УДК 519.854.2

КОСТИК Д. Ю.

ОПТИМІЗАЦІЯ ПРОГРАММНОЇ РЕАЛІЗАЦІЇ МОДИФІКОВАНОГО ПДС-АЛГОРИТМУ ЗАДАЧІ МІНІМІЗАЦІЇ СУМАРНОГО ЗАПІЗНЕННЯ ВИКОНАННЯ ЗАВДАНЬ

Наводиться порівняння програмних реалізацій різних версій, дається опис оптимізації програмної реалізації ПДС-алгоритму задачі мінімізації сумарного запізнення. На прикладах проілюстровано дві проблеми: неоптимальне використання оперативної пам'яті та обчислення тільки за допомогою одного ядра на багатоядерних комп'ютерах.

A comparison of different versions of software implementations, describes the software optimization of PDS algorithm to minimize total tardiness. The examples illustrate two problems: non optimal usage of memory and calculation using only one core on multicore computers.

1. Вступ

На практичну ефективність алгоритму впливають дві складові, а саме: ефективність математичного алгоритму та ефективність програмної реалізації. Аналізуючи обидві складові можемо отримати по-справжньому ефективний алгоритм та його програмну реалізацію. Найбільш проблемні місця у будь якого програмного продукту, який реалізує певний математичний алгоритм, це: витік пам'яті, неоптимальність простих операцій (впорядкування, вставка, і т.д.), занадто часте використання постійної пам'яті, використання тільки одного потоку або одного процесору, дуже часті рекурсивні виклики, мале навантаження на центральний процесор, невикористання графічного процесору для математичних операцій. У статті розглядається програмна реалізація алгоритму, яка біла наведена у [2].

2. «Мінімізація сумарного запізнення виконання незалежних завдань з директивними строками одним приладом» (МСЗ)

Припустимо, що задано множину незалежних завдань $J = \{j_1, j_2, \ldots, j_n\}$, кожне з яких складається з однієї операції. Для кожного завдання відома тривалість виконання l_j і директивний строк виконання D_j. Завдання надходять у систему одночасно в момент $d_j = 0, j = \overline{1, n}$. Переривання не допускаються. Необхідно побудувати розклад виконання завдань для одного приладу, що мінімізує сумарне запізнення при виконанні завдань:

$$f = \sum_{j=1}^{n} \max\left(0, C_j - D_j\right),$$

де C_j – момент завершення виконання завдання j.

Огляд відомих методів розв'язання цієї задачі приведений у [1]. Згідно Ду і Люнгу, задача є NP-складною.

На кафедрі АСОІУ було запропоновано ефективний точний ПДС-алгоритм (алгоритм із поліноміальною й експоненційною складовими) розв'язання задачі, заснований на новому підході до розв'язання задач з директивними строками, що полягає в оптимальному використанні резервів часу незапізнених завдань, що дозволяє розв'язувати задачі з числом завдань, істотно більшим, ніж 500.

3. Ефективність програмної реалізації

Найбільш проблемні місця у будь якого програмного продукту, який реалізує певний математичний алгоритм, це:

- витік пам'яті;
- неоптимальність простих операцій (впорядкування, вставка, і т.д.);
- занадто часте використання постійної пам'яті;
- використання тільки одного потоку або одного процесору;
- дуже часті рекурсивні виклики;
- мале навантаження на центральний процесор;
- невикористання графічного процесору для математичних операцій.

Якщо розглянути всі алгоритми в сукупності, то їх можна поділити на дві категорії:

— ті, що використовують переважно пам’ять;

— ті, що використовують переважно процесор.

Щодо модифікованого ПДС-алгоритму, то його перша реалізація використовувала переважно процесор. Розмір всіх даних у оперативній пам’яті ледве сягав 20 мегабайт. Ще однією неприємністю було використання лише одного процесора – це пов’язано з структурою самого алгоритму. Математичний алгоритм є послідовним у всіх його місцях, тобто не містить блоків для розгалуження.

На рис. 1 показано навантаження на ЦП, та розмір оперативної пам’яті.

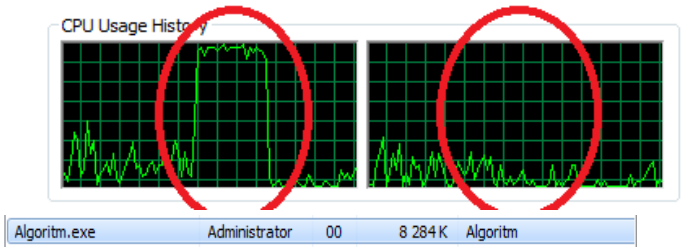

Рис. 1. Навантаження на ЦП та розмір ОП

Більшість персональних комп’ютерів сьогодні є багатоядерними та з підтримкою технології віртуальних ядер. Тобто навіть найдешевший ноутбук має 4-и ядра (2-а фізичних, та 2-а віртуальних). А якщо брати до уваги серверні комп’ютери, навіть одноюнітові, то в них кількість незалежних ядер може сягати 32. Тобто, запустивши алгоритм на такому комп’ютері, ми будемо використовувати лише 1/32 ресурсів, що не є оптимальним ні з точки зору використання такого комп’ютера, ні з точки зору швидкодії та часу вирішення великих прикладів. Розмір оперативної пам’яті такого комп’ютера може бути від 8 до 128 гігабайт. А як зазначалося вище алгоритм потребував лише 20 мегабайт. Тобто знову таки, маємо ситуацію таку ж як і з процесорами – ресурсів багато, а використовуємо лише маленьку частину.

Проаналізувавши ці дві проблеми, біли проведені дослідження алгоритму на можливість адаптації до розгалужування розрахунків, тобто виділення незалежних операцій в один момент часу, а також перенос навантажень з процесору на оперативну пам'ять.

Як зазначалося вище, і це проілюстровано на блок-схемі алгоритму [2], математично алгоритм не містить місць, які можна робити паралельно. Тобто всі етапи залежні один від одного та повинні виконуватися у чіткій послідовності. Так, єдиний варіант, який є можливим для розгалужування, – це впорядкування.

Продивившись та проаналізувавши детальні протоколи роботи алгоритму на задачах з різною кількістю завдань, було виявлено, що у деяких випадках, коли алгоритм занурюється у рекурсію, він робить оптимізацію на під інтервалі, на якому, можливо, вже виконувалась оптимізація минулими етапами алгоритму. Тобто є ймовірність того, що деякі дії повторюються. Це є невід’ємною рисою всіх NP-складних задач.

Було розроблено два додаткові етапи алгоритму:

— етап побудови таблиць з найкращими результатами;

— етап заміщення рекурсивних викликів вже оптимізованої послідовністю.

Таким чином отримали метод, який дозволяє значно знизити навантаження на центральний процесор, та перенести їх на оперативну пам’ять. Головна ідея цього метода в тому, що ми можемо заповнювати оперативну пам'ять тимчасовими проміжними даними. В цих даних зберігаються проміжні результати оптимальних підпослідовностей. Це дає змогу підставляти вже готові результати у випадку, коли ми вже робили такі оптимізації. Тобто, після кожного глобального циклу (одна ітерація), ми додаємо до оперативної пам’яті саму послідовність та вже отриманий оптимальний розклад. Таким чином формується таблиця з найкращими результатами для кожної підпослідовності.

Далі, коли є таблиця з кращими результатами, перед кожним рекурсивним кроком намагаємося знайти вже оптимізований розклад. Тут, замість того, щоб процесор робив обчислення рекурсивного виклику, йому достатньо знайти комірку в оперативній пам’яті, де зберігається вже оптимальний розклад. Дійсно, пошук у масиві даних, найчастіше, займає менше часу, ніж побудова оптимального розкладу за експоненційною складовою. Але продивившись протоколи отримані при розв’язані прикладів різної розмірності, виявилося, що у 24% прикладів, оптимізація займає менше часу, ніж пошук оптимального розкладу.

Наступний етап для покращення ефективності був етап розробки ефективного пошуку у таблиці тимчасових послідовностей. Таким

чином ми отримали наступну структуру для зберігання даних про тимчасові оптимальні результати:

— контрольна сума – число, яке є ідентифікатором послідовності;

— масив завдань – таблиця завдань підпослідовності;

— масив індексів початкової послідовності, мається на увазі ще не оптимізованої послідовності;

— масив індексів вже після оптимізації, тобто та таблиця, яку ми підставляємо замість рекурсивних викликів.

Після кожної ітерації зберігаємо такі відомості. Контрольна сума була введена для швидкого пошуку, щоб звести процесорне навантаження зовсім до мінімуму. При зберіганні послідовності по масиву індексів у початковій послідовності, ми рахуємо контрольну суму. А при пошуку в таблиці тимчасових результатів для послідовності, яку треба оптимізувати, ми рахуємо контрольну суму за тим самим алгоритмом (32-бітна контрольна сума), що були розраховані перед занесенням в таблицю. Таким чином, перший етап пошуку зводиться до порівняння лише одного 32 бітного числа. Другий етап залишився, тобто по елементний пошук, але він викликається тільки тоді, коли була знайдена вже невелика кількість збігань контрольних сум. Найчастіше маємо збігання один до одного.

Після аналізу ефективності нового методу і розподілу навантажень між процесором та оперативною пам’яттю, ми отримали 81% ефективності, у порівняні з частими рекурсивними викликами. Ще 10% припадають на приклади малої розмірності (до 50 завдань), тобто на ті, котрі вирішуються за прийнятний час. І які нема сенсу оптимізувати.

Одну проблему, а саме мале навантаження на оперативну пам’ять, вирішили. На рис. Рис. ***2*** показано завантаженість оперативної пам’яті комп’ютера.

Algoritm.vshost.exe	Administrator	50	201 084 K	vshost.exe

Рис. 2. Розмір ОП

На рис. 3 показано завантаженість оперативної пам’яті комп’ютера. Тобто, як можна переконатися з графіку, найбільшу частину пам’яті займає саме етап алгоритму 14.3, в якому відбувається запис до тимчасових таблиць.

Name	Bytes %
PDS.Core.PDS.nStep14_3(int32,int32,int32)	94,17
PDS.Core.PDS.PDSRec(int32,int32,int32,int32)	2,50
PDS.Core.PDS.FreeExchange(int32,int32)	2,41
System.BitConverter.GetBytes(int32)	0,73

Рис. 3. Розподіл пам’яті між етапами алгоритму

На рис. 4 показано, як розподіляється пам’ять між типами даних.

Name	Bytes %
Algoritm.Interface.Record[]	95,02
System.Boolean[]	2,41
System.Byte[]	2,38
Algoritm.Interface.AlgoritmBase.Comparer_L	0,14
PDS.Core.PDSRecResult[]	0,04

Рис. 4. Розподіл пам’яті між типами даних

На рис. 5 показано, як розподіляється звертання до пам’яті між типами даних.

Name	Instances %
System.Byte[]	50,22
Algoritm.Interface.Record[]	25,30
System.Boolean[]	18,67
Algoritm.Interface.AlgoritmBase.Comparer_L	5,78
System.String	0,01

Рис. 5. Розподіл звертань пам’яті між типами даних

На вищенаведених рисунках можна побачити, що найбільш вимогливими до оперативної пам’яті комп’ютера є тип даних Algoritm.Interface.Record[], в ньому зберігаються всі тимчасові таблиці з проміжними оптимальними результатами підпослідовностей. Але найчастіше пам'ять використовує тип даних Syste.Byte[], в ньому зберігаються індекси тимчасових таблиць, по яким відбувається пошук.

Для оптимізації алгоритму було використано дуже потужний інструмент для розробників програмного забезпечення Microsoft Performance Profiler, який входить до складу останньої версії Microsoft Visual Studio 2010. З його допомогою можна проаналізувати «вузькі» місця алгоритму, продивитися ієрархію викликів методів.

Як зазначалося вище, вузьким місцем алгоритму є навантаження тільки на онде ядро процесора (рис. 6).

При досліджені алгоритму на прикладах різної розмірності було отримані наступні дані (рис. 7, 8).

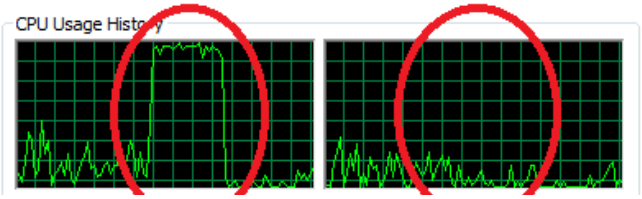

Рис. 6. Навантаження на ЦП

Function Name	Inclusive Samples %	Exclusive Samples %
PDS.Core.PDS.PDSRec(uint32,uint32,int32,int32)	21,92	0,00
PDS.Core.PDS.nStep14(int32,int32&,int32,int32)	8,22	2,05
PDS.Core.PDS.nStep14_3(int32,int32,int32)	6,16	0,00
PDS.Core.PDS.PDSRec(uint32,uint32,int32,int32)	5,48	0,00
PDS.Core.PDS.FreeExchange(uint32,uint32)	**4,11**	**4,11**

Рис. 7. Найбільш ресурсоємні функції

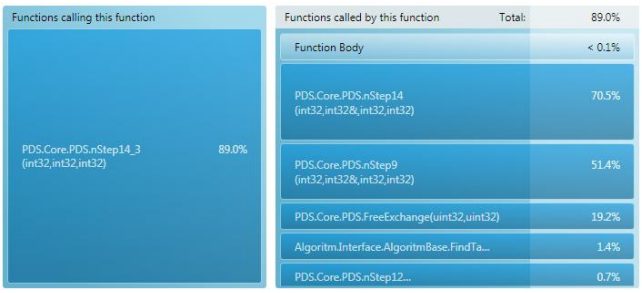

Рис. 8. Найбільш ресурсоємні функції

З протоколу можна побачити, що функцією, яка найбільш часто викликається, є nStep14_3, вона є точкою входу до рекурсивного виклику. Тобто саме з цього місця починається експоненційна складова алгоритму. Проаналізувавши багато прикладів різної розмірності, та той факт, що треба розгалужувати алгоритм, було запропоновано та реалізовано алгоритм тимчасових таблиць. Після впровадження алгоритму вдалося знизити навантаження на експоненційну складову алгоритму з 70% до 39%. Але проблема розгалужування ще не була вирішена.

Єдине місце, де це можливо було зробити, є пошук по тимчасовим таблицям. Так ми можемо перекласти частину навантаження за рахунок хоча б 11% пошуку по тимчасовим таблицям.

Для паралельного програмування була використана бібліотека PLINQ. Результати другої версії алгоритму наведені на рис. 9–10.

Function Name	Inclusive Samples %	Exclusive Samples %
PDS.Core.PDS.PDSRec(uint32,uint32,int32,int32)	77,23	0,00
PDS.Core.PDS.nStep14(int32,int32&,int32,int32)	34,65	3,96
PDS.Core.PDS.nStep14_3(int32,int32,int32)	26,73	0,00
PDS.Core.PDS.PDSRec(uint32,uint32,int32,int32)	25,74	0,99
System.Linq.ParallelEnumerable.ToList(class System.Linq.ParallelQuery`1...	**6,93**	**6,93**

Рис. 9. Найбільш ресурсоємні функції

З протоколів видно, що за рахунок паралельного пошуку нам вдалося знизити рекурсивні виклики на 5 відсотків (з 39% до 34%), та «перекласти» частину навантаження на 2-е ядро центрального процесора (рис. 11).

Name	Exclusive Samples %
PDS.Core.PDS.FreeExchange(uint32,uint32)	42,57
System.Linq.ParallelEnumerable.ToList(class System.Linq.ParallelQuery`1<!0>)	23,76
[clr.dll]	9,90
PDS.Core.PDS.nStep14(int32,int32&,int32,int32)	5,94
PDS.Core.Crc32..ctor()	2,97

Рис. 10. Найбільш ресурсоємні функції за індивідуальними викликами

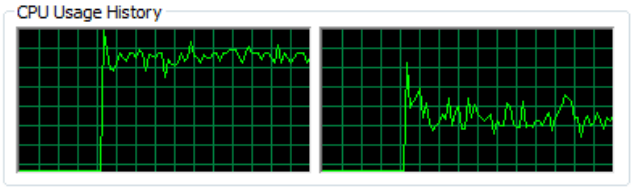

Рис. 11. Навантаження на ЦП

Такий алгоритм для пошуку проміжних результатів по тимчасовим таблицям залишимо у версії для прикладів з кількістю завдань до 200, та для задач з кількістю завдань більше 200 запропоновано дворівневий пошук. Це нагадує індексний пошук в сучасних реляційних базах даних, але є деякі відмінності, специфічні тільки для нашого випадку.

В якості індексу для пошуку було обрано контрольну суму, яка у нас все одно рахується для кожної послідовності. Але на відміну від першої реалізації, де ми використовували 16 біт для зберігання контрольної суми, в третій версії алгоритму було прийнято рішення використовувати 8 біт для зберігання контрольної суми. У цього рішення є як позитивні, так і негативні риси. До позитивних рис можна віднести:

— зменшення математичних операцій при підрахунку контрольної суми;

— менший об’єм оперативної пам’яті;

— прискорення пошуку (перебір 256 комірок, замість 65536).

До негативних рис відноситься:

— не унікальність контрольної суми;

— дворівневий пошуку.

Після визначення індексу пошук перетворився на дворівневий, за умов того, що ми використовуємо 8 біт для зберігання контрольної суми, а тимчасових таблиць, розмірність яких вище 200, більше ніж 256 можливих значень.

Першим кроком є пошук всіх однакових контрольних сум, які відповідають тимчасовим послідовностям. Ділі, коли ми вже отримали набір таблиць з однаковими контрольними сумами, серед них шукаємо необхідну нам таблицю, якщо така є.

У іншому алгоритму пошук заміни та вставки, такі самі, як і в першій його версії.

Результати, які вдалося отримати за рахунок нової оптимізації, наведені у протоколі рис. 12 – 13.

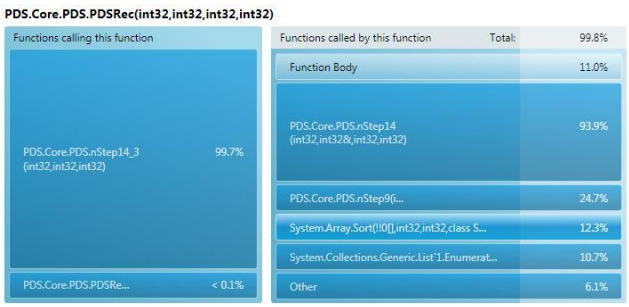

Рис. 12. Найбільш ресурсоємні функції

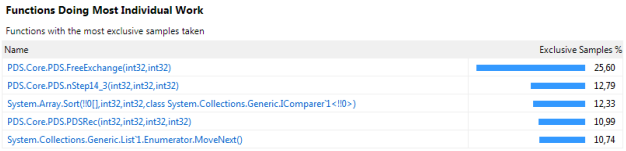

Рис. 13. Найбільш ресурсоємні функції за індивідуальними викликами

З протоколів видно, що за рахунок нової версії напошуку вдалося покращити показники швидкодії на 11 відсотків (з 23% до 12%).

8. Висновок

Показана ефективність модифікацій до ПДС-алгоритму. Показана різниця в розподілах оперативної пам'яті комп'ютера, та навантаження на центральний процесор та розподіл задач між багатоядерними системами. Розроблено методику для дослідження ефективності програмної реалізації алгоритму. Проведені експериментальні дослідження розв'язування генерованих задач за допомогою програмної реалізації ПДС-алгоритму, інтегрованої до розробленої системи моделювання, дозволили зібрати значиму статистичну інформацію як про класи вхідних задач, так і про активність використання окремих процедур алгоритму та провести оптимізацію програмної реалізації ПДС алгоритму.

Перелік посилань

1. Павлов А.А., Теленик С.Ф. Информационные технологии и алгоритмизация в управлении.– К.: Техника.– 2002.– 344 с., ISBN 966-575-045-3.
2. Система моделювання для дослідження ефективності ПДС-алгоритму задачі мінімізації сумарного запізнення виконання завдань // Павлов А.А., Місюра О.Б., Халус О.А, Беньковський С.Б., Лисецький Т.М., Костик Д.Ю. / Вісник НТУУ "КПІ". Інформатика, управління та обчислювальна техніка. К.: "ВЕК+", 2007.– №47.– С.215-220
3. Павлов О.А., Місюра О.Б., Халус О.А. Модифікований ефективний ПДС-алгоритм рішення задачі мінімізації сумарного запізнення виконання незалежних завдань одним приладом / Радиоэлектроника и информатика.– Харьков: ХНУР, 2007.– №1(36).– С.21-23
4. Павлов А.А., Мисюра Е.Б. Эффективный точный ПДС-алгоритм решения задачи о суммарном запаздывании для одного прибора // Системні дослідження та інформаційні технології. – 2004.– №4.– С.30-59
5. Шейко В., Кушнаренко Н. Організація та методика науково-дослідницької діяльності: Підручник. – К.: Знання-Прес, 2002. – 295 с.
6. Fisher, M.L. (1976) A dual algorithm for the one machine scheduling problem. Mathematical Programming, 11, 229-251. Тестові задачі доступні в Інтернет на сторінці http://www.bilkent.edu.tr/~bkara.