Современная система параллельного программирования. Лаконизм. Конструктивность. Расширяемость

Л.В. Городняя
Институт систем информатики им. А.П. Ершова СО РАН,
Новосибирск

Статья посвящена проблеме разработки систем параллельного программирования и подходов к реализации языков параллельного программирования. В центре внимания вопросы выбора лаконичных форм представления программ, обеспечения конструктивных построений, гарантирующих сохранение свойств программ при их реорганизации, и поддержки расширяемости языка программирования по мере развития средств и методов параллельного программирования.

Введение

«Ничем не скроешь фундаментальную трудность параллелизма» – так прозвучало заявление Мартина Одерски (Martin Odersky) в его приглашенном докладе на 20-й Международной конференции «Конструирование компиляторов», состоявшейся в марте 2011 г. в Германии под председательством Дженса Кнопа (Jens Knoop) [1]. Несмотря на многочисленные призывы и конкурсы пока не появилось общих идей по новому поколению языков и систем программирования (ЯСП), в которых бы параллелизм имел самостоятельное значение, а не рассматривался бы (по понятным причинам) как пристройка к традиционному программированию.

Современное развитие ЯСП фактически ориентировано на решение задач параллельного программирования. Как правило, новые ЯСП включают в себя библиотечные модули, обеспечивающие организацию процессов, или подъязыки, допускающие многопоточное программирование [2]. Это, увы, не исключает реальную практику ручного распараллеливания ранее отлаженных обычных программ, приведения их к виду, удобному для применения производственных систем поддержки параллельных вычислений. Значительная часть таких работ носит технический характер и заключается в систематической 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 жесткая 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 многопоточных программ для многопроцессорных конфигураций. Язык должен допускать представление любых моделей параллелизма, проявляемого на уровне решаемой задачи или 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 языка по мере развития средств и методов параллельных вычислений.

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 средств и методов описания семантики языков программирования.

Лаконизм

История языков программирования накопила целый ряд примеров лаконичных форм представления программ, начиная с умолчаний и неявных циклов в языке Fortran. С точностью до реализационной семантики при разработке языков параллельного программирования можно унаследовать языковые конструкции и механизмы из привычных парадигм программирования и зарекомендовавших себя языков параллельного программирования.

Прежде всего, это алгебраические механизмы просачивания функций и операций относительно структур данных, предложенные в первом языке параллельного программирования APL. Дальнейшее упрощение изобразительных средств управления параллелизмом дает предложенный в языке Sisal подход к неявному распараллеливанию циклов на основе выстраивания пространства итераций по пространству обрабатываемых данных.

Функциональное программирование вносит свой вклад техникой рекурсивных определений и отображений, параллелизмом обработки аргументов функций, ленивых вычислений, а также сведением понятия «условия» в ветвлениях к понятиям «страж» или «образец». Объектно-

ориентированное программирование активизировало перегрузку операций и функций. Более специфичные и эффективные формы выражений возникают при обработке составных значений с помощью фильтров и проекций в новых мультипарадигматических языках. Как правило, лаконизм достигается расширением используемых понятий или варьированием их реализации.

Конструктивность

Результаты анализа сложившейся практики системной поддержки конструктивных построений, гарантирующих правильность программ при их реорганизации, показывают, что, как правило, все сводится к комбинаторике многократно используемых компонент, 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 форматам структур данных. Языки и системы функционального программирования допускают факторизацию более общего вида. Они позволяют выделять такие компоненты, как схемы управления, что дает возможность типизации управления, действий, моделей вычисления, реорганизации памяти и т.д. Эффективность параллельного программирования потребует факторизации относительно схем управления и дисциплины доступа к памяти.

Появление нового поколения языков программирования, таких как C# и F#, показывает общую тенденцию включения средств обработки кода программ, что позволяет при реализации параллельных алгоритмов решать задачи верификации и оптимизации программ, включая их распараллеливание. Следует отметить, 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.

Тем не менее в докладах на конференциях по методам компиляции пока представляют отдельные, ранее сложившиеся, средства реализации ЯСП, лишь адаптированные к переносу 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.п. [1]. Конструктивные методы параллельного програм-

мирования желательно нацеливать на обеспечение нужных свойств по построению.

Расширяемость

Следует отметить проблему расширяемости ЯСП по мере развития средств и методов параллельных вычислений, обусловленную высоким темпом прогресса в области элементной базы и информационных технологий в целом. Многие понятия языка программирования – схемы управления программой, образующие программу действия, вычисления и данные – при переходе к параллельному программированию претерпевают изменения, и возникает необходимость в дополнительных понятиях.

Основные методы представления вычислений связаны с использованием неявных циклов, позволяющих избежать выписывания однотипных схем над стандартными структурами данных типа многомерных векторов. Так, например, результат операции над скалярами в языках параллельного программирования распространен на произвольные однородные структуры данных. Список операций, допускающих распространение, определен реализацией. Обычно это арифметические операции (+, −, ×, /). Этот метод может использоваться более широко распространением на технику применения функций, что повышает лаконизм выражений.

Из бинарных операций можно конструировать фильтры. Результат фильтрации исчезает из аргумента – он переносится в другую структуру данных или сохраняется как значение. Структура из фильтров дает структуру из результатов их применения к одному и тому же аргументу. Результативность вычислений можно повышать с помощью специально устроенных данных – так называемых «рецептов». Это отложенное исполнение фрагмента программы, его замыкание, т.е. пара из действия и контекста. Общие решения по обеспечению лаконизма, конструктивности и расширяемости приводят к трансформационной семантике языка программирования.

Трансформации

Трансформационная семантика обеспечивает сведение конструкций языка программирования к его базовым средствам, что позволяет упростить операционную семантику, а также выбрать реализационное ядро системы программирования при его экспериментальной раскрутке. 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.

Задача трансформационной семантики – сведение программы к нормализованной форме, удобной для интерпретации программ или генерации исполнимого кода. В случае многопоточных программ преобразование сети потоков нацелено на сведение к однородной системе потоков, однозначно отображаемых на заданный комплекс процессоров – размещение потоков по процессам или назначение процессоров для выполнения потоков. Такое требование можно выразить формулой

[(b0: ; b1: ; … bK:), p1!(b0: d0i; …), … pN!(b0: d0j; ...)],

где **i** и **j** обозначают принадлежность потоку,

, – параллельное или одновременное исполнение,

; – последовательное исполнение,

! – назначение процессора,

b0:, b1:, … – барьеры,

p1, pN, … – процессоры,

d0i, d0j, … – действия,

(b0:; b1:; … bK:) – шкала событий/барьеров,

pM!(b0: d0i; …) – программа (последовательность) действий для процессора **pM**.

В такой как бы «причесанной» форме все потоки начинаются с барьеров и общая шкала событий упорядочена так, что последовательность событий потока ей не противоречит. Можно считать, что процессоры включаются сами. Шкала событий содержит списки ожидающих потоков. Действия, выполняемые процессорами, соотнесены с их исходными потоками.

Достаточно простые преобразования сети потоков позволяют варьировать схемы потоков и многие конструкции языка программирования сводить к взаимодействию простых потоков:

(A;B) ↔ (a:A; b:B) # – расстановка – стирание барьеров.

(a:A; b:B) ↔ [[a:A, b:B], (a: ; b:)] – вынесение последовательного управления в отдельный поток – восстановление последовательности действий.

(a: A; b: B) ↔ [(a: A; b:), (b: B)] – разрез последовательности – перенос «хвоста», если нет локальной информационной зависимости между **A** и **B**.

(a: A; b: B) ↔ [a: A, b: B] – разбиение последовательности на потоки – сплющивание линии в слой, если **A** и **B** информационно не связаны.

[a: A, b: B] ↔ { (a: A; b: B) | (b: B; a: A)} – слияние потоков в одну последовательность – вытягивание слоя в линию. Выбор варианта требует учета последовательности барьеров в других потоках и общей шкале событий. Критерий оптимальности – объем выполнимых вычислений.

((a: A; b: B) ; c: C) ↔ (a: A; (b: B ; c: C)) ↔ (a: A; b: B ; c: C).

[[a: A, b: B] , c: C] ↔ [a: A, [b: B , c: C]] ↔ [a: A, b: B, c: C].

([A,B];C) ↔ [(A;C), (B;C)] – исключение слияний – слияние совпадающих продолжений.

[(A;B),C]↔ ([A,C]; [B,C]) – распределение параллельного потока – слияние совпадающих потоков.

[[a: A], [a: B]]↔ [a: [A, B]] – варьирование числа одновременных потоков.

Обратимость преобразований и чувствительность их результата к информационным связям между фрагментами программы требуют формализации критериев применимости трансформаций и выбора подходящего варианта. Можно констатировать, что выяснение информационной связанности действий **B** и **A** сводится к проверке существования контекста, в котором различны результаты программы **C** при изменении порядка вычислений **B** и **A**.

[(A; B), C] ≠ [(B; A), C].

Из числа базовых средств можно вывести ветвления, циклы и вызовы функций, реализуя их средствами синхронизации потоков:

(if A then B) ↔ [(A;B:);(B:B)] – сведение обхода к синхронизации и обратно.

При отладке формируется ряд контекстов, на которых демонстрируются конкретные свойства фрагментов, из которых собирается полная программа. Это контексты для отдельных потоков, для пар синхронизованных потоков, для интегрированной из потоков программы, а кроме того, контексты для удостоверения наличия-отсутствия информационных связей между фрагментами. Возможны пользовательские преобразования схем управления процессами, что позволит не только минимизировать «ручную» аранжировку распараллеливаемых программ, но и даст основу для формирования библиотек преобразования схем программ.

Абстрактная машина

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. (Похожий механизм был реализован в форме символьного макроассемблера в качестве выходного языка будущей системы БЕТА, получивший название языка СИГМА [3]).

При подготовке примеров для демонстрации учебного языка параллельного программирования были приняты следующее ограничения:

− взаимодействия потоков выполняются только через синхронизацию;
− одновременно исполняемые фрагменты могут обмениваться данными через общую память;
− при взаимоисключении каждый вариант работает в своей копии контекста;

– поддерживается список восстановления многократно выполняемых фрагментов.

Заключение

Данной статье предшествует работа по определению языка начального обучения параллельному программированию Кубик, в которой представлена базовая модель организации взаимодействующих процессов, позволяющая показать природу трудностей параллельного программирования и типичные пути решения известных проблем. Программа факультативного курса для школьников по параллельному программированию на базе этого языка изложена в материалах секции «Информатика образования» Ершовской конференции по информатике в июне 2011 г. [4].

Следующий шаг – разработка языка высокого уровня параллельного программирования, приспособленного к ознакомлению студентов с более сложными моделями вычислений и с методами верификации программ, без которых надежность параллельного программирования весьма проблематична. Для нужд этого шага потребуется более строгая формализация семантики в трансформационно-операционном стиле. Представленный в статье подход к определению языка 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 Saarbrьcken, Germany, March 26 – April 3, 2011. – Berlin: Springer, 2011. – 330 p.
2. **Воеводин В.В., Воеводин Вл.В.** Параллельные вычисления. – СПб.: БХВ-Петербург, 2002. – 608 с.
3. **Степанов Г.Г**. [http://www.iis.nsk.su/files/artcles/mozaika_stepanov_sigma.pdf].
4. **Городняя Л.В.** О курсе «Начала параллелизма» // Ершовская конференция по информатике. Сек. Информатика образования. 27 июня – 1 июля 2011 г. – Новосибирск, 2011. – С. 51–54.