УДК 004.432.42

СТИРЕНКО С.Г.
ГРИБЕНКО Д.В.
ЗИНЕНКО А.И.

ФОРМА ПРЕДСТАВЛЕНИЯ ФУНКЦИОНАЛЬНЫХ ПРОГРАММ ДЛЯ АВТОМАТИЧЕСКОГО ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ

Автоматическое распараллеливание, одна из важных задач параллельных вычислений, достаточно развито для императивных языков, но имеет частичное решение для функциональных. Предлагается внутренняя форма представления функциональных программ, позволяющая построить и частично оптимизировать граф зависимостей данных программы. На основе полученного графа программно определяются части программы, которые могут быть выполнены параллельно.

Automatic parallelizing is one of the important tasks in the parallel computing. Many solutions designed for imperative programming languages, but there are only partial solutions for functional ones. This notice propose an inner representation form of functional programs, which allows building a data dependency graph and partially optimizing it. Analysis of built graph determines program parts to be computed in multiple threads.

Подходы к распараллеливанию программ

Активное развитие вычислительной техники в основном обусловлено значительным улучшением элементной базы: уменьшением размеров логических элементов, увеличением тактовой частоты и плотности хранения данных. Однако в последнее десятилетие такой способ развития практически исчерпал свои возможности и новым подходом к увеличению скорости работы вычислительной техники стали параллельные вычисления [1]. Широкое распространение многопроцессорных (многоядерных) систем требует разработки параллельных программ, эффективно использующих вычислительные мощности. В то же время существует большое количество активно использующихся ранее написанных и отлаженных программ и библиотек для однопроцессорных систем. Преобразование их к параллельной форме зачастую требует больших затрат времени и ресурсов.

Наиболее простым решением проблемы адаптации уже готовых программ с доступным исходным кодом для многопроцессорных систем является преобразование их кода таким образом, чтобы обеспечить параллельное выполнение. Интерфейс программирования приложений OpenMP [2] предлагает для языков C/C++ и FORTRAN специальные директивы компилятора, в которых программист явно указывает в исходном коде возможность параллельного выполнения отдельных частей или блоков программы в исходном коде, необходимость синхронизации и взаимного исключения при работе. Однако на данный момент существуют реализации только для императивных языков программирования. Вместе с тем использование подобных средств также требует больших затрат ресурсов: анализ программы на возможность параллельного выполнения и размещение директив OpenMP выполняется вручную.

Другим возможным подходом к решению данной проблемы являются средства автоматического распараллеливания программ, которые проводят анализ исходного кода и размещают соответствующие директивы (для компиляторов) или непосредственно выполняют отдельные части программы параллельно (распараллеливающие интерпретаторы).

К таким относятся различные распараллеливающие интерпретаторы и компиляторы: Bert77 для языка FORTRAN; KAP, V-Ray и открытая распараллеливающая система OPC для языка C. Но практически все реализации автоматически распараллеливающих систем работают с императивными языками, в то время как решения для функциональных программ остаются неполными и требуют новых предложений. Основной сложностью при разработке таких средств является анализ зависимостей данных в программе.

Для проведения анализа зависимостей вычислений одних данных от результата вычисления других, как правило, используются графы зависимостей данных. Необходимо разработать форму внутреннего представления в автоматически распараллеливающем компиляторе (ин-

терпретаторе) функциональных программ, удобную для анализа зависимостей данных.

Функциональные языки программирования основываются на формальной системе лямбда-исчисления, созданной для анализа задачи вычислимости. Особенностями функциональных языков, отличающими их от императивных, является наличие анонимных вложенных функций, т. н. замыканий; возможность сохранить состояние программы и вызвать функцию с состоянием, отличным от текущего, т. н. продолжения; отсутствие циклов в императивном стиле и аналог их реализации через хвостовую рекурсию. Предлагаемая форма представления должна учитывать все перечисленные особенности и позволять легко анализировать их на предмет возможности параллельного выполнения.

Распараллеливание функциональных программ

В языках программирования, реализующих только функциональную парадигму, все данные являются неизменяемыми и могут быть связаны с именами переменных только один раз. Таким образом, в рамках данной парадигмы задача анализа зависимостей данных сводится к анализу взаимного расположения места записи данных в исходном коде, и все мест обращения к этим данным для чтения. Подобная задача уже реализована в концепции отложенных вычислений [3]. Компиляторы, использующие эту концепцию, размещают код вычисления значения переменной непосредственно перед первым обращением к ней. Однако большинство языков сейчас представляют комбинирование функциональной и императивной парадигмы, вводя для удобства написания программ переменные в функциональных программах, обработку исключительных ситуаций и другие средства императивных языков. Функции, не обращающиеся при работе к переменным верхнего уровня видимости, не осуществляющие операции ввода/вывода, не вызывающие исключительных ситуаций и их обработчиков являются чистыми (pure). Для чистых функций работают те же правила анализа.

Во многих LISP-подобных функциональных языках, в частности в языке Scheme [4], порядок вычисления аргументов функций явно не определён спецификацией. Тогда при разработке программы аргументы функции, если они являются функциями, не должны изменять одни и те же данные верхнего уровня видимости, иначе на системах, реализующих различный порядок вычисления аргументов, результаты вычисления функции с одинаковыми данными будут разными. Следовательно, аргументы одной функции можно вычислять параллельно, т. к. они не имеют общих зависимостей по данным.

Внутренняя форма представления

Все остальные конструкции языка предлагается преобразовать к специальной форме связывания имен, на основе конструкций let, letrec, let*, letrec*, определенных стандартом языка Scheme, на основе формальных правил преобразования. Для этих форм явно определен порядок вычисления аргументов:

- let, let* связывают одну или несколько пар имя-значение, которые могут быть вычислены в произвольном порядке, причем в случае совпадения имен, используются переменные верхнего уровня видимости;
- letrec, letrec* связывают одну или несколько пар имя-значение, которые должны быть вычислены строго в порядке перечисления, причем в значении каждого следующего имени могут использоваться значения любого из предыдущих.

Такое представление для данных может быть легко преобразовано в граф зависимостей данных, в котором все элементы конструкций let/let* будут находиться на одном уровне, а элементы letrec/letrec* последовательно подчинены. Тем не менее, такое представление не всегда является оптимальным, т. к. конструкции letrec могут изначально существовать в исходном коде программы, но зависимость по данным может отсутствовать или быть не полностью последовательной. Например в конструкции

```
(letrec (a 4)
        (b a+2)
        (c a+5))
```

переменные b и c зависят только от переменной a, при этом b не зависит от c. В таком случае вычисление b и c может быть выполнено параллельно, хотя конструкция предполагает строго последовательное вычисление.

Для анализа сложных операций и конструкций с определённым порядком вычислений предлагается во внутреннем представлении связывать со служебным именем каждую из вычисляемых конструкций. После этого анализ проводить по одинаковым правилам для поль-

зовательских и служебных имен. Например, конструкция

```
(define a (+ (+ 1 2) (+ 3 4)))
```

будет преобразована к виду

```
(letrec (_sys-1 (+ 1 2))
        (_sys-2 (+ 3 4))
        (a (+ _sys-1 _sys-2))).
```

Сложение более чем двух операндов не будет разделено попарно, а будет выполняться последовательно с накоплением результата:

```
(define a (+ 1 2 3 4))
```

будет преобразовано в

```
(letrec (_sys-1 (+ 1 2))
        (_sys-2 (+ _sys-1 3))
        (a (+ _sys-3 4))).
```

Данное ограничение связано с тем, что в общем случае применения оператора (функции высшего порядка) к большому числу аргументов требуется аналитически доказать ассоциативность операции, что нереализуемо в автоматическом режиме.

Формы связывания имен могут в качестве значения использовать не только непосредственные данные или переменные, но и функции, включая функции высших порядков. Для сохранения однородности формы представления и алгоритма анализа предлагается со всеми именами связывать не только значения, но и набор меток об использовании при вычислении данного значения других имен. Для реализации меток на уровне языка программирования рекомендуется использовать концепцию символов (‘() quotes) в функциональной парадигме. Для каждого связываемого имени необходимо сохранять набор меток, каждая из которых обозначает определённое действие с другим конкретным именем. Метки будут иметь вид ‘read_x, ‘write_x, ‘readwrite_x, где x – имя переменной, определенной ранее данной и видимой для неё. Значения, которые после анализа не обращаются ни к одной переменной, являются чистыми (‘pure). Значения, для которых невозможно однозначно определить имена, к которым они обращаются, например функции высших порядков, принимающие функции в качестве аргументов, помечаются для последовательного выполнения как ‘serial. Сопоставление меток различных значений позволяет определить значения, которые могут вычисляться параллельно.

‘pure-значения могут быть вычислены в любой момент времени до первого непосредственного обращения к ним, параллельно с любыми другими значениями, кроме ‘serial, т. к. не обращаются и не модифицируют никакие данные.

‘serial-значения всегда должны вычисляться последовательно в одном потоке выполнения, при этом все остальные потоки должны быть заблокированы.

‘read_x-значения могут быть выполнены параллельно с любыми другим ‘read- или ‘pure-значениями.

‘write_x-значения могут вычисляться параллельно с любыми ‘pure-значениями или такими ‘write_y-значениями, для которых x и y разные имена.

‘readwrite_x-значения комбинируют две предыдущих метки, и могут вычисляться параллельно аналогично ‘write_x значениям; однако данная метка предполагает в дальнейшем более подробный анализ обращений к именам.

Между вычислениями выражений, содержащими различные метки по одной переменной, которые производятся в разных потоках выполнения, обязательно должна произойти синхронизация всех потоков, в которых выполнялись вычисления с доступом к данной переменной.

Используя метки ‘write_x и ‘readwrite_x можно условно разделить все значения, связанные с именами на группы по признаку наличия меток. Результатом такого разделения является формальное правило: две функции из одной группы не могут быть выполнены параллельно. Добавлением к нему вышеописанные правила для ‘pure- и ‘serial-значений будет получена система правил параллельного вычисления значений, представленных в предлагаемой форме.

Граф зависимостей данных для подобной формы представления может быть оптимизирован путем сохранения в нём только тех рёбер, которые соответствуют меткам зависимостей между определёнными значениями. После данного преобразования к графу могут быть применены и другие способы оптимизации зависимостей.

Перспективы разработки распараллеливающего интерпретатора

Предложенная форма удовлетворяет перечисленным требованиям: является однородной и предполагает обработку по единому алгоритму анализа зависимостей данных, позволяет легко реализовать и проанализировать замыкания и продолжения, позволяет выполнить оптимизацию хвостовой рекурсии, преобразование к данной форме может быть выполнено на

уровне исходного кода автоматически после синтаксического анализа.

Дальнейшей работой над предлагаемой формой представления является другие способы анализа и оптимизации графа зависимостей данных после преобразования с метками. Также требуется проанализировать возможность представления в аналогичной форме не LISP-подобных функциональных языков программирования. Как отдельную возможность распараллеливающей системы возможно реализовать основанную на профилировании оптимизацию (profile-guided optimization / PGO). В подобной реализации некоторые элементы анализа, например ассоциативность операций или формально недоказуемые утверждения, возможно оставить для ручного размещения меток пользователем в интерактивном режиме.

Следующим важным решением для разработки автоматически распараллеливающего компилятора функционального языка программирования является предложение способа параллельного вычисления выражений, поддерживающего динамическую сборку мусора и другие служебные задачи в процессе выполнения, и оптимизирующего нагрузку на планировщик операционной системы. Также требуется разработать широко используемые чисто-функциональные потокобезопасные структуры данных, в частности очереди и деревья.

Список литературы:

1. *Sutter, H.* Software and concurrency revolution / H. Sutter, J. Laurus // Queue. – New York: ACM, 2005. – Vol. 3, № 7. – P. 54-62. – ISSN 1542-7730.
2. *Sato, M.* OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors / M. Sato // ISSS `02: Proceedings of the 15th international symposium on System Synthesis. – New York: ACM, 2002. – P. 109-111. – ISBN 1-58113-576-9.
3. *Guibas L. J.* Compilation and delayed evaluation in APL / L.J. Guibas, D. K. Wyatt // POPL `78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. – New York: ACM, 1978. – P. 1-8.
4. *Sperber M.* Revised 6 report on the algorithmic language Scheme / M. Sperber et. al. // Journal of Functional Programming. – Cambridge, UK: Cambridge Press, 2009. – Vol. 19, Supplement 1. – P. 1-301. – ISSN 0956-7968.
5. *Bacon, D.F.* Compiler transformations for high performance computing / D.F. Bacon, S.L. Graham, O.J. Sharp // ACM Computing Surveys. – New York: ACM, 1994. – Vol. 26, № 4. – P. 345-420. – ISSN 0360-0300.
6. *Hendren, L. J.* Parallelizing programs with recursive data structures / L. J. Hendren, A. Nicolau // IEEE Transactions on Parallel and Distributed Systems. – Piscataway, New Jersey, USA: IEEE Press, 1990. – Vol. 1, № 1. – P. 35-47. – ISSN 1045-9219.
7. Orbit: an optimizing compiler for scheme / D. Kranz, R. Kelsey, J. Rees et al. // ACM SIGPLAN Notices. – New York: ACM, 2004. – Vol. 39, № 4. – P. 175-191. – ISSN 0362-1340.
8. *Steel Jr., G. L.* Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful / G. L. Steele, Jr. // ACM SIGPLAN Notices. – New York: ACM, 2009. – Vol. 44, № 9. – P. 1-2. – ISSN 0362-1340.