УДК 004.27 DOI: 10.31114/2078-7707-2022-4-101-108

Исследование различных вариантов реализации программной конструкции «цикл» в потоковой модели вычислений

Н.Н. Левченко, Д.Н. Змеев

Институт проблем проектирования в микроэлектронике РАН, г. Москва, nick@ippm.ru

***Аннотация* — В статье рассматриваются варианты отображения классической программной конструкции «цикл» в потоковую парадигму программирования, реализованную в архитектуре параллельной потоковой вычислительной системы (ППВС). Кратко описаны потоковая модель вычислений с динамически формируемым контекстом и реализующая её архитектура ППВС. Приведены примеры, поясняющие специфику использования вариантов отображения конструкции «цикл». Отдельно описаны методы контроля окончания циклов. Проанализированы преимущества и недостатки описанных вариантов и методов.**

***Ключевые слова* — параллельная потоковая вычислительная система, отображение программной конструкции, контроль окончания цикла, потоковая модель вычислений.**

I. Введение

В настоящее время в суперкомпьютерных технологиях чрезвычайно актуальна проблема распараллеливания вычислений, поскольку рост производительности отдельных процессоров фактически прекратился. Перспективным развитием в этом направлении является разработка вычислительных систем, которым на аппаратном уровне имманентно присущ параллелизм. К подобным системам относится разрабатываемая архитектура параллельной потоковой вычислительной системы (ППВС) «Буран», реализующая потоковую модель вычислений с динамически формируемым контекстом.

При создании многопроцессорных высокопроизводительных вычислительных систем основное внимание разработчиков направлено на аппаратную часть, поскольку традиционно считается, что аппаратура «должна» поддерживать всё программное обеспечение, которое было создано до этого.

Однако данный принцип препятствует развитию новых идей, поскольку они зачастую плохо сопрягаются с имеющимся программным обеспечением. Низкая реальная производительность (не превышает 3-5%, о чем свидетельствует результаты теста HPCG [1]) современных суперкомпьютеров при решении актуальных задач, в которых присутствует работа со сложноорганизованными данными, обусловлена отчасти тем, что многопроцессорные многоядерные вычислительные системы кластерного типа основываются на массовых микропроцессорах и ускорителях вычислений GPGPU [2]. А микропроцессоры и ускорители вычислений в свою очередь реализуют традиционную (фоннеймановскую) модель вычислений, которая создавалась как изначально последовательная, что приводит к трудностям при масштабировании многопроцессорных систем и программ для них. Обычно программы для суперкомпьютеров пишутся исходя из двухуровневой модели системы, в которой имеются узлы с быстрым доступом к локальной памяти и большим временем передачи данных другим узлам, причем это время часто принимается одинаковым для всех пар узлов. В реальности эти времена сильно различаются и такие программы работают плохо. В результате возникает нетривиальная задача адаптировать программы к таким неоднородностям.

В Институте проблем проектирования в микроэлектронике РАН ведется работа по созданию универсальной вычислительной системы, базирующейся на оригинальной потоковой модели вычислений с динамически формируемым контекстом.

Потоковая модель вычислений с динамически формируемым контекстом не ставит целью поддержать традиционную модель программирования. В ней отсутствуют традиционные циклы и массивы. Вместо них имеется «огромное» пространство виртуальных узлов, идентификация которых происходит по имени и набору индексов (полей контекста). Концептуальные различия в принципах программирования с императивной парадигмой программирования, применяемой в традиционных системах как раз и являются одним из препятствий к повсеместному распространению вычислительных систем с потоковой моделью вычислений. Для решения этой проблемы было предложено несколько языков программирования (DFL, HPL), основанных на потоковой модели вычислений с динамически формируемым контекстом. Представление алгоритмов на этих языках радикально отличается от привычного представления алгоритмов в традиционных языках, таких как C/C++, Фортран, Python и т.п, а также функциональных: Lisp, ML, Haskel и т.п. Автоматическое отображение программ из традиционных языков в языки DFL, HPL возможно только для очень ограниченного класса исходных программ. В общем случае требуется полное перестроение алгоритма вручную. Исходный текст на традиционном языке может служить лишь ориентиром. Принципы программирования на языках DFL, HPL значительно отличаются от используемых в программировании для традиционных систем.

Конструкция «Цикл», которая обеспечивает в основном многократное повторение некоторого набора инструкций (тела цикла), является одной из основных в традиционных языках программирования. Данная статья посвящена вопросу отображения программной конструкции «цикл» в потоковую парадигму программирования, которая реализуется параллельной потоковой вычислительной системой.

II. Модель вычислений и архитектура ППВС

В общем виде модель вычислений можно представить в виде сопоставляющего устройства (рабочего пространства токенов) бесконечного размера, на вход которого приходят данные (токены, представляющие собой структуру данных, содержащую сам операнд и некоторый набор признаков), которые могут быть как внешними (начальными данными задачи), так и внутренними (образованными в результате выполнения задачи). Внутри сопоставляющего устройства происходит поиск данных с «совпадающими» признаками. При обнаружении таких данных активируется выполнение программы, связанной с этими данными, а сами данные удаляются из сопоставляющего устройства. Выполнение программы обработки данных осуществляется на вычислителе. В результате выполнения программы образуются новые данные, которые либо подаются на вход сопоставляющего устройства, либо выдаются в качестве результата выполнения программы.

Потоковая модель вычислений с динамически формируемым контекстом [3-8] воплощена в языке параллельного программирования – DFL [9]. Программа на языке DFL представляет собой виртуальный граф вычислений в виде набора программных узлов, соединенных ребрами, которые обозначают пути обмена данными (токенами) между узлами согласно программе. Токен содержит операнд, ключ (контекст), который однозначно определяет положение операнда в виртуальном адресном пространстве задачи, и ряд служебных полей (рис. 1).

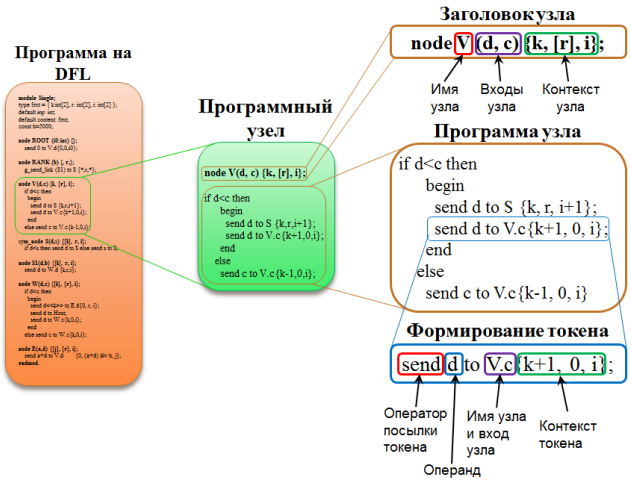

Рис. 1. Структура программы на параллельном языке DFL ППВС

Программный узел состоит из заголовка узла (включающего имя программного узла, список входов и контекст) и программного кода (набора операций, последовательно исполняемых на вычислителе). Активация конкретного экземпляра программного узла происходит только при условии, что на все его входы поступили токены с «одинаковым» именем узла и контекстом. Результатом активации программного узла является его исполнение на вычислителе, которое не прерывается на подкачку дополнительных данных, поскольку оперирует исключительно входными данными, контекстом и константами. Между собой программные узлы обмениваются только токенами, которые образуются как результат исполнения программного кода. При такой организации вычислительного процесса исключена вероятность его искажения и самоблокирования (при правильно составленной программе), так как программа узла обрабатывает только данные, поступившие к ней на вход. А поскольку отсутствует повторное использование цепей графа, т.е. данные в виртуальный узел поступают только один раз, что эквивалентно принципу, заложенному в языках однократного присваивания, то при параллельно выполняемой обработке данных исключается сама возможность использования «устаревших» данных.

Потоковая модель вычислений с динамически формируемым контекстом характеризуется тем, что атрибуты ключа токенов, формируемых в ходе выполнения программы узла, вычисляются непосредственно в этом же коде согласно программе узла. Это означает, что данная модель вычислений функционирует в парадигме «раздачи», когда тот, кто формирует новое значение, знает кому оно потребуется и обеспечивает его рассылку получателям. Традиционная модель вычислений функционирует в парадигме «сбора», при которой производитель новых данных ничего не знает об их потребителе, и, соответственно, у потребителя отсутствует информация об их готовности.

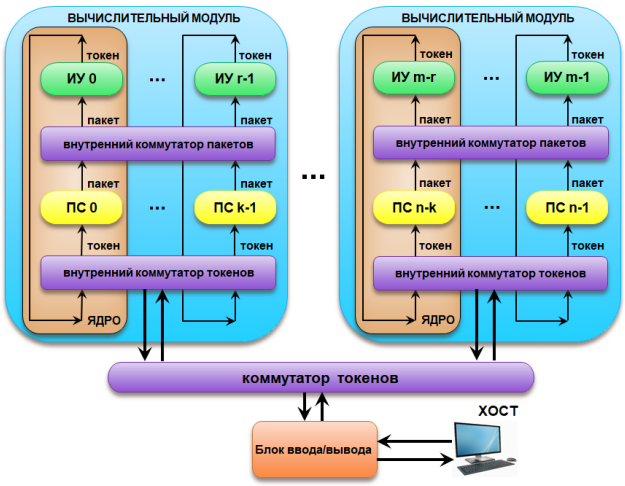

Рис. 2. Архитектура параллельной потоковой вычислительной системы

Потоковая модель вычислений реализована в архитектуре параллельной потоковой вычислительной системы (ППВС) «Буран». ППВС представляет собой па-

раллельную многопроцессорную вычислительную систему (рис. 2), состоящую из вычислительных модулей, объединенных общей коммуникационной сетью, и блока ввода/вывода. По коммуникационной сети передаются исключительно токены. Вычислительный модуль является процессором (в рамках одного кристалла), который может объединять несколько вычислительных ядер, связанных между собой локальным коммутатором токенов и локальным коммутатором пакетов. Коммутация между вычислительными ядрами выполняется по номеру ядра, которое вырабатывается хэш-функцией на основе контекста токена.

В состав вычислительного ядра входят процессор сопоставлений и исполнительное устройство. Процессор сопоставлений из поступающих на его вход токенов формирует готовую к исполнению структуру данных – пакет, который поступает в локальный коммутатор пакетов, выполняющий роль аппаратного балансера. Локальный коммутатор пакетов направляет готовые к исполнению пакеты на любое из свободных исполнительных устройств вычислительного модуля. Исполнительные устройства (ИУ) в архитектуре ППВС обезличены. Пакет всегда обрабатывается от начала и до конца на одном ИУ. Программа узла никогда не бывает статически приписана к какому-либо конкретному ИУ. Результатом обработки пакета является формирование новых токенов, которые передаются на локальный коммутатор токенов, где определяется, на какой вычислительный модуль направлен токен – внешний или внутренний.

Более подробно архитектура ППВС описана в работах [10-12].

III. Методы организации программной конструкции «цикл» на параллельной потоковой вычислительной системе

Благодаря использованию циклов и многократному запуску процедур и функций (в том числе и рекурсивных), которые обеспечивают выполнение отдельных частей программы множество раз, относительно небольшая традиционная (последовательная) программа может выполнять миллионы операций.

При программировании в потоковой парадигме присутствуют аналогичные возможности. Однако особенности (и возможности) процессора сопоставлений, а также реализованная в вычислительной системе парадигма однократного присваивания, позволяют во многих случаях избегать затрат на организацию циклического или рекурсивного выполнения отдельных частей программы. «Правильный» выбор реализации цикла на параллельном языке ППВС (DFL) оказывает существенное влияние на итоговую эффективность выполнения программы на ППВС.

В парадигме «раздачи» существует несколько основных методов отображения конструкции «Цикл».

A. Отображение цикла через контекст

Основным методом отображения циклов в парадигме «раздачи» является отображение цикла через контекст, в том случае, когда конструкция содержит переменную цикла.

Рассмотрим его на примере задачи «Сложение двух векторов». Требуется сложить два вектора:

$$(\mathbf{a_0}, \mathbf{a_1}, \ldots, \mathbf{a_n}) \text{ и } (\mathbf{b_0}, \mathbf{b_1}, \ldots, \mathbf{b_n}).$$

В традиционном программировании один из вариантов реализации задачи выглядит следующим образом:

```
for i := 1 to n do
    c[i] := a[i] + b[i];
```

Для сложения векторов на вычислительной системе ППВС необходимо **n** раз выполнить программу узла **Sum**, которая заключается в сложении данных, пришедших на её вход, на параллельном языке DFL выглядит следующим образом:

```
node Sum (x, y);
    send (x+y) to Host;
```

Решение этой задачи на ППВС выполняется без организации цикла в явном виде (как это делается в традиционном программировании). Для этого нужно представить, что не один и тот же узел **Sum** активируется **n** раз, а существует **n** копий узла **Sum**, каждая из которых активируется только один раз независимо от других копий этого узла.

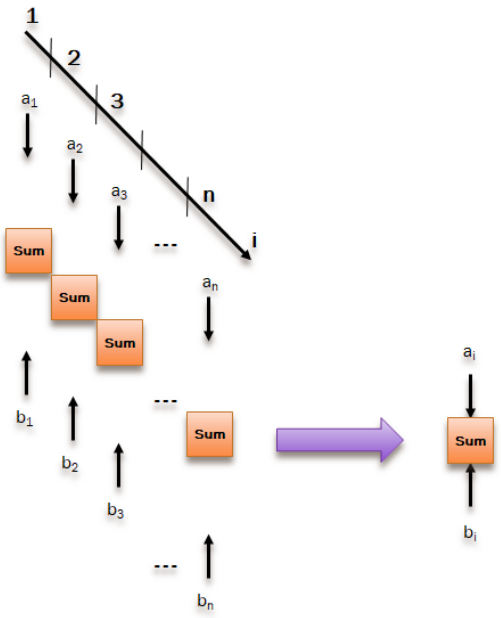

Рис. 3. Алгоритм решения задачи «сложение двух векторов»

Копии узла **Sum** расположены в виртуальном адресном пространстве вдоль оси **i** (рис. 3). Каждая из этих копий узла имеет пространственную координату **i (i = 1, 2, …, n)**.

Тогда для выполнения этой программы следует послать элементы входных векторов (в виде входных

токенов) на виртуальные экземпляры узлов. Элементы вектора **a** на первый вход (**x**) узла **Sum**:

send a_1 to Sum.x {1};

send a_2 to Sum.x {2};

…

send a_n to Sum.x {n};

Элементы вектора **b** на второй вход (**y**) узла **Sum**:

send b_1 to Sum.y {1};

send b_2 to Sum.y {2};

…

send b_n to Sum.y {n};

В поле контекста каждого токена указывается координата (**i**) в виртуальном адресном пространстве задачи экземпляра узла, на который направляется этот токен (или другими словами, место операнда, содержащегося в токене, в виртуальном адресном пространстве задачи). Далее, уже процессор сопоставлений на аппаратном уровне обеспечивает функционирование такого представления и, фактически, моделирует независимую работу виртуальных экземпляров узлов.

Такой подход к программированию полностью согласуется с архитектурой параллельной потоковой вычислительной системы и с принципами работы процессора сопоставлений, главным блоком которого является ассоциативная память. Исполнительные устройства ППВС являются физическими узлами, на которых выполняются виртуальные экземпляры узлов, причем параллельно и независимо друг от друга. Обозначим общее число ИУ **m**, тогда для эффективной работы системы должно выполняться правило **m<<n**, т.е. на каждый физический узел приходится некоторый набор виртуальных узлов.

Координата **i** является одним из полей контекста (по которому в ПС происходит поиск токенов), что обеспечивает сопоставление в ПС токенов с элементами исходных векторов **a** и **b** с соответствующими координатами: $\mathbf{a_1}$ с $\mathbf{b_1}$, $\mathbf{a_2}$ с $\mathbf{b_2}$ и до $\mathbf{a_n}$ с $\mathbf{b_n}$, т.е. те токены, которые были направлены на один и тот же виртуальный узел сопоставятся. В результате будет сформировано **n** пакетов, которые являются входными данными для **n** виртуальных экземпляров узлов. Эти пакеты будут автоматически распределяться между имеющимися физическими исполнительными устройствами, каждое из которых содержит копию программы узла **Sum**. Все **n** копий программы узла **Sum** будут выполнены на **m** исполнительных устройствах независимо от соотношения **n** и **m**.

На вход каждого экземпляра программы узла **Sum**, активированного в исполнительном устройстве, помимо входных данных $\mathbf{a_i}$ и $\mathbf{b_i}$, поступает и координата **i** (в контексте пакета), т.е. информация, которая однозначно определяет, какой именно виртуальный экземпляр программы **Sum** выполняется на данном ИУ. В рассматриваемом примере, эта информация не используется при вычислениях в программе, но сформированному выходному (результирующему) токену будет присвоен этот контекст. Благодаря этому, результат, поступающий на ХОСТ-машину, будет представлять собой **n** токенов, каждый из которых содержит ($\mathbf{a_i}$+$\mathbf{b_i}$) и контекст **i**.

Вложенные циклы реализуются аналогичным образом. В этом случае, для каждого из циклов (переменной цикла) задается своё поле в контексте виртуального узла. Если вложенные циклы имеют общее тело цикла, то оно может быть отображено в один программный узел (в том случае, когда удовлетворяет по числу входных данных). В противном случае, когда каждый из циклов содержит собственное тело цикла, или тело циклов оперирует с большим числом данных, они отображаются через группу программных узлов. Контекст же однозначно определяет положение программного узла (фактически, тела узла) в виртуальном адресном пространстве задачи. Чем ниже зависимость тела узла друг от друга на каждом из уровней цикла, тем выше уровень параллелизма может быть достигнут на параллельной потоковой вычислительной системе.

Описанный подход к организации цикла связан прежде всего с принципом однократного присваивания, реализованного на аппаратном уровне, а также с двухвходовыми программными узлами. Данный метод реализации цикла на ППВС является наиболее универсальным.

B. Метод развертывания цикла

Вторым методом организации циклов в парадигме «раздачи» является последовательное развертывание цикла.

Этот метод находит свое применение при отображении циклов, не имеющих счётчика циклов – в традиционном программировании это циклы типа **while** с выходом по достижению какого-либо условия.

С формальной точки зрения такой тип цикла может быть реализован в потоковой парадигме без использования полей контекста. В этом случае в вычислительной системе в один момент существует только одна итерация такого цикла (представляющая собой набор программных узлов). Любые токены, которые образуются при ее выполнении и посылаются на свои же экземпляры программных узлов, фактически, будут принадлежать следующей итерации благодаря реализованному принципу однократного присваивания. Существенным минусом такой реализации цикла является невозможность извлечь из нее параллелизм.

Дальнейшим развитием этого метода является использование нескольких бит полей контекста для различия токенов разных итераций друг от друга (в циклическом режиме). В этом случае любой имеющийся в программе параллелизм будет автоматически выявлен аппаратурой. Кроме того, имеется возможность за счёт спекулятивных вычислений, обрабатывать при наличии данных следующие итерации цикла еще до

проверки условия на выход из цикла основной итерации. Поскольку обработка таких итераций выполняется с более низким приоритетом, то это не вредит эффективности выполнения основной итерации, и позволяет задействовать свободные аппаратные ресурсы вычислительной системы.

При наличии же свободных полей контекста для удобства программирования они могут быть выделены в целях идентификации итераций между собой. Этот метод похож на первый в плане использования поля контекста, но принципиально отличается в способе активаций экземпляров программного узла. В первом случае они могут активироваться параллельно. Во втором – последовательно. Это связано с тем, что операнд на один из входов (первый - **x**) программного узла, должен прийти от предыдущего экземпляра программного узла (в соответствии с пространственной координатой, которая содержится в контексте).

Рассмотрим метод подробнее на примере задачи «сложение элементов вектора» (рис. 4).

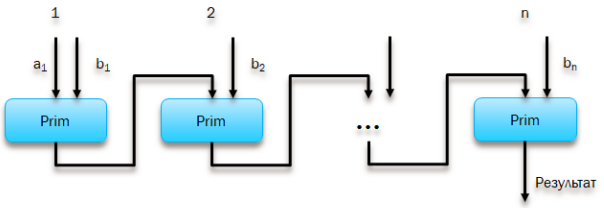

Рис. 4. Алгоритм решения задачи «сложение элементов вектора»

Требуется получить сумму элементов вектора:

$$(b_1, b_2, \ldots, b_n).$$

В традиционном программировании данная задача решается, например, при помощи следующего цикла:

```
sum := 0;
i := 1;
while i <= n do
    begin
        sum := sum + b[i];
        i := i + 1;
    end;
```

В парадигме «раздачи» данная задача может быть решена следующим образом. Для получения искомого значения, следует **n** раз выполнить программу узла **Prim**, которая сумму пришедших на её вход данных передает на экземпляр узла **Prim** при этом контекст узла (**i**) увеличивается на единицу. На языке DFL программный узел выглядит следующим образом:

```
node Prim (x, y) {i};
    send (x+y) to Prim.x {i+1}.
```

Так же, как и в предыдущем примере, эта задача решается без непосредственной организации цикла. Существует **n** копий узла **Prim**, каждая из которых активируется только один раз «независимо» от других копий этого узла. Копии узла **Prim** расположены в виртуальном адресном пространстве вдоль оси **i**. Каждая из этих копий узла имеет свою пространственную координату **i** (**i = 1, 2, …, n**). В итоге, для выполнения этой программы следует послать элементы входного вектора (в виде входных токенов) на виртуальные экземпляры узлов. Элементы вектора **b** на второй вход (**y**) узла **Prim**:

send b_1 to Prim.y {1};

send b_2 to Prim.y {2};

…

send b_n to Prim.y {n}.

А на первый вход (**x**) узла **Prim** с контекстом {**1**} посылается нулевое значение.

В поле контекста каждого токена указывается координата в виртуальном адресном пространстве (**i**) экземпляра узла, в который направляется этот токен.

Данная программа может быть модернизирована для сокращения числа активаций экземпляра узла **Prim** (с **n** до **n-1**). В этом случае на первый вход (**x**) узла **Prim** с контекстом {**1**} посылается последний элемент вектора b_n (**send b_n to Prim.x {1}**), а не нулевое значение; все остальные элементы посылаются также, как и в предыдущем случае.

Стоит отметить, что конкретно этот пример можно решить и при помощи суммирования пирамидой. Это позволяет значительно (примерно в два раза) ускорить выполнение данной программы за счёт параллельной обработки экземпляров узла.

Вложенные циклы этим методом отображаются следующим образом. На первом этапе происходит развертывание цикла по одной координате (с использованием полей контекста). На следующем этапе (при активации экземпляров узла) идёт развертывание цикла по второй координате и т.д.

C. Традиционный подход

Третьим методом организации циклов является традиционный подход – выполнение цикла в рамках одного программного узла. При создании программ в потоковой парадигме этот метод находит ограниченное применение, которое связано с несколькими факторами. Первым является ограничение на число входов в стандартную программу узла равное двум. При организации цикла внутри одного программного узла это означает, что число «внешних» переменных, которыми можно оперировать внутри тела цикла, также будет ограничено двумя. Вторым является отсутствие возможности экстрагирования параллелизма из такого цикла, что существенно снижает эффективность выполнения программы на многопроцессорной параллельной системе. Тем не менее, этот метод организа-

ции циклов находит свое применение на начальных этапах задачи при ее «развертывании».

При отображении вложенных циклов данный метод может быть совмещен с предыдущими, что позволяет при определенных условиях экономить аппаратные ресурсы вычислительной системы.

IV. Способы контроля окончания программной конструкции «цикл» на параллельной потоковой вычислительной системе

Рассмотрение методов отображение конструкции «Цикл» в потоковую парадигму будет неполным без рассмотрения способов контроля окончания циклов.

В традиционных языках программирования контроль выхода из цикла встроен в саму структуру конструкции цикл, и, соответственно, проверяется при каждой обработке тела цикла.

В потоковых параллельных программах, которые выполняются на ППВС, фактически отсутствует организация циклического повторения тела цикла, которая как раз характерна для традиционного программирования таких задач.

В ППВС используются следующие способы контроля окончания цикла:

- контроль границ;
- контроль наличием данных;
- стандартный контроль.

A. Контроль границ

Основной способ контроля окончания цикла заключается в автоматическом контроле границ виртуального адресного пространства задачи при помощи различных средств, основным из которых является токен-косвенность. Этот токен позволяет перенаправлять вычислительный процесс (т.е. пересылать токен на другой узел).

Рассмотрим на примере решения предыдущей задачи "сложения элементов вектора" (рис. 5). Согласно программе экземпляра узла **Prim**, результат последнего суммирования данных с контекстом {**n**} будет отправлен не на ХОСТ-машину, а на вход экземпляра узла **Prim.y** с контекстом **{n+1},** который расположен за пределами виртуального пространства задачи.

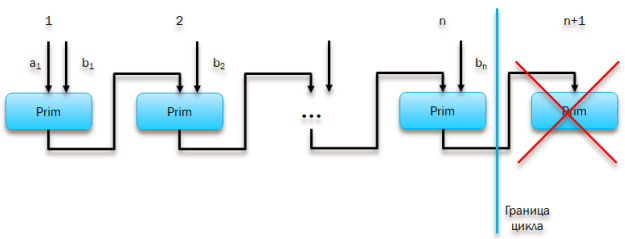

Рис. 5. Алгоритм решения задачи «сложение элементов вектора»

Однако зацикливания программы не произойдет. Работа программы завершится, поскольку на другой вход (**x**) узла **Prim** не поступит «парный» токен. В результате если не предпринять дополнительных действий, то вычисленное значение суммы элементов просто останется в ассоциативной памяти дожидаться «парного» токена.

Контроль окончания цикла необходим не для ограничения числа повторений тела цикла (экземпляра программного узла), поскольку оно будет повторено (в рассматриваемом примере) **n** раз в независимости от наличия или отсутствия этой проверки в программе узла, а для того, чтобы определить, в какую точку виртуального адресного пространства задачи следует направить результат суммирования.

Для этого необходимо сформировать токен-косвенность и направить его на узел **Prim** с контекстом {**n+1**}. В общем случае, этот токен будет ожидать в процессоре сопоставлений токен отправленный на любой вход узла **Prim** с контекстом {**n+1**}. Как только такой токен поступит в процессор сопоставлений, то будет сформирован пакет из «перехваченного» токена и отправлен на обработку в программный узел, адрес которого хранится в токене-косвенность, и который уже выдаст результат на ХОСТ-машину.

Очевидно, что способ «контроль границ» совсем не эквивалентен традиционной организации цикла по затратам. Аппаратные затраты в первом случае берет на себя ассоциативная память, которая, фактически, и контролирует границу цикла или, в более широком смысле, границу виртуального адресного пространства задачи. Учитывая, что в ППВС используется тернарная ассоциативная память, которая помимо самого значения сравнивает еще и маску обоих токенов (замаскированные разряды считаются заведомо совпавшими), то всего четыре токена-косвенности могут контролировать границы матрицы, тем самым значительно уменьшив «лишние» сравнения её внутренних значений при обработке тела цикла.

B. Контроль наличием данных

В ряде случаев контроль окончания цикла не требуется благодаря самим данным (токенам) и специфики работы ассоциативной памяти. В этом случае активируются только те экземпляры программных узлов, на все входы которых пришли токены, т.е. входные данные определяют границу виртуального адресного пространства задачи, а сам алгоритм решения задачи не предполагает выход за его пределы.

Максимальный эффект от данного метода можно получить при решении задач с разреженными данными. Например, в программе «перемножение матриц» это позволяет эффективно работать с разреженными матрицами, активируя и выполняя только те программные узлы, оба входа которого содержат ненулевые значения (нулевое значение просто не посылается на обработку) [13]. Таким образом, программный узел (отвечающий за перемножение данных), «содержащий» нулевое значение входа просто не будет активирован.

Именно этот способ контроля используется в примере «сложение векторов», который был рассмотрен в пункте 3.А (рис.3). Обеспечив посылку по 100 данных каждого из векторов, на выходе получим 100 результатов. Если же у первого вектора будет отправлено 100 данных, а у второго 50, то и результатов получено будет только 50. Оставшиеся без взаимодействия 50 данных первого вектора так и останутся в ассоциативной памяти дожидаться парных им данных.

C. Стандартный контроль

Третьим способом контроля окончания циклов является стандартная проверка условия в каждом из программных узлов. Вначале или при окончании обработки каждого соответствующего экземпляра программного узла проверяется, выполняется ли условие завершения (выхода из) цикла. Данный способ значительно менее эффективен по сравнению с предыдущими способами контроля, поскольку на каждой итерации выполняются такие «паразитные» проверки (ведь для окончания цикла, фактически, требуется проверка только на «последней» итерации). И эффективность тем ниже, чем больше число итераций в цикле.

V. Заключение

Аппаратное экстрагирование параллелизма в параллельной потоковой вычислительной системе осуществляется на различных уровнях:

- на уровне разных итераций;
- на уровне активаций одного программного узла;
- на уровне различных программных узлов параллельной программы;
- на уровне подзадач.

Эффективное отображение такой конструкции языков программирования как «цикл» в парадигму «раздачи» является залогом того, что аппаратура параллельной потоковой вычислительной системы сможет в полной мере экстрагировать весь параллелизм, имеющийся в программе.

Это достигается как за счёт аппаратуры системы – главным образом процессора сопоставлений, в основе которого лежит аппаратно реализованная ассоциативная память, так и благодаря парадигме «раздачи», лежащей в основе потоковой модели вычислений с динамически формируемым контекстом. Необходимость в барьерных синхронизациях по большому счету отпадает, что позволяет с большей эффективность использовать аппаратные ресурсы вычислительной системы.

Описанные в данной статье методы организации циклов и способы контроля их окончания позволяют создавать программы, которые эффективно выполняются на ППВС.

Литература

[1] TOP-500. HPCG – JUNE 2022. URL: https://top500.org/lists/hpcg/2022/06/ (дата обращения: 04.07.2022)

[2] CUDA Zone Library of Resources. URL: https://developer.nvidia.com/cuda-zone (дата обращения: 04.07.2022).

[3] Левченко Н.Н., Змеев Д.Н., Климов А.В., Окунев А.С., Стемпковский А.Л. Перспективы использования потоковой модели вычислений в высокопроизводительных вычислительных системах // Сборник трудов SoRuCom-2017. Четвертая Международная конференция «Развитие вычислительной техники в России и странах бывшего СССР: история и перспективы», 3–5 октября 2017 года, Москва, Зеленоград. 2017. С. 177 – 184.

[4] Левченко Н.Н., Окунев А.С., Стемпковский А.Л. Преимущества потоковой модели вычислений // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2018. Вып. 3. С. 24-30. DOI:10.31114/2078-7707-2018-3-24-30

[6] Змеев Д.Н., Климов А.В., Левченко Н.Н., Окунев А.С., Стемпковский А. Л. Потоковая модель вычислений как парадигма программирования будущего // Информатика и её применения. 2015. Т. 9. Выпуск 4. С. 29-36. DOI:10.14357/1992264150403

[7] Климов А.В., Левченко Н.Н., Окунев А.С., Стемпковский А. Л. Суперкомпьютеры, иерархия памяти и потоковая модель вычислений // Программные системы: теория и приложения: электрон. научн. журн. 2014. Т. 5. № 1(19). С. 15–36.

[8] Левченко Н.Н., Окунев А.С., Стемпковский А.Л. Использование модели вычислений с управлением потоком данных и реализующей ее архитектуры для систем экзафлопсного уровня производительности // Проблемы разработки перспективных микро- и наноэлектронных систем - 2012. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2012. С. 459-462.

[9] Климов А.В., Левченко Н.Н. Механизм ветвей в потоковом метаязыке UPL (METAL) и методы его реализации в ППВС «Буран» // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2018. Вып. 3. С. 31-37. DOI:10.31114/2078-7707-2018-3-31-37

[10] Стемпковский А.Л., Левченко Н.Н., Окунев А.С., Цветков В.В. Параллельная потоковая вычислительная система – дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // Журнал «ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ». 2008. №10. С. 2–7.

[11] Zmeev D.N., Klimov A.V., Levchenko N.N., Okunev A.S., Stempkovskii A.L. Features of the Architecture Implementing the Dataflow Computational Model and Its Application in the Creation of Microelectronic High-Performance Computing Systems // Russian Microelectronics. 2019. V. 48. № 5. P. 292-298. DOI:10.1134/S1063739719050111

[12] Бобков, С.Г., Левченко Н.Н., Окунев А.С. Параллельный потоковый процессор на новых архитектурных принципах для решения широкого круга задач // Наноиндустрия. 2020. Т. 13. № S5-2(102). С. 285-296. DOI:10.22184/1993-8578.2020.13.5s.285.296

[13] Змеев Д.Н., Окунев А.С. Разработка и исследование алгоритма задачи перемножения разреженных матриц для параллельной потоковой вычислительной системы «Буран» // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2018. Вып. 3. С. 16-23. DOI:10.31114/2078-7707-2018-3-16-23

Research of Various Options for Implementing Program Construction "Loop" in Dataflow Computing Model

N.N. Levchenko, D.N. Zmejev

Institute for Design Problems in Microelectronics of Russian Academy of Sciences, Moscow, nick@ippm.ru

***Abstract* — When creating multiprocessor high-performance computing systems, the main attention of developers is directed to the hardware, since it is traditionally believed that the hardware "should" support all the software that was created before. However, this principle prevents the development of new ideas, since they often do not fit well with existing software. The Institute for Design Problems in Microelectronics of the Russian Academy of Sciences is working on the creation of a universal computing system based on an original dataflow computing model with a dynamically formed context. This computing model lacks traditional loops and arrays. Instead, there is a "huge" space of virtual nodes, which are identified by name and a set of indices (context fields). This article is devoted to the issue of mapping the program construction "loop" into the dataflow programming paradigm. The article describes three methods of organizing the "loop" construction - mapping the cycle through the context, the method of unfolding the cycle and the traditional approach. Examples of their use are given. Three ways to control the end of cycles are also given: boundary control, data presence control and standard control, as well as options for their use. Efficient mapping of the programming languages "loop" construction into the "distribution" paradigm is a guarantee that the hardware of the parallel dataflow computing system will be able to fully extract all the parallelism that exists in the program.**

***Keywords* — parallel dataflow computing system, mapping of program construction, loop termination control, dataflow computing model.**

REFERENCES

[1] TOP-500. HPCG – JUNE 2022. URL: https://top500.org/lists/hpcg/2022/06/ (access date: 04.07.2022)

[2] CUDA Zone Library of Resources. URL: https://developer.nvidia.com/cuda-zone (access date: 04.07.2022).

[3] Levchenko N.N., Zmeev D.N., Klimov A.V., Okunev A.S., Stempkovskij A.L. Perspektivy ispol'zovanija potokovoj modeli vychislenij v vysokoproizvoditel'nyh vychislitel'nyh sistemah (Prospects for using dataflow computing model in high-performance computing systems) // Sbornik trudov SoRuCom-2017. Chetvertaja Mezhdunarodnaja konferencija «Razvitie vychislitel'noj tehniki v Rossii i stranah byvshego SSSR: istorija i perspektivy», 3–5 oktjabrja 2017 goda, Moskva, Zelenograd. 2017. C. 177 – 184.

[4] Levchenko N.N., Okunev A.S., Stempkovsky A.L. Advantages of Dataflow Computing Model // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2018. Issue 3. P. 24-30. doi:10.31114/2078-7707-2018-3-24-30

[6] Zmeev D.N., Klimov A.V., Levchenko N.N., Okunev A.S., Stempkovskij A. L. Potokovaja model' vychislenij kak paradigma programmirovanija budushhego (Dataflow computing model as a paradigm of future mainstream of software development) // Informatika i ejo primenenija. 2015. T. 9. Vypusk 4. S. 29-36.

[7] Klimov A.V., Levchenko N.N., Okunev A.S., Stempkovskij A. L. Superkomp'jutery, ierarhija pamjati i potokovaja model' vychislenij (Supercomputers, memory hierarchy and dataflow computation model) // Programmnye sistemy: teorija i prilozhenija: jelektron. nauchn. zhurn. 2014. T. 5. № 1(19). S. 15–36.

[8] Levchenko N.N., Okunev A.S., Stempkovsky A.L. The usage of dataflow computing model and architecture realizing these for exaflops performance system // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2012. Proceedings / edited by A. Stempkovsky, Moscow, IPPM RAS, 2012. P. 459-462.

[9] Klimov A.V., Levchenko N.N. Branches in the Dataflow Metalanguage UPL (METAL) and Methods of their Implementation in the PDCS “Buran” // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2018. Issue 3. P. 31-37.

[10] Stempkovskij A.L., Levchenko N.N., Okunev A.S., Cvetkov V.V. Parallel'naja potokovaja vychislitel'naja sistema – dal'nejshee razvitie arhitektury i strukturnoj organizacii vychislitel'noj sistemy s avtomaticheskim raspredeleniem resursov (Parallel dataflow computing system - the further development of architecture and the structural organization of the computing system with automatic distribution of resources) // INFORMACIONNYE TEHNOLOGII. 2008. №10. S. 2-7.

[11] Zmeev D.N., Klimov A.V., Levchenko N.N., Okunev A.S., Stempkovskii A.L. Features of the Architecture Implementing the Dataflow Computational Model and Its Application in the Creation of Microelectronic High-Performance Computing Systems // Russian Microelectronics. 2019. V. 48. № 5. P. 292-298.

[12] Bobkov, S.G., Levchenko N.N., Okunev A.S. Parallel'nyj potokovyj processor na novyh arhitekturnyh principah dlja reshenija shirokogo kruga zadach (Parallel dataflow processor based on new architectural principles for solving a wide range of tasks) // Nanoindustrija. 2020. T. 13. № S5-2(102). S. 285-296.

[13] Zmejev D.N., Okunev A.S. Development and Investigation of Algorithm of Sparse Matrices Multiplication Task for the Parallel Dataflow Computing System "Buran" // Problems of Perspective Micro- and Nanoelectronic Systems Development - 2018. Issue 3. P. 16-23. doi:10.31114/2078-7707-2018-3-16-23