# Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ

[1]А.И.Легалов <legalov@mail.ru>
[1]В.С.Васильев <rrrFer@mail.ru>
[1]И.В.Матковский <alpha900i@mail.ru>
[1]М.С.Ушакова <ksv@akadem.ru>
[1]Сибирский федеральный университет,
660041, Российская Федерация, г. Красноярск, пр. Свободный, 79

**Аннотация**. В работе рассмотрен нетрадиционный подход к созданию параллельных программ, их анализу и трансформации с использованием языка функционально-потокового параллельного программирования, обеспечивающего написание программ без учёта ресурсных ограничений, что позволяет изначально ориентироваться на алгоритмы с максимальным параллелизмом. Разработанные инструментальные средства обеспечивают трансляцию, выполнение, отладку, оптимизацию и верификацию функционально-потоковых параллельных программ. Выполнение разработанных программ осуществляется путём «сжатия» параллелизма с учётом ограниченных ресурсов реальных вычислительных систем. Вычислительный процесс рассматривается как наложение управляющего графа, определяющего организацию вычислений, на информационный граф, описывающий функциональные особенности реализуемого алгоритма. Возможно использование различных стратегий управления путём модификации управляющих графов. Предложенные инструменты обеспечивают формирование промежуточных представлений, на основе которых возможны дальнейшие преобразования исходных программ в программы для реальных архитектур параллельных вычислительных систем.

**Ключевые слова:** архитектурно-независимое параллельное программирование; функционально-потоковое параллельное программирование; трансформация программ; средства разработки программ; информационный граф, управляющий граф.

**DOI:** 10.15514/ISPRAS-2017-29(5)-10

**Для цитирования:** Легалов А.И., Васильев В.С., Матковский И.В., Ушакова М.С. Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 165-184. DOI: 10.15514/ISPRAS-2017-29(5)-10





## 1. Введение

Параллельное программирование давно перестало использоваться только для высокопроизводительных вычислений. Вместе с тем, подходы к разработке параллельных программ мало изменились по сравнению с 80-ми годами прошлого века. Как и раньше, наиболее применяемым является написание кода с учётом особенностей целевой вычислительной системы. При переносе программы на другую архитектуру приходится либо полностью её переписывать, либо подвергать существенной модификации. Попытки перейти на нетрадиционные параллельные вычислительные системы (ПВС), как и ранее, подвергаются критике, основной идеей которой является несовместимость создаваемых для них программ с уже существующими. Хотя о такой несовместимости можно говорить практически для всех вновь разрабатываемых параллельных программ [1, 2].

Широкое применение императивного подхода вносит в разработку параллельных программ свои сложности. Программист явно или неявно формирует отношения между программными объектами, которые можно охарактеризовать следующим образом [3]:

- отношения между данными (информационные отношения), или информационный граф программы, задающий связь между обрабатываемыми данными и операциями;
- отношения управления, определяющие порядок выполнения операций и функций программы;
- отношения между вычислительными ресурсами (памятью, процессорными узлами), в которых должны выполняться операции, описываемые в программе.

В большинстве случаев разработчику программы приходится явно учитывать взаимосвязь между этими отношениями, добиваясь того, чтобы не возникали логические противоречия, ведущие к ошибочному выполнению. Для современных параллельных программ это труднодостижимо, что затрудняет отладку и верификацию, несмотря на применение специализированных инструментальных средств, поддерживающих, например, формальную верификацию с использованием проверки моделей (model checking) [2, 4]. При переходе на другую архитектуру весь процесс разработки и отладки практически приходится повторять заново.

Помимо этого, наличие зависимости от конкретных архитектур, сочетающих параллелизм с последовательным программированием, не способствует внедрению истинно параллельных алгоритмов на начальных стадиях их разработки. Это ведёт к искажению исходного параллелизма задачи, зависимости от ресурсных ограничений, которые при последующем распараллеливании не всегда позволяют прийти к более эффективным решениям.

Несмотря на это, ключевым направлением в параллельном программировании является разработка архитектурно-зависимых параллельных программ. Существуют различные подходы, сильно отличающиеся по идеологии поддержания параллелизма. К наиболее популярным среди них следует отнести: распараллеливание на основе передачи сообщений [5], многопоточное и многоядерное программирование для систем с общей памятью [6], применение графических ускорителей [7], а также сочетание всех трех подходов в различных комбинациях при программировании для гибридных и распределенных архитектур [8-11].

Несмотря на то, что концепция неограниченного параллелизма в настоящее время не пользуется популярностью при разработке реальных параллельных программ [12], у нее имеются определенные перспективы в качестве основы для формирования систем программирования, обеспечивающих последующую трансформацию в ресурсно-ограниченные и архитектурно-зависимые параллельные программы. Поэтому актуальной является задача создания языковых и инструментальных средств, обеспечивающих построение параллельных программ, не связанных на начальных этапах разработки с архитектурами реальных ПВС, выбор которых мог бы осуществляться уже после проведения процессов верификации, тестирования и отладки.

## 2. Особенности модели вычислений и языка функционально-потокового параллельного программирования

Основными идеями предлагаемого подхода, определяющими концепцию архитектурно-независимого параллельного программирования, являются исключение ресурсных ограничений и неявное управление вычислениями. Предполагается, что виртуальная машина, имеющаяся в распоряжении программиста, обладает неограниченными вычислительными ресурсами, а язык программирования позволяет описывать только информационные зависимости между выполняемыми функциями. Взаимодействие между функциями при этом осуществляется по готовности данных, что позволяет создавать программы с максимально возможным параллелизмом, сжатие которого к конкретным вычислительным ресурсам и условиям эксплуатации будет происходить после верификации и отладки написанных программ на уровне исходного представления. Это позволяет повысить эффективность процесса разработки параллельных программ. Например, можно создать единую библиотеку функций, адаптируемую к различным архитектурам вычислительных систем.

Для реализации системы, ориентированной на создание архитектурно-независимых параллельных программ, необходимо обеспечить соответствующую поддержку на уровне модели вычислений. Лежащая в основе





языка программирования модель функционально-потоковых параллельных вычислений [13] задается тройкой:

$$\mathbf{M = ( G, P, S_0 )},$$

где **G** - ациклический ориентированный граф, определяющий информационную структуру программы (ее информационный граф), **P** - набор правил, определяющих динамику функционирования модели (механизм формирования разметки), $\mathbf{S_0}$ - начальная разметка.

Информационный граф

$$\mathbf{G = ( V, A )},$$

где **V** - множество вершин, определяющих программо-формирующие операторы, а **A** - множество дуг, задающих пути передачи информации между ними.

Имеется только один оператор, реализующий функцию: оператор интерпретации. Он принимает два аргумента, один из которых является значением функции, а другой – исходными данными, подлежащими обработке (рис. 1).

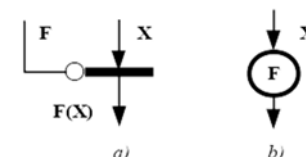


*Рис. 1. Графическое обозначение оператора интерпретации (a – общая форма, b – в случае, когда значение функции известно)*
*Fig. 1. Graphical designation of the interpretation operator (a is the general form, b is in the case when the value of the function is known)*

На выходе формируется результат. Для достижения более наглядного текстового представления программ с управлением по гот)овности данных оператор имеет префиксную и постфиксную форму записи, что позволяет выражение **F(X)** представить двумя способами:

**X:F** или **F^X**.

Функции могут являться изначально заданными элементарными операциями или создаваться программистом. Аксиоматика модели и её алгебра преобразований определяют базовые варианты реализации этого оператора.

Остальные операторы модели являются программо-формирующими, порождая связи между функциями интерпретации за счёт группировки порождаемых ими данных в различные структуры (списков), базовый набор которых представлен на рис. 2 и включает: копирование данных, формирование констант, создание списка данных из независимых величин, формирование параллельного списка, список задержанных вычислений.

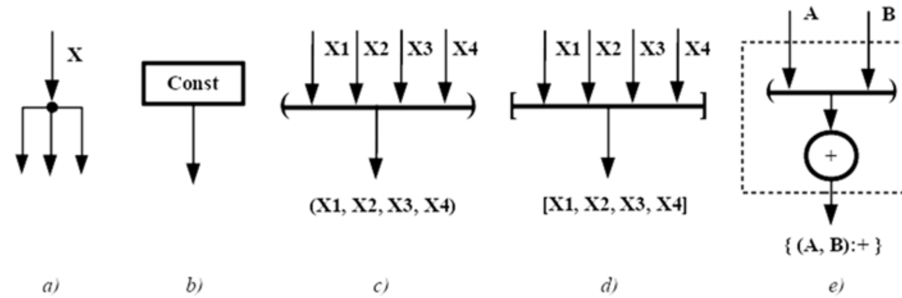


*Рис. 2. Графическое обозначение программо-формирующих операторов (a – копирование данных, b – задание константы, c – группировка в список данных, d – группировка в параллельный список, e – создание задержанного списка)*
*Fig. 2. Graphical designation of program-forming operators (a - copying data, b – constant assignment, c – grouping in the data list, d - grouping in the parallel list, e - creating the delayed list)*

**Оператор копирования** (рис. 2a) обеспечивает размножение данных. В языке определяется через именование значения и дальнейшее использование введенного обозначения в требуемых точках программы. Используются как префиксное, так и постфиксное именование связей:

**значение >> имя**, или **имя << значение**.

**Константный оператор** не имеет входов (рис. 2b). У него есть только один выход, на котором постоянно находится разметка, определяющая предписанное значение. В языковом представлении константный программо-формирующий оператор задается значением соответствующего типа.

**Оператор группировки в список данных** (рис. 2c) имеет несколько входов и один выход. Он обеспечивает структуризацию и упорядочение данных, поступающих по дугам из различных источников. Порядок элементов определяется номерами входов, каждому из которых соответствует натуральное число в диапазоне от **1** до **N**. В текстовом виде задается ограничением элементов списка круглыми скобками «**(**» и «**)**». Например:

**(X1, X2, X3, X4)**.

**Оператор создания параллельного списка** (рис. 2d) группирует элементы, аналогично списку данных. Однако на выходе предполагается наличие множественной связи, кратность которой определяется количеством входов в оператор. При выполнении оператора интерпретации осуществляется параллельная обработка всех аргументов одной функцией:

**[x, y, z]:sin ≡ [x:sin, y:sin, z:sin]**.

Аналогично при наличии списка функций все они параллельно обрабатывают один и тот же аргумент:

**(x, 0):[<, =, >] ≡ [(x, 0):<, (x, 0):=, (x, 0):>]**.





В целом алгебра преобразований языка описывает различные более общие эквивалентные преобразования параллельных списков.

**Оператор группировки в задержанный список** (рис. 2e) задается вершиной, содержащей допустимый вычисляемый подграф (находится внутри контура, выделенного пунктиром), в котором возможно несколько входов и выходов. Входы связаны с дугами, определяющими поступление аргументов, а выход определяет выдаваемый из подграфа результат (в общем случае в виде параллельного списка). Особенностью такой группировки является то, что ограниченные данным оператором вершины, не могут выполняться, даже при наличии всех аргументов. Их активизация возможна только при снятии задержки (раскрытии контура), когда ограниченный подграф становится частью всего вычисляемого графа. В языке список задержанных вычислений задается охватом соответствующих операторов фигурными скобками «{» и «}». Раскрытие происходит, когда задержанный список оказывается одним из аргументов оператора интерпретации. Использование задержанного списка определяет формирование условных ветвлений в программе.

Применение программо-формирующих операторов и алгебры преобразований заимствованы из работы [14] и сопоставимо с представленными в ней функциональными формами. Однако в данном случае именно разнообразие этих операторов является основной идеей по повышению гибкости при написании параллельных программ и реализации различных нетрадиционных идей к построению параллельных алгоритмов.

В качестве примера приведено описание функции нахождения факториала с расщеплением диапазона от 1 до **n** пополам и параллельным вычислением для левой и правой ветвей. Головная функция **parfact** осуществляет проверку начального значения аргумента для определения хода вычислений. Эта проверка на равенство нулю осуществляется операциями меньше, равно и больше одновременно в соответствии с алгеброй эквивалентных преобразований языка:

```
parfact<< funcdef n {
  selector<< (n,0):(<,=,>):?;
  selector^ (-1, 1, {(n,1):mult}):. >>return
}
```

В результате сравнения аргумента **n** с нулевым значением возвращается список данных с булевскими величинами, определяющими истинность каждого из проверяемых условий. Встроенная функция «**?**» формирует на выходе параллельный список с номерами элементов, принявшими истинные значения в ходе проведенной проверки. В данном случае это будет одно из трех значений, которое и выступает в качестве селектора одного из вариантов дальнейшего выполнения данной функции. Например, при **n = 5** формирование селектора будет осуществляться следующим образом:

```
(5,0):(<,=,>):? => (false, false, true):? => 3
```

В операции интерпретации сформированное целочисленное значение выступает в роли индекса, по которому из списка данных выбирается соответствующий элемент. При значении **n > 0** из списка с альтернативными параметрами выбирается задержанный список, который раскрывается оператором интерпретации, содержащим в качестве функции пустое значение, обозначенное точкой:

```
{(n,1):mult}:. => (n,1):mult
```

После раскрытия задержанного списка запускается функция mult, вычисляющая произведение чисел от 1 до **n** с использованием рекурсивно-параллельного алгоритма, расщепляющего диапазон до элементарных операций умножения с последующей сверткой частичных произведений:

```
mult<< funcdef pair {
  selector<< pair:(<,=,>):?;
  selector^ (
    -1,
    {pair:1},
    {([(pair:1,((pair:+,2):div>>half,1):+),
       (half, pair:2)
      ]:mult):*
    }
  ):. >>return
}
```

Первоначально, как и в предшествующей функции, осуществляется проверка диапазона. Сформированный селектор позволяет выбрать из списка данных одно из трех значений: -1, если диапазон задан некорректно; одно значение, если оба числа равны; задержанный список, осуществляющий дробление диапазона пополам и одновременное рекурсивное вычисление произведения для полученных интервалов. Встроенная функция **div** обеспечивает при этом целочисленное деление, позволяя тем самым найти середину диапазона. Информационный граф данной функции представлен на рис. 3.

## *3. Инструментальная поддержка функционально-потокового параллельного программирования*

Для использования подхода разработан ряд инструментальных средств, обеспечивающих поддержку функционально-потоковой парадигмы параллельного программирования. Разработанный язык, обеспечивает представление параллелизма на уровне элементарных операций, при котором каждая функция описывает только информационный граф алгоритма без каких-либо управляющих связей. Транслятор преобразует исходный текст функции в промежуточное представление, которое используется для оптимизации существующих зависимостей по различным критериям, а также для построения на его основе управляющего графа, задающего порядок выполнения в соответствии с выбранной стратегией управления вычислениями [3].





Трансформация управляющего графа и его оптимизация позволяют получать стратегии, отличающиеся от управления по готовности данных и учитывающие различные ограничения, свойственные, например, реальным вычислительным системам.

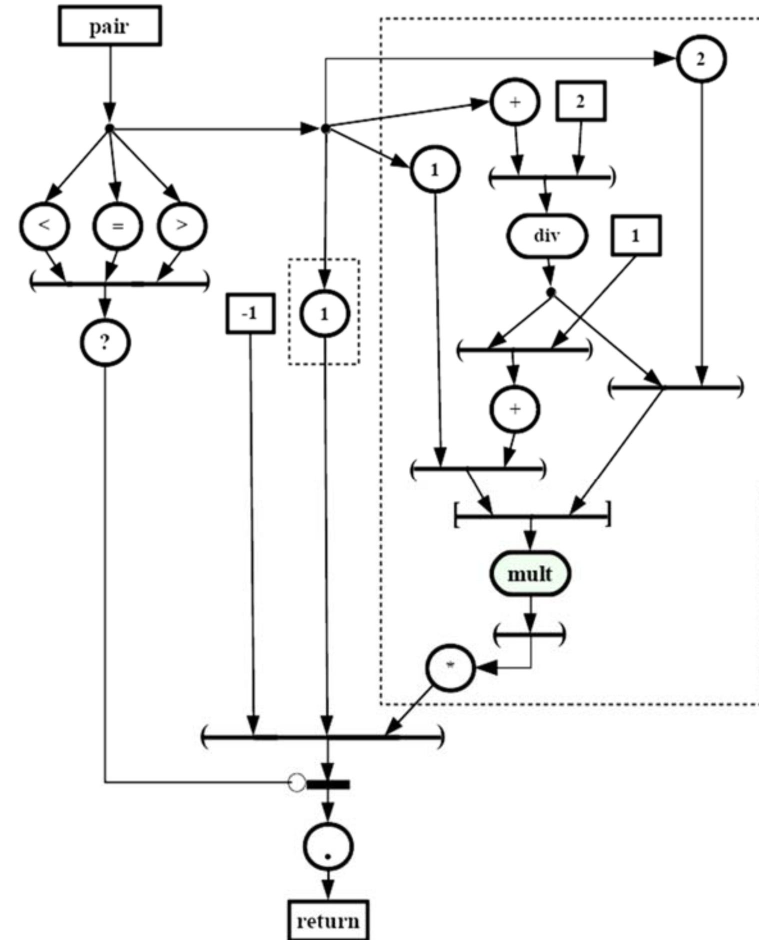


*Рис. 3. Графическое представление функции перемножения в заданном диапазоне положительных чисел*
*Fig. 3. Graphical representation of multiply function for given range of positive numbers*

Общая схема, отображающая различные варианты использования предлагаемых инструментальных средств, приведена на рис. 4. В рамках создаваемой среды выделяются следующие подсистемы:

- транслятор с языка функционально-потокового параллельного

программирования в промежуточное представление, называемое реверсивным информационным графом (РИГ);

- генератор управляющего графа (УГ), формирующий граф управления вычислениями;
- событийная машина, обеспечивающая выполнение функционально-потоковых параллельных (ФПП) программ в автоматическом и отладочном режимах, использующая в качестве программы РИГ и УГ;
- средства оптимизации реверсивного информационного графа;
- средства оптимизации управляющего графа;
- средства формальной верификации ФПП программ;
- средства преобразования ФПП программ в программы для других архитектур ПВС.

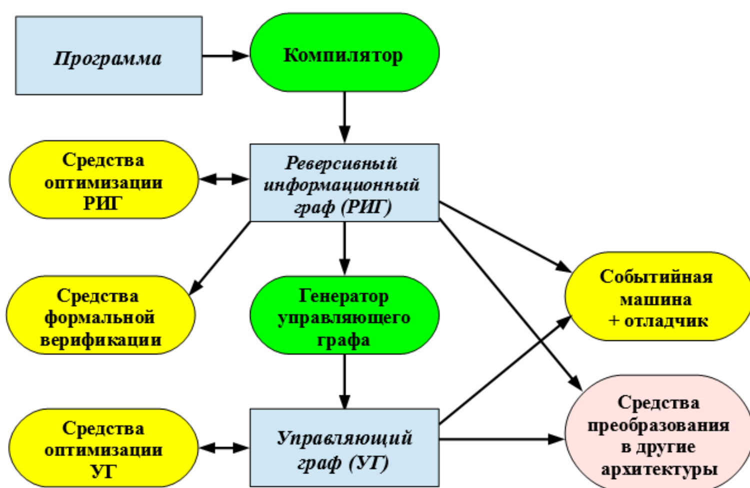


*Рис. 4. Состав инструментальных средств, поддерживающих функционально-потоковое параллельное программирование*
*Fig. 4. System of tools for functional-dataflow parallel programming support*

## 4. Трансляция функционально-потоковых параллельных программ

Транслятор ориентирован на обработку текстовых файлов, каждый из которых может содержать множество функций, написанных на языке ФПП программирования «Пифагор» [13]. Для каждой функции в памяти компьютера порождается РИГ, который сохраняется в репозитории функций в текстовой





форме. Выбор текстового представления для описания РИГ обусловлен тем, что формирование на его основе внутреннего представления в памяти компьютерной системы может быть легко выполнено с помощью простых транслирующих программ. Помимо этого разработчик может легко читать и анализировать оттранслированные функции, рассматривая данную форму как аналог языка ассемблера.

Формируемое для каждой функции в процессе синтаксического анализа промежуточное представление определяет свой РИГ как набор взаимосвязанных структур данных. Перед завершением трансляции эти представления сохраняются в текстовых файлах и могут использоваться внешними программами. В частности, функция вычисления факториала **parfact** преобразуется в следующее представление РИГ:

```
External
    0   parfact
    1   mult
Local
    0   0
    1   -1
    2   1
    3   1
    4   {1}7
id  delay   operation   links
0   0       arg
1   0       (---)       0 loc:0
2   0       (---)       < = >
3   0       :           1 2
4   0       :           3 ?
5   1       (---)       0 loc:3
6   1       :           5 ext:1
7   1       [---]       6
8   0       (---)       loc:1 loc:2 loc:4
9   0       :           8 4
10  0       :           9 .
11  0       return      10
```

При наличии внешних функций и констант данное представление содержит на них ссылки (ссылка на функцию **mult**). Помимо этого в нем задаются внутренние константы, а также реверсивный информационный граф заданной функции. Область описаний внешних ссылок начинается с ключевого слова ***External*** и содержит список строк, в каждой из которых задан дескриптор (номер) внешней ссылки и её имя. На нулевой позиции всегда располагается ссылка на транслируемую функцию. Это позволяет использовать обращение функции к самой себе в случае рекурсивных вызовов.

Область локальных констант начинается с ключевого слова ***Local*** и содержит список констант, используемых в ходе выполнения функции. Каждой константе соответствует свой дескриптор (номер), который при выполнении программы

обеспечивает доступ к соответствующим данным. Наряду с числовыми и символьными данными в этой области хранятся константы, определяющие параметры задержанных списков. Каждая из таких констант хранит номер задержанного списка, а также дескриптор узла РИГ, возвращающего вычисленное значение из данного задержанного списка (подобный узел существует в любом задержанном списке).

Описание РИГ содержит список вершин, их связей с локальными константами и внешними ссылками. Данная область начинается с заголовка, описывающего содержание каждой строки:

***id delay operation links***

Столбец ***id*** задаёт дескриптор (номер) вершины РИГ. В столбце ***delay*** указывается номер задержанного списка, в котором расположена соответствующая вершина. Если вершина РИГ не находится в задержанном списке, в данный столбец заносится ноль. В столбце ***operation*** размещается операция, выполняемая в соответствующей вершине РИГ. Столбец ***links*** указывает на связи вершин. В нем задаётся список ссылок на источники данных для текущей вершины. Источниками информации могут быть: внешние ссылки, локальные константы, предопределённые символы и узлы РИГ. Каждый из этих источников данных идентифицируется в качестве элемента списка связей с использованием своего варианта представления. Различие в описании связей в дальнейшем определяет обращение к разным областям памяти в вычислителе.

Указание на внешнюю ссылку задаётся в следующем формате:

***ext:<символическое_имя_внешней_ссылки>***

Для локальной константы используется формат:

***loc:<значение_константы>***

Предопределённые символы, обычно связанные со знаками различных операций, задаются своими значениями, например: +, -, * и так далее. Если источником операндов служит другая вершина РИГ, то в качестве связи задаётся целое число, равное дескриптору этой вершины.

Помимо этого в дополнительных файлах содержится информация для отладки программы, связывающая узлы РИГ с исходным текстом функции. Также, при наличии типов формируется файл, в котором описываются типы данных, порождаемые в каждом из узлов РИГ.

## 5. Формирование управляющего графа

Реверсивный информационный граф, порождённый транслятором, позволяет построить управляющий граф, определяющий выполнение функции. Для этого предназначена специальная утилита, которая формирует УГ, задающий управление вершинами РИГ по готовности данных. УГ сохраняется в репозитории в текстовом виде. Для ранее приведённой функции вычисления факториала он будет выглядеть следующим образом:





```
parfact
id  delay   automat     inode   links
0   0       arg,0       0       links:1,1;5,1;
1   0       (---),0     1       links:3,1;
2   0       (---),0     2       links:3,2;
3   0       :,0         3       links:4,1;
4   0       :,0         4       links:9,2;
5   1       (---),0     5       links:6,1;
6   1       :,0         6       links:7,1;
7   1       [---],0     7
8   0       (---),0     8       links:9,1;
9   0       :,0         9       links:10,1;
10  0       :,0         10      links:11,1;
11  0       return,0    11
Signals: (Number, Node, Input)
0   1   2
1   2   1
2   2   2
3   2   3
4   4   2
5   5   2
6   6   2
7   8   1
8   8   2
9   8   3
10  10  2
Dynamic links: (Number, Delay list, Node)
0   1   7
```

Первая строка формируемого представления служит для идентификации УГ. В ней задаётся имя выполняемой функции. Следующая строка носит справочный характер, описывая названия столбцов управляющего графа, узлы которого располагаются в виде списка строк непосредственно за ней. Столбец ***id*** используется для задания дескриптора (номера) узла УГ. В столбце ***delay*** указывается номер задержанного списка, в который входит тот или иной узел ИГ. Если узел не входит ни в один из имеющихся задержанных списков, то в данном поле стоит значение, равное нулю. В столбце ***automat*** задаётся тип автомата, используемого для управления узлом, и через запятую указывается его начальное состояние. Обычно при формировании УГ автоматы находятся в начальном (нулевом) состоянии. Их состояния могут изменяться в ходе оптимизации управляющего графа с использованием соответствующих утилит. В столбце ***inode*** указывается узел РИГ, непосредственно связанный с данной вершиной УГ. При первоначальной генерации УГ количество его вершин совпадает с числом вершин РИГ, и вершины обоих графов находятся в однозначном соответствии. Однако в ходе оптимизации УГ, возможна модификация этого соответствия. Может измениться как состояние автомата, размещённого в вершине УГ, так и его тип. Столбец ***links*** указывает на список управляющих связей, направленных от источников управляющих сигналов к их

приёмникам. В отличие от РИГ, в УГ направление связей определяется таким образом, чтобы каждый источник данных инициировал выполнение процессов в связанных с ним приёмниках. Описание каждой связи управляющего графа содержит номер узла-приёмника и номер входа в нем.

Область управляющих сигналов следует после описания вершин УГ, отделяясь от него заголовком:

***Signals: (Number, Node, Input)***

Представленные в УГ сигналы обеспечивают начальную инициализацию вычислений, определяя готовность к использованию констант РИГ. Именно эти сигналы, наряду с сигналом, информирующим о появлении аргумента функции, определяют начало вычислений. Каждый сигнал имеет свой дескриптор (номер) и задаёт узел-приёмник, а также номер входа в этом узле.

Для перемещения между узлами РИГ в ходе вычислений нераскрытых задержанных списков необходима дополнительная управляющая информация. Она задаётся в области задержанных связей. Описание каждой связи содержит номер задержанного списка и узел РИГ, являющийся источником результата, порождаемого раскрытым задержанным списком. Данная область начинается с заголовка:

***Dynamic links: (Number, Delay list, Node)***

## 6. Параллельная событийная машина

Выполнение функционально-потоковых параллельных программ на текущем этапе осуществляется специальным интерпретатором (событийной машиной), состоящим из множества событийных процессоров (СП), управляемых менеджером событийной машины. Каждый из этих процессоров (рис. 5) осуществляет обработку только одной функции, запускаемой в отдельном потоке. Выполнение операций внутри функции в настоящий момент осуществляется последовательно за счёт изменения состояния вершин УГ, которые инициируют вычисления в вершинах РИГ.

Функционирование СП осуществляется следующим образом. Исходные сигналы, фиксирующие протекание в системе различных событий и определяемые начальной разметкой УГ, загружаются в очередь, из которой передаются обработчику в соответствии с дисциплиной обслуживания. В простейшем случае это может быть дисциплина FIFO. Обработчик управляющих сигналов анализирует поступившее событие и выбирает указанный в нем узел управляющего графа. На основе анализа состояния узла УГ он может обратиться к связанной с ним вершине информационного графа за кодом выполняемой операции. В случае, когда операция обработки данных должна быть выполнена, происходит обращение к обработчику узлов РИГ, который осуществляет требуемые функциональные преобразования и сохраняет промежуточные результаты. После обработки данных управляющий узел переходит в новое состояние и при необходимости формирует сигнал,





передаваемый следующему узлу, который поступает в очередь управляющих сигналов.

*Рис. 5. Обобщённая структура событийного процессора*
*Fig. 5. Event processor's general structure*

Перед запуском событийной машины производится компоновка программы из отдельных функций, размещённых в репозитории. Компоновщик проверяет наличие всех элементов, перечисленных в разделе внешних ссылок РИГ. В том случае, если какой-то из этих элементов отсутствует, интерпретация объявляется невозможной. Для каждого из найденных элементов также производится компоновка. Все найденные в процессе компоновки РИГ и УГ сохраняются в памяти, откуда легко могут быть извлечены, при необходимости, по имени. Таблицу загруженных компоновщиком РИГ и УГ хранит менеджер событийной машины. Процесс интерпретации начинается с создания первого (начального) СП. Ему передаются ссылки на РИГ и УГ той функции, с которой выполнение должно начаться. Процессор сохраняет данные в рабочей памяти узлов РИГ, а состояния автоматов в рабочей памяти узлов УГ, после чего начинает анализ управляющего графа. Автоматы, связанные с вершинами УГ, которым в РИГ соответствуют константы, будут изначально находиться в состоянии порождения новых событий. Эти события, следуя по связям УГ, активируют принимающие их автоматы. Процесс передачи событий по связям продолжается до тех пор, пока не будет обработана вершина, соответствующая в РИГ вершине «***return***» (в этом случае функция считается выполненной), или

пока очередь событий не опустеет. В последнем случае СП перейдёт в спящий режим, предварительно оповестив об этом менеджер событийной машины. Данная ситуация возможна, когда обработаны все внутренние сигналы и не пришли управляющие сигналы, сигнализирующие о получении результатов в вызываемых функциях.

## 7. Оптимизация функционально-потоковых параллельных программ

Основные методы оптимизации, разработанные в настоящее время, затрагивают преобразование промежуточных представлений функционально-потоковых параллельных программ. Они направлены на изменение информационного и управляющего графов. Проводимые преобразования во многом аналогичны методам, используемым при оптимизации исходных текстов программ и их промежуточных представлений в других языках программирования, и предназначены для решения схожих задач. Специфика функционально-потоковой модели параллельных вычислений накладывает свои особенности на реализацию этих методов. Она обусловлена алгеброй эквивалентных преобразований модели, реализованной в языке: информационный и управляющий графы могут изменяться независимо друг от друга. В ходе оптимизации необходимо обеспечивать согласованность РИГ и УГ, однако, для многих задач достаточно обработки РИГ. В таких случаях оптимизация управляющего графа должна осуществляться после трансформации информационного графа и построении на его основе нового УГ. Следует отметить, что разрабатываемые в настоящее время утилиты не затрагивают распределение реальных вычислительных ресурсов. В настоящий момент реализованы следующие методы.

1. Удаление неиспользуемого кода, то есть если вычисления не влияют на получение результата, то их можно удалить из функции. Соответствующая утилита проходит по реверсивному информационному графу от вершины возврата из функции и собирает все используемые вычисления, остальные — удаляются. Специфика данного преобразования определяется особенностью функционального языка, в котором каждая функция должна завершаться возвратом результатов.

2. Оптимизация в повторяющихся вычислениях. Традиционно в компиляторах подобное преобразование выполняется для циклов. В рассматриваемом случае аналогичным образом осуществляется преобразование рекурсии и имеющихся в модели и языке параллельных списков. В частности:

- при оптимизации в рекурсивной функции, вычисления, не зависящие по данным от изменяемых параметров рекурсии, выносятся во вспомогательную функцию, а результат передаётся в основную функцию в качестве дополнительного параметра;
- при оптимизации параллельных списков, анализируется функция, применяемая к списку, и из неё в вызывающую функцию выносятся вычисления, не зависящие от аргумента [15].





3. Непосредственная (inline) подстановка простых функций. Если функция достаточно мала, то доля полезных вычислений в ней может быть невелика относительно издержек на вызов функции. В связи с этим тело функции вставляется вместо ее вызова.

4. Удаление повторяющегося кода. Если подграфы информационного графа выполняют одинаковые действия над одними и теми же аргументами и при этом находятся в одном или иерархически вложенных задержанных списках, то их можно слить, удалив тем самым избыточные вычисления. В реализации этого преобразования из РИГ сначала устраняются повторяющиеся константы. Тогда, вершины являются «повторяющимися» только в том случае, если выполняют одну и ту же операцию над одними и теми же узлами.

5. Оптимизация на основе эквивалентных преобразований, определяемых алгеброй преобразований модели функционально-потоковых параллельных вычислений. В РИГ выполняется поиск конструкций определённого вида и их трансформация в эквивалентную, но более простую для вычисления форму. В частности, модель допускает следующие эквивалентные преобразования:

- упрощение одноэлементного параллельного списка;
- свертку в один непосредственно вложенных параллельных списков;
- предварительное упрощение для списков, размер которых известен при компиляции.

6. Удаление избыточных управляющих зависимостей. Реверсивный информационный граф описывает зависимости по данным, а управляющий строится на его основе в соответствии со стратегией управления по готовности данных. Однако в ряде случаев часть построенных управляющих связей является избыточной. Поэтому их удаление не скажется на порядке выполнения программы.

7. Порождение управляющего графа, реализующего последовательный обход вершин РИГ. В данном случае осуществляется формирование вершин управляющего графа обеспечивающих запуск вершин информационного графа без дополнительного анализа готовности данных.

## 8. Формальная верификация ФПП программ

Наличие в программе только информационных зависимостей и отсутствие ресурсных ограничений позволяют облегчить формальную верификацию. Основными задачами в данном направлении работ являются: исследование специфики применения формальных методов доказательства корректности; разработка инструментальных средств, упрощающих верификацию.

Акцент сделан на доказательство корректности программы с использованием дедуктивного анализа на основе исчисления Хоара [16]. Тройка Хоара представляется как информационный граф программы, к входной и выходной дуге которого привязаны формулы на языке спецификации (предусловие и постусловие). Процесс доказательства корректности программы заключается в разметке дуг информационного графа формулами на языке спецификации, в модификации графа и его свёртке. В результате, получается несколько информационных графов, у которых все дуги размечены. Каждый из полностью размеченных графов может быть преобразован в формулу на языке логики. Тождественная истинность всех полученных формул свидетельствует о корректности программы [17].

Процесс доказательства достаточно трудоёмок, так как требует рассмотрения большого количества различных вариантов графов и преобразований. Поэтому разработаны основные концепции архитектуры инструментального средства для поддержки формальной верификации программ на языке ФПП программирования [18]. Система получает на вход информационный граф программы и формулы предусловия и постусловия на языке спецификации. Она находит неразмеченные дуги графа и помогает с выбором аксиом и теорем, необходимых для их разметки. Весь процесс доказательства представляется в виде дерева, каждый узел которого является частично размеченным графом. Построение дерева завершается, когда все его листья содержат полностью размеченные информационные графы программы. После этого для каждого графа из листа генерируется формула на языке логики. Если все формулы тождественно истинны, то программа корректна.

## 9. Заключение

Разработанные инструментальные средства позволяют создавать архитектурно-независимые параллельные программы, на которые впоследствии можно накладывать различные стратегии управления вычислениями без изменения логики функционирования. Ничто не мешает проводить предварительную оптимизацию информационного графа, осуществлять тестирование и отладку. Последующие преобразования промежуточных представлений в программы для реальных вычислительных ресурсов могут проводиться на уже отлаженном коде с применением формализованных подходов, что позволит повысить надёжность программ. Можно также проводить дополнительные оптимизации, например, повышение эффективности использования памяти. Но все это будет делаться после того, как основная проблема разработки, ручное создание корректно функционирующей программы, уже решена. Вместе с тем следует отметить, что в настоящий момент решена только первая часть задачи, связанная с выполнением программ на эмуляторе событийной машины. Переход к





преобразованию на реальные вычислительные системы планируется на следующем этапе работы.

Исследование выполняется при финансовой поддержке РФФИ в рамках научного проекта № 17-07-00288.

## Литература


[1]. Matloff Norman. Parallel Computing for Data Science With Examples in R, C++ and CUDA. CRC Press. Taylor & Francis Group, 2016.

[2]. McKenney Paul E. Is Parallel Programming Hard, And, If So, What Can You Do About It? 2014.

[3]. Легалов А.И. Об управлении вычислениями в параллельных системах и языках программирования. Научный вестник НГТУ, № 3 (18), 2004, стр. 63-72.

[4]. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010, 560 с.

[5]. Корнеев В.Д. Параллельное программирование в MPI. Новосибирск, Изд-во ИВМиМГ СО РАН, 2002, 215 с.

[6]. Эхтер Ш., Роберте Дж. Многоядерное программирование. СПб., Питер, 2010, 316 с.

[7]. Cheng John, Grossman Max, McKercher Ty. Professional CUDA ® C Programming. John Wiley & Sons, Inc., Indianapolis, Indiana, 2014.

[8]. Tay Raymond. OpenCL Parallel Programming Development Cookbook. Published by Packt Publishing Ltd., 2013

[9]. Lastovetsky Alexey L. Parallel Computing on Heterogeneous Networks. John Willey & Sons, 2003

[10]. Grid Computing – Technology and Applications, Widespread Coverage and New Horizons. Edited by Soha Maad. Published by InTech Janeza Trdine 9, 51000 Rijeka, Croatia, 2012.

[11]. Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa. Heterogeneous Computing with OpenCL. Advanced Micro Devices, Inc. Published by Elsevier Inc. 2013

[12]. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб., БХВПетербург, 2002, 608 с.

[13]. А.И. Легалов. Функциональный язык для создания архитектурно-независимых параллельных программ. Вычислительные технологии, № 1 (10), 2005, стр. 71-89

[14]. Backus J. Can programming be liberated from von Neuman style? A functional stile and its algebra of programs CACM. 1978, Vol. 21, N 8, pp. 613–641

[15]. Васильев В.С., Рыженко И.Н., Матковский И.В. Оптимизация параллельных списков функционально-потокового языка программирования «Пифагор». Системы. Методы. Технологии. № 3 (23), 2014, стр. 102-107.

[16]. Hoare, C. A. R. An axiomatic basis for computer programming. Communications of the ACM, 1969, Vol. 10, No. 12, pp. 576–585. DOI: 10.1145/363235.363259

[17]. Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. – St. Petersburg, Russia. Lecture Notes in Computer Science 7979, Springer, 2013, pp. 80-89

[18]. Ushakova M.S., Legalov A.I. Automation of Formal Verification of Programs in the Pifagor Language. Modeling and Analysis of Information Systems, 2015, 22(4) pp. 578-589. DOI:10.18255/1818-1015-2015-4-578-589

# Support tools for creation and transformation of functional-dataflow parallel programs

*A.I.Legalov <legalov@mail.ru>*
*V.S.Vasilyev <rrrFer@mail.ru>*
*I.V.Matkovskii <alpha900i@mail.ru>*
*M.S.Ushakova <ksv@akadem.ru>*
*Siberian Federal Univerisity,*
*660041, Russian Federation, Krasnoyarsk, 79 Svobodniy pr.*

**Abstract.** In the article, a novel approach to the development, analysis and transformation of parallel programs is considered. A functional dataflow parallel programming language is used. It supports writing programs independently of any resource limitations. This allows to develop algorithms with maximal level of parallelism. Support tools for translation, execution, debugging, optimization and verification of functional dataflow parallel programs are developed. The translator transforms source code of a program to an intermediate representation, in which each function is defined by its dataflow graph. A dataflow graph describes data dependencies in the function and allows to construct the control graph, which defines the organization of computations by specifying the order of the operators execution. The optimization and verification of the program is carried out on their dataflow and control graphs. In order to execute a program the maximal parallelism is to be «compressed» according to particular computing systems' resource limitations. A computation process is considered as a juxtaposition of the control graph and the dataflow graph. It is possible to employ different control strategies by means of control graphs modification. The developed support tools allow to change computation control strategies adapting them to the peculiarities of a computational environment. The suggested tools provide generation of intermediate representation, which could be used as a basis for the following transformations of a program to the program for existed parallel computing systems architecture.

**Keywords:** architecture-independent parallel programming; functional-dataflow parallel programming; transformation of programs; software development tools, dataflow graph, control graph

**DOI:** 10.15514/ISPRAS-2017-29(5)-10

**For citation:** Legalov A.I., Vasilyev V.S., Matkovskii I.V., Ushakova M.S. Support tools for creation and transformation of functional-dataflow parallel programs. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 5, 2017. pp. 165-184 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-10





## References

[1]. Matloff Norman. Parallel Computing for Data Science With Examples in R, C++ and CUDA. CRC Press. Taylor & Francis Group, 2016.
[2]. McKenney Paul E. Is Parallel Programming Hard, And, If So, What Can You Do About It? - 2014.
[3]. Legalov A.I. About computation control in parallel system and programming languages. Nauchiy Vestnik NGTU [Scientific Bulletin of NSTU], № 3 (18), 2004, pp. 63-72 (in Russian).
[4]. Karpov Y.G. Model Checking. Parallel and distributed program system verification. - SPb., BHV-Petersburg, 2010. - 560 p. (in Russian).
[5]. Korneev V.D. Parallel Programming in MPI. Novosibirsk, ICMMG SB RAS, 2002, 215 p. (in Russian).
[6]. Shameem Akhter, Jason Roberts. Multi-Core Programming. Intel Press, 2006.
[7]. Cheng John, Grossman Max, McKercher Ty. Professional CUDA ® C Programming. John Wiley & Sons, Inc., Indianapolis, Indiana, 2014
[8]. Tay Raymond. OpenCL Parallel Programming Development Cookbook. Published by Packt Publishing Ltd., 2013
[9]. Lastovetsky Alexey L. Parallel Computing on Heterogeneous Networks. John Willey & Sons, 2003
[10]. Grid Computing – Technology and Applications, Widespread Coverage and New Horizons. Edited by Soha Maad. Published by InTech Janeza Trdine 9, 51000 Rijeka, Croatia, 2012.
[11]. Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa. Heterogeneous Computing with OpenCL. Advanced Micro Devices, Inc. Published by Elsevier Inc., 2013
[12]. Voevodin V.V., Voevodin Vl.V. Parallel computing. SPb., BHV-Petersburg, 2002, 608 p. (in Russian).
[13]. A.I.Legalov Functional language for architecture-independent programming] Vichislitelnye Technologii [Computing technologies], № 1 (10), 2005, pp. 71-89 (in Russian).
[14]. Backus J. Can programming be liberated from von Neuman style? A functional stile and its algebra of programs CACM. 1978, Vol. 21, N 8, pp. 613–641.
[15]. Vasilyev V.S., Ryzhenko I.N., Matkovskii I.V. Parallel list optimization for function dataflow programming language “Pifagor”. Systemy. Methody. Technologii [Systems. Methods. Technologies]. № 3 (23), 2014, pp. 102-107. (in Russian).
[16]. Hoare, C. A. R. An axiomatic basis for computer programming. Communications of the ACM, 1969, Vol. 10, No. 12, pp. 576–585. DOI: 10.1145/363235.363259
[17]. Kropacheva M., Legalov A. Formal Verification of Programs in the Pifagor Language. Parallel Computing Technologies, 12th International Confernce PACT September-October, 2013. St. Petersburg, Russia. Lecture Notes in Computer Science 7979, Springer, 2013, pp. 80-89.
[18]. Ushakova M.S., Legalov A.I. Automation of Formal Verification of Programs in the Pifagor Language. Modeling and Analysis of Information Systems, 2015, 22(4), pp. 578-589. DOI:10.18255/1818-1015-2015-4-578-589