другие специальные токены языка DFL, например «косвенность» и «стирание». С появлением ветвей необходимость в их поддержке отпадает.

Вместе с тем механизм ветвей создает гораздо более широкие возможности, которые ранее были невозможны. В качестве иллюстрации рассмотрим классическую задачу о «пяти обедающих философах», предложенную Э. Дийкстрой [6]. За круглым столом сидят 5 философов (A, B, C, D, E), в промежутках между ними лежат 5 вилок (f1, f2, f3, f4, f5). Каждый философ в каждый момент времени либо «думает», либо «ест». Он может начать «есть», только если две вилки имеются справа и слева от него — тогда он их одновременно забирает и использует. «Наевшись», он кладет обе вилки обратно (можно в любом порядке) и начинает «думать». «Проголодавшись», он может попытаться опять начать «есть». Изначально все вилки «лежат», и все философы «думают». Понятно, что никакие два соседних философа не могут «есть» одновременно. Требуется написать в какой-либо форме алгоритм, реализующий (недетерминировано) любой допустимый порядок действий философов (цель

недопущения бесконечного «голодания» сейчас не ставится).

Решением является программа (см. рис. 3), в которой основной узел **T** — «стол» — имеет 10 входов и 5 ветвей. Входами являются сами философы в состоянии «голоден» и вилки в состоянии «лежит». Еще 5 простых одновходовых узлов активируются, когда «философ поел и начал думать».

Оператор отправки значения **x** на вход **p** узла **n** с индексом **i** имеет вид: **x** → **n.p(i)**. Вход не пишем, если узел одновходовый. Имя узла (как и индекс) опускаем, если оно (он) совпадает с текущим. Когда общее значение посылается в несколько мест, перечисляем последние через запятую. Значение опускаем, если оно пустое (типа **void**). Синтаксис условный.

Процесс протекает во времени, но активация узла происходит одновременно. Затем через какое-то время выполняется программа узла, которая выдает некоторое число новых токенов. Больше никак выполнение программы со средой не взаимодействует. Передача любого токена занимает неопределенное, но конечное время. Именно передача токена **Xe** или **T.hX** «изображает» состояние философа **X**, соответственно, «ест» или «думает».

```
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.hE, T.f1; end
```

Рис. 3. Программа задачи о пяти философсах

Начальная инициация производится посылкой 10 токенов — по одному на каждый вход узла **T**. Далее присутствующие на входах узла **T** токены изображают «лежащие на столе вилки» и «голодных философов».

Данное «решение» есть фактически спецификация. Ее работа в соответствии с требованиями задачи обеспечивается самим механизмом ветвей. Все 5 ветвей считаются равно-приоритетными. Поэтому, когда есть возможность активировать несколько ветвей, будет активирована любая. При активации ветви все участвующие в ней токены одновременно (атомарно!) снимаются со своих входов, поэтому две ветви, имеющие общий вход, на котором «сидит» один токен, не могут активироваться вместе. Активации ветви для философа **X** приводит к тому, что обе «вилки» справа и слева от него одновременно снимаются вместе с токеном **T.hX**, чем исключается активация соседних философов. Активированная ветвь

посылает токен на узел **Xe**. Это означает, что философ **X** начинает «есть». Когда токен **Xe** дойдет до своего входа, соответствующий одновходовый узел будет безусловно активирован, в результате те же «вилки» посылаются обратно вместе с **T.hX**. Теперь **X** «думает».

Структура нашего решения и его записи на UPL почти повторяют запись аналогичного решения на JoCAML [7]. При этом оба решения не вполне полноценные, поскольку не являются распределенными. В нашем подходе это связано с тем, что весь выбор производится внутри одного узла **T**.

III. СХЕМЫ РЕАЛИЗАЦИИ

Предполагается, что при поступлении очередного токена СП должна «просмотреть» набор ветвей и их входных шаблонов и активировать первую (или любую) из ветвей, для которых все необходимые токены присутствуют. Однако, учитывая, что СП «не знает» кода программы, ей недоступны и шаблоны для ветвей. Возникает необходимость передачи всей этой информации через токены. (Разумеется, об этом должен будет заботиться не программист, а компилятор.) Передавать в каждом токене информацию о всех входах всех ветвей слишком накладно (при **B** ветвях и **P** входах надо еще как минимум $B \cdot P$ дополнительных бит). Поэтому желательно вложить в каждый токен лишь некую часть этой информации, чтобы в совокупности по получении только необходимой части токенов было достаточно информации для принятия решения об активации готовой ветви.

Фиксация представления токена создает ограничения на диапазоны значений его компонентов. Будем считать, что число входов **P** не превосходит 16 (в формате токена номер входа занимает 4 бита), таким же значением ограничим и число ветвей **B**. Возможно, будет дополнительно ограничено наибольшее число входов для каждой ветви p_m и наибольшее число ветвей b_m , в которых используется данный вход.

Поскольку каждый вход может участвовать в нескольких ветвях, то направленный на него токен должен содержать некоторое число добавочных бит по каждой из **b** своих ветвей ($b \leq b_m \leq B$). Предполагается, что сначала идет шкала из **B** бит, содержащая **b** единиц, об участии токена в каждой ветви узла. Понятно, что только этих бит недостаточно для принятия решения об активации, поэтому приходится добавить в токен на каждую релевантную ветвь по **k** дополнительных бит — всего **kb** бит. Назовем это значение *кодом полноты*. Заметим, что теперь нет необходимости включать в токен общее число входов.

Далее опишем три схемы наполнения **k**-битного элемента кода полноты. Начнем с самой простой.

A. Схема « $b \cdot \log$ »

Первое простое решение состоит в том, чтобы в элементе записать число входов данной ветви (можно

уменьшенное на 1). При $p_m = 16$ на это потребуется 4 бита для каждой из b ветвей, в которых данный токен участвует, при $p_m = 8$ потребуется 3 бита, а в общем случае на все ветви будет $b \lceil \log(p_m) \rceil$ бит, что и отражено в названии схемы.

Рассмотрим алгоритм работы блока активации для этой схемы. Как обычно, когда приходит первый токен, создается дескриптор. Теперь в нем дополнительно должен быть создан массив **НТОК** из B ячеек, в которых будет накапливаться для каждой ветви количество ее входов, принявших хотя бы один токен. Когда приходит токен (и если он первый на своем входе), прибавим 1 к ячейкам **НТОК** для всех ветвей, в которых токен участвует. Попутно проверим, совпадает ли новое значение с соответствующим значением кода полноты. Если да, то ветвь готова и может быть активирована. Если будут готовы несколько ветвей, выбирается одна из них (с учетом приоритетов, если они были заданы). При активации по одному токenu с каждого участвующего в ней входа снимается, и если при этом вход становится пустым, то из соответствующих элементов массива **НТОК** выполняется вычитание 1.

В. Схема « $b + \log$ »

Предыдущую схему можно улучшить, уменьшив расход бит, если передавать число входов данной ветви не во всех токенах, участвующих в ней, а только в одном, выбранном в качестве «главного». В каждой ветви выбираем свой вход в качестве главного. Первый бит элемента кода полноты (его формирует компилятор) токена сообщает, является ли вход главным для данной ветви, и только у главного в остальные $\lceil \log(p_m) \rceil$ бит заносится число входов ветви p . Поэтому общее число бит в коде полноты токена будет $b + \lceil \log(p_m) \rceil$.

В дескрипторе надо иметь еще один массив **NREQ** того же размера B – количество требуемых токенов для каждой ветви, изначально он нулевой. Если новый токен является главным для ветви j , соответствующее значение p из кода полноты заносится j -й элемент **NREQ**. Теперь после прибавления 1 к элементу **НТОК** результат сравнивается с соответствующим элементом **NREQ**, и при совпадении выполняется активация.

С. Схема « $2 \cdot b$ »

В первой схеме информация по всем токенам ветви распределена поровну, но с дублированием, и потому затраты (в битах) велики, во второй дублирования нет и потому в среднем затраты меньше, но они неравномерны. Спрашивается, можно ли объединить достоинства обоих решений, чтобы и затраты были меньше, и равномерность была. Оказывается, можно.

В схеме с условным названием « $2 \cdot b$ » код полноты содержит ровно 2 бита на каждую ветвь. Вся информация о числе входов ветви равномерно распределена среди токенов, идущих на эти входы.

Итак, $k = 2$ и содержимое этих k бит будем называть «цветом». Теперь задача в том, чтобы, используя $K = 2^k$ цветов (в данном случае 4) и распределяя их между входными токенами ветви, обеспечить для СП возможность «понять», присутствуют ли все входные токены ветви, или еще каких-то не хватает (и так для каждой ветви). Для каждого числа входов p от 1 до некоторого $N = p_m$, надо выбрать количество токенов каждого цвета в форме целочисленного вектора $\mathbf{v}$ длины K . (Эти количества цветов компилятор будет распределять между токенами, посылаемыми на разные входы ветви.) При этом для разных значений p_1 и p_2 соответствующие вектора $\mathbf{v}_1$ и $\mathbf{v}_2$ должны быть несравнимы в смысле наличия отношения ' $\leq$ ', одновременно между всеми парами компонентов. В противном случае, если $p_1 < p_2$ и $\mathbf{v}_1 \leq \mathbf{v}_2$, то по приходу набора токенов с количествами цветов, задаваемыми вектором $\mathbf{v}_1$, будет неясно, то ли это полный набор для p_1 входов, то ли частичный для p_2 входов. И наоборот, если при каком-то наборе, описываемом вектором $\mathbf{v}_1$ нельзя понять, на всех ли входах имеются токены, то это означает, что такой $\mathbf{v}_1$ является полным набором для $p_1 = \text{sum}(\mathbf{v}_1)$, но при этом также есть некоторое $p_2, p_2 > p_1$, и соответствующий вектор $\mathbf{v}_2$, для которых $\mathbf{v}_1$ является неполным набором, то есть $\mathbf{v}_1 \leq \mathbf{v}_2$.

При $K = 4$ выбор векторов можно делать согласно Таблице 1. В ней всего $N = 12$ векторов. Для больших N потребуется 5 или более цветов.

Таблица 1

Число токенов каждого цвета для $p \leq 12$ входов

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

Таблица построена по следующему правилу. Номера векторов-столбцов заданы в верхней строке. Сумма элементов столбца с номером i равна i . Первый столбец имеет 1 в старшей позиции и 0 в остальных. Каждый следующий вектор-столбец выбирается так, чтобы он был лексикографически¹ меньше

¹ При лексикографическом сравнении сначала сравниваются элементы в старшей (в таблице – нижней) позиции, если они равны, то в следующей и т.д.

предыдущего, но при этом был лексикографически наибольшим из всех таких. Первое условие гарантирует отсутствие сравнимых пар. Второе максимизирует число векторов. Можно доказать, что при $K = 4$ цветах достигнут максимум $N = 12$ для числа столбцов. При $K = 5$ цветах можно достичь $N = 65532$.

Для работы алгоритма определения полноты теперь в дескрипторе надо иметь двумерный массив из $K \cdot V$ ячеек, в которых поддерживается для каждой ветви количество пришедших токенов каждого цвета (как и раньше, для каждого входа учитывается только один токен, если их несколько; аналогично при удалении токена со входа значения сумм уменьшаются только для последнего удаляемого с данного входа токена). Критерий полноты, если известны количества имеющихся токенов каждого цвета (v_3, v_2, v_1, v_0) , можно записать в виде равенства (для $K = 4$):

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

где 2^- это символ функции «возведения двойки в степень аргумента» ($2^-(n) = 2^n$), а степень над символом функции понимается как композиция указанного числа экземпляров функции ($f^0 = I$ – тождественная функция, $f^1 = f$, $f^2 = f \circ f$, ...). А в общем случае (для любых K) критерий полноты выражается так:

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

где функции f_m рекурсивно определены как

$$\begin{aligned} f_0 &= n + 1, \\ f_{m+1}(n) &= f_m^{n+1}(1). \end{aligned}$$

Видно, что $f_m(n)$ есть не что иное, как известная функция Аккермана $A(m, n)$ [8]. Поскольку последний (лексикографически наименьший) вектор имеет вид $(0, 0, 0, N)$, максимальное N при K цветах равно $f_K(0) - 1$.

Таблица 2

Начальные значения функций 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	$2^{2^{n+3}}-3$
f_5	65533	$2^{65536}-3$	...

Для проверки равенства (1) его левую часть можно вычислять по шагам, по ходу проверяя следующие условия:

$$\begin{aligned} \text{шаг 1: } v_0 + 4 &= 2i \quad (\text{четно}) \\ \text{шаг 2: } 2v_1 + 2i &= 2^j \quad (\text{степень } 2) \end{aligned}$$

Далее переходим к логарифму:

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

При обнаружении первого нарушения дальше можно не проверять. В последнем равенстве, поскольку $v_3 \leq 1$, левая часть равносильна² ($v_3 = 0 ? l : 2^l$).

Возвращаясь к задаче о философях, видим, что каждая ветвь имеет по 3 входа из 10. Ограничение $p \leq 12$ удовлетворено. Для $p=3$ назначение задается вектором $(0, 1, 2, 0)$, где младшие цвета справа. Поэтому по каждой ветви надо каким-то двум токенам назначить цвет 1, а третьему – цвет 2. Это сделает компилятор. Здесь удобно токены-философы метить цветом 2, а токены-вилки – цветом 1 (но можно и как попало, и вообще в разных ветвях один и тот же токен может иметь разные цвета).

Рассмотрим работу блока активации. Предположим, на какую-то ветвь пришел один токен цвета 1 и один цвета 2. Имеем вектор $(0, 1, 1, 0)$. Выполняем проверки. Шаг 1: $0+4 = 4$ четно, ок. Шаг 2: $2 \cdot 1 + 4 = 6$ – не степень 2, значит, набор не полон. Пусть теперь приходит токен цвета 1, имеем вектор $(0, 1, 2, 0)$. Тогда: шаг 1 тот же. Шаг 2: $2 \cdot 2 + 4 = 8 = 2^3$, ок, $j = 3$. Шаг 3: $1+3 = 4$, ок. Шаг 4: $(2^-)^0(4) = I(4) = 4$, ок. Значит, набор полон, и можно активировать данную ветвь.

IV. ЗАКЛЮЧЕНИЕ

Для задачи обеспечения определения готовности ветвей к активации предложены три схемы решения. Наиболее интересным является третий способ, в котором достаточно два бита (четыре цветов) на каждую релевантную ветвь, что обеспечивает полноту решения для узлов с не более чем 12 входами в каждой ветви. Для пяти цветов, при затратах по $\log_2(5) \approx 2\frac{1}{2}$ бита на каждую ветвь, можем иметь практически неограниченное число входов — до 65532.

Если ставить более общую задачу: при заданном наборе цветов $\{1, \dots, K\}$ добиться максимизации (при заданном n) диапазона $[n..N]$ допустимых количеств входов в каждой ветви, то третий способ дает наилучшее решение, в котором $N = f_K(n-1) - 1$, где f_K это функция Аккермана K -го уровня, и есть основания полагать, что его нельзя улучшить. С другой стороны, если задано максимальное число входов N , то обратная функция Аккермана,

² Условный оператор в синтаксисе языка С.

надлежащим образом определенная, покажет минимум набора цветов, которых будет достаточно для обеспечения возможности определения полноты набора токенов для каждой ветви.

Важным результатом данной статьи является сама постановка новой интересной практической задачи, в построении решения которой участвует функция Аккермана.

ПОДДЕРЖКА

Работа выполнялась в рамках базового бюджетного финансирования по проекту «Создание метаязыка и средств его трансляции для ППВС «Буран».

ЛИТЕРАТУРА

- [1] Климов А.В., Окунев А.С. Графический потоковый метаязык для асинхронного распределенного программирования // Проблемы разработки перспективных микро- и наноэлектронных систем — 2016 (МЭС–2016). Сб. трудов. М.: ИППМ РАН, 2016. Часть II. С. 151–158.
- [2] Стемковский А.Л., Левченко Н.Н., Окунев С.А., Цветков В.В.. Параллельная потоковая вычислительная

система — дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // Информационные технологии, 2008, №10. С. 2–7.

- [3] Климов А.В. О парадигме универсального языка параллельного программирования // Языки программирования и компиляторы — 2017 (PLC–2017). Труды конференции. Ростов-на-Дону: ЮФУ, 2017. С. 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] Котов В.Е. Сети Петри. — М: Наука, 1984. — 160 с.
- [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 (дата обращения: 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.

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 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) using which the device can decide on the completeness of a set of relevant tokens for each branch. Three ways to encode this information into tokens are considered.

The first way is to simply place into each token for each relevant branch the total number of ports needed for that branch. Thus k bits for each relevant branch are required while maximum number of ports per branch $p_m = 2^k$. The second way is to put this number only into the token directed to the branch's main port while all other ports get just a single bit indicating it is not the main. Finally, the third way, which is

optimal in number of bits, is to allocate for each relevant branch just k extra bits for a number of one of $K \leq 2^k$ colours (typically, $k=2$ suffices). Colours are assigned such that the completeness of a subset of ports could be always established. We propose an optimal way of assigning colours for which we get $p_m < A(k, 0)$, where $A(m, n)$ is a very fast growing Ackerman function. For $k=2$ and $K=4$ we get $p_m = 12$. With $K=5$ we would get $p_m = 65532$.

The important result of the paper is the very formulation of a new interesting practical problem, in which the Ackerman function is involved in the solution.

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

REFERENCES

- [1] Klimov A.V., Okunev A.S. Graficheskij potokovyj metazyk dlya asinkhronnogo raspredelennogo programirovaniya (A graphical dataflow meta-language for asynchronous distributed programming) // Problemy razrabotki perspektivnykh mikro- i nanelektronnykh 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 — dalnejshee razvitie arkhitektury i strukturnoj

- organizacii vychislitelnoj sistemy s avtomaticheskimi raspredeleniem resursov (Parallel dataflow computing system — further development of the architecture and structural organization of the computing system with automatic allocation of resources). Zhurnal «INFORMACIONNYE TEHNOLOGII». 2008, № 10, pp. 2–7 (In Russian).
- [3] Klimov A.V. O paradigme universal'nogo yazyka parallel'nogo programirovaniya (On the paradigm of the universal language for parallel programming) // Yazyki programirovaniya 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.