Исследование возможностей параллельной трансляции логических программ

Дацун Н.Н., Шуликов А.В.
Донецкий национальный технический университет
datsun@pmi.dgtu.edu.ua
shulikov@gmail.com

Abstract

Datsun N., Shulikov A. "Research of possibilities of parallel translation of logic programs". *The article examines methods of paralleling of Prolog program translation process. The organisation of internal representation of the program for performance on parallel architecture is offered.*

Введение

Для решения сложных прикладных задач требуется большой объем вычислений и значительное количество вычислительных ресурсов. Это порождает необходимость увеличения быстродействия компьютерных систем. Существует два способа повышения производительности вычислительных систем - экстенсивный и интенсивный. Экстенсивный способ предполагает улучшение архитектуры, использование процессоров с большей тактовой частотой, интенсивный - оптимизацию, разгрузку кода, распараллеливание.

Наиболее доступным решением является построение кластера из нескольких недорогих компьютеров. Распараллеливание в процессе трансляции даёт преимущество в скорости работы программы. Трансляторы языков программирования для многопроцессорных вычислительных систем должны проектироваться в виде системы параллельных и взаимодействующих между собой процессов, позволяющих выполнять участки кода на различных процессорах над независимыми наборами данных.

Выполнение сложных запросов на логическом языке Пролог в больших базах знаний, затрагивающих множество различных правил, на однопроцессорных системах может затянуться на длительное время. Решение данной проблемы заключается в исследовании способов распараллеливания в процессе трансляции программ на языке Пролог.

Объектом исследования являются способы и методы распараллеливания программы на языке Пролог в процессе трансляции, включая внутренне представление программы на этапе трансляции.

Предметом исследования выступает оценка эффективности распараллеливания программ на языке Пролог с использованием стандарта MPI для организации взаимодействия процессов, задействованных в решении задачи.

Целью работы является исследование возможностей распараллеливания логических программ на фазах лексического и синтаксического анализа с последующим построением транслятора с полной прозрачностью параллелизма типа «ИЛИ».

1 Анализ трансляторов логического языка, реализующих параллелизм вычислений

В трансляторах Пролога возможно выделить следующие виды параллелизма:

- «ИЛИ»-параллелизм имеет место, когда одна подцель может быть унифицирована с заголовками нескольких утверждений, и тела этих клауз вычисляются различными вычислительными модулями;
- «И»-параллелизм - имеет место, когда подцели, принадлежащие данному запросу выполняются параллельно различными вычислительными модулями;
- мультитрединг - получение и обработка одновременно «многих» потоков команд и данных: при этом устройство управления распределяет потоки для параллельного выполнения на внутренних вычислительных устройствах процессора.

«ИЛИ»-параллелизм заключается в формировании множества правил исходной программы, сопоставимых с предикатом запроса, и разделении этого множества между различными вычислительными модулями. Это связано с тем, что различные правила в программе, совпадающие с предикатом запроса, связаны между собой отношением «ИЛИ» [1].

Правило Пролога имеет вид:

$$P_m : - P1, P2, \ldots, P_n .$$

Подцели в правой части правила связаны между собой отношением «И». Если

подцели не имеют общих переменных среди фактических аргументов, можно выполнять P1, P2, …, P*n* параллельно, а не последовательно. В случае наличия общих переменных в аргументах, подцели правила возможно выполнить только последовательно. Данный вид параллелизма называется «И»-параллелизмом. В трансляторе может быть реализован как один, так и два вида параллелизма одновременно.

Исходя из структуры программы на языке Пролог, вытекает проблема интеграции различных парадигм программирования: логической парадигмы с одной стороны, и, например, функциональной - с другой. Решение задачи такой интеграции (а значит, и сам процесс распараллеливания Пролог-программы) может быть основано на одной из трех степеней прозрачности [2] этого процесса по отношению к тексту программы:

- полная прозрачность - процесс распараллеливания происходит внутри транслятора за счет особой организации внутреннего представления данных и процесса получения результата (без участия программиста);
- промежуточная прозрачность - программист должен каким-то образом декларировать предикаты распараллеливания, после чего можно работать с ними, как и с любыми другими предикатами. В этом случае остается скрытым факт, что во время сопоставления с предикатом используется механизм распараллеливания;
- отсутствие прозрачности - в программе на логическом языке должны быть явно декларированы операторы языка параллельных вычислений.

Реализация любого вида параллелизма и использование любого из принципов прозрачности требуют модификации всех блоков транслятора логического языка.

В случае полной прозрачности необходима модификация внутренних структур данных и алгоритмов трансляции. Промежуточная прозрачность требует расширения диалекта языка и введения дополнительных предикатов, управляющих параллельными вычислениями. При отсутствии прозрачности требуется доработка интерфейса для интеграции транслятора с модулями на других языках программирования, которые будут явно реализовать параллелизм [3].

Среди реализаций языка Пролог существуют как последовательные, так и параллельные варианты. Основное отличие заключается в способах выбора редуцируемой цели. Последовательный Пролог и его расширения основаны на последовательном выполнении по принципу чистого Пролога. Параллельные реализации, такие, как PARLOG [4], Параллельный Пролог [5] и GHC [6], основаны на параллельном выполнении.

Информация о параллельных реализациях языка Пролог приведена в табл. 1.

Таблица 1 - Реализации трансляторов языка Пролог

Название	Параллелизм	Прозрачность	Разработчик, источник
Parlog	И-параллелизм	промежуточная	Parallel Logic Programming Ltd., www.parlog.com
QuProlog	мультитрединг	промежуточная	University of Queensland, www.itee.uq.edu.au/~pjr/HomePages/ QuPrologHome.html
Акторный Пролог	ИЛИ-параллелизм	промежуточная	Морозов А. А., www.cplire.ru/Lab144/1251/09010000.html
SWI Prolog	мультитрединг	промежуточная	The SWI-Prolog Foundation, www.swi-prolog.org
YAP Prolog	ИЛИ-параллелизм	полная	LIACC/Universidade do Porto, www.ncc.up.pt/~vsc/Yap
Arity Prolog	мультитрединг		Arity, www.arity.com/www.pl/products/ap.htm
BinProlog	мультитрединг		Binnet Corp., www.binnetcorp.com/BinProlog
Brain Aid Prolog	параллельные процессы на последовательной архитектуре		M. Ostermann, www.comnets.rwth-aachen.de/~ost/private.html
BePOP	последовательное программирование, «И»-параллелизм		http://www.faqs.org/faqs/prolog/resource-guide/part2/section-4.html

Parlog реализует параллелизм типа «И» с защитой и недетерминированным совершением выбора. Реализован только обычный перебор с возвратом [7]. Qu-Prolog — это расширение Пролога, разработанное в первую очередь как язык прототипов и язык для интерактивного доказательства теорем. Он предоставляет поддержку для символьных вычислений на нотациях, которые возникают в математике и в спецификации языков, таких как Z. Включает в себя базовую поддержку произвольных кванторов с параллельным связыванием и опциональным вводом информации. Поддерживаются обозначения более высокого порядка, такие как функциональные выражения. Алгоритм унификации построен с поддержкой данных возможностей. Qu-Prolog поддерживает мультитрединг и предоставляет высокоуровневую коммуникацию между нитями, процессами, компьютерами. В комплексе эти возможности предоставляют простой и в то же самое время более мощный механизм для агентного программирования [8].

Акторный Пролог - объектно-ориентированный логический язык, предназначенный для программирования информационных систем, функционирующих в динамическом внешнем окружении (интеллектуальных агентов Интернет, систем интерактивного проектирования и др.). Акторный Пролог воплощает новый подход к объединению логического и объектно-ориентированного программирования. Процессами называются экземпляры классов объектно-ориентированного логического языка Акторный Пролог, предложения которых исполняются параллельно по отношению к предложениям других экземпляров классов [9]. Таким образом, Акторный Пролог реализует параллелизм типа «ИЛИ». Но чтобы воспользоваться данным интерпретатором, необходимо существенно модифицировать программы, написанные на классическом языке Пролог. Это усложняет или делает невозможным использование данной реализации для уже решенных задач.

SWI Prolog предлагает комплексное свободное программное обеспечение среды разработки на языке Пролог. Широко используется в научных исследованиях и образовании, а также для коммерческих приложений. Среди достоинств очень маленький размер, быстрая компиляция, масштабируемость для очень крупных приложений. Поддерживает параллелизм на основе мультитредингa, заключающийся в запуске нескольких процессов Пролога на одной и той же базе данных [10].

Реализация YAP Prolog базируется на WAM (Warren Abstract Machine) с некоторыми оптимизациями для улучшения скорости работы. Реализация поддержки «ИЛИ»-параллелизма в данном трансляторе, получила название YAPOr. В этой системе параллелизм реализуется неявно в виде запуска нескольких альтернатив в параллели. YAPOr является экспериментальным расширением YAP Prolog и обладает целым рядом недостатков: не поддерживается параллельное обновление базы данных, не поддерживается открытие и закрытие потоков во время параллельного выполнения, не работает сборщик мусора, не поддерживается приостановка работы [11].

Brain Aid Prolog реализует совокупность параллельных процессов, работающих в последовательном режиме, базируется на стандартном Прологе и предназначен для транспьютерных сетей. Среди специальных возможностей можно выделить автоматическую адаптацию к топологии и отладку уровня параллелизма [12].

BePOP комбинирует последовательное и параллельное логическое программирование, объектно-ориентированное программирование. Компоненты логического программирования предоставляют обе особенности – недетерминированный выбор цели в резольвенте, а также «И»-параллелизм. Объектно-ориентированные возможности включают в себя идентификаторы объектов, инкапсуляцию, передачу сообщений, обновление состояния и изменение поведения объекта.

Большинство существующих трансляторов реализуют принцип промежуточной прозрачности, требующий для параллельного выполнения модификации кода программы, включающего указание точек распараллеливания тем или иным образом. Принцип полной прозрачности характерен для YAP Prolog. В данном трансляторе без модификации исходной программы на языке Пролог происходит распараллеливание вычисления альтернативных правил. Однако данная функция носит экспериментальный характер и не включена в основные сборки транслятора, что требует пересборки транслятора из исходного кода с включением опции распараллеливания.

В данной работе рассматривается транслятор логического языка с полной прозрачностью параллелизма, ориентированный на теоретические исследования. Задачами этих исследований являются:

- анализ логических программ на фазе трансляции;
- определение вариантов повышения их производительности.

2 Постановка задачи

Анализ характеристик существующих трансляторов языка Пролог на параллельных архитектурах позволил выделить требования к разрабатываемому транслятору:

- полная прозрачность ИЛИ-параллелизма;
- наличие инструментов анализа программ для определения целесообразности распараллеливания на фазе трансляции.

Параллелизм логических программ может быть использован как SIMD-архитектурах, так и на MIMD- архитектурах. Для SIMD-архитектур более естественным является ИЛИ-параллелизм.

Полная прозрачность параллелизма требуется для обеспечения автоматического рефакторинга уже существующих последовательных логических программ, чтобы исключить модификацию программистом текстов программ при их переносе на параллельные архитектуры.

Для проведения тестирования и упрощения построения тестового транслятора на реализуемый диалект языка Пролог наложены определенные ограничения:

- исключена поддержка арифметических операций, предикатов отсечения и возврата;
- не используется анонимная переменная;
- в аргументах допускается использование только переменных и констант;
- запрещена перегрузка предикатов.

3 Методы и алгоритмы на этапе трансляции Пролог-программы

Для языков логического типа чаще всего трансляторы реализуют в виде интерпретаторов [13].

Для транслятора языка Пролог характерна такая последовательность фаз работы:

- фаза лексического анализа;
- фаза синтаксического анализа;
- фаза унификации и доказательства целей.

3.1 Фаза лексического анализа

Лексический анализатор (сканер) выделен в отдельный блок, реализуя концепцию двухпроходного транслятора. Для определения грамматики лексического анализатора выбрана грамматика типа 3 по Хомскому. Использованы таблицы разделителей, переменных, констант, предикатов, также замена лексем на целочисленные коды, определяющие их положение в соответствующей таблице. Одной из важных задач лексического анализатора является построение таблиц имен, хранящих соответствие кодов и имен объектов в исходной программе. Эти данные используются на фазе унификации.

Рассмотрим вопрос возможности и целесообразности распараллеливания процесса лексического анализа с целью принятия решения о необходимости данного шага на пути к получению лучших результатов по скорости работы программы.

Введем следующие обозначения:

- n_1 – количество лексем;
- n_2 – количество утверждений;
- n_3 – количество символов;
- n_4 – скорость каналов связи между узлами;
- n_5 – размер таблицы имен (постоянная величина);
- n_6 – количество узлов вычислений.

В случае последовательного варианта лексического анализатора общее время работы этой фазы транслятора линейно зависит от n_1, n_2, n_3.

При реализации лексического анализатора на параллельной архитектуре общее время работы будет состоять из:

- $t_{разб}$ - времени: разбиения программы на части;
- $t_{сер\_об}$ – времени сериализации полученных объектов для передачи по сети;
- $t_{пер1}$ – времени пересылки данных по сети;
- $t_{десер1}$ - времени сборки данных на узлах вычислений;
- $t_{обр}$ – времени собственно обработки на каждом узле;
- $t_{сер\_т}$ – времени сериализации сформированных таблиц имен;
- $t_{пер2}$ - времени пересылки их в центральный узел;
- $t_{десер2}$ – времени сборки;
- $t_{скл}$ - времени склейки разрозненных таблиц имен в окончательную таблицу

$$T_{общ} = t_{разб} + t_{сер\_об} + t_{пер1} + t_{десер1} + t_{обр} + t_{сер\_т} + t_{пер2} + t_{десер2} + t_{скл}$$

Наиболее затратной по времени будет операция склейки полученных таблиц имен в единую таблицу. Временная сложность данной операции зависит от n1 и $n5^{n6}$.

В связи с этим целесообразно реализовать последовательный вариант лексического анализатора. Это позволит значительно сократить время разработки

транслятора без существенного ухудшения его временных характеристик.

3.2 Фаза синтаксического анализа

Синтаксический анализатор (парсер) представляет собой достаточно сложный блок транслятора. В нем можно выделить следующие составляющие:

- распознаватель;
- блок семантического анализа;
- формирование внутреннего представления программы.

Благодаря упрощению моделируемого диалекта языка Пролог целесообразно совместить функционал синтаксического и семантического анализатора в едином блоке. Это позволило более эффективно формировать внутреннее представление программы, минуя формирование промежуточных структур данных и их передачу между блоками транслятора.

Внутреннее представление данных, формируемое синтаксическим анализатором, во многом определяет эффективность работы транслятора на этапе унификации. Для реализации распараллеливания необходимо выбрать внутреннее представление таким образом, чтобы упростить как сам процесс унификации, так и процесс обмена данными между процессами, реализующими унификацию.

При проектировании данного транслятора на этапе работы синтаксического анализатора было предложено формировать базу данных внутреннего представления логической программы в виде набора следующих основных таблиц: фактов, правил, арностей, индексации, зависимостей.

В ходе последовательного разбора входной цепочки синтаксический анализатор формирует временную лексему, признаком окончания которой служит точка. Когда временная лексема сформирована, происходит определение ее типа, проверка семантической корректности (отсутствие изменения типа текущего предиката, отсутствие изменения арности); в случае правил фиксируется зависимость предикатов в таблице зависимостей, заполняется таблица индексации.

Таблица фактов содержит атрибуты: код факта и код предиката. Код факта позволяет однозначно идентифицировать данный факт и является ключевым атрибутом. Для описания аргументов используется связанная таблица аргументов фактов. Атрибутами этой таблицы являются код факта, номер аргумента, код аргумента.

Таблица правил содержит атрибуты: идентификатор, код правила, номер предиката, код предиката. Для описания аргументов предикатов, встречающихся в правиле, используется связанная таблица аргументов правил, содержащая атрибуты: идентификатор, номер аргумента, код аргумента.

Таблица арностей предикатов содержит атрибуты: код предиката и арность предиката. Данная таблица фиксирует информацию о количестве аргументов, содержащихся в предикатах каждого вида. Она используется как на этапе проверки семантической корректности при синтаксическом анализе, так и на фазе унификации.

Таблица индексации содержит атрибуты: код предиката, тип предиката, количество объектов в программе с данным предикатом. С ней связана таблица индексов, определяющих все вхождения данного предиката внутри таблицы фактов или правил. Таблица индексации обеспечивает быстрый доступ к конкретным фактам и правилам, сокращая время поиска в таблицах.

Важной структурой данных является перекрестная матрица зависимостей. Заголовками строк и столбцов в данной матрице являются коды предикатов. Значения ячеек матрицы определяют зависимость данного предиката от другого. Это имеет существенное значение для правил. В случае реализации рассылки базы данных внутреннего представления это позволит не рассылать множество лишних данных узлам, а передавать лишь информацию, которая необходима на этапе унификации конкретной цели.

Пусть множество Rules объединяет все правила, входящие в программу в виде линейных целочисленных массивов. Нулевой элемент представляет собой код предиката заголовка. Все последующие элемент – это коды предикатов тела правила. Матрицу зависимостей обозначим MD. Процедура Length возвращает количество элементов в переданном объекте. Алгоритм формирования матрицы зависимостей приведен на рис. 1:

```
1. Rules_count = Length(Rules)
2. Для i от 1 до Rules_count
     2.1. Для j от 1 до Rules_count
       2.1.1. MD[i,j] = 0
3. Для i от 1 до Rules_count
     3.1. PR = Rules[i]
     3.2. N = Length(PR)
     3.3. HP = PR[0]
     3.4. Для j от 1 до N-1
       3.4.1. MD[HP,j] = 1
4. Выход
```

Рисунок 1 – Алгоритм формирования матрицы зависимостей

Так как Пролог чувствителен к порядку следования предикатов в программе, на фазе трансляции также выполняется анализ структуры логической программы. Последовательность состояний позволяет проверить правильность расположения объектов программы в порядке следования. При выделении синтаксическим анализатором очередного выражения, его проверкой занимается семантический анализатор. Производится проверка на первичное вхождение данного предиката. В случае, если вхождение первое, то инициализируются для предиката с данным именем такие характеристики как арность и тип предиката. Если вхождение повторное, то производится проверка этих характеристик на соответствие первому вхождению.

Распараллеливание процесса унификации при условии необходимости рассылки данных между узлами (запросов, результатов, базы данных) подразумевает использование при пересылке линейных структур данных. Выбор структур данных для внутреннего представления логической программы при реализации на параллельных архитектурах должен учитывать указанное ограничение.

Существует два варианта разрешения этого ограничения. Вариант 1: рассылаемые данные необходимо заведомо представлять в виде линейной структуры данных (например, множеством однотипных объектов). Вариант2: включать в работу транслятора алгоритмы сериализации составных типов данных.

Упрощение внутренних алгоритмов синтаксического анализатора происходит за счет передачи на его вход не лексем в текстовом представлении, а лишь кодов, получаемых на выходе лексического анализатора. Коды однозначно определяют лексемы по отношению к одному из следующих типов: предикат, константа, переменная, разделитель. Синтаксическому анализатору остается только проверка синтаксической правильности и семантической корректности.

В описанных выше фазах лексического и синтаксического анализа практически все объекты логической программы представлены своими целочисленными кодами. Характеристики объектов, как правило, также кодируются целочисленными кодами. Значит, внутри программы фигурирует множество целочисленных данных, которые легко выкладываются в однородную линейную структуру, пригодную для передачи узлам. Поэтому выбор такой организации структур данных для внутреннего представления Пролог-программы при полной прозрачности параллелизма исключает накладные расходы на сериализацию объектов программы.

4 Реализация транслятора

4.1 Выбор программных средств

Список современных средств и инструментов для разработки программного обеспечения достаточно обширный. Среди них имеются как платные интегрированные решения, такие как Microsoft Visual Studio или Borland C++ Builder, а также бесплатные компоненты, такие как коллекция компиляторов GCC [14], среда разработки Eclipse [15], средство построения компиляторов Flex [16]. В виду проблемы лицензионных притязаний при использовании проприетарного программного обеспечения было принято решение использовать в разработке программные продукты, распространяемые бесплатно со своим открытым исходным кодом. Данные продукты не уступают в необходимой функциональности платным аналогам, а значит, подходят для разработки транслятора языка логического программирования.

В качестве языка программирования был выбран язык C++ [17]. Одним из существенных плюсов является наличие в современных реализациях C++ привязок к библиотекам промежуточного программного обеспечения (middle-ware) для облегчения реализации параллельного выполнения, а также гибкое управление памятью.

Для построения лексического анализатора выбран открытый инструмент GNU Flex [16]. При помощи описания грамматики в виде регулярных выражений и состояний он позволяет легко реализовать сканер разрабатываемого транслятора. Регулярные выражения представляют собой систему разбора текстовых фрагментов по шаблону. В инструменте Flex применяются Perl-совместимые регулярные выражения. Пример скрипта для анализатора, разбирающего файл с фактами Пролога приведен на рис. 2: входная Пролог-программа разбирается на лексемы, из нее удаляются разделители, в стандартный выходной поток выводятся эти лексемы.

4.2 Лексический анализатор

При помощи утилиты flex++, входящей в состав Flex генерируется код анализатора на языке C++. Интерфейсы сгенерированных классов находятся в заголовочном файле FlexLexer.h, который подключается к проекту.

Для вызова анализатора необходим объект класса yyFlexLexer. Конструктор данного класса принимает в качестве параметров источник данных, в данном случае объект класса файловый входной поток (рис. 3):

```
%{
#include <stdio.h>
%}
%option noyywrap

IntConst [1-9][0-9]*
FloatConst [1-9][0-9]*\.[0-9]*
RuleName [a-z][a-zA-Z0-9_]*
Variable [A-Z_][a-zA-Z0-9_]*
StrConst [a-z][a-zA-Z0-9_]*
StrConstInQuotas \"[a-zA-Z0-9_ ]+\"

%s RArgs
%s RSkipSpace
%s RAfterArgs
%s REndHeader
%%

<INITIAL>{RuleName} { printf("%s\n", yytext); BEGIN(RSkipSpace); }
<RSkipSpace,RArgs,RAfterArgs,REndHeader,INITIAL>[ \t\n]+ {  }

<RSkipSpace>\( { BEGIN(RArgs); }
<RArgs>{StrConstInQuotas} { printf("%s\n", yytext); BEGIN(RAfterArgs); }
<RArgs>{Variable} { printf("%s\n", yytext); BEGIN(RAfterArgs); }
<RArgs>{FloatConst} { printf("%s\n", yytext); BEGIN(RAfterArgs); }
<RArgs>{IntConst} { printf("%s\n", yytext); BEGIN(RAfterArgs); }
<RArgs>{StrConst} { printf("%s\n", yytext); BEGIN(RAfterArgs); }
<RAfterArgs>, { BEGIN(RArgs); }
<RAfterArgs>\) { BEGIN(REndHeader); }
<REndHeader>\. { BEGIN(INITIAL); }
. printf( "Unrecognized character: %s\n", yytext );
```

Рисунок 2 – Скрипт Flex для фрагмента лексического анализатора

```
yyFlexLexer *lexer;
lexer =
      new yyFlexLexer(new std::ifstream(argv[1]));
lexer->yylex();
```

Рисунок. 3 - Вызов лексического анализатора

В коде лексического анализатора используется функция, формирующая ленту кодов. В результате работы в программе присутствует указатель на целочисленный массив, содержащий коды объектов программы.

Таблицы имен реализованы в виде классов. В классе имеется метод получения кода имени, возвращающий код, присвоенный данному объекту. При обнаружении значащей лексемы анализатор обращается к данному методу для получения кода и вызывает процедуру формирования ленты кодов.

4.3 Синтаксический анализатор

В связи с упрощением диалекта языка Пролог, реализуемого транслятором, парсер выполнен без применения генератора синтаксических анализаторов. Он представляет собой функцию, которая выполняет разбор ленты кодов, полученной от лексического анализатора.

Распознаватель принимает на вход цепочку лексем и на ее основе осуществляет разбор в соответствии с используемыми правилами грамматики. Лексемы, при успешном разборе правил, передаются семантическому анализатору, который строит таблицы фактов и правил и фиксирует фрагменты синтаксической структуры. Кроме этого, между фактами и правилами фиксируются дополнительные семантические связи.

Синтаксический анализатор имеет определенный набор состояний. Это позволило реализовать его при помощи переключателя switch-case. Попадание в состояние

завершенного выражения (факт или правило) вызывает выполнение кода, анализирующего семантическую корректность данного выражения, а также фиксирующего данные об этом выражении.

В результате работы парсера формируется внутреннее представление Пролог-программы.

4.4 Внутреннее представление логической программы

Внутреннее представление программы внутри транслятора играет существенную роль в реализации процесса распараллеливания на фазе выполнения. Рассылка данных при помощи MPI использует линейные структуры однотипных данных. В связи с этим факты, правила и другие структуры должны быть представлены в виде целочисленных массивов.

Таблицы фактов и правил представляют собой векторы указателей на целочисленные массивы, содержащие описание конкретных объектов. Описание факта содержит в первом элементе код предиката, в последующих – коды аргументов. Описание правила имеет аналогичную структуру, содержащую последовательность из кодов предикатов и их аргументов.

Таблица зависимостей предикатов реализована при помощи указателя на целочисленную матрицу.

Таблица индексации представлена в виде вектора указателей на целочисленные массивы. Позиция элемента в векторе, описывающего заданный предикат, определяется кодом предиката. Таким образом, для получений информации о предикате достаточно обратиться к элементу вектора по коду предиката.

Важной является структура задач для вычисления фактов и правил.

Задачи для вычисления факта содержит такую информацию:

- task_id – идентификатор задачи;
- query_str – вектор запроса, в виде кода предиката и кодов аргументов;
- vars – текущее значение переменных, присутствующих в задаче;
- vars_count – количество переменных, присутствующих в запросе;
- resolve_answer – логический результат унификации (истина/ложь);
- vars_eq – эквивалентные коды переменных в вызывающем процессе;
- answers – вектор указателей на массивы, содержащие значения переменных, полученные при успешном сопоставлении.

Сформированная задача для унификации факта передается в качестве фактического аргумента функции UniFact.

Задача для унификации правила имеет более сложную структуру. Она содержит поля, отличные от задачи унификации факта:

- query_str_len – длина строки запроса на унификацию;
- vars_count_in_header – количество переменных в заголовке правила.

Задача для унификации правила создается для каждой альтернативы с совпадающим предикатом в заголовке правила. В строке запроса помещается правило из таблицы правил с замененными значениями кодов переменных. Сформированная задача для унификации правила передается фактическим аргументом в функцию UniRule.

Результаты экспериментального исследования разработанного транслятора логического языка с полной прозрачностью ИЛИ-параллелизма приведены авторами в работе [18].

Выводы

Предложены методы трансляции логических программ применительно к транслятору языка логического программирования Пролог с ИЛИ-параллелизмом. Выполнена разработка основных алгоритмов и структур данных транслятора. Описаны принципы предварительной обработки программы, анализа ее структуры для параллельного выполнения, построения внутреннего представления. Обосновано отсутствие параллелизма на фазах лексического и синтаксического анализа. Разработана структура внутреннего представления Пролог-программы для реализации параллелизма на этапе унификации с упрощенным доступом к объектам, необходимым в процессе доказательства целей.

На основании полученных теоретических результатов выполнена реализация транслятора с полной прозрачностью ИЛИ-параллелизма. Проведенные авторами эксперименты показали работоспособность созданного транслятора [18]/

Результаты проектирования транслятора могут быть использованы в качестве дополнительной информации при написании трансляторов логического языка для параллельных архитектур.

Литература

1. Хоггер К. Введение в логическое программирование. – М.: Мир, 1988. – 384 с.
2. Чери С., Готлоб Г., Танка Л., Логическое программирование и базы данных – М.: Мир, 1992.

3. Шуликов А.В., Дацун Н.Н. Увеличение производительности транслятора языка Пролог с использованием механизма параллельных вычислений/ Інформатика та комп'ютерні технології 2007. Матеріали III науково-технічної конференції молодих учених та студентів. - 2007. - С. 516-518.
4. Gregory S. Parallel Logic Programming in PARLOG, The Language and Its Implementation. - Boston: Addison-Wesley Longman Publishing, 1987.
5. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. – М.: Мир, 1990. – 235с.
6. The Glasgow Haskell compiler. - Режим доступа: http://www.haskell.org/ghc.
7. Clark K., Gregory S. Parlog: A Parallel Logic Programming Language. - London: Imperial College, 1983.
8. Qu-Prolog Home Page. - Режим доступа: http://www.itee.uq.edu.au/~pjr/HomePages/QuPrologHome.html
9. Акторный Пролог. Определение языка программирования. - Режим доступа: http://www.cplire.ru/Lab144/koi8/09010000.html
10. SWI Prolog Home Page. - Режим доступа: http://www.swi-prolog.org/
11. Yet Another Prolog. - Режим доступа: http://www.dcc.fc.up.pt/~vsc/Yap/
12. Brain Aid Prolog Home Page. – Режим доступа: http://www.comnets.rwth-aachen.de/~ost/private.html
13. Хантер Р. Проектирование и конструирование компиляторов.- М.: Финансы и статистика, 1984.
14. GNU Compiler Collection. – Режим доступа: http://gcc.gnu.org.
15. Eclipse Platform. – Режим доступа: http://www.eclipse.org.
16. The Fast Lexical Analyzer. – Режим доступа: http://www.gnu.org/software/flex/.
17. Солтер Н., Клеппер С. Язык программирования C++ для профессионалов. Си. – К.: Диалектика, 2006. – 912 с.
18. Дацун Н.Н., Шуликов А.В. Моделирование параллельных вычислений в трансляторе языка логического программирования/ Моделирование и компьютерная графика-09. Труды 3 Межд. научн. конф. – Донецк: ДонНТУ, 2009. – с.260-264.