УДК 681.5

П.А. ИВАНЕНКО, А.Е. ДОРОШЕНКО

МЕТОД АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ АВТОТЮНЕРОВ ДЛЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Аннотация. Представлена формальная модель метода автоматизированной настройки параллельных приложений (автотюнинга). Описана программная реализация этой модели в виде гибкой системы программных средств для автоматической генерации автотюнеров, которая основана на системе переписывания термов и использовании экспертных знаний в качестве источника оптимизационных преобразований.

Ключевые слова: автотюнинг, параллельные вычисления, оптимизация программ, системы переписывания термов.

ВВЕДЕНИЕ

Для решения сложных научно-технических задач требуются большие вычислительные мощности параллельных систем, поэтому этап оптимизации прикладных программ для конкретной высокопроизводительной среды вычислений занимает значительное место в процессе их разработки. Классическое высказывание «эффективные алгоритмы всегда лучше суперкомпьютеров» [1] точно отражает проблему эффективного использования вычислительных ресурсов: наличие мощного вычислителя не гарантирует успешного выполнения задачи в приемлемых временных рамках.

Программирование эффективных алгоритмов всегда было нелегкой задачей, которая еще более усложняется в связи с массовым переходом к использованию многоядерных микропроцессоров. До этого перехода мощность процессоров удваивалась каждые 18 месяцев (закон Мура), в результате чего старые программы работали в два раза быстрее на новых моделях вычислителей. В настоящее время рост производительности системы вследствие увеличения количества ядер одного вычислителя при почти неизменной тактовой частоте ядра может ограничиваться законом Амдала [2]. Эффективный параллельный алгоритм, кроме удачной декомпозиции задачи на независимые подзадачи, должен минимизировать затраты на синхронизацию вычислений и пересылку данных, что весьма сложно сделать без учета архитектуры среды исполнения.

Традиционным способом автоматического получения кода параллельных программ являются так называемые «распараллеливающие компиляторы» [3, 4]. Однако, несмотря на значительное улучшение качества автоматического распараллеливания в последние годы, до сих пор область его применения ограничивается программами с достаточно простыми схемами вычислений. Развитие этого подхода затруднено вследствие большой сложности статического анализа кода, разнообразия архитектур вычислительных систем, а также постоянного их усложнения.

Альтернативой распараллеливающим компиляторам является автотюнинг [5] — новый, гибкий и эффективный подход. Он позволяет эмпирически в автоматическом режиме подобрать наилучший вариант программы для данной целевой среды выполнения. Главное достоинство этого подхода — абстрагирование логики программы от численных характеристик вычислительной системы таких, как количество ядер, размер кэша процессоров или скорость доступа к различным уровням памяти. В результате параллельный алгоритм оперирует высокоуровневыми понятиями из предметной области задачи (размер и количество независимо выполняемых подзадач) или особенностями численного метода решения

© П.А. Иваненко, А.Е. Дорошенко, 2014

задачи (например, метод обхода структуры данных). Однако недостатком автотюнинга могут быть значительные (хотя и однократные) временные затраты на оптимизацию программ: если количество параметров, релевантных производительности программы, велико, затраченное время может измеряться часами или даже днями. Кроме того, возникает необходимость написания дополнительного приложения — непосредственно автотюнера.

Хорошим решением для устранения описанных недостатков автотюнинга может быть использование специализированных инструментальных систем, которые работают с исходным кодом программы, основываясь на экспертных знаниях (expert knowledge) разработчика, представленных в виде некоторых метаданных (прагмы, аннотации и др.). Это позволяет существенно уменьшить количество испытуемых вариантов программы и, следовательно, сократить время оптимизации. Работа системы с исходным кодом программы, в свою очередь, избавляет разработчика от задачи написания дополнительного приложения-автотюнера. Такая система автотюнинга в отличие от широко известных библиотечных систем ATLAS [6] и FFTW [7] не зависит от области решаемой задачи и универсальна в своем классе.

Рассматриваемая в настоящей работе система автотюнинга TuningGenie использует описанный выше подход и во многом схожа с языковым расширением для автотюнинга Atune-IL [8]. Существенным отличием TuningGenie является выполнение трансформаций исходного кода программы с помощью системы обработки термов, основанной на технике переписывающих правил. Представление кода программы в виде терма и манипулирование им с помощью переписывающих правил (TermWare [9, 10]) позволяет измененять код и структуру программы в декларативном стиле, что значительно расширяет возможности системы.

Далее описано построение формальной модели системы автотюнинга TuningGenie, доказаны основные свойства выполняемых преобразований, приведены инструментарий системы и примеры ее применения для задач параллельного программирования.

АВТОТЮНИНГ

Рассмотрим детальнее концепцию автоматической программной оптимизации — автотюнинга.

Ключевой особенностью данного подхода является то, что оптимизация выполняется автоматизированным способом с помощью эмпирических испытаний, проводимых программой-автотюнером на том же оборудовании, на котором далее будет выполняться оптимизируемая прикладная программа. Затем на основе анализа проведенных испытаний и выполненения оптимизирующих преобразований на уровне исходного кода получается результирующая программа, оценка производительности которой может давать ощутимо лучшие результаты по сравнению со статическим анализом исходного кода.

Конечно, полностью автоматизировать процесс не возможно, это объясняется несколькими причинами. Во-первых, все варианты оптимизируемой прикладной программы задаются разработчиками, поскольку они имеют опыт в области решаемой задачи и знают, какие модификации программы могут существенно повлиять на производительность. Во-вторых, только разработчик может правильно оценить производительность и эффективность прикладной программы, поскольку часто значимыми являются не только временные затраты на выполнение, а и качество получаемого результата вычислений.

Следовательно, главная задача любой системы автотюнинга заключается в минимизации взаимодействия с разработчиком на этапе оптимизации прикладной программы, а также в сокращении времени разработки этой программы при достижении достаточного уровня производительности результирующего кода.

Предлагаемая в настоящей работе система автотюнинга в полной мере решает эту задачу за счет инкапсуляции экспертных знаний разработчика в исходном коде программы в виде специальных метаданных. Такой подход позволяет автоматически генерировать программу-автотюнер, сокращает затраченное на эксперимент время за счет минимизации количества испытуемых вариаций исходного кода и требует минимальных изменений исходного кода программы. Разработчик должен дополнительно задать метрику эффективности полученного кода.

Любой механизм автотюнинга выполняет задачу адаптации оптимизируемого программного обеспечения (ПО) к условиям его выполнения, которые можно разделить на четыре группы:

- особенности входных данных, например, их размер;
- свойства архитектуры процессоров, размер кэшей и т.п.;
- условия вычислительной среды, например, наличие других задач, выполняемых одновременно с оптимизируемым ПО;
- свойства установленного в операционной системе ПО (используемые библиотеки, оптимизация, выполняемая компилятором и т.п.).

В данной работе не рассматривается задача адаптации ПО к динамически изменяемым характеристикам вычислительной среды. Свойства доступных программе ресурсов (количество процессоров, тактовая частота, скорость доступа к оперативной памяти и т.п.) считаются неизменными как для этапа оптимизации, так и для последующего выполнения программы. Во время оптимизации осуществляется наблюдение за производительностью ПО в зависимости от его конфигурируемых параметров. В результате выбирается наилучшая конфигурация, которая используется до тех пор, пока не изменятся характеристики вычислительной среды (в таком случае придется заново провести оптимизацию).

Для оптимизации ПО с учетом изменяемых в реальном времени условий обычно выделяют ключевые характеристики среды выполнения (СВ), значения которых легко оценить. Далее проводится анализ производительности ПО в зависимости от его конфигурации при различных состояниях СВ. Система управления, которой может быть автотюнер, исходя из полученных результатов, выбирает наилучшую конфигурацию для текущего состояния СВ во время выполнения прикладной программы.

Главное преимущество рассматриваемого подхода — это адаптация ПО к неявным особенностям условий выполнения. Этот подход позволяет абстрагироваться от конкретных характеристик СВ и достаточно легко подбирать оптимальный вариант ПО для данных условий. Именно вследствие этого свойства эмпирическая адаптация, несмотря на большую сложность адаптации ПО к динамически-изменяемым условиям, стала доминирующей методологией автотюнинга.

На рис. 1 представлена общая схема итеративного процесса автотюнинга, выполняемого системой TuningGenie.

Вначале извлекается и анализируется метаинформация исходного кода, в результате формируется множество конфигураций оптимизируемой программы, которое представляет совокупность характеристик вычислительного метода, влияющих на производительность и/или точность вычислений (например, зернистость параллелизма). Обозначим это мно-

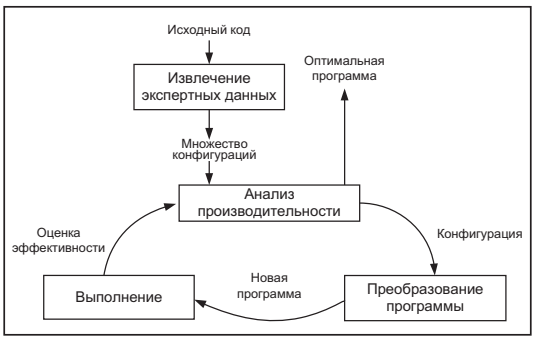

Рис. 1

жество C. Поиск наиболее эффективной вариации выполняется в пределах этого множества. Каждая конфигурация задает уникальную трансформацию кода, в результате которой генерируется новый вариант программы. Далее выполняется эмпирическая оценка эффективности. Результат передается автотьюнеру, который ассоциирует его с текущей конфигурацией, на этом итерация тьюнинга заканчивается. Множество конфигураций, для которых была проведена эмпирическая оценка, обозначим $C_{\text{exp}} \in C$. По окончании всех итераций тьюнинга (их количество в общем случае равно мощности $|C|$ множества конфигураций) TuningGenie выбирает оптимальную конфигурацию среди элементов C_{exp} и генерирует по ней тот вариант исходной программы, который далее будет выполняться в целевой СВ.

Как уже отмечалось, затраченное на автотюнинг время прямо пропорционально мощности множества $|C_{\text{exp}}|$. Иногда для правильной оценки производительности оптимизируемой программы необходимо для каждой итерации тюнинга полностью выполнить последовательность вычислений, производимых программой, что значительно увеличивает временные затраты. В таком случае рационально введение дополнительного временного ограничения. В случае, когда множество C чересчур велико, возможно использование оптимизационных поисковых алгоритмов для выбора следующей испытуемой конфигурации [11]. Оба подхода существенно уменьшают мощность $|C_{\text{exp}}|$, но не гарантируют нахождение глобального максимума для множества C.

В качестве альтернативного подхода к уменьшению времени выполнения автотюнинга можно рассмотреть написание небольшой и, следовательно, быстрой тестовой задачи, результаты выполнения который зададут более узкое исходное множество конфигураций C. Подобное решение для оптимизации вычислений, выполняемых на графическом ускорителе, рассмотрено в [12]. Далее предложен инструментарий, позволяющий интегрировать выполнение такой тестовой задачи в процесс автотюнинга. Отметим что, несмотря на то, что эта концепция не имеет недостатков двух предыдущих подходов, такое решение требует дополнительных ресурсов и существенной модификации исходного кода оптимизируемой программы.

РАСШИРЕННАЯ МОДЕЛЬ PRAM

Для более детального формального описания всего процесса оптимизации, выполняемого автотюнером (см. рис. 1), возьмем за основу классическую абстрактную модель PRAM [13] и дополним ее следующими ограничениями:

- предположим, что вычисления проводятся на параллельной системе, где единицей оборудования вычислителя является «процессор» (соответствует понятию ядра в современных вычислительных системах), что количество процессоров не ограничено, но фиксировано в каждом конкретном случае (обозначим числом N), а также, что все процессоры однородны;
- размер локальной памяти (регистров) каждого процессора ограничен числом M_{loc}, данные, не помещающиеся в локальной памяти, хранятся в общей памяти M_{shar}, времена доступа к локальной T_{loc} и общей памяти T_{shar} различны, причем $T_{\text{loc}} << T_{\text{shar}}$;
- одновременный доступ к общей памяти регулируется стратегией Concurrent Read Exclusive Write (CREW), допускается параллельное чтение из одной ячейки памяти, но только один процессор может записывать данные в конкретную ячейку памяти в единицу времени, остальные процессоры в это время находятся в состоянии ожидания.

Обозначим $T_{N_{\text{ew}}}$ общее время, затраченное процессорами на ожидание разблокировки общей памяти для записи.

Далее такую расширенную модель будем называть PRAM*.

Инструкции этой модели выполняются в три этапа:

1) чтение данных из общей памяти, если они не находятся в регистрах процессора (и если они нужны для выполнения инструкции);

2) локальные вычисления;

3) запись данных в общую память (если требуется).

Время, затрачиваемое i-м процессором на доступ к памяти (этапы 1 и 3 при условии отсутствия блокировок во время записи), вычисляется следующим образом: $T_{i_{\text{mem}}} = T_{i_{\text{loc}}} + T_{i_{\text{shar}}}$. Как видно из формулы, оно состоит из затрат на доступ к локальной и глобальной памяти.

Введем понятие синхронизации локальных вычислений и соответствующих временных затрат. Если выполняется критическая секция, то все процессоры, кроме активного, находятся в режиме ожидания и выполнение таких секций аналогично последовательной версии программы в модели RAM. Обозначим $T_{N_{\text{seq}}}$ временные затраты на такие критические секции и $T_{N_{\text{par}}}$ — время «чистых» параллельных вычислений.

Введенные ограничения позволяют более точно охарактеризовать архитектуру среды вычисления, а также выполняемых автотюнером оптимизирующих преобразований.

Общее время вычислений параллельной программы описывает формула

$$T_N = T_{N_{\text{par}}} + T_{N_{\text{seq}}} + T_{N_{\text{ew}}}.$$

Задача автотюнера заключается в минимизации временных затрат на синхронизацию $T_{N_{\text{seq}}} + T_{N_{\text{ew}}}$, а также сокращении времени $T_{N_{\text{par}}}$.

Временные потери на синхронизацию можно сократить за счет оптимального распределения вычислительных задач между процессорами, например, используя традиционную схему крупноблочного распараллеливания, при которой входящие данные распределяются таким образом, чтобы каждый процессор выполнял независимо максимально крупный блок вычислений и при этом все процессоры были бы равномерно загруженными. Возможна также структурная модификация выполняемых программой вычислений. Далее приведен пример такого преобразования для специфического класса итеративных алгоритмов с барьерной синхронизацией в конце каждой итерации. Эти преобразования направлены на улучшение программной реализации такой парадигмы параллельного программирования, как параллелизм по данным (геометрический параллелизм) [14].

Для сокращения времени параллельного выполнения $T_{N_{\text{par}}}$ представленная в работе система автотюнинга осуществляет преобразования, направленные на оптимизацию использования локальной памяти, т.е., вычисления распределяются между процессорами так, чтобы они помещались в локальную память, доступ к которой на порядок быстрее, чем чтение/запись в глобальную память. Таким образом, для каждого процессора $T_{i_{\text{shar}}} \to 0$. Система TuningGenie позволяет легко проектировать эффективно работающие с кэшем алгоритмы без явного использования информации о его архитектуре. Например, рассмотренная в [15] модификация алгоритма сортировки QuickSort имеет параметр, влияющий на эффективность использования кэша и, следовательно, классифицируется как осведомленный о кэше алгоритм (Cache-aware Algorithm [16]), но в то же время значение этого параметра принадлежит предметной области задачи, за счет чего весь алгоритм абстрагирован от свойств архитектуры среды выполнения.

Оба предложенных подхода к оптимизации параллельных алгоритмов сводятся к противоположным по своему характеру стратегиям распределения вы-

числений, эффективность которых напрямую зависит от таких характеристик вычислительной среды, как объем и скорость доступа к общей памяти и кэшу процессора. Очевидно, что оптимальные стратегии для различных архитектур отличаются, и мощность предлагаемой методологии автотьюнинга заключается в том, что поиск стратегии выполняется в автоматическом режиме.

АВТОТЮНЕР КАК ДИСКРЕТНАЯ ДИНАМИЧЕСКАЯ СИСТЕМА

В данной работе вводится расширенное понятие дискретной динамической системы [14] как средства для построения формальной модели автотюнинга. Суть состоит в том, что при моделировании параллельного выполнения прикладных задач учитываются две главные характеристики алгоритма: производительность и качество [17, 18]. Первая является традиционным приоритетом при разработке параллельных алгоритмов, а вторая — в различных предметных областях представляет разные аспекты вычислений (например, в вычислительных задачах это точность вычислений, в алгоритмах сохранения данных — степень сжатия данных). Практически всегда оказывается, что обеспечение качества предполагает выполнение дополнительных вычислений, что влияет на производительность. Таким образом, одновременное достижение высоких показателей производительности и качества возможно только в результате не которого компромисса или оптимизации по заданному критерию.

Далее ограничимся рассмотрением класса вычислительных задач, где основной характеристикой качества является требование точности вычислений. Используем понятие Δ-отклонения, которое обозначает величину отклонения результата вычислений параллельного алгоритма от некоторого «эталонного» значения, полученного на PRAM*-машине с одним процессором.

Выполнение любой параллельной программы можно смоделировать с использованием теории дискретных динамических систем (ДДС) [14], определяемой тройкой (S_0, S, d), где S — множество, называемое пространством состояний; $S_0 \subseteq S$ — множество начальных состояний; $d \subset S \times S$ — бинарное отношение переходов в пространстве состояний. Система может перейти из состояния s_i в состояние s_j, если $(s_i, s_j) \in d$. Состояние памяти системы характеризуется отображением $b : X \to D$, где X — множество переменных программы, D — область значений.

Далее рассмотрим автотюнер как итеративное расширение ДДС для многопоточных программ S^{mt}, представленное в [19]. В описанной модели ДДС переменные S^{tune} не имеют типа и принимают значения в некотором универсальном множестве D из нормированного пространства [20].

Распространенными примерами норм, которые можно использовать в S^{tune}, являются следующие:

- евклидова норма — это число $|x| = \sqrt{\sum_{k=1}^{n} x_k^2}$, называемое длиной или нормой n-мерного вектора $x = (x_1, \ldots, x_n)$ в пространстве R^n. Расстояние между векторами $x = (x_1, \ldots, x_n)$ и $y = (y_1, \ldots, y_n)$ действительного пространства R^n определяется по формуле $|x - y| = \sqrt{\sum_{k=1}^{n} (x_k - y_k)^2}$;
- чебышевская (равномерная) норма [20] $||x||_{[a,b]} = \max_{t \in [a,b]} |x(t)|$, где x — непрерывная на отрезке $[a, b]$ функция (эта норма использована в [15] для оценки расхождения промежуточных результатов итераций для задачи прогнозирования метеорологических условий).

Формализуем понятие Δ-отклонения как характеристику качества результатов вычислений. Пусть выделен вектор $x \subset V$ результирующих переменных про-

граммы. Финальное состояние памяти программ P и P^* обозначим $b(x)$ и $b^*(x)$ соответственно. Расстояние между значениями результирующих переменными этих программ обозначим $r(b(x), b^*(x))$. Пусть имеется исходная программа P и преобразованная автотюнером ее версия P^*. Считаем, что результат вычислений программы P^* отклоняется на величину Δ, если программы P и P^* с одинаковыми начальными данными b_0 одновременно приходят (или не приходят) в финальные состояния соответственно $s_f = (b, \varepsilon, \phi)$ и $s_f^* = (b^*, \varepsilon, \phi)$, для которых $r(b(x), b^*(x)) \le \Delta$ (при пустой остаточной программе ε и пустом заключительном отношении переходов ϕ).

Рассматриваемая система автотюнинга — это тройка $< \text{PRAM}^*, \text{ОФ}(P), \text{OUT} >$, где P — исходный код программы; PRAM^* — рассмотренная ранее абстрактная модель вычислителя; OUT — «эталонный» результат вычисления исходного варианта параллельной программы на одном процессоре, используемый для анализа точности оптимизируемой программы; $\text{ОФ}(P) = T_N * k_\Delta * \Delta$ — функция эмпирической числовой оценки продуктивности выполнения программы, учитывающая точность полученных результатов вычислений.

Задача тюнинга заключается в минимизации значения ОФ. Коэффициент k_Δ используется для задач, которые имеют четкое требование к точности полученных результатов и времени выполнения вычислений (задачи реального времени).

Обозначим $\Delta_{\max}$ максимально допустимое Δ-отклонение. Для того чтобы исключить быстрый, но неточный вариант оптимизируемой программы, примем $k_\Delta = 1$ при $\Delta \in [0, \Delta_{\max}]$ и $k = 1000000$ (достаточно большое число) при $\Delta >> \Delta_{\max}$.

Каждая итерация тюнинга качественно оценивает сгенерированный вариант программы

$$< P, C_i > \to P^*, \text{ОФ}_i,$$

где P^* — новый вариант исходной программы, ОФ_i — оценка продуктивности выполнения программы из i-й итерации.

Преобразование, проведенное во время итерации автотюнера, будем считать результативным, если $\text{ОФ}(P^*) < \text{ОФ}(P^{\text{opt}})$, т.е., стоимостная оценка интерпретации программы строго уменьшилась. Если выполненное в пределах итерации тюнинга преобразование оказалось результативным, полученный вариант исходной программы считается улучшенным и принимается в общем случае как исходный для следующей итерации автотюнера.

В простейшем случае автотюнер осуществляет итерацию по всему множеству C. В более сложных случаях решение о продолжении и/или окончании процесса тюнинга принимает эксперт-разработчик. После окончания всех итераций тюнинга полученная программа P^{opt} считается оптимальной и сохраняется для дальнейшего выполнения.

ОПТИМИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ В МОДЕЛИ АВТОТЮНИНГА

Большинство параллельных алгоритмов, ориентированных на геометрический параллелизм [2], имеют итеративную схему организаций вычислений. На рис. 2 показана такая диаграмма вычислений, где t_{split}, t_{parallel} и t_{merge} — время, затраченное на декомпозицию данных, выполнение параллельной секции и обработку результатов вычислений соответственно. Во многих случаях для ускорения вычислений такие алгоритмы позволяют выполнять «склейку» нескольких итераций цикла в одну (рис. 3).

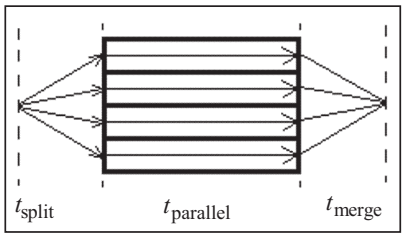

Рис. 2

Обозначим m количество объединенных параллельных секций. Будем считать, что схема вычислений, позволяющая выполнять такую склейку (см. рис. 3), имеет свойство асинхронного цикла со степенью свободы m. Для многих задач такое изменение структуры вычислений приводит к изменению результата вычислений. Далее это изменение будем характеризовать введенным раньше понятием Δ-отклонения.

Отметим, что замена схемы вычислений на асинхронный цикл увеличивает производительность параллельного алгоритма за счет сокращения времени на декомпозицию данных и синхронизацию результатов промежуточных вычислений. Покажем это формально, а так же оценим границы ускорения полученного алгоритма S^{tuned}.

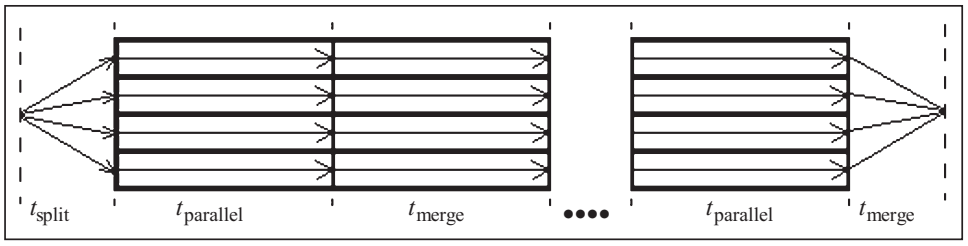

Рис. 3

Пусть n — количество итераций в исходной программе. Тогда общее время выполнения исходной параллельной программы вычисляется

$$T_p = n \times (t_{\text{split}} + t_{\text{parallel}} + t_{\text{merge}}).$$

Допустим, что m — делитель n, $m \in [1, n]$. Тогда время выполнения программы с использованием асинхронного цикла определяется

$$T_{pm} = \frac{n}{m}(t_{\text{split}} + t_{\text{merge}}) + n \times t_{\text{parallel}}.$$

Пусть $t_{\text{exchange}} = t_{\text{split}} + t_{\text{merge}}$, тогда

$$S_{\text{tuned}} = \frac{T_p}{T_{pm}} = \frac{t_{\text{parallel}} + t_{\text{exchange}}}{\frac{1}{m} t_{\text{exchange}} + t_{\text{parallel}}},$$

$$\lim_{m=n\to\infty} S_{\text{tuned}} = \frac{t_{\text{parallel}} + t_{\text{exchange}}}{t_{\text{parallel}}} = 1 + \frac{t_{\text{exchange}}}{t_{\text{parallel}}}.$$

При $m = 1$ имеем нижнюю оценку границы ускорения

$$S_{\text{tuned}} = \frac{t_{\text{parallel}} + t_{\text{exchange}}}{t_{\text{parallel}} + t_{\text{exchange}}} = 1.$$

Таким образом, полученное в результате применения асинхронного цикла ускорение лежит в следующих границах:

$$S^{\text{tuned}} \in \left[1,\ 1 + \frac{t_{\text{exchange}}}{t_{\text{parallel}}}\right].$$

Описанная оптимизация имеет смысл, когда количество итераций n достаточно велико, а время выполнения параллельной секции и время обмена данными — величины одного порядка. Легко видеть, что с увеличением параметра m значение S^{tuned} увеличивается, как и значение Δ-отклонения для большинства задач. Верхнее значение m ограничивается только ожидаемой точностью вычислений и необходимостью получения промежуточных результатов (например, для визуализации). Поэтому имеем двойственную задачу оптимизации производительности и качества полученного результата, эмпирическое решение которой возлагается на систему TuningGenie. Далее представлены примеры оптимизации двух задач с помощью асинхронного цикла.

ИНСТРУМЕНТАЛЬНАЯ СИСТЕМА ДЛЯ РЕАЛИЗАЦИИ АВТОТЮНЕРОВ TUNINGGENIE

Программная реализация рассматриваемой системы автотюнинга основана на системе обработки термов TermWare [9, 10]. В TermWare реализована концепция терминальных систем — объектных структур и правил переписывания с явными акциями взаимодействия с базой знаний. С ее помощью TuningGenie извлекает экспертные знания из исходного кода программы и генерирует новый вариант программы на каждой итерации тюнинга. Система TermWare транслирует исходный код программы в терм и предоставляет инструментарий для его преобразования с помощью переписывающих правил, что позволяет TuningGenie выполнять структурные изменения проводимых программой вычислений в декларативном стиле (без изменений исходного кода). Последнее свойство качественно отличает TuningGenie от похожих систем, например AtuneIL [8], и особенно удобно, когда нужно оптимизировать уже написанную и весьма большую параллельную программу.

Система TermWare содержит модуль для работы с JAVA и C#-кодом. Для работы с другими языками программирования необходимо написать парсер для транслирования исходного кода в язык TermWare и принтер для обратного преобразования. В настоящее время версия TuningGenie работает с программами, написанными на JAVA.

На этапе «Анализ» автотюнер, исходя из экспертных данных, строит множество конфигураций программы, которые адаптируют ее под целевую среду выполнения. Каждая конфигурация транслируется в набор переписывающих правил, которые выполняются для исходного терма параллельной программы. После преобразования полученный терм транслируется в код JAVA и компилируется, в результате чего получается готовая для эмпирической оценки новая версия оптимизируемой программы.

Ранее отмечалось, что экспертные знания разработчика сохраняются в исходном коде программы в виде специальных директив — прагм. Синтаксически последние являются комментариями, поэтому их добавление не меняет структуры вычислений параллельной программы и ее исходный код все также компилируется любым компилятором языка JAVA.

Инструментарий TuningGenie состоит из трех прагм.

Первая прагма tuneAbleParam задает область значений численной переменной. Например, для переменной numTasks границы выбора значений [1...10] с шагом 2 задаются следующим образом:

```
//tuneAbleParam name=numTasks start=1 stop=10 step=2
int numTasks = 1;
```

Эту прагму прежде всего применяют к алгоритмам, использующим геометрический параллелизм (парадигма программирования «параллелизм по данным»

[14]), поскольку она позволяет легко подобрать оптимальную декомпозицию области вычислений, т.е. «зернистость» алгоритма. С помощью этой прагмы можно также подобрать некоторое пороговое значение для изменения схемы вычислений программы. Пример использования этой прагмы для оптимизации классического алгоритма сортировки QuickSort рассмотрен далее.

Вторая прагма calculatedValue инициализирует переменную значением, которое вычисляется в целевой СВ. Этот тип прагм создан для алгоритмов, которым необходима информация о СВ, например, о скорости доступа к различным уровням памяти или скорости выполнения базовых арифметических операций. Все значения таких переменных вычисляются до выполнения итераций тюнинга и сохраняются в базе знаний TermWare. Вследствие того, что информация из базы знаний доступна переписывающим правилам, прагма calculatedValue позволяет также задать область значений для переменных предыдущей прагмы tuneAbleParam. Таким образом, можно выполнить небольшую тестовую задачу, результаты которой сузят область поиска оптимальной конфигурации и, следовательно, сократят временные затраты на автотюнинг. Похожий подход для вычислений на GPU рассмотрен в [12]. Пример использования прагмы:

```
//calculatedValue name=hdReadSpeed method=
//"org.org.tuning.EnvironmUtils.getHDReadSpeed()"
int hdKbPerSec = 1;
```

Третья прагма bidirectionalCycle идентифицирует циклы, вычисления внутри которых не зависят от порядка итераций цикла. Например, TuningGenie эмпирически опробует инкрементную и декрементную версии для следующего цикла:

```
int[] data = new int[SIZE];
//bidirectionalCycle
for (int i=0; i < SIZE; i++) {
  doSomethingWith(data[i]);
}
```

Изменение направления обхода данных влияет на эффективность использования процессорного кэша и, следовательно, на общее время выполнения параллельной программы (для задачи [15] разность между наихудшим и наилучшим вариантами составила приблизительно 15%).

СВОЙСТВО КОММУТАТИВНОСТИ ПРАГМ

Важно, что переписывающие правила, в которые транслируются прагмы TuningGenie, имеют свойство коммутативности (конфлюэнтности [21]), позволяющее применять правила преобразования кода настраиваемой программы в произвольном порядке.

Различие между прагмами calculatedValue и tuneAbleParam заключается в том, что для первой прагмы подстановочное значение вычисляется до выполнения итераций тюнинга, а не задается в параметрах, как для tuneAbleParam. Обе прагмы транслируются в переписывающие правила языка TermWare одинакового синтаксиса. Свойство коммутативности этих правил следует из требования к семантической правильности использования прагм $\forall b \exists! R_b \in R$, где R — множество переписывающих правил программы; b — некоторая переменная из множества переменных программы V; R_b — переписывающее правило, которое инициализирует значение переменной b.

Следовательно, достаточно показать коммутативность переписывающих правил, относящихся к прагмам bidirectionalCycle (ПП1) и tuneAbleParam (ПП2). Докажем это свойство в терминах ДДС, для чего введем следующие уточненные операторы присваивания значения переменной и условия продолжения ветвления цикла.

Обозначим l, z, y — литералы; $A(b, l)$ — оператор присваивания переменной b значения l; $B(b, z), B'(b, z)$ — условные операторы; $C(b), C'(b)$ — операторы изменения значения переменной, например инкремент.

Переписывающее правило, в которое транслируется прагма bidirectionalCycle, меняет управление программы следующим образом:

$$A(b,l);\ while(B(b,z),C(b),P) \to A(b,z);\ while(B'(b,l),C(b),P). \qquad (1)$$

Манипуляции с литералами ПП2 выполняет, не изменяя операторы

$$A(b,?) \to A(b,y), \qquad (2)$$

где символ ? обозначает универсальный литерал.

Очевидно, что преобразования (1) и (2) коммутативны и, следовательно, коммутативны прагмы bidirectionalCycle и tuneAbleParam, порождающие ПП1 и ПП2. Из представления ПП1 и ПП2 также видно, что они трансформируют (но не порождают) различные операторы, и, следовательно, вся фаза трансформации исходного кода программы имеет свойство конечности для конечных входящих данных.

ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ

Пример 1. Рассмотрим классический алгоритм сортировки QuickSort, в котором используется фреймворк TuningGenie для оптимизации программ. Данный алгоритм можно значительно улучшить, применяя для подзадач (массивов) небольшого размера более эффективный алгоритм «внутренней сортировки» (без использования дополнительной памяти и, следовательно, с более эффективным использованием ее кеширования). Прагма tuneAbleParam позволяет легко для любой вычислительной среды подобрать это оптимальное пороговое значение. Поскольку замена алгоритма сортировки не меняет конечного результата, единственным критерием оценки качества выполненной оптимизации будет затраченное на сортировку время.

Исходный код имеет вид

```
……
//tuneAbleParam name=threshold start=10 stop=1500 step=10
int threshold = 1;
if (high – low < threshold) {
        insertionsort (array, low, high);
} else {
        addPartitionForQuickSort (array, low, high)
}
……
```

Эксперимент проводился на персональном компьютере со следующей конфигурацией:

- Intel® Core™ i5-2410M Processor (3M Cache, up to 2.90 GHz);
- 4 GB DDR2 RAM.

График зависимости времени сортировки массива с двумя миллионами элементов от значения переменной threshold приведен на рис. 4.

Система TuningGenie подобрала оптимальное значение threshold = 90, при котором рассматриваемая модификация QuickSort оказалась приблизительно на 30% быстрее классической версии.

Пример 2. Рассмотрим задачу моделирования броуновского движения в идеальном газе. В этой модели не будем различать частицы, что позволит за счет абсолютно упругого столкновения частиц между собой игнорировать этот процесс, поскольку вследствие одинаковой скорости и массы в момент столкновения частицы как бы меняются местами, продолжая траекторию движения до столкновения. Иными словами, учитываем только столкновения молекул со стенками сосуда и можем моделировать движение каждой частицы независимо от других. Пусть необходимо смоделировать положение частиц через некоторый интервал времени Δt. Разобьем этот интервал на y равных интервалов и будем считать, что каждая частица за один интервал делает один шаг — перемещение на единицу расстояния. Таким образом, задача сводится к моделированию положения

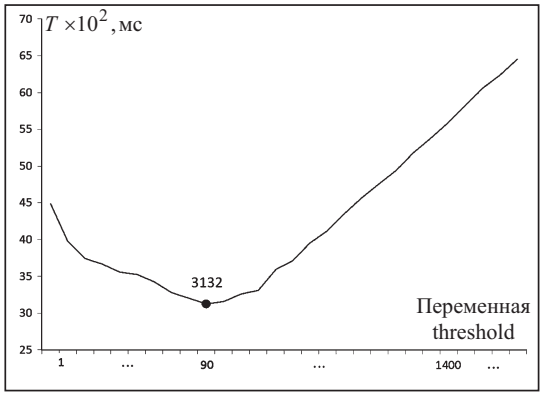

Рис. 4

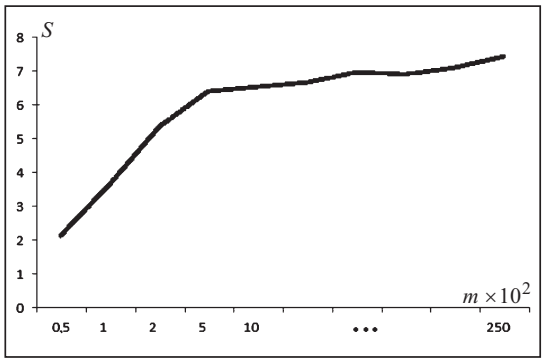

Рис. 5

частиц идеального газа внутри сосуда через y шагов.

Параллельная схема вычислений этой задачи имеет свойство асинхронного цикла со степенью свободы z, значение которой определяется необходимостью получения промежуточных результатов вычислений. Если они не нужны (например, нет необходимости визуализировать процесс), то $m = n$.

Как и в примере 1, единственным критерием оценки качества выполненной оптимизации будет затраченное на вычисление время. Влияние значения параметра m на общую производительность очевидно: чем больше количество независимых итераций, тем меньше затрат на синхронизацию и общее время вычислений. Этот факт подтверждает проведенный эксперимент.

Конфигурация тестовой площадки такова:

- 2 x Quad Core Intel Xeon E5405 2 GHz Intel EM64T;
- 16 GB DDR3 1066 Mhz RAM;
- $n = 300000$.

График зависимости мультипроцессорного ускорения S от значения параметра z приведен на рис. 5.

Как видно из графика, при большом значении z мультипроцессорное ускорение достигало значения $S = 7.5$, что согласно закону Амдала близко к теоретическому пределу (вычислительная система состояла из восьми ядер).

Приведенные примеры дают общее представление о классах задач, которые можно оптимизировать с помощью TuningGenie. Применение методологии автотюнинга к сложной вычислительной задаче со свойством асинхронного цикла, а также с жесткими ограничениями по времени и точности вычислений детально рассмотрено в [15].

ЗАКЛЮЧЕНИЕ

Известное утверждение «алгоритмы + структуры данных = программы» [22] верно и для параллельных программ. Однако задача построения эффективного параллельного алгоритма и выбора подходящей структуры данных для него оказывается намного сложнее. Для достижения высоких показателей производительности программы она должна максимально эффективно использовать ресурсы вычислительно среды, а сложность и разнообразие современных параллельных архитектур делает практически невозможным создание программы, эффективной в любой вычислительной среде, без длительного процесса ее конфигурирования и настройки. Методология автотюнинга позволяет существенно сократить ресурсоемкий и рутинный этап оптимизации параллельных программ за счет автоматизации процесса их эмпирической адаптации к целевой вычислительной среде.

Рассматриваемая в работе модель и система генерации автотюнеров TuningGenie позволяет оптимизировать обе составляющие параллельной программы: алгоритмы и структуры данных. Основываясь на системе переписывающих правил, TuningGenie автоматически генерирует автотюнер для оптимизируемой программы и требует для этого минимальных изменений в исходном коде. Использование экспертных знаний, в свою очередь, максимально сокращает процесс оптимизации.

СПИСОК ЛИТЕРАТУРЫ

1. Sedgewick R., Wayne K. Algorithms (4th ed.). — Boston: Addison-Wesley Profess., 2011. — 976 p.
2. Дорошенко А.Ю. Курс лекцій «Паралельні обчислювальні системи». — К.: Вид. дім КМА, 2003. — 42 с.
3. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html
4. http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers
5. Naono K., Teranishi K., Cavazos J., Suda R. Software automatic tuning from concepts to state-of-the-art results. — New York: Springer, 2010. — 240 p.
6. Whaley R., Petitet A., Dongarra J.J. Automated em-pirical optimizations of software and the ATLAS project // Parallel Comput. — 2001. — **27**(1-2). — P. 3–35.
7. Frigo M. and Johnson S. FFTW: An adaptive software architecture for the FF // Acoustics, Speech and Signal Processing. — 1998. — **3**. — P. 1381–1384.
8. Schaefer C.A., Pankratius V., and Tichy W.F. Atune-IL: An instrumentation language for auto-tuning parallel applications // Euro-Par'09 Proc. 15th Int. Euro-Par Conf. on Parallel Processing. — Berlin; Heidelberg: Springer-Verlag, 2009. — P. 9–20.
9. http://www.gradsoft.ua/products/termware sub rus.html
10. Дорошенко А.Е., Шевченко Р.С. Система символьных вычислений для программирования динамических приложений // Проблемы программирования. — 2005. — № 4. — С. 718–727.
11. Іваненко П.А. Застосування різних алгоритмів пошуку оптимальної конфігурації для паралельного алгоритму чисельного розв'язання багатовимірної задачі моделювання навколишнього середовища // Theoretical and Applied Aspects of Cybernetics: Proc. of the Intern. Sci. Conf. of Students and Young Scientists. — Kyiv: Bukrek, 2011. — 352 p.
12. Krishnamoorthy Ma W., Agrawal S.G. Parameterized micro-benchmarking: An auto-tuning approach for complex applications // Proc. of the 9th Conf. on Comput. Frontiers. — 2012. — P. 213–222.
13. Eppstein D., Galil Z. Parallel algorithmic techniques for combinatorial computation // Ann. Rev. Comput. — 1988. — N 3. — P. 233–283.
14. Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А. Алгебро-алгоритмические модели и методы параллельного программирования. — Киев: Академпериодика, 2007. — 631 с.
15. Іваненко П.А., Дорошенко А.Ю. Автоматична оптимізація виконання для задачі метеорологічного прогнозування // Пробл. програмування. — 2012. — № 2–3. — С. 426–434.
16. Prokop H. Cache-oblivious algorithms // FOCS'99 Proc. of the 40th Ann. Symp. on Foundations of Comput. Sci. — 1999. — N 40. — P. 285–299.
17. Akl S. Superlinear performance in real-time parallel computation // The Journal of Supercomp. — 2004. — N 29. — P. 89–111
18. Трауб Дж., Васильковский Г., Вожьняковский Х. Информация, неопределенность, сложность. — М.: Мир, 1988. — 184 с.
19. Дорошенко А.Е., Жереб К.А. Алгебро-динамические модели для распараллеливания программ // Пробл. програмування. — 2010. — № 1. — С. 39–55.
20. Колмогоров А.Н., Фомин С.И. Элементы теории функций и функционального анализа. — М.: Наука, 1976. — 544 с.
21. Bezem M., Klop J.W., Vrijer R. Term rewriting systems. — New York: Cambridge Univ. Press, 2003. — 908 p.
22. Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.

Поступила 04.07.2013