ИЗВЕСТИЯ РАН. ТЕОРИЯ И СИСТЕМЫ УПРАВЛЕНИЯ, 2014, № 3, с. 46–60

КОМПЬЮТЕРНЫЕ МЕТОДЫ

УДК 519.687

РЕАЛИЗАЦИЯ ЯЗЫКА ФУНКЦИОНАЛЬНОГО ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ FPTL НА МНОГОЯДЕРНЫХ КОМПЬЮТЕРАХ*

© 2014 г. В. П. Кутепов, П. Н. Шамаль

Москва, МЭИ (национальный исследовательский ун-т)

Поступила в редакцию 18.07.13 г., после доработки 29.01.14 г.

Дается описание функционального языка программирования, поддерживающего неявное распараллеливание программ. Язык основан на четырех операциях композиции, три из которых позволяют выполнить распараллеливание вычислений. Рассмотрено схемное представление функциональных программ, позволяющее применить динамический алгоритм распараллеливания. Реализованные алгоритмы позволяют динамически распределять нагрузку между процессорами и контролировать зернистость распараллеливания. Приведены экспериментальные данные эффективности реализованной системы на примерах типовых задач.

DOI: 10.7868/S000233881403010X

Введение. Понятие функции является фундаментальным как в математике, так и в программировании. Поэтому всегда был велик интерес к созданию языков функционального программирования. Задачи вычислительной математики, сортировки данных, трансляции языков и многие другие имеют функциональную природу, и методы их решения могут быть достаточно просто описаны на функциональных языках. Кроме того, общепринятая в математике композиционная форма задания функций с использованием привычных операций подстановки, суперпозиции и рекурсии позволяет строить простые и эффективные алгоритмы для реализации параллельного вычисления значений функций. Язык FPTL (Functional Parallel Typified Language) создавался на этой основе [1–4]. Другое направление создания языков функционального программирования базируется на теоретической модели λ-абстракций Черча [5], которая была изначально нацелена на решения проблемы контекстуального разделения переменных с одними и теми же именами, используемыми в разных местах текстов, в частности математических выражений. Операции λ-связывания переменных и подстановки λ-выражений без явного использования рекурсии (она представляется неявно) уже достаточны для того, чтобы этими средствами можно было выразить любую вычислимую функцию. Простые правила редукции λ-выражений, всегда приводящие к одному и тому же результату, если он существует, дают основания рассматривать процессы приведения λ-выражений к нормальной форме как функциональные. Эта особенность λ-исчисления, а также способность единообразно путем применения операции λ-связывания переменных в выражениях строить таким косвенным путем функции любого порядка, по-видимому, сыграли определяющую роль в создании серии языков λ-основанного функционального программирования. LISP [6], ML [7], Haskell [8], F# [9] – примеры известных языков этого направления. Однако реальная практика заставила использовать рекурсию для упрощения задания функций, вводить систему базисных типов и функций для построения других функций (в LISP, например, введен набор базисных функций для выполнения операций над списками: CAR, CDR, CONS, APPEND, EQ и др.).

Сравнивая языки функционального программирования обоих направлений, следует отметить, что на стороне λ-основанных языков более универсальные средства задания функций высших порядков. Минусы этого типа функциональных языков – в их часто неоправданно усложненной форме программы, функциональная природа которой выражается неявно и является следствием уже выше отмеченной однозначности ее выполнения. Проблема автоматического распараллеливания функциональных программ на этих языках трудно разрешима как до выполнения программы, так и в процессе ее выполнения. Поэтому распараллеливание программ на

* Работа выполнена при финансовой поддержке РФФИ (проект № 13-07-00810).

этих языках выполняет сам программист, используя специально вводимые в язык примитивы задания параллелизма [9].

Язык FPTL [3, 4] построен на композиционной основе, он достаточно прост в освоении, поскольку сохраняет общепринятую форму задания функций посредством использования четырех простых операций композиции функций и рекурсивных определений, задаваемых в виде систем функциональных уравнений. Он строго типизирован и позволяет однообразно, как и функции, определять абстрактные типы данных в виде систем уравнений. FPTL самодостаточен в том смысле, что не требует введения, как в LISP, множества базисных функций; они непосредственно извлекаются из определения типов данных аргументов и результатов функций, как конструкторы этих данных и обратные к ним деструкторы. Кроме того, в FPTL в качестве базисных функций можно использовать арифметические и другие уже реализованные в компьютере функции. Но самое важное с точки зрения построения параллельных программ состоит в том, что для задания параллелизма в них не требуется использование каких-либо специальных примитивов и он автоматически легко распознается при выполнении программы. Это является следствием того, что три из четырех операций композиции функций являются параллельными. Эти особенности FPTL создали основу для разработки качественных параллельных программ, не прибегая к применению других средств задания параллелизма в программе [3].

В описываемой в статье реализации FPTL на многоядерных компьютерных системах использовались операционные средства MULTITHREADING для управления параллельными процессами. Теоретическая основа языка FPTL подробно описана в [1–4], в данной статье будут рассмотрены только те внесенные в язык изменения, которые были направлены на его более простую и эффективную параллельную реализацию на многоядерных компьютерных системах. Далее кратко опишем теоретическую модель FPTL и модель параллельного вычисления значений функций, а затем реализацию языка и экспериментальные данные эффективности параллельного выполнения FPTL-программ на многоядерных компьютерах.

В разд. 1 статьи приведена основа языка FPTL, в разд. 2 – реализация FPTL на многоядерных компьютерах, в разд. 3 – результаты экспериментального исследования эффективности параллельного выполнения FPTL-программ. В приложение вынесены исходные тексты FPTL-программ, используемых в экспериментах.

1. Теоретическая основа языка FPTL. Язык FPTL, как было отмечено во Введении, создавался как прообраз общепринятой в математической практике формы задания функций в общем случае в виде систем рекурсивных функциональных уравнений, в которых используется операция подстановки функций вместо функциональных переменных и условный оператор.

В FPTL введены четыре простые бинарные композиции функций, которые позволяют легко отразить композиционное представление функции и, что более важно в данном контексте, явно отразить параллелизм.

Функции в FPTL рассматриваются, как типизированные $t'_1 \times t'_2 \times \ldots \times t'_m \to t''_1 \times t''_2 \times \ldots \times t''_n$ (m, n)-арные соответствия ($m \geq 0$, $n \geq 0$) между кортежами данных, где t'_i, $i = \overline{1,m}$ и t''_j, $j = \overline{1,n}$ – типы элементов входного и выходного кортежей, m – длина кортежа на входе функции, n – на выходе. Функции арности (0, 1) рассматриваются в FPTL как константы. Для m и n, равных 0, имеем кортеж нулевой длины, обозначаемый λ, со свойствами $\lambda\alpha = \alpha\lambda = \alpha$, где α – произвольный кортеж. Кортеж данных в языке представляется в виде последовательной записи его элементов.

В FPTL в отличие от общепринятой формы задания функций с явным указанием ее аргументов (так называемая форма задания общего значения функции) строго различается собственно функция как отображение одного множества в другое и ее аппликация к конкретным данным. Роль переменных в задании функций в FPTL выполняют функции выбора необходимого элемента из кортежа данных. Формально функция выбора аргумента, обозначаемая $I(i, m)$ (в языке – просто $[i]$), при применении к кортежу произвольного типа данных длины m ($m > 0$) выбирает его i-й элемент, $i = \overline{0,m}$, $m > 0$. Для $i = 0$ выбираемое значение есть λ. Функции в FPTL являются в общем случае частичными, причем неопределенное значение функции может быть выражено либо как неограниченный процесс вычисления ее значения, либо как специальное вычисленное неопределенное значение, обозначаемое ω, со свойствами $\omega\alpha = \alpha\omega = \omega$ для любого кортежа α.

Формально функции определяются как системы функциональных уравнений $F_i = \tau_i$, $i = \overline{1,n}$, где τ_i – функциональные термы, построенные из заданных (базисных) функций и функциональных переменных F_i путем применения четырех операций композиции функций: $\to$, $+$, $*$, $\bullet$. Для функций и функциональных переменных задана их арность, а для базисных функций – также их

тип. Тип функциональных переменных однозначно определяется из задания типов базисных функций и правил вывода типов для функций, построенных путем применения операций композиции.

Пусть $f^{(m,n)}$ – (m, n)-арная функция, $f(\alpha)$ – результат ее применения к кортежу α, f_1, f_2 – заданные функции, α, β, γ – обозначение кортежей. Синтаксис и семантика операций композиции определяются следующим образом.

1. Последовательная композиция (•):

$$f^{(m,n)} \stackrel{\text{def}}{=} f_1^{(m,k)} \cdot f_2^{(k,n)};$$

$$f(\alpha) \stackrel{\text{def}}{=} f_2(f_1(\alpha)),$$

где $\alpha = \alpha_1\, \alpha_2\, \alpha_3 \ldots \alpha_m$ – кортеж данных типа $t_1 \times t_2 \times \ldots \times t_m$; t_i – тип данных i-го аргумента функции. Здесь и далее сначала задается синтаксис операции композиции, а затем ее семантика, определяемая через применение функций к кортежу данных. Предполагается, что типы кортежа значений функции f_1 и кортежа аргументов функции f_2 одинаковы.

Заметим, что в FPTL используется префиксная форма записи операции последовательной композиции, задающая последовательный характер вычисления значений функций f_1 и f_2 и эквивалентная последовательному характеру задания следования операторов при выполнении последовательных программ.

2. Операция конкатенации кортежей значений функций (*):

$$f^{(m,n_1+n_2)} \stackrel{\text{def}}{=} f_1^{(m,n_1)} * f_2^{(m,n_2)};$$

$$f(\alpha) \stackrel{\text{def}}{=} f_1(\alpha) f_2(\alpha).$$

Предполагается, что типы аргументов функций f_1 и f_2 одинаковы.

3. Операция условной композиции:

$$f^{(m,n)} \stackrel{\text{def}}{=} f_1^{(m,k)} \to f_2^{(m,n)};$$

$$f(\alpha) \stackrel{\text{def}}{=} \begin{cases} f_2(\alpha), & \text{если} \quad f_1(\alpha) \text{ отлично от значения “ложь” или } \omega, \\ \omega \text{ иначе.} \end{cases}$$

Предполагается, что типы кортежей аргументов функций f_1 и f_2 одинаковы.

4. Операция объединения (графиков) ортогональных функций:

$$f^{(m,n)} \stackrel{\text{def}}{=} f_1^{(m,n)} + f_2^{(m,n)};$$

$$f(\alpha) = \begin{cases} f_1(\alpha), \text{ если значение } f_1(\alpha) \text{ определено,} \\ f_2(\alpha), \text{ если значение } f_2(\alpha) \text{ определено.} \end{cases}$$

Предполагается, что типы аргументов и значений функций f_1 и f_2 одинаковы. Напомним, что функции f_1 и f_2 считаются ортогональными, если для всякого кортежа данных α определена не более чем одна из них. Операция объединения ортогональных функций была введена с целью представления параллельных функций (известный пример – функция голосования в телефонии) [4].

Все операции композиции являются ассоциативными, а операция + выступает как коммутативная. Приоритет операций композиции определяется следующем образом (в порядке возрастания): +, $\to$, *, • .

Как уже было отмечено, термы в задании функций в виде систем функциональных уравнений представляют собой композиции, построенные из базисных функций и функциональных переменных путем применения операций композиции. Предполагается, что арности и типы терма и определяемой им функциональной переменной в системе функциональных уравнений одинаковы. Функциональные переменные выполняют двойную роль при задании функции (построении функциональной программы): одни из них появляются как необходимые элементы при задании рекурсивных функций, другие определяются далее (в следующем уравнении уточняемой функ-

ции), позволяя просто реализовать пошаговую разработку функциональной программы по технологии проектирования “сверху-вниз”.

Абстрактные типы данных в FPTL задаются по аналогии с функциями в виде систем реляционных уравнений. В FPTL можно использовать встроенные типы: *bool*, *real, int*, *string* и др. вместе с определенными в конкретной компьютерной системе операциями над ними.

Для определения новых типов данных используются конструкторы и деструкторы (обратные конструкторам функции), т.е. те же операции композиции, которые применяются для задания функций, за исключением того факта, что операция + трактуется как операция объединения двух множеств данных. Используемые в определении абстрактного типа данных (системе реляционных уравнений) конструкторы и деструкторы выполняют роль базисных функций. Как доказано в [1, 2], их достаточно для того, чтобы в FPTL можно было представить любую вычислимую функцию над этими данными. Также в языке имеются конструкторы констант, позволяющие создавать данные встроенных типов.

Приведем пример определения в FPTL абстрактного типа данных (списка натуральных чисел):

```
data ListOfNat {
    Nat = c_null + Nat.c_succ;
    ListOfNat = c_nil + (Nat * ListOfNat).c_cons;
}.
```

Здесь функции-конструкторы *c_null, c_succ, c_nil* и *c_cons* имеют арности (0,1), (1,1), (0,1) и (2,1) соответственно и следующие типы: $\{\lambda\}\to$Nat, Nat $\to$ Nat, $\{\lambda\}\to$ ListOfNat, ListOfNat $\to$ ListOfNat.

Обратные к ним функции (деструкторы), обозначаемые как *~c_null, ~c_succ, ~c_nil, ~c_cons*, автоматически извлекаются из определения типа и имеют следующую интерпретацию:

$$\sim c\_null(x) = \begin{cases}\lambda, \text{ если } x = 0,\\ \omega \text{ в противном случае;}\end{cases}$$

$$\sim c\_succ(x) = \begin{cases}y, \text{ если } x = c\_succ(y),\\ \omega \text{ в противном случае;}\end{cases}$$

$$\sim c\_nil(x) = \begin{cases}\lambda, \text{ если } x = c\_nil,\\ \omega \text{ в противном случае;}\end{cases}$$

$$\sim c\_cons(x) = \begin{cases}y, \text{ если } x = c\_cons(y),\\ \omega \text{ в противном случае.}\end{cases}$$

Приведем пример функции-предиката, которая проверяет принадлежность кортежа типу данных ListOfNat:

```
IsListOfNat = ~c_nil + ~c_cons.(IsNat * IsListOfNat);
IsNat = ~c_null + ~c_succ.IsNat.
```

Функция IsListOfNat определена на любом кортеже данных, принадлежащих ListOfNat, и ее значением является λ, что может трактоваться как “истина”. Для других отличных от ListOfNat данных в качестве результата применения IsListOfNat будет неопределенное значение ω, которое может также трактоваться как “ложь”.

Пример программы вычисления длины списка приведен ниже.

```
data ListOfNat {
        Nat = c_null + Nat.c_succ;
        ListOfNat = c_nil + (Nat * ListOfNat).c_cons;
}
scheme Length {
    Length = ~c_nil -> c_null, ~c_cons.[2].Length.c_succ;
}
application
    list = (c_null.c_succ * (c_nil * c_null).c_cons).c_cons;
    % Length(list)
```

Ключевое слово scheme используется в языке FPTL для явного выделения в программе системы рекурсивных уравнений, определяющих функцию, которая следует за этим описанием.

В FPTL-программах можно определять параметризованные функции и типы данных. Например, следующий абстрактный тип данных List при использовании параметра ‘t (в FPTL штрих добавляется к обозначению такого параметра) будет иметь вид:

```
data List[‘t] {
        List= c_nil + (‘t * List[‘t]).c_cons;
}.
```

Присвоив параметру ‘t значение Nat (тип Nat), мы получаем ранее определенный абстрактный тип ListOfNat.

Для использования в FPTL-программах функций и данных, которые вызываются в программе из других функциональных программ и библиотек, применяется конструкция:

import *“имя функции”* **from** *“имя внешнего модуля”*.

Здесь указывается имя внешнего по отношению к FPTL-программе модуля, из которого импортируется требуемая функция.

В Приложении приведены примеры FPTL-программ “типичных” задач, которые использовались в качестве примеров для экспериментальной проверки эффективности созданной системы управления параллельным выполнением FPTL-программ на многоядерных компьютерах.

В заключение этого параграфа отметим, что язык FPTL создавался с осознанным выбором минимального, легко осваиваемого программистом и привычного для математической функциональной нотации набора средств, позволяющих, не прибегая к введению дополнительных и обычно нарушающих общую архитектуру языка примитивов, отражать и “регулировать” параллелизм в программе. Только одна операция последовательной композиции в FPTL требует последовательного вычисления значений функций, к которым она применяется. Три остальные операции композиции являются параллельными и явно указывают, значения каких функций могут вычисляться одновременно.

2. Реализация языка FPTL на многоядерных компьютерах. Одна из первых попыток реализации языка FPTL в режиме интерпретации была предпринята в рамках работы по созданию системы выполнения FPTL-программ для кластеров [3, 4]. В качестве языка реализации использовался язык Java. При этом была использована модель параллельного выполнения программ, основанная на свертывании и развертывании деревьев, представляющих ветви параллельных процессов [4]. Данная модель выполнения оказалась неэффективной, поскольку работа с деревьями вносила большую долю накладных расходов, что существенно замедляло вычислительный процесс в целом. Кроме того, распараллеливание в данной реализации осуществлялось на уровне элементарных функций, а такой мелкозернистый параллелизм эффективно реализовать даже на многоядерных компьютерах с общей памятью практически невозможно. Поэтому в новой реализации FPTL пришлось использовать специальные методы планирования динамически порождаемых процессов при выполнении программы и управления их сложностью (зернистостью). В частности, в качестве параллельно выполняемых процессов в этой реализации рассматриваются вызовы рекурсивных функций, вычисление значений которых, как показывает практика, представляет более сложный по времени процесс, чем вычисление значений базисных функций.

С целью повышения эффективности (сокращения времени) выполнения параллельных программ в данной реализации FPTL на многоядерных компьютерах был введен целый ряд механизмов, направленных на уменьшение накладных расходов при работе с общими очередями порождаемых процессов, требующими использования средств синхронизации. Очевидно, что реализация FPTL в режиме интерпретации будет проигрывать компиляции выполняемой программы, что особенно заметно для численных алгоритмов, в которых не используются абстрактные типы данных, операции над которыми не имеют прямой компьютерной реализации. Однако интерпретация имеет свои плюсы, существенно упрощая отладку программ и их модификацию.

В описываемой в данной статье реализации FPTL на многоядерных компьютерах была предпринята попытка в определенной степени преодолеть указанные проблемы.

Во-первых, модель параллельного выполнения программ, основанная на развертывании деревьев (ими по сути являлись деревья синтаксического разбора текста программы) и последующем их свертывании (непосредственно выполнении) была заменена на другую, сетевую (или схемную) модель вычисления значений функций в программе [4]. Сетевое представление позволяет управлять параллелизмом по готовности данных на входах соответствующих элементов се-

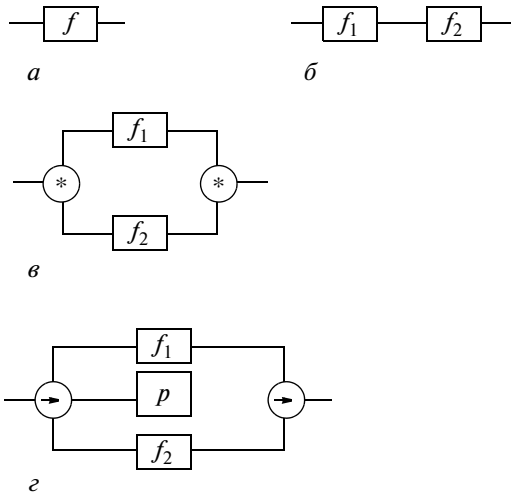

Рис. 1. Графическое представление: *а* – базисной функции f, *б* – функции $f = f_1 \cdot f_2$, *в* – функции $f = (f_1 * f_2)$, *г* – функции $f = p \to f_1, f_2$

тей, в качестве которых выступают базисные функции и функциональные переменные. Предварительное преобразование текста программы в схемную форму также устраняет необходимость выполнения развертывания деревьев.

Во-вторых, были внесены изменения в синтаксис и семантику самого языка, основным из которых является введение тернарной операции условной композиции взамен двух операций $\to$ и $+$, которые главным образом предназначены для описания условных конструкций. Новая тернарная операция – аналог условного оператора *if-then-else* в традиционных языках программирования и представляется теперь в FPTL как $(p \to f_1, f_2)$, где p – предикат, а f_1 и f_2 – функции, значение одной из которых будет использовано в зависимости от того, истинно или ложно значение $p(x)$. Хотя это несколько сужает выразительные возможности языка (в нем сложно теперь представлять так называемые параллельные функции [4]), тем не менее программист возвращается в принятое в языках программирования задание условных конструкций и, что более важно, это позволяет более эффективно их выполнять. Напомним, что в принципе значения всех трех функции p, f_1 и f_2 можно вычислять одновременно, при этом результат $p(x)$ всегда необходим для определения дальнейшего направления вычислений. Реализация упреждающих вычислений не будет эффективной, если не использовать прерывание одного из процессов ветвления $-f_1(x)$ или $f_2(x)$ сразу, как только будет вычислено значение $p(x)$. В настоящей реализации FPTL на многоядерных компьютерах возможность упреждающих вычислений значений функций, связанных с условными конструкциями, пока не используется. Управление сначала вычисляет значение предиката, а затем осуществляется переход к вычислению одной из функций f_1 или f_2 на рис. 1. Параллельные вычисления реализованы только для операции *. Это существенно упрощает управление, платой за которое является заметное уменьшение степени реализуемого параллелизма [4].

2.1. Сетевое представление функций. В реализации FPTL перед выполнением функциональной программы по ее тексту строится сетевое представление правых частей для всех функциональных переменных в системе функциональных уравнений, задающих функцию. Формально сеть есть графическое представление функций, которое строится по следующим правилам (рис. 1).

На рис. 2 приведен пример сетевого представления программы вычисления факториала.

```
scheme Fact {
    Fact = ([1] * 0).equal 1, (([1] * 1).sub.Fact * [1]).mul;
}
```

Вычисление значения функции в виде сети можно описать как направленное “движение” вычислений слева направо так, что значение каждой базисной функции начинает вычисляться по поступлению данных (dataflow-принцип), а вместо функциональных переменных при поступлении данных подставляются копии их сетей.

В реализации этой модели вычисления значений функций сеть представляется в виде многосвязных списков, а параллелизм вычислений достигается путем одновременного развертывания

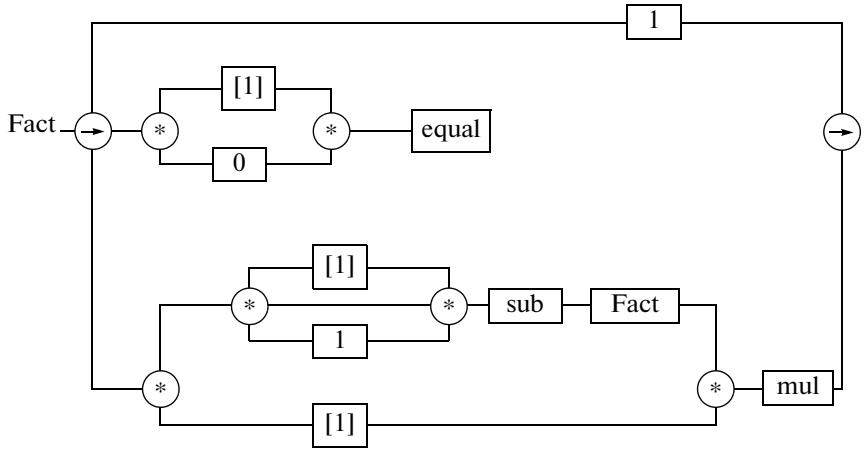

Рис. 2. Представление функциональной программы вычисления факториала в виде сети

вычислений по ветвям, исходящим из узлов, которые обозначают операцию композиции *. Однако в качестве параллельных процессов, реализующих параллелизм и назначаемых на различные процессоры или ядра компьютера, рассматриваются только процессы, порождаемые при подстановке сетей вместо рекурсивных элементов, представляющих рекурсивно определенные функциональные переменные. Эти процессы образуют очередь процессов, которые управление назначает для выполнения на свободных процессорах или ядрах компьютера. Как было отмечено, это позволяет увеличить сложность параллельно выполняемых процессов (значения базисных функций вычисляются для каждого такого процесса последовательно), что дает возможность существенно повысить эффективность вычислений (уменьшить время выполнения программы).

2.2. Реализация параллельного выполнения программ. Управление процессом параллельного выполнения FPTL-программ на многоядерных компьютерах реализовано с использованием средств MULTITHREADING, которые предоставляют описания на системном уровне параллельных процессов в виде независимых подпроцессов (нитей), одновременно выполняющихся в общем адресном пространстве памяти многоядерного компьютера.

Заметим, что реализация параллелизма в функциональных языках Haskell [8], F# [9] и др. осуществляется путем использования набора примитивов для явного задания параллелизма в программе. В отличие от этого FPTL сохраняет свою “чистую” функциональную основу, избавляя программиста от необходимости описания параллелизма на процессном уровне, мало как связанном с функциональной семантикой программы.

Перейдем непосредственно к описанию реализации управления выполнением FPTL-программы. Перед этим пользователь может указать количество нитей, которые будут использованы для параллельного выполнения FPTL-программы (обычно оно выбирается большим либо равным количеству физических ядер в компьютере). Каждая нить выполняет *задание* (task) – один из параллельных процессов вычислений, порожденных в программе операцией *. По сути, соединенные операцией * два терма эквивалентны порождению двух процессов вычисления, один из которых нить выполняет сама, а второй помещается в очередь, видимую другими нитями. При этом у каждой рабочей нити образуется своя очередь процессов (заданий), которые она обрабатывает. Если рабочая очередь пуста, то нить последовательно пытается взять задание на выполнение из очередей других нитей. Рисунок 3 иллюстрирует организационную схему работы нити при выполнении заданий. У рабочей нити PH_i есть своя динамически поддерживаемая ею очередь заданий PO_i, которые она выполняет. К этой очереди может обращаться другая нить PH_j, если она исчерпала все задания в своей очереди.

Заметим, что можно было бы использовать общую очередь для заданий, сохраняя описанную логику управления их выполнением. Однако обращение к общей очереди множества нитей требует синхронизации доступа к ней, которая является достаточно затратной по времени операцией. Управление реализовано таким образом, что перед выполнением FPTL-программы создается главная нить, которая порождает заданное программистом необходимое количество рабочих нитей и передает одной из них (в ее очередь заданий) начальное задание на выполнение: FPTL-про-

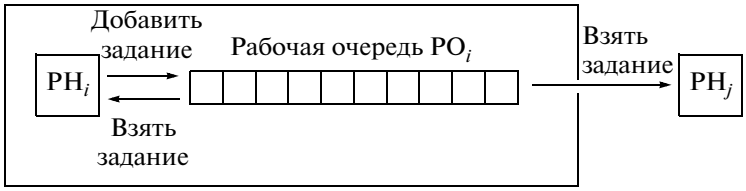

Рис. 3. Схема работы нити

грамму вместе с данными, к которым она применяется. Рабочая нить (РН) выполняет в итерационном режиме задание, которое она берет из своей очереди. Задание – это некоторый линейный участок в сетевом представлении функции. Линейные участки – последовательности функций, заключенные между двумя узлами * в сетевом представлении (рис. 1).

РН выполняет задание, производя вычисления базисных функций на линейном участке и либо завершает процесс выполнения FPTL-программы, либо "встречает" функциональную переменную, либо переходит к операции *, открывающей два новых линейных участка. В первом случае передается сообщение главной нити об окончании выполнения FPTL-программы. Во втором случае осуществляется подстановка в сеть вместо функциональной переменной ее подсхемы, и вычисления продолжаются. В третьем случае РН переходит к продолжению вычислений "верхнего" линейного участка, предварительно поместив в свою очередь новое задание – ссылку на начало "нижнего" линейного участка (см. схемы функций в разд. 2.1). Когда процесс вычислений достигает конца сети, то РН осуществляет поиск нового задания по следующей схеме: сначала производится попытка взять задание из своей очереди; если это не удается (собственная очередь пуста), то последовательно просматриваются очереди других нитей. Если обнаруживается непустая очередь, из нее извлекается задание и процесс вычислений продолжается. Поскольку выполняемые одновременно разными нитями два задания, соответствующие двум линейным участкам между операцией * в сети, могут завершиться асинхронно, т.е. в произвольном порядке, в том числе одновременно, в реализации предусмотрен механизм контроля корректного выполнения необходимых действий. Сформулируем основные решения, которые были приняты для оптимизации процессов, выполняемых нитями.

1. Количество рабочих нитей задает программист, и в процессе выполнения функциональной программы оно не изменяется. Для того чтобы полностью задействовать имеющиеся в компьютере процессоры (ядра), количество РН следует брать равным количеству физических ядер в компьютере.

2. Нить активна (выполняется), если есть задание хотя бы в одной очереди всех нитей.

3. Прерывание нити и поиск нового задания происходит только при достижении закрывающей операции * и только в том случае, если другой линейный участок этой операции, исходящий от соответствующей открывающей операции *, еще выполняется другой нитью.

4. РН берет новое задание для выполнения из конца своей очереди по принципу LIFO. Однако если эта очередь пуста, то нить вынуждена искать новое задание в очереди другой нити, которое извлекается из очереди по принципу FIFO. Это, во-первых, позволяет осуществлять миграцию более сложных с вычислительной точки зрения заданий на другие нити и, во-вторых, локализовать данные при выполнении заданий, взятых из собственной очереди.

Отметим, что схожие решения по оптимизации управления приняты при реализации fork-join API для языка Java [10], а также в библиотеках Microsoft TPL [11] и Intel TBB [12].

Реализация интерпретатора языка FPTL выполнялась на языке C++. Для работы с нитями задействовалась библиотека boost::thread [13]. В качестве системы управления памятью было использовано готовое решение в виде консервативного сборщика мусора Boehm Garbage Collector [14]. Применение автоматического управления памятью позволяет достигнуть следующих целей.

1. Обеспечивается локальность динамической памяти. При этом каждая нить имеет свою локальную "кучу". Использование общей кучи для всех нитей привело бы к увеличению накладных расходов на синхронизацию доступа к ней (при проведении экспериментов накладные расходы для общей кучи составляли порядка 35% от всего времени выполнения программы).

2. При "ручном" управлении памятью проблема определения времени жизни данных при их миграции между нитями становится нетривиальной. В то же время применение автоматического управления памятью вносит существенные ограничения на максимальное ускорение времени

Описание набора тестовых функциональных программ

Программа	Описание
Fib	Вычисление 34-го числа Фибоначчи по рекурсивной схеме
Integ	Вычисление интеграла функции $f(x) = 1/(x \times e^x)$ на отрезке $[10^{-6}, 10]$ с точностью 10^{-5} адаптивным методом трапеций
Sort	Быстрая сортировка односвязных списков размером 50 и 100 тыс. случайных вещественных чисел, равномерно распределенных на отрезке $[-1, 1]$
FFT	Быстрое преобразование Фурье суммы трех синусоидальных сигналов, заданных дискретным множеством значений функции на 8192 точках

параллельного выполнения программ, так как при своей работе сборщик мусора производит принудительное прерывание работы всех нитей. Данное ограничение особенно существенно для программ, активно использующих данные абстрактного типа, так как они содержатся только в динамической памяти.

3. Результаты экспериментов. Набор из четырех функциональных программ на FPTL (см. Приложение), взятых для экспериментов, описан в таблице. Эти программы были выбраны как примеры типовых задач, наиболее часто используемых для экспериментальной проверки эффективности реализации параллельных вычислений на многоядерных компьютерах.

Эффективность реализации FPTL оценивалась путем ее сравнения с реализациями других языков параллельного программирования для многоядерных компьютеров (Java и Haskell). В качестве критериев эффективности применялась зависимость времени выполнения и ускорения выполнения параллельной программы от количества ядер компьютера.

Все эксперименты проводились на компьютере с четырехядерным процессором Intel Core i7 с 6 Гб оперативной памяти, работающим под управлением ОС Windows 7 Pro 64-bit. Для выполнения программ на языке Haskell использовалась среда выполнения GHC 7.6.2. Результаты получены на основе трех прогонок с применением медианного фильтра к результирующим данным.

На рис. 4 представлены диаграммы зависимости времени выполнения функциональных программ, указанных в таблице с использованием разного количества ядер (количество ядер от 1 до 4 и равно количеству нитей).

На рис. 5 приведены графики ускорения выполнения тестовых программ. Отчетливо заметно, что программы на языке FPTL для вычисления чисел Фибоначчи и численного интегрирования обладают линейной масштабируемостью (время выполнения уменьшается прямо пропорционально количеству задействованных нитей или ядер), что нельзя сказать о программах быстрой сортировки и быстрого преобразовании Фурье. Как было отмечено выше, данный факт обусловлен увеличением нагрузки на систему управления памятью: для представления элементов списков используются абстрактные типы данных, которые всегда создаются в динамической памяти. Так как алгоритм сборки мусора не обладает необходимой

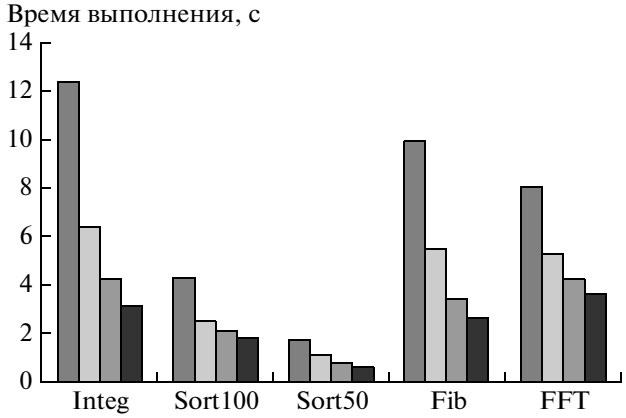

Рис. 4. Диаграммы времени выполнения тестовых программ в зависимости от количества ядер

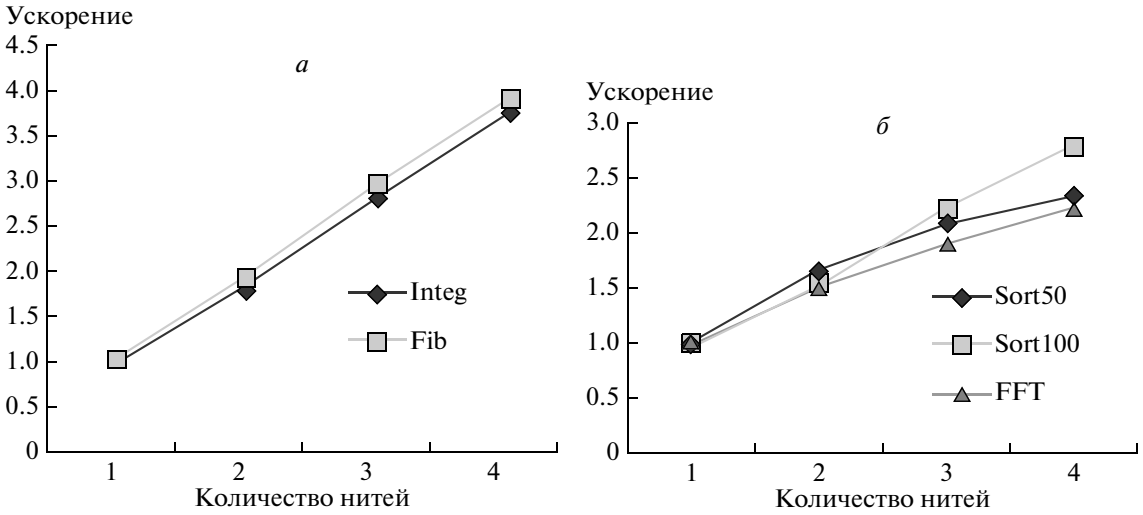

Рис. 5. Графики ускорения выполнения тестовых программ

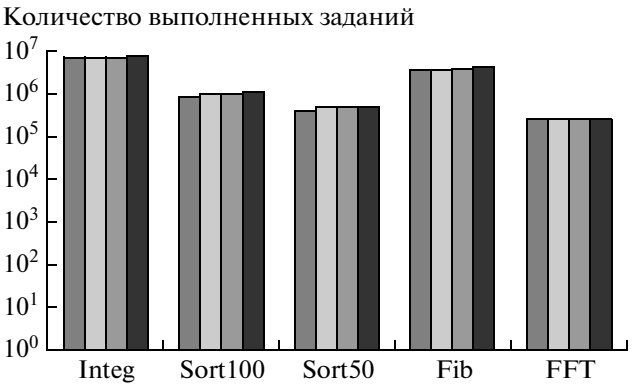

Рис. 6. Распределение количества заданий между РН

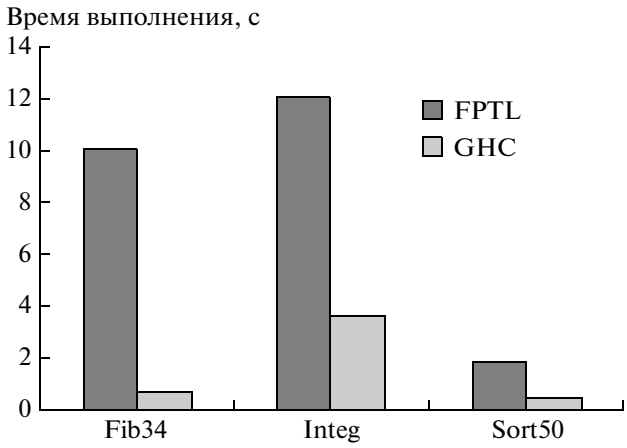

Рис. 7. Сравнение времени выполнения функциональных программ на FPTL и Haskell (на одноядерной конфигурации)

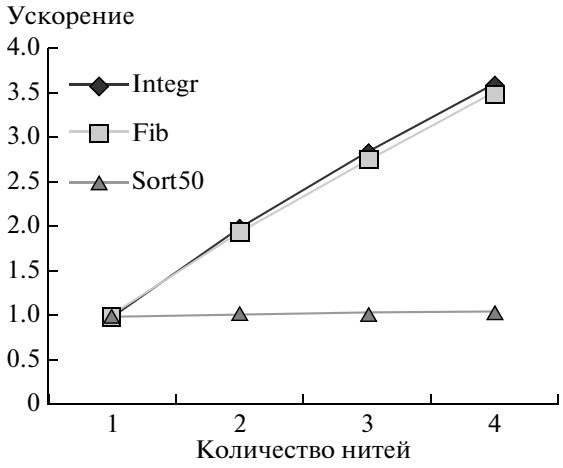

Рис. 8. Ускорение параллельного выполнения программ на языке Haskell

масштабируемостью, то общее значение ускорения времени выполнения заметно отстает от идеального линейного вида. В то же время программы численного интегрирования и вычисления чисел Фибоначчи оперируют над примитивными типами данных (целые и вещественные) и практически не используют динамическую память и, следовательно, не требуют применения сборщика мусора при их выполнении.

На рис. 6 представлена диаграмма распределения количества выполненных заданий для каждой из PH. График на рис. 6 свидетельствует о том, что алгоритм планирования позволяет примерно одинаково распределить порожденные задания между PH, тем самым обеспечивая их равномерную загрузку.

На рис. 7 приведено сравнение времени выполнения функциональных программ на FPTL с их аналогами, написанными на языке Haskell. Различие во времени выполнения объясняется тем, что GHC-реализация Haskell применяет компиляцию исходной программы. На рис. 8 для сравнения описаны графики зависимости ускорения параллельного выполнения программ на языке Haskell от количества задействованных ядер. Видно, что программа быстрой сортировки, оперирующая списковыми структурами, практически не ускоряется при увеличении количества ядер. Если сравнивать графики зависимости ускорения выполнения параллельных программ сортировки на FPTL и Haskell, то можно увидеть, что реализация Haskell проигрывает реализации FPTL. Это объясняется тем, что в обоих случаях используются абстрактные типы данных – списки, обработку которых на компьютере компиляции не может улучшить.

Заключение. Несмотря на то, что настоящая реализация FPTL уступает по эффективности наиболее известным языкам и средам программирования, поддерживающим параллельное выполнение на многоядерных компьютерах, тем не менее это первая реализация чисто функционального языка параллельного программирования, в котором для распараллеливания программ не требуется использовать введение в язык специальных средств для задания параллелизма. Это существенно упрощает разработку функциональных программ, их отладку и выполнение. В FPTL программист может “регулировать” параллелизм в программе, используя эквивалентные преобразования. Созданная система функционального программирования в настоящее время применяется в обучении методам и средствам параллельного программирования студентов и аспирантов кафедры Прикладной математики Московского энергетического института.

Система развивается в следующих направления: реализуется алгоритм типового контроля, осуществляемого на этапе компиляции программы. Это позволит существенно сократить время выполнения FPTL-программ. Также выполняется проект создания FPTL-подобного языка как синтаксического подмножества языков C++ и Java.

ПРИЛОЖЕНИЕ

Тексты тестовых программ на языке FPTL

```
// Вычисление 35-го числа Фибоначчи.
Scheme Fib {
      Fib = ([1]*0).equal -> 1,
                  ([1]*1).equal -> 1,
                  ((([1]*2).sub.Fib*([1]*1).sub.Fib).add);
}
Application
% Fib(34)
// Численное интегрирование.
Scheme Integ
{
     Integ = (10.0 * 0.000001 * 0.00001).Integrate(TestFunc).print;
     TestFunc = (1.0*([1].exp*[1]).mul).div;
     Fun Integrate[fFunction]
     {
          @ = (((([1] * Mid).Trp * (Mid * [2]).Trp).add * ([1] *
[2]).Trp).sub.abs
       [3]).less -> ([1] * [2]).Trp, ((([1] * Mid * ([3] *
2.0).div).Integrate * (Mid
       * [2] * ([3] * 2.0).div).Integrate).add);
          Mid = (([1] * [2]).add * 2.0).div;
          Trp = ((([2] * [1]).sub * ([2].fFunction * [1].fFunc-
tion).add).mul * 2.0).div;
       }
}
Application
% Integ
// Быстрая сортировка.
// Тип данных: параметризованный список.
Data List['t]
{
   List = c_nil ++ 't * List['t].c_cons;
}
Scheme Fact
{
   @ = 30000.RandomList(-999999,999999).QSort;
   // Создание случайного списка вещественных чисел.
   Fun RandomList[Max, Min]
   {
          @ = (id * 0).equal -> c_nil,
                  ((Min * (rand * (Max * Min).sub).mul).add * (id
* 1).sub.RandomList).c_cons;
   }
   Fun QSort
   {
       @ = ~c_nil -> c_nil,
              (((id*Pivot).Filter(less).QSort * (id*Pivot).Fil-
ter(equal)).Concat * (id*Pivot).Filter(greater).QSort).Concat;
          Fun Filter[fPredicate]
          {
              @ = [1].~c_nil -> c_nil,
```

```
                    [1].~c_cons -> Args.(([1]*[3]).fPredicate -> ([1]
* ([2]*[3]).Filter).c_cons, ([2]*[3]).Filter);
                Args = [1].~c_cons * [2];
            }
            // Возвращает опорный элемент (первый в списке).
            Pivot = id.~c_cons.[1];
            Concat = [1].~c_nil -> [2],
                     [2].~c_nil -> [1],
                     ([1].~c_cons.[1]  *  ([1].~c_cons.[2]*[2]).Con-
cat).c_cons;
      }
  }
  Application
  % Fact
  // Быстрое преобразование Фурье.
  // Тип данных: параметризованный список.
  Data List['t]
  {
       List = c_nil ++ 't * List['t].c_cons;
  }
  // Тип данных: комплексное число.
  Data Complex
  {
       Complex = double * double.c_complex;
  }
  Scheme Fact
  {
       @ = id.GenSinSignal.FFT;
       // Быстрое преобразование Фурье. Входные данные A - список ве-
щественных чисел, длина - степень двойки.
       Fun FFT
       {
            @ = (N * 1).equal -> id,
                (OddCoeff.FFT  *  EvenCoeff.FFT  *  W  *  Wn).(Form(Op1,
CompMul) * Form(Op2, CompMul)).Concat;
            // Выбирает элементы списка с четными номерами.
            EvenCoeff =
                ~c_nil -> c_nil,
                ~c_cons.[2].~c_nil -> c_nil,
                ~c_cons.[2].~c_cons.([1] * [2].EvenCoeff).c_cons;
            // Выбирает элементы списка с нечетными номерами.
            OddCoeff =
                ~c_nil -> c_nil,
                ~c_cons.[2].~c_nil -> ~c_cons.[1],
                ~c_cons.([1] * [2].~c_cons.[2].OddCoeff).c_cons;
            N = id.Length;
            Wn = (0.0 * ((2.0 * 3.141592).mul * N).div).c_complex.Com-
pExp;
            W = (1.0 * 0.0).c_complex;
            // Формирует преобразованный список.
            // Входные данные EvenCoeff, OddCoeff, W, Wn
            Fun Form[aOperation, aMul]
            {
                 @ = [1].~c_nil -> c_nil,
```

```
                    ((Ec * Oc * W).aOperation * ([1].~c_cons.[2] *
[2].~c_cons.[2] * NewW * Wn).Form).c_cons;
                Ec = [1].~c_cons.[1];
                Oc = [2].~c_cons.[1];
                W = [3];
                Wn = [4];
                NewW = (W * Wn).aMul;
            }
            CompAdd = (([1].Real * [2].Real).add * ([1].Im *
[2].Im).add).c_complex;
            CompSub = (([1].Real * [2].Real).sub * ([1].Im *
[2].Im).sub).c_complex;
            CompMul = ( (([1].Real*[2].Re-
al).mul*([1].Im*[2].Im).mul).sub * (([1].Im*[2].Real).mul * ([1].Re-
al*[2].Im).mul).add).c_complex;
            CompExp = ((Real.exp * Im.cos).mul * (Real.exp *
Im.sin).mul).c_complex;
            Real = ~c_complex.[1];
            Im = ~c_complex.[2];
            Op1 = ([1] * ([2] * [3]).CompMul).CompAdd;
            Op2 = ([1] * ([2] * [3]).CompMul).CompSub;
            // Вычисление длины списка.
            Length = ~c_nil -> 0, (~c_cons.[2].Length * 1).add;
            // Конкатенация двух списков.
            Concat = [1].~c_nil -> [2],
                      [2].~c_nil -> [1],
                     ([1].~c_cons.[1] * ([1].~c_cons.[2]*[2]).Con-
cat).c_cons;
        }
        // Генерация сигнала.
        Fun GenSignal[Generator]
        {
            @ = (id * 0).equal -> c_nil,
                (id.Generator * (id * 1).sub.GenSignal).c_cons;
        }
        SinSignal = ((id * 1024.0).div * 3.141592).mul.(id *
sin).c_complex;
        GenSinSignal = GenSignal(SinSignal);
    }
    Application
    % Fact(8192)
```

СПИСОК ЛИТЕРАТУРЫ

1. *Кутепов В.П., Фальк В.Н.* Модели асинхронных вычислений значений функций в языке функциональных схем // Программирование. 1978. № 3.
2. *Кутепов В.П., Фальк В.Н.* Направленные отношения: теория и приложения // Изв. РАН. Техн. кибернетика. 1994. № 4, 5.
3. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А.* Язык функционального параллельного программирования и его реализация на кластерных системах // Программирование. 2005. № 5.
4. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А.* Структурный анализ и планирование процессов параллельного выполнения функциональных программ // Изв. РАН. ТиСУ. 2005. № 6.
5. *Church A.* The Calculi of Lambda-conversion // Ann. of Math. Studies, Princeton, N.J.: Princeton University Press, 1941. V. 6.

6. *McCarthy J.* Recursive Functions of Symbolic Expressions and Their Computation by Machine. Cambridge, Mass.: MIT, 1960.
7. *Milner R.G.* The Standard ML Core Language. Polymorphism // The ML/LCF/Hope Newsletter. 1985. V. 2. № 2.
8. *Peyton Jones S.L.* The Implementation of Functional Programming Languages. London: Prentice Hall, 1987.
9. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ack.aspx.
10. *Bluemofe R.D., Leiserson C.E.* Scheduling Multithreaded Computations by Work Stealing // J. ACM. 1999.
11. *Lea D.* A Java Fork/Join Framework. Oswego: State University of New York, 2000.
12. *Leijen D., Schulte W., Burckhardt S.* The Design of a Task Parallel Library // Microsoft Research, 2009. http://research.microsoft.com/pubs/77368/TheDesignOfATaskParallelLibrary(oopsla2009).pdf.
13. http://threadingbuildingblocks.org.
14. http://boost.org.
15. *Boehm H.J.* The Boehm-Demers-Weiser Conservative Garbage Collector // HP Labs 2004. http://www.hpl.hp.com/personal/Hans_Boehm/gc/04tutorial.pdf.