УДК 681.3.06:800.92+681.3.24

ВОЗМОЖНОСТЬ ИСПОЛЬЗОВАНИЯ ДЕКЛАРАТИВНЫХ ЯЗЫКОВ ДЛЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Вишневский Ю.А., аспирант, Пожуев В.И., д. ф.-м. н., профессор

Запорожская государственная инженерная академия

В статье исследуется возможность применения декларативных языков для организации параллельных вычислений. Показывается возможность реализации параллельной версии языка пролог, приводится его грамматика. В статье также описываются особенности реализации декларативного языка, предназначенного для параллельных распределенных вычислений.

Ключевые слова: параллельные вычисления, декларативные языки, пролог, транслятор, грамматика, распределенные вычисления.

Вишневський Ю.О., Пожуєв В.І. МОЖЛИВІСТЬ ЗАСТОСУВАННЯ ДЕКЛАРАТИВНИХ МОВ ДЛЯ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ / Запорізька державна інженерна академія, Україна

У статті досліджується можливість застосування декларативних мов для організації паралельних обчислень. Показується можливість реалізації паралельної версії мови пролог, наводиться її граматика. У статті також описуються особливості реалізації декларативної мови, призначеної для паралельних розподілених обчислень.

Ключові слова: паралельні обчислення, декларативні мови, пролог, транслятор, граматика, розподілені обчислення.

Vishnevsky Y.A., Pozhuev V.I. THE POSSIBILITY OF USING DECLARATIVE LANGUAGES FOR PARALLEL COMPUTATIONS / Zaporizhzhya State Engineering Academy, Ukraine

The possibility of using declarative languages in parallel computations organizing is discussed in this article. The possibility of creating the parallel version of prolog language is shown, the grammar of such a language is also given. The peculiarities of the declarative language, intended for parallel distributed computations are described too.

Key words: parallel computing, declarative languages, prolog, translator, grammar, distributed computing.

При решении задач параллельного программирования важную роль играет автоматизация процесса создания программ. Особенно это важно для распределенных параллельных программ, ориентированных на использование вычислительных кластеров. Для этих программ характерна сложность написания и отладки в силу того, что используются вычислительные средства с распределенной памятью и скоростью межпроцессорного, а следовательно, и межпроцессового, взаимодействия гораздо ниже, чем в многопроцессорных системах с общей памятью. Таким образом, процесс создания параллельных программ для таких вычислительных кластеров более сложен, поскольку программист должен заботиться об эффективности межпроцессорного взаимодействия, для обеспечения приемлемых результатов.

Применение специализированных средств позволит упростить процесс написания распределенных программ для вычислительных кластеров. Встраивание средств автоматизации непосредственно в язык программирования позволит получать эффективный код, обеспечивая при этом удобную среду для создания программ.

Стратегию автоматического распараллеливания можно представить в виде таких этапов:

1. Разбиение программы на логически независимые последовательности команд, которые могут быть выполнены независимо друг от друга.
2. Выделение блоков данных, над которыми работает каждая из этих ветвей.
3. Распределение данных, над которыми работает программа по вычислительным модулям (ВМ) кластера.
4. Распределение этих команд между ВМ кластера.

Приведенная стратегия характерна для распараллеливания как по командам, так и по данным. Однако иногда встречаются задачи, для которых требуется только один тип распараллеливания – либо по данным, либо по операциям.

Процесс автоматического или полуавтоматического распределения программы в значительной мере зависит от выбранного языка программирования и степени интеграции с ним средств автоматического распараллеливания.

Рассмотрим два типа языков программирования, существующих сегодня: *декларативные* и *процедурные*. Для обоих типов языков характерны свои достоинства и недостатки в плане применимости для параллельного программирования, в качестве языка со встроенными средствами автоматического распараллеливания.

Для процедурных языков сложным является именно распараллеливание по операциям. Поскольку вся последовательность действий в программе на таком языке определяется программистом, то выделение частей программы, указанных в пункте 1 стратегии, является достаточно нетривиальной задачей, не только для программного средства, но и для самого программиста.

Перечисленные выше недостатки процедурных языков для распараллеливания по операциям заставляют обратиться к декларативным языкам (таким, как пролог и лисп). Такие языки являются более выгодными с точки зрения распараллеливания по операциям, так как для них характерны следующие особенности:

- Последовательности команд из пункта 1 уже являются практически выделенными в самой программе (примером таких последовательностей являются правила пролога).
- Процесс выполнения программы управляется транслятором программы и машиной вывода.
- Абстрагирование от конкретной вычислительной среды.
- Развитые средства интерфейса с процедурными языками.

Основная причина, по которой декларативные языки мало применяются для написания распределенных программ - это их ориентированность на задачи искусственного интеллекта, для которых редко возникает необходимость в объемных вычислениях. Мы намерены показать, что применение декларативных языков (возможно, в связке с процедурными языками) для решения вычислительных задач с автоматическим распараллеливанием оправдано, поскольку дает возможность максимально абстрагироваться от реализации параллельных распределенных вычислений. Это, в свою очередь, обеспечит переносимость и масштабируемость программы.

Если декларативный язык кроме средств для параллельного программирования дополнить развитым математическим аппаратом, то это даст возможность решать с их помощью вычислительные задачи, требующие распараллеливания.

СТРАТЕГИИ АВТОМАТИЧЕСКОГО РАСПАРАЛЛЕЛИВАНИЯ ПРОГРАММ НА ПРОЛОГЕ

Рассмотрим возможность автоматического распараллеливания программ на языке пролог. Программу на языке пролог можно представить как рекуррентную последовательность вычисления предикатов. То есть, вычисление целевого предиката влечет за собой вычисление одного или нескольких предикатов, которые, в свою очередь, требуют вычисления других предикатов и т.д. Вычисление носит иерархический рекурсивный характер. А это, в свою очередь, означает, что вычисление любого предиката из правой части правила ничем не отличается от вычисления целевого предиката и представляет собой, по сути новой цели на языке пролог.

С точки зрения параллельных вычислений такое построение программы является очень выгодным, поскольку вычисления для любого предиката могут проводиться независимо, на отдельном процессоре или отдельном ВМ кластера.

Например:

pred1(P1, P2, P3) :- pred1(P1, P2, T1), pred2(P1, P2, T2), P3=T1+T1.

В этом примере возможно параллельное вычисление предикатов `pred1` и `pred2`.

В случае, если для одного из предикатов решение не вычислено, то все правило из примера считается невычисленным. В этом случае следует прервать вычисление и для другого предиката. Параллельные вычисления и в таком случае имеют выигрыш перед последовательными – в случае, если предикат `pred2` не имеет решения, это может быть выяснено ещё до окончания вычислений `pred1`, а в случае последовательно работающей программы вычисления `pred1` были бы доведены до конца, и только после этого стало бы ясно, что все указанное правило не имеет решений.

Очевидно, что распараллеливаемость программ зависит от типа решаемой задачи. В случае, если в правой части правила предикаты зависят друг от друга через параметры, то выполнить такое правило параллельно не удастся. Например:

pred1(P1, P2, P3) :- pred2(P1, P2, T), pred3(P2, T, P3).

При вычислении `pred1(1, 2, P)`, выполнить правила `pred2` и `pred3` из правой части правила не удастся, поскольку для выполнения предиката `pred3` требуется знать значение переменной `T`, которая определяется при вычислении предиката `pred2`.

Однако и в случае такой зависимости может быть возможность для ускорения выполнения программы за счет использования параллельных вычислений. Если для вычисления предиката существует несколько правил, то возможна параллельная обработка этих правил. Например, один ВМ обрабатывает первую часть правил, а второй ВМ – вторую.

В случае, когда вычисляемому предикату подходят два правила, то есть, когда возможно несколько решений задачи, применение параллельных вычислений становится также оправданным, так как задача разбивается на две подзадачи, которые можно решать параллельно. Например, к правилу, указанному выше, добавим правило:

pred1(P1, 2, P3) :- pred4(P1, P3).,

которое соответствует дополнительному решению, когда `P2=2`.

Вычисление указанной выше цели, `pred1(1, 2, P)`, приведет к появлению двух решений. И в этом случае возможно выполнение правой части первого правила на одном ВМ, а правой части второго правила – на другом ВМ.

Как видно из приведенного примера, в прологе существует два типа параллелизма: *И-параллелизм* и *ИЛИ-параллелизм*. Первый порожден возможностью параллельного вычисления нескольких предикатов из правой части правила; второй – наличием нескольких правил, отвечающих вычисляемому предикату.

И-параллелизм обрабатывается путем порождения новых задач с их последующим объединением. Созданием задач и их завершением управляет процесс их породивший.

ИЛИ-параллелизм обрабатывается при помощи механизма повторного вычисления как показано на рисунке 1.

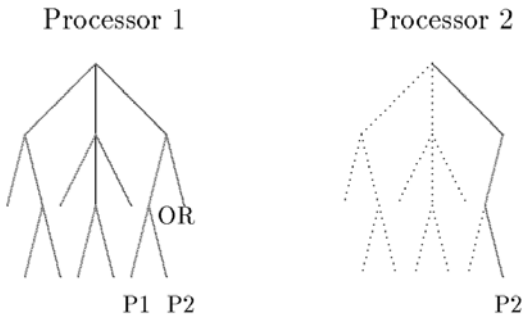


Рис. 1 Повторное вычисление при обработке ИЛИ-параллелизма

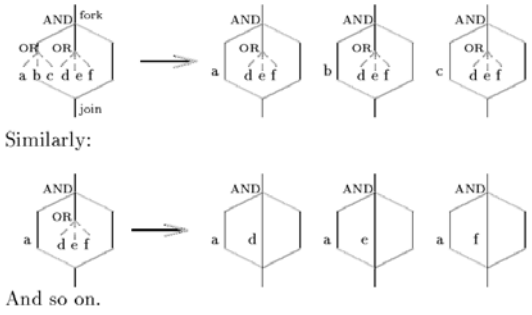


Рис. 2 Использование ИЛИ-параллелизма внутри И-параллелизма

Процессор 1 доходит до места, где возможен *ИЛИ-параллелизм*. После этого он порождает новую ИЛИ-задачу для вычисления цели "P2". При вычислении цели "P2" процессору 2 не приходится повторять весь поиск по дереву, он просто повторно вычисляет успешный путь до цели P2.

Важно также рассмотреть возможность комбинирования обоих типов параллелизма. Возможность и доказательство правильности распределенной реализации пролога приводится в [4]

ИЛИ-параллелизм, появляющийся внутри *И-параллелизма,* используется при помощи рекурсивного разбиения выполнения программы на независимые ветви вычислений (каждый при помощи обычного *И-параллельного* вызова как показано на рис. 2). Как видно из рис. 2, процесс использования *ИЛИ-внутри-И-параллелизма* сходен с раскрытием скобок для логического выражения.

Вычисление цели в распределенном прологе называется задачей. Для использования разных типов параллелизма создаются *И-задача* и *ИЛИ-задача*. Они же используются для смешанных случаев параллелизма. Если *И-параллелизм* встречается после *ИЛИ-параллелизма,* задачи порождаются так же, как и в случае чистого *И-параллелизма*. Если *ИЛИ-параллелизм* встречается внутри *И-параллелизма*, то *ИЛИ-задача,* порожденная в таком случае, должна повторно вычислить цель, чтобы найти новые решения. Но если делать это вслепую, то получатся *повторные решения*. Чтобы избежать этого, надо использовать **правило комбинации**, основная идея которого заключается в том, что надо

зафиксировать левые части решения в родительской цели и скомбинировать их со *всеми* решениями для правой части.

Программа на языке пролог имеет как декларативный, так и процедурный смыслы. При решении логических задач программисту следует уделять внимание именно декларативному смыслу программы, не вдаваясь в детали реализации её пролог-системой. В случае же решения вычислительных задач программу лучше рассматривать как процедурную, когда процесс вычислений рассматривается как последовательные вызовы процедур, последовательность которых определяется пролог-системой, исходя из структуры программы и результатов вычислений. Так, в нашем случае, логика пролога отвечает за автоматизацию распараллеливания программ, а процедурный смысл – за математические вычисления.

РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНОЙ ВЕРСИИ ЯЗЫКА ПРОЛОГ

Для реализации описанной версии языка можно использовать такую схему:

- Транслятор языка начинает выполнять программу на том ВМ кластера, на котором он был запущен.
- В случае обнаружения ситуации, допускающей параллельное выполнение программы, запускаются новые процессы на других ВМ кластера (возможно и на том же самом), исходя из их загрузки.
- Процесс, породивший задачи, дожидается их завершения и получает от них результаты вычисления.

Грамматика языка реализуется в виде обычных контекстно-зависимых грамматик. Для реализации транслятора была выбрана связка программ `lex + bison` – генераторы лексического и синтаксического сканеров. На данный момент с использованием этих средств реализован лексический и синтаксический блок компилятора.

Была реализована следующая грамматика (в нотации bison, но без символов действия):

```
input:
        | '\n'
        | ATOM '('  args ')' '.' input
        | ATOM '(' args ')' FOLLOW actlist '.' input;
args:
        | term
        | term ',' args;
actlist:    ATOM '(' args ')' acttail
        | math RELATION math acttail
        | VAR IS math acttail
        | term '=' term acttail;
acttail:
        | ',' actlist
        | ';' actlist;
math:       math '+' math
        | math '-' math
        | math '*' math
        | math '/' math
        | '-' math %prec NEG
        | '(' math ')'
        | mterm;
mterm:      VAR
        | ATOM
        | FLOAT
        | INT;
term:       mterm
        | '[' args ']'
```

| '[' args '|' VAR ']'
| BOOL;

Терминальные символы написанные большими буквами, вырабатываются лексическим блоком, реализованным при помощи `lex`. `ATOM`, `FLOAT`, `INT`, `BOOL` – соответствуют переменным базовых типов пролога, `VAR` – имени переменной, `FOLLOW` – значку `:-` пролога, `IS` – математическому присваиванию, а `RELATION` – любой из логических операций отношения.

В приведенной грамматике реализуется упрощенная версия языка пролог, в которой отсутствует поддержка структур данных, однако её хватит для демонстрации возможностей для параллельного программирования языка.

Для реализации сетевых взаимодействий можно использовать как существующие библиотеки для организации параллельных вычислений (MPI, PVM), так и стек TCP/IP. Оба подхода имеют как преимущества, так и недостатки. Первый обеспечивает большее удобство и уровень абстракции, независимость от конкретной реализации сетевой подсистемы, однако скорость взаимодействия будет ниже. Второй способ обеспечит более высокую скорость, однако требует реализации своего протокола взаимодействия.

Пролог обычно является интерпретируемым языком, однако для вычислительных задач, для которых он может быть использован в нашем случае, требуется большая скорость вычислений, поэтому желательна реализация компилятора указанного языка. Реализация компилятора, особенно мультиплатформенного, является сложной задачей, однако она даст *значительный* выигрыш в производительности. Желательна также реализация интерпретируемой версии для платформенной независимости и совместимость между интерпретируемой и компилируемой версиями.

Для взаимодействия с пользователем можно использовать либо связку *пролог + процедурный язык* или добавить возможность интерфейса с пользователем в саму пролог-систему. Второй вариант является более предпочтительным, так как он придает системе большую целостность, особенно если реализовать интерфейс пользователя, как в математических системах Maple и Mathlab. В этом случае систему можно использовать не только как язык программирования, но и как среду для решения задач математиками, не являющимися программистами-системщиками.

ЛИТЕРАТУРА

1. Братко И. Программирование на языке пролог для искусственного интеллекта: Пер. с англ. - М.:Мир, 1990, 560 с.
2. Элементы параллельного программирования / В.А. Васильковский, В.Е. Котов, А.Г. Марчук, Н.Н. Миренков / Под ред. В.Е. Котова. - М.:Радио и связь, 1983 - 240 с.
3. Корнеев В.В. Параллельные вычислительные системы. - М.:Нолидж, 1999 - 320 с.
4. Lourdes Adaujo. Correctness Proof of a Distributed Implementation of Prolog by means of Abstract State Machines. // Journal of Universal Computer Science. - 1997. - Vol. 3, № 5 - P. 568-602.
5. Egon Bőrger, Dean Rosenzweig. A Mathematical Definition of Full Prolog // Science of Computer Programming. -1995. - № 24 – P. 248-286.