НА ПУТИ К АВТОМАТИЗАЦИИ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Л.В. Городняя

«Машина поручает человеку то, что человек не сумел поручить машине»

Аннотация

В докладе рассматриваются схемы экспериментальной разработки параллельных программ, ориентированные на автоматизацию распараллеливания верифицируемых программ и настраиваемой кодогенерации.

Введение

В 1962 году К.Е. Айверсон, автор первого языка параллельного программирования APL, самим названием «A Programming Language» выразил понимание, что настоящее программирование – это параллельное программирование. И вот через 50 лет, на 20-ой Международной конференции «Конструирование компиляторов», состоявшейся в марте 2011-го года в Германии под председательством Дженса Кнопа (Jens Knoop), Мартин Одерски (Martin Odersky) в его приглашенном докладе констатировал «Ничем не скроешь фундаментальную трудность параллелизма». В докладах на этой конференции рассмотрены отдельные, ранее сложившиеся, средства реализации языков и систем программирования (ЯСП), пригодные для переноса в мир параллельного программирования. В их числе – «компиляция на лету», выделение чисто функциональных подмножеств и форм с однократным присваиванием, обратимая компиляция и интерпретация, транзакционная память, анализ достижимости действий, преобразования циклов над большими массивами и т.д.[1]

Несмотря на многочисленные призывы и конкурсы пока не появилось общих идей по новому поколению ЯСП, в которых бы параллелизм имел самостоятельное значение, а не рассматривался бы как пристройка к последовательному программированию. Нередко переход от последовательного к параллельному алгоритму сопровождается пересмотром решений как относительно структур данных, так и относительно методов управления вычислениями. Поэтому спецификация абстрактных схем управления должна быть приспособлена к согласованной реорганизации дисциплины доступа к структурам данных и выбора методов управления вычислениями.

Лидирующие в области поддержки параллелизма фирмы Intel и Microsoft предлагают разработчикам высокопроизводительных вычислений новые решения, направленные на преобразования программ в процессе их создания и распараллеливания:

1) Помощник по параллельному программированию (Parallel Advisor XE)
2) Компиляторы-отладчики и библиотеки (Parallel Composer XE2011)
3) Анализатор потоков и памяти (Parallel Inspector XE)
4) Профилировщик производительности (Parallel Amplifier XE)

Помощник по параллельному программированию анализирует исходный текст программы на языках Fortran или C до ее компиляции и отмечает узкие места, препятствующие распараллеливанию. Анализатор потоков и памяти исследует полученный в результате компиляции объектный код и выявляет неудачное распределение ресурсов, снижающее производительность программы. Профилировщик производительности позволяет наблюдать экспериментальную проверку достигнутой производительности программы.[2]

Технология применения всех этих инструментов предполагает, что программист по ходу дела многократно принимает решения об изменении исходного кода программы и в ручную включает в него необходимые прагмы, указывающие компилятору допустимость оптимизирующих преобразований, что делает работу компилятора по распараллеливанию программы проще и надежнее. Возникает проблема автоматизации преобразования программ с целью приведения их к виду, удобному для распараллеливания компилятором и настройки объектного кода на конкретную многопроцессорную конфигурацию. В докладе рассматривается схема подготовки параллельных программ с использованием библиотек преобразований исходного и объектного кода.

Общее представление

Обычно языки параллельного программирования включают в себя механизмы, характерные для разных парадигм. Это определяет необходимость трансформационного подхода к накоплению правильности программных решений при разработке и модернизации программ на разных языках в рамках общей системы программирования. Развитие ЯСП ориентировано на решение задач параллельного программирования на основе библиотечных модулей, обеспечивающих организацию процессов, или подъязыков, допускающих многопоточное программирование [3]. Это не исключает практику ручного распараллеливания ранее отлаженных программ, приведения их к виду, удобному для производственных систем поддержки параллельных вычислений [4]. Значительная часть таких работ носит технический характер и заключается в реорганизации

структур данных, изменении статуса переменных и включении в программу аннотаций, сообщающих компилятору об информационно-логических взаимосвязях. Существенным ограничением результата ручного распараллеливания является не только опасность повторной отладки алгоритма, но и его зависимость от характеристик целевой архитектуры.

Разнообразие моделей параллельных вычислений и расширение спектра доступных архитектур следует рассматривать как вызов разработчикам ЯСП, как проблему создания методов компиляции программ для многопроцессорных конфигураций. Язык должен допускать представление любых моделей параллелизма, проявляемого как на уровне решаемой задачи, так и реализуемого с помощью реальной архитектуры. Причем такое представление требует лаконичных форм и конструктивных построений, гарантирующих нужные свойства программ при их создании и реорганизации. Не менее важна расширяемость системы программирования по мере развития средств и методов параллельных вычислений. [5]

Рассматривая задачу формализации языков параллельного программирования как путь к решению проблемы адаптации программ к различным особенностям используемых многопроцессорных комплексов и многоядерных процессоров, приходится видеть, что решение этой проблемы требует разработки новых методов компиляции программ и развития методов описания семантики языков программирования, нацеленных на распараллеливание и настраиваемую кодогенерацию.

Применительно к высокопроизводительному программированию появляются высокоуровневые методы представления иерархии данных в системах программирования. Такие методы обеспечивают унификацию реализуемых понятий, включая абстрагирование данных и процессов в рамках единой методики представления программ на основе выделения базовых функций и структурирования схем управления вычислениями с помощью специально разрабатываемых абстрактных форм и конкретных шаблонов. В этом плане возрастает роль определения реализационной семантики языка программирования, уточняющего его операционную семантику для многопроцессорных конфигураций, в особенности задающего взаимодействие схемы управления вычислениями и конкретных вычислений.

Конструктивность

Результаты анализа системной поддержки конструктивных построений, гарантирующих правильность программ при их реорганизации, показывают, что, как правило, все сводится к комбинаторике многократно используемых компонентов, таких как структуры данных, функции, модули, классы объектов. Снижение сложности отладки программ достигается факторизацией программ исходя из определения структур данных и проявления подобия программ обработки данных структурам данных. Языки и системы функционального программирования допускают факторизацию более общего вида [6]. Они позволяют выделять такие компоненты как схемы управления, что дает возможность типизации управления, действий, моделей вычисления, реорганизации памяти и т.д.

Функциональное программирование вносит свой вклад в результативность распараллеливания техникой рекурсивных определений и отображений, параллелизмом обработки аргументов функций, ленивых вычислений, а также сведением понятия «условия» в ветвлениях к понятиям «страж» или «образец». Использование методов функционального программирования радикально снижает трудоемкость отладки программных компонент благодаря предпочтению достаточно универсальных определений и динамическому контролю корректности вычислений. В центре внимания – выделение базовых функций, реализуемых в наиболее общей форме. Благодаря этому снижается комбинаторная сложность тестирования комплексов из отлаженных компонент, что обеспечивает полноту и надежность параллельного программирования. Появление нового поколения языков программирования, таких как C# и F#, показывает общую тенденцию включения в системы программирования средств обработки кода программ, что позволяет при реализации параллельных алгоритмов решать задачи верификации и оптимизации программ, включая их распараллеливание.

Весьма существенно, что кроме собственно правильности, понимаемой как соответствие аргументов результатам, параллельные программы должны отвечать достаточно сложным критериям, таким как корректность синхронизации процессов, надежность, живучесть, справедливость и др., не отслеживаемые обычной схемой компиляции программ. В дополнение к понятию «ошибка» возникает не менее важное направление анализа разного рода тормозящих эффектов, снижающих производительность программ при их формальной корректности. Конструктивные методы параллельного программирования желательно нацеливать на обеспечение нужных свойств по построению, основных на определении расширяемых и трансформируемых схем, что приводит к трансформационно-операционной семантике языков параллельного программирования.

Трансформационно-операционная семантика

Согласно Венской методике определения языков программирования различимость компиляторов можно определять как отображение абстрактного синтаксиса в абстрактную машину [7]. Синтаксически различные языки с общей системой понятий могут быть сведены к общему абстрактному синтаксису. Абстрактная машина должна допускать реализацию на любой целевой архитектуре. При исследовании принципов структурного программирования (Э. Дейкстра) было показано, что любая императивно-процедурная программа может быть формально приведена к нормальной структурированной форме. При сравнении

парадигм операторного и функционального программирований было показано, что по произвольной операторной схеме программы можно вывести эквивалентную ей функциональную программу. На этом фоне задача трансформационной семантики – сведение программы к нормализованной форме, удобной для интерпретации программ, их распараллеливания и генерации настраиваемого объектного кода. Похожая техника применяется при сведении грамматик языка к форме, удобной для автоматизации построения анализатора, и при оптимизации программ.

Направление преобразований программ обычно связано с определенными критериями применимости и оптимальности, учитывающими результаты анализа логических и информационных связей. При организации параллельных процессов такие критерии обладают спецификой, отражающей особенности эксплуатации многоядерных архитектур. В частности, возрастает роль учета времени доступа к разнородной памяти и статического планирования загрузки процессоров наряду с обеспечением обратимости обработки данных и динамического управления производительностью вычислений. Поддержка такой семантики вычислений выходит за границы традиционных решений по реализации языков высокого уровня.

При декомпозиции трансформационно-операционной семантики языка параллельного программирования возникает коллекция нормализованных функциональных моделей отдельных аспектов управления процессами. Нормализация моделей направлена на ограничение сложности их анализа, приспособленность к интеграции с другими моделями, поддержку быстрого прототипирования многопоточных программ, верификацию различных свойств программных систем и их факторизацию в процессе разработки и усовершенствования. Такие механизмы могут быть включены в систему параллельного программирования.

Необходимость согласования большого числа разноплановых факторов приводит к бипланарному описанию семантики языков параллельного программирования (ЯПП), в котором разделены уровни абстрактных схем управления вычислениями и наполнения схем конкретными вычислениями. Чтобы избежать повторной компиляции при согласовании выбора схемы управления с целевой архитектурой, при формировании внутреннего представления многопоточных программ обычно используются методы смешанных вычислений и техника макрогенерации. Практичность таких методов обуславливается вполне конкретными, не редко диктуемыми аппаратурой, требованиями к фрагментам наполнения, связанными с целесообразностью достижения конечности отладки фрагментов. (Целостность действий, однократность присваиваний, одномерные вектора, функции без рекурсии, циклы со статически определенной кратностью, потоки действий, выполняемых по готовности данных - как арифметические выражения.)

Обычно операционная семантика языка программирования базируется на определении абстрактной машины, которая в случае многопроцессорных конфигураций становится сложной конструкцией из абстрактных процессоров. Выделение в схеме компиляции параллельных программ уровня трансформационной семантики позволяет пересмотреть формат абстрактной машины. Вместо базового исполнителя команд, определяемых как переходы от одного состояния к другому, становится естественным использовать абстрактный макроассемблер, допускающий настройку на конкретную конфигурацию при исполнении программы – настраиваемую кодогенерацию

Настраиваемая кодогенерация

Похожий механизм был предложен в форме символьного макроассемблера в качестве выходного языка системы БЕТА. В середине 1960-ых годов была создана система программирования АЛЬФА, входным языком которой был ранее разработанный под руководством А.П. Ершова АЛЬФА-язык. Система представляла собой многофазный оптимизирующий компилятор (АЛЬФА-транслятор) с входного языка в код М-20. В системе использовался ряд промежуточных, внутренних языков, разработанных специально для удобства анализа и отдельных преобразований программ. Вскоре возник замысел системы БЕТА, начинавшийся также с разработки входного языка (БЕТА-язык), но в условиях появления заметного числа новых серийных ЭВМ. Это привело к идее создания символьного макроассемблера в качестве выходного языка будущей системы БЕТА. Такой макроассемблер, получивший название языка СИГМА, был реализован Г.Г. Степановым. Язык СИГМА концептуально близок понятию абстрактной машины (АМ) Венской методике [7] с некоторым существенным отличием. АМ определяется как языково-ориентированный машинно-независимый семантический базис, реализация которого выполняется отдельно для каждой машины в предположении, что трудоемкость реализации невелика и необходимость переноса программ возникает не слишком часто – примерно раз в полгода. Язык СИГМА создан как языково-независимое представление программ, допускающее автоматизированную настройку на конкретную машину. Такое решение позволяет оперативно переносить программы в рамках широкого семейства машин, расширяемого по мере появления новых архитектур.

Экспериментальные верификаторы

При бипланарном описании семантики языка параллельного программирования наполнение программ может развиваться независимо от схем управления вычислениями, а схемы можно реорганизовывать без дополнительной отладки наполнения. Выбор целевой архитектуры и зависящей от нее схемы управления может быть обусловлен как архитектурой, так и критериями эффективность (компактность программы, объем памяти, скорость обработки данных). Абстрактные схемы управления вычислениями определяются независимо от

наполнения узлов схемы. Они играют роль макетов или моделей программ и работают подобно макросам (открытая подстановка), но с контролем соответствия параметров объявленным синтаксическим типам фрагментов. В языках программирования, поддерживающих параллелизм, таких как императивный C# и функциональный F#, возможность манипулировать внутренним кодом программы может рассматриваться как средство реализации таких макросов.

Выделение схемного уровня упрощает включение в схему разработки программ механизмов верификации (подобие модели или соответствие аксиомам), а на их основе возможна проверка программ на правдоподобие, логический вывод свойств, выполнение индуктивных и дедуктивных построений. Кроме того техника отладки программ обогащается возможностью привлечения протоколов ранее выполненных вычислений и приведения программ к нормальным формам, удобным для сведения к базовым/стандартным моделям параллельных вычислений.

При теоретико-экспериментальном исследовании параллельных программ нередко используют ручное создание эквивалентов на языках спецификаций, таких как Promela для Spin-анализатора. [8] Promela – язык верификации программ с помощью моделей, обеспечивающий методы абстрагирования распределенных систем. Он рассчитан на проверку отдельных аспектов поведения процессов. Полная верификация складывается из серии шагов конструирования все более точных Promela-моделей программы. Каждая модель может быть верифицирована Spin-анализатором с учетом разнотипных допущений относительно контекста исполнения программы. Единожды установленная корректность программы распространяется на все ее последующие модели. Promela-модель строится из процессов, каналов сообщений и переменных. Процессы – это глобальные объекты для представления независимых составляющих распределенной системы. Каналы сообщений и переменные могут быть глобальными или локализованными внутри процесса. Процессы задают поведение, а каналы сообщений и переменные определяют контекст исполнения программы. Значения переменных типизированы и могут быть агрегированы в вектора и структуры. Поведение процесса задается как схема управления изменением состояний контекста исполнения программы. Для Spin-анализатора Promela-модель сопровождается специальными ltl1-формулами, специфицирующими условия корректности верифицируемой программы. Такая техника верификации достаточна для обнаружения мало очевидных ошибок времени выполнения, наличия тупиков, оттеснения выполнимых действий, а также нарушения требований справедливости, корректной завершаемости, причинно-следственного и темпорального порядка и др. формализуемых условий. Переход от практики ручного моделирования к формальному выводу Promela-моделей и/или ltl-формул может стать важным звеном обеспечения надежности параллельного программирования.

Заключение

Таким образом, путь к автоматизации параллельного программирования лежит через создание системы конструирования языково-зависимых библиотек преобразований программ и логических формул, компиляция которых поддержана верификацией и настраиваемой кодогенерацией программ с раздельным формированием схем управления и их наполнения.

Следующий шаг – разработка языка высокого уровня параллельного программирования, приспособленного к ознакомлению студентов с разнообразием моделей вычислений и с методами верификации программ, без которых надежность параллельного программирования весьма проблематична. Для нужд этого шага требуется строгая формализация семантики в трансформационно операционном стиле. Представленный в докладе подход к выбору механизмов для системы параллельного программирования предназначен для поддержки экспериментов по разработке новых языков, ориентированных на учебно-исследовательские проекты в области создания распределенных информационных систем.

ЛИТЕРАТУРА:

1. LNCS 6601. Jens Knoop Compiler Construction. 20th International Conference, CC 2011. Held as Part of the Joint European Conferences on Theory and |Practice of Software, ETAPS 2011 Saarbr?cken, Germany, March 26 – April 3, 2011. Springer. 330 p.
2. Рогожин К. http://www.swconf.ru/Novosibirsk/ Материалы фирм Intel и Microsoft, представленные на семинаре для разработчиков
3. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления.– СПб.: БХВ-Петербург, 2002. – 608 с.
4. Катаев Н.А.Статический анализ последовательных программ в системе автоматизированного распараллеливания САПФОР# / Новосибирск, ПАВТ 2012.
5. Городняя Л.В. Современная система параллельного программирова-ния. Лаконизм. Конструктивность. Расширяемость / Шестая Сибирская конференция по параллельным и высоко-производительным вычислениям /Под ред. проф. А.В. Старченко.– Томск: Изд-во Том. ун-та, 2012. с. 29-36 (http://ssspc.math.tsu.ru/6TH/sbornik.pdf)
6. Хендерсон П. Функциональное программирование. Применение и реализация. М.:Мир. 1983. 349 с.
7. Оллонгрен А. Определение языков программирования интерпретирующими автоматами. Пер.с англ. Серия: Математическое обеспечение ЭВМ. М. Мир 1977г. 288 с. ил. http://www.vdmportal.org/twiki/bin/view

8. Савенков К. Верификация программ на моделях. / Курс ВМК МГУ имени М.В.Ломоносова. http://savenkov.lvk.cs.msu.su/mc/lect02.pdf 1 Linear time temporal logic formulae