Дослідження віртуальної паралельної машини .NET Framework на прикладі алгоритму Габова.

Погорілий С.Д., Равський О.С.
Київський національний університет імені Т. Шевченка
sdp@rpd.univ.kiev.ua, reva_@mail.ru

Abstract

Pogorilyy S.D., Ravskiy O.S. the virtual parallel machine .NET Framework research by Gabov algorithm example. *The Gabov algorithm of the strong component search in a directed graph is realized. Its realization is created for architectures with common and distributed memory. The comparative analysis of these realization is resulted.*

Вступ

Проблема обробки та пошуку у великих масивах даних була та є вельми актуальною. Однією з задач при розгляді масивів даних є пошук релевантних документів в мережі веб-ресурсів для процедури ідентифікації тематичних веб-повідомлень [1]. Сама процедура грунтується на виявленні в сформованому на основі «зернових» ресурсів локальному веб-графі компоненти сильної зв'язності з використанням алгоритму пошуку сильних компонент [2-4]. У 1999 році Математик Г. Габов отримав просту реалізацію задачі пошуку сильних компонент у орієнтованих графах. Метод грунтується на послідовному обході графа модифікованим пошуком в глибину, і є одним з найоптимальніших [3].

Сьогоденний рівень розвитку систем обробки даних показує, що використання послідовних алгоритмів не є оптимальним рішенням. Створення застосувань на основі паралельної роботи декількох потоків обробки показує у більшості випадків свої переваги [6-7].

Робота присвячена дослідженню двох модифікованих версій алгоритму Габова, для систем зі спільною та розподіленою пам'яттю, на основі концепції паралелізму.

Постановка задачі

Мета роботи: Пошук сильних компонент орієнтованого графу алгоритмом Габова, в середовищі паралельної машини .NET Framework. Створення паралельної реалізації для систем із спільною та розподіленою пам'яттю, з метою прискорення ефективності роботи.

Алгоритм Габова

Алгоритм складається з таких частин:

- Init – визначення змінних, що використовуються лише для поточної ітерації і створюються заново для кожної з наступних кроків.
- Prepare – у наперед визначені стеки додається поточна вершина і для вектору, збільшується лічильник.
- ModDFSSearch (модифікований пошук в глибину) – використовуючи ітератор абстрактного типу графу, виконується рекурсивна функція пошуку.
- FreeMark – звільнення стеків, за умови досягнення кожного зворотного ребра. А також заповнення вектору, що вказую на номер сильної компоненти для кожної вершини.

В загальному випадку записується у такий спосіб:

```
ForEachNotMarkVertex (G)
    Gabov (v)
        Init
        Prepare
        ModDFSSearch (v+1)
        FreeMark

ModDFSSearch (v)
    IfNotMark
        Gabov (v)
    IfMark
        Free
    GoToNextVertex
```

Обрахунок для кожної вершини являє собою перегляд суміжних вершин. Алгоритм використовує один стек для відстеження поточного шляху пошуку. В нього додається імена вершин на вході в рекурсивну функцію. Також використовується другий стек, щоб знати, коли виштовхувати з головного стеку всі вершини кожної сильної компоненти. В нього додаються вершини, що входять в траєкторію пошуку. Коли зворотне ребро вказує, що деяка послідовність таких вершин цілком належить одній і тій самій сильній компоненті, стек звільняється, аж до вершини, на яку вказує зворотне ребро. Якщо для вершини закінчено обрахунок і вона знаходиться у верхівці другого стеку, то відомо, що всі вершини, розташовані після поточної у головному стеку знаходяться в

одній і тій же сильній компоненті. Далі, після відвідання завершального елемента кожної сильної компоненти, головний стек звільняється та присвоюється номер нової компоненти [3].

Цей метод являє собою незначну модифікацію пошуку в глибину, при цьому додані декілька перевірок, а також операцій запису та читання з оперативної пам'яті, що потребують постійних затрат часу на їх виконання. З цього слідує, що пошук сильних компонент на орграфах алгоритмом Габова виконується з лінійними затратами часу, пропорційними V^2 для представлення графу у вигляді матриці суміжності та V + E у вигляді списку суміжних вершин.

Приклад роботи алгоритму наведено на рис. 1. та 2.

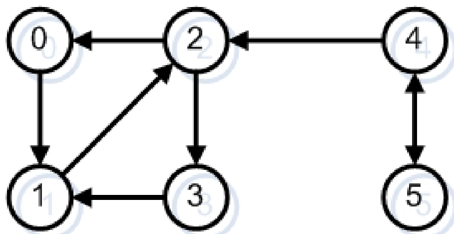

Рис 1. Приклад роботи алгоритму Габова

0-0	0	0
0-1	0 1	0
1-2	0 1 2	0
2-0	0 1 2	0
2-3	0 1 2 3	0 3
3-1	0 1 2 3	0
0		
4-4	4	4
4-2	4	4
4-5	4 5	4 5
5-4	4 5	4
4		

Рис 2. Посередині зображено трасування головного стеку, праворуч другого, ліворуч – трасування ребер

На початку роботи першою вершиною для обробки є вершина 0. Вона додається до стеків. По суміжним ребрам можна перейти до вершини 1. Далі по суміжним ребрам можна перейти до вершини 0, але вона вже позначена, тому перехід відбувається до вершини 3. Непозначених вершин не залишилось, тому при виході з рекурсії до нульової ітерації видно, що знайдену першу сильну компоненту. Потім точно такі ж кроки проходить алгоритм для вершин 4 та 5 й присвоює їм наступний номер сильної компоненти. В результаті вершини від 0 до 3 знаходяться в одній сильній компоненті з номером 0, а вершини 4, 5 у другій з номером 1.

Підхід до підвищення швидкодії

В основі алгоритму лежить модифікований пошук в глибину. По своїй структурі - це рекурсивний метод обходу графу. Оскільки, це нетривіальна рекурсія, застосування методу паралелізму за кодом неможливо. Наступнім варіантом є паралелізм за даними. Аналіз показав, що однозначно й незалежно вхідні дані не можна розбити на частини без попередньої обробки. Для виділення сильної компоненти, необхідна інформація, яку алгоритм зберігав у хронологічній послідовності своєї роботи. В залежності від структури орграфа інформація, яка була збережена в стеках на самому початку, може бути використана в будь який момент роботи алгоритму [3]. З іншого боку, як буде доведено, алгоритм зовсім виходить з рекурсивної функції, тоді й тільки тоді, коли на його шляху зустрічаються дерева. Якщо граф має декілька дерев, то за означенням, між ними немає жодного ребра, що їх з'єднує. З цього слідує, що під час роботи алгоритму, інформація про вершини одного дерева буде використана лише у тому самому дереві.

Доведення: алгоритм під час своєї роботи переходить від вершини до вершини через суміжні ребра, що зустрічаються йому на шляху. Означення дерева каже, що підграф заданого графу є деревом, якщо немає ребер, що з'єднують ці підграфи. Звідси очевидно, що почавши огляд графа в одному з його піддерев, алгоритм без повного виходу з рекурсивної функції не може перейти в інше піддерево, й не може використовувати вершини одного дерева при розгляді іншого.

Структура створеного застосування

В основу для створення застосування покладено парадигму об'єктно-орієнтованого програмування та основні його принципи: інкапсуляція**,** успадкування**,** поліморфізм.

Абстрактний тип даних графа являє собою інтерфейс, що надає базові механізми, для представлення графу у пам'яті.

```
public struct EDGE
{
    Int16 v, w;
    EDGE(Int16 v, Int16 w){};
}
public class Graph
{
    protected Int16 Vcnt;
    protected Int32 Ecnt;
    protected bool digraph;

    public Graph(Int16 V, bool digraph);
    public Int16 V ();
    public Int32 E ();
    public bool directed ();
    public randE (Int32 E);
    public Int32 randG (Int32 E);
    public Int32 randSC (Int16 SCcnt, bool build_sc);
    public bool edge (Int16 v, Int16 w);
    public insert (EDGE e) ;
    public remove (EDGE e);
    protected class GraphEnumerator: iGraphEnumerator
```

```
    {
        protected Int16 i, v;
        protected Graph G;
        public GraphEnumerator(Graph G, Int16 pV)
        public void Reset()
        public bool MoveNext()
        public Int16 Current
    }
}
public class DenseGRAPH Graph, iGraph,
        iGraphEnumerable
public class SparseMultiGRAPH : Graph, iGraph,
        iGraphEnumerable
public class SC
{
    protected Graph G;
    protected Stack<Int32> S, Path;
    protected Int32[] pre, id;
    protected bool[] dfs_pre;
    protected Queue<Int32> queue;
    protected Int32 cnt, scnt;
    protected long ticks;

    public SC(Graph pG)
    public long time

    public void GabovClasic()
    protected void GabovClasicIteration(Int32 w)
    public void GabovMultiThreadsStart(Int16 CountThreads)
    public void GabovRemotingStart(Int16 CountRemoting)
    protected void GabovDFSSearch(Int32 v)
}
```

Основними складовими частинами є:

- EDGE, являє собою структуру для ребра графа, яка має початок та кінець.
- Graph – абстрактний клас, який розроблений згідно з парадигмою ООП і не залежить від представлення графа. Конструктор графа приймає два аргументу: задану кількість вершин та логічне значення, чи являється граф орієнтованим. До можливостей можна віднести: побудову (конструктор), підрахунку кількості вершин (V()) та ребер (E()), а також додавання (insert(EDGE e)) та вилучення ребер (remove(EDGE e)). Реалізація інтерфейсу IEnumerator дозволяє використовувати стандартний для C# метод обходу графу. Його реалізація складається з таких елементів: Current – повертає поточний елемент при обході; MoveNext() – пересуває маркер поточного елементу на наступний, та повертає true. Якщо елементів більше немає, повертається значення flase; Reset() – повертає маркер поточного елементу на початок.
- DenseGRAPH – реалізує абстрактний тип данних Graph для графів у представленні матрицею суміжності.
- SparseMultiGRAPH – реалізує абстрактний тип даних Graph для графів у представленні списком суміжних вершин.
- SC – реалізує методи та властивості для роботи з класичним алгоритмом (GabovClasic()), алгоритмами для систем із спільно (GabovMultiThreadsStart(Int16 CountThreads)) та розподіленою пам’яттю (GabovRemotingStart(Int16 CountRemoting)). За допомогою властивості time можна отримати час роботи алгоритму, що тільки-но завершився.

Представлення графу у пам’яті може бути наступним:

Матриця суміжних вершин – Представлена у вигляді матриці булівських значень розміром V * V, елемент якої, що стоїть на перехресті v_i строчки та v_j стовпчика приймає значення true, якщо в графі є ребро, що відповідає вершинам (v_i, v_j) та false в протилежному випадку.

Ця реалізація більше підходить для насичених графів, чим вона й відрізняється від інших. (кількість ребер пропорційна V^2)

Список суміжних вершин - Стандартне представлення графа, якому надають перевагу, коли граф не належить до насичених, є представленням у вигляді списку суміжних вершин. Для цього використовується список, кількість елементів якого дорівнює кількості вершин графа. Його елемент є посиланням на зв’язні списки суміжних вершин [3]. Необхідно лише задати номер вершини, щоб отримати доступ до списку її суміжних вершин. Розмір пам’яті, який необхідний для представлення графа дорівнює V + E (кількість вершин та ребер відповідно).

При додаванні до графа ребра, що з’єднує вершини (v_i, v_j), до списку суміжних вершин вершини v_i необхідно додати вершину v_j. При вилученні ребра (v_i, v_j) необхідно проглядати список суміжних вершин вершини v_i до знаходження v_j і використовуючи операції для видалення елементу із зв’язного списку, вилучити елемент v_j.

Для того щоб обробляти будь-які вершини, суміжні до заданої, використовується правила пересування між елементами зв’язного списку.

Така реалізація більше підходить для розряджених графів, чим вона й відрізняється від інших. (кількість ребер пропорційна V + E)

Введення-виведення структур - Для спрощення тестування застосування та збереження результатів було розроблено елементи, що відповідають за ввід та вивід графів на стандартний потік виводу інформації (екран), у заданий файл та введення графу з заданого файлу.

Генератор випадкових графів

Для перевірки розроблених алгоритмів можливо використати два підходи:

- Перевірку можна робити запуском почергово паралельного та послідовного алгоритму, порівнюючи кількість різних сильних компонент. Це призводить до збільшення часу виконання загальної програми тестування.
- Саме тому другою реалізацією створення випадкових графів є розроблений генератор з заданою кількістю сильних компонент.

Випадковий граф - Класична математична модель випадкових графів розглядає всі можливі ребра та включає в граф кожне ребро з фіксованою імовірністю p. Якщо необхідно, щоб очікуване число ребер дорівнювало заданому числу E, необхідно обрати вірогідність p = E/V(V - 1).

```
public randG(Int32 E) {…}
```

Ця модель не допускає паралельних ребер, однак число ребер у графі дорівнює E в середньому. Ця реалізація гарно підходить для насичених графів, але час її роботи завжди пропорційний кількості вершин графа.

Генератор заданої кількості сильних компонент - Суть методу полягає у виконанні зворотного алгоритму пошуку сильних компонент. Спочатку будується масив проіндексований іменами вершин, в якому знаходиться номер сильної компоненти для кожної вершини. Далі для вершин з однаковим номером сильних компонент з заданою великою вірогідністю генеруються випадкові ребра, методом випадкових графів. Якщо необхідно згенерувати не лише дерева, а саме сильні компоненти, то з нульової компоненти, з дуже малою вірогідністю, генеруються зв'язки зі всіма іншими компонентами окрім нульової.

```
public Int32 randSC(Int32 SCcnt, bool build_sc) {…}
```

При виклику методу, до нього слід передати бажану кількість сильних компонент та параметр, який вказує на потребу будувати не лише дерева, а сильні компоненти.

Реалізація алгоритму на платформі зі спільною пам'яттю

Спираючись на доведене твердження про можливість розпаралелювання алгоритму, його реалізація складається з наступних частин:

1. Створення потоків. З самого початку створюються задана кількість потоків, але на одиницю менше. Це зумовлене наявністю потоку самої програми, яка теж потребує ресурсів. Новостворені потоки одразу переводяться в стан очікування даних.
2. Пошук в глибину. Використовується класичний рекурсивний пошук в глибину. Після повернення з рекурсії, вершина, з якої вона почалась, вказує на наявність дерева. Ця вершина ставиться в чергу типу FIFO для передачі до потоків.
3. Сигнал. Після знаходження кожного нового дерева, відбувається пошук вільного потоку. Якщо такий знайдено, посилається сигнал активізації.
4. Алгоритм Габова. Після отримання сигналу, задача робить ініціалізацію змінних й застосовує не модифікований варіант алгоритму Габов для кожної вершини, що знаходяться в черзі. Після цього оновлюється масив сильних компонент заданого графу.
5. Завершення пошуку DFS. По завершенню попереднього пошуку з постановкою всіх виявлених дерев у чергу, основний потік також виконує обробку черги таким же алгоритмом, що і всі потоки.
6. Завершення роботи. Якщо є задачі, що не активізовані, то їм посилається сигнал. Далі очікуються їх завершення з подальшим поверненням до основної програми.

Рис 3. Розподіл потоків послідовної та паралельної версій алгоритму (спільна пам'ять)

На рис. 3 зображено порівняння роботи послідовної та паралельної версій алгоритму. Моделювання проводиться на графі з двома сильними компонентами. За рахунок паралелізму досягається виграш у швидкодії.

Проводячи аналогію послідовної та паралельної версій з точки зору швидкодії, видно: виконуючи простий пошук в глибину, час, якого менший за сам алгоритм Габова, паралельний алгоритм відстає. Послідовний метод в той час вже обробляє граф. Знайшовши дерево, починає працювати новий потік обрахунку. По закінченню попереднього пошуку головний потік починає обробляти граф, беручи з черги вершини кожного знайденого дерева, паралельно з іншими. Швидкодія залежить лише від структури вихідного графу.

Паралельна модель з початку затримується, по відношенню з послідовною, на час знаходження першого дерева. Надалі за рахунок роботи декількох потоків, в залежності від структури вихідного графу, нова реалізація випереджає класичну. Припущення про те, що збільшення кількості паралельно працюючих потоків лінійно буде підвищувати швидкодію,

хибне, оскільки існують об’єкти синхронізації роботи всієї системи. Додавання яких, накладає додаткові накладні витрати на ресурси, як пам’яті так і процесору. Якщо розглянути граф, що є сильно зв’язним, то час роботи паралельного алгоритму буде більшим за послідовний на час попереднього пошуку.

Основні складові, що визначають швидкість алгоритму – це швидкість та кількість процесорів, об'єм та швидкість оперативної пам’яті.

Реалізація алгоритму на платформі з розподіленою пам’яттю

В середовищі .Net реалізовано нову модель взаємодії віддалених об’єктів .Net Remoting. Структурний підхід, що використовувався раніше, був замінений в основі на принципи ООП. В застосуванні були реалізовані наступні елементи:

- реалізація дистанційного типу – структури, що мають передаватись до клієнту чи серверу мають бути серіалізованими.
- вибір серверного домену – для проведення швидких тестів та розробки прототипів використовується консольне застосування.
- вибір моделі активації – оскільки неохідно, щоб сервер виконував попередній пошук в глибину й був точкою збору результатів, використовується серверна модель активації. У противному випадку час життя серверної частини застосування залежав би від роботі клиентів. Режим активації обраний Singleton тому, що будь яку кількість клиентів має обробляти один екземпляр об’єкту.
- вибір каналу – в якості каналу передачі даних обрано TcpChannel, оскільки цей канал пересилає дані в двійковому вигляді і є більш швидкодіючим. Якщо необхідно виконувати застосування на машинах, які розділені брандмауером, то необхідно використати канал HttpChannel.
- налаштування параметрів .Net Remoting – щоб зробити застосування більш раціональним та легким у модифікації, використовуеться файл конвігурації у форматі XML [8-9].

Спираючись на доведене твердження про можливість розпаралелювання алгоритму, його реалізація серверної частини виглядає наступним чином:

1. Запуск сервера.
2. Синхронізація вузлів.
3. Пошук в глибину. Використовується класичний рекурсивний пошук в глибину. Після повернення з рекурсії, вершина, з якої вона почалась, вказує на наявність дерева. Ця вершина додається в чергу типу FIFO для подальшої обробки.
4. Обслуговування черги. Після знаходження кожного нового дерева, відбувається пошук вільного вузла. Якщо знайдено, посилається сигнал активації й передається вершина для початку обробки. У протилежному випадку продовжується попередній пошук.
5. Обрахунок даних. Якщо черга не порожня, то головний потік починає об орляти чергу.
6. Збір результатів.
7. Збереження результатів й завершення роботи.

Реалізація клієнтської частини виглядає наступним чином:

1. Запуск та синхронізація клієнтів. Всі вузли кластеру мають бути досяжними и знаходитись в стані готовності.
2. Очікування та отримання даних. Після знаходження нового дерева серверна частина починає пересилку даних необхідних для обрахунку.
3. Алгоритм Габова. Після отримання даних, клієнт виконує ініціалізацію змінних й застосовує не модифікований варіант алгоритму Габова. Після цього оновлюється масив сильних компонент заданого графу й результат повертається серверу.

Результати моделювання

Результати моделювання різних паралельних версій алгоритму наведено на рис. 4. Графи, що використовувались для тестування мали 6000, 9000, 12000, 15000 вершин.

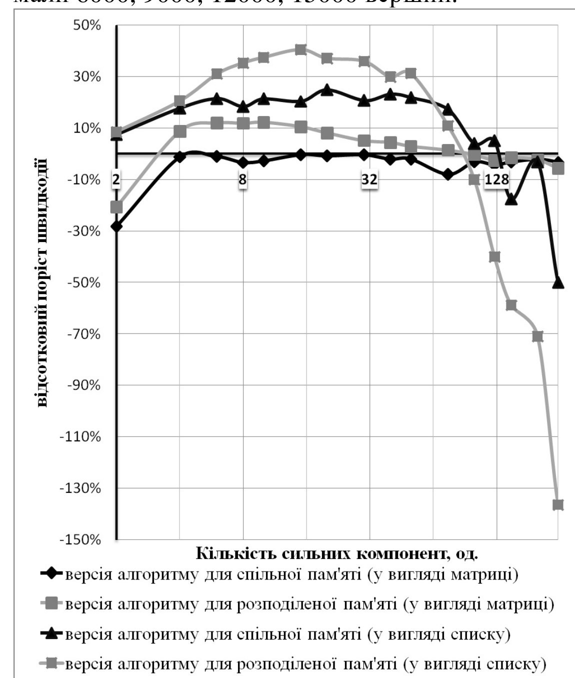

Рис 4 Відсотковий приріст швидкодії в залежності від кількості сильних компонент

На рисунку 4 зображено відсотковий приріст швидкості роботи двох запропонованих

алгоритмів по відношенню до послідовного. Зникнення приросту при збільшенні сильних компонент можна пояснити збільшенням затримок на синхронізацію. Це спостерігається для обох представлень графа. Для графів представлених списком суміжних вершин різкий спад швидкодії можна пояснити наступним чином: при збільшенні сильних компонент час роботи модифікованого алгоритму перестає зменшуватись і прямує до константного значення. Це зумовлено збільшенням затримок на синхронізацію. Оскільки час роботи послідовного алгоритму продовжує зменшуватись, то відношення між часом роботи обох алгоритмів збільшується.

Висновки

.Net Framework – це нова модель для створення систем як в сімействі операційної системи Windows, так і багатьох інших, таких як Unix та Mac OS. Вона реалізує нові принципи взаємодії розподілених систем, основаних на принципах ООП.

Алгоритм Габова для систем з розподіленою пам'яттю працює аналогічно до систем з спільною пам'яттю. Однак додаються додаткові складові, що визначають швидкість алгоритму - це швидкість та кількість процесорів, кількість вузлів мережі, об'єм та швидкість оперативної пам'яті, швидкість каналу передачі даних між вузлами. Останній елемент зумовлює досить суттєві затримки і є найвужчим місцем.

Використання систем з розподіленою пам'яттю дає можливість збільшити кількість одночасно працюючих паралельних задач. Це дало змогу збільшити швидкість роботи методу до 40% у порівнянні із класичним алгоритмом Габова, для графів з кількістю сильних компонент не більше від 100. Подальше збільшення кількості сильних компонент призводить до додаткових затрат на синхронізацію та на швидкість роботи попереднього пошуку. Це є причиною відсутності приросту швидкодії для графів з великою кількістю сильних компонент.

Література

1. Сычев А.В., Баженов М.М, стаття – «Автоматическое пополнение веб-каталога на основе идентификации веб-сообществ с последующей фильтрацией документов по контенту»
2. С.Д.Погорілий "Основоположні математичні відомості. Програмне забезпечення" Навчальний посібник / За ред. О.В.Третяка.- К.
3. С.Д.Погорілий "Програмне конструювання" вид. «Київський університет» 2005 р.
4. Роберт Седжвик - "Фундаментальные Алгоритмы на C++. Алгоритмы на графах" изд. ООО «ДиаСофтЮП», 2002. – 496 с.
5. www.algolist.manual.ru – Алгоритмы методи исходники
6. К.Ю.Богачев "Основы паралельного программирования" – М.БИНОМ. Лаборатория знаний, 2003. – 342 с.
7. www.rsdn.ru - Russian Software Developer Network
8. Маклин С., Нафтел Дж., Уильямс К. «Microsoft .NET Remoting» издательство-торговий дом «Русская Редакция», 2003
9. Троелсен Э. «Язык програмирования C# 2005 и платформа .NET 2.0» - 3-е издание: ООО «И.Д. Вильямс», 2007. – 1168 с.