ФУНКЦИОНАЛЬНЫЙ ЯЗЫК ДЛЯ СОЗДАНИЯ АРХИТЕКТУРНО-НЕЗАВИСИМЫХ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ*

А. И. Легалов
Красноярский государственный технический университет, Россия
e-mail: `legalov@mail.ru`

A new functional parallel programming language is suggested. It supports evaluation and control of data flow without limitation of computer resources. Specific features of the computational model, syntax and semantics of the proposed language are considered. Possibilities of an application of the proposed language in portable parallel programming are described.

Введение

Достижение мобильности (переносимости) играет важную роль в создании программ. Удовлетворение этому критерию позволяет безболезненно переносить их с одной компьютерной архитектуры на другую, поддерживая тем самым эволюционное развитие и использование. В настоящее время практически решены проблемы, связанные с переносимостью программ, написанных для фон-неймановских архитектур. При этом используются различные подходы:

1. Разработаны компиляторы и интерпретаторы с языков высокого уровня в уникальный машинный код. Переносимость обеспечивается на уровне исходных текстов программ.

2. Существуют интерпретаторы промежуточного кода, унифицирующие виртуальную машину, которая используется в качестве основного инструмента для выполнения программ.

3. Промежуточный код используется для унификации трансляторов. Но вместо его прямой интерпретации осуществляется дополнительная трансляция в машинное представление перед непосредственным исполнением.

Следует однако отметить, что при параллельном программировании использование только этих приемов не позволяет успешно и эффективно обеспечивать мобильность. Во многом это обусловливается следующим:

1. Разработка параллельных программ всегда опиралась на достижения в области технологии производства компьютерных систем. Это связано с необходимостью извлечь максимальный эффект из предоставленных вычислительных ресурсов. Вместе с тем, огра-

*Работа выполнена при поддержке Российского фонда фундаментальных исследований (грант № 02-07-90135).

© Институт вычислительных технологий Сибирского отделения Российской академии наук, 2005.

ниченность этих ресурсов привязывает принципы разработки программ к особенностям конкретной архитектуры, что, в свою очередь, ведет к необходимости полной переработки написанного кода при переносе на другие параллельные вычислительные системы (ПВС).

2. Создание и использование новых методов обработки данных тоже сильно зависит от особенностей вычислительных систем. Разработанные программы необходимо отлаживать и эксплуатировать в конкретных условиях. Поэтому изменение компьютерных архитектур ведет к тому, что методы, успешно применявшиеся раньше, оказываются совершенно неэффективными.

3. Существуют проблемы и при разработке архитектурно-независимых инструментов параллельного программирования. Во многом они обусловлены тем, что нет четкого разделения между средствами, зависимыми от архитектуры и не зависимыми от нее. Дуализм заключается в том, что любая система программирования, претендующая на независимость, обычно определяется через соответствующую виртуальную машину. А описание подобной машины является ни чем иным, как ее архитектурой.

Таким образом, разработка инструментов для переносимого параллельного программирования ведет к формированию архитектуры вычислительной системы, тем или иным образом обобщающей принципы построения программ. Основной задачей является адаптация этих программ к конкретному вычислителю, используемому в данный момент или предполагаемому к использованию в дальнейшем, после смены поколения компьютеров. Существует еще много проблем, требующих дальнейшего анализа. Наиболее популярным способом параллельного программирования продолжает оставаться написание кода с учетом особенностей конкретных архитектур. Поэтому исследование и разработка методов создания мобильных параллельных программ актуальны и по сей день.

1. Использование неявного управления вычислениями

Разработка параллельных программ, без особых проблем переносимых с одной ПВС на другую, возможна только в том случае, если язык программирования позволяет скрыть механизмы управления вычислениями, специфичные для конкретных архитектур. Особенности обычно проявляются в явном управлении различными ресурсами. Например, дополнительные воздействия со стороны программиста вносятся при распределении ячеек памяти, задании последовательности вычислений и распараллеливании по собственному усмотрению. Для избавления от подобных зависимостей была предложена концепция неограниченного параллелизма, направленная на разработку программ, не связанных с какими-либо ресурсными ограничениями, позволяющая создавать максимально параллельные программы с использованием различных языков.

Однако средства, обеспечивающие поддержку явного управления вычислениями со стороны программиста, стимулируют внесение трудно обнаруживаемых ошибок и ограничений. Поэтому были предложены языки с неявным заданием параллелизма. В этом случае достаточно указать информационную зависимость между функциями, осуществляющими преобразование данных. Использование таких языков позволяет:

— создавать программы, поддерживающие параллелизм на уровне элементарных операций, ограниченный лишь методом решения задачи;

— обеспечивать перенос на конкретную архитектуру, не распараллеливая программу, а “сжимая” ее максимальный параллелизм;

— проводить оптимизацию программы по множеству параметров с учетом архитектуры ПВС, для которой осуществляется трансляция.

В ходе исследований, постоянно проводимых в этой области уже в течение нескольких десятилетий, были разработаны различные модели параллельных вычислений и языки, использующие для описания параллелизма только информационные зависимости между выполняемыми функциями. Условно можно выделить два направления разработок:

— методов управления вычислениями по готовности данных,

— языков и методов функционального программирования.

1.1. Управление по готовности данных

Для того чтобы язык программирования мог обеспечить формирование параллелизма любого типа, он должен, на уровне элементарных операций, поддерживать механизмы размножения информационных потоков, их группирования в структуры данных различного уровня вложенности, одновременного выделения из этих структур нескольких независимых и разнородных по составу групп. Подобные проблемы решаются построением модели "идеального" вычислителя, обладающего ресурсами, мгновенно выделяемыми по первому требованию для выполнения любой операции и хранения любых промежуточных данных. Обычно такие возможности заложены в потоковых моделях вычислений (МВ) и языках, построенных на их основе.

Существуют различные МВ, поддерживающие управление по готовности данных. Многие из них описаны в обзоре [1]. Однако в большинстве случаев эти модели не обеспечивают поддержку максимального параллелизма из-за ограничений, присущих механизмам управления вычислениями.

В частности, потоковые схемы Карпа — Миллера (одна из первых моделей такого типа) используют ограниченные очереди, являющиеся сдерживающим ресурсом для независимо обрабатываемых наборов данных. Помимо этого, в них отсутствуют средства установления условий связи между данными и управлением, что не позволяет представлять произвольные вычислимые функции.

Более обобщенными являются схемы Адамса, в которых используются специальные вершины для управления вычислениями. Эта модель позволят описывать различные параллельные программы. Однако данные, как и в схемах Карпа — Миллера, выстраиваются в очереди к операционным узлам, что, в общем случае, не позволяет описывать неограниченный параллелизм. Подобные же ограничения, связанные с наличием очередей, присутствуют и в модели Дейвиса.

Наиболее проработанными из потоковых моделей являются схемы Денниса [2], которые использовались для создания ряда языков программирования и машин потока данных. Так же, как и представленные выше модели, схемы Денниса поддерживают одновременную обработку нескольких независимых наборов данных. Однако вместо очередей для их взаимной синхронизации используется механизм неявных подтверждений. Именно он и является сдерживающим фактором, не позволяющим независимым наборам обгонять друг друга.

Использование схем Родригеса тоже позволяет реализовать потоки данных. Вместе с тем, в них используются явное управление вычислениями и непосредственное обращение к ячейкам памяти, что позволяет программисту вносить изменения, искажающие информационный граф.

Для преодоления указанных ресурсных ограничений были предложены потоковые модели с раскрашенными фишками, в частности, модель Арвинда — Гостелоу. Используемая в ней дополнительная идентификация позволяет продвигать разные наборы данных независимо друг от друга. Вместе с тем следует отметить, что механизм раскраски не проработан достаточно глубоко, чтобы говорить о его пригодности для описания независимых наборов данных, порождаемых внутри циклов и рекурсивно вложенных циклов. В этих случаях, возможно, придется пользоваться механизмом глобальной раскраски фишек, являющимся сдерживающим ресурсным фактором, или иерархической раскраской, сложность организации которой нигде не рассматривается. Проблема усложняется необходимостью синхронизации взаимосвязанных раскрашенных наборов в ходе их слияния, обусловленного реализуемым алгоритмом.

1.2. Функциональное программирование

Функциональное программирование (ФП) имеет давнюю историю. Оно определяется как "способ построения программ, в которых единственным действием является вызов функции, единственным способом расчленения программы на части является введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции — оператор суперпозиции функции" (из предисловия А.П. Ершова к [3]). В ходе своего развития ФП значительно обогатилось и расширилось, предоставив в распоряжение программиста развитые средства модульности и функциональной декомпозиции. Современные функциональные языки позволяют писать мощные и в то же время компактные программы. Среди первых, обеспечивших поддержку ограниченного параллелизма, был FP [4]. Использование параллелизма было реализовано в одной из версий Haskell [5].

К общим факторам, ограничивающим параллелизм существующих функциональных языков, следует отнести:

1. Набор базовых операций и структур данных изначально ориентирован на поддержку последовательных вычислений. Например, списки состоят из "головы" и "хвоста" [3], что требует последовательной рекурсивной развертки и затрудняет одновременную обработку всех элементов. Поэтому последующие расширения таких языков могут служить только надстройкой, не поддерживающей параллелизм на уровне отдельных операций.

2. Включение механизмов поддержки параллелизма во время создания ряда функциональных языков осуществлялось без полного анализа всех возможных вариантов. Как результат, этим языкам присущи только ограниченные формы организации параллельных вычислений [6].

3. Ограничение параллелизма зависит и от текстового представления конструкций языка. В некоторых из них просто отсутствует возможность описать параллелизм задачи из-за ограничений, присущих синтаксису.

Для преодоления указанных недостатков были разработаны функциональные языки, использующие потоковые модели вычислений. К ним относится SISAL [7], предназначенный для решения научно-технических задач. Он имеет ядро, обеспечивающее поддержку управления на основе потоков данных. В языке также присутствуют средства явного управления вычислениями, что обусловлено стремлением к эффективной реализации реально разрабатываемых прикладных программ.

Ориентация на функционально-потоковые вычисления присуща и Т-системе [8], ис-

пользующей парадигму функционального программирования для обеспечения динамического распараллеливания программ. Функциональный стиль поддерживается за счет расширения языков C, C++, что в общем ведет к смешанному способу написания программ. В T-системе отсутствуют явные параллельные конструкции, а описание параллелизма осуществляется за счет использования "чистых" функций.

Язык программирования DCF [9] ориентирован на гибридную data-control flow модель вычислений. Его задачей является эффективная поддержка существующей вычислительной аппаратуры. Он тоже реализован как расширение языка программирования C, что обеспечивает задание явного и неявного параллелизма. Уровень DCF позволяет использовать его в качестве промежуточного языка представления программ при трансляции традиционных языков программирования высокого уровня, языков потока данных и декларативных языков программирования высокого уровня для dataflow-компьютеров и многопоточных архитектур, а также для непосредственного программирования.

Ориентация только на потоковые вычисления присуща языку программирования Норма [10]. В нем используется принцип единственного присваивания, а сама программа по существу является записью численного метода решения конкретной задачи. Подобный подход определяет декларативный характер языка и прямую ориентацию на решения научных задач вычислительного характера. Отсутствие ориентации на непосредственное выполнение вычислений ведет к необходимости дополнительной "раскрутки" программы в промежуточное потоковое представление, которое может иметь в основе произвольную модель вычислений. То есть данный язык не имеет прямого отношения к функциональному и потоковому программированию.

Следует отметить, что рассмотренные функционально-потоковые языки в основном ориентированы на эффективное решение конкретных прикладных задач, что ведет к использованию смешанных моделей вычислений, обеспечивающих явное управление ресурсами. Практическое отсутствие интереса к созданию "идеальных" систем, обеспечивающих разработку переносимых параллельных программ, во многом обусловливается семантическим разрывом между методами написания программ, которые должны использовать параллелизм на уровне операций, и современными техническими средствами, обеспечивающими реальную поддержку только крупноблочного распараллеливания. Эффективнее и проще осуществлять кодирование, непосредственно поддерживающее архитектурно-зависимые методы.

Вместе с тем исследования в области создания мобильных, ресурсно-независимых параллельных программ представляют интерес, так как в перспективе позволяют избежать переписывания кода при изменении архитектур вычислительных систем. Усилия должны быть направлены не на создание языка, являющегося основой машинной архитектуры, а на решение следующих задач:

— исследование и разработку систем программирования, обеспечивающих написание программ, не зависящих от архитектур конкретных ПВС;

— разработку промежуточных интерпретирующих средств, позволяющих выполнять архитектурно-независимые программы на реальных системах;

— создание методов преобразования архитектурно-независимых параллельных программ в архитектурно-зависимые.

Это позволит приступить к накоплению программного обеспечения, которое не придется переписывать заново при неизбежном изменении вычислительных систем. Достаточно будет разработать новые методы преобразования, предназначенные для "сжатия" максимального параллелизма.

1.3. Особенности поддержки неограниченного параллелизма

Несмотря на определенные достижения в области неявного управления вычислениями, проблемы, связанные с построением максимально параллельных программ, свободных от всех ресурсных ограничений, до конца не решены. Это обусловлено наличием соответствующих ограничений как в моделях вычислений, так и в языках программирования.

Для преодоления ресурсных ограничений в работе предлагаются методы построения параллельных программ с использованием функционального языка программирования, разработанного на основе МВ, использующей концепцию неограниченных ресурсов и неявное управление по готовности данных. Особенностями модели являются:

— ориентация на организацию вычислений в бесконечных ресурсах, что достигается использованием принципа единственного выполнения каждой операции в сочетании с принципом единственного присваивания;

— представление программы в виде ациклического информационного графа, с поддержкой циклических процессов на основе рекурсивной организации вычислений;

— отсутствие явно выраженных ветвлений, позволяющее рассматривать программу как безусловный граф, в ходе выполнения которого все дуги имеют разметку.

Для апробации этой модели предлагается функциональный язык параллельного программирования (ФЯПП) “Пифагор” [11–15], обладающий следующими характеристиками:

— распараллеливание программ на уровне операций;

— архитектурная независимость, достигаемая за счет описания только информационных связей;

— асинхронный параллелизм, полученный выполнением операций по готовности данных;

— отсутствие переменных, позволяющее избежать конфликтов, связанных с совместным использованием памяти параллельными процессами;

— отсутствие операторов цикла, позволяющее избежать конфликтов при использовании различными наборами данных одних и тех же фрагментов параллельной программы.

Следует отметить, что, в отличие от других языков подобного класса, ориентированных на решения конкретных задач, целью создания данного языка является практическая апробация предлагаемой модели, исследование и дальнейшее развитие методов организации ресурсно-независимых параллельных программ и их преобразования в представление, эффективно выполняемое на современных архитектурах. Поэтому в языке используется набор операций, ориентированных на обработку только базовых типов данных.

Для проверки концепций, заложенных в ФЯПП, уточнения его синтаксиса, семантики и аксиоматики были разработаны транслятор, последовательный и параллельный интерпретаторы.

2. Модель вычислений

Модель определяет концепцию языка, основные механизмы управления вычислениями и формирования структуры программы. Она представляет программу в виде информационного графа, обладающего рядом специфических особенностей:

— в основе модели лежит управление по готовности данных, определяемое для процессов, протекающих внутри бесконечных ресурсов, что позволяет описать параллелизм без ресурсных конфликтов;

— распараллеливание программ обеспечивается на уровне операций;

— выбор набора операций и аксиом, определяющих набор примитивов, ориентирован на наглядное текстовое представление информационного графа.

Базовые функции модели и языка определены как операторы. Вершины графа, соответствующие операторам, обеспечивают преобразования данных, их структуризацию и размножение. Существуют следующие типы вершин:

— оператор интерпретации;
— константный оператор;
— оператор копирования данных;
— оператор группировки в список;
— оператор группировки в параллельный список;
— оператор группировки в список задержанных вычислений.

Динамика выполнения операторов задается механизмом продвижения начальной разметки графа по дугам модели, что соответствует обработке исходных данных, получению результатов и их использованию в дальнейших вычислениях. Наличие разметок на всех дугах некоторой вершины графа позволяет запустить ее и получить выходную разметку.

Правила распространения разметки по графу складываются из общих правил межоператорных переходов, правил срабатывания операторов, правил выполнения оператора интерпретации над предопределенными функциями, правил эквивалентных преобразований операторов и связей допустимого графа (более полное описание приведено в [13]). Моделирование вычислительного процесса выполняется следующим образом:

1. Если входные дуги вершины имеют разметку, то на выходных дугах происходит формирование разметки в соответствии с правилами срабатывания вершины, определяющей оператор.

2. Если входные разметки имеют кратность, превышающую единицу, то для заданной вершины формирование выходной разметки может начинаться при неполной входной разметке независимо для каждого из сформированных наборов входных данных и осуществляется в соответствии с аксиомами срабатывания операторов.

3. В процессе распространения разметка не убирается и не замещается (обеспечивая принцип единственного выполнения операции). Каждая дуга графа может получить разметку только один раз. Из требования о недопустимости повторной разметки вытекает требование ацикличности информационного графа.

4. Процесс распространения разметки заканчивается, когда все дуги графа имеют полную разметку в соответствии с предписанной кратностью или при невозможности распространения разметки.

Поддержка разметкой дуги нескольких независимых наборов данных позволяет описывать на уровне МВ массовый параллелизм. При этом инициализация вычислений в вершине может начинаться до формирования полной разметки, так как обработка каждого из входных наборов осуществляется независимо.

2.1. Операторы

Оператор интерпретации описывает функциональное преобразование аргумента. Он имеет два входа, на которые, через информационные дуги, поступают функция F и аргумент X (рис. 1).

Аргумент и функция могут являться результатами предшествующих вычислений. Оператор запускается по готовности данных, что фиксируется появлением разметки на входных дугах. Получение результата задается разметкой выходной дуги. При текстовом опи-

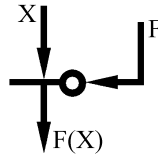

Рис. 1. Оператор интерпретации с входами аргумента X и функции F.

сании оператор интерпретации имеет две формы: постфиксную, обозначаемую знаком “:”, и префиксную, при которой функция отделяется от аргумента знаком “^”. Наличие двух способов записи одного оператора позволяет в дальнейшем комбинировать их для получения более наглядного текста программы. Следовательно, выражение F(X) оператор интерпретации задает как `X : F` или `F^X`.

Константный оператор не имеет входов (рис. 2). У него есть только один выход, на котором постоянно находится разметка, определяющая предписанное значение (константу). Множество констант информационного графа определяет внутреннюю начальную разметку. В языковом представлении константный оператор задается значением соответствующего типа.

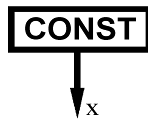

Рис. 2. Константный оператор.

Оператор копирования осуществляет передачу данных с одного своего входа на множество выходов. В текстовой форме копирование определяется через именование передаваемой величины и дальнейшее использование введенного обозначения. Используется постфиксное именование размножаемого объекта в форме: `величина >> имя`, и его префиксный эквивалент, имеющий вид: `имя << величина`. Например:

```
y << F^x; (x, y):+ >> c;
```

В графическом представлении (рис. 3) передача фиксируется установкой разметки на дугах, связанных с выходами вершины при размеченной входной дуге.

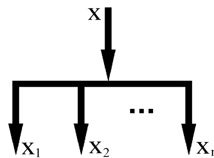

Рис. 3. Оператор копирования данных.

Оператор группировки в список (рис. 4) 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 оператор задается ограничением элементов списка круглыми скобками “(” и “)”. Например:

$$(\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4).$$

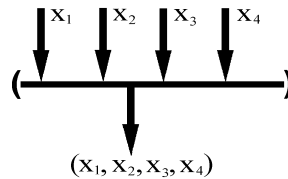

Рис. 4. Оператор группировки в список данных.

Нумерация элементов списка в данном случае задается неявно в соответствии с порядком их следования слева направо (это же соглашение предполагается и в графическом представлении при отсутствии явной нумерации входов).

Оператор создания параллельного списка (рис. 5) обеспечивает группирование элементов, аналогичное списку данных.

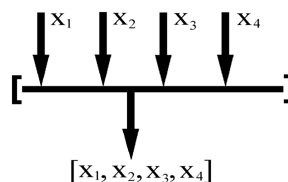

Рис. 5. Оператор группировки в параллельный список.

В текстовом виде группировка в параллельный список задается ограничением его элементов квадратными скобками “[” и “]”. Например:

$$[x_1, x_2, x_3, x_4].$$

Кратность разметки, определяющая выходной набор данных, равна сумме кратностей разметок всех входных дуг. В соответствии с алгеброй преобразований языка считается, что функция использует каждый элемент данного списка как независимый аргумент. Если же параллельный список определяет набор функций, то все они выполняются одновременно над одним и тем же аргументом. Данная конструкция обеспечивает массовый параллелизм.

Оператор группировки в задержанный список задается вершиной, содержащей подграф, в котором возможно несколько входов и выходов. Входы связаны с дугами, определяющими поступление аргументов, а выход определяет выдаваемый из подграфа результат (рис. 6).

Специфической особенностью такой группировки является то, что ограниченные оператором задержки вершины (на графе ограничение задается контуром), представляющие другие операторы, 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.

Задержанный подграф создает на своем единственном выходе константную разметку, которая является образом (иконкой) данного подграфа. Эта разметка распространяется по дугам графа от одного оператора к другому, размножаясь, входя в различные списки и выделяясь из них до тех пор, пока не поступит на один из входов оператора интерпретации. При наличии на его обоих входах готовых данных происходит подстановка, вместо иконки, ранее определенного задержанного графа с сохранением входных связей. Опоясывающий

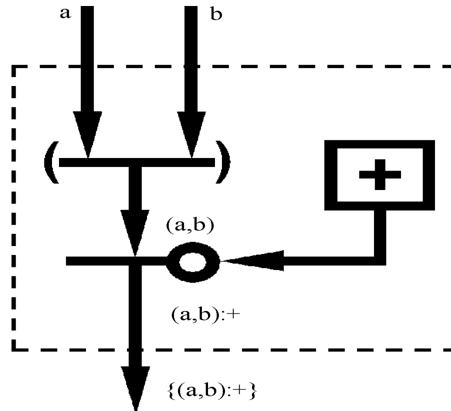

Рис. 6. Оператор группировки в параллельный список.

подграф контур оператора задержки при этом “убирается”, и происходит выполнение активированных операторов. В результате на выходной дуге раскрытого подграфа вновь формируется результирующая разметка, которая и является одним из окончательных аргументов оператора интерпретации, раскрывшего задержанный подграф. Данная процедура называется раскрытием задержанного подграфа. В текстовом виде группирование в список задержанных вычислений (для краткости будем также использовать термин “задержанный список”) задается охватом соответствующих операторов фигурными скобками “{” и “}”. Например:

$$\{\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4\} \text{ или } \mathtt{\{(a,b):+\}}.$$

Наличие этой конструкции позволяет откладывать момент начала некоторых вычислений или вообще не начинать их, что необходимо при организации выборочной обработки данных.

Правила срабатывания конкретизируют формирование разметок на выходных дугах для каждого из ранее введенных операторов.

Оператор интерпретации обеспечивает преобразование входного набора данных X, выступающего в качестве аргумента, в выходной набор Y, играющий роль результата, используя при этом входной набор F в качестве функции, определяющей алгоритм преобразования. В постфиксной нотации, выбранной для дальнейших иллюстраций, данное преобразование можно записать следующим образом:

$$\mathtt{X} : \mathtt{F} \Rightarrow \mathtt{Y}\ .$$

Можно рассмотреть множество унарных функций F, разделив его при этом на два подмножества: $\mathtt{F} = \{\mathtt{f}_1, \mathtt{f}_2\}$, где $\mathtt{f}_1$ — множество предопределенных функций языка, для каждой из которых аксиоматически задаются области определения и изменения; $\mathtt{f}_2$ — множество функций, порождаемых при программировании. Следует отметить, что выбор базового набора предопределенных функций осуществляется в некоторой степени субъективно, исходя из соображений удобства пользования разрабатываемым языком. Введены арифметические функции, функции сравнения и прочие, аналогично тому, как это сделано и в других языках программирования. Например, функция сложения двух чисел $\mathtt{x}_1$, $\mathtt{x}_2$, порождающая в качестве разметки число y, задается следующим образом:

$$(\mathtt{x}_1, \mathtt{x}_2) : + >> \mathtt{y},$$

где первый аргумент оператора интерпретации является двухэлементным списком данных. Каждый элемент этого списка должен быть числом. Второй аргумент оператора интерпретации является функцией сложения, обозначенной значком “+”. Результат функции сложения, значение $\mathtt{y}$, является атомарным элементом.

Наряду с определением функций, присущих всем языкам программирования, целесообразно определить множество функций, нестандартных в традиционном понимании. Например, целое число может непосредственно интерпретироваться как функция выбора элемента списка

$$(\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_\mathtt{i}, \ldots, \mathtt{x}_\mathtt{n}) : \mathtt{i} \Rightarrow \mathtt{x}_\mathtt{i},$$

где $\mathtt{i}$ — натуральное число, $\mathtt{x}_\mathtt{i}$ — элемент списка. Данная функция выделяет из списка данных $\mathtt{i}$-й элемент, который и определяет разметку выходной дуги.

Другой полезной предопределенной функцией является

$$(\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \ldots, \mathtt{b}_\mathtt{n}) :? \Rightarrow [\mathtt{i}_1, \mathtt{i}_2, \ldots, \mathtt{i}_\mathtt{k}] \ ,$$

где $(\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{n})$ — список булевых величин; $[\mathtt{i}_1, \ldots, \mathtt{i}_\mathtt{k}]$ — параллельный список натуральных чисел, определяющих номера тех компонент булева списка, которые имеют истинные значения. Наличие данной функции позволяет формировать условия, обеспечивающие выполнение нескольких альтернативных ветвей программы. Полный список предопределенных функций представлен в описании языка.

Наряду с определением операции интерпретации для аксиоматически определенных функций, она также определяется и для уже существующих операторов. Например, определены следующие правила раскрытия задержанного списка:

$$\{\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}\} : \mathtt{f} \Rightarrow [\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}] : \mathtt{f}, \tag{1}$$

$$\mathtt{x} : \{\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{k}\} \Rightarrow \mathtt{x} : [\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{k}], \tag{2}$$

$$\{\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}\} : \{\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{k}\} \Rightarrow [\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}] : [\mathtt{f}_1, \ldots, \mathtt{f}_\mathtt{k}]. \tag{3}$$

Выражение (1) показывает, что, при наличии разметки на дуге, определяющей вход f, задержанный список данных $\{\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}\}$ преобразуется в параллельный. Далее, если $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n}$ являются допустимыми подграфами, следует получение их значений, после чего осуществляется выполнение заданного оператора интерпретации. Выражение (2) описывает аналогичное раскрытие задержанного списка функций при появлении разметки на входе, определяющем $\mathtt{x}$. Если же оба аргумента оператора интерпретации являются задержанными списками (3), то они вначале воспринимаются как константные значения, что определяет их немедленное преобразование в параллельные списки. После этого каждый список вычисляется, что приводит к разметке входных дуг описанного оператора интерпретации и его выполнению. Следует заметить, что вид операторов интерпретации, приведенный в выражениях (1)–(3), не является окончательным. Необходимо еще провести дополнительное приведение к элементарным вычислительным действиям в соответствии с правилами эквивалентных преобразований.

Правила эквивалентных преобразований определяют алгебру модели и языка программирования. Они позволяют осуществить трансформацию графа, обеспечивающую сведение сложных структурированных операций к набору элементарных действий над предопределенными компонентами. Возможна также обратная структуризация элементарных действий, полезная при адаптации полученной функциональной параллельной программы

к архитектуре конкретной ПВС. Эквивалентные преобразования определены на множестве операторов языка и отражают общие алгебраические свойства модели. Их проведение может происходить как перед началом вычислений, когда они применяются к исходному информационному графу, так и непосредственно в ходе выполнения программы. В этом случае преобразования проходят на частично размеченном графе.

Из всего разнообразия эквивалентных преобразований, определенных в описании модели вычислений [11], остановимся на обработке параллельных списков, которая сводится к формированию множества отдельных операторов интерпретации. Параллельный список функций над некоторым аргументом эквивалентен параллельному списку операций интерпретации для каждой из функций над одним и тем же аргументом:

$$\mathtt{X} : [\mathtt{f}_1, \mathtt{f}_2, \ldots, \mathtt{f}_\mathtt{n}] \equiv [\mathtt{X} : \mathtt{f}_1, \mathtt{X} : \mathtt{f}_2, \ldots, \mathtt{X} : \mathtt{f}_\mathtt{n}].$$

Функция над параллельным списком аргументов эквивалентна параллельному списку операций интерпретации для этой функции над каждым из аргументов:

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_\mathtt{n}] : \mathtt{f} \equiv [\mathtt{x}_1 : \mathtt{f}, \mathtt{x}_2 : \mathtt{f}, \ldots, \mathtt{x}_\mathtt{n} : \mathtt{f}].$$

Эти выражения показывают, каким образом кратную разметку некоторой дуги можно представить эквивалентной одноэлементной разметкой множества дуг. В более общем случае, когда разметка с кратностью больше единицы имеется на обеих входных дугах оператора интерпретации (т. е. как для функций, так и для аргументов), на выходе формируется параллельный список кратностью, равной произведению кратностей разметок его входных дуг. Порядок следования элементов сформированного параллельного списка определен таким образом, что данные выступают в качестве приоритетного параметра независимо от префиксной или постфиксной формы записи:

$$\begin{gathered}[\mathtt{x}_1, \mathtt{x}_2, ..., \mathtt{x}_\mathtt{n}] : [\mathtt{f}_1, \mathtt{f}_2, ..., \mathtt{f}_\mathtt{m}] \Rightarrow \\ \Rightarrow [\mathtt{x}_1 : \mathtt{f}_1, ..., \mathtt{x}_1 : \mathtt{f}_\mathtt{m}, \mathtt{x}_2 : \mathtt{f}_1, ..., \mathtt{x}_2 : \mathtt{f}_\mathtt{m}, ..., \mathtt{x}_\mathtt{n} : \mathtt{f}_1, ..., \mathtt{x}_\mathtt{n} : \mathtt{f}_\mathtt{m}] \equiv \\ \equiv [\mathtt{f}_1, \mathtt{f}_2, ..., \mathtt{f}_\mathtt{n}]\hat{}[\mathtt{x}_1, \mathtt{x}_2, ..., \mathtt{x}_\mathtt{n}].\end{gathered}$$

Правила эквивалентных преобразований и интерпретации списков тесно взаимосвязаны с набором предопределенных функций. Их комбинации определяют широкие возможности по формированию структуры функциональной параллельной программы.

3. Язык программирования

Язык программирования “Пифагор” реализует управление вычислениями, определяемое моделью. Полное описание его текущей версии приведено в [13]. Ниже рассматриваются некоторые особенности программирования.

3.1. Использование параллельных списков

Отсутствие операторов цикла в функциональных языках ведет к использованию рекурсии. В ряде случаев ее можно избежать, если алгоритм задачи предусматривает выполнение одной функции или списка функций над списком независимых аргументов. Тогда можно воспользоваться механизмом параллельных списков.

3.1.1. Параллельный список данных

Рассмотрим пример умножения элементов числового вектора на скаляр. В качестве аргумента предполагается передавать двухэлементный список, первый элемент которого является вектором $\mathbf{X} = (\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n)$, а второй — скаляром $\mathtt{y}$:

$$((\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n), \mathtt{y}).$$

Функциональная программа организована следующим образом:

```
VecScalMult << funcdef Param \{         // 1
     X << Param:1;   y << Param:2;      // 2
    Len << X:|;                         // 3
    V << (y,Len):dup;                   // 4
    ((X,V):\#:[]:*) >>return \}         // 5
}                                       // 6
```

Определение функции задано в первой строке. Во второй строке из двухэлементного списка `Param` выделяется первый элемент, являющийся вектором. Он обозначается именем **X**. Скаляр (второй элемент) обозначается через `y`. В третьей строке предопределенная функция “`|`” используется для вычисления длины вектора, обозначенной через `Len`. В строке 4 создается вектор **V** длиной `Len`, состоящий из одинаковых величин `y`, путем их дублирования функцией `dup`. В пятой строке векторы **X** и **V** объединяются в список, образуя матрицу размерностью `[2, Len]`. Операция “`#`” осуществляет ее транспонирование, преобразуя в матрицу `[Len, 2]`. Следующая затем операция “`[]`” создает параллельный список двухэлементных векторов, над которыми одновременно выполняется операция умножения “`*`”. Результат оказывается размещенным внутри круглых скобок, определяющих сформированный вектор. Его обозначение ключевым словом `return` ассоциируется с выдачей из функции полученного результата.

Пример выполнения:

$$((3, 5.02, -2, 0, 1.5), 10) : \mathtt{VecScalMult} \Rightarrow (30, 5.020000\mathsf{e}+001, -20, 0, 1.500000\mathsf{e}+001)$$

Функциональный стиль, поддерживаемый языком, позволяет во многих случаях сводить исходный текст функции к одному оператору, определяющему все вычисления. Подобная версия программы для рассматриваемого примера будем выглядеть следующим образом:

```
VecScalMultBrief << funcdef Param {
  ((Param:1,(Param:2,Param:1:|):dup):\#:[]:*) >>return
}
```

Следует отметить, что не всегда целесообразно придерживаться такого стиля, поскольку затрудняется восприятие программы и происходит дублирование одних и тех же вычислений.

3.1.2. Параллельный список функций

Функции тоже могут задаваться параллельным списком, определяя тем самым множество потоков функций над одним потоком данных. Следующий пример иллюстрирует параллельное нахождение суммы, разности, произведения и частного двух чисел:

```
ParAddSumMultDiv << funcdef Param {          // 1
  // Формат аргумента: (число, число)       // 2
  (Param:[+,-,*,/]) >>return                // 3
}                                            // 4
```

Результатом вычислений является четырехэлементный вектор, полученный применением разных операций к одному и тому же двухэлементному числовому списку. Пример выполнения:

$$(3,5) : \texttt{ParAddSumMultDiv} \Rightarrow (8,-2,15,6.000000\texttt{e}-001)$$

3.1.3. Использование задержанных списков

Для программирования вычислительных алгоритмов, предусматривающих ветвление, применяются задержанные списки с последующим выбором и раскрытием элемента, соответствующего дальнейшим вычислениям. Рассмотрим пример функции, находящей абсолютное значение скалярного аргумента:

```
Abs << funcdef Param { // Аргумент является числом      // 1
  ({Param:-}, Param):                                  // 2
  [(Param,0):(<,=>):?]:. >>return                      // 3
};                                                     // 4
```

Во второй строке задаются два альтернативных аргумента. Первый из них оформлен в виде задержанного списка, так как вычисление унарного минуса зависит от результата проверки. Второй элемент списка является уже сформированным числом. Поэтому задержка в его вычислении не нужна. В третьей строке осуществляется одновременное сравнение аргумента с нулем на “меньше” и “больше или равно”. Размещение этих операций внутри круглых скобок позволяет получить список данных, состоящий из двух булевых величин. Операция “?” обеспечивает формирование параллельного списка, состоящего из порядковых номеров элементов, имеющих истинное значение. В данной ситуации возможен только один вариант. Полученное значение используется для выбора одного из элементов списка, указанного в строке 1. Функция “.” (точка) используется в качестве пустой операции. Она “раскрывает” задержанные списки и пропускает без изменения прочие аргументы.

Рассмотрим выполнение программы при обработке отрицательного аргумента, равного -5:

$$(\{-5:-\},-5):[(-5,0)(<,>=):?]:.\Rightarrow$$
$$\Rightarrow(\{-5:-\},-5):[(\texttt{true},\texttt{false}):?]:.\Rightarrow$$
$$\Rightarrow(\{-5:-\},-5):[1]:.\Rightarrow\{-5:-\}:.\Rightarrow-5:-\Rightarrow 5$$

При положительном аргументе, например, равном 3, результат будет получаться иначе:

$$(\{3:-\},3):[(3,0)(<,>=):?]:.\Rightarrow(\{3:-\},3):[(\texttt{false},\texttt{true}):?]:.\Rightarrow$$
$$(\{3:-\},3):[2]:.\Rightarrow 3:.\Rightarrow 3$$

3.1.4. Использование параллельной рекурсии

Если вычислительный алгоритм предусматривает древовидное или рекуррентное использование функции для множества однотипных аргументов, количество которых может быть произвольным (например, функция суммирования всех элементов вектора), то применяется параллельная рекурсивная декомпозиция списка аргументов, на самом нижнем уровне которой выполняется операция над одноэлементными или двухэлементными списками, полученными в результате разложения. После этого следует обратный ход со сверткой отдельных результатов. Рассмотрим вычисление суммы элементов числового вектора произвольной длины:

```
VecSum << funcdef Param {                                      //  1
  // Формат аргумента: (x1,...,xn), где x1,...,xn - числа      //  2
  Len<<Param:|;                                                //  3
  return<< .^[((Len,2):[<,=,>]):?]^                            //  4
  (                                                            //  5
    {Param:[]},                                                //  6
    {Param:+},                                                 //  7
    {                                                          //  8
      block {                                                  //  9
        OddVec <<  Param:[(1,Len,2):..];                       // 10
        EvenVec << Param:[(2,Len,2):..];                       // 11
        ([OddVec,EvenVec]: VecSum):+                           // 12
      >>break                                                  // 13
      } // конец блока                                         // 14
    } // конец задержанного списка                             // 15
  )                                                            // 16
}                                                              // 17
```

В строке 3 определяется длина числового вектора, что позволяет в дальнейшем выбрать один из трех вариантов вычислений. При одноэлементном векторе возвращается значение атома, размещенного в списке (строка 6). Для двух элементов в списке (строка 7) осуществляется их суммирование. При большей длине (строки 9–12) происходит разбиение вектора на два (состоящих из четных и нечетных элементов), для каждого из которых одновременно осуществляется рекурсивный вызов функции `VecSum`, и суммирование возвращаемых результатов (строка 12). Строка 4, записанная в префиксной форме, обеспечивает проверку длины, селекцию одного из вариантов, раскрытие задержанного списка и возврат результата.

Пример выполнения:

$$(-3, 6, 10, 25, 0) : \texttt{VecSum} \Rightarrow 38$$

3.1.5. Использование функций в качестве параметров

Если в предыдущем примере нам понадобится находить не сумму, а произведение элементов списка произвольной длины, то необходимо переписывать всю программу. Однако ФЯПП позволяет написать общую функцию декомпозиции, в качестве первого аргумента которой будет выступать обрабатываемый вектор, а вторым аргументом может быть любая бинарная функция, которую предполагается выполнять при свертке дерева.

```
BinTreeReduction << funcdef Param {
 // Формат аргумента: ((x1, x2, ... , xn),f)
 Len << Param:1:|; // длина списка-аргумента
 Func << Param:2;  // Переданная функция
 return<< .^[((Len,2):[<,=,>]):?]^
 (
   {Param:1:[]},      // Первый элемент при длине меньшей двух
   {Param:1:Func}, // Свертка с использованием функции
   { // Блок, определяющий рекурсивные вычисления
   block {
      OddVec  << Param:1:[(1,Len,2):..]; // нечетные элементы
      EvenVec << Param:1:[(2,Len,2):..]; // четные элементы
      // Рекурсивная параллельная декомпозиция со сверткой
     ([(OddVec,Func),(EvenVec,Func)]: BinTreeReduction):Func
    >>break} // конец блока
    } // конец третьего задержанного аргумента
  ) // конец всех альтернатив
}
```

Различные варианты использования функции в качестве параметра приведены в следующем тестовом примере:

```
BinTreeReductionTest << funcdef Param {
  // Формат аргумента: (x1, x2, ... , xn)
  (
    (Param,+):BinTreeReduction,
    (Param,*):BinTreeReduction,
    (Param,Min):BinTreeReduction,
    (Param,AbsAdd):BinTreeReduction
  ) >>return
}
```

В представленном тесте функции “+” и “*” являются предопределенными операциями. Функции `Min` (находит минимум для двух элементов списка) и `AbsAdd` (суммирует абсолютные величины двух чисел) реализованы программно следующим образом:

```
Min << funcdef Param { // Аргумент: (число1, число2)
 Param:[Param:(<=,>):?] >>return
}

AbsAdd << funcdef Param { // Аргумент: (число1, число2)
 (Param:[]:Abs):+ >>return
}
```

Пример использования:

$$(-3, 6, -1, 2, -5) : \texttt{BinTreeReductionTest} \Rightarrow (-1, -180, -5, 17)$$

4. Выполнение функциональных параллельных программ

Для проведения экспериментов были разработаны инструментальная среда, обеспечивающая последовательное выполнение функциональных программ, и параллельный интерпретатор.

4.1. Инструментальная среда

Разработанная инструментальная система предназначена для отладки и демонстрационного выполнения функциональных программ. Она [13] включает:

— встроенный транслятор с функционального языка параллельного программирования “Пифагор” в промежуточное представление;

— интерпретатор оттранслированных программ с возможностью пошагового выполнения, выводом промежуточной информации и результатов вычислений. Данные функциональные модули были объединены интегрированной средой, обеспечивающей непосредственную подготовку программы, ее выполнение и отладку и состоящей из:

— многооконного текстового редактора;

— панели инструментов, с помощью которой осуществляется управление трансляцией, выполнением и пошаговой отладкой разрабатываемых программ непосредственно под управлением данной среды;

— окна для отображения результатов трансляции функциональной программы;

— окна для ввода значения аргумента выполняемой или отлаживаемой функции;

— окна информации пошагового отладчика;

— окна, используемого для отображения результатов выполнения программы.

Использование последовательного интерпретатора позволило начать работы по написанию и отладке функциональных программ. Именно в рамках этой среды первоначально решаются все вопросы, связанные с дальнейшим расширением языка.

4.2. Параллельный интерпретатор

Развитие системы привело к созданию интерпретатора, поддерживающего реальный, а не виртуальный параллелизм [14]. Он функционирует под управлением Mosix, который обеспечивает динамическое распределение выполняемых функций по узлам кластера. Порождение процессов осуществляется стандартными средствами ОС Linux, что позволяет запускать интерпретатор как на кластере, так и на однопроцессорных системах, не использующих данный пакет. Помимо этого Mosix обеспечивает поддержку статического управления, что при необходимости позволяет явно накладывать ограничения на модель вычислений с целью достижения максимальной эффективности конкретной кластерной архитектуры.

Разработка параллельного интерпретатора обеспечила практическую поддержку экспериментов, связанных с исследованием параллельного выполнения функциональных программ, написанных на языке “Пифагор”. Полученные результаты показывают, что даже при простых алгоритмах динамического распределения ресурсов использование кластерных систем позволяет повысить производительность вычислений.

Заключение

Несмотря на ряд недостатков, разработанные на сегодняшний день средства позволяют проводить практические эксперименты по созданию библиотеки параллельных функций и их выполнению на кластерных системах. И хотя еще рано говорить о достижении высокой эффективности разработанной системы, следует отметить возможность ее использования для проведения реальных экспериментов с языковыми конструкциями и кластерными архитектурами.

Продолжаются работы над совершенствованием языка. В частности, реализован динамический контроль типов данных, обеспечена перегрузка функций с одинаковой сигнатурой, что обеспечивает эволюционное расширение программы без изменения уже написанного кода. Реализована поддержка типов, динамически порождаемых пользователем [15]. Планируется поддержка модульной структуры и строгой типизации. Ведутся работы по модификации среды разработки и интерпретаторов, взаимодействию с другими программными средствами.

Проводятся исследования, связанные с преобразованием функциональных программ. В частности, исследуется эквивалентность функций, обеспечивающих замену одних фрагментов на другие, что позволяет сужать или расширять параллелизм задачи. Отрабатывается схема сжатия параллелизма функциональной программы путем замены параллельных фрагментов кода на эквивалентные им последовательные участки. Наличие последовательных участков кода позволяет в дальнейшем осуществить перевод функций на языки, используемые в современных кластерных архитектурах.

Список литературы

[1] Алгоритмы, математическое обеспечение и проектирование архитектур, многопроцессорных вычислительных систем / Под ред. А.П. Ершова. М.: Наука, 1982.

[2] Деннис Дж. Б., Фоссин Дж. Б., Линдерман Дж. П. Схемы потока данных // Теория программирования. Ч. 2 / ВЦ СО АН СССР. Новосибирск, 1972. С. 7–43.

[3] Хендерсон П. Функциональное программирование. Пер. с англ. М.: Мир, 1983.

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

[5] Thompson S. Haskell: The Craft of Functional Programming. 2nd edition. Addison-Wesley, 1999.

[6] Казаков Ф.А., Легалов А.И. Параллельное программирование в языках Haskell и Пифагор // Проблемы информатизации региона. ПИР-2001: Сб. науч. тр. / ИПЦ КГТУ. Красноярск, 2002. С. 48–55.

[7] Касьянов В.Н., Бирюкова Ю.В., Евстигнеев В.А. Функциональный язык SISAL 3.0 // Поддержка супервычислений и Интернет-ориентированные технологии. Новосибирск, 2001. С. 54–67.

[8] Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002.

[9] Головков С.Л., Ефимкин К.Н. Реализация языка программирования для модели вычислений, основанной на принципе потока данных // Методы и средства обработки информации. Тр. первой всероссийской научной конференции. М.: Издат. отдел ф-та вычисл. мат. и кибернетики МГУ им. М.В. Ломоносова, 2003. С. 354–360.

[10] Норма. Описание языка. Рабочий стандарт / Андрианов А.Н., Бугеря А.Б., Ефимкин К.Н., Задыхайло И.Б. М., 1995. (Препр. РАН. ИПМ им. М.В. Келдыша. № 120).

[11] Легалов А.И., Казаков Ф.А., Кузьмин Д.А., Водяхо А.И. Модель параллельных вычислений функционального языка // Изв. ГЭТУ. 1996. Вып. 500. С. 56–63.

[12] Kuzmin D.A., Kazakov F.A., Legalov A.I. Description of parallel-functional programming language // Advances in Modeling & Analysis. A. AMSE Press. 1995. Vol. 28, N 3. P. 1–17.

[13] Описание текущей версии функционального языка параллельного программирования “Пифагор”, функциональная модель параллельных вычислений, примеры программ, транслятор, интерпретаторы. http://www.softcraft.ru/parallel.shtml.

[14] Кузьмин Д.А., Рыженко И.Н., Легалов А.И. Интерпретация функциональных программ на кластере под управлением MOSIX // Вест. Красноярского государственного технического ун-та. 2003. Вып. 33. С. 196–205.

[15] Легалов А.И., Привалихин Д.В. Особенности функционального языка параллельного программирования “Пифагор” // Высокопроизводительные вычисления на кластерных системах: Матер. Четвертого международного научно-практического семинара и Всероссийской молодежной школы / Под ред. В.А. Сойфера. Самара, 2004. С. 173–179.

Поступила в редакцию 10 июля 2003 г.,
в переработанном виде — 2 апреля 2004 г.