MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 2, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

Dynamically Changing Parallelism with the Asynchronous Sequential Data Flows

A. I. Legalov[1], I. V. Matkovskii[1], M. S. Ushakova[1], D. S. Romanova[1]

DOI: 10.18255/1818-1015-2020-2-164-179

[1]Siberian Federal University, 79 Svobodny pr., 660041 Krasnoyarsk, Russia.

MSC2020: 68N15, 68Q10, 68Q85
Research article
Full text in Russian

Received May 27, 2020
After revision June 8, 2020
Accepted June 10, 2020

A statically typed version of the data driven functional parallel computing model is proposed. It enables a representation of dynamically changing parallelism by means of asynchronous serial data flows. We consider the features of the syntax and semantics of the statically typed data driven functional parallel programming language Smile that supports asynchronous sequential flows. Our main idea is to apply the Hoar concept of communicating sequential processes to the computation control on the data readiness. It is assumed that on the data readiness a control signal is emitted to inform the processes about the occurrence of certain events. The special feature of our approach is that the model is extended with the special asynchronous containers that can generate events on their partial filling. These containers are a stream and a swarm, each of which has its own specifics. A stream is used to process data which have identical type. The data comes sequentially and asynchronously at arbitrary time moments. The number of the incoming data elements is initially unknown, so the processing completes on the signal of the end of the stream. A swarm is used to contain independent data of the same type and may be used for the massive parallel operations performing. Unlike a stream, the swarm's size is fixed and known in advance. General principles of the operations with the asynchronous sequential flows with an arbitrary order of data arrival are described. The use of the streams and the swarms in various situations is considered. We propose the language constructions which allow us to operate the swarms and streams and describe the specifics of their application. We provide the sample functions to illustrate the use of the different approaches to description of the parallelism: recursive processing of the asynchronous flows, processing of the flows in an arbitrary or predefined order of operations, direct access and access by the reference to the elements of the streams and swarms, pipelining of calculations. We give a preliminary parallelism assessment which depends on the ratio of the rates of data arrival and their processing. The proposed methods can be used in the development of the future languages and tool-kits of architecture-independent parallel programming.

Keywords: parallel computations, asynchronous computations, static typing, dynamically changing parallelism.

INFORMATION ABOUT THE AUTHORS

Alexander I. Legalov correspondence author	orcid.org/0000-0002-5487-0699. E-mail: legalov@mail.ru Head of the scientific and educational laboratory of programming technologies, professor of the department of computer engineering, doctor of technical sciences, professor.
Ivan V. Matkovskii correspondence author	orcid.org/0000-0002-4801-7982. E-mail: alpha900i@mail.ru Senior teacher.
Mariya S. Ushakova correspondence author	orcid.org/0000-0003-4234-2714. E-mail: ksv@akadem.ru Assistent of the department of computer engineering.
Darya S. Romanova correspondence author	orcid.org/0000-0002-9020-4802. E-mail: daryaooo@mail.ru Graduate student.

For citation: A. I. Legalov, I. V. Matkovskii, M. S. Ushakova, and D. S. Romanova, "Dynamically Changing Parallelism with the Asynchronous Sequential Data Flows", *Modeling and analysis of information systems*, vol. 27, no. 2, pp. 164-179, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 2, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

Динамически изменяющийся параллелизм с асинхронно-последовательными потоками данных

А. И. Легалов[1], И. В. Матковский[1], М. С. Ушакова[1], Д. С. Романова[1]

DOI: 10.18255/1818-1015-2020-2-164-179

[1]Сибирский федеральный университет, 660041, Красноярский край, г. Красноярск, пр. Свободный, 79.

УДК 004.042
Научная статья
Полный текст на русском языке

Получена 27 мая 2020 г.
После доработки 8 июня 2020 г.
Принята к публикации 10 июня 2020 г.

Предлагается статически типизированная версия модели функционально-потоковых параллельных вычислений, которая за счет использования асинхронных последовательных потоков обеспечивает представление динамически изменяющегося параллелизма. Рассмотрены особенности синтаксиса и семантики статически типизированного языка функционально-потокового параллельного программирования Smile, обеспечивающие поддержку асинхронных последовательных потоков. Основная идея подхода базируется на использовании концепции взаимодействующих последовательных процессов Т. Хоара применительно к управлению по готовности данных. Предполагается, что готовность данных сопровождается выдачей управляющих сигналов, информирующих процессы о свершении тех или иных событий. Отличительной особенностью подхода является включение в модель специальных асинхронных контейнеров, которые могут порождать события по частичному заполнению. Этими контейнерами являются поток и рой, каждый из которых имеет свою специфику. Поток используется для обработки данных одного типа, поступающих последовательно и асинхронно в произвольные промежутки времени. Размерность поступающих данных изначально неизвестна, поэтому завершение обработки осуществляется по признаку конца потока. Рой используется для описания независимых данных одного типа, над которыми возможно выполнение массовых параллельных операций. В отличие от потока, его размерность фиксирована и известна заранее. В работе описаны общие принципы организации асинхронных последовательных потоков с произвольным поступлением данных. Рассматривается использование потоков и роев в различных ситуациях. Предлагаются языковые конструкции, позволяющие описывать работу с роями и потоками и особенности их применения. Представлены примеры функций, при реализации которых использованы различные подходы к описанию параллелизма: рекурсивная обработка асинхронных потоков, обработка потоков при недетерминированном и упорядоченном выполнении операций, прямое и ссылочное обращение к элементам потоков и роев, конвейеризация вычислений. Дается предварительная оценка параллелизма в зависимости от временных соотношений между темпом поступления данных и скоростью их обработки. Предложенные методы могут быть использованы при разработке перспективных языковых и инструментальных средств архитектурно-независимого параллельного программирования.

Ключевые слова: параллельные вычисления, асинхронные вычисления, статическая типизация, динамически-изменяющийся параллелизм.

ИНФОРМАЦИЯ ОБ АВТОРАХ

Александр Иванович Легалов автор для корреспонденции	orcid.org/0000-0002-5487-0699. E-mail: legalov@mail.ru Руководитель научно-учебной лаборатории технологий программирования, профессор кафедры вычислительной техники, доктор технических наук, профессор.
Иван Васильевич Матковский автор для корреспонденции	orcid.org/0000-0002-4801-7982. E-mail: alpha900i@mail.ru Старший преподаватель.
Мария Сергеевна Ушакова автор для корреспонденции	orcid.org/0000-0003-4234-2714. E-mail: ksv@akadem.ru Ассистент кафедры вычислительной техники.
Дарья Сергеевна Романова автор для корреспонденции	orcid.org/0000-0002-9020-4802. E-mail: daryaooo@mail.ru Аспирант.

Для цитирования: A. I. Legalov, I. V. Matkovskii, M. S. Ushakova, and D. S. Romanova, “Dynamically Changing Parallelism with the Asynchronous Sequential Data Flows”, *Modeling and analysis of information systems*, vol. 27, no. 2, pp. 164-179, 2020.

Введение

Разработка архитектурно-независимых параллельных программ определяется двумя следующими подходами:

- разработкой и отладкой последовательных программ, не связанных с параллельными вычислительными системами (ПВС), с последующим распараллеливанием под целевую архитектуру;
- изначальной разработкой программ или алгоритмов, описывающих максимальный параллелизм решаемой задачи с последующим «сжатием» этого параллелизма к целевой архитектуре.

В реальной ситуации чаще используется промежуточное решение, когда параллельная программа сразу же разрабатывается с учетом особенностей целевой архитектуры. Однако, в отличие от первых двух вариантов, формируемый код оказывается жестко связан с целевым решением, что затрудняет его перенос на другие ПВС.

Вместе с тем, независимо от используемых подходов следует отметить, что практически в любом из этих трех вариантов параллелизм программы фиксируется за счет использования не только изначально заданного базиса операций, но и из-за методов описания параллельных процессов. Зачастую это не связано с ориентацией на архитектурную зависимость, а определяется спецификой как моделей вычислений, так и построенных на их основе языковых и инструментальных средств.

Фиксация параллелизма как сверху (максимальным параллелизмом), так и снизу (последовательным выполнением) ведет к семантическому разрыву на уровне реального вычислителя. То есть, при выполнении программы в реальных вычислительных ресурсах происходит потеря эффективности и сбалансированности из-за того, что зафиксированные при разработке характеристики параллельного алгоритма вступают в противоречие с динамическими характеристиками подсистем ПВС. Именно поэтому максимальный параллелизм зачастую ужимается при доводке программы под особенности вычислителя, что ведет к решению задачи, противоположной распараллеливанию последовательных программ. Это, в свою очередь, ведет к потере эффективности процесса разработки параллельного программного обеспечения, не позволяя писать программу один раз и для различных параллельных архитектур.

В связи с этим актуальной является задача поиска моделей параллельных вычислений и построение на их основе языковых и инструментальных средств, обеспечивающих адаптивную подстройку параллелизма однажды написанной программы для ее эффективного наложения на различные вычислительные ресурсы.

Одним из таких решений является использование данных по мере их поступления. То есть, без накопления в массивах перед последующими вычислениями. Предполагается, что их обработка происходит по мере готовности отдельных элементов. Параллелизм при обработке таких данных зависит как от темпа поступления этих элементов в массивы, так и от времени выполнения их последующей обработки. Отношение между этими временами позволяет рассматривать разные уровни параллелизма соответствующих алгоритмов — от последовательных вычислений до неограниченного распараллеливания. Одна из таких структур данных, названная асинхронным списком, была предложена в ходе расширения функционально-потоковой модели параллельных вычислений (ФПМПВ) [1].

Вместе с тем следует отметить, что данный вид асинхронных потоков данных обладает следующим недостатком: порядок следования аргументов, поступающих на выполнение, может не совпадать с порядком их следования в выходном потоке, используемом для сбора результатов. Это изменение может быть некритичным для некоторых алгоритмов. Однако в большинстве случаев подобная ситуация недопустима. Одним из вариантов решения данной проблемы является организация упорядоченных очередей, когда результаты формируются в той же последовательности,

что и порождающие их аргументы. Однако в этой ситуации происходит неявная синхронизация потока асинхронно поступающих данных, что, в свою очередь, ведет к замедлению их обработки.

Для решения проблемы предлагается подход, базирующийся на расширении возможностей параллельного списка ФПМПВ. Предполагается, что данные, попадающие в этот список, могут сразу же поступать на обработку, после которой они устанавливаются на те же позиции в списке результатов. Дополнительное изменение семантики можно внести также и в рассмотренный ранее асинхронный список. Применение предлагаемых изменений позволяет создавать новую разновидность модели вычислений, описывающей асинхронные алгоритмы с динамически изменяемым параллелизмом.

В работе рассматриваются особенности семантики операторов, обеспечивающих поддержку асинхронных потоковых вычислений, а также их отображение на синтаксис и семантику статически типизированного языка функционально-потокового параллельного программирования Smile. Приводятся примеры, демонстрирующие специфику предлагаемых конструкций. Дается предварительная оценка параллелизма в зависимости от временных соотношений между темпом поступления данных и скоростью их обработки.

1. Основные идеи, определяющие подход

Основная идея подхода базируется на концепции, предложенной Хоаром [2], в которой параллелизм описывается как взаимодействие процессов, порождаемых через последовательно формируемые события. Их последовательность обуславливается тем, что моменты возникновения событий считаются мгновенными и на оси времени выстраиваются в очередь. Это позволяет игнорировать одновременность. Считается, что вместо одновременности событий появляется их недетерминированность, когда два конкурирующих события могут появиться в произвольной последовательности. Аналогичный подход также используется в ряде систем моделирования на основе сетей Петри [3].

В управлении по готовности данных порождение таких событий можно связать с моментами порождения данных. Отделение событий от данных можно рассматривать как поток управляющих сигналов, взаимодействие которых позволяет формировать различные стратегии управления вычислениями [4, 5]. На основе данного подхода в рамках концепции функционально-потокового параллельного программирования предложена модель событийного процессора [6], осуществляющего обход информационного графа функционально-потоковой параллельной программы и выполнение обработки данных на основе реагирования только на управляющие сигналы.

Использование последовательности асинхронных сигналов позволяет описать параллелизм без явных распараллеливающих схем. Предложенные в ФПМПВ асинхронные списки [1] позволяют описывать вычисления через последовательные запуски рекурсивных вызовов. Показано также, что использование асинхронных списков в этом случае обеспечивает динамическое изменение параллелизма в зависимости от отношения между темпом поступления данных и временем выполнения операций обработки данных от максимального параллелизма до последовательных вычислений. То есть, ориентация на асинхронно поступающие данные и последовательности сигналов, информирующих об их поступлении, позволяет более гибко описывать параллельные алгоритмы и в дальнейшем адаптировать их к конкретным условиям, используя только один способ описания алгоритма на основе последовательно обрабатываемых асинхронных потоков. Недостаток асинхронных списков, связанный с недетерминированностью поступления в них обрабатываемых данных предлагается исправить за счет модификации параллельных списков, расширенных функциями, поддерживающими дополнительные механизмы управления на основе сигналов о готовности данных.

Использование событийного управления вычислениями в системах с управлением на основе готовности данных было предложено в [6]. В рамках этой концепции разработан интерпретатор функционально-потоковых параллельных программ, написанных на языке программирования Пифагор [7]. Применение событийного управления позволило отделить граф управления вычислениями от информационного графа программы. При этом стало возможным изменение стратегии управления вычислениями за счет изменения управляющего графа без изменения информационного графа программы.

Дальнейшее развитие представленных работ связано с расширением семантики ФПМПВ и введением в модель и язык статической типизации данных, что обеспечивает более гибкую трансформацию в другие параллельные архитектуры. Внесенные изменения привели к созданию статически типизированной модели функционально-потоковых параллельных вычислений (СТМФППВ) и разработке на основе этой модели статически типизированного языка функционально-потокового параллельного программирования Smile [8].

2. Описание ключевых понятий модели вычислений, обеспечивающих поддержку асинхронных последовательных потоков

Предлагаемые новые понятия по сути расширяют возможности ряда программоформирующих операторов ФПМПВ. Однако использование в новой модели и языке программирования статической типизации данных вместо динамической типизации, с одной стороны, накладывает свои ограничения, но, с другой стороны, предоставляет дополнительные возможности по трансформации функционально-потоковых параллельных программ в программы для реальных архитектур.

В СТМФППВ, по сравнению с ФПМПВ, изменилась семантика контейнерных типов данных, обеспечивающих поддержку массовых операций. В частности, для поддержки анализа типов во время компиляции, вместо списка данных появился вектор (vector), все элементы которого должны быть одного предопределенного (именованного) типа. Однотипные элементы также должны быть у роя и потока. Это позволяет ввести над этими конструкциями массовые поэлементные операции, а также, наряду с ними, независимо от контейнера, выполнять функции, воспринимающие контейнерные типы как единое целое.

Использование статической типизации также привело к разделению оператора интерпретации на два разных вида: одиночный (одноаргументный) и групповой (массовый, поэлементный). Одиночный оператор интерпретации, обозначаемый в текстовом представлении, как и ранее [9], через «:» (постфиксная форма) или «^» (префиксная форма) предназначен для задания обычных функций, воспринимающих аргумент в качестве единого целого. Массовый оператор интерпретации используется для задания вычислений над каждым однотипным элементом контейнера, порождая на выходе контейнер с элементами тип которых соответствует типу результата выполняемой функции. Обозначается двойным значком «::» для постфиксной или «^^» для префиксной форм соответственно.

Использование разных обозначений позволяет однозначно применять функцию с одним и тем же именем в разных контекстах. Например, функция вычитания «-» над аргументом (10, -3), воспринимаемом как вектор, состоящий из двух целых чисел, порождает следующие значения:

```
// двуместная функция вычитания над одним аргументом
(10, -3):- => 13

// функция смены знака, массово применяемая
// к двум однотипным аргументам
(10, -3)::- => (-10, 3)
```

Разделение оператора интерпретации на массовый и одноаргументный позволяет ввести более гибкий одноаргументный набор дополнительных функций для потока и роя, обеспечивающих обработку асинхронно поступающих данных.

2.1. Организация асинхронных последовательных потоков с произвольным поступлением данных

Введенное в СТМФППВ понятие потока (stream), расширяет концепцию ранее предложенного асинхронного списка. Основная идея, связанная с асинхронным поступлением данных, сохраняется. Однако предполагается, что все элементы имеют один и тот же именованный тип, который, в свою очередь, не может являться потоком или роем. Это вполне соответствует концепциям универсальных статически типизированных языков. Поток можно рассматривать как сущность (рисунок 1), к основным характеристикам которой относятся:

- при появлении в потоке хотя бы одного готового элемента данных, он порождает сигнал, информирующий об его готовности;
- готовый элемент может быть прочитан из потока для обработки;
- если во время обработки элемента, выбранного из потока в него поступают новые элементы данных, они также могут асинхронно выбираться из потока в порядке поступления и обрабатываться параллельно;
- параллельно обрабатываемые элементы потока могут поступать после обработки в другой поток, тип которого определяется типом результата функции, при этом порядок их поступления может отличаться от первоначального в зависимости от времени обработки;
- поток можно проверить на отсутствие дальнейшего поступления данных, что позволяет завершить работу с ним.

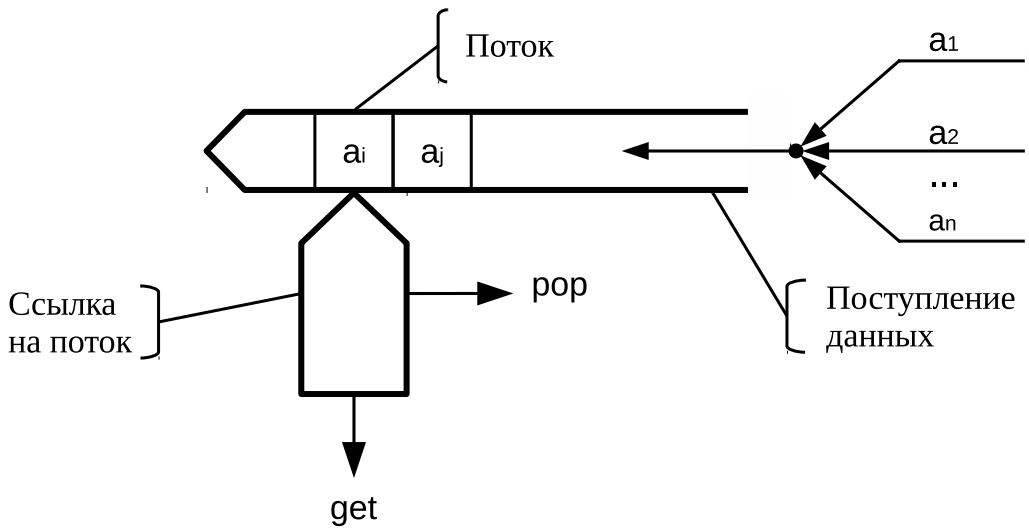

Fig. 1. General scheme of the stream

Рис. 1. Обобщенная схема потока

2.1.1. Основные операции с потоком

Описание потока в языке программирования Smile определяется следующим синтаксисом:

```
поток = имя_типа_элементов "{}".
```

Выполнение операции с потоками может осуществляться с использованием массовой операции интерпретации. В этом случае все элементы обрабатываемого потока после выполнения над ними функции поступают в формируемый на выходе поток с результатами вычислений. Например, вычисление функции `sin` над всеми элементами входного потока `X` с формированием на выходе потока `Y` на языке Smile можно записать следующим образом:

```
X::sin >> Y
```

где поток `X` предварительно описан следующим образом: `X@float{}`. Символ `@` отделяет имя используемой сущности от ее типа. Автоматически формируемый на выходе поток имеет такой же тип, что и тип результата функции `sin` и определяется ее сигнатурой:

```
sin << func float -> float
```

При этом порядок результатов в потоке, обозначенном через `Y` может отличаться от порядка поступления аргументов в поток `X`.

Принято допущение, что операции над потоком не выполняются непосредственно. Это связано с тем, что прямое обращение к потоку может привести к побочным эффектам, изменяющим его состояние, что не позволит корректно взаимодействовать с потоком другим операциям, выполняемым параллельно. Вместо этого одноаргументный оператор интерпретации обращается к потоку через ссылку. Наличие нескольких ссылок на один поток, играющих по сути роль итераторов, позволяет обрабатывать его различным образом в разных частях программы. Синтаксис ссылки на поток в Smile имеет следующий вид:

```
ссылка_на_поток = имя_типа_элементов "{*}".
```

Потоки через ссылки могут передаваться в функции в качестве параметров. Функция вычисления синуса для всех элементов потока в Smile будет выглядеть следующим образом:

```
sinStream << funcdef X@float{*} -> float{*} {
    X::sin:return
}
```

То есть, передача через ссылку допускает массовое выполнение функций. Следует также отметить, что при использовании потока в качестве аргумента массового оператора интерпретации внутри последнего автоматически создается скрытая локальная ссылка на этот поток, что позволяет избежать побочных эффектов.

В большинстве случаев использования массовых операций над потоком недостаточно для гибкого асинхронного программирования, когда требуется поэлементная обработка или объединение данных, поступающих из нескольких потоков. Для этого необходимо применять дополнительные конструкции, во многом аналогичные итераторам. Эти конструкции задаются специальными одноаргументными функциями над потоком. В частности, перед непосредственным доступом к элементу потока необходимо предварительно проверить, что поток еще порождает элементы (по аналогии с проверкой признака конца файла). Это обуславливается тем, что количество элементов, которые порождает поток, может быть заранее неизвестно. Проверка потока на то, что данные в нем

еще порождаются осуществляется функцией is, имеющей следующую сигнатуру:

```
is << func any{*} -> bool
```

где any — ключевое слово, обозначающее любой тип. Функция возвращает значение true, если поток еще может формировать данные или уже содержит их. В противном случае возвращается false.

Для получения данных из потока используется функция get, которая читает элемент потока, стоящий в его очереди первым. Если такой элемент еще не сформировался, функция get ожидает его поступления. При наличии в очереди потока нескольких элементов выбирается только один. При попытке выполнить эту функцию для уже завершенного потока формируется ошибка, ведущая к прерыванию функции. Функция имеет следующую сигнатуру:

```
get << func any{*} -> any
```

Перед тем как прочитать следующий элемент из входного потока необходимо убрать из ссылки уже прочитанный элемент. Для этого используется функция pop. При попытке выполнить эту функцию для уже завершенного потока формируется ошибка, ведущая к прерыванию функции. Функция имеет следующую сигнатуру:

```
pop << func any{*} -> any{*}
```

То есть, она возвращает новую ссылку на тот же поток, но уже без обработанного элемента (этот элемент уже недоступен через возвращаемую ссылку).

В качестве примера использования одноаргументных функций можно рассмотреть нахождение суммы элементов поступающих во входной поток:

```
sum << func X@float{*} -> float {
    if << X:is;
    if^({(X:get,X:pop:sum):+}, 0):return
}
```

Проверка X:is порождает булевское значение true/false, которое используется оператором интерпретации в качестве селектора. При истинном значении выбирается первый элемент кортежа, запускающий левую рекурсию для функции sum. Значение false формируется, когда поток завершен. В этом случае возвращается значение 0. При обратном ходе рекурсивного процесса осуществляется суммирование элементов.

2.1.2. Занесение информации в поток из различных источников

В представленном выше примере суммирования элементов потока, по сути, реализована последовательная рекурсия, так как при обратном ходе накопление суммы осуществляется путем сложения очередного элемента потока с накопленным промежуточным значением. В работе [1] показано, что асинхронный список через последовательные рекурсивные вызовы позволяет реализовать суммирование параллелизм которого, в зависимости от временных соотношений между интенсивностью поступления данных и скоростью их обработки, может достигать максимального, эквивалентного каскадной свертке. Использование потоков в языке программирования Smile позволяет написать аналогичную функцию. При этом наличие возможности создавать хранилища обеспечивает занесение в поток не только исходных данных, но и промежуточных результатов.

Соответствующая функция суммирования значений, поступающих в поток, выглядит следующим образом:

```
// Асинхронное суммирование элементов потока
// с внесением в него результатов промежуточных вычислений
sum << func X@float{*} -> float {
    // Проверка, что данные в поток еще могут поступить
    notEmpty << X:is;
    notEmpty^(
        // Есть хотя бы один элемент
        {block{
            // Элемент выбирается из потока
            a << X:get;
            // Формируется ссылка на следующую позицию
            Y << X:pop;
            // и делается проверка на наличие следующего элемента
            notEmptySecond << Y:is;
            notEmptySecond^(
                {block{
                    // При наличии второго элемента можно его сложить с первым
                    // И через любую доступную ссылку переслать в поток
                    (a, Y:get):+ -> Y;
                    // Также создать новую ссылку и без второго элемента
                    // рекурсивно продолжив вычисления
                    Y:pop:sum:break
                },
                // В противном случае в потоке только один элемент,
                // значение которого и является суммой
                a
            ):break
        }},
        // При отсутствии данных возвращается 0
        0.0
    ):return
}
```

В данной ситуации функция, при наличии хотя бы двух элементов в потоке, суммирует их. Полученная сумма через ссылку пересылается в этот же поток. Процесс рекурсивно повторяется для вновь поступающих элементов, чередующихся с промежуточными вычислениями сумм до момента, когда в потоке останется только одна величина, которая и является окончательной суммой.

2.1.3. Недетерминированность поведения потока при выполнении асинхронных вычислений

Использование потоков позволяет организовывать асинхронные вычисления с динамически изменяемым параллелизмом, зависящим от временных соотношений между интервалами поступления данных в поток и скоростью их обработки функциями, взаимодействующими с потоком. Однако высокая вероятность того, что порядок поступления аргументов не будет совпадать с порядком получения результатов на выходе, не позволяет во многих случаях организовать детерминированные и предсказуемые вычисления. В качестве такого примера можно рассмотреть вычисление

массива данных, поступающих из входного потока, а после обработки направляющихся в выходной поток. Пусть для вычислений используется формула:

```
y[i] = sin(x[i])*sin(x[i])+cos(x[i])*cos(x[i])
```

При использовании потоков в качестве промежуточных хранилищ результатов без особых сложностей организуются конвейерные вычисления (при соответствующих временных соотношениях). Однако в связи с возможностью различного времени выполнения функций над элементами потоков, корректные последовательности значений в результирующем потоке могут быть не получены.

Эта ситуация может быть описана следующим кодом на языке Smile:

```
SumSin2Cos2Stream << func X@float{*} -> float{*} {
    result@float{};
    (X, result):GetStreamResult >> ok;
    result:ok:return
}
```

где:

```
GetStreamResult << func (arg@float{*}, result@float{*})->signal {
    // Проверка потока на возможное поступление данных
    if << arg:is;
    if^(
        // Занесение результата в выходной поток
        // после добавления в него данных
        {block {
           x << arg:get;    // Получение элемента из потока
           s << x:sin; Sin2 << (s,s):*; // Синус в квадрате
           c << x:cos; Cos2 << (c,c):*; // Косинус в квадрате
           // Вычисление текущего значения с передачей в выходной поток
           (Sin2,Cos2):+ -> result;
           // Убирается обработанный элемент из потока
           // и переход к обработке следующего элемента
           (arg:pop, result):GetStreamResult:break}
        },
        // Сигнал без заполнения результата,
        // если данные в поток больше не поступают
        !
    ):return
}
```

Основная функция `SumSin2Cos2Stream` получает данные из входного потока через ссылку `X`. Результат вычислений через ссылку на поток возвращается из функции. Сам поток реализован внутри функции через хранилище `result`, а накопление в нем результатов вычислений происходит в функции `GetStreamResult`, в которую он передается в качестве параметра.

Функция `GetStreamResult` производит основные вычисления для первого текущего элемента, поступившего во входной поток. Полученное значение суммы передается, используя обращение к элементу выходного потока (обозначено через ->). Одновременно с этим происходит рекурсивный вызов функции `GetStreamResult`, в которую ссылка на входной поток передается уже без учета

первого аргумента. Блок `block` используется для локализации группы операторов, из которой только один возвращает результат посредством выполнения функции `break`. Данные в блок поступают через имена, описанные вне его.

При передаче результатов вычислений в новый поток порядок поступления элементов может изменяться относительно первоначального потока, что ведет к появлению недетерминированности вычислений и некорректному результату. Пример показывает, что необходимо расширить модель вычислений конструкциями, обеспечивающими сохранение порядка следования данных и при этом поддерживающими асинхронные взаимодействия.

2.2. Организация упорядоченных данных с сохранением порядка следования

Для сохранения порядка следования при обработке данных необходимо использовать контейнерные типы, обеспечивающие асинхронное формирование отдельных элементов. В ФПМПВ такой сущностью является параллельный список. Однако он поддерживает выполнение только массовых операций над его элементами и не допускает обработки самого списка как единого аргумента. В СТМФППВ вводятся расширения, обеспечивающие поддержку необходимой функциональности. Вместо параллельного списка используется рой, который, наряду с массовыми операциями, как и поток, допускает свое использование в качестве единственного аргумента (рисунок 2).

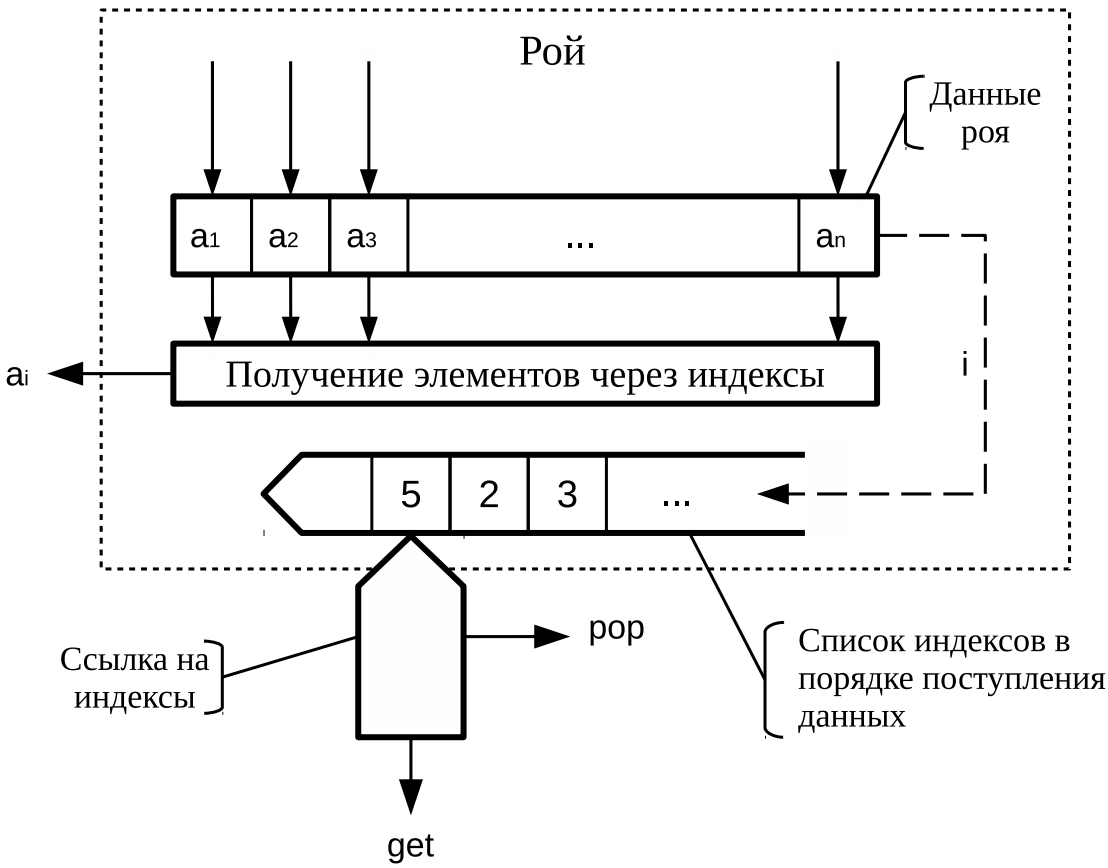

Fig. 2. General scheme of the swarm and its reference

Рис. 2. Обобщенная схема роя и ссылки на него

Спецификой предлагаемой СТМФППВ и разрабатываемой на ее основе статически типизированного языка функционально-потокового параллельного программирования Smile является

наличие информации о типах во время компиляции. Это ведет к изменению алгебры эквивалентных преобразований и семантики многих базовых операций, ориентированных не на интерпретацию исходной программы, а на генерацию кода для целевых архитектур. В частности, запрещается непосредственная вложенность роев, что облегчает анализ аргументов оператора интерпретации во время компиляции и позволяет определить, является ли функция массовой над всеми элементами роя или это функция над всем роем. Ряд преобразований роя для использования в массовых операциях может происходить уже во время компиляции. Для представления роя используется список из элементов, заключенных в квадратные скобки.

Передача роя в функции и возврат их осуществляется, как и для потоков, через ссылки. Синтаксис ссылки на рой в Smile имеет следующий вид:

```
ссылка_на_рой = имя_типа_элементов "[*]".
```

Используя их можно написать следующий вариант функции одновременного упорядоченного вычисления синуса для всех элементов роя:

```
sinSwarm << funcdef X@float[*]->float[*] {
    X::sin:return
}
```

В этой ситуации непосредственное использование роевых функций позволяет избавиться от дополнительных преобразований и синхронизации данных как внутри создаваемых функций, так и при их использовании:

```
[0.10, 2.1, 0.33, 1.43]:sinSwarm => [0.0998, 0.8632, 0.324, 0.9901]
```

Компилятор, анализируя тип аргумента функции `sinSwarm`, без проблем может распознать, что она принимает весь рой, а не применяется к каждому из его элементов.

2.2.1. Использование роя для упорядоченной асинхронной обработки потока

В отличие от потоков каждый даже частично сформированный рой имеет предопределенный размер. Его можно вычислить в любой момент, используя функцию `size`, сигнатура которой описывается следующим образом:

```
size << func any[*] -> int
```

Например:

```
[10,21,33,43]:size => 4
```

Нумерация элементов роя, как и вектора, начинается с единицы. Сами элементы роя формируются асинхронно. При этом поступление каждого из них сопровождается выдачей в связанный с ним оператор интерпретации сигнала, информирующего о формировании очередного значения по определенному индексу. Эти индексы можно упорядочить в порядке поступления и, следовательно, осуществить последовательную выборку отдельных элементов по ним. То есть, можно организовать итератор, делающий обход элементов роя по мере их появления. В отличие от обхода элементов потока, в которых обращение идет непосредственно за созданными элементами, в рое ключевую роль играет получение значения индекса. Для его получения предлагается использовать функцию `get`, которая имеет для роя следующую сигнатуру:

```
get << func any[*] -> int
```

То есть, возвращается индекс элемента, поступившего в рой первым.

Для перехода к следующему индексу, используется функция pop. Она возвращает ссылку на тот же рой, но уже без указания убранного индекса:

```
pop << func any[*] -> any[*]
```

Таким образом можно перебрать все элементы роя в порядке их формирования. В случае, когда через ссылку произойдет перебор всех элементов роя (в порядке их поступления), функция get возвращает нулевое значение индекса, которое, по сути, и определяет завершение обхода.

Помимо этого рой, как и поток, может использоваться для последовательной обработки асинхронно поступающих данных с сохранением порядка следования элементов на выходе. Это позволяет переписать функцию нахождения сумм квадратов синусов и косинусов роя таким образом, что она обеспечивает правильную последовательность результатов на выходе:

```
SumSin2Cos2Swarm << func X@float[*]->float[*] {
    L << X:size;
    result@float[L];
    (X, result):GetSwarmResult >> ok;
    result:ok:return
}
```

Для накопления данных функция использует дополнительное роевое хранилище result, которое заполняется с использованием принципа единственного присваивания. То есть формируется такой код, который позволяет записать данные по одному и тому же индексу не более чем один раз. При нарушении этого правила происходит прерывание выполнения программы. Через ссылку хранилище передается в функцию GetSwarmResult, обеспечивающую его заполнение, после чего полученное значение возвращается из функции SumSin2Cos2Swarm. Само вычисление осуществляется в функции GetSwarmResult:

```
GetSwarmResult << func (X@float[*],Y@float[*])->signal {
    i << X:get; // Получение индекса элемента из X
    if << (i,0):!=; // Проверка наличия элементов
    if^(
        {block {
           s << X:i:sin; Sin2 << (s,s):*; // Синус в квадрате
           c << X:i:cos; Cos2 << (c,c):*; // Косинус в квадрате
           // Вычисление текущего значения с передачей в выходной рой
           // по полученному индексу
           (Sin2,Cos2):+ -> Y[i];
           // Убирается обработанный индекс из ссылки на рой
           // и переход к обработке следующего элемента
           (X:{i:signal}:pop, Y):GetSwarmResult:break}
        },  // Занесение результата во второй рой
        // Сигнал, формируемый при завершении вычислений без заполнения,
        // если значение индекса = 0
        !
       ):return
}
```

Первоначально в данной функции вычисляется индекс первого поступившего в рой X элемента. Если значение не равно 0, то получен очередной индекс, который непосредственно используется для выборки из роя i-го элемента, после чего над ним производится вычисление суммы квадратов синуса и косинуса. Полученное значение заносится через ссылку Y на i-е место. Вычисления рекурсивно повторяются до полного заполнения хранилища result, передаваемого в данную функцию через ссылку Y.

2.2.2. Прямое обращение к элементам роя

Наряду с обработкой элементов роя в порядке их поступления возможен и непосредственный доступ по индексу. В этом случае, если элемент еще не поступил, происходит его ожидание. Во время ожидания можно инициировать выборку других элементов, используя для этого параллельно выполняемые рекурсивные вызовы. Недостатком такого подхода является возможность появления множества параллельных ветвей, ожидающих поступления данных. Однако при обработке данных, поступающих из нескольких роев данный подход облегчает синхронизацию вычислений. Сигнатура функции доступа по индексу имеет следующий вид:

```
base_function<целое> << func any[*] -> any
```

В данном случае в качестве функции используется целое число в диапазоне от 1 до размера роя. Если число не попадает в этот диапазон, то возникает прерывание в работе программы.

В качестве примера рассмотрим скалярное перемножение данных, поступающих в рои. Функция ScalMultSignal осуществляет вычисления, принимая в качестве аргументов два роя через ссылки X и Y. Помимо этого она получает ссылку R на рой, собирающий результаты, а также число, определяющее количество элементов в роях. Последнее используется в качестве индекса для обращения к элементам.

```
//  функция, непосредственно выполняющая скалярное умножение роев
ScalMultSignal << func (X@float[*], Y@float[*], R@float[*], L@int)->signal {
    if << (L,0):!=;
    if^(
        {block{
            (X:L, Y:L):* -> R[L];
            (X, Y, R, L:--):ScalMultSignal:break
        }},
        // Завершение перебора
        !
    ):return
}
```

Перемножение элементов с одинаковыми индексами осуществляется пока передаваемое значение индекса не обнулится функцией «--», формирующей значение на единицу меньше предыдущего. Рекурсивный вызов осуществляется сразу же после раскрытия задержки, охватывающей блок, независимо от того, будет или нет выполнена операция умножения.

Окончательная функция предоставляет интерфейс для взаимодействия с другими функциями:

```
// Функция, используемая для перемножения роев.
// Предполагается, что размер роев одинаков
ScalMult << func (X@float[*], Y@float[*]) -> float[*] {
    L << X:size;
    result@float[L]; // Хранилище результатов
    ok << (X, Y, R, L):ScalMultSignal;
    result:ok:return
}
```

2.2.3. Конвейеризация асинхронных потоковых вычислений

Организация вычислений функций на основе передачи между ними потоков и роев позволяет организовать взаимодействия, обеспечивающие совмещение вычислений в функциях, взаимосвязанных между собой. В качестве примера можно рассмотреть функцию векторного произведения с использованием функций скалярного произведения двух векторов `ScalMult` и нахождения суммы элементов потока `sum`:

```
VecMult << func (X@float[*], Y@float[*]) -> float {
    (X, Y):ScalMult:stream:sum:return
}
```

Данная функция принимает два роя, над которыми осуществляется выполнение скалярного произведения. По мере того, как на выходе функции `ScalMult` формируются результаты перемножения отдельных пар элементов, они поступают в поток, связанный со входом функции `sum`. Конвейеризация в данном случае формируется автоматически в зависимости от темпа поступления исходных данных и скорости выполнения операций внутри функции `VecMult`.

Заключение

Рассмотренные в работе механизмы обеспечивают поддержку нового подхода к разработке параллельных программ, позволяя описывать параллелизм с использованием асинхронных последовательно порождаемых потоков с управлением по готовности данных. При этом характеристики параллелизма зависят от темпа поступления данных. Совместное использование функций, реализованных с применением возможностей данной модели вычислений обеспечивает поддержку конвейерных вычислений. Показано, что предложенные методы описания функционально-потоковых параллельных вычислений могут быть реализованы в статически типизированном языке функционально-потокового параллельного программирования.

References

[1] A. I. Legalov, “The Usage of Asynchronous Lists within the Dataflow Model of Computations”, in *The Third Siberian School Seminar on Parallel Computations*, In Russian, 2006, pp. 113–120.

[2] C. A. R. Hoar, *Communicating Sequential Processes*, 8. Communications of the ACM, 1978, vol. 21, pp. 666–677.

[3] M. Diaz, *Petri Nets: Fundamental Models, Verification and Applications*. UK: ISTE Ltd, 2009.

[4] A. I. Legalov, “About Computation Control in Parallel System and Programming Languages”, *Nauchiy Vestnik NGTU*, vol. 18, no. 3, pp. 63–72, 2004, In Russian.

[5] A. I. Legalov, V. S. Vasiliev, and I. V. Matkovsky, “Changing Computing Management Strategies for Architecture-Independent Parallel Programming”, in *Proceedings of the XIX All-Russian Scientific Conference “Scientific Service on the Internet”*, In Russian, 2017, pp. 341–350.

[6] A. V. Redkin and A. I. Legalov, "Event Based Control of Computations for Functional Dataflow Programming", *Scientific Bulletin of Novosibirsk State Technical University*, vol. 32, no. 3, pp. 111–120, 2008, In Russian.

[7] A. I. Legalov, A. V. Redkin, and I. V. Matkovsky, "Data Driven Functional Parallel Programming with Data Coming Asynchronously", in *Parallel Computing Technologiws (PCT'2009)*, In Russian, 2009, pp. 573–578.

[8] A. I. Legalov, I. A. Legalov, and I. V. Matkovsky, "Specifics of Semantics of a Statically Typed Language of Functional and Dataflow Parallel Programming", in *Scientific Conference "Scientific Service on the Internet"*, 2019, pp. 489–500.

[9] A. I. Legalov, "Functional Language for Architecture-Independent Programming", *Computation technologies*, vol. 10, no. 1, pp. 71–89, 2005, In Russian.