Проект за паралелно умножаване на матрици

Целта на проекта е създаването на програма, която умножава две матрици понишково. Програмата трябва да пресмята времето необходимо за извършване на операцията. Размерностите на матриците и броя на отделните нишки се задават от потребителя. Трябва да има възможност за „тих”(отчита се само времето за обработка) режим, който се задава от потребителя.

Програмата е реализирана на Java, тъй като Java e един от най-широко разпространените езици за програмиране. Java предлага унифицирана и безплатна платформа за реализация на софтуер. Може да се използва под различни операционни системи и предоставя мощни средства за паралелни алгоритми.

Използва се Java виртуална машина(версия 1.6, ъпдейт 20), която компилира и реализира Java кода. Java виртуалната машина е безплатна и е собственост на *Sun Microsystems*. Достъпна е в Интернет на адрес: <http://www.java.com/en/download/> .Текущата програма работи и на по-старата версия 1.5 ъпдейт 22.

Програмата е разработена, чрез Eclipse. Eclipse е софтуер базиран на Java, който предоставя мощно средство за управление и разработка на софтуер. Eclipse проследява различни видове грешки, както синтактични така и логически, които се допускат при писане на софтуер.

За целите на проекта са използвани 4 Java библиотеки:

* **java.io.IOException** – предоставя функционалност за проследяване и реагиране на грешки, причинени от четенето на данни, които потребителят управлява.
* ***java.util.concurrent.ExecutorCompletionService*** – предоставя функционалснот за паралелно обработване на задачи.
* ***java.util.concurrent.ExecutorService*** – услуга за задаване на определен брой нишки и разпределяне на задачи.
* ***java.util.concurrent.Executors*** – обща библиотека служеща за допълване на горните две.

Програмата е реализирана с помоща на класове, т.к. Java e обектно ориентиран език и изисква обектно-ориентиран подход. Налични са два основни класа: *class Matrix* и *class CoputeRow*. Класът Matrix е главен и се използва за входна точка на програмата. Функцията main(string args[]) e в този клас. На практика цялата програма е реализирана с помоща на няколко метода: *static double[][] RandMatrixValues*, *static boolean equals*, *static void printMatrix(double [][] а, char quiet )*, *static void doit(int aRows, int aCols,int aRows2,int aCols2, int t,char quiet), public static double [][] mult(double [][] a, double [][] b)*, *public static double [][] mtmult(double [][] a, double [][] b,int t).*

* **static double[][] RandMatrixValues** **(int aRows, int aCols) –** Взима за параметри редове на матрица и колони на матрица, връща матрица с aRows на брой редове и aCols на брой колони. Стойностите са произволни, генерирани с Math.random() – функцята, вградена със стандартния Java пакет. Стойностите са произволни положителни реални числа.
* **boolean equals** **(double[][] a,double[][] b)** – Взима две матрици и ги сравнява стойност по стойност. Ако са равни връща 1 – истина в противен случай 0 – лъжа.
* **void printMatrix(double [][] а, char quiet )** – Отпечатва матрица в конзолата. Има параметър quiet, който оказва дали функцията да отпечати матрицата или не.
* **static void doit(int aRows, int aCols,int aRows2,int aCols2, int t,char quiet)** – Oбща функция, която е сбор от всички останали. Всички изчисления и операции се извършват от функцията *doit*. Функцията *doit* взима всички необходими първоначални параметри за изчисленията. aRows-брой редове на първата матрица, аCols – брой колони на първата матрица, респективно aRows2 – брой редове на втората матрица, aCols2 – брой колони на втората матрица, t – броя на нишките, quiet – знак посочващ режима на работа.
* **public static double [][] mult(double [][] a, double [][] b)** – Умножава две матрици последователно и връща резултат. Използват се стандартните методи за умножение на матрици чрез цикъл for.
* **public static double [][] mtmult(double [][] a, double [][] b,int t)** – Умножава две матрици паралелно и връща резултат матрица. Тук параметарът t- обозначава броя на нишките.

След като се вкарат входните параметри, програмата създава две матрици с произволни стойности изполвайки функцията *RandMatrixValues*. Отпечатва ги в конзолата и задава параметър time който взима текущото време в милисекудни, използвайки вградения в Java метод *Sytem.currentTimeMillis();* След това умножава матриците последователно чрез функцията *mult.* За да се пресметне времето за умножение се пресмята пак текущото време (вече след умножението), и се изважда с *time*. По този начин се получава времето за умножаване на матриците.

По същата схема се пресмята и времето необходимо за паралелното пресмятане на матрицата. Това се прави с цел да се сравни, ефективността на последователното пресмятане на матрици с паралелното пресмятане.

За да може да се пресмята паралелно матрицата се разделя на редове. Като пресмятането на всеки ред се пуска по отделна нишка. Броя на нишките е от съществено значение за времето необходимо на програмата да умножи две матрици. Броя на нишките се определя статично от потребителя. Ако нишките са по-малко от, в момента изпълнимите задачи, се изчаква да се освободи нишка, след което тя се заема да изчисли друга задача.

За да се определи точният брой на нишките се използва класа *ExecutorService*, който има метод *Executors.newFixedThreadpool(t),* служещ за създаване на „басейн” от фиксиран брой нишки. Чрез обект на класа ExecutorComplitionService е възможно да се използва *submit()* метод, който пуска задачи за паралелна обработка. В нашия случай всяка задача представлява изчисление на отделен ред от матрицата. За целта се използва класа *ComputeRow*, който е вътрешен за класа *Matrix*. ComputeRow импелментира инерфейса runnable, който трябва да бъде имплементиран от всеки клас, чийто инстанции са предназначени да се изпълняват от нишки. Runnable предоставя метод run(), който се стартира при пускането на всяка нишка. За нашата програма, при умножаване на две матрици [][]a и [][]b, трябва да се умножават само стойностите, които сформират един ред от матрицата [][]c (матрицата получена при умножаване на [][]a и [][]b).Метода run() е пренаписан, така че да изчислява само по-един ред от матрицата [][]c. Обектите на класа ComuteRow при инициализиране взимат следните параметри, дефинирани в конструктора на класа: матрица [][]a, матрица [][]b, матрица [][]c и номер на реда който трябва да бъде изчислен. Използваме new ComputeRow(a,b,c,i), за да заделим памет за нова инстанция на класа ComputeRow. Tук *а* е матрица, *b* e матрица, *c* e матрицата в която се записва резултата и *i* e определен ред на матрицата *c*.

Метода *submit(ComputeRow(a,b,c,i))*, взима инстанция на класа ComputeRow и пуска метода *Run()*, паралелно за всяка нова инстанция. Програмно идеята се реализира с цикъл for:

***for*** *(****int*** *i = 0; i < a.length; ++i) {*

*ecs.submit(****new*** *ComputeRow(a,b,c,i),****null****);*

*}*

ecs – e обект на класа ExecutorComplitionService

Резултатите от отделните нишки се взимат чрез ecs.take():

***for*** *(****int*** *i = 0; i < a.length; ++i) {*

*ecs.take();*

}

Програмата беше тествана на лаптоп ASUS F5V, Intel Core duo CPU T2250 1.73 GHz, 2GB ram. Операционна система - Windows 7, 32 битова .

Таблица на времето за умножение на матрици:

|  |  |  |  |
| --- | --- | --- | --- |
| **Матрици** | **Нормално умножение** | **3 нишки** | **4 нишки** |
| А[60][50]\*B[50][70] | 7 милисек. | 18 милисек. | 24 милисек. |
| А[150][200]\*B[200][130] | 55 милисек. | 45 милисек. | 55 милисек. |
| А[200][350]\*B[350][400] | 539 милисек. | 309 милисек. | 303 милисек. |
| А[350][500]\*B[500][400] | 1563 милисек. | 923 милисек. | 893 милисек. |
| А[500][700]\*B[700][600] | 5953 милисек. | 2943 милисек. | 2850 милисек. |

От таблицата ясно си личи ползата от паралелното умножение на матрици, особено при работа с по-големи матрици. Вижда се също, че при работа с по-малки матрици ефектът от многонишковото умножение се губи.Това е така, защото разпределянето по-нишки отнема повече време от колкото нормалното умножение. При работа с големи размерности обаче, явно се открояват предимствата времето за смятане намалява близо два пъти. С увеличаване броя на нишките се увеличава и ефективността.

Горните данни показват, че при статично задаване на броя на нишките много внимателно трябва да се прецени, колко е оптималният брой нишки за решаване на дадена задача.

Иван Иванов фн:71129.