Государственное образовательное учреждение высшего профессионального

образования

«Московский государственный технический университет

имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №5

По теме:

«Исследование параллельных алгоритмов»

Монахов Д. И.

ИУ7-54

Москва 2018

Оглавление

[1. Аналитическая часть 3](#_Toc529713799)

[1.1. Возможность распараллеливания перемножения матриц 3](#_Toc529713800)

[Отсюда видно, что каждый элемент результирующей матрицы вычисляется независимо от остальных. Следовательно, можно повысить скорость перемножения матриц разбивая результирующую матрицы на несколько и обрабатывая каждую часть параллельно. 3](#_Toc529713801)

[1.2. Описание алгоритмов 3](#_Toc529713802)

[2. Конструкторская часть 5](#_Toc529713803)

[2.1 Разработка алгоритмов 5](#_Toc529713804)

[3. Технологическая часть 7](#_Toc529713805)

[3.1 Требования к программному обеспечению 7](#_Toc529713806)

[3.2 Средства реализации 7](#_Toc529713807)

[3.3 Листинг кода 8](#_Toc529713808)

[4. Экспериментальная часть 11](#_Toc529713809)

[4.1 Постановка эксперимента 11](#_Toc529713810)

[Заключение 13](#_Toc529713811)

# 1. Аналитическая часть

В этой части рассматривается, что такое параллельное выполнение алгоритма и преобразование простого алгоритма перемножения матриц для разделения задачи на несколько потоков

## 1.1. Возможность распараллеливания перемножения матриц

Умножение матриц — одна из основных операций над матрицами. Матрица, получаемая в результате операции умножения, называется произведением матриц.

Пусть даны две матрицы: A с размерами l x m и B с размерами m x n. Произведением матрицы A на B будет матрица C, элементы которой можно найти по формуле (1).

## Отсюда видно, что каждый элемент результирующей матрицы вычисляется независимо от остальных. Следовательно, можно повысить скорость перемножения матриц разбивая результирующую матрицы на несколько и обрабатывая каждую часть параллельно.

## 1.2. Описание алгоритмов

Простой алгоритм перемножения матриц использует формулу (1) напрямую. Во вложенном цикле происходит обход всех элементов результирующей матрицы и происходит запись очередного вычисленного значения. Алгоритм является простым и понятным, однако для матриц больших размеров может быть эффективным использование некоторых других алгоритмов.

Алгоритм Винограда использует идеи, схожие с алгоритмом Штрассена. Рассматривая результат умножения двух матриц очевидно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее. Рассмотрим два вектора V = (v1, v2, v3, v4) и W = (w1, w2, w3, w4). Их скалярное произведение равно: V • W = v1w1 + v2w2 + v3w3 + v4w4.

Это равенство можно переписать в виде: V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) — v1v2 — v3v4 — w1w2 — w3w4.

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем первое: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволяет выполнять для каждого элемента лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

1.3 Модификация алгоритма. Распараллеливание

Как было сказано выше, результирующую матрицу можно разбивать на несколько матриц меньшего размера. Одним из самых простых решений будет группировка строк матрицы. Размер всех групп в идеальном случае должен быть одинаковым, чтобы нагрузка на каждый поток была одинаковой. Но в случае, если количество строк не будет кратно количеству потоков, наиболее оптимальной будет модель, отражаемая формулами (2) и (3). Левая часть формулы (2) – количество строк, обрабатываемых всеми потоками, не включая последний, а левая часть формулы (3) – количество строк для последнего потока. Такая модель позволяет минимизировать время простоя, так как только один поток завершится быстрее всех остальных.

# 2. Конструкторская часть

В этой части будут составлен псевдокод алгоритмов.

## 2.1 Разработка алгоритмов

Псевдокод простого алгоритма умножения матриц

Функция ПростоеУмножение (Матрица1, Матрица2); Возвращает Матрицу

Начало

Для I от 0 до количества столцов в Матрице1

Для J от 0 до количество столбцов в Матрице2

R[i, j] = 0

Для K от 0 до количества строк в Метрице1

R[i, j] = R[i, j] + Матрица1[i, k] \* Матрица2[k, j]

КЦ

КЦ

КЦ

Вернуть R

Конец

Псевдокод параллельного простого алгоритма умножения матриц

Функция ПараллельноеУмножение (Матрица1, Матрица2); Возвращает Матрицу

Начало

Старт = номер\_потока \* размер\_группы

Финал = Старт + размер\_группы

Если Финал > количества строк то

Финал = количество\_строк

Для I от Старт до Финал в Матрице1

Для J от 0 до количество столбцов в Матрице2

R[i, j] = 0

Для K от 0 до количества строк в Метрице1

R[i, j] = R[i, j] + Матрица1[i, k] \* Матрица2[k, j]

КЦ

КЦ

КЦ

Вернуть R

Конец

Псевдокод параллельного алгоритма Винограда умножения матриц

Старт = номер\_потока \* размер\_группы

Финал = Старт + размер\_группы

Если Финал > количества строк то

Финал = количество\_строк

d = b/2

// вычисление rowFactors для G

for i = Старт to Финал do

rowFactor[i] = G[i, 1] \* G[i, 2]

for j = 2 to d do

rowFactor[i] = rowFactor[i] + G[i, 2j — 1] \* G[i, 2j]

end for j

end for i

// вычисление columnFactors для H

for i = Старт to Финал do

columnFactor[i] = H[1, i] \* H[2, i]

for j = 2 to d do

columnFactor[i] = columnFactor[i] + H[2j — 1, i] \* H[2j, i]

end for j

end for i

// вычисление матрицы R

for i = Старт to Финал do

for j = 1 to c do

R[i, j] = -rowFactor[i] — columnFactor[j]

for k = 1 to d do

R[i, j]=R[i, j]+(G[i, 2k-1]+H[2k, j])\*(G[i, 2k] + H[2k-1, j])

end for k

end for j

end for i

// прибавление членов в случае нечетной общей размерности

if (2 \* (b/2) /= b) then

for i = Старт to Финал do

for j = 1 to c do

R[i, j] = R[i, j] + G[i, b] \* H[b, j]

end for j

end for i

# 3. Технологическая часть

В этой части обосновывается выбор языка программирования, необходимых библиотек и будет приведен листинг кодов каждого из алгоритмов.

## 3.1 Требования к программному обеспечению

Для того, чтобы запустить программу, необходимо иметь:

ОС windows 10, объем оперативной памяти не менее 512 мб, среда visual studio 2017.

## 3.2 Средства реализации

В качестве языка программирования был выбран C#. Его выбор основан на следующих причинах:

1. язык программирования C# является частью платформы .NET Framework, которая хорошо документирована, обладает хорошей стабильностью работы, и имеет богатый набор библиотечных решений;
2. C# позволяет программисту уделять больше времени написанию кода и не тратить время на побочные действия;
3. C# просто отлаживать, он не обладает уязвимостями языка C++, например, проблемами с указателями;
4. C# по скорости работы почти не уступает языку C++.

Время выполнения замеряется средством .NET под названием StopWatch;

## 3.3 Листинг кода

Листинг параллельного простого алгоритма умножения матриц

private void Multiply(object threadNumObj)

{

int threadNum = (int) threadNumObj;

int startRow = threadNum \* \_oneThreadRange;

int endRow = startRow + \_oneThreadRange;

if (endRow > \_lineCount)

endRow = \_lineCount;

for (int i = startRow; i < endRow; i++)

{

for (int j = 0; j < \_result.Cols; j++)

{

\_result[i, j] = 0;

for (int k = 0; k < \_a.Rows; k++)

{

\_result[i, j] = \_result[i, j] + \_a[i, k] \* \_b[k, j];

}

}

}

}

Листинг параллельного алгоритма Винограда умножения матриц

\_totalThreads = threadCount;

\_g = a;

\_h = b;

\_a = \_g.Rows;

\_b = \_g.Cols;

\_c = \_h.Cols;

\_d = \_b / 2;

\_result = new Matrix(\_a, \_c);

rowFactors = new int[\_a];

colFactors = new int[\_b];

List<Thread> threadPool = new List<Thread>();

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

// Processing row factors

for (int i = 0; i < \_totalThreads; i++)

{

Thread thread = new Thread(GetRowFactors);

thread.Start(i);

threadPool.Add(thread);

}

WaitForAll(threadPool);

threadPool.Clear();

// Processing col factors

for (int i = 0; i < \_totalThreads; i++)

{

Thread thread = new Thread(GetColFactors);

thread.Start(i);

threadPool.Add(thread);

}

WaitForAll(threadPool);

threadPool.Clear();

// Calculating matrix

for (int i = 0; i < \_totalThreads; i++)

{

Thread thread = new Thread(MatrixCalculation);

thread.Start(i);

threadPool.Add(thread);

}

WaitForAll(threadPool);

threadPool.Clear();

// Adding remainders

if (\_b % 2 == 1)

{

for (int i = 0; i < \_totalThreads; i++)

{

Thread thread = new Thread(AddRemainders);

thread.Start(i);

threadPool.Add(thread);

}

WaitForAll(threadPool);

threadPool.Clear();

}

stopwatch.Stop();

\_ticks = stopwatch.ElapsedTicks;

}

private void WaitForAll(IEnumerable<Thread> theads)

{

foreach (var thread in theads)

{

thread.Join();

}

}

private void GetRowFactors(object threadNumObj)

{

int threadNum = (int) threadNumObj;

int range = (int)Math.Ceiling((double)\_a / \_totalThreads);

int startRow = threadNum \* range;

int endRow = startRow + range;

if (endRow > \_a)

endRow = \_a;

for (int i = startRow; i < endRow; i++)

{

rowFactors[i] = \_g[i, 0] \* \_g[i, 1];

for (int j = 2; j <= \_d; j++)

{

rowFactors[i] = rowFactors[i] + \_g[i, 2 \* j - 2] \* \_g[i, 2 \* j - 1];

}

}

}

private void GetColFactors(object threadNumObj)

{

int threadNum = (int) threadNumObj;

int range = (int)Math.Ceiling((double)\_c / \_totalThreads);

int startRow = threadNum \* range;

int endRow = startRow + range;

if (endRow > \_c)

endRow = \_c;

for (int i = startRow; i < endRow; i++)

{

colFactors[i] = \_h[0, i] \* \_h[1, i];

for (int j = 2; j <= \_d; j++)

{

colFactors[i] = colFactors[i] + \_h[2 \* j - 2, i] \* \_h[2 \* j - 1, i];

}

}

}

private void MatrixCalculation(object threadNumObj)

{

int threadNum = (int) threadNumObj;

int range = (int)Math.Ceiling((double)\_a / \_totalThreads);

int startRow = threadNum \* range;

int endRow = startRow + range;

if (endRow > \_a)

endRow = \_a;

for (int i = startRow; i < endRow; i++)

{

for (int j = 0; j < \_c; j++)

{

\_result[i, j] = -rowFactors[i] - colFactors[j];

for (int k = 1; k <= \_d; k++)

{

\_result[i, j] = \_result[i, j] +

(\_g[i, 2 \* k - 2] + \_h[2 \* k - 1, j]) \*

(\_g[i, 2 \* k - 1] + \_h[2 \* k - 2, j]);

}

}

}

}

private void AddRemainders(object threadNumObj)

{

int threadNum = (int) threadNumObj;

int range = (int)Math.Ceiling((double)\_a / \_totalThreads);

int startRow = threadNum \* range;

int endRow = startRow + range;

if (endRow > \_a)

endRow = \_a;

for (int i = startRow; i < endRow; i++)

{

for (int j = 0; j < \_c; j++)

{

\_result[i, j] = \_result[i, j] + \_g[i, \_b - 1] \* \_h[\_b - 1, j];

}

}

}

# 4. Экспериментальная часть

В этой части лабораторной работы будут поставлены эксперименты над реализованными алгоритмами и сделаны выводы.

## 4.1 Постановка эксперимента

Поставим эксперимент, в котором будет выявлена зависимость времени работы алгоритмов от количества потоков, открытых в программе. Замеры проводились на компьютере с 8-миядерным процессором. Все матрицы имели фиксированный размер 600 на 600. Количество потоков будет увеличиваться в геометрической прогрессии от 1 до 1024 с множителем 2.

На рисунках (1) и (2) представлены графики зависимости времени работы от количества потоков. Видно, что в обоих случаях минимум времени достигается в точке, где количество потоков равно количеству ядер процессора.



Рисунок 1. Алгоритм Винограда



Рисунок 2. Простой алгоритм

# Заключение

По результатам можно сделать вывод о том, что распараллеливание алгоритмов может дать существенный выигрыш по времени, причем наибольшая эффективность достигается, когда количество потоков равно количеству доступным процессоров (физических или логических ядер).