УДК 004.4`2

ПРО АНАЛІЗ КОРЕКТНОСТІ АВТОТЬЮНІНГУ ПРОГРАМ З ВИКОРИСТАННЯМ ТЕХНІКИ ПЕРЕПИСУВАЛЬНИХ ПРАВИЛ

П.А. Іваненко [0000-0001-5437-9763]

Інститут програмних систем НАН України, 03187, м. Київ-187, проспект Академіка Глушкова, 40.

У роботі запропоновано підхід до перевірки коректності оптимізаційних перетворень паралельних програм, що виконуються автотьюнером. Автотьюнер розглядається як еволюційна дискретна динамічна система й перевірка коректності зводиться до перевірки властивості еквівалентності за результатом представлень вихідної і оптимізованої версій програм у формальній моделі автотьюнінгу. Цю перевірку у часткових випадках можна виконати автоматично за вихідним кодом за допомогою техніки переписувальних правил.

Ключові слова: автоматизація оптимізації паралельних програм, автотьюнінг , алгебро-динамічні моделі, техніка переписувальних правил, перевірка коректності перетворень.

Software automated tuning is a well-known methodology of performance optimization. Its typically applicable to parallel programs that are frequently executed on the same hardware. The main idea of autotuning is to perform the empirical evaluation of different variations of a program in a target environment and choose the best one. Usually evaluation is performed by an external application that is called ‘tuner’. Depending on how complex this tuner is, it can be supplied by a collection of program’s variants to choose from or a set of configurations and program’s source code to generate these variants.

In previous works author introduced an auto-tuning model, based on Glushkov’s systems of algorithmic algebras and term rewriting technique, and developed auto-tuning framework that works with program’s source code. This framework uses rule-based rewriting system TermWare for source-to-source code transformations. Such approach is quite flexible as it allows to easily change code’s structure.

This work focuses on analyzing correctness of applied transformations. The correctness of transformations means that the original and converted programs return the same result. Such programs are meant to be equivalent by result. According to defined lemma, check of this characteristic is reduced to validating that initial and modified program versions are deadlock-free, data race-free and equivalent by operators (with respect of commutative operators). In general, these properties are very difficult to verify. However, for some methods, partial cases can be formulated for which these properties can be verified with the help of a rewriting rules technique. This paper demonstrates how to check correctness of changing direction of cycle iterations – technique that aims to improve cache locality during multi-dimensional arrays traversals. Results of its application to optimization of a parallel algorithm for short-term meteorological forecasting are also presented.

Key words: automation of parallel program optimization, autotuning, algebra-dynamic models, rewriting rules technique, validation of code transformations correctness.

В работе представлен подход к проверке корректности оптимизационных преобразований параллельных программ, выполняемых автотьюнером. Автотьюнер рассматривается как эволюционная дискретная динамическая система, а проверка корректности сводится к проверке свойству эквивалентности по результату интерпретаций исходной и оптимизированной программ в формальной модели автотьюнинга. Эту проверку в частных случаях можно выполнять автоматически по исходному коду программы с помощью техники переписывающих правил.

Ключевые слова: автоматизация оптимизации параллельных программ, автотьюнинг, алгебро-динамические модели, техника переписывающих правил, проверка корректности преобразований.

Вступ

Сучасний стан розвитку науки свідчить про те, що для розв'язання складних науково-технічних задач необхідні значні обчислювальні потужності, а їх раціональне використання завжди було однією з головних проблем під час розробки прикладних програм. Сучасні процесори мають багато ядер й багато рівнів кеш-пам'яті, що додатково ускладнює архітектуру мультипроцесорних обчислювальних систем. Оптимальна програма має ефективно розподіляти обчислення між усіма ядрами, а також максимально ефективно використовувати усі рівні пам'яті: від процесорного кешу до жорсткого накопичувача. Таку програму важко створити, навіть знаючи наперед характеристики цільового обчислювального середовища (ОС), а етап налаштування потребує значних зусиль. Написання програми, яка буде одразу оптимальною у різних ОС виявляється неможливою задачею через занадто велике розмаїття сучасних процесорів.

Отже, для досягнення максимальної ефективності програми (тобто, мінімального часу її виконання) необхідне її додаткове налаштування (тюнінг) під ОС, в якому вона буде виконуватися. Сучасна методологія самоналаштування (автотюнінгу) дозволяє автоматизувати це налаштування. Ідея автотюнінгу полягає в емпіричному оцінюванні декількох варіантів програми й вибору найкращого. Традиційно підбір виконує окрема програма-тюнер. Вона ж відповідає за генерацію різних модифікацій вихідної програми. Основними критеріями оцінки, зазвичай, є швидкодія й точність отриманих результатів.

У цілому дії програми-тюнера є досить шаблонними – обрати/створити нову версію програми й отримати емпіричну оцінку її швидкодії. Програмне рішення, запропоноване у роботі, дозволяє тюнеру автоматично генерувати нові коректні версії програм, спираючись на експертне знання розробника, оформлене у вигляді метаданих у вихідному коді.
В роботі, на прикладі методу вибору оптимальної стратегії обходу індексованих структур даних для покращення швидкодії оптимізовуваної програми, проілюстровано як можна автоматизувати перевірку коректності перетворень вихідного коду програми. Ця перевірка спирається на властивості, сформульовані у термінах дискретних динамічних систем для паралельних програм.

Властивості програм

Виконання будь-якої програми можна змоделювати з використанням теорії дискретних динамічних систем (ДДС) [1]. ДДС задається як трійка (S_0, S, d), де S – простір станів; $S_0 \subseteq S$ – множина початкових станів; $d \subset S \times S$ – бінарне відношення переходів у просторі станів. Система може перейти із стану s_i в стан s_j якщо $(s_i, s_j) \subset d$. Повне визначення ДДС з переліком базових операторів й виразів можна знайти у [1]. Формальна модель автотюнінгу S^{tuning}, що представлена як еволюційне розширення моделі дискретних динамічних систем будується у роботі [2]. Вона базується на ДДС для мультипоточних програм S^{mpr}, яка в свою чергу побудована на моделі гпослідовних програм S^{spr}. Програми у цих моделях представлені у вигляді виразів алгебри Глушкова (АГ) [7], які інтерпретуються як функції ДДС.
Визначимо деякі властивості програм, які допоможуть проаналізувати коректність й ефективність оптимізаційних перетворень, що виконує тьюнер. Під *коректністю* перетворень будемо розуміти те що вихідна й перетворена програми повертають однаковий результат. Формально цю умову можна сформулювати наступним чином. Нехай задана підмножина $V_0 \subset V$ результуючих змінних. Тоді дві програми P^1 й P^2 будемо називати еквівалентними за результатом, якщо для однакових початкових даних b_0 програми одночасно приходять або не приходять в фінальні стани $s_f^1 \quad (b^1, \varepsilon, \emptyset)$ й $s_f^2 \quad (b^2, \varepsilon, \emptyset)$, причому ці фінальні стани пам'яті співпадають за результуючими змінними $b_{V_0}^1 \quad b_{V_0}^2$.
Аналогічним чином, використовуючи поняття Δ- відхилення з попереднього підрозділу, введемо поняття Δ(дельта)-коректності для класу задач, які дозволяють поступитися точністю результатів обчислень заради пришвидшення швидкодії програми. Дві програми P^1 й P^2 будемо називати еквівалентними за результатом з Δ-точністю, якщо для однакових початкових даних b_0 програми одночасно приходять або не приходять в фінальні стани $s_f^1 \quad (b^1, \varepsilon, \emptyset)$ й $s_f^2 \quad (b^2, \varepsilon, \emptyset)$, причому ці фінальні стани пам'яті відхиляються не більше ніж на величину Δ: $r\left(b_{V_0}^1, b_{V_0}^2\right) \le \Delta$.
Еквівалентність за результатом – достатньо загальна властивість, проте реалізувати перевірку цієї властивості в загальному випадку практично неможливо оскільки для цього необхідний аналіз усіх можливих шляхів виконання програми в залежності від вхідних даних й варіантів виконання різних потоків. Тому необхідно визначення більш часткових властивостей для опису коректності перетворень, які можуть бути практично перевірені.
Спочатку визначимо деякі властивості операторів АГ. Для кожного оператора $y \subset Y$ визначені множини $R(y) \subset V$ змінних, від яких залежить результат застосування оператора, й $W(y) \subset V$ змінних, які змінюють значення в результаті виконання оператора. Два оператора $y_1, y_2 \subset Y$ будемо називати залежними (за даними), якщо $R(y_1) \cap W(y_2) \neq \emptyset$ або $W(y_1) \cap R(y_2) \neq \emptyset$, або $W(y_1) \cap W(y_2) \neq \emptyset$. Два оператори $y_1, y_2 \subset Y$ є перестановочними або, комутативними (відносно операції композиції) якщо $y_1 y_2 \quad y_2 y_1$. Очевидним є те що незалежні оператори є перестановочними, проте обернене твердження не є істинним – наприклад, дві копії одного оператора $y \subset Y$ завжди перестановочні, але вони залежні якщо $W(y) \neq \emptyset$.
Тепер визначимо наступні властивості програм*: беструпиковість, безконфліктність* й *еквівалентність за операторами*. Програму P будемо називати *беструпиковою*, якщо при її виконанні не з'являються тупикові стани (стан коли неможливе подальше виконання залишку програми у ДДС). Очевидно що усі послідовні програми в моделі S^{spr} є безтупіковими.
В моделі S^{mpr} будемо називати конфліктним переходом ситуацію коли існують залежні оператори, які виконуються одночасно у різних потоках. Програму P будемо називати безконфліктною якщо при її виконанні не з'являються конфліктні переходи. Для безконфліктних програм можна не використовувати функцію merge (2.2). Замість об'єднання результатів одночасного виконання декількох операторів можна виконати ці оператори послідовно у будь-якій послідовності.
Якщо відома послідовність виконання програми (послідовність переходів ДДС), можна визначити історію використання базових операторів [9]. Для цього додамо до стану ДДС ще одну компоненту $h \subset Y^*$. В початковому стані $h \quad \varepsilon$. Також модифікуємо правило переходу (1): $(b, \gamma R, F, h) \to (\gamma(b), R, F, h\gamma)$. При одночасному виконанні декількох правил (1) у різних потоках будемо додавати відповідні оператори y у довільному порядку. Оператор h у фінальному стані будемо називати історією використання операторів.
Дві історії h_1, h_2 будемо вважати еквівалентними (відносно визначеної системи рівності операторів АГ) якщо h_2 можна отримати з h_1 послідовним застосуванням операції рівності до операторів історії. Зокрема, система рівності завжди включає усі відношення які визначають комутативність операторів. Тобто h_2 також можна

отримати із h_1 послідовними перестановками комутативних пар операторів. Дві програми P^1 й P^2 будемо називати еквівалентними за операторами якщо для однакових вхідних даних b_0 вони породжують еквівалентні історії виконання операторів.
Тепер можна сформулювати наступне твердження:
Лема 1. Якщо дві програми P^1 й P^2 безтупікові, безконфліктні й еквівалентні за операторами, то вони еквівалентні за результатом.
Доведення. Безтупикові програми не переходять у тупикові стани, отже вони або переходять в фінальний стан або не зупиняються взагалі. В останньому випадку історія операторів буде нескінченно, а тому, якщо така ситуація має місце дляP^1, то вона буде виконуватися й для P^2. Отже перша частина визначення еквівалентності за результатом доведена. Перевіримо тепер, що на однакових вихідних даних програми обчислюють однаковий результат. Обчислення програми можна представити у вигляді виконання деякого комбінованого оператора y_f над вихідним станом пам'яті b_0: $b_f = y_f b_0$. Виходячи з побудови ДДС S^{sar} й S^{sxt}, комбінований оператор є послідовною композицією операторів, які виконуються на кожному переході ДДС. Для системи S^{sxt} кожен перехід поєднує множину переходів окремих потоків; проте для безконфліктних програм перехід системи, вцілому, еквівалентний послідовній композиції переходів окремих потоків, які розглядаються у довільному порядку.
Таким чином для безконфліктних програм комбінований оператор співпадає з історією виконання операторів, тобто $y_f = h$. З визначення еквівалентності за операторами виплаває, що $\forall b \subset B, (h^2 b)_{y_{i1}} = (h^2 b)_{y_{i1}}$, тобто історії операторів діють однаково на результуючі данні. Тому $b^1_{y_{i1}} = (h^1 b_0)_{y_{i1}} = (h^2 b_0)_{y_{i1}} = b^2_{y_{i1}}$, що й треба було довести.
Отже чином перевірка еквівалентності за результатом зводиться до перевірки цих трьох властивостей, що є цілком можливим для багатьох задач.

Перехід між різними представлення програми

Якщо обирати підхід до трансформації вихідного коду програм виходячи суто з "практичних" міркувань то в деяких випадках достатньо було б представлення програм у вигляді текстового файлу. Наприклад, для заміни значення константи яка задає початкове значення деякої змінної програми. Проте, такі структурні зміни як обернення ітерацій циклу потребують більш високорівневого представлення, зокрема для того щоб трансформація не залежала від форматування вихідного коду.
Також можна використовувати представлення програми у вигляді дерева синтаксичного розбору. Для побудови такого представлення використовують синтаксичний аналізатор відповідної мови програмування, який іноді називають парсером. Дерево синтаксичного розбору все ще можна віднести до низькорівневого представлення оскільки кожна синтаксична конструкція мови програмування явно представлена в моделі.
Як вже зазначалося, модель програми у вигляді операторів АГ – більш високорівневе представлення, яке є більш компактним. Також вона краще підходить для теоретичного аналізу. Проте вона потребує додаткових спеціалізованих засобів для побудови моделі й перетворення її в код програми. Також варто зауважити, що, на відміну від низькорівневих моделей, побудована модель мультипоточних програм не залежить від мови програмування, хоч і є специфічною для деякої предметної області.
Також для виконання перетворень можна використовувати фреймворк Termware [3], [4], який спирається на техніку переписувальних правил. Цей фреймворк дозволяє для деякої предметної області визначити відповідність між операторами й предикатами АГ й програмним кодом деякої(деяких) мови програмування [8]. Така відповідність дає змогу використовувати переписувальні правила Termware для автоматичного переходу між двома рівнями представлення програми.
Наведемо поняття паттернів для Termware, за допомогою яких будемо визначати відповідність. Паттерн буде задаватися парою $< t_p, t_g >$, де t_p – позначення паттерна який задає новий елемент моделі високого рівня а t_g - зразок що визначає паттерн (елемент моделі низького рівня).
Простим прикладом може бути паттерн *Join* для операції очікування завершення роботи іншого потоку. Для нього t_p = *_Join($Sx)* а комбінований оператор t_g = *wait(_TerminationEvent($Sx))*. Цей паттерн містить змінну $Sx й використовує додатковий паттерн *_TerminationEvent($Sx)* для визначення частини терму.
Якщо для деякої предметної області визначені предикати й оператори високорівневої моделі – це робить можливим створення програми на будь-якому з рівнів: низькорівневий вихідний код та дерево синтаксичного розбору або високорівнева модель АГ. В роботі розглядається застосування методів автотьюнінгу до вже написаних програм, тобто парсер Termware трансформує вихідний код у синтаксичне дерево. Далі застосовуються патерни й переписувальні правила для переходу до алгебро-алгоритмічного представлення, в якому можна перевірити коректність. Далі з використанням іншої системи переписувальних правил можна виконати зворотній перехід до низькорівневої моделі, по якій генератор Termware може відтворити початкову версію вихідного коду, так і породити нову з додатковими модифікаціями.

Перевірка властивостей

Властивості безтупіковості, безконфліктності й еквівалентності за операторами, які було визначено у попередньому підрозділі, в загальному випадку важко перевірити. Проте, для окремих часткових випадків, можна вказати достатні умови для забезпечення цих властивостей, які легше перевірити. Розглянемо деякі такі умови.

Умова еквівалентності за операторами практично значить що деякі комутативні оператори програми були переставлені у історії виконання. Вимогу комутативності для перестановки операторів в межах послідовної композиції будемо позначати як CommutativeOperators. У випадках коли окремі ітерації циклу виконуються в іншій послідовності для еквівалентності за операторами необхідно щоб комутативними були окремі ітерації (будемо позначати цю умову як CommutativeIterations) а також щоб була взаємно однозначна відповідність між ітераціями у обох програмах (вихідна й перетворена). Останню умову позначимо IdenticalIterations.

Перевірка умов комутативності – достатньо складна задача у загальному випадку, оскільки потребує аналізу властивостей усіх операторів, викликів функцій й можливих варіантів виконання програми. У багатьох випадках комутативність можна встановити на основі незалежності операторів циклу (для умови CommutativeIterations), що значно легше зробити з використанням техніки переписувальних правил. У інших випадках будемо виконувати перевірку результатів програми.

Наприклад, для перевірки властивості CommutativeIterations деякого циклу програми можна згенерувати програму з оберненим порядком ітерацій. Якщо у вихідній програмі ітерації були перестановочними то модифікована програма повинна обчислити той самий результат. Очевидно що співпадіння результатів тесту ще не гарантує комутативності ітерацій оскільки можлива ситуація коли на деякому іншому наборі даних результати обчислень будуть відрізнятися. Для зменшення вірогідності такої ситуації можна використовувати тести з більш складними трансформаціями. Наприклад, розгорнути дві ітерації циклу у одну й перемішати оператори цих ітерацій.

Важливою перевагою використання фреймворку TermWare й техніки переписувальних правил є можливість автоматизувати створення великого набору різних тестів для кожної конкретної програми.

Умову безконфліктності можна сформулювати наступним чином: будь-які два залежні оператора, що можуть виконуватися у різних потоках, повинні бути захищеними відповідними критичними секціями (секціями з однаковими ідентифікаторами). Далі будемо позначати цю умову як AllNeededCS. Безпосередня перевірка цієї умови вимагає виконання повного аналізу залежностей програми і є складною задачею [11]. Проте, в деяких часткових випадках перевірка значно спрощується. Наприклад, незалежними будуть оператори які змінюють лише локальні змінні для потоку.

Перейдемо до розгляду властивості безтупиковості й почнемо з випадка коли в програмі відсутні оператори *wait*. Цю умову легко перевірити за допомогою простої системи правил TermWare:

```
wait($x) ->NIL [error()];
```

Без операторів *wait* потенційні тупикові ситуації можуть з'явитися лише через використання операторів *lock*. В цьому випадку залежність між критичними секціями можна зобразити у вигляді орієнтованого графа: вершинами будуть усі критичні секції програми cs_i а ребро з вершини cs_i в cs_j додається у випадку коли деякий потік T_k може виконати оператор $lock(cs_j)$ з критичної секції cs_i. У такій інтерпретації безтупиковість програми еквівалентна властивості ациклічності (позначимо як NoCyclicCriticalSections) побудованого графа.

Частковим випадком є ситуація коли програма не містить критичних вкладених секцій (умова NoEmbeddedCriticalSections) – граф критичних секцій буде ациклічним оскільки в ньому взагалі не буде ребр. Цю умову досить легко автоматично перевірити з використанням переписувальних правил TermWare:

```
1. Method($head,$body) -> Method ($head,[m0:$body],_Mark);
2. [m0: [lock($cs):$x]] -> [lock($cs): [m1($cs):$x]];
3. [m1($cs): [unlock($cs):$x]] -> [unlock($cs): [m0:$x]];
4. [m1($cs): [lock($cs1):$x]] ->NIL [error()];
5. [m0: [$x:$y]] -> [$x: [m0:$y]];
6. [m1($cs): [$x:$y]] -> [$x: [m1($cs):$y]].
```

Тут правило 1 додає спеціальний маркувальний терм $m0$ в початок кожного методу. Цей терм використовується під час обходу усіх операторів методу. Правило 2 замінює маркер $m0$ на $m1$ якщо при обході зустрічається критична секція. Правило 3 виконує зворотне перетворення якщо маркер пересувається до точки завершення критичної секції й нові секції не зустрічаються (виконується властивість NoEmbeddedCriticalSections). Якщо ж умова NoEmbeddedCriticalSections порушується – система правил сигналізує про помилку за рахунок правила 4. Правила 5 й 6 забезпечують пересування маркерів $m0$ й $m1$ між операторами методу.

Звісно, порушення умови NoEmbeddedCriticalSections ще не вказує на те що у програмі є помилка яка призводить до тупика. В таких випадках необхідно перевіряти більш загальну вимогу NoCyclicCriticalSections.

Варто зауважити що багато прикладних паралельних програм задовольняють властивості NoEmbeddedCriticalSections оскільки такий стиль використання синхронізації суттєво простіше для розуміння.
Перевірка безтупиковості суттєво ускладнюється якщо в програмі використовуються оператори *wait* – у цьому випадку для кожного оператора $wait(ev_2)$ має бути присутнім відповідний оператор $signal(ev_1)$. Це дуже важко перевірити для усіх можливих варіантів виконання програми.
Тому замість загальних умов безтупиковості при використанні операторів wait будемо формулювати й розглядати деякі шаблони їх використання, для яких гарантується коректність.
Наприклад, оператор бар'єрного очікування $Barrier(ts_1)$, який задає точку в програмі у якій кожен потік очікує усі інші або деяку частину, може бути заданий наступним чином:

$$Barrier(k, b_1) \quad lock(cs_{b_1}); inc(count\_b_1); if\,(count\_b_1 < k, unlock(cs\_b_1); wait(ev\_b_1), assign(count\_b_1, 0);$$
$$signal\_all(ev\_b_1); unlock(cs\_b_1))$$

Як видно з визначення, цей складений оператор використовує додаткову змінну $count\_b_1$, критичну секцію cs_b_1 й точку синхронізації ev_b_1. У додатковій змінній зберігається кількість потоків які вже очікують за бар'єром. Якщо їх менше за параметр k – поточний потік також буде очікувати. Як тільки потоків збирається k штук – усі вони вивільнюються й у змінну $count\_b_1$ записується 0.
Оператор бар'єру не буде спричиняти тупики якщо загальна кількість потоків де він використовується рівна k й у кожному потоці виконується однакова кількість операторів $Barrier$. У багатьох сучасних мовах з підтримкою багатопоточності є безпосередня імплементація оператора бар'єру тому немає необхідності реалізовувати його через базові оператори *wait* та *signal*. Проте сформульована умова безтупіковості програми з бар'єрами в цьому випадку також залишається істинною.

Метод вибору оптимальної стратегії обходу індексованих структур даних

Розглянемо метод оптимізації використання процесорного кешу програмами з ітеративною схемою обчислень. Як відомо, сучасні процесори мають декілька рівнів кешів з різною швидкістю доступу. Розміри цих кешів зазвичай відрізняються на порядок тому швидкодія програми сильно залежить від частоти переміщення даних між цими рівнями. Запропонований метод автоматично згенерує і оцінить швидкодію інверсного напрямку ітерування даними й, у першу чергу, орієнтований на задачі що працюють з великими багатомірними даними. Для одномірних масивів послідовне зчитування елементів масиву у більшості випадках автоматично буде найшвидшим за рахунок апаратної оптимізації – доступ до "сусідніх" елементів значно швидший якщо вони вміщуються у одну кеш-лінію. Проте, якщо програма працює з багатовимірними масивами або з їх представленням у одновимірному масиві – частіше за все сусідні з точки зору ітерацій циклу елементи не будуть потрапляти до однієї кеш-лінії. Тому варто оцінити усі комбінації ітерування – їх кількість рівна 2^d, де d – кількість вкладених циклів.

Позначимо вихідний варіант програми через *P1* а її трансформовану варіацію зі зміненим напрямком ітерування довільного циклу через *ReversedCycleP1*. Перевірка властивості комутативності ітерацій *CommutativeIterations* в загальному вигляді є складною для перевірки тому розглянемо декілька часткових випадків, для яких така перевірка можлива.

Введемо нову властивість яка характеризує обчислення що виконуються в межах ітерацій одного циклу. Будемо казати що ітерації циклу задовольняють вимоги *AllLocalVariables* якщо внутрішні оператори використовують лише внутрішні змінні. Ця властивість перевіряється системою переписувальних правил Termware яка дозволяє використовувати лише одну зовнішню змінну – лічильник циклу.

Теорема 1. Ітерації циклу є комутативними (властивість *CommutativeIterations)* якщо вони задовольняють властивості *AllLocalVariables*.

Доведення досить очевидне оскільки властивість *AllLocalVariables* виключає будь-яку залежність за даними між різними ітераціями циклу.

Зауважимо що властивість *AllLocalVariables* не має великої практичної цінності – оператори циклу не використовують індексованих структур (одно/багатовимірні масиви) й зміна напрямку ітерування не впливає на ефективність використання кешів, тому швидкодія програми не буде залежати від виконуваної трансформації.

Тому розглянемо більш універсальний випадок коли оператори циклу або читають або записують значення у зовнішні змінні але ніколи не роблять це одночасно. Така властивість (далі будемо позначати як *ReadOrWriteAccessToVariables*) виключає рекурентну схему обчислень. Ця властивість, як і *AllLocalVariables*, виключає залежність за даними між ітераціями циклу, з чого випливає виконання умови їх комутативності *CommutativeIterations*. Типовим прикладом такої схеми обчислень є простий послідовний алгоритм множення матриць – кожна ітерація циклу зчитує відповідні значення з вихідних матриць й записує результат у іншу змінну.

Сформулюємо ще одну властивість – властивість "локальності" операторів циклу у тому сенсі що вони виконуються в одному потоці й не залежать від даних з інших потоків. Будемо позначати таку властивість циклів через *OnlySequentialCalculations.* Цю властивість легко перевірити системою переписувальних правил

яка забороняє використання операторів для багатопоточної взаємодії *call_thread,wait,signal,signal_all,lock* та *unlock*.

Теорема 1. Програми *P1* й *ReversedCycleP1* еквівалентні за результатом якщо ітерації трансформованих циклів задовольняють властивостям *OnlySequentialCalculations* й *ReadOrWriteAccessToVariables*.

Доведення. Для доведення теореми, згідно з лемою 1 необхідно перевірити властивості безтупиковості, безконфліктності й еквівалентності за операторами програм *P1* й *ReversedCycleP1*.

Безтупиковість обох програм випливає з властивості *OnlySequentialCalculations* й характеру трансформації – змінюється лише напрям ітерування циклу(ів).

Відсутність взаємодії між потоками гарантує відсутність конфліктних переходів у *P1* й *ReversedCycleP1* а тому й безконфліктність програм.

Для доведення еквівалентності за операторами необхідно щоб виконувалися умови *CommutativeIterations* й *IdenticalIterations*. Умова *IdenticalIterations* гарантується самою трансформацією – змінюється лише порядок ітерацій циклу а не їх кількість. Властивість же *CommutativeIterations* (комутативність ітерацій циклу) випливає з умови *ReadOrWriteAccessToVariables.*

Програмна реалізація

Як було зазначено раніше, на додачу до перевірки властивостей переписувальні правила також дозволяють автоматизувати різноманітні шаблони трансформацій вихідного коду програм. Застосування методу вибору оптимальної стратегії обходу індексованих структур даних на практиці можна звести до розмічення циклів у вихідному коді програми прагмами-коментарями. За цими мітками автотьюнер згенерує систему переписувальних правил, яка ‘розверне’ ітерації циклу у зворотному напрямку. Наприклад, у мові Java для циклів типу:

```
//bidirectionalCycle
for (int loopVar = 0; loopVar <42; loopVar ++){
……..……
}
```

Відповідне переписувальне правило виглядає наступним чином:

```
ForStatement(
LoopHead(
ForInit(VariableDeclarator(Id("loopVar"),Literal($initialValue))),
RelationalExpression(Identifier("loopVar"),"<",Literal($endValue)),
ForUpdate(StatementExpression(Identifier( "loopVar "),"++"))),
Block(.....)
)       ->
ForStatement(
LoopHead(
ForInit(VariableDeclarator(Id("loopVar"),Literal($endValue))),
RelationalExpression(Identifier("loopVar"),">",Literal($initialValue)),
ForUpdate(StatementExpression(Identifier( "loopVar "),"--"))),
Block(.....)
)
```

Це правило, використовуючи пропозиційні змінні [5], міняє місцями початкове і фінальне значення лічильника циклу й замінює оператор зміни значення цього лічильника на кожній ітерації циклу.

Приклад застосування

Запропонований метод було використано під час оптимізації паралельного алгоритму задачі короткотермінового метеорологічного прогнозування [9]. Ця задача була обрана для практичного розгляду, оскільки вона має високу обчислювальну складність і природну необхідність у максимально швидкому и точному результаті. Її програмна реалізація поєднує геометричне та операторне розщеплення а схема обчислень є ітеративною. Для зберігання даних використовуються чотиривимірні масиви. Повний опис математичної моделі, декомпозиції області обчислень та методу розщеплення можна знайти у [6].

Чисельний експеримент проводився на гомогенній ЕОМ зі спільною пам’яттю й 24-ма задіяними процесорами. Зміна лише напрямку ітерування даними в результаті дала покращення загального часу виконання обчислень у 13 %. Що, у сукупності з геометричною декомпозицією вхідних даних й паралельною реалізацією операторного розщеплення, дозволило досягти мультипроцесорного прискорення $W(24) \quad 18.36$ загальної ефективність $E(24) \quad 0.76$, що є гарним показником.

Висновки

В роботі запропоновано підхід до перевірки коректності оптимізаційних перетворень паралельних програм, що виконуються автотьюнером, з використанням техніки переписувальних правил. Гнучкість підходу полягає у тому, що цю перевірку у часткових випадках можна виконати автоматично за вихідним кодом програми. Підхід було проілюстровано на прикладі методу вибору оптимальної стратегії обходу індексованих структур даних для програм з ітеративною схемою обчислень. Цей метод застосовано до оптимізації паралельного алгоритму складної прикладної задачі короткотермінового метеорологічного прогнозування. Результати практичного експерименту підтвердили доцільність застосування методу.

Література

1. Андон Ф.И., Дорошенко А.Е., Цейтлин Г.Е., Яценко Е.А. Алгеброалгоритмические модели и методы параллельного программирования. Киев: Академпериодика, 2007. 631 с.
2. Иваненко П.А., Дорошенко А.Ю. Метод автоматической генерации автотюнеров для параллелных программ. *Кибернетика и системный анализ*. 2014. № 3 С. 75–83
3. TermWare http://www.gradsoft.ua/products/termware_rus.html
4. Дорошенко А.Е., Шевченко Р.С. Система символьных вычислений для программирования динамических приложений. Проблеми програмування. 2005. № 4. С. 718–727.
5. TermWare tutorial, http://www.gradsoft.ua/rus/Products/termware/docs /tutorial_rus.html
6. Черниш Р.І. Модифіковане адитивно-усереднене розщеплення, його паралельна реалізація та застосування до задач метеорології: автореф. дис. канд. фіз.-мат. наук, Київ: КНУ ім. Т. Шевченка. 2010 р.
7. Глушков В.М. Алгебра. Языки. Программирование, 3-е изд. К: Наукова думка, 1989. с. 376.
8. Дорошенко А.Ю., Бекетов О.Г., Жереб К.А., Іваненко П.А., Овдій О.М., Шевченко Р.С., Яценко Е.А. Формальні та адаптивні методи й інструментальні засоби паралельного програмування. *Проблеми програмування*. 2017, № 3. С. 19–30.
9. Дорошенко А.Ю., Іваненко П.А., Овдій О.М., Яценко О.А. Автоматизоване проектування програм для розв'язання задачі метеорологічного прогнозування. *Проблеми програмування*. 2016. № 1. С. 102–115.

References

1. Andon P.I., Doroshenko A.Y., Zhereb K.A., Yatsenko O.A. Algebra-Algorithmic Models and Methods of Parallel Programming. Kyiv: Akadem- periodyka, 2018. 192 p. (in Ukrainian).
2. Ivanenko P.A., Doroshenko A.Y. Method of Automated Generation of Autotuners for Parallel Programs. *Cybernetics and Systems Analysis*. 2014. N 3. P.75–83. (in Russian).
3. TermWare http://www.gradsoft.ua/products/termware_rus.html. (in Russian).
4. Doroshenko A.Y., Shevchenko R.S. System of symbolic calculations for dynamic applications programming. *Problems in programming*. 2005. N 4. P. 718–727. (in Ukrainian).
5. TermWare tutorial, http://www.gradsoft.ua/rus/Products/termware/docs /tutorial_rus.html. (in Russian).
6. Chernysh R.I. Modified additive-averaged decomposition method, it's parallel implementation and application to meteorology tasks. (2010): abstract of dissertation of candidate of science in Physics and Math. Kyiv: Taras Shevchenko National University of Kyiv. (in Ukrainian).
7. Hlushkov V.M., Algebra. Languages. Programming (1989), 3rd edition. Kyiv: Naukova dumka. 376 p. (in Russian)
8. Doroshenko A.Y., Beketov, Zhereb K.A., Ivanenko P.A., Ovdii O.M., Shevchenko R.S., Yatsenko O.A. Formal and adaptive methods and software tools of parallel programming. *Problems in programming*. 2017. N 3. P. 19–30. (in Ukrainian).
9. Doroshenko A.Y., Ivanenko O.M. Ovdii O.A. Yatsenko. Automated program design for solution of weather forecasting problem. *Problems in programming*. 2016. N 1. P. 102–115. (in Ukrainian)

Одержано 10.03.2020

Про авторів:

Іваненко Павло Андрійович,
кандидат фізико-математичних наук,
науковий співробітник.
Кількість наукових публікацій в українських виданнях – 28.
Кількість наукових публікацій в зарубіжних виданнях – 3.
https://orcid.org/0000-0001-5437-9763.
https://scholar.google.com.ua/citations?user=aV5SrrAAAAAJ.

Місце роботи авторів:

Інститут програмних систем НАН України,

03187, м. Київ-187, проспект Академіка Глушкова, 40.
Тел.: (38)(044) 526-21-48.
E-mail: paiv@ukr.net