УДК 004.4'24

АВТОТЮНІНГ ПАРАЛЕЛЬНИХ ПРОГРАМ ІЗ ВИКОРИСТАННЯМ СТАТИСТИЧНОГО МОДЕЛЮВАННЯ ТА МАШИННОГО НАВЧАННЯ

А.Ю. Дорошенко, П.А. Іваненко, О.С. Новак, О.А. Яценко

Автотюнінг для складних і нетривіальних програмних систем зазвичай вимагає багато часу внаслідок емпіричного оцінювання великої множини варіантів значень параметрів вхідної паралельної програми у цільовому середовищі виконання. У даній роботі запропоноване вдосконалення методу автотюнінгу на основі використання статистичного моделювання та нейромережевих алгоритмів, що дозволяє суттєво звузити простір можливих значень параметрів, що аналізуються. Застосування підходу продемонстроване на прикладі автотюнінгу паралельної програми сортування, яка комбінує декілька методів сортування, на основі автоматичного навчання нейромережевої моделі на результатах "традиційних" циклів тюнінгу з подальшою заміною частини запусків автотюнера оцінкою зі статистичної моделі.
Ключові слова: автотюнінг, автоматизація розробки програмного забезпечення, машинне навчання, нейромережа, паралельні обчислення, статистичне моделювання.

Автотюнинг для сложных и нетривиальных программных систем обычно требует много времени вследствие эмпирического оценивания большого множества вариантов значений параметров входной параллельной программы в целевой среде выполнения. В данной работе предложено усовершенствование метода автотюнинга на основе использования статистического моделирования и нейросетевых алгоритмов, что позволяет существенно сузить пространство возможных значений параметров, которые анализируются. Применение подхода продемонстрировано на примере автотюнинга параллельной программы сортировки, комбинирующей несколько методов сортировки, на основе автоматического обучения нейросетевой модели на результатах "традиционных" циклов тюнинга с дальнейшей заменой части запусков автотюнера на оценку из статистической модели.
Ключевые слова: автотюнинг, автоматизация разработки программного обеспечения, машинное обучение, нейросеть, параллельные вычисления, статистическое моделирование.

Auto-tuning for complex and nontrivial parallel systems is usually time-consuming because of empirical evaluation of huge amount of combinations of parameter values of an initial parallel program in a target execution environment. This paper proposes the improvement of the auto-tuning method using statistical modeling and neural network algorithms that allow to reduce significantly the space of possible combinations of parameters values to analyse. The resulting optimization is illustrated by an example of tuning of parallel sorting program, that combines several sorting methods, by means of automatic training of a neural network model on results of "traditional" tuning cycles with subsequent replacement of some auto-tuner calls with an evaluation from the statistical model.
Key words: auto-tuning, software development automation, machine learning, neural network, parallel computing, statistical modeling.

Вступ

Для вирішення сучасних прикладних наукових задач необхідним є використання серйозних обчислювальних потужностей паралельних платформ, і тому етап оптимізації програм для конкретного високопродуктивного середовища обчислень займає значне місце в процесі їх розробки. Проблема полягає в тому, що потужна обчислювальна платформа не гарантує високу продуктивність та ефективність прикладних застосувань: для досягнення ефективності необхідними є зусилля розробника. Парадигма автотюнінгу [1, 2], яка за останнє десятиріччя стала стандартом для вирішення проблеми оптимізації програмних застосувань, дозволяє повністю автоматизувати цей процес для будь-якого обчислювального середовища. Її популярність зумовлена в першу чергу простотою застосування й незалежністю від якісних характеристик обчислювача та його операційної системи. Автотюнінг традиційно використовує емпіричний підхід для отримання якісної оцінки оптимізованої програми (під якістю зазвичай розуміють швидкодію й точність результату). Він автоматизує пошук оптимального варіанту програми з множини передбачених можливостей шляхом виконання кожного варіанта та вимірювання його продуктивності на заданій паралельній архітектурі. Основною перевагою даного підходу є високий рівень абстракції – програма оптимізується без знання деталей апаратної реалізації, таких, як кількість процесорних ядер, розмір кешу та швидкість доступу до пам'яті. Замість цього, автотюнінг оперує високорівневими поняттями з предметної області, такими, як кількість і розмір незалежних задач, або особливостями алгоритму розв'язання задачі.

У попередніх роботах [3–8] авторами розроблялися теорія, методологія та інструментальні засоби автоматизованого проектування, синтезу й автотюнінгу програм, що ґрунтуються на системах алгоритмічних алгебр В.М. Глушкова та переписуванні термів. Запропоновані модель оптимізації паралельних програм та інструментарій генерації автотюнерів TuningGenie, призначений для автоматизованого налаштовування програм на цільові платформи [5, 6]. Інструментарій працює з вихідним кодом паралельних програм і виконує перетворення програмного коду з використанням системи переписувальних правил TermWare [3].

Для задачі розробки автотюнерів у літературі запропоновано багато підходів [1, 2]. Відомими прикладами автотюнерів є ATLAS [9] та FFTW [10] – спеціалізовані бібліотеки, що вносять високопродуктивну реалізацію деяких специфічних функцій. На відміну від розробленого нами інструментарію TuningGenie, який забезпечує оптимізацію, незалежну від предметної області, згадані системи прив'язані до предметної області та

мови програмування. Система TuningGenie є подібною до Atune-IL [11] – мовного розширення для автотюнінгу, яке використовує прагми і не прив'язане до конкретної мови програмування. Основна відмінність TuningGenie полягає у використанні переписувальних правил для трансформації вихідного коду. Подання програмного коду у вигляді терму дозволяє модифікувати структуру програми у декларативному стилі. Ця особливість значно розширює можливості системи автотюнінгу.

Основним недоліком автотюнінгу є значні накладні витрати на процес оптимізації: якщо кількість програмних варіантів є достатньо великою, процес оптимізації може тривати багато годин і навіть днів. У даній статті запропоноване вдосконалення гібридного підходу до автотюнінгу [7], який використовує статистичне моделювання, за рахунок застосування методів машинного навчання для зменшення часу, необхідного для пошуку оптимального варіанту програми. Підхід полягає в автоматичному навчанні моделі на результатах звичайних циклів тюнінгу й подальшій підміні частини запусків програми оцінкою зі статистичної моделі.

1. Інструментарій автотюнінгу та машинне навчання

В роботах [5, 6] було розроблено програмну систему TuningGenie, призначену для автоматизованої генерації автотюнерів з вихідного коду. Ідея автотюнера полягає в емпіричному оцінюванні декількох версій вхідної програми та вибору найкращого, при цьому критерієм оцінки є менший час виконання вхідної програми та точність отриманих результатів. Інструментарій працює з вихідним кодом програм, використовуючи експертні знання розробника. Розробник додає певні метадані (назви та діапазони значень параметрів) до вихідного коду у вигляді спеціальних коментарів-прагм. Використання прагм дозволяє не виконувати складний аналіз залежності між даними у вихідному коді застосунку, а також суттєво звужує область пошуку оптимальної конфігурації.

Для трансформацій вихідного коду використовується система TermWare [3], що ґрунтується на техніці переписувальних правил, яка дозволяє подавати код застосунків у вигляді термів. Поточна версія TermWare містить компоненти для взаємодії з мовами програмування Java та C#. TuningGenie використовує TermWare для вилучення експертних знань з вихідного програмного коду і генерації нової версії програми на кожній ітерації автотюнінгу. База знань використовується як сполучна ланка між фазою, що є передує автотюнінгу, і фазою автотюнінгу. Маніпулюючи термами, що подані як абстрактні синтаксичні дерева (abstract syntax trees, AST), TuningGenie виконує структурні зміни в обчисленнях програми, використовуючи декларативний стиль системи TermWare.

Застосування автотюнінгу для складних програмних систем зазвичай вимагає багато часу внаслідок емпіричного оцінювання великої множини варіантів значень параметрів вхідної паралельної програми у цільовому середовищі виконання (позначимо цю множину через C). У даній роботі пропонується оптимізація методу автотюнінгу на основі використання статистичного моделювання та машинного навчання. Покращення полягає у зменшенні кількості запусків автотюнера шляхом побудови апроксимаційної моделі, що дозволяє "відкинути" найменш вірогідні заміри. Результатом апроксимації моделі зазвичай є скорочення розмірності вхідних параметрів множини C, що означає значне прискорення процесу автотюнінгу.

У широкому розумінні методи машинного навчання ґрунтуються на понятті отримання певного досвіду з даних [2, 12]. Отриманий досвід може мати різну природу, наприклад, це може бути класифікація, модель функції та ін. У контексті автотюнінгу досвідом є оптимальні значення параметрів програми. Спочатку метод машинного навчання оцінює декілька альтернатив в рамках простору пошуку для n різних вхідних програм $P_1,...,P_n$, що визначаються конфігураціями $C_1,...,C_n$. Множина оцінених альтернатив називається навчальними даними (training data). Процес генерації та оцінки навчальних даних і отримання досвіду з цих даних називається навчанням (training). Після завершення навчання, маючи нову версію програми P', яка підлягає оцінці, виконання P' замінюється на оцінку, отриману з навченої моделі.

Машинне навчання тісно пов'язане (і часто перетинається) з обчислювальною статистикою, дисципліною, яка також фокусується на прогнозуванні шляхом застосування комп'ютерів [13]. Усі статистичні алгоритми (у тому числі й алгоритми машинного навчання) потребують значної кількості статистичних даних для аналізу й побудови моделі. В контексті задач автотюнінгу збір великої кількості статистичних даних може бути доволі тривалим процесом. Тому досить гостро встає проблема алгоритмів, які дозволять максимально звузити область пошуку при мінімальній кількості реальних запусків автотюнера. Щоб частково вирішити цю проблему, в даній роботі запропоновано використовувати нейромережу для екстраполяції даних (див. розділ 2). В такому випадку необхідна буде відносно невелика кількість реальних запусків для побудови наближеної моделі, після чого нейромережева модель може бути використана іншими алгоритмами за принципом "чорної скрині".

Автотюнери, що використовують методи машинного навчання, досліджуються, зокрема, в роботах [14–16]. Так, в [14] розглядається компілятор з відкритим кодом Milepost GCC, який самоналаштовується і застосовує машинне навчання для передбачення оптимальних установок флагів компіляції програм при використанні GCC. У роботі [15] нейромережі використовуються для отримання досвіду про задану трансформацію програми (параметричне розбиття циклу на блоки) для різних значень вхідного параметра (розміру блоку); отримана модель далі використовується для пошуку оптимальних значень параметрів. В [16]

машинне навчання застосовується на автоматичної оптимізації розбиття на задачі для мови OpenCL для різних вхідних розмірів задач та різних неоднорідних архітектур, що містять багатоядерні центральні процесори та графічні прискорювачі. В даній статті нейромережі використовуються для отримання досвіду на основі результатів циклів тюнінгу (час виконання програм при різних значеннях внутрішніх параметрів програми) з подальшою заміною частини запусків автотюнера на оцінку зі статистичної моделі.

2. Практичний експеримент

В основу процесу розробки програм у даній роботі покладено використання модифікованих систем алгоритмічних алгебр (САА-М), інструментарію проектування й синтезу програм (ІПС) [4, 8] та системи генерації автотюнерів TuningGenie. Інструментарій ІПС ґрунтується на методі діалогового конструювання синтаксично правильних програм, що орієнтований на виключення можливості появи синтаксичних помилок в процесі побудови алгоритму. Основна ідея методу полягає в порівневому конструюванні алгоритмів зверху вниз шляхом суперпозиції мовних конструкцій САА-М. Алгоритми в САА-М можуть бути подані в трьох формах: аналітичній (регулярні схеми), природньо-лінгвістичній (САА-схеми) та графовій (граф-схеми). На основі побудованої схеми алгоритму виконується автоматична генерація тексту програми цільовою мовою програмування (Java або C++). Далі програма подається на вхід системі TuningGenie для подальшого налаштування та перетворення.

Як приклад алгоритму для автотюнінгу розглянемо гібридний алгоритм сортування `PARALLEL HYBRID SORT`, який виконує сортування злиттям чи вставкою в залежності від розміру блоку вхідного числового масиву з даними. Далі наведено початкову САА-схему цього алгоритму, яка складається з опису класів `ParallelMergeSort` та `MergeSortTask`. Назви методів класів від реалізації методів відділені за допомогою ланцюжків символів "`====`". У методі `parallelMergeSort(array)` класу `ParallelMergeSort` прагма `tuneAbleParam` задає діапазон значень для кількості паралельних потоків (змінної `parallelism`), тобто рівня паралелізму. В методі `compute` класу `MergeSortTask` вказано прагми `tuneAbleParam`, які задають область пошуку оптимальних значень числових змінних `insertionSortThreshold` (розмір блоку, при якому застосовується сортування вставками) та `mergeSortBucketSize` (порогове значення розміру блоку для послідовного сортування в межах одного потоку). За допомогою системи ІПС на основі САА-схеми було виконано генерацію багатопотокової програми сортування мовою Java.

```
SCHEME PARALLEL HYBRID SORT ====

CLASS ParallelMergeSort

     CLASS METHODS

     "insertionSort(array)"
     ==== "Declare a variable (n) of type (int)";
          (n := "Array length (array)");
          FOR (i FROM 1 TO n - 1)
          LOOP
            "Declare a variable (j) of type (int) = (i)";
            "Declare a variable (B) of type (int) = (array[i])";
            WHILE (j > 0) AND (array[j - 1] > B)
            LOOP
               (array[j] := array[j - 1]);
               (j := j - 1);
            END OF LOOP;
            (array[j] := B);
          END OF LOOP;

     "parallelMergeSort(array)"
     ==== "Comment (tuneAbleParam name=parallelism start=1 stop=8 step=1)";
          "Declare a variable (parallelism) of type (int) = (1)";
          PARALLEL(i = 0, ..., parallelism - 1)
          (
             "MergeSortTask(array)";
          );
          WAIT 'All threads completed work';

     "sequentialMergeSort(array)"
     ==== IF 'Length of array (array) is greater or equal to (2)'
```

```
        THEN "Declare an array (left) of type (int)";
             "Declare an array (right) of type (int)";
             (left := "Copy from array (array) the elements in the range
                       from (0) to (array.length / 2)");
             (right := "Copy from array (array) the elements in the range
                        from (array.length / 2) to (array.length)");
             "sequentialMergeSort(left)";
             "sequentialMergeSort(right)";
             "merge(left, right, array)";
        END IF;

    "merge(left, right, array)"
    ==== "Declare a variable (i1) of type (int) = (0)";
         "Declare a variable (i2) of type (int) = (0)";
         FOR (i FROM 0 TO array.length - 1)
         LOOP
           IF (i2 >= right.length) OR
              ((i1 < left.length) AND (left[i1] < right[i2]))
           THEN (array[i] := left[i1]);
                "Increase (i1) by (1)";
           ELSE (array[i] := right[i2]);
                "Increase (i2) by (1)";
           END IF
         END OF LOOP;

    CLASS MergeSortTask EXTENDS RecursiveAction

         CLASS FIELDS

         "Declare an array (array) of type (int)";

         CLASS METHODS

        "MergeSortTask(array)"
         ==== (this.array := array);

         "compute"
         ==== "Comment (tuneAbleParam name=insertionSortThreshold
                        start=10 stop=200 step=10)";
              "Declare a variable (insertionSortThreshold) of
               type (int) = (100)";

              "Comment (tuneAbleParam name=mergeSortBucketSize
                        start=10000 stop=100000000 step=10000)";
              "Declare a variable (mergeSortBucketSize)
               of type (int) = (50000)";

              IF 'Length of array (array) is less or equal to
                  (insertionSortThreshold)'
              THEN "insertionSort(array)"
              ELSE IF 'Length of array (array) is less or equal to
                       (mergeSortBucketSize)'
                   THEN "sequentialMergeSort(array)";
                   ELSE
                       "Declare an array (left) of type (int)";
                       "Declare an array (right) of type (int)";
                       (left := "Copy from array (array) the elements in
                                 the range from (0) to (array.length/2)");
                       (right := "Copy from array (array) the elements in
                                  the range from (array.length/2) to
                                  (array.length)");
                       |"MergeSortTask(left)"
                        PARALLEL
                        "MergeSortTask(right)"|;
```

```
                    "merge(left, right, array)";
                END IF
            END IF;

    END OF CLASS MergeSortTask
END OF CLASS ParallelMergeSort
END OF SCHEME PARALLEL HYBRID SORT
```

В рамках даного експерименту виконувалось сортування набору з 2×10^7 випадкових чисел. Параметрами автотюнера є $C = \{T_{cn}, T_s, T_h\}$, де T_{cn} – кількість потоків, що виконують обчислення одночасно; T_s – порогове значення розміру блоку для послідовного сортування в межах одного потоку (блоки розміром $size > T_s$ розбиваються на менші блоки і призначаються різним потокам); T_h – розмір блоку, при якому застосовується сортування вставками. Експеримент виконувався в обчислювальному середовищі з такими параметрами: 4-ядерний процесор Intel Core i7, частота 2.7 ГГц, кеш L3, 8 Мбайт; оперативна пам'ять 16 Гбайт, частота 2133 МГц; Apple SSD SM0512L, 512 Гбайт; операційна система MacOS 10.12.

Експеримент складався з двох фаз. В першій фазі автотюнер виконувався без статистичної моделі для оцінки швидкодії алгоритму. У другій фазі було підключене статистичне моделювання для розуміння того, наскільки суттєво може бути скорочений простір пошуку при збереженні продуктивності алгоритму, близької до оптимальної.

Результати першої фази наведено у таблиці. Зазначено три конфігурації:

- *повільна* – конфігурація "за замовчуванням", яка працює майже як класичне сортування злиттям;
- *оптимальна* – найшвидша, що була автоматично обрана автотюнером;
- *інтуїтивна* – значення заповнюються інтуїтивно з урахуванням специфікацій апаратного забезпечення та деталей алгоритмів.

Оптимальна конфігурація в 4.93 рази швидша за *повільну*. Цей результат є досить гарним для 4-ядерного процесора і досягнутий передусім завдяки комбінації двох факторів: оптимального використання кешів процесора (за рахунок переходу до сортування без додаткової пам'яті при малих наборах даних) та ефективної схеми розпаралелювання (сортування злиттям легко розпаралелюється за методом "розділяй та володарюй"). *Інтуїтивна* конфігурація в 3.1 рази швидша за *повільну*, що також є непоганим результатом, але цей варіант є легким для передбачення завдяки відносній простоті тестованого алгоритму. Зазвичай оптимальні конфігурації не є очевидними для реальних паралельних застосувань. *Оптимальна* конфігурація все ж таки є значно швидшою – на 58 %, і, таким чином, застосування автотюнінгу було доцільним.

Таблиця. Результати першої фази автотюнінгу

Конфігурація	***Повільна***	***Оптимальна***	***Інтуїтивна***
Рівень паралелізму T_{cn} (кількість потоків)	1	8	4
Порогове значення для сортування вставками T_h	0 (не переходити до сортування вставками зовсім)	120	30
Порогове значення для послідовного сортування T_s	100 000 000 (це значення є більшим за розмір тестових даних, тому декомпозиція даних не застосовується)	50 000	10 000
Розмір тестових даних	20 000 000 цілих чисел		
Середній час сортування	4432 мс	898 мс	1426 мс

Тепер перейдемо до розгляду другої фази, для того щоб з'ясувати, як простір пошуку автотюнера може бути скорочений за допомогою методів статистичного аналізу. Параметр T_s був виключений з моделі на початковій фазі аналізу через його незначний вплив на загальну продуктивність: як тільки кількість підзадач після декомпозиції вхідних даних є вдвічі більшою ніж рівень паралелізму, майже немає різниці, яке значення

використовується. Це можна пояснити високою ефективністю механізму *RecursiveAction* мови Java [17], використаного у програмній реалізації.

Первинний аналіз даних виконувався за допомогою мови Python із використанням бібліотеки Scikit-learn [18]. Подальший аналіз виконувався за допомогою мови R, призначеної для програмування статистичних обчислень, аналізу та подання даних у графічному вигляді [19]. Експеримент складався з декількох стандартних етапів: підготовка та завантаження записів роботи автотюнера, підготовка даних (в тому числі нормалізація), побудова нейромережевої моделі на навчальній виборці даних та перевірка моделі на контрольній виборці даних.

Процес аналізу даних показано на рис. 1. Спочатку автотюнер проводить N експериментів та зберігає дані для нейромережі в окремий файл. Ці дані використовуються нейромережею для навчання. Після навчання нейромережа екстраполює дані, генеруючи новий набір даних (в окремий файл). В кінці обидва набора даних аналізуються й порівнюються вже людиною. Як нейромережу обрано багатошаровий перцептрон з трьома вхідними нейронами, трьома прихованими шарами (по 20-10-5 нейронів відповідно) та одним вихідним. Як активатор у нейронів використовувалась випрямлена лінійна функція ($f(x) = \max(0, x)$). Як метод навчання використовувався метод зворотного поширення помилки та алгоритм Бройдена – Флетчера – Гольдфарба – Шанно [20] для оптимізації вагових коефіцієнтів.

На основі результатів 3300 запусків побудовано нейромережу, яку використано для подальшої генерації даних. В контексті даного експерименту використання нейромережі для початкового наближення дозволило звузити область пошуку на 58 % (з 10^6 до 4.2×10^5). Для перевірки якості отриманих результатів було зроблено більше 30 тисяч реальних запусків автотюнера (рівномірно розподілених по всій множині варіантів).

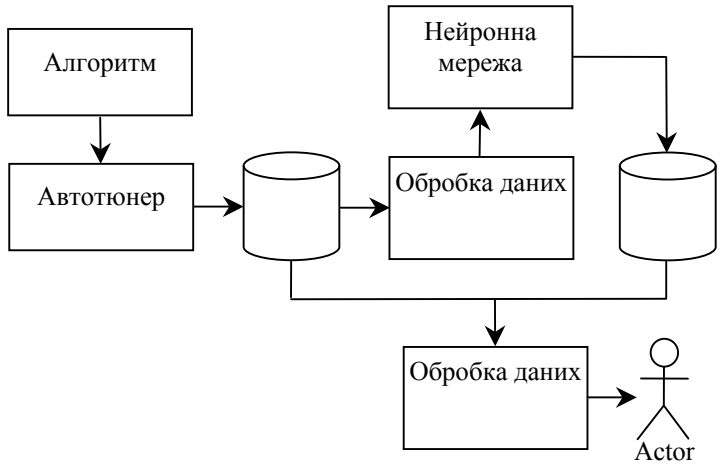

Рис. 1. Процес аналізу

На рис. 2 показано залежність середньої точності Acc отриманої моделі для 10 нейромереж від частки даних, яку було використано для навчання. Оцінка точності Acc ґрунтується на використанні матриці похибок [21] та обчислюється за формулою $Acc = \frac{TP + TN}{P + N}$, де TP – кількість позитивних результатів; TN – кількість негативних результатів; P – дзеркальна кількість позитивних значень; N – загальна кількість негативних значень.

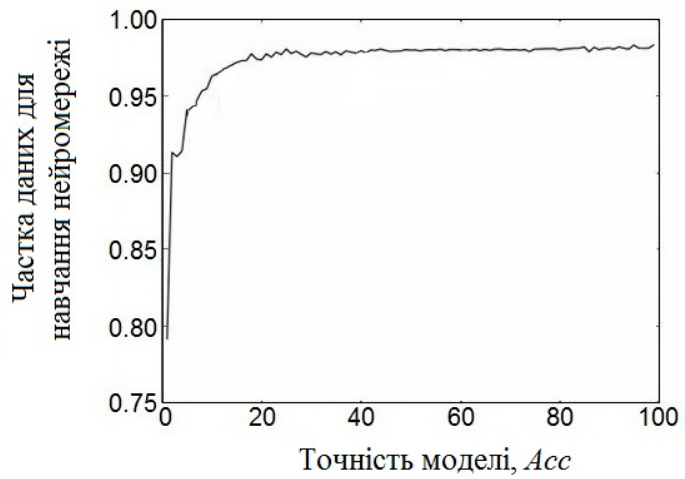

Рис. 2. Залежність точності моделі від частки даних, використаних для навчання нейромережі

З отриманих результатів можна зробити висновок, що при вирішенні задачі про звуження області пошуку для автотюнера системі для отримання досить точних результатів потрібна відносно невелика кількість замірів.

Висновки

Запропоноване вдосконалення методу автотюнінгу паралельних програм на основі використання статистичного моделювання та машинного навчання. Метод дозволяє значною мірою позбутися головної слабкості автотюнінгу, а саме, суттєво скоротити загальний час пошуку оптимального варіанту програми за рахунок автоматичного навчання нейромережевої моделі на результатах традиційних циклів тюнінгу з подальшою заміною частини запусків автотюнера оцінкою зі статистичної моделі. Застосування методу продемонстроване на задачі оптимізації гібридного паралельного алгоритму сортування чисел за допомогою розробленої інструментальної системи генерації автотюнерів TuningGenie. Результати експерименту підтвердили ефективність запропонованого методу й доцільність його подальшого розвитку та використання для оптимізації більш складних паралельних програм та програм з інших предметних областей.

Література

1. Naono K., Teranishi K., Cavazos J., Suda R. Software automatic tuning: from concepts to state-of-the-art results. Berlin: Springer, 2010. 377 p.
2. Durillo J., Fahringer T. From single- to multi-objective auto-tuning of programs: advantages and implications. *Scientific programming*. 2014. Vol. 22, N 4. P. 285–297.
3. Doroshenko A., Shevchenko R. A rewriting framework for rule-based programming dynamic applications. *Fundamenta Informaticae*. 2006. Vol. 72. N 1–3. P. 95–108.
4. Яценко Е.А. Интеграция инструментальных средств алгебры алгоритмов и переписывания термов для разработки эффективных параллельных программ. *Проблеми програмування.* 2013. № 2. С. 62–70.
5. Иваненко П.А., Дорошенко А.Е. Метод автоматической генерации автотьюнеров для параллельных программ. *Кибернетика и системный анализ.* 2014. № 3. С. 161–173.
6. Ivanenko P., Doroshenko A., Zhereb K. TuningGenie: auto-tuning framework based on rewriting rules. Proc. 10th International Conference "ICT in Education, Research, and Industrial Applications" (ICTERI 2014), Revised Selected Papers, Kherson, Ukraine (9–12 June 2014). Berlin: Springer, 2014. Vol. 469. P. 139–158.
7. Дорошенко А.Ю., Іваненко П.А., Новак О.С. Гібридна модель автотюнінгу з використанням статистичного моделювання. *Проблеми програмування.* 2016. № 4. С. 27–32.
8. Андон Ф.И., Дорошенко А.Е., Жереб К.А., Шевченко Р.С., Яценко Е.А. Методы алгебраического программирования: формальные методы разработки параллельных программ. К.: Наукова думка, 2017. 440 с.
9. Whaley R., Petitet A., Dongarra J.J. Automated empirical optimizations of software and the ATLAS Project. *Parallel computing*. 2001. Vol. 27, N 1–2. P. 3–35.
10. Frigo M., Johnson S. FFTW: an adaptive software architecture for the FF. *Acoustics, speech and signal processing.* 1998. Vol. 3. P. 1381–1384.
11. Schaefer C.A., Pankratius V., Tichy W.F. Atune-IL: an instrumentation language for auto-tuning parallel applications. Proc. 15th International Euro-Par Conference (Euro-Par 2009), Delft, The Netherlands (25–28 August 2009). LNCS. 2009. Vol. 5704. P. 9–20.
12. Mitchell T.M. Machine learning. 1st edn. New York: McGraw-Hill Education, 1997. 432 p.
13. Givens G.H.,ʃHoeting J.A. Computational statistics. 2nd edn. Chichester: Wiley, 2012. 496 p.
14. Fursin G., Kashnikov Y., Memon A.W., Chamski Z. et al. Milepost GCC: machine learning enabled self-tuning compiler. *International journal of parallel programming.* 2011. Vol. 39, N 3. P. 296–327.
15. Rahman M., Pouchet L.-N., Sadayappan P. Neural network assisted tile size selection. Proc. 5th International Workshop on Automatic Performance Tuning (IWAPT'2010), USA, Berkeley, CA (22 June 2010). Berkeley, CA: Springer, 2010.
16. Kofler K., Grasso I., Cosenza B., Fahringer T. An automatic input-sensitive approach for heterogeneous task partitioning. Proc. 27th ACM International Conference on Supercomputing (ICS'13), USA, Eugene, Oregon (10–14 June 2013). New York: ACM, 2013. P. 149–160.

17. Oracle Help Center. Class RecursiveAction (Java SE 9 & JDK 9). [Електронний ресурс]. Режим доступу до ресурсу: https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/RecursiveAction.html (дата звернення: 24.01.2018)
18. Pedregosa F., Varoquaux G., Gramfort A., Michel V. et al. Scikit-learn: machine learning in Python. *Journal of machine learning research.* 2011. Vol. 12. P. 2825–2830.
19. Crawley M.J. The R book. 2nd edn. Chichester: Wiley, 2012. 1076 p.
20. Fletcher R. Practical methods of optimization. 2nd edn. Chichester: Wiley, 2000. 456 p.
21. Fawcett T. An introduction to ROC analysis. *Pattern recognition letters*. 2006. Vol. 27, N 8. P. 861–874.

References

1. Naono, K., Teranishi, K., Cavazos, J. & Suda, R. (2010) *Software automatic tuning: from concepts to state-of-the-art results*. Berlin: Springer.
2. Durillo, J. & Fahringer, T. (2014) From single- to multi-objective auto-tuning of programs: advantages and implications. *Scientific programming*. 22 (4). P. 285–297.
3. Doroshenko, A. & Shevchenko, R. (2006) A rewriting framework for rule-based programming dynamic applications. *Fundamenta informaticae*. 72 (1-3). P. 95–108.
4. Yatsenko, O.A. (2013) Integration of tools of algebra of algorithms and term rewriting system for developing efficient parallel programs. *Problems in programming.* (2). P. 62–70. (in Russian)
5. Ivanenko, P.A. & Doroshenko, A.Yu. (2014) Method of automated generation of autotuners for parallel programs. *Cybernetics and systems analysis*. 50 (3). P. 465–475.
6. Ivanenko, P., Doroshenko, A. & Zhereb, K. (2014) TuningGenie: auto-tuning framework based on rewriting rules. In *Proc. 10th International Conference "ICT in Education, Research, and Industrial Applications" (ICTERI 2014), Revised Selected Papers*. Kherson, Ukraine, 9-12 June 2014. Berlin: Springer. 469. P. 139–158.
7. Doroshenko, A.Yu., Ivanenko, P.A. & Novak, O.S. (2016) Hybrid autotuning model with statistic modelling. *Problems in programming*. (4). P. 27–32. (in Ukrainian)
8. Andon, P.I., Doroshenko, A.Yu., Zhereb, K.A., Shevchenko, R.S. & Yatsenko, O.A. (2017) *Methods of algebraic programming: formal methods of parallel program development*. Kyiv: Naukova dumka. (in Ukrainian)
9. Whaley, R., Petitet, A. & Dongarra, J.J. (2001) Automated empirical optimizations of software and the ATLAS Project. *Parallel computing*. 27 (1-2). P. 3–35.
10. Frigo, M. & Johnson, S. (1998) FFTW: an adaptive software architecture for the FF. *Acoustics, speech and signal processing*. 3. pp. 1381-1384.
11. Schaefer, C.A., Pankratius, V. & Tichy, W.F. (2009) Atune-IL: an instrumentation language for auto-tuning parallel applications. In *Proc. 15th International Euro-Par Conference (Euro-Par 2009)*. Delft, The Netherlands, 25-28 August 2009. LNCS. 5704. P. 9–20.
12. Mitchell, T.M. (1997) *Machine learning*. 1st edn. New York: McGraw-Hill Education.
13. Givens, G.H. &ՙHoeting, J.A. (2012) *Computational statistics*. 2nd edn. Chichester: Wiley.
14. Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, Z. et al. (2011) Milepost GCC: machine learning enabled self-tuning compiler. *International journal of parallel programming*. 39 (3). P. 296–327.
15. Rahman, M., Pouchet, L.-N. & Sadayappan, P. (2010) Neural network assisted tile size selection. In *Proc. 5th International Workshop on Automatic Performance Tuning (IWAPT'2010)*. USA, Berkeley, CA, 22 June 2010. Berkeley, CA: Springer.
16. Kofler, K., Grasso, I., Cosenza, B. & Fahringer, T. (2013) An automatic input-sensitive approach for heterogeneous task partitioning. In *Proc. 27th ACM International Conference on Supercomputing (ICS'13)*. USA, Eugene, Oregon, 10-14 June 2013. New York: ACM. P. 149–160.
17. ORACLE HELP CENTER. (2018) *Class RecursiveAction (Java SE 9 & JDK 9)* [Online]. Available from: https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/RecursiveAction.html [Accessed: 24 January 2018]
18. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V. et al. (2011) Scikit-learn: machine learning in Python. *Journal of machine learning research.* 12. P. 2825–2830.
19. Crawley, M.J. (2012) *The R book*. 2nd edn. Chichester: Wiley.
20. Fletcher, R. (2000) *Practical methods of optimization*. 2nd edn. Chichester: Wiley.
21. Fawcett, T. (2006) An introduction to ROC analysis. *Pattern recognition letters*. 27 (8). P. 861–874.

***Про авторів*:**

Дорошенко Анатолій Юхимович,
доктор фізико-математичних наук,
професор, завідувач відділу теорії комп'ютерних обчислень
Інституту програмних систем НАН України,
професор кафедри автоматики та управління в технічних системах
НТУ України "КПІ імені Ігоря Сікорського".
Кількість наукових публікацій в українських виданнях – понад 150.
Кількість наукових публікацій в зарубіжних виданнях – понад 50.
Індекс Хірша – 5.
http://orcid.org/0000-0002-8435-1451,

Іваненко Павло Андрійович,
молодший науковий співробітник Інституту програмних систем НАН України.
Кількість наукових публікацій в українських виданнях – 19.
Кількість наукових публікацій в зарубіжних виданнях – 4.
http://orcid.org/0000-0001-5437-9763,

Новак Олександр Сергійович,
аспірант Інституту програмних систем НАН України.
Кількість наукових публікацій в українських виданнях – 4.
http://orcid.org/0000-0002-1665-7360,

Яценко Олена Анатоліївна,
кандидат фізико-математичних наук,
старший науковий співробітник Інституту програмних систем НАН України.
Кількість наукових публікацій в українських виданнях – 37.
Кількість наукових публікацій в зарубіжних виданнях – 12.
http://orcid.org/0000-0002-4700-6704.

Місце роботи авторів:

Інститут програмних систем Національної академії наук України.
03187, м. Київ-187, проспект Академіка Глушкова, 40, корпус 5.
Тел.: +380 (44) 526 3559.
Факс +380 (44) 526 6263.
E-mail: doroshenkoanatoliy2@gmail.com,
paiv@ukr.net,
novak.as@i.ua,
oayat@ukr.net