ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
им.М.В.Келдыша РАН

С.Л. Головков, К.Н. Ефимкин, Э.З. Любимский

О языке программирования для модели вычислений, основанной на принципе потока данных

Москва
2002

С.Л. Головков, К.Н. Ефимкин, Э.З. Любимский

О языке программирования для модели вычислений, основанной на принципе потока данных. ♦

Аннотация

Определяются основные возможности языка программирования высокого уровня, предназначенного для программирования в гибридной модели вычислений data-control flow – языка DCF. Язык DCF проектируется как расширение языка последовательного программирования Си специальными возможностями для представления параллелизма в модели dataflow. Возможности языка Си предлагается использовать для программирования вычислений, составляющих тела нитей, а специальные функции и описания, составляющие его расширение – для описания dataflow-механизмов взаимодействия нитей.

S.L. Golovkov, K.N. Efimkin, E.Z. Ljubimskii

About the programming language for computational model based on dataflow principle.

Abstract

Main possibilities of the high level programming language intended for programming in hybrid data-control flow computational model - DCF language are defined. DCF language is designed as extension of the serial programming language C with special possibilities for presentation of the parallelism in dataflow models. The possibilities of C language is offered to use for programming the calculations, forming bodies of the threads, but special functions and descriptions, forming its extension - for description dataflow-mechanism of a threads interaction.

♦ Работа выполнена при частичной финансовой поддержке Российского Фонда Фундаментальных Исследований (проект N 02-01-01114)

Содержание

1. Модель dataflow – параллельные вычисления, основанные на принципе потока данных

Модель вычислений dataflow, с момента своего появления примерно 30 лет назад, активно исследовалась и применялась как в области архитектур вычислительных систем, так и программного обеспечения. История развития и эволюция этой модели, а точнее, различных dataflow-моделей, основанных на принципе dataflow (принципе потока данных), приведена, например, в [1-3].

Обычно модель вычислений (модель выполнения программы) определяет базовый уровень абстракции программирования, исходя из которого могут быть определены архитектура вычислительной системы, языки программирования, стратегии компиляции, runtime-системы и другие компоненты программного обеспечения. Модель dataflow является параллельной по своей природе - параллелизм заложен в определение dataflow-графа и правил его интерпретации. Надежды на эффективное выполнение программ в этой модели связаны с возможностью решения двух известных проблем последовательной модели вычислений фон Неймана – задержки при обращении к памяти (время от запроса к памяти до получения соответствующего ответа) и синхронизации (необходимость упорядочивать выполнение инструкций в соответствии с зависимостями по данным).

Исследования показали, что поддержка “чистой” модели dataflow в аппаратуре является сложной задачей, и, кроме этого, выполнение некоторых типов вычислений, например, строго последовательного вычисления, в модели dataflow неэффективно. Это привело к появлению [например, 4-5] “гибридных” моделей dataflow-фон Неймана (data-control flow моделей) и интенсивным исследованиям в этой области. В настоящее время эти исследования сконцентрированы, в основном, вокруг многонитевых (multithreaded) моделей [6-8]. *Нить* (*thread*) – это набор инструкций, выполняемых последовательно, как в модели фон Неймана, а сами нити могут выполняться параллельно и запускаются при готовности своих входных аргументов, как в модели dataflow. Существуют различные многонитевые модели, допускающие различные правила выполнения тела нити: с блокировкой выполнения нити на время ожидания выполнения “долгих” операций типа обращения к памяти и без блокировки, а также различные правила переключения нитей.

Исследования многонитевых моделей привели в итоге к разработке архитектур многонитевых суперскалярных процессоров [9-11], которые, помимо достоинств суперскалярной архитектуры, позволяют выполнять несколько нитей одновременно за счет нескольких контекстов выполнения, а также поддерживают спекулятивные вычисления [12-13]. Новые вычислительные модели стимулировали также развитие новых языков для программирования в этих моделях и методов компиляции [14-17].

Таким образом, вычислительная модель dataflow уже оказала положительное влияние на развитие теории и практики программирования, и в перспективе ее использование также может оказаться весьма полезным. При этом

многонитевые архитектуры являются только одним классом dataflow-архитектур, хотя и неплохо продвинутым и проработанным. Весьма интересен и перспективен с этой точки зрения проект dataflow-компьютера [18,19], в котором dataflow-механизм инициализации вычисления реализуется при помощи ассоциативной памяти, что может дать принципиально новое по эффективности решение проблемы синхронизации вычислений. Мы считаем необходимым отметить, что знакомство именно с этим проектом стало отправной точкой нашего исследования вопросов программирования в модели потока данных, первые результаты которых излагаются в данном препринте.

Новые модели вычислений, основанные на принципе dataflow, и языки программирования, основанные на этих моделях, могут оказаться полезными и в ближайшее время для программирования на существующих параллельных вычислительных системах, программирование на которых в настоящее время вызывает значительные трудности.

В данном препринте предпринимается попытка определения основных возможностей языка программирования высокого уровня, предназначенного для программирования в гибридной модели вычислений data-control flow – языка DCF (Data-Control Flow). Язык DCF проектируется достаточно традиционным способом - как расширение языка последовательного программирования Си специальными возможностями для представления параллелизма в модели dataflow. Неформально, возможности языка Си предлагается использовать для программирования вычислений, составляющих тела нитей, а специальные операторы и описания, составляющие его расширение – для описания dataflow-механизмов взаимодействия нитей.

Уровень языка DCF дает возможность использовать его в качестве промежуточного языка представления программ при трансляции традиционных языков программирования высокого уровня, dataflow-языков и декларативных языков программирования высокого уровня для dataflow-компьютеров и многонитевых архитектур, а также для непосредственного программирования. После исследования свойств языка DCF с применением имитационной модели dataflow-компьютера и уточнения его возможностей предполагается использование языка DCF в качестве промежуточного при реализации декларативного языка Норма [20] для dataflow-компьютера.

2. Основные термины и понятия языка DCF

Основными понятиями рассматриваемой модели вычислений являются **нить** и **токен**.

Нить является логической единицей выполнения программы. Она представляет собой цепочку операций, выполняемых в модели вычислений фон Неймана. При этом различные нити могут выполняться параллельно. Запуск нити происходит при появлении готовых значений всех входных параметров нити.

На уровне языка нить описывается функцией, единственным внешним отличием которой от “обычной” функции языка Си является наличие спецификатора **thread** вместо типа возвращаемого значения. Например:

```
thread Func( int N, float Var ) { … }
```

Главное отличие такой функции (будем называть ее **функцией-нитью**) от “обычной” функции состоит в том, что можно **запустить нить**, послав **токены**, содержащие значения всех аргументов, на имя функции-нити.

Запуск нитей, синхронизация нитей и передача значений между ними осуществляется с помощью **токенов** - единиц данных для dataflow управления.

Токен представляет собой объект, содержащий **тег**, идентифицирующий возможного получателя токена, и **данные** некоторых типов языка Си: int, float, char, адрес и др. Тег содержит два компонента: **имя** функции-назначения (функции-нити или request-функции), для которой предназначен токен, и некоторую характеристику, называемую **цветом**.

Токены используются для **активации** нитей – **запуска** или **возобновления выполнения** нитей. Токены порождаются при выполнении специальной стандартной функции.

Если нить передает данные другой нити, то она посылает в “окружающее пространство” токены, рассчитывая, что какая-нибудь другая нить примет их (что, в общем случае, не обязательно должно произойти). Если нить нуждается в какой-то информации, то она ожидает прихода токенов из “окружающего пространства” токенов. **Пространство токенов** играет в этих процессах активную роль, принимая токены от нитей, собирая и группируя токены, а затем используя **готовые группы токенов** для активации нитей. **Готовая группа** содержит токен для каждого из аргументов активируемой функции-назначения, при этом теги всех этих токенов в некотором смысле совпадают.

3. Базовый язык

В этом разделе приведено неформальное описание ключевых возможностей языка DCF: описание расширений и ограничений языка Си, который положен в основу языка DCF.

3.1. Структура программы

Программа на языке DCF представляет собой набор функций, некоторые из которых имеют спецификатор **thread**. Такие функции (функции-нити) используются при порождении (запуске) нитей. Для того, чтобы запустить новую нить, необходимо послать на имя этой функции-нити токены, содержащие значения всех ее аргументов; при этом теги посылаемых токенов должны в некотором смысле совпадать (точное определение совпадения тегов будет описано в разделе 3.5). С помощью любой функции-нити можно породить произвольное количество экземпляров нити.

В программе среди описаний функций-нитей обязательно должно быть описание функции-нити с именем **main**. Запуск программы состоит в активации этой функции-нити. Функция-нить main может иметь аргументы, через которые программа получает параметры из внешней среды выполнения. Повторная активация функции-нити main запрещена.

Таким образом, в любой момент времени выполнения программы происходит параллельное выполнение множества нитей, запущенных с помощью активации функций-нитей.

Программа завершается, когда завершаются все нити.

3.2. Активация нити

Активация (запуск) нити осуществляется посылкой на имя функции-нити токенов, каждый из которых содержит значение некоторого аргумента функции-нити. Посылка токенов для запуска нити осуществляется с помощью стандартной функции **token**:

token(***значение_токена***, ***тег_токена*** [**,** ***число_копий***])

значение_токена имеет вид ***имя_арг : знач_арг***, где
имя_арг - имя или номер аргумента функции-нити,
знач_арг - значение аргумента;
тег_токена имеет вид ***имя_функции-нити***([***цвет***])
число_копий указывает количество экземпляров токена. Если этот параметр опущен, то это означает, что ***число_копий*** равно единице. Если в качестве значения этого параметра указать символ “*”, то порождается “неограниченное” число копий токена.

Примеры: token(Var : 36.6, Func(5));
token(2 : i+j, Func(i), 3);

Первая функция посылает один токен на имя функции-нити Func. Токен имеет цвет, значение которого равно 5. Токен содержит значение 36.6 для аргумента с именем Var.

Вторая функция посылает три токена на имя функции-нити Func. Все они содержат значение i+j для второго аргумента функции-нити Func и имеют цвет, равный значению переменной i.

Заметим, что параметр *имя_арг* однозначно определяет тип значения *знач_арг*, поскольку этот тип определен в заголовке функции-нити.

Если функция-нить имеет один аргумент или не имеет его вовсе, то параметр *имя_арг* можно опустить. Функции-нити с пустым списком аргументов активируются посылкой на имя такой функции токена с пустым значением (NULL).

Цвет представляется целочисленным вектором (i,j,k,…). Цвет можно замаскировать, указав в качестве его значения символ “*”. Можно замаскировать только некоторые элементы вектора цвета, поставив вместо

соответствующего элемента символ “*” (смысл маскирования описан ниже в разделе 3.5). Например, функция

```
token( NULL, thMult( 2, *, i, j, *) )
```

пошлет один токен, активирующий не имеющую входных параметров нить thMult. Токен имеет цвет с двумя замаскированными элементами вектора цвета.

Нить может определить цвет активировавших ее токенов, выполнив стандартную функцию **getThreadColor**. В этом смысле нить “имеет цвет”. Формат обращения к функции:

```
int getThreadColor( int *V )
```

Выполнение функции состоит в том, что она последовательно заполняет элементы вектора V значениями элементов вектора цвета до тех пор, пока либо исчерпается вектор V, либо исчерпается вектор цвета. Возвращает функция значение, равное длине вектора цвета.

Если некоторый компонент вектора цвета замаскирован, то соответствующий элемент вектора V получает значение NULL. Значения элементов вектора V, для которых не нашлось значений элементов вектора цвета (т.е. длина вектора V больше длины вектора цвета) не определены.

Пример. Допустим, есть функция-нить myFunc:

```
thread   myFunc( char a )
{  int  lengthColor, myColor[3];
   lengthColor = getThreadColor( myColor );
}
```

и мы запустили два экземпляра этой нити с помощью токенов

```
token( ‘f’, myFunc(1,*) );
token( ‘s’, myFunc(1,*,3,4,*) );
```

Тогда в первом экземпляре нити переменные lengthColor и myColor приобретут значения:

```
lengthColor = 2;
myColor = { 1, NULL, неопределенное_значение }
```

а во втором:

```
lengthColor = 5;
myColor = { 1, NULL, 3 }
```

Нить может породить другие нити, имеющие такой же цвет, что и она сама. Для этого в функции token параметр *цвет* должен быть опущен, что означает: цвет посылаемого токена равен цвету нити, то есть токенов, запустивших данную нить.

Нить main имеет цвет “NULL”.

В языке имеется стандартная функция **int getNewColor()**, каждый вызов которой генерирует уникальное значение цвета.

С помощью одной функции token можно посылать токены для разных аргументов некоторой функции-нити. Для этого вместо одного *значения_токена* нужно указать несколько элементов *значения_токена*, разделенных запятой.

Например:

token(arg1:value1, arg2:value2, *тег_токена* [, *число_копий*])

Выполнение этой функции идентично выполнению двух функций token:

token(arg1:value1, *тег_токена* [, *число_копий*])

token(arg2:value2, *тег_токена* [, *число_копий*])

3.3. Завершение нити

Завершение нити происходит при выполнении функции **exit_thread**. Если эта функция является последней в теле функции-нити, то она может быть опущена.

Нить не может возвратить значение в смысле языка Си, поэтому функции языка Си return(…) и exit(…) в теле функции-нити интерпретируются как функция exit_thread.

Проиллюстрируем средства активации и завершения нитей на следующем примере.

Пример. Пусть необходимо выполнить вычисление заданной функции Func над каждым из значений a[i], где i=0,1, …, 99; при этом мы хотим, чтобы вычисления Func над разными значениями выполнялись параллельно. Программа на языке DCF может выглядеть так:

```
thread main()
{ int i;
  float a[99]={ … };
  for(i=0; i<100; i++)
        token(a[i], Func() );
}
thread Func(float a)
{
  <вычисления над a >
    exit_thread;
}
```

Сложность программы практически не отличается от последовательного варианта, хотя программа и стала параллельной. В этом примере нить main порождает сто параллельных нитей Func, отправив на имя функции-нити Func сто токенов со значениями a[i].

3.4. Синхронизация и обмен данными между нитями

Часто возникает необходимость в синхронизации выполнения нитей и в обмене данными между нитями. Например, в приведенном выше примере может потребоваться обмен данными между нитями Func и нитью main для того, чтобы последняя могла просуммировать результаты вычислений над значениями a[i]. Для организации такого обмена используются функции со спецификатором **request** вместо типа возвращаемого значения (остановка выполнения нити, ожидание и прием токенов) и уже знакомая функция **token** (посылка токена).

Описание функции со спецификатором **request** (будем называть такие функции “**request-функциями**”) имеет следующий формат:

request ***имя*** (***адрес_переменной,*** ([***цвет***]))

где: ***имя*** имя request-функции;
адрес_переменной адрес переменной, которой необходимо присвоить значение, присланное токеном;
цвет цвет ожидаемого токена.

Request-функции имеют три важных отличия от «обычных» функций языка Си:

- описание request-функции может появиться в любом месте тела любой функции;
- описание request-функции является одновременно и определением функции и ее вызовом;
- имя request-функции должно быть уникальным в пределах всей программы, поэтому полное имя request-функции имеет вид

имя_функции . идентификатор

где ***имя_функции*** является именем функции-нити или обычной функции, в которой определена request-функция;
идентификатор является уникальным в пределах функции ***имя_функции***.

В пределах функции ***имя_функции*** первую компоненту полного имени request-функции можно опустить и указывать только компоненту ***идентификатор***, однако, посылая токены на request-функцию, необходимо указывать ее полное имя.

Выполнение request-функции состоит в следующем. Функция останавливает выполнение нити и сообщает пространству токенов о потребности в токене с указанным цветом.

Как только в пространстве токенов появляется токен, запрашиваемый ждущей request-функцией (или если он уже там есть), то данные, содержащиеся в токене, присваиваются переменной с адресом *адрес_переменной* и выполнение нити возобновляется.

Если в описании request-функции опущен параметр *цвет*, то это означает, что функция ожидает токен, имеющий тот же цвет, что и нить, в которой выполняется эта функция.

Request-функция позволяет вместо *адрес_переменной* указывать список адресов переменных, т.е. последовательность *адрес_переменной* через запятую. В этом случае request-функция ожидает прихода **готовой группы токенов**. Готовая группа содержит токен для каждой переменной request-функции, при этом теги всех этих токенов в некотором смысле совпадают (точное определение совпадения тегов будет описано в разделе 3.5).

Формат обращения к функции **token** при посылке токена(ов) для request-функции имеет практически тот же вид, что и при посылке токенов для запуска нитей (т.е. для функции-нити):

token(***значение_токена***, ***тег_токена*** [,***число_копий***])

значение_токена имеет вид ***номер_арг : знач_арг***, где
номер_арг - номер аргумента request-функции,
знач_арг - значение аргумента;
тег_токена имеет вид ***имя_request-функции***([***цвет***])
число_копий указывает количество экземпляров токена.

Обратим внимание на отличия в синтаксисе функции token, используемой при посылке токенов для request-функции:

- поскольку адресатом токенов является request-функция, то ***тег_токена*** должен включать в себя не имя ***функции-нити,*** а полное имя ***request-функции***, то есть ***тег_токена*** имеет вид

имя_функции . идентификатор ([***цвет***])

Пример. Дополним приведенный выше пример запуска ста нитей Func таким образом, чтобы нить main могла просуммировать результаты работы всех ста экземпляров нити Func. Программа на языке DCF может выглядеть так:

```
thread main()
{ int i;
  float a[99]={ … };
  float rez, Result=0.0;
//  запускаем нити Func
  for(i=0; i<100; i++)
          token(a[i], Func() );
//  ждем результатов работы нитей Func
    for(i=0; i<100; i++)
       { request main.Sum( &rez, () );
         Result =+ rez;
       }
}
thread Func(float a)
{  float var;
   <вычисления над a и присвоение значения переменной var>
   token( var, main.Sum() );
   exit_thread;
}
```

3.5. Маскирование тега токена. Группировка токенов

Тег токена можно расматривать как своего рода “адрес” получателя токена в функции token. В ряде случаев можно добиться заметного упрощения программы и/или увеличения степени ее параллельности, если этот адрес указывать не совсем “точно”, т.е. указывать вместо конкретного отправителя (получателя) множество допустимых отправителей (получателей).

Такая возможность реализуется путем **маскирования** компонент тега. Маскирование осуществляется путем указания специального значения “*” в качестве значения соответствующего компонента тега. При этом цвет можно маскировать как целиком, указав значение “*” на месте параметра *цвет*, так и отдельные элементы вектора, указывающего цвет.

Точная семантика маскирования компонентов тега определяется логикой поведения пространства токенов, которое получает токены, формирует из них готовые группы и использует эти группы для активации нитей.

В данном разделе мы рассмотрим возможность маскирования только одного компонента тега токена, а именно – цвета. Возможность маскирования другого компонента тега – имени функции-назначения токена – рассматривается в разделе 4.2.

Структура пространства токенов.

Пространство токенов состоит из групп токенов. Каждая группа содержит тег, который так же, как и тег токена, состоит из двух компонентов: имени и цвета.

Построение групп токенов.

Если в пространстве токенов появляется новый токен, то делается попытка включить его в одну из существующих групп. Порядок просмотра групп при этом не определен.

Сначала определим правила включения в группу токена, у которого *число_копий* = 1 (который существует в единственном экземпляре).

Если включить токен в какую-либо группу можно, то он включается в эту группу.

Если включить токен в какую-либо группу не удается, то создается новая группа, токен включается в эту группу и тег группы устанавливается равным тегу токена. Таким образом, имя группы устанавливается равным имени функции назначения, а цвет группы – равным цвету токена.

В случае, если для токена было задано *число_копий* > 1, но *число_копий* ≠ “*” указанная процедура производится для каждой копии токена.

Если для *числа_копий* было задано значение “*” (токен с неограниченным количеством копий), то попытка включения проводится для всех уже существующих групп, а также для всех групп, которые будут возникать в дальнейшем.

При включении токена в группу тег этой группы может измениться.

Включение токена в группу.

Токен можно включить в группу в том и только том случае, если токен **подходит группе по имени, по цвету** и **по номеру аргумента-получателя** (то есть аргумента в заголовке функции-нити или переменной в request-функции).

Токен подходит группе по имени, если имя назначения в теге токена совпадает с именем группы.

Если цвет группы равен “*”, то ей подходит (по цвету) токен с любым цветом; при включении токена в такую группу цвет токена становится цветом группы.

Цвет группы (если он не равен “*”) представляет собой в общем случае целочисленный вектор (c1,c2, …, cN), где каждый элемент вектора ci может либо иметь целочисленное значение, либо быть замаскирован (т.е. иметь значение “*”). В такую группу могут быть включены либо токены с замаскированным цветом, либо токены, цвет которых имеет значение (t1,t2,…,tN), где если ci ≠ ti, то либо ci = “*”, либо ti = “*”. Если токен включается в группу, то цвет группы “доопределяется”: замаскированные элементы цвета группы заменяются незамаскированными элементами цвета токена (если ci = “*”, а ti ≠ “*”, то элемент ci заменяется на элемент ti).

Токен **подходит группе по номеру аргумента-получателя**, если в группе не существует токена с таким же номером аргумента (следовательно, в группе не может быть несколько токенов с одинаковым номером аргумента-получателя).

Выбор готовой группы для запуска нити.

Готовой группой для запуска является группа, в которой есть **ровно один токен для каждого аргумента** функции-нити с именем, подходящим имени группы.

После включения токена в группу, пространство токенов определяет, возникла ли готовая группа. Если готовая группа возникла, то аргументам функции-нити присваиваются значения, взятые из соответствующих им токенов группы, и запускается новая нить, имеющая цвет, равный цвету группы.

После этого готовая группа и составлявшие ее токены удаляются.

Выбор готовой группы для возобновления нити.

Готовой группой для возобновления является группа, в которой есть **ровно один токен для каждой переменной,** чей адрес указан в request-функции, с тегом, который подходит тегу, указанному в request-функции.

При выполнении request-функции пространство токенов проверяет, существует ли готовая группа. Если такая группа находится, то переменным, чьи адреса указаны в request-функции, присваиваются значения, взятые из соответствующих им токенов группы, и возобновляется выполнение нити.

После этого готовая группа и составлявшие ее токены удаляются.

Если при выполнении request-функции подходящей группы не находится, то создается новая группа с тегом, равным тегу, указанному в request-функции.

Удаление токенов и групп из пространства токенов.

В языке имеются стандартные функции **kill_group** и **kill_token**, позволяющие удалять определенные токены или группы токенов из пространства токенов.

Формат функций:

kill_group(***тег_токена*** [, ***число_копий***])

kill_token(***тег_токена*** [, ***число_копий***])

где ***тег_токена*** - тег токена, возможно, с замаскированными компонентами тега;

число_копий - число удаляемых токенов или групп.

Выполнение функции kill_group приводит к удалению групп токенов указанного типа, чей тег подходит тегу *тег_токена*. Число удаляемых групп задается параметром *число_копий,* при этом если он равен “*”, удаляются все группы. Данная функция может быть использована, в частности, для чистки пространства токенов.

Выполнение функции kill_token приводит к удалению токенов, чей тег подходит тегу *тег_токена*. Число удаляемых токенов также задается параметром *число_копий,* при этом если он равен “*”, удаляются все токены. В последнем случае данная функция позволяет удалить из пространства токенов те токены, которые имели неограниченное количество копий.

Если группа состоит из одного удаляемого токена, то выполнение функции kill_token приводит также к удалению группы, в которой этот токен находится.

3.6. Видимость переменных в языке

Согласно стандарту Си, переменные могут быть внешними (описанными вне функций) и автоматическими (описанными внутри функций).

Внешние переменные являются глобальными и доступны для всех нитей; спецификатор static ограничивает видимость только теми нитями, описания которых (функции-нити) находятся в данном файле.

Автоматические переменные порождаются при запуске нити и доступны только внутри данной нити. Спецификатор static в отношении автоматических переменных игнорируется.

Таким образом, обмен данными между нитями осуществляется либо при помощи внешних переменных (в этом случае дисциплину доступа к таким переменным из разных нитей определяет и обеспечивает программист), либо при помощи токенов (в этом случае семантику токенов поддерживает система программирования).

3.7. Пример использования базовых средств языка DCF

Допустим, нам нужна функция HowMany, подсчитывающая число вхождений литеры letter в строку. Построим эту функцию следующим образом. Функция запустит нить SplitString, передав ей начало и конец строки.

Нить SplitString анализирует длину полученной строки. Если длина не превышает 10, то нить подсчитывает число вхождений литеры letter и посылает

функции HowMany это число, а также длину обработанного участка. Если же длина больше 10, то нить запускает две новые нити SplitString, передав каждой из них соответствующую половину своей строки, и так далее.

Функция HowMany принимает токены от нитей SplitString, суммирует число вхождений и подсчитывает общую длину обработанных участков строки. Когда эта длина станет равной длине исходной строки, функция возвращает искомое число вхождений и завершает свою работу.

И, наконец, сделаем так, чтобы при каждом вызове функции HowMany она порождала нить SplitString уникального цвета (все следующие порождаемые нити будут того же цвета). Тогда этой функцией могут одновременно пользоваться разные функции-нити программы.

```
//функция получает литеру letter, адрес строки S и длину этой строки N
int HowMany(char letter, char *S, int N)
{  int rezCount, rezLength, result=0;
   // Создаем уникальный цвет порождаемых нитей
   int threadColor = getNewColor();
// Запускаем нить SplitString
   token( 1:letter, 2:S, 3:0, 4:N-1, SplitString( threadColor ) );
// Ожидаем токены уникального цвета от нитей SplitString,
// содержащие число вхождений и длину обработанного участка строки
   while(N>0)
      { request F( &rezCount, &rezLength, (threadColor) );
        result = + rezCount;
        N =  -  rezLength;
      }
// Возвращаем результат – число вхождений литеры letter в строку S
   return  result;
}
thread SplitString( char letter, char *S, int ps, int pe )
{  int pm, i, count=0;
   if( pe-ps+1>10 )
// Строка слишком длинная.
// Делим ее пополам и запускаем две нити SplitString
   {   pm = (pe-ps+1)/2;
       token( 1:letter, 2:S, 3: ps, 4:pm-1, SplitString() );
       token( 1:letter, 2:S, 3: pm, 4:pe, SplitString() );
   }
else
// Строка короткая. Считаем сами и посылаем результат нити HowMany
   {   for( i=ps; i<=pe; i++ )
             if( S[i] == letter) count++;
       token( 1:count, 2:pe-ps+1, HowMany.F() );
   }
```

```
}
```

4. Дополнительные возможности

В этом разделе описаны некоторые дополнительные средства языка DCF, позволяющие увеличить его выразительные возможности. Конечно же, возможности расширения языка не исчерпываются описанными ниже средствами; они скорее иллюстрируют некоторые важные направления таких расширений.

4.1. Функция getRequestColor

В разделе 3.2 была определена функция getThreadColor, которая позволяет определить цвет нити, т.е. цвет токена (группы токенов), запустивших данную нить. В ряде случаев могут оказаться полезными функции, позволяющие определить цвет токена (группы токенов), возобновивших выполнение нити (т.е. полученных request-функцией). Примером может служить функция **getRequestColor**, позволяющая определить цвет последней группы токенов, полученных request-функцией. Формат обращения к функции таков:

int getRequestColor (*имя_request-функции*, int *V)

Выполнение функции состоит в том, что она последовательно заполняет элементы вектора V значениями элементов вектора цвета последней группы токенов, полученных request-функцией *имя_request-функции,* до тех пор, пока либо исчерпается вектор V, либо исчерпается вектор цвета. Возвращает функция значение, равное длине вектора цвета.

Если некоторый компонент вектора цвета замаскирован, то соответствующий элемент вектора V получает значение NULL. Значения элементов вектора V, для которых не нашлось значений элементов вектора цвета (т.е. длина вектора V больше длины вектора цвета) не определены.

Если request-функция *имя_request-функции* еще не получала ни одной группы токенов, то функция возвращает значение нуль (при этом значение вектора V не определено).

Функция getRequestColor позволяет нити одновременно обрабатывать несколько потоков токенов, идущих из разных источников. Рассмотрим, например, следующий фрагмент функции-нити Func:

```
while( signal )
   { request Func.R1( &x, (*) );
     getRequestColor( Func.R1, inColor );
     switch( inColor[0] )
        { case 1:    sum =+ x;
                     break;
          case 2:    mul =* x;
                     break;
          case 3:    signal = false;
                     break;
        }
```

```
}
```

В этом цикле ожидается приход токенов трех цветов, возможно, от множества разных источников. Если приходит токен с цветом 1, то присланное им значение используется для увеличения значения переменной sum, если с цветом 2, то для увеличения значения переменной mul, а если с цветом 3, то это является сигналом окончания цикла.

4.2. Общедоступные функции-нити

Описанная в разделе 3.7 функция HowMany имеет существенный недостаток: если некоторая функция-нить вызвала функцию HowMany, то она должна дождаться возврата результирующего значения. В ряде случаев хотелось бы иметь возможность вызвать функцию HowMany и продолжить работу, а результат получить позже, когда он потребуется.

В рамках описанных выше средств языка DCF такую возможность можно реализовать с помощью введения дополнительной функции-нити. Рассмотрим, например, функцию-нить MyFunc, которой необходимо выполнить функцию HowMany, но которая хотела бы между вызовом функции HowMany и получением результата ее работы поделать что-нибудь “полезное”. Программа может выглядеть следующим образом:

```
thread MyFunc( … )
{
   …
   // здесь мы хотим вызвать функцию HowMany
   token( 1:letter, 2:String, 3:Length, StartHowMany() );
   …
   // здесь мы хотим получить результат работы функции HowMany
   request R( &n, () );
   …
}

thread StartHowMany( char letter, char *S, int N )
{  int n = HowMany( letter, S, N );
   token( n, MyFunc.R() );
}
```

В этом примере нить StartHowMany играет чисто вспомогательную роль – она дает возможность нити MyFunc не блокироваться при вызове функции HowMany.

Возникает вопрос – можно ли изменить описание **функции** HowMany таким образом, чтобы она стала общедоступной **функцией-нитью** HowMany? Препятствием служит то обстоятельство, что функция token требует указывать конкретное место назначения посылаемого токена – конкретную request-функцию или конкретную функцию-нить. А нам нужно, чтобы общедоступная

функция-нить могла посылать токены разным функциям, а именно тем функциям, которые ее запустили.

Существует по меньшей мере два пути решения этой проблемы.

Первый путь состоит в том, чтобы разрешить маскировать в теге токена не только цвет, но и имя функции-назначения. В этом случае токен будет отправляться без конкретного адреса и его смогут “подхватить”вообще говоря разные request-функции или функции-нити.

Второй путь состоит в том, чтобы дать возможность передавать общедоступной функции-нити в качестве параметра указатель на request-функцию (или, в общем случае, на функцию-назначение). Тогда общедоступная функция-нить сможет отправлять токены с результатами своей работы на указанную ей функцию-назначение.

Авторы оставляют решение вопроса о необходимости языковой поддержки этих способов до более детального изучения особенностей реализации и использования языка DCF при описании различного рода параллельных алгоритмов.

4.3. Системные токены

В ряде ситуаций пространство токенов само порождает токены, которые нити могут обнаруживать и использовать для идентификации и обработки возникшей ситуации или события. Такие токены называются **системными токенами**.

Программа пользователя также может порождать системные токены в тех случаях, когда требуется получить какой-то сервис со стороны системного окружения.

Системные токены имеют тег с именем, идентифицирующим группу особых событий, и уникальным цветом, идентифицирующем конкретное событие в группе. Значение токена в некоторых случаях несет дополнительную информацию о событии.

Примеры системных токенов:

token(*значение*, THREAD_ERROR(THREAD_ABORT))
: возникает при аварийном завершении какой-то нити;

token(*значение*, SYS_ERROR(IO_ ERROR))
: возникает при появлении ошибки ввода/вывода;

token(*значение*, *имя_системной_нити*(EXCEPTION))
: такой токен посылает пользовательская программа при необходимости обработки исключительной ситуации, определяемой пользователем.

Если программист желает определить свою собственную обработку какой-то исключительной ситуации, то он должен определить функцию-нить со стандартным именем, связанным с данным классом исключительных ситуаций. Эта функция-нить должна на основе анализа цвета запустивших ее системных токенов и полученного значения идентифицировать возникшую ситуацию и выполнить необходимые действия.

5. Заключение

Предлагая в данном препринте эскиз языка программирования DCF, предназначенного для программирования в гибридной модели вычислений data-control-flow, авторы хотели не только изложить свои соображения о возможных принципах построения такого языка, но и привлечь внимание к новым моделям вычислений, которые могут оказаться полезными в параллельном программировании уже в настоящее время или в ближайшем будущем. В этом смысле современные варианты моделей data-control flow, в частности, модели с разделением проверки готовности нити к выполнению и выполнения (decoupled data-driven models) выглядят перспективно и интенсивно исследуются.

В дальнейшем авторы предполагают провести исследования возможностей эффективной реализации языка DCF для различных архитектур параллельных компьютеров. При этом особо следует отметить, что реализация пространства токенов или, другими словами, dataflow-управления, может быть эффективно поддержано на аппаратном уровне за счет использования ассоциативной памяти, например, как в проекте [18]. По крайней мере, в данный момент не видно принципиальных трудностей для этого.

Авторы выражают благодарность за полезные обсуждения вариантов языка Ю.Л.Сазанову.

6. Литература

1. J.Silc, B.Robic, T.Ungerer. Asynchrony in parallel computing: From dataflow to multithreading. *Parallel and Distributed Computing Practices*, N1, v.1, 1998, pp.3-30.

2. W.A.Najjar, E.A.Lee, G.R.Gao. Advances in the dataflow computational model. *Parallel Computing*, N25, 1999, pp.1907-1929.

3. J.-L.Gaudiot, L.Bic. Advanced Topics in Data-Flow Computing. Prentice Hall, New York, 1991.

4. R.Buehrer, K.Ekanadham. Incorporating dataflow ideas into von Neumann processors for parallel processing. *IEEE Trans. Computers*, C-36, 1987, pp.1515-1522.

5. R.A.Ianucci. Toward a dataflow/von Neumann hybrid architecture, in *Proc. 15th ISCA*, May, 1988, pp. 131-140.

6. J.Dennis, G.Gao. Multithreaded Architecture: Principles, Projects and Issues. Multithreaded Computer Architecture: a Summary of the State of Art, Kluwer Academic Publishers, Dordrecht, 1994.

7. C.Kim, J.-L.Gaudiot. Dataflow and Multithreaded Architecture. Encyclopedia of Electrical and Electronics Engineering, Wiley, New York, 1997.

8. P.Evripidou. D3-Machine: A decoupled data-driven multithreaded architecture with variable resolution support. *Parallel Computing*, N9, v.27, 2001, pp.1197-1225.

9. M.Loikkanen, N.Bagerzadeh. A fine-grain multithreading superscalar architectures, in *Proc. PACT'96*, Oct. 1996, pp.163-168.

10. U.Sigmund, R.Ungerer, Identifying bottlenecks in multithreaded superscalar microprocessor. *Lecture Notes Comp. Sci*., N1123, 1996, pp.797-800.
11. W.F.Wong, K.S.Loh. Multiple context multithreaded superscalar architecture. *J. of System Architecture*, N3, v.46, 2000, pp.243-258.
12. J.Gonzalez, A.Gonzalez. Data value speculation in superscalar processors. *Microprocessors and Microsystems*, N22, 1988, pp. 293-302.
13. P.Marcuello, A.Gonzalez, J.Tubella. Speculative multithreaded processors. in *Proc. of Int. Symp. on Supercomputing*, ACM, 1998, New York, pp.77-84.
14. A Tutorial Introduction to Sisal, A High Performance, Portable, Parallel Programming Language. http:// www.sys.uea.ac.uk/~jrwg/Sisal/
15. A.C.Sodan. Application on a multithreaded architectures: a case study with EARTH-MANNA. *Parallel Computing*, N1, v.28, 2002, pp.3-33.
16. MIDC Language Tutorial. http:// www.cs.colostate.edu/~dataflow/papers.html
17. X.Tang, J.Wang, K.B.Teobald, G.R.Gao. Thread partitioning and sheduling based on cost model. in Proc. of the SPAA'97, Newport, June 1997, pp.272-281.
18. В.С.Бурцев. Система массового параллелизма с автоматическим распределением аппаратных средств СуперЭВМ в процессе решения задачи. Юбилейный сборник трудов ОИВТА РАН, М., 1993, с.5-27.
19. В.С.Бурцев, Л.Г.Тарасенко. Использование стандартных микропроцессоров в системе потока данных. В сб. Вычислительные системы с нетрадиционной архитектурой. Супер ВМ, М., 1995, с.3-30.
20. А.Н.Андрианов, А.Б.Бугеря, К.Н.Ефимкин, И.Б.Задыхайло. Норма. Описание языка. Рабочий стандарт. Препринт ИПМ им.М.В.Келдыша РАН, N120, 1995, 50с.