© 2012 г. В.П. КУТЕПОВ, д-р техн. наук,
П.Н. ШАМАЛЬ,
(Национальный исследовательский университет Московский энергетический институт)

РЕАЛИЗАЦИЯ ЯЗЫКА ФУНКЦИОНАЛЬНОГО ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ FPTL НА МНОГОЯДЕРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

В статье описана реализуемая на многоядерных компьютерных системах система функционального параллельного программирования. Система включает язык композиционного функционального параллельного программирования и средства управления параллельным выполнением программ на многоядерных компьютерах.

IMPLEMENTATION OF FUNCTIONAL PARALLEL PROGRAMMING LANGUAGE ON MULTICORE COMPUTER SYSTEMS / V.P. Kutepov, P.N. Shamal (National Research University “Moscow Power Engineering Institute”, Krasnokazarmennaya 14, Moscow, 111250 Russia, E-mail: uvs@mpei.ru). In the paper, a functional parallel programming system implemented on multicore computer systems is discussed. It includes a language of compositional functional parallel programming and tools for controlling parallel code execution on multicore computer systems.

1. Введение

Доклад посвящен реализуемому на кафедре прикладной математики НИУ МЭИ проекту создания системы функционального параллельного программирования для многоядерных вычислительных систем.

Проект содержит три независимые части: высокоуровневый язык композиционного функционального параллельного программирования, инструментальные средства программирования на нем и система управления процессом параллельного выполнения функциональных программ на многоядерных системах. Внимание уделяется описанию реализации интерпретатора и средств управления параллельным выполнением функциональных программ на многоядерных вычислительных системах. Сначала будет дано краткое описание теоретической базы языка

2. Язык FPTL

Язык FPTL создавался как прообраз общепринятой в математической практике форме 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 определить параллелизм.

Функции в FPTL рассматриваются, как *(m, n)* - арные соответствия *(m ≥ 0, n ≥ 0)* между кортежами данных, где *m* - длина кортежа на входе функции, *n* - на выходе. Функции арности (0, 1) рассматриваются в FPTL как константы. Для *m* или *n* равным 0 имеются два кортежа нулевой длины, обозначаемые 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 данных длины m, $m > 0$, выбирает его i-й элемент, $0 \le i \le m,\ m > 0$. 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 $F_i = \tau_i$, $i=1,2,\ldots,n$, где τ_i – функциональный термы, построенные из заданных (базисных) функций и функциональных переменны F_i путем применения четырех операций композиции функций: →, +, *, •. Для функций и функциональных переменных задана их арность, а для базисных функций также их тип. Пусть f_1, f_2 заданные функции, а f – определяемая новая функция, α, β, γ – обозначение кортежей. Семантика операций композиции определяется следующим образом:

1. Последовательная композиция (•):

$$f^{(m,n)} \underset{def}{\Longleftrightarrow} \left(f_1{}^{(m,k)} \cdot f_2{}^{(k,n)}\right), f(\alpha) \underset{def}{\Longleftrightarrow} f_2(f_1(\alpha))$$

где $\alpha = \alpha_1\ \alpha_2\ \alpha_3\ \ldots\ \alpha_m$ – кортеж данных, каждый элемент которого имеет тип t_i. Здесь и далее сначала задается синтаксис операции композиции, а затем ее семантика, определяемая через применение функций к кортежу данных. Предполагается, что типы данных кортежа значений функции f_1 и типы данных кортежа аргументов функции f_2 одинаковы.

Заметим, что в FPTL используется «префиксная» форма записи операции последовательной композиции, определяющая 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 (*):

$$f^{(m,n_1+n_2)} \underset{def}{\Longleftrightarrow} \left(f_1{}^{(m,n_1)} * f_2{}^{(m,n_2)}\right), f(\alpha) \underset{def}{\Longleftrightarrow} f_1(\alpha_1) f_2(\alpha_2)$$

(конкатенация кортежей значений двух функций).

3. Операция условной композиции:

$$f^{(m,n)} \underset{def}{\Longleftrightarrow} (f_1{}^{(m,k)} \rightarrow f_2{}^{(m,n)})$$

$f(\alpha) = f_2(\alpha)$, если значение $f_1(\alpha)$ определено (отлично от ω и не равно булевской константе «ложь»), в противном случае считается неопределенным.

4. Операция объединения (графиков) ортогональных функций:

$$f^{(m,n)} \underset{def}{\Leftrightarrow} ({f_1}^{(m,n)} + {f_2}^{(m,n)})$$

$$f(\alpha) \underset{def}{\Leftrightarrow} \begin{cases} f_1(\alpha), \textit{если } f_1(\alpha) \textit{ определено,} \\ f_2(\alpha), \textit{если } f_2(\alpha) \textit{ определено.} \end{cases}$$

Напомним, что функции f_1 и f_2 ортогональны, если для всякого кортежа α, всегда определена не более, чем одна из них. Общепринятая форма условного оператора *if p(x) then f_1(x) else f_2(x)* в FPTL представляется как $(p \to f_1) + (\overline{p} \to f_2)$.

Все операции композиции являются ассоциативными, а операция + коммутативна. Приоритет операций композиции определяется следующем образом (в порядке возрастания): +, →, *, •.

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 указанных операций композиции. Предполагается, что арности и типы терма и определяемой им функциональной переменной одинаковы. Функциональные переменные выполняют двойную роль при задании функции (построении функциональной программы): одни из них появляются как необходимые элементы при задании рекурсивных функций, другие выполняют роль далее (в следующем уравнении уточняемой функции), определяя пошаговую разработку функциональной программы по технологии проектирования «сверху-вниз».

Определяемые абстрактные типы данных в FPTL задаются в виде реляционных уравнений. Кроме того можно использовать встроенные типы: *bool*, *double*, *int*, *string* и т.д. вместе с определенными в системе операциями над ними.

Для определения новых типов данных используются конструкторы и деструкторы (обратные конструкторам функции). Для определения данных используются 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] они достаточны для того, чтобы средствами задания функций в FPTL можно было определить любую вычислимую функцию над этими данными. Также в языке имеются конструкторы констант, позволяющие создавать данные встроенных типов.

Приведем пример определения 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.

Обратные к ним функции (деструкторы), обозначаемые как ~c_null, ~c_succ, ~c_nil, ~c_cons имеют следующую интерпретацию:

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

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

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

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

Приведем функцию-предикат, которая проверяет принадлежность кортежа на входе типу данных ListOfNat:

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

IsListOfNat определена на любом кортеже данных, принадлежащих ListOfNat, и ее значением является λ(что может быть трактовано, как«истина»), для других отличных от ListOfNat значений в качестве результата применения IsListOfNat будет ω (неопределенное значение).

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 -> 0 + ~c_cons -> (~c_cons.[2].Length * 1).add;
}

application
    list = (c_null.c_succ * (c_nil * c_null).c_cons).c_cons;
    % Length(list)
```

В FPTL программах можно определять параметризованные функции и типы данных. Например, можно определить абстрактный тип данных List, использую параметр ‘t (в FPTL штрих добавляется при определении имени параметра):

```
data List[‘t] {
```

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

Определяя ‘t как тип Nat, мы получаем ранее определенный абстрактный тип ListOfNat.

3. Реализация интерпретатора языка FPTL на многоядерных компьютерах

Теоретическая модель параллельного вычисления значений функций в FPTL рассматривалась в [4] и она основана на правилах развертывания и свертывания (собственно вычислений) бинарных деревьев. Эта модель была реализована в [3], однако, по ряду причин она оказалась неэффективной из-за большой доли накладных расходов при работе с деревьями. В новой версии реализации языка на многоядерных компьютерах с целью повышения ее эффективности (достижения большего ускорения выполнения программ) были введены изменения как в сам язык, так и в реализуемую модель параллельного выполнения FPTL программ. Взамен двух операций → и +, которые, главным образом, предназначены для описания условных конструкций, введена одна тернарная операция в форме $p \rightarrow f_1, f_2$, где p – предикат, а f_1 и f_2 функции, значения одной из которых вычисляется 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 конструкции, и что более важно, 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 упреждающий вычислений значений функций, связанных с условными конструкциями (рис. 1) 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].

Кроме того, была существенно изменена сама модель параллельного выполнения FPTL программ. Вместо операций свертывания и развертывания деревьев процесс параллельного вычисления значений функций в новой модели использует сетевое представление термов правых частей в системе функциональных уравнений [4] – структурированные функциональные схемы. Это представление позволяет управлять параллелизмом по готовности данных на входах соответствующих элементов сетей, в качестве которых выступают базисные функции и функциональные переменные. Для первых при поступлении кортежа данных на вход начинается применение функции к этим данным. Для вторых осуществляется переход к соответствующему схемному представлению правой чести функционального уравнения. Представленные операции композиции функциональных схем приведены на рис. 1. На рис. 2 приведен пример схемного представления функции вычисления факториала натурального числа.

В текущей реализации FPTL функциональной программе по ее тексту строится схемное представление правых частей для всех функциональных переменных в системе функциональных уравнений, задающих функцию. Выполнение программы, описанной в

виде структурированной схемы, можно представить в виде направленного «движения» вычислений слева направо так, что каждая базисная функция начинает вычисляться по поступлению данных, а вместо функциональных переменных при поступлении данных подставляется копия ее подсхемы и процесс вычисления продолжается для нее. Чтобы данная подстановка была возможной, ссылка на следующий за функциональной переменной узел сохраняется в стеке. После того, как вычислительный процесс достигает последнего узла схемы, выполнение программы продолжается для узла, сохраненного на вершине стека и т.д. Использование собственного стека позволяет избежать нагрузки на стек интерпретатора (native stack), объем которого строго фиксирован и не может быть изменен после создания нити.

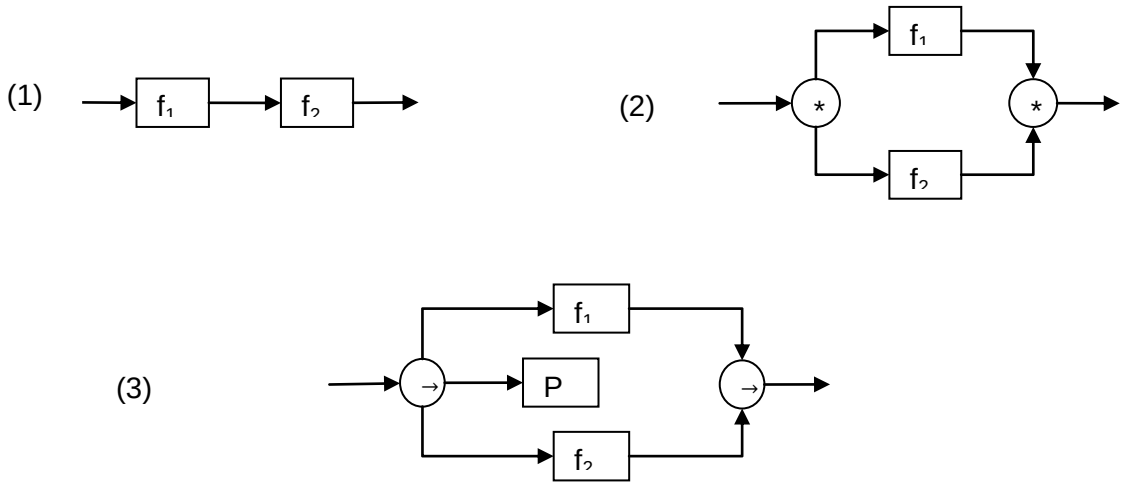

Рисунок 1. Представление операций композиции в виде структурированных функциональных схем.

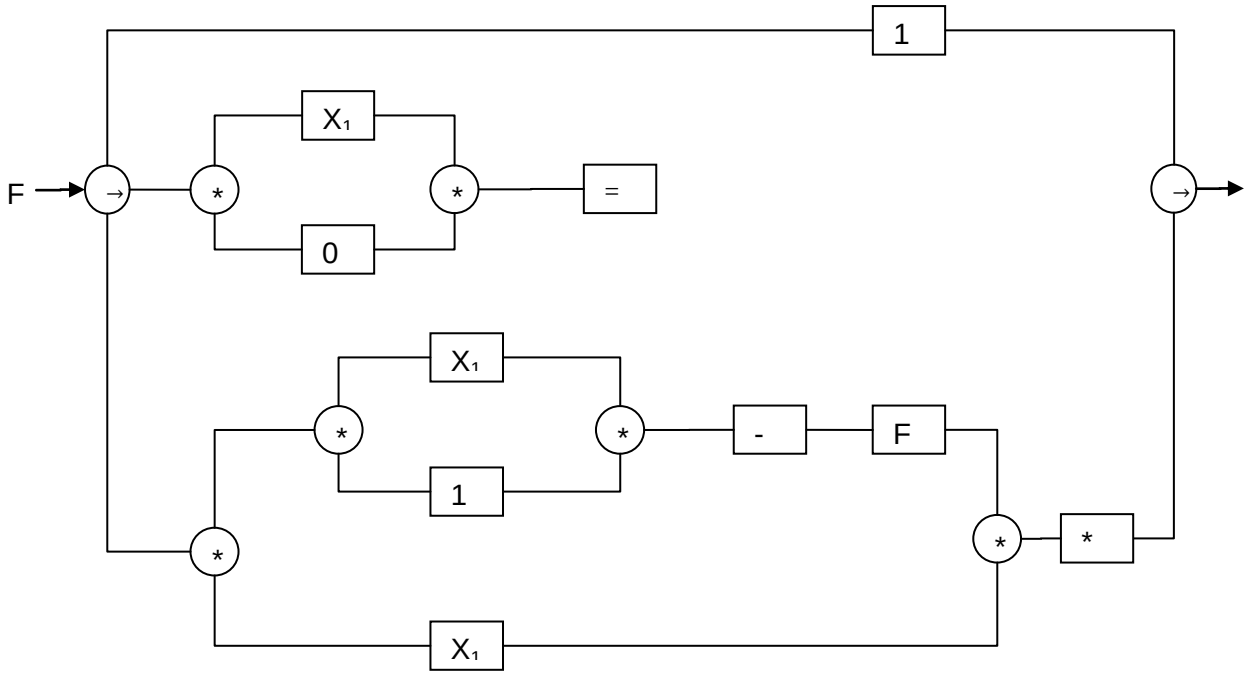

Рисунок 2. Схемное представление функциональной программы для вычисления факториала.

Следует отметить, что интерпретационная реализация языка неизбежно приводит к увеличению накладных расходов, при этом время выполнения программы, как правило, будет больше, чем при произведенной трансляции программы в набор команд для специфической компьютерной архитектуры (решение компиляции).

Управление процессом параллельного выполнения FPTL-программ на многоядерных компьютерах реализовано с использование средств multithreading, основанных на модели выполнения параллельных процессов. Multithreading предоставляет средства описания на системном уровне параллельных процессов в виде независимых, одновременно выполняющихся подпроцессов (нитей), оперирующих в едином адресном пространстве памяти. Динамическое порождение нитей позволяет описывать рекурсивные параллельные процессы.

Реализация параллелизма в функциональных языках Haskell [5], F# [6] и др. также осуществляется путем введении в сам язык набора примитивов для задания параллелизма. В отличие от этого FPTL сохраняет свою «чистую» функциональную основу избавляя программиста от необходимости использования средств явного задания параллелизма на процессном уровне, мало как связанном с функциональной семантикой программы.

Перейдем непосредственно к описанию реализации управления выполнением параллельных программ. Перед выполнением программы пользователь может указать количество нитей, которые будут использованы для параллельного выполнения FPTL программы. 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 термов слева и справа от оператора *, один из которых выполняющая эту операцию нить заносит в очередь, а другой выполняет сама. Таким образом, у каждой нити образуется своя очередь рабочих

заданий, часть которых она выполняет сама, а часть – нити, которые не имеют заданий в своих очередях.

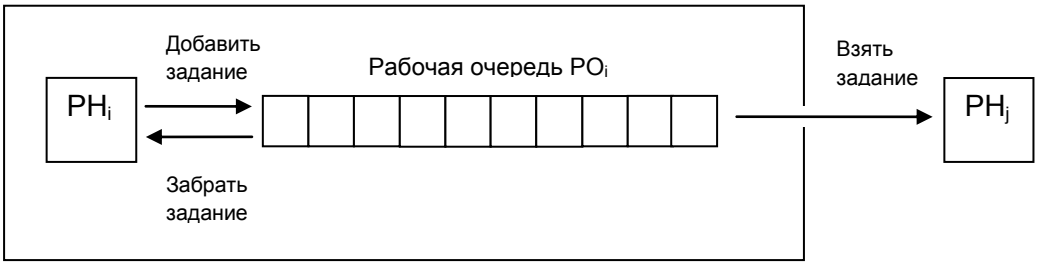

Рисунок 3. Схема работы нити.

Рис. 3 иллюстрирует организационную схему работы нити при выполнении заданий. У рабочей нити PH_i есть своя динамически поддерживаемая ей очередь заданий, которые она выполняет. К этой очереди может обращаться другая нить PHj, если она исчерпала все задания в своей очереди.

Заметим, что можно было использовать общую очередь для готовых к выполнению заданий, сохраняя описанную логику управления их выполнением. Однако, обращение к общей очереди множества нитей требует синхронизации доступа к ней. Как правило, реализация синхронизации обращения многих нитей к одной общей очереди - достаточно затратная по времени операция, кроме того, только один процесс может иметь доступ к этой очереди в произвольный момент времени. 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 начальное задание для выполнения - FPTL программу 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 структурном представлении схемы функции. Линейные участки представляют собой последовательности функций, заключенные между двумя операторами * в структурном представлении схемы (рис. 2). Следует отметить, что большинство порождаемых таким способом заданий, будут представлять собой слишком малый объем вычислений, что повлечет за собой сильное уменьшение «зернистости» распараллеливания. Для устранения этого недостатка настоящая реализация FPTL производит порождение нового задания в том, и только в том случае, если оба участка схемы между открывающей и закрывающей операцией * содержат рекурсивные подсхемы.

Рабочая нить выполняет задание (в интерпретационном режиме вычисляется последовательность задания функций на линейном участке) либо завершает процесс выполнения FPTL программы, либо «встречает» функциональную переменную, либо переходит к операции *, открывающей два линейных участка. В первом случае передается сообщение главной нити об окончании выполнения FPTL программы. Во втором случае осуществляется подстановка вместо функциональной переменной ее подсхемы. В третьем случае РН переходит к продолжению вычислений левого линейного участка, предварительно поместив в свою очередь новое задание – ссылку на начало правого линейного участка. Поскольку выполняемые одновременно разными нитями два линейных участка может 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 оно не меньше количества ядер компьютера. Обычно оно равно количеству ядер и не изменяется в процессе выполнения FPTL программы.
2. Ни одна из нитей не простаивает, 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 это операции выполняется другой нитью.
4. Рабочая нить берет новое задание для выполнения из конца своей очереди по принципу LIFO. Однако, если нить вынуждена искать некое задание в очереди другого процесса, то оно извлекается 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 РН.

Отметим, что схожие решения по оптимизации управления также приняты при реализации fork-join API для языка Java [8], а также в библиотеках Intel TBB [9] и Microsoft TPL [10].

Интерпретатор языка FPTL реализован на языке C++. Для работы с нитями использовалась библиотека boost::thread [11]. В качестве системы управления памятью было использовано готовое решение в виде консервативного сборщика мусора Boehm Garbage Collector [12]. Использования 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 использование автоматического управления памятью вносит существенные ограничения на максимальное ускорение времени параллельного выполнения программ, т.к. при своей работе сборщик мусора производит принудительное прерывание работы всех нитей. Данное ограничение особенно существенно для программ, активно использующих данные абстрактного типа, т.к. они содержатся только в динамической памяти.

4. Результаты экспериментов

Набор из четырех функциональных программ, описан в таблице 1. Они были выбраны, как примеры типовых вычислительных задач, которые могут быть решены с помощью языка FPTL, а также в качестве источников для практических результатов исследования эффективности принятых решений. Также две из четырех представленных программ были реализованы на языке Java. Несмотря на сильное отличие как самого языка Java, так системы его выполнения, данный язык был выбран как обладающий схожими [8] с использованными в интерпретаторе FPTL средствами распараллеливания.

Имя программы	Описание
Fib	Вычисление 35-го числа Фибоначчи по рекурсивной схеме.
Integ	Вычисление интеграла функции $f(x) = 1/(x * e^x)$ 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 списком дискретным множеством значений функции в 1024 точках.

Все эксперименты проводились на компьютере с 4-х ядерным процессором Intel Core i7 и 8 Гб оперативной памяти, работающим под управлением ОС Windows 7 Pro 64-bit. Программы на языке Java выполнялись под управлением виртуальной машины Hotspot. Результаты получены на основе трех прогонок с применением медианного фильтра к результирующим данным.

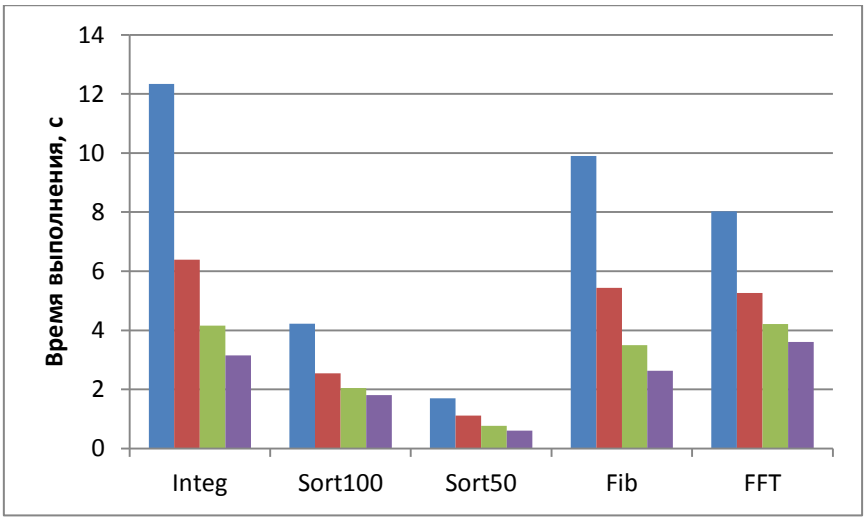

Рис 4. Время выполнения тестовых программ на различном количестве нитей.

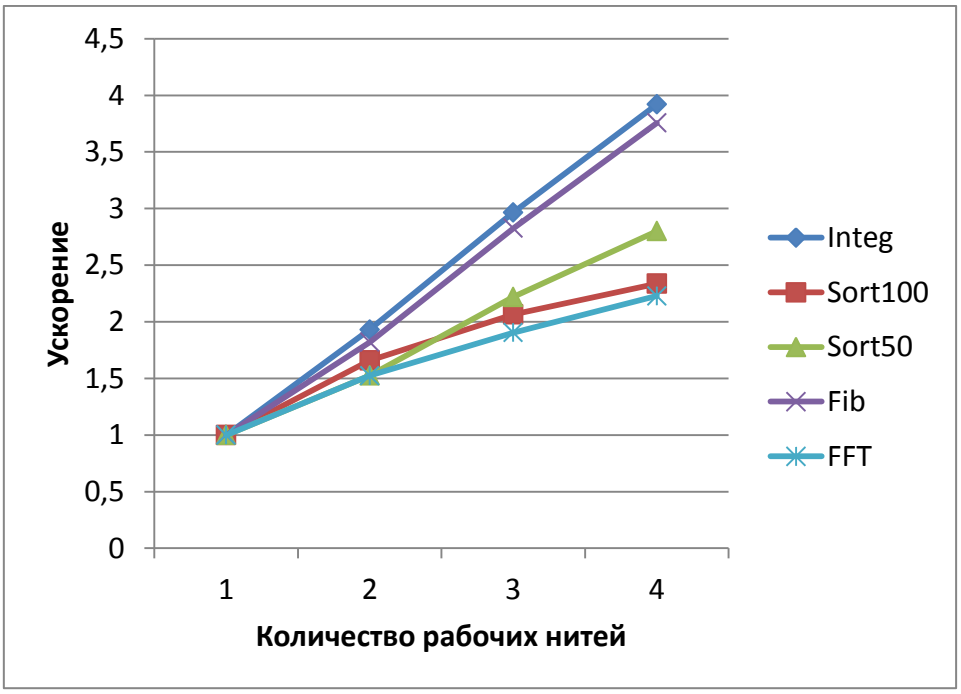

Рис 5. Ускорение выполнения тестовых программ.

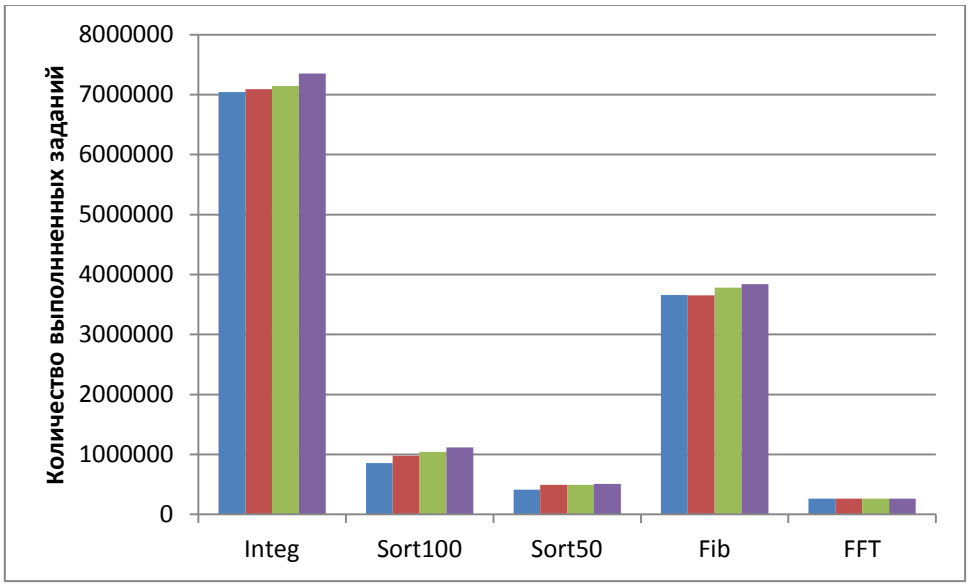

Рис 6. Распределение нагрузки между рабочими нитями.

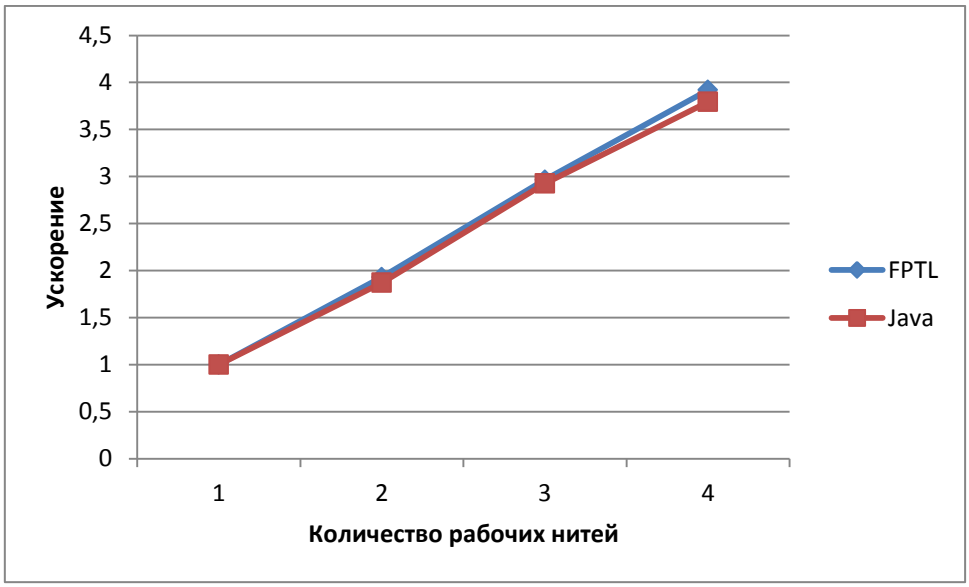

Рис 7. Сравнение ускорения выполнения программы численного интегрирования.

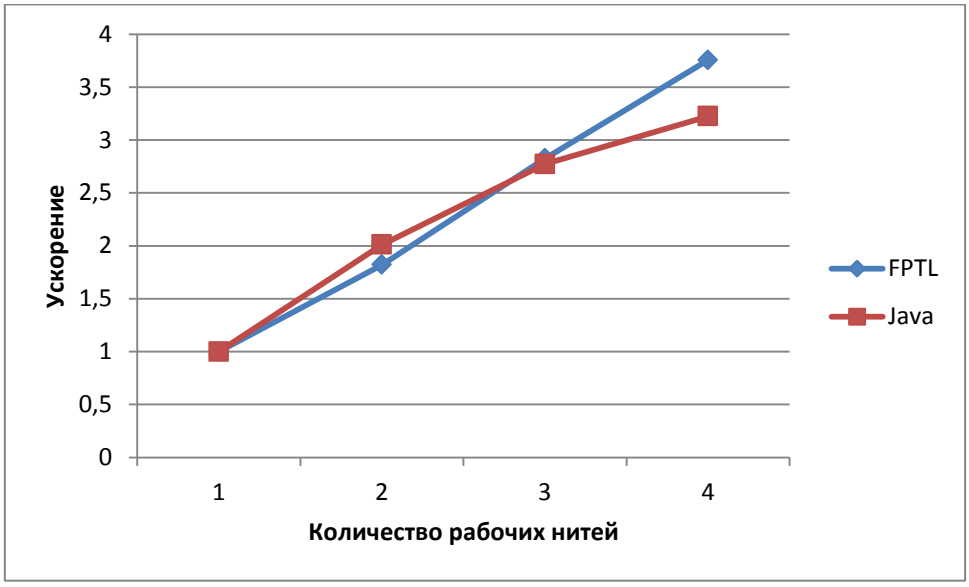

Рис 8. Сравнение ускорения выполнения программы вычисления числа Фибоначчи.

На графике рис. 5 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), что нельзя сказать о программах быстрой сортировки и быстрого преобразовании Фурье. Как отмечалось в предыдущем разделе, данный факт обусловлен увеличением нагрузки на систему управления памятью: для представления элементов списка используются абстрактные типы данных, которые всегда создаются в динамической памяти. Т.к. алгоритм сборки мусора не обладает необходимой масштабируемостью, то общее значение ускорения времени выполнения заметно отстает от идеального линейного

вида. В тоже время программы численного интегрирования и вычисления чисел Фибоначчи оперируют над встроенными типами данных (целыми и вещественными) и практически не используют динамическую память, благодаря чему масштабируются линейно.

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

На рис. 7 и 8 представлено сравнение времени выполнения программ вычисления чисел Фибоначчи и численного интегрирования на языке FPTL и языке Java. Реализация на языка Java выполнена с использованием *fork-join* библиотеки, подробнее описанной в [8].

В ходе экспериментов программы на языке Java в среднем выполнялись в 8-10 раз быстрее, чем соответствующие программы на языке FPTL при одинаковом количестве рабочих нитей. Данный факт можно объяснить следующими причинами: виртуальная машина Java использует в процессе выполнения оптимизирующую компиляцию в машинный код, в то время как среда выполнения FPTL выполняет программу в режиме интерпретации и использует при этом высокоуровневое списковое представление программы. Все это ведет к увеличению накладных расходов, причиной которых является наличие большого количества вспомогательных операций при работе с внутренним представлением при расчете на одну полезную операцию.

5. Заключение

Главным результатом работы является разработка и реализация базовой версии системы выполнения функциональных программ для многоядерных вычислительных систем. В ходе работы был разработан интерпретатор языка и адаптированы известные алгоритмы управления процессом выполнения на многоядерных вычислительных системах. Также был выявлен ряд технических проблем, основной из которых является низкая «скорость» работы интерпретатора по сравнению с компилируемыми языками.

Для дальнейшей работы ставятся следующие цели:

- перевод языка на платформу JVM или Microsoft .NET с целью увеличения интеграционных возможностей с другими языками программирования;
- расширение набора встроенных функций и типов данных;
- использование механизма *Just-in-time* (JIT) компиляции для ускорения времени работы интерпретатора.

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 РАН, 2005
4. *Бажанов С.Е., Кутепов В.П., Шестаков Д.А.* Структурный анализ и планирование процессов параллельного выполнения функциональных программ. // Известия РАН. Теория и системы управления, 2005
5. *Peyton Jones S. L.* The implementation of functional programming languages. Prentice Hall, 1987.
6. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ack.aspx
7. *Robert D. Bluemofe, Charles E. Leiserson.* Scheduling multithreded computations by work stealing.// 35th Annual Symposium on Foundations of Computer Science (FOCS 1994)
8. *Doug Lea.* A Java Fork/Join Framework, 2000
9. http://threadingbuildingblocks.org
10. *Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt.* The Design of a Task Parallel Library // Microsoft Research, 2009
11. http://www.boost.org/doc/libs/1_44_0/doc/html/thread.html
12. *H. J. Boehm.* The Boehm-Demers-Weiser Conservative Garbage Collector. // HP Labs 2004

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(35)

// 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].fFunction).add).mul * 2.0).div;
	}
}
Application
% Integ

// Быстрая сортировка.
Scheme Sort {
	Sort = [1].getRandomList.QSort;

	Fun QSort {
		QSort = isEmptyList -> emptyList,
			(((id*Pivot).Filter(less).QSort * (id*Pivot).Filter(equal)).concatList *
(id*Pivot).Filter(greater).QSort).concatList;

		Fun Filter[fPredicate]
		{
			Filter = [1].isEmptyList -> emptyList,
				Args.(([1]*[3]).fPredicate -> ([1] * ([2]*[3]).Filter).prependList,
([2]*[3]).Filter);

			Args = [1].splitList * [2];
		}

		NoFilter = (1 * 1).equal;

		// Возвращает опорный элемент (первый в списке).
		Pivot = id.splitList.[1];
	}
}
Application
% Sort(100)
// Быстрое преобразование Фурье.
```

```
Data Complex {
	Complex = double * double.c_complex;
}

Scheme FFT {
	FFT = [1].getSinSignal.FFT;

	Fun FFT {
		FFT = (N * 1).equal -> id,
		   ((OddCoeff.FFT * EvenCoeff.FFT * W * Wn).Form(Op1, CompMul) * (OddCoeff.FFT *
EvenCoeff.FFT * W * Wn).Form(Op2, CompMul)).concatList;
		EvenCoeff =
			isEmptyList -> emptyList,
			splitList.[2].isEmptyList -> emptyList,
			splitList.[2].splitList.([1] * [2].EvenCoeff).prependList;
		OddCoeff =
			isEmptyList -> emptyList,
			splitList.[2].isEmptyList -> splitList.[1],
			splitList.([1] * [2].splitList.[2].OddCoeff).prependList;

		N = id.sizeOfList;
		Wn = (0.0 * ((2.0 * 3.141592).mul * N).div).c_complex.CompExp;
		W = (1.0 * 0.0).c_complex;

		// Входные данные EvenCoeff, OddCoeff, W, Wn
		Fun Form[aOperation, aMul]
		{
			Form = [1].isEmptyList -> emptyList,
				((Ec * Oc * W).aOperation * ([1].splitList.[2] * [2].splitList.[2] * NewW *
Wn).Form).prependList;

			Ec = [1].splitList.[1];
			Oc = [2].splitList.[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].Real).mul*([1].Im*[2].Im).mul).sub * (([1].Im*[2].Real).mul *
([1].Real*[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;
	}

}
Application
% FFT(1024)
```