Минобрнауки России

Санкт-Петербургский государственный технический университет

Факультет технической кибернетики  
Кафедра Информационных и Управляющих Систем

**Курсовой проект**

Методы оптимизации

Оптимизация Java приложения

Выполнил студент гр. 5084/12 Лукашин А.А.

Руководитель

Иванов А. С.

«\_\_\_»\_\_\_\_\_\_\_\_2012

Санкт-Петербург   
2012

Оглавление

[Оптимизация Java приложения 1](#_Toc341998296)

[Постановка задачи 3](#_Toc341998297)

[Описание 3](#_Toc341998298)

[Сравнение 3](#_Toc341998299)

[Платформа 3](#_Toc341998300)

[Реализация 3](#_Toc341998301)

[Класс запуска 3](#_Toc341998302)

[ApacheCalculate 4](#_Toc341998303)

[JamaCalculate 4](#_Toc341998304)

[HandMadeCalculate 4](#_Toc341998305)

[Оптимизация 1000х1000 4](#_Toc341998306)

[Без оптимизации 4](#_Toc341998307)

[Алгоритм 4](#_Toc341998308)

[Результаты 5](#_Toc341998309)

[Final размерность 5](#_Toc341998310)

[Алгоритм 5](#_Toc341998311)

[Результаты 5](#_Toc341998312)

[Оптимизация использования кэш 5](#_Toc341998313)

[Алгоритм 6](#_Toc341998314)

[Результаты 6](#_Toc341998315)

[Объединение циклов 6](#_Toc341998316)

[Алгоритм 6](#_Toc341998317)

[Результаты 7](#_Toc341998318)

[Замена проверки в цикле на обработку исключения 7](#_Toc341998319)

[Алгоритм 7](#_Toc341998320)

[Результаты 7](#_Toc341998321)

[Оптимизация 5000х5000 7](#_Toc341998322)

[С проверкой в циклах 7](#_Toc341998323)

[Алгоритм 8](#_Toc341998324)

[Результаты 8](#_Toc341998325)

[С обработкой исключений 8](#_Toc341998326)

[Алгоритм 9](#_Toc341998327)

[Результаты 9](#_Toc341998328)

[Флаги jvm 9](#_Toc341998329)

[Заключение 9](#_Toc341998330)

# Постановка задачи

## Описание

В рамках данного проекта необходимо применить различные методы оптимизации Java приложений. В качестве Программы для оптимизации был взят модельный пример – перемножение двух матриц большого размера, заполненных числами с плавающей точкой двойной точности (double).

## Сравнение

Для сравнения производительности были взяты две сторонних библиотеки: Apache Math Commons 2.2 и Jama 1.0.2, а так же был написан собственный алгоритм перемножения двух матриц, над которым в дальнейшем и проводилась оптимизация.

## Платформа

В качестве платформы для запуска использовалась:

* Intel core 2 Duo 3MGHz
* 4 gb RAM
* Window 7 x64
* Java 1.6u32

# Реализация

В рамках данной задачи выполнялось перемножение двух матриц. Как происходили вызовы методов сторонних библиотек будет показано ниже. В качестве собственно реализации было взято обычное перемножение тремя вложенными циклами (O(n3)), которое затем перерабатывалось.

## Класс запуска

**public** **class** Main {

**private** **static** **final** **int** *LENGTH* = 5000;

**public** **static** **void** main(String[] args) {

System.*out*.println("---------------------------------------");

System.*out*.println("Apache library test:");

System.*out*.println("Initialisation:");

System.*out*.println(ApacheCalculate.*init*(*LENGTH*, *LENGTH*));

System.*out*.println("Calculation:");

System.*out*.println(ApacheCalculate.*calculate*());

System.*out*.println("---------------------------------------");

System.*out*.println("Jama library test:");

System.*out*.println("Initialisation:");

System.*out*.println(JamaCalculate.*init*(*LENGTH*, *LENGTH*));

System.*out*.println("Calculation:");

System.*out*.println(JamaCalculate.*calculate*());

System.*out*.println("---------------------------------------");

System.*out*.println("HandMade library test:");

System.*out*.println("Initialisation:");

System.*out*.println(HandMadeCalculate.*init*(*LENGTH*, *LENGTH*));

System.*out*.println("Calculation:");

System.*out*.println(HandMadeCalculate.*calculate*());

System.*out*.println("---------------------------------------");

}

}

## ApacheCalculate

**public** **class** ApacheCalculate {

**static** Array2DRowRealMatrix *matrix1*;

**static** Array2DRowRealMatrix *matrix2*;

**public** **static** **long** init(**int** n1, **int** n2) {

**long** start = System.*currentTimeMillis*();

*matrix1* = **new** Array2DRowRealMatrix(Utils.*getRandomArray*(n1, n2));

*matrix2* = **new** Array2DRowRealMatrix(Utils.*getRandomArray*(n1, n2));

**return** System.*currentTimeMillis*() - start;

}

**public** **static** **long** calculate() {

**long** start = System.*currentTimeMillis*();

*matrix1*.multiply(*matrix2*);

**return** System.*currentTimeMillis*() - start;

}

}

## JamaCalculate

**public** **class** JamaCalculate {

**private** **static** Matrix *matrix1*;

**private** **static** Matrix *matrix2*;

**public** **static** **long** init(**int** n1, **int** n2) {

**long** start = System.*currentTimeMillis*();

*matrix1*= **new** Matrix(Utils.*getRandomArray*(n1, n2));

*matrix2*= **new** Matrix(Utils.*getRandomArray*(n1, n2));

**return** System.*currentTimeMillis*() - start;

}

**public** **static** **long** calculate() {

**long** start = System.*currentTimeMillis*();

*matrix1*.times(*matrix2*);

**return** System.*currentTimeMillis*() - start;

}

}

## HandMadeCalculate

Данный класс будет представлен далее, так как он менялся в зависимости от применяемых методов оптимизации.

# Оптимизация 1000х1000

Первый цикл оптимизации проведем на массивах размерности 1000 на 1000

## Без оптимизации

### Алгоритм

private static double[][] multiply(double[][] m1, double[][] m2) {

double[][] result = new double[m1.length][m2[0].length];

for (int i = 0; i < m1[0].length; i++) {

for (int j = 0; j < m1.length; j++) {

double summand = 0.0;

for (int k = 0; k < m2.length; k++) {

summand += m1[i][k] \* m2[k][j];

}

result[i][j] = summand;

}

}

return result;

}

### Результаты

---------------------------------------

Apache library test:

Initialisation:

128

Calculation:

8458

---------------------------------------

Jama library test:

Initialisation:

79

Calculation:

1659

---------------------------------------

HandMade library test:

Initialisation:

77

Calculation:

9759

---------------------------------------

## Final размерность

Дабы не спрашивать на каждой итерации размеры массива, можно ввести константы, хранящие подобную информацию

### Алгоритм

private static double[][] multiply(double[][] m1, double[][] m2) {

double[][] result = new double[m1.length][m2[0].length];

final int aColumns = m1.length;

final int aRows = m1[0].length;

final int bColumns = m2.length;

final int bRows = m2[0].length;

for (int i = 0; i < aRows; i++) {

for (int j = 0; j < aColumns; j++) {

double summand = 0.0;

for (int k = 0; k < bColumns; k++) {

summand += m1[i][k] \* m2[k][j];

}

result[i][j] = summand;

}

}

return result;

}

### Результаты

---------------------------------------

HandMade library test:

Initialisation:

93

Calculation:

8925

---------------------------------------

## Оптимизация использования кэш

Для того, чтобы Java могла использовать кэш в полном объеме, необходимо транспонировать одну из матриц, чтобы обращаться к ней построчно.

### Алгоритм

private static double[][] multiply(double[][] m1, double[][] m2) {

double[][] result = new double[m1.length][m2[0].length];

final int aColumns = m1.length;

final int aRows = m1[0].length;

final int bColumns = m2.length;

final int bRows = m2[0].length;

double m2T[][] = new double[bColumns][bRows];

for (int i = 0; i < bRows; i++) {

for (int j = 0; j < bColumns; j++) {

m2T[j][i] = m2[i][j];

}

}

for (int i = 0; i < aRows; i++) {

for (int j = 0; j < aColumns; j++) {

double summand = 0.0;

for (int k = 0; k < bColumns; k++) {

summand += m1[i][k] \* m2T[j][k];

}

result[i][j] = summand;

}

}

return result;

}

### Результаты

---------------------------------------

HandMade library test:

Initialisation:

89

Calculation:

1480

---------------------------------------

## Объединение циклов

Попытаемся объединить циклы транспонирования и умножения

### Алгоритм

private static double[][] multiply(double[][] m1, double[][] m2) {

double[][] result = new double[m1.length][m2[0].length];

final int aColumns = m1.length;

final int aRows = m1[0].length;

final int bColumns = m2.length;

final int bRows = m2[0].length;

double thatColumn[] = new double[bRows];

for (int j = 0; j < bColumns; j++) {

for (int k = 0; k < aColumns; k++) {

thatColumn[k] = m2[k][j];

}

for (int i = 0; i < aRows; i++) {

double thisRow[] = m1[i];

double summand = 0;

for (int k = 0; k < aColumns; k++) {

summand += thisRow[k] \* thatColumn[k];

}

result[i][j] = summand;

}

}

return result;

}

### Результаты

---------------------------------------

HandMade library test:

Initialisation:

89

Calculation:

1714

---------------------------------------

Как мы видим, ситуация скорее ухудшилась.

## Замена проверки в цикле на обработку исключения

### Алгоритм

private static double[][] multiply(double[][] m1, double[][] m2) {

double[][] result = new double[m1.length][m2[0].length];

final int aColumns = m1.length;

final int aRows = m1[0].length;

final int bColumns = m2.length;

final int bRows = m2[0].length;

double m2T[][] = new double[bColumns][bRows];

for (int i = 0; i < bRows; i++) {

for (int j = 0; j < bColumns; j++) {

m2T[j][i] = m2[i][j];

}

}

try{

for (int i = 0; ; i++) {

for (int j = 0; j < aColumns; j++) {

double summand = 0.0;

for (int k = 0; k < bColumns; k++) {

summand += m1[i][k] \* m2T[j][k];

}

result[i][j] = summand;

}

}

}catch (IndexOutOfBoundsException ignored) { }

return result;

}

### Результаты

---------------------------------------

HandMade library test:

Initialisation:

90

Calculation:

1489

---------------------------------------

Как мы видим, это ничего не изменило. Однако мы еще вернемся к этому методу.

# Оптимизация 5000х5000

Далее увеличим объем данных до массивов 5000 на 5000

## С проверкой в циклах

Оставим пока проверки в циклах (без обработки исключений)

### Алгоритм

private static double[][] multiply(double[][] m1, double[][] m2) {

double[][] result = new double[m1.length][m2[0].length];

final int aColumns = m1.length;

final int aRows = m1[0].length;

final int bColumns = m2.length;

final int bRows = m2[0].length;

double m2T[][] = new double[bColumns][bRows];

for (int i = 0; i < bRows; i++) {

for (int j = 0; j < bColumns; j++) {

m2T[j][i] = m2[i][j];

}

}

for (int i = 0; i < aRows; i++) {

for (int j = 0; j < aColumns; j++) {

double summand = 0.0;

for (int k = 0; k < bColumns; k++) {

summand += m1[i][k] \* m2T[j][k];

}

result[i][j] = summand;

}

}

return result;

}

### Результаты

---------------------------------------

Apache library test:

Initialisation:

3099

Calculation:

Profiler Agent: Waiting for connection on port 5140 (Protocol version: 9)

Profiler Agent: Established local connection with the tool

Profiler Agent: Connection with agent closed

1738352

29 минут

Jama library test:

Initialisation:

2307

Calculation:

190548

3 минуты 12 секунд

HandMade library test:

Initialisation:

2283

Calculation:

180532

3 минуты 1 секунда

## С обработкой исключений

Теперь замени в двух циклах условия на обработку исключений

### Алгоритм

**private** **static** **double**[][] multiply(**double**[][] m1, **double**[][] m2) {

**double**[][] result = **new** **double**[m1.length][m2[0].length];

**final** **int** aColumns = m1.length;

**final** **int** aRows = m1[0].length;

**final** **int** bColumns = m2.length;

**final** **int** bRows = m2[0].length;

**double** m2T[][] = **new** **double**[bColumns][bRows];

**try** {

**for** (**int** i = 0; i < bRows; i++) {

**for** (**int** j = 0; j < bColumns; j++) {

m2T[j][i] = m2[i][j];

}

}

} **catch** (Exception e) {

}

**try** {

**for** (**int** i = 0;; i++) {

**for** (**int** j = 0; j < aColumns; j++) {

**double** summand = 0.0;

**for** (**int** k = 0; k < bColumns; k++) {

summand += m1[i][k] \* m2T[j][k];

}

result[i][j] = summand;

}

}

} **catch** (Exception e) {

}

**return** result;

}

### Результаты

HandMade library test:

Initialisation:

2283

Calculation:

177061

2 минуты 57 секунд

# Флаги jvm

В рамках проводимой оптимизации было проведено исследование работы JIT компилятора на уровне флагов, задаваемых для JVM при запуске на исполнение.

**-XX:+UseLargePages** - Use large page memory.

**-XX:+AggressiveOpts** – turns on performance compiler optimization.

**-XX:+PrintCompilation** – Print message when method is compiled

**-XX:CompileThreshold** - Number of method invocations/branches before compiling

Однако флаги или не давали ощутимого прироста, или ухудшали показатели производительности.

# Заключение

В рамках проделанной работы была произведена оптимизация алгоритма перемножения двух матриц большой размерности. По сравнению с первоначальным вариантом было получено повышение производительно примерно в 6 раз. Были исследованы различные библиотеки, а так же влияние флагов JVM при выполнении задачи.