Національна академія наук України

Інститут програмних систем

ІВАНЕНКО Павло Андрійович

УДК 004.424

МЕТОДИ АВТОМАТИЗАЦІЇ СТВОРЕННЯ АВТОТЮНЕРІВ ДЛЯ ПАРАЛЕЛЬНИХ ПРОГРАМ

01.05.03 – математичне та програмне забезпечення обчислювальних машин і систем

Автореферат
дисертації на здобуття наукового ступеня
кандидата фізико-математичних наук

Київ – 2018

Дисертацією є рукопис.

Робота виконана в Інституті програмних систем НАН України.

Науковий керівник: доктор фізико-математичних наук, професор
Дорошенко Анатолій Юхимович,
Інститут програмних систем НАН України,
завідувач відділу.

Офіційні опоненти: доктор фізико-математичних наук,
старший науковий співробітник
Летичевський Олександр Олександрович,
Інститут кібернетики імені В.М. Глушкова
НАН України, старший науковий співробітник,

кандидат фізико-математичних наук, доцент
Панченко Тарас Володимирович,
Київський національний університет
імені Тараса Шевченка, доцент.

Захист відбудеться «14» грудня 2018 р. об 13 годині на засіданні спеціалізованої вченої ради Д 26.194.02 в Інституті кібернетики імені В.М. Глушкова НАН України за адресою:
03187, Київ-187, проспект Академіка Глушкова, 40.

З дисертацією можна ознайомитися в науково-технічному архіві Інституту кібернетики імені В.М. Глушкова НАН України за адресою:
03187, Київ-187, проспект Академіка Глушкова, 40.

Автореферат розісланий “12” листопада 2018 р.

Учений секретар
спеціалізованої вченої ради П.І. СТЕЦЮК

ЗАГАЛЬНА ХАРАКТЕРИСТИКА РОБОТИ

Актуальність теми дослідження. 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. Відомий вислів “ефективні алгоритми завжди кращі за суперкомп'ютери” дуже влучно формулює проблематику паралельних обчислень: наявність потужного обчислювача ще не гарантує успішного виконання обчислень у прийнятних часових межах. Під ефективністю у дисертаційній роботі розуміється швидкодія програми, тобто час її виконання.

Програмування ефективних алгоритмів значною мірою ускладнилося після переходу до багатоядерної архітектури мультипроцесорів. До цього переходу потужність процесорів зростала за законом Мура, а з нею зростала й швидкодія програм. Для старих програм все це відбувалося “безкоштовно” – без необхідності внесення змін у логіку обчислень. 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. Завдяки такому розподілу вихідна програма займається тільки розв'язанням своєї основної задачі й абстрагована від кількісних характеристик ОС, що значно спрощує її розробку.

Методологія автотюнінгу активно досліджується у міжнародній науковій спільноті (В. Панкратіус та К. Шафер в Технологічному університеті Карлсруе; К. Єлік у Каліфорнійському університеті (США); Р. Вудук у Технологічному інституті Джорджії (США); Томас Фарінгер у Інсбрукському університеті (Австрія); Такахіро Катагірі, Токіо (Японія) та багато інших). Вона вже довела свою універсальність й ефективність, проте не позбавлена недоліків, серед яких зазвичай виділяють значний час роботи автотюнера й необхідність його написання. Тому актуальним є пошук методів, які усувають ці недоліки.

Зв'язок роботи з науковими програмами, планами, темами. Наукові результати дисертаційної роботи отримані в рамках наступних науково-дослідних тем: “Розробка формальних та адаптивних методів, технологій та засобів паралельного програмування для неоднорідних мультипроцесорних обчислювальних систем” (№ держреєстрації 0112U002760, 2012–2016); “Розробка методів та комп’ютерних засобів автоматизації проектування сервісно-орієнтованого програмного забезпечення для обробки наукових даних на grid-платформах” (№ держреєстрації 0113U002160, 2014–2015); “Розробка математичних моделей і програмних засобів високопродуктивних обчислень для ефективного розв’язання задачі прогнозування метеорологічних процесів на території України на основі відеографічних прискорювачів” (№ держреєстрації 0114U005005, 2014).

Мета і завдання дослідження. Мета дисертаційної роботи – розробка методів автоматизації створення автотюнерів для прикладних програм, а також створення їх програмної реалізації. Головні вимоги до запропонованих методів:

- коректність змін у програмі, що оптимізується;
- універсальність, а саме: незалежність від предметної області застосування й мови програмування вихідного програмного додатку;
- автотюнер має генеруватися автоматично.

З цих вимог у дисертації виникають наступні задачі:

- розробити формальну модель автоматичного створення автотюнерів з вихідного коду програми;
- в рамках моделі розробити методи для покращення швидкодії оптимізовуваної програми;
- обґрунтувати коректність перетворень що використовується розробленими методами;
- забезпечити незалежність автотюнерів від предметної області й мов програмування;
- застосувати розроблені засоби для розробки ефективних паралельних програм метеорологічного прогнозування.

Об'єкт дослідження. Методи автотюнінгу прикладних програм, ядро яких становлять паралельні програми.

Предмет дослідження. Автоматизація процесу створення автотюнерів для підвищення ефективності паралельних програм.

Методи дослідження. У роботі використано як загальнонаукові, так і специфічні методи математичного аналізу й дискретної математики, розширення класичної моделі паралельної машини з довільним доступом (PRAM) та теорія переписувальних систем.

Наукова новизна отриманих результатів. В роботі отримано нові результати, які виносяться на захист:

- уперше розроблено формальний підхід до автоматичного створення автотюнерів з вихідного коду програми, що досягається за рахунок використання техніки переписувальних правил;
- побудовано формальну модель автотюнінгу, в якій формалізовані п’ять методів для покращення швидкодії оптимізовуваної програми з

контролем *Δ*-відхилення результатів обчислень: *метод пошуку оптимальної декомпозиції*; *метод вибору оптимального алгоритму*; *метод асинхронних циклічних обчислень*; *метод вибору оптимальної стратегії обходу індексованих структур даних*; *метод підстановки емпірично обчисленого значення*;

- у термінах дискретно-динамічних систем обґрунтована коректність методів налаштування паралельних програм;
- створено універсальну, у сенсі незалежності від предметної області й підтримки імперативних мов програмування, систему програмних засобів, яка реалізує розроблені методи;
- уведено поняття *Δ*-відхилення як характеристику якості роботи перетвореної автотюнером прикладної програми;
- розроблені засоби застосовано для оптимізації класу паралельних програм, що реалізують складні паралельні обчислення для важливої прикладної задачі метеорологічного прогнозування.

Теоретичне значення отриманих результатів полягає у розвитку методології автоматизованого самоналаштування прикладних програм на цільову платформу. Дисертаційна робота формалізує запропоновані методи самоналаштування і забезпечує їх ефективність і коректність. Ефективність методів обґрунтована й оцінена у запропонованому розширенні класичної теоретичної моделі PRAM, яке більш точно характеризує архітектуру сучасних обчислювальних систем. Для аналізу еквівалентності за результатом з заданими межами точності введено нове поняття Δ-відхилення результатів обчислень.

Практична цінність отриманих результатів. Розроблено систему програмних засобів, яка повністю реалізує запропоновані методи налаштування програмних додатків. Ця система може працювати в будь-якій операційній системі з підтримкою Java та програмними застосунками для будь-якої предметної галузі. Система була використана для оптимізації розв’язання практичної задачі – створення паралельного алгоритму короткотермінового метеорологічного прогнозування.

Особистий внесок здобувача. Всі основні результати дисертаційної роботи отримано здобувачем самостійно. Науковому керівнику належить постановка задачі й скеровування виконаних досліджень. У наукових працях, опублікованих у співавторстві, здобувачу належать: [1] – оптимізація програмної реалізації задачі метеорологічного програмування для систем з розподіленою пам’яттю; [2, 3] – проектування й програмна реалізація алгоритму автоматичної адаптації задачі метеорологічного прогнозування під цільове ОС; [4, 5] – запропоновано використання методології автотюнінгу для налаштування задачі метеорологічного прогнозування й розроблено окремий додаток-автотюнер; [6–8] – розробка системи автоматичного генерування автотюнерів TuningGenie; [9, 13] – проектування й розробка сервісної частини системи надання послуг метеорологічного прогнозування; [10–12] – формалізація методів системи TuningGenie.

Апробація результатів дослідження. Основні тези та окремі результати дисертаційної роботи обговорювалися на наукових семінарах кафедри Теорії та Технологій програмування факультету інформатики Київського національного

університету імені Тараса Шевченка та в Інституті програмних систем НАН України. Окрім цього тези дисертації доповідалися на таких міжнародних наукових конференціях: “Theoretical and Applied Aspects of Cybernetics” (Київ, 2011, 2012); “Теоретичні та прикладні аспекти побудови програмних систем” (Київ, 2011, 2012); “Математика, інформаційні технології, освіта” у 2013 р.; міжнародна конференція «Сучасна інформатика: проблеми, досягнення та перспективи розвитку» (Київ, 2013, 2017); УкрПРОГ (Київ, 2012, 2014); “High Performance Computing HPC-UA” (Київ, 2014); Computational and Experimental Physics (Краков, 2015).

Публікації. Основні наукові результати дисертаційної роботи опубліковано в 13 статтях у фахових виданнях, затверджених ДАК України. Стаття [7] опублікована у виданні, яке включено до міжнародних наукометричних баз і перекладається англійською мовою видавництвом “Springer”. Стаття [8] опублікована закордонним видавництвом “Springer” й входить до наукометричної бази “Scopus”. Робота [12] опублікована у науковому періодичному виданні іноземної держави, що входить до наукометричної бази “Scopus”. Також 6 публікацій у збірниках тез та матеріалів міжнародних наукових конференцій.

Структура та обсяг роботи. Дисертація складається із вступу, чотирьох розділів, висновків, списку використаних джерел із 64 найменувань. Загальний обсяг дисертації складають 149 сторінки, у тому числі основний текст викладено на 129 сторінках. Робота містить 22 рисунки, 4 таблиці та 4 додатки.

ОСНОВНИЙ ЗМІСТ РОБОТИ

У **вступі** обґрунтовано актуальність теми дисертаційного дослідження, сформульована мета дисертації, схарактеризовано завдання, об’єкт, предмет і методи дослідження, визначено зв’язок роботи з науковими програмами, темами, розкрито наукову новизну, її теоретичну та практичну значущість, наведено відомості про апробацію результатів дослідження та публікації, здійснено огляд основних результатів дослідження, описано структуру роботи.

У **першому** розділі досліджено основні поняття методології автотюнінгу, яка дозволяє автоматизувати процес налаштування програми за рахунок створення окремого додатку (тюнера). Цей тюнер має автоматично вибрати найкращу версію програми для цільового ОС, використовуючи її *емпіричну* оцінку. Основними критеріями оцінки, зазвичай, є швидкодія й точність отриманих результатів. Множина версій програми, з якої здійснюється вибір, має фіксований розмір й подається тюнеру на вхід.

У цілому дії програми-тюнера є досить шаблонними – обрати/створити нову версію програми й отримати емпіричну оцінку її швидкодії. Програмне рішення, запропоноване у дисертаційній роботі, дозволяє тюнеру *автоматично* генерувати нові коректні версії програм, спираючись на експертне знання розробника, оформлене у вигляді метаданих у вихідному коді.

Також у розділі пояснені переваги застосування методології щодо налаштування паралельних програм порівняно з альтернативними підходами

(розпаралелюючі компілятори), виконано огляд існуючих програмних рішень, що реалізують концепцію автотюнінгу, серед яких присутні: AtuneIL, ABCLibScript, Fiber Framework, наведена їх розширена класифікація, зазначені переваги й недоліки, а також сформульовано підхід, який дозволить якісно покращити методологію автотюнінгу.

У **другому** розділі розглянуто теоретичні моделі автотюнінга й методи налаштування програм у цих моделях.

У **підрозділі 2.1** запропоновано спеціальну модель PRAM*, яка розширює класичну модель PRAM додатковим рівнем пам'яті й використовує лише одну стратегію для регулювання одночасного доступу до обох рівнів. Нова модель PRAM* більш точно характеризує архітектуру сучасних обчислювальних систем (хоча й не претендує на абсолютну повноту) й дозволяє пояснити характер оптимізаційних перетворень, які виконує автотюнер. Розглянемо її визначення:

- будемо вважати, що обчислення виконуються в паралельній системі, що складається з N процесорів, де $N \in \mathbb{N}$. Тобто кількість процесорів є необмеженою, проте фіксована у кожному конкретному випадку. Під *процесором* будемо розуміти одиницю обчислювача, здатну автономно виконувати обчислення (відповідає поняттю ядра мікропроцесора у сучасних обчислювальних системах). Усі процесори системи будемо вважати однорідними;
- розмір локальної пам'яті (регістрів) кожного процесора будемо вважати обмеженим і позначимо M_{loc}. Час доступу до локальної пам'яті позначимо T_{loc}. Усі дані, що не вміщуються у локальну пам'ять, зберігаються в спільній пам'яті M_{shar}. Її обсяг необмежений, але й швидкість доступу значно повільніша: $T_{loc} \ll T_{shar}$;
- одночасний доступ до спільної пам'яті регулюється стратегією Concurrent Read Exclusive Write (CREW). Тобто різні процесори можуть паралельно читати дані з однієї комірки пам'яті, проте тільки один процесор може їх записувати в конкретну комірку пам'яті. Усі інші процесори, які в цей час хочуть прочитати чи записати данні в цю комірку, очікують у черзі. Позначимо загальні часові затрати процесорів на очікування розблокування $T_{N_{ew}}$.

Підрозділ 2.2 присвячено налаштуванню паралельних програм у моделі PRAM*.

У PRAM* кожна інструкція виконується за трьома фазами:

- читання даних із спільної пам'яті, якщо вони необхідні для виконання інструкції й не знаходяться в локальній пам'яті;
- локальні обчислення;
- запис результатів операції у глобальну пам'ять (якщо це необхідно).

Розглянемо часові затрати *i*-го процесора на доступ до пам'яті на 1-й та 3-й фазі за умови відсутності блокування з боку інших процесорів: $T_{i_{mem}} = = T_{i_{loc}} + T_{i_{shar}}$. Тобто цей час складається з доступу до локальної та глобальної пам'яті.

Також уведемо поняття синхронізації локальних обчислень процесора й

відповідних часових затрат. Якщо виконується критична секція – усі процесори, за винятком активного, перебувають у стані очікування, тому виконання таких секцій аналогічне виконанню у послідовній моделі RAM. Час обчислень таких секцій позначимо $T_{N_{seq}}$. А час «чистих» паралельних обчислень назвемо $T_{N_{par}}$. Тепер ми можемо перейти до розгляду прийомів оптимізації паралельних програм.

Загальний час виконання паралельної програми описуємо так:

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

Для пришвидшення програми необхідно мінімізувати часові затрати на синхронізацію $T_{N_{seq}} + T_{N_{ew}}$ й час виконання паралельних обчислень $T_{N_{par}}$.

Синхронізаційні затрати можна скорочувати за рахунок оптимального розподілу обчислювальних задач між процесорами. Наприклад, використовуючи традиційну схему крупнозернистого розпаралелювання, при якій вхідні дані розділяються таким чином, щоб кожен процесор незалежно обраховував якомога більший блок обчислень, і водночас усі процесори мають бути рівномірно завантажені. Тобто, необхідно підібрати оптимальну зернистість для задачі з паралелізмом за даними. Також можлива структурна модифікація обчислень програми, за якої довжина паралельної секції збільшиться за рахунок зменшення кількості критичних секцій. Така модифікація для специфічного класу ітеративних алгоритмів з бар'єрною синхронізацією у кінці кожної ітерації детально розглядається у підрозділі 3.3, а приклад реальної задачі представлено в пункті 4.2 четвертого розділу.

Для скорочення часу паралельних обчислень $T_{N_{par}}$ необхідно оптимізувати використання локальної пам'яті. Тобто, розподілити обчислення між процесорами таким чином, щоб вони повністю вмістилися у швидку локальну пам'ять. Таким чином для кожного процесора $T_{i_{shar}} \to 0$. У пункті 4.1.1 показано, як такий прийом працює для алгоритму сортування QuickSort.

Обидва розглянуті підходи до оптимізації паралельних алгоритмів, що зводяться до протилежних стратегій розподілення обчислень, ефективність яких безпосередньо залежить від таких характеристик обчислювального середовища, як обсяг та швидкість доступу до різних рівнів пам'яті й процесорних кешів. Очевидним є той факт, що оптимальна стратегія буде різною для різних архітектур. Потужність методології автотюнінгу полягає у тому, що підбір оптимальної стратегії виконується автоматично у будь-якому обчислювальному середовищі.

Як приклад розглянемо паралельну реалізацію блочного алгоритму множення квадратних матриць. Нехай він отримує на вхід дві матриці розмірністю $n \times n$, і нехай вони розбиваються на квадратні блоки розмірністю $m \times m$, де $m \in [1, n]$ й $s \times m = n$. Як відомо, обчислювальна складність блочного алгоритму збігається із складністю звичайного алгоритму множення й рівна $O(n^3)$. Швидкодія алгоритму залежить від часу обчислення блоку матриці й часових затрат на обмін даними. У багатьох випадках існує таке число m, при якому обсяг даних, необхідних для обчислення кожного блоку результуючої

матриці, не перевищує обсяг кеш-пам'яті процесора. У цьому випадку більш повільна (приблизно у 20 раз) оперативна пам'ять майже не використовується, що значно пришвидшує обрахування блоку. Проте, разом із зменшенням розміру блоку квадратично збільшується їх кількість, а отже й час на пересилку даних.

Оптимізація блочного алгоритму множення матриць – це пошук сідлової точки функції залежності часу обчислень від розбиття s. Змоделювати цю функцію складно, оскільки вона не лінійна, різна для різних обчислювальних середовищ, і для її побудови потрібна інформація про розміри кешів процесора, швидкість обмінів даними між процесорами й швидкість виконання базових арифметичних операцій. Методологія автотюнінгу дозволяє виконати налаштування автоматично, без додаткової інформації й для будь-якого середовища, а методи, запропоновані в дисертаційній роботі, дозволяють зробити це без модифікації структури блочного алгоритму.

У **підрозділі 2.3** побудована формальна модель автотюнінгу як еволюційне розширення дискретних динамічних систем (ДДС), сформульовані й доведені леми перевірки еквівалентності програм за результатом та за операторами у ДДС.

При побудові паралельних прикладних програм зазвичай найбільше уваги приділяється двом головним характеристикам: швидкодії та якості. Традиційно найвищий пріоритет має швидкодія. Якість, залежно від предметної галузі, може мати відношення до різних аспектів обчислень. Наприклад, в обчислювальних задачах – це точність результатів, в алгоритмах – архівування даних: ступінь компресії. Проте, зазвичай якість потребує додаткових затрат і впливає на швидкодію. Таким чином, одночасне отримання високих показників обох характеристик можливе лише за умови компромісу.

У дисертаційній роботі аналіз обмежується класом обчислювальних задач для яких головною характеристикою якості є точність обчислень. Далі будемо використовувати поняття Δ-відхилення для позначення величини розбіжності результатів обчислень паралельного алгоритму з «еталонним» результатом, отриманим на $PRAM^*$ машині з одним процесором.

Спочатку введемо поняття паралельної дискретної динамічної системи, а потім перейдемо до формального визначення автотюнера.

Як відомо, виконання будь-якої програми можна змоделювати з використанням теорії ДДС. Остання визначається як трійка (S_0, S, d), де S – простір станів системи, $S_0 \subseteq S$ – множина початкових станів системи, $d \subset S \times S$ – бінарне відношення переходів у просторі станів. Система може перейти з стану s_i в стан s_j, якщо $(s_i, s_i) \in d$. Під станом розуміється стан пам'яті $b: X \to D$, тобто відображення множини змінних програми X в область значень D. Виконання програми моделюється як переходи у відповідній базовій моделі виконання *послідовних* програм S^{ser}. У S^{ser} відношення переходів задаються правилами виконання базових операторів, розгалужень та ітерації, виклику процедури й виходу з неї.

Модель для мультипоточних програм S^{mt} складається з двох рівнів. На нижньому рівні це S^{ser}, розширена додатковими операторами для породження

($call\_thread(P_i,T_j)$) й синхронізації потоків. Цей рівень також називають рівнем потоків. На вищому рівні, який називають рівнем програми, станами є часткові відображення з множини потоків на множину станів нижчого рівня: $s^2 = \{T_1 \to s_1^1,...,T_k \to s_k^1\}$. Множина потоків задається стартовим потоком T_0 з усіма іншими потоками програми, які створюються оператором $call\_thread(P_i,T_j)$: $T = \{T_0\} \cup \{T_j \mid call\_thread(P_i,T_j) \in P\}$.

Відношення переходів S^{mt} складаються з правил S^{ser}, правила створення нового потоку, правил задання критичних секцій cs_i, правил виконання синхронізаційних конструкцій і правила завершення роботи потоку.

Як вже пояснювалося у першому розділі, автотюнінг зводиться до емпіричної оцінки різних модифікацій вихідної версії програми (P). Далі будемо позначати цю множину C. Оскільки у запропонованому в дисертаційний роботі підході усі модифікації отримуються за рахунок внесення змін у P, й кожній модифікації відповідає окрема ітерація тюнінгу, то можна розглядати C як послідовну еволюцію P.

Далі будемо вважати автотюнер еволюційною ДДС для паралельних програм і позначимо її S^{tune}. Виконання паралельної програми в S^{tune} повністю ідентичне її виконанню у моделі S^{mt}. У моделі S^{tune} змінні не мають типу й приймаються значення деякої універсальної множини D з нормованого простору. Нормою, наприклад, може бути Евклідова або Чебишевська (рівномірна) норма.

Перевірка коректності оптимізаційних перетворень. Під *коректністю перетворень,* які виконує автотюнер будемо розуміти те, що вихідна й перетворена програми повертають однаковий результат. Нехай задана підмножина $V_R \subset V$ *результуючих змінних*, тоді дві програми P^1 і P^2 будемо називати *еквівалентними за результатом*, якщо для однакових початкових даних b_0 програми одночасно приходять або не приходять у фінальні стани $s_f^1 = (b^1, \varepsilon, \emptyset)$ і $s_f^2 = (b^2, \varepsilon, \emptyset)$, при цьому ці фінальні стани пам'яті збігаються за результуючими змінними $b_{V_R}^1 = b_{V_R}^2$.

Визначимо наступні властивості програм: безтупиковість, безконфліктність й еквівалентність за операторами. Програму P будемо називати *безтупиковою*, 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^{ser} є безтупиковими.

У моделі S^{mt} 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.

Якщо відома послідовність виконання програми (послідовність переходів ДДС), то можна визначити історію використання базових операторів. Для цього додамо до стану ДДС ще одну компоненту $h \in Y$. У початковому стані $h = \varepsilon$. 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.

Лема 2.1. *Я*кщо дві програми P^1 й P^2 безтупикові, безконфліктні й еквівалентні за операторами, то вони еквівалентні за результатом.

Таким чином, перевірка еквівалентності за результатом зводиться до перевірки цих трьох властивостей, що є цілком можливим для багатьох оптимізаційних перетворень. У підрозділі 3.2 для методу вибору оптимальної стратегії обходу індексованих структур даних сформульовано властивості обчислень ітерацій програми, які гарантують еквівалентність за результатом вихідної й трансформованої програм, що доводиться відповідними теоремами.

Про перетворення, які змінюють структуру лише критичних секцій програми за рахунок переміщення, додавання або видалення операторів *lock* й *unlock*, будемо говорити, що вони задовольняють властивості *IdenticalCalculations.*

Для програм без блокуючих операторів *wait/signal* можна сформулювати наступне твердження, яке дозволяє перевіряти коректність перетворень, що не змінюють структуру критичних секцій.

Лема 2.2. *Я*кщо програми P^1 й P^2 задовольняють властивості *IdenticalCalculations*, то вони еквівалентні за операторами.

Лема 2.3. *Я*кщо програми P^1 й P^2 задовольняють властивостям *IdenticalCalculations й AdditionalLocks*, то з безконфліктності програми P^1 випливає безконфліктність P^2.

Лема 2.4. *Я*кщо програми P^1 й P^2 задовольняють властивостям *IdenticalCalculations й FewerLocks*, то з безтупиковості програми P^1 випливає безтупиковість P^2.

Таким чином побудована модель дозволяє доводити коректність перетворень програм. Усі теореми про коректність мають загальний вид: перетворення є коректним, якщо вихідна програма задовольняє певним умовам. У розділі 3 розглянуто способи автоматичної перевірки таких умов.

Для оптимізаційних перетворень, які змінюють історію викликів операторів у програмі, визначимо поняття *Δ-відхилення* як характеристику якості результатів обчислень.

Нехай ми маємо програму P та її модифікацію P^*. Нехай є вектор $x \subset V$ результуючих змінних програми. Фінальні стани пам'яті програм P і P^* позначимо $b(x)$ і $b^*(x)$ відповідно. Відстань між значеннями результуючих змінних позначимо $r(b(x), b^*(x))$. Нехай є вихідна програма P та її версія P^*,

перетворена автотюнером. Будемо вважати, що результат обчислень програми P^* *відхиляється на величину* Δ, якщо для однакових початкових даних b_0 програми P і P^* одночасно приходять (чи не приходять) у фінальні стани $S_f = (b^*, \varepsilon, \emptyset)$ $S_f^* = (b^*, \varepsilon, \emptyset)$ відповідно, для яких $r(b(x), b^*(x)) \leq \Delta$ при порожній залишковій програмі ε й порожньому заключному відношенні переходів $\emptyset$.

Тепер можна ввести формальне визначення автотюнінга:

система автотюнінга – це четвірка $< \text{ОФ(Р)}, Out, \varepsilon, P >$,

де P – вихідний варіант програми;

Out – «еталонний» результат обчислень програми P. Використовується для аналізу точності еволюційних версій P;

ε – допустима похибка обчислень.

$\text{ОФ}(P^*, Out, \varepsilon): Z$ – функція емпіричної числової оцінки продуктивності виконання програми з поточної ітерації тюнінгу. Повертає значення часових затрат на виконання обчислень у мілісекундах або -1, якщо Δ-відхилення перевищило порогове значення ε**.**

Задача автотюнера – це мінімізація значення ОФ. Тобто, автотюнер обирає найшвидшу еволюційну версію програми, яка задовольняє початковим умовам точності результату.

Кожна ітерація тюнінгу якісно оцінює генеровану версію програми:

$< P, C_i > \rightarrow P^*, \text{ОФ}_i$, де i – номер ітерації, P^* – новий варіант вихідної програми P, ОФ_i – її оцінка продуктивності.

На першій ітерації тюнінгу оптимальною програмою вважається її вихідний варіант $P^{opt} = P$. Перетворення програми на поточній ітерації тюнінгу будемо вважати *результативним*, якщо $0 < \text{ОФ}(P^*) < \text{ОФ}(P^{opt})$. Якщо виконане перетворення виявилося результативним, то отриманий варіант вихідної програми вважається покращеним і приймається за P^{opt}.

У загальному випадку автотюнер ітерує за множиною конфігурацій C. Після закінчення усіх ітерацій тюнінгу отримана програма P^{opt} вважається оптимальною для цільового ОС й зберігається для подальшого виконання.

У розділі 3 запропоновано п’ять методів автотюнінгу. Трансформації програм, що виконують ці методи, будемо аналізувати у формальній моделі автотюнінгу (підрозділ 2.3). Вихідний варіант програми позначимо P, її модифікацію – P_{tuned}. Поняття Δ-відхилення будемо використовувати для оцінки розбіжності результатів програм P і P_{tuned}.

Підрозділ 3.1 описує метод пошуку оптимальної декомпозиції області обчислень задачі. Цей метод підходить для налаштування дуже широкого класу задач з паралелізмом за даними й дозволяє автоматично підібрати оптимальну декомпозицію. Як відомі приклади таких задач згадаємо задачу блочного множення матриць й блочні методи Гаусса для розв’язання лінійної системи рівнянь та пошуку оберненої матриці.

Дуже часто оптимальна декомпозиція – це результат балансування між крупнозернистою схемою обчислення з мінімальними обмінами даних і затратами на синхронізацію з однієї сторони й дрібнозернистою декомпозицією

даних, за якої усі необхідні дані розміщуються в кеші процесора (якщо така декомпозиція можлива), з іншої.

З погляду ДДС історії виконання операцій над результуючими змінними програми ідентичні для різних декомпозицій за винятком порядку виконання арифметичних операцій. Тому, якщо для програми S виконується умова комутативності послідовних операторів *CommutativeOperators*, то результати обчислень (фінальні значення результативних змінних ДДС) збігаються, тобто дельта-відхилення рівне 0.

У підрозділі 2.2 на прикладі моделі PRAM* детально пояснено, за рахунок чого скорочується час виконання задачі при зміні зернистості декомпозиції. Зауважимо лише те, що для сучасних комп'ютерів швидкість зчитування даних з L1, L2 кешів процесора й спільної RAM пам'яті відрізняються приблизно на один порядок, тому заміна навіть невеликої частини операцій з RAM на аналогічні операції з L1/L2 кешами процесорів зазвичай суттєво покращує швидкодію програми.

Описаний у **підрозділі 3.2** метод вибору оптимальної стратегії обходу індексованих структур даних створено для оптимізації використання програмами процесорного кешу. Метод автоматично згенерує й оцінить швидкодію інверсного напрямку ітерування даними. У першу чергу цей метод запропоновано для задач що працюють з багатомірними даними. Для одномірних масивів послідовне зчитування елементів масиву у більшості випадків автоматично буде найшвидшим за рахунок апаратної оптимізації – доступ до "сусідніх" елементів значно швидший, якщо вони вміщуються в одну кеш-лінію. Проте, якщо програма працює з багатовимірними масивами або з їх представленням у одновимірному масиві, частіше за все сусідні елементи з точки зору ітерацій циклу не будуть потрапляти до однієї кеш-лінії. Тому варто оцінити усі комбінації ітерування: їх кількість рівна 2^d, де d – кількість вкладених циклів.

Єдина умова застосування методу: для трансформованих циклів має виконуватися вимога *CommutativeIterations* (комутативність ітерацій циклу). У цьому випадку коректність методу випливає з еквівалентності за операторами усіх варіантів вихідної програми. У загальному випадку автоматичну перевірку вимоги *CommutativeIterations* реалізувати важко, проте для часткових випадків це можливо.

Позначимо вихідний варіант програми *P1*, а її трансформовану варіацію із зміненим напрямком ітерування довільного циклу *ReversedCycleP1*. Будемо вважати, що ітерації циклу задовольняють вимозі *AllLocalVariables*, якщо внутрішні оператори використовують лише внутрішні змінні.

Теорема 3.1. Ітерації циклу є комутативними (властивість *CommutativeIterations*), якщо вони задовольняють властивості *AllLocalVariables*.

Тому розглянемо випадок, коли оператори циклу або читають, або записують значення у зовнішні змінні, але ніколи не роблять це одночасно. Така властивість (далі позначатимемо *ReadOrWriteAccessToVariables*) виключає рекурентну схему обчислень.

Сформулюємо 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 *OnlySequentialCalculations*.

Теорема 3.2. Програми *P1* й *ReversedCycleP1* еквівалентні за результатом, якщо ітерації трансформованих циклів задовольняють властивостям *OnlySequentialCalculations* й *ReadOrWriteAccessToVariables*.

Сформульовані властивості можна автоматично перевірити системою переписувальних правил TermWare, яка використовується у розробленому фреймворку автотюнінгу TuningGenie. Сам метод реалізовано за допомогою окремої прагми *bidirectionalCycle*. Її синтаксис й приклад застосування можна знайти у розділі 4.

У **підрозділі 3.3** запропоновано метод вибору оптимального алгоритму. Цей метод дозволяє емпірично підібрати найшвидший варіант алгоритму з декількох еквівалентних варіантів. Під еквівалентністю розуміється *еквівалентність за результатами обчислень*. Зауважимо, що множину вибору задає розробник й система автотюнінгу не перевіряє, чи є вони взаємозамінними, тим паче, не вигадує нові алгоритми.

Як приклад згадаємо різноманітні алгоритми сортування. Добре відомо, що сортування вставками InsertionSort у середньому буде виконуватися швидше за QuickSort, коли масив чисел цілком вміщається у кеш-пам'ять процесора, не зважаючи на вищу асимптотичну складність. Для сучасних процесорів таке явище можна спостерігати для масивів розмірністю до декількох сотень елементів.

Таким чином, метод дозволяє підібрати найкращу реалізацію алгоритму для конкретних ОС та вхідних даних програми (за умови що вони якісно не змінюються). З точки зору ДДС еквівалентність за історією виконання операцій не має місця – історії для різних алгоритмів, зазвичай, суттєво відрізняються, тому коректність методу перевіряється аналізом *Δ-відхилення* результатів обчислень. Для цілих результатів $\Delta = 0$ для еквівалентних алгоритмів. У випадку дійсних чисел $\Delta \to 0$ через похибку у машинному представленні чисел з рухомою комою.

Підрозділ 3.4 присвячено методу асинхронних циклічних обчислень, який пропонує відмовитися від деяких обмінів проміжними результатами обчислень.

Існує досить широкий клас паралельних алгоритмів з ітеративною схемою обчислень. Для деяких з них “склейка” декількох ітерацій дозволяє скоротити загальний час обчислень при збереженні прийнятної точності результатів. На рис. 1 показано трансформацію ітерацій.

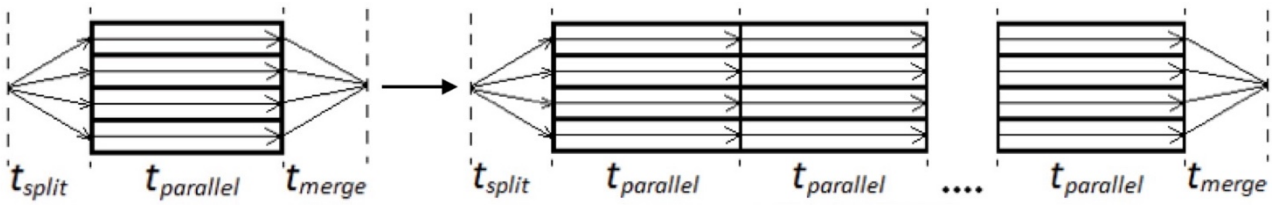

Рис. 1. Трансформація ітерацій циклу

Тут $t_{split}, t_{parallel}, t_{merge}$ – часові затрати на декомпозицію даних, виконання паралельної секції й обробку результатів обчислень відповідно.

Далі кількість поєднаних паралельних секцій будемо позначати m. Клас задач, до яких застосовна така модифікація схеми обчислень, будемо називати класом задач з властивістю *асинхронного циклу з ступенем свободи* m.

Легко побачити, що заміна схеми обчислень на асинхронний цикл підвищує ефективність паралельного алгоритму за рахунок скорочення часових затрат на декомпозицію даних й збір проміжних результатів обчислень. Оцінимо теоретичну межу прискорення модифікованої програми S_{tuned}.

Нехай n – кількість ітерацій у програмі P. Тоді її загальний час виконання рівен:

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

Нехай m – дільник n, $\mathrm{m} \in [1, \mathrm{n}]$. Тоді час виконання програми з використанням асинхронного циклу є наступним:

$$T_{P_{tuned}} = \frac{n}{m} \times \left(t_{split} + t_{merge}\right) + n \times t_{parallel}.$$

Позначимо $t_{exchange} = t_{split} + t_{merge}$, тоді мультипроцесорне прискорення буде рівне:

$$S_{tuned} = \frac{T_P}{T_{P_{tuned}}} = \frac{t_{parallel} + t_{exchange}}{\frac{1}{m} t_{exchange} + t_{parallel}}.$$

Верхня межа прискорення:

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

Нижня межа прискорення при $m = 1$:

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

Таким чином, $S_{tuned} \in \left[1{,}1 + \frac{t_{exchange}}{t_{parallel}}\right]$.

З точки зору ДДС функціональна еквівалентність не має місця для даного методу, тому що явно змінюється історія виклику операторів – скасовуються $\frac{n}{m}(m-1)$ секцій декомпозиції даних й обробки результатів обчислень ітерацій. З-за цього коректність методу зводиться до перевірки еквівалентності результату обчислень з заданими межами Δ-відхилення.

Зауважимо, що цей метод особливо ефективний коли $n \gg m$ і часові затрати на паралельну секцію й обмін даними – величини одного порядку. Легко бачити, що збільшення значення m призводить до збільшення S_{tuned}, проте також варто мати на увазі, що для багатьох задач використання великих значень m збільшує величину Δ-відхилення, тобто похибку. Зазвичай верхня межа для m визначається очікуваною точністю результатів чи необхідністю отримання проміжних результатів обчислень (наприклад, для візуалізації). Тому насправді цей метод системи TuningGenie емпірично розв'язує двоїсту задачу оптимізації швидкодії паралельного алгоритму та якості його результату.

У пунктах 4.1.3 й 4.2 подаються результати застосування методу асинхронних циклічних обчислень до оптимізації двох задач: задачі короткотермінового метеорологічного прогнозування й задачі моделювання броунівського руху.

Підрозділ 3.5 присвячено останньому розробленому методу – методу підстановки емпірично обчисленого значення. Цей метод дозволяє обчислити деяку функцію до виконання ітерацій тюнінгу й ініціалізувати її результатом якусь змінну в програмі. Він створювався для класу задач, які використовують у обчисленнях деяку якісну *незмінну* характеристику ОС й дозволяє автоматизувати її підстановку у код програми.

Очевидно, що залежно від використання цієї змінної історія викликів операторів відповідної ДДС може збігатися з її історією викликів вихідної, а може суттєво відрізнятися. Тому, перевіряти коректність трансформації можна лише через еквівалентність за результатами обчислень ДДС у межах заданого Δ-відхилення.

Підрозділ 3.6 описує архітектуру розробленого фреймворку. Програмна реалізація TuningGenie базується на системі обробки термів TermWare. У TermWare реалізована концепція термінальних систем об’єктних структур й переписувальних правил з явною взаємодією з фактологічною базою. TuningGenie за допомогою інструментарію TermWare отримує експертну інформацію з вихідного коду програми й генерує новий варіант програми на кожній ітерації тюнінгу. TermWare транслює вихідний код програми в терм й дозволяє міняти цей терм за використанням переписувальних правил, що дозволяє TuningGenie вносити структурні зміни у схему обчислень вихідної програми у декларативному стилі, тобто без безпосередньої зміни вихідного коду. Саме ця риса якісно вирізняє TuningGenie між схожими фреймворками й є дуже зручною, коли необхідно оптимізувати вже створену велику паралельну програму.

TermWare містить модулі для обробки Java й C# коду проте не обмежений цими двома мовами. Для підтримки інших імперативних мов програмування необхідно створити парсер, який буде транслювати вихідний код у мову TermWare й принтер для зворотного перетворення. Поточна версія TuningGenie працює лише з програмами на Java.

Інструментарій TuningGenie складається з трьох типів прагм:

1) **tuneAbleParam** – визначає область значень чисельної змінної. Підходить для підбору оптимальної зернистості розподілу даних паралельного алгоритму. З використанням цієї прагми можна підібрати деяке порогове значення для зміни схеми обчислень. Приклад такого застосування наведено у підрозділі 4.1.1;

2) **calculatedValue** – ініціалізує змінну значенням, яке обчислюється в цільовому ОС. Цей тип прагм створений для алгоритмів, які потребують для обчислень деяку чисельну характеристику ОС. Наприклад, швидкість доступу до різних рівнів пам’яті чи швидкість виконання базових арифметичних операцій;

3) **bidirectionalCycle** – ідентифікує цикли, для яких не важливий напрямок ітерування. TuningGenie спробує обидві версії ітерування й вибере найшвидшу. Зміна напрямку обходу даних впливає на ефективність використання процесорного кешу для задач, які працюють з багатомірними даними. Для задачі, яка сформульована у підрозділі 4.2, різниця часу виконання між найкращим і найгіршим варіантами склала майже 15 %.

У **четвертому** розділі представлені результати практичного застосування розробленого фреймворка.

В **підрозділі 4.1** розглянуті результати налаштування трьох ілюстративних обчислюваних задач з використанням розробленого фреймворка.

Підрозділ 4.1.1 містить опис налаштування демонстраційної задачі сортування чисел. Для оптимізації обрано класичний алгоритм сортування QuickSort. Його можна значно покращити, якщо для сортування підзадач невеликої розмірності використати більш ефективний з точки зору використання пам'яті алгоритм "внутрішнього сортування". Наприклад, сортування вставками не потребує додаткової пам'яті, отже краще працює на малих розмірах масивів. Оскільки зміна алгоритму сортування не впливає на результат обчислень, то оптимізація задачі виконується за єдиною характеристикою – часовими затратами. Оптимальне порогове значення розміру підзадачі для зміни алгоритму є різним для різних ОС, тому що залежить від розмірів процесорних кешів. Автотюнер за допомогою прагми *tuneAbleParam* сам знайде це значення для будь-якої ОС. Для ноутбука з двоядерним процесором тюнер визначив, що при сортуванні цілих чисел типу *integer* варто переходити до сортування вставками на підзадачах розміром менше ніж 90 елементів. Це скоротить час сортування в середньому на 30 відсотків.

Підрозділ 4.1.2 описує результати застосування системи TuningGenie для оптимізації паралельного алгоритму сортування MergeSort. Цей приклад є хорошою демонстрацією оптимізації паралельної програми з декількома залежними параметрами. Як і в попередньому випадку, використано прийом переходу до сортування вставками для масивів малої розмірності. До параметру *insertionSortThreshold,* який ми вже бачили у попередньому експерименті, додається два нових: кількість потоків (*parallelism*), задіяних у програмі, та мінімальний розмір масиву, для якого породжується нова незалежна обчислювальна задача *mergeSortBucketSize*.

Практичний експеримент виконувався на ПК з чотирма ядрами, 8MB L3 кеш пам'яті й 16 GB RAM. Заміри виконувалися для сортування $2\times10e^7$ цілих чисел. Автотюнер знайшов наступну оптимальну конфігурацію: parallelism = 8, insertionSortThreshold = 120, mergeSortBucketSize = 50000. Середній час сортування (10 замірів): 898 мс. Мультипроцесорне прискорення: 4.93. Результат можна вважати добрим, оскільки отримане мультипроцесорне прискорення виявилося більшим за кількість ядер процесора (4).

Підрозділ 4.1.3 присвячено результатам застосування системи TuningGenie для оптимізації паралельної реалізації спрощеної моделі броунівського руху. Ця задача є прикладом, коли застосування методу асинхронних циклічних обчислень не впливає на точність результатів.

Нехай нам потрібно знати положення молекул через деякий інтервал часу Δt. Поділимо цей інтервал на n рівних відрізків і будемо вважати, що молекула за один відрізок робить один "крок" – переміщується на одиницю відстані. Таким чином задача зводиться до моделювання положення молекул ідеального газу через n "кроків". Ступінь свободи асинхронного циклу m для задачі визначається необхідністю отримання проміжних результатів обчислень. Якщо вони не потрібні (наприклад, візуалізація не потрібна), то $n = m$. Вплив значення параметра m на загальну швидкодію очевидний – збільшення кількості незалежних ітерацій зменшує затрати на синхронізацію, а тому й загальний час виконання.

У тестовому середовищі з двома процесорами по 4 ядра кожен, 16 GB RAM для експерименту з $\mathrm{n} = 300000$ при значеннях $m > 150$ мультипроцесорне прискорення досягало значення $S = 7.5$, що згідно з законом Амдала, є близьким до теоретичної межі для тестового середовища.

У **підрозділі 4.2** подано результати налаштування паралельного алгоритму короткотермінового метеорологічного прогнозування. Ця задача промислового масштабу була обрана для практичного розгляду, оскільки вона має високу обчислювальну складність і природну необхідність у максимально швидкому й точному результаті.

Загальна схема обчислень є ітеративною і показана на рис. 2. На вхід першої ітерації подаються поточні значення метеорологічних величин у області, що розглядається, наприклад напрям й швидкість вітру, вологість, температура повітря тощо. Результатом обчислення кожної ітерації є прогноз стану системи через деякий час Δt, який подається на вхід наступній ітерації. Чим більший часовий крок Δt, тим більша прогностична похибка. Рекомендовано використання часового кроку від десятків до сотень секунд.

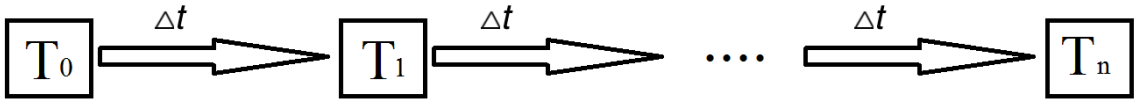

Рис. 2. Схема обчислень задачі метеорологічного прогнозування

Чисельний метод розв'язання задачі дозволяє виконувати геометричну декомпозицію області обчислень, незалежно обчислювати кожну фізичну величину, а також використати асинхронний цикл, за умови що кількість "вільних" ітерацій менша за 10. При дотриманні цього обмеження зберігається"вільних" ітерацій менша за 10. При дотриманні цього обмеження зберігається збіжність чисельного методу, й похибка залишається у прийнятних межах, що математично доведено розробниками чисельного методу.

Розглянутий чисельний метод можна схарактеризувати як досить зручний для паралельного виконання, оскільки для більшої частини обчислень виконується умова *CommutativeOperators* (їх можна виконувати незалежно), а геометрична декомпозиція дозволяє підлаштовувати кількість підзадач до кількості процесорів ОС.

Чисельний експеримент здійснювався на гомогенній ЕОМ з 24 процесорами Dual Core Intel Itanium 2 Series 9000 із частотою 1.6 ГГц та архітектурою IA64. Система доступу до пам'яті NUMAlink 4, топологія доступу fat-tree, загальний обсяг оперативної пам'яті 96 ГБ. Обчислювався прогноз на 24 години з часовим кроком $\Delta t = 10$ секунд. Кількість "вільних" ітерацій була задана як $m = 10$. Обчислення проводилися на сітці розмірності $180 \times 180 \times 11$.

У результаті чисельного експерименту була знайдена декомпозиція вхідної області $S = 4$, при якій програма показала мультипроцесорне прискорення $W(24) = 18.36$ й загальну ефективність $E(24) = 0.76$, що є хорошим показником.

ВИСНОВКИ

У дисертаційній роботі розроблені, обґрунтовані, формалізовані та реалізовані методи автоматизації створення автотюнерів для прикладних програм. Отримано наступні результати, що мають наукову й практичну цінність:

1. Запропоновано п'ять методів автотюнінгу які підходять для оптимізації широкого класу паралельних програм.
2. Формалізовано процес автотюнінгу. Запропоновані методи, пояснені у теоретичній моделі PRAM*, яка базується на відомій класичній моделі PRAM. Коректність перетворень, що виконують методи, доведено у новій еволюційній дискретно-динамічній системі для паралельних програм S^{tune}. Для цього введено поняття Δ-відхилення множини результуючих змінних, нове для класичних ДДС.
3. Відповідно до теоретичних результатів дисертації розроблено систему TuningGenie, яка реалізує запропоновані методи автотюнінгу. TuningGenie майже повністю автоматизує процес оптимізації програмних додатків – користувачу необхідно лише додати коментарі-підказки до вихідного коду програми. Система вирізняється від аналогічних рішень універсальністю (не залежить від предметної області й підтримує усі імперативні мови програмування) і гнучкістю (дозволяє декларативно виконувати структурні зміни програми, що оптимізується), досягнуті за рахунок використання техніки переписувальних правил для роботи з вихідним кодом програми.
4. Система TuningGenie практично застосована до оптимізації декількох програм, зокрема, паралельної програми, що розв'язує складну обчислювальну задачу короткотермінового метеорологічного прогнозування.

СПИСОК ОПУБЛІКОВАНИХ РОБІТ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

1. Черниш Р.І., Тирчак Ю.М., Іваненко П.А. Побудова паралельного алгоритму чисельного розв'язання багатовимірної задачі моделювання навколишнього середовища. *Проблеми програмування*. 2009. № 3. С. 85–91.
2. Іваненко П.А., Дорошенко А.Ю., Суслова Л., Черниш Р.І. Автоматично налагоджуваний паралельний алгоритм чисельного розв’язання багатовимірної задачі моделювання навколишнього середовища. *Проблеми програмування*. 2010. № 2–3. С. 202–208.
3. Іваненко П.А., Дорошенко А.Ю. Автоматична оптимізація виконання для задачі метеорологічного прогнозування. *Проблеми програмування*. 2012. № 2–3. С. 426–434.
4. Іваненко П.А., Дорошенко А.Ю. "Засоби створення систем автоматичного настроювання для ефективного виконання прикладних паралельних програм". *Інформаційні технології в освіті*. Збірник наукових праць. 2013. № 14. С. 17–21.
5. Іваненко П.А, Дорошенко А.Ю., Овдій О.М, Суслова Л.М. Автотюнер та візуалізація для задачі метеорологічного прогнозування. *Проблеми програмування*. 2013. № 4. С. 64–73.
6. Ivanenko P.A., Doroshenko A.Y., Zhereb K.A. TuningGenie: Auto-Tuning Framework Based on Rewriting Rules. Information and Communication Technologies in Education, Research, and Industrial Applications. *10th International Conference*, ICTERI 2014, Kherson, Ukraine, June 9–12, 2014. P. 139–158.
7. Иваненко П.А., Дорошенко А.Ю. Метод автоматической генерации автотюнеров для параллельных программ. *Кибернетика и системный анализ*. 2014. № 3. С. 75–83.
8. Ivanenko P.A., Doroshenko A.Y. Application of rule rewriting system to software automated tuning. Proceedings of Fourth International Conference "*High Performance Computing*" HPC-UA. 2014. P. 41–47.
9. Дорошенко А.Ю., Овдій О.М., Павлючин Т.О., Вітряк Є.А., Іваненко П.А. До створення інтернет-порталу надання послуг метеорологічного прогнозування на мультипроцесорній платформі. *Проблеми програмування*. 2015. № 3. С. 24–32.
10. 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.
11. Дорошенко А.Ю, Іваненко П.А., Новак О.С. Гібридна модель автотюнінгу з використанням статистичного моделювання. *Проблеми програмування*. 2016. № 4. С. 27–32.
12. Doroshenko A., Ivanenko P., Ovdii O., Yatsenko O. Automated Program Design – an Example Solving a Weather. Forecasting Problem. *Open Physics*. 2016. Vol. 14, Issue 1. P. 410–419.
13. Дорошенко А.Ю., Іваненко П.А., Новак О.С. Оптимізація автотюнінгу програм з використанням нейромереж. *Проблеми програмування*. 2017. № 2. С. 40–47.

14. 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*. Proceedings of the International Scientific Conference of Students and Young Scientists. Kyiv: Bukrek. 2011. P. 88–89.
15. Іваненко П.А., Дорошенко А.Ю. Автотюнер для багатовимірної задачі моделювання навколишнього середовища. International Conference "*Theoretical and Applied Aspects of Program Systems Development (TAAPSD'2011)*". Abstracts. 2011. P. 71–75.
16. Ivanenko P. TuningGenie - an autotuning framework for optimization of parallel applications. *Theoretical and Applied Aspects of Cybernetics*. Proceedings of the 2nd International Scientific Conference of Students and Young Scientists. Kyiv: Bukrek, 2012. P. 27–30.
17. Дорошенко А.Ю., Жереб К.А., Яценко О.А., Іваненко П.А. Програмний комплекс для автоматизації програмування високопродуктивних обчислень. Тези Міжнародної конференції «*Сучасна інформатика: проблеми, досягнення та перспективи розвитку*» присвячена 90-річчю від дня народження академіка В.М. Глушкова. 2013. Інститут кібернетики імені В.М. Глушкова НАН України. С. 176.
18. Yatsenko Olena, Doroshenko Anatoliy, Ivanenko Pavlo, Ovdii Olga. Automated Program Design – an Example of Solving Weather Forecasting Problem. Proceedings of the Congress on Information Technology, Computational and Experimental Physics (CITCEP 2015), Kraków, Poland, 18–20 December 2015. AGH University of Science and Technology Press. Kraków, 2016. P. 17–20.
19. Дорошенко А.Ю, Бекетов О.Г., Жереб К.А., Іваненко П.А. та інші. Формальні й адаптивні методи та інструментарій автоматизації паралельного програмування на основі алгеброалгоритмічного підходу. Тези Міжнародної наукової конференції «*Сучасна інформатика: проблеми, досягнення та перспективи розвитку*» присвячена 60-річчю заснування Інституту кібернетики імені В.М. Глушкова НАН України. Інститут кібернетики імені В.М. Глушкова НАН України. Київ, 2017. С. 45–50.

АНОТАЦІЯ

Іваненко П.А. Методи автоматизації створення автотюнерів для паралельних програм.

Дисертація на здобуття наукового ступеня кандидата фізико-математичних наук за спеціальністю 01.05.03 – математичне та програмне забезпечення обчислювальних машин і систем. – Інститут програмних систем НАН України, Київ, 2018.

Дисертація присвячена розробці методів і моделей для автоматизації оптимізації програм у сучасних паралельних платформах. Розроблені методи формалізовані у побудованих моделях автотюнінга. Ефективність методів пояснюється й аналізується у моделі PRAM* – запропонованому розширенні класичної моделі додатковим рівнем пам'яті. Коректність методів доводиться у

термінах дискретних динамічних систем. Для цього було побудовано алгебро-динамічну модель автотюнінгу для мультипроцесорних платформ. Також створено програмну реалізацію усіх запропонованих методів автотюнінга, що не залежить від предметної області задачі й операційної системи обчислювального середовища. Система автотюнінга базується на системі правил переписування й орієнтована в першу чергу на програмні додатки на мові Java. Загалом запропоноване рішення підходить для будь-якої імперативної мови програмування. Розроблені засоби було застосовано для оптимізації складних обчислювальних задач, в результаті досягнуто значні показники їх ефективності.

Ключові слова: автоматизація оптимізації паралельних програм, автотюнінг, паралельні програми, алгебро-динамічні моделі, техніка переписувальних правил, доведення коректності перетворень.

ABSTRACT

Ivanenko P.A. Methods for automation of development of autotuners for parallel programs. – Manuscript.

Candidate's thesis on Physics and Mathematics (Ph.D.), specialty 01.05.03 – Mathematics and software for computing systems. Institute of Software Systems of NAS of Ukraine, Kyiv, 2018.

The thesis is devoted to the development of methods and models for automating optimization of software for parallel platforms. Complex scientific tasks are known to require significant computing resources and optimal utilization of the latter is one of the main problems in software development. Things got much more complicated with a relatively recent switch to multicore architecture. Modern processors contain multiple cores and several layers of cache memory so the architecture of multiprocessor systems is way more complex, compared to their ancestors.

Drop in memory prices (both RAM and hard drives) and expanse of volumes caused a shift in optimization paradigm – modern algorithms must to effectively distribute data over all available computing resources and benefit from faster memory layers utilization (CPU caches/RAM). The classical approach is to move tuning of your program to a specific environment into separate phase – after development is complete.

Autotuning is a modern optimization methodology that earned its popularity due to simplicity and versatility. The main idea of autotuning is to perform the empirical evaluation (benchmark) of different variations of a program in a target environment and choose the best one. Usually benchmarking is performed by an external application that is called 'tuner'. The tuner also chooses/creates a new version of the program to benchmark so program you want to optimize is unaware of optimization logic.

The autotuning methodology is actively researched in the scientific community and proved to be efficient. However, there are two main drawbacks: empirical experiments require a significant amount of time and you have to create an additional application (tuner). Therefore, methods for addressing mentioned shortcomings are in

high demand.

The scientific novelty of the obtained results is a formal description of proposed autotuning methods. Work presents autotuning framework TuningGenie that uses term rewriting approach for source code transformations. With such approach, some formulated characteristics of computational logic can be automatically checked so correctness of optimizing transformations can be validated. Also utilizing means of rewriting rule systems is more agile comparing to text-like transformations – you can apply structural changes to your code like reversing the order of iterations over multidimensional data, etc.

This thesis offers five new autotuning methods: method for finding optimal decomposition; method of choosing an optimal algorithm; method of asynchronous cyclic computing; method of choosing an optimal strategy for indexed data structures traversal; method of substituting an empirically calculated value.

Effectiveness and nature of proposed methods are explained and estimated in a new model PRAM*. This model extends the conventional PRAM model with an additional layer of quick but limited memory and uses only one strategy for concurrent memory access orchestration. PRAM* does not claim absolute completeness but provides more precise view on the architecture of modern computing systems. Main purpose of PRAM* is to explain the nature of the optimizing transformations, performed by the autotuner.

The *correctness* of the methods is proved in terms of discrete dynamic systems. Dynamic algebra model for multicore systems was created for this purpose.

Autotuning is to empirically evaluate various modifications of the original version of the program P. These versions are defined by a set of configurations C. C is constructed from source code metadata. So, for each element from configurations set tuner generates a new version of the initial program and this version is evaluation within separate tuning iteration. That's why process autotuning is considered as a sequential evolution of initial program P and autotuner is defined by an *evolutional discrete dynamic system for parallel programs*.

As already mentioned, in this research correctness of optimizing transformations is justified in terms of discrete dynamic systems. 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 three attributes of initial and modified program versions – *an absence of deadlocks, an absence of data race conflicts and equivalence by operators*. In general, these properties are very difficult to verify. However, for some methods, partial cases have been formulated for which these properties can be verified with the help of a rewriting rules technique. For all other cases, the concept of Δ-deviation for discretely dynamic systems was introduced. It is used as characteristics of the quality of the results of calculations to check characteristic of *equivalence by the result with Δ-deviation.*

The results of the dissertation work are of a theoretical and practical nature and were motivated directly by a practical application. Therefore, a program implementation of all proposed methods of auto-tuning was created. This implementation is domain-independent and is suitable for any operating system that

has Java virtual machine implementation. The auto-tuning system is based on rewriting rules framework and is focused primarily on software applications written in Java language. In general, the proposed solution is suitable for any imperative programming language.

The effectiveness of the developed methods and tools was demonstrated with well-known model examples – optimization of sequential and parallel sorting algorithms, as well as the problem of modeling Brownian motion in an ideal gas. Also, the autotuning system was used to optimize the complex practical real-time task – a parallel algorithm for short-term meteorological forecasting, which is used in the internet portal of the Institute of Software Systems of the National Academy of Sciences.

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

АННОТАЦИЯ

Иваненко П.А. Методы автоматизации создания автотюнеров для параллельных программ.

Диссертация на соискание ученой степени кандидата физико-математических наук по специальности 01.05.03 – математическое и программное обеспечение вычислительных машин и систем. – Институт программных систем НАН Украины, Киев, 2018.

Диссертация посвящена разработке методов и моделей для автоматизации оптимизации программ для современных параллельных платформ. Разработанные методы формально описываются в построенных моделях автотюнинга. Анализ эффективности и объяснение природы оптимизационных трансформаций выполняется в модели PRAM* – расширении классической модели PRAM дополнительным уровнем быстрой, но ограниченной по размеру памяти. Корректность методов доказывается в терминах дискретных динамических систем. Для этого была построена алгебро-динамическая модель автотюнинга для мультипроцессорных платформ. Также была разработана независимая от предметной области задач и операционной системы программная реализация всех предложенных методов автотюнинга. Система автотюнинга использует технику переписывающих правил для трансформации исходного кода и ориентирована в первую очередь на программы, написанные на языке Java. В общем случае предложенное решение подходит для любого императивного языка программирования. Разработанные средства были продемонстрированы на известных модельных примерах задач сортировки и моделирования броуновского движения, а также использованы для оптимизации сложной вычислительной задачи кратковременного метеорологического прогнозирования.

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

Підп. до друку 01.11.2018. Формат 60x84/16. Папір офс.
Цифровий друк. Ум. друк. арк. 1,39.
Обл.-вид. арк. 1,0. Тираж 100 прим.

Редакційно-видавничий відділ
Інституту кібернетики імені В.М. Глушкова НАН України
03680, Київ-187, проспект Академіка Глушкова, 40