**Sprawozdanie**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *PWr-pion* | Wydział Informatyki i Zarządzania | | | | |
| Laboratorium  Programowania Równoległego i Rozproszonego | | | | |
| Kierunek: | Informatyka | Rok studiów nr: | 2 | Semestr nr: | 3 |
| Rok akademicki: | 2016/2017 | Grupa administracyjna: | 32b | Grupa ćwiczeniowa: | 32b-g1 |

SPRAWOZDANIE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nr ćwiczenia | Temat ćwiczenia | | | |
| 2b | Implementacja algorytmu rozwiązywania układu równań  liniowych metodą Gaussa-Jordana | | | |
| Termin złożenia sprawozdania |
| 27.3.2017 |
| Data faktycznego  złożenia sprawozdania |
|  |
| Wykonawcy | Nazwisko | Imię | Nr indeksu | Ocena |
| Arciszewski | Jacek | 200412 |  |
| Buczel | Kamil | 200420 |  |
|  |  |  |  |

Data i podpis prowadzącego ćwiczenia

1. **Temat ćwiczenia**

Tematem wykonywanego ćwiczenia była implementacja algorytmu rozwiązywania układu równań liniowych metodą Gaussa-Jordana (dalej GJ).

1. **Zakres ćwiczenia**

Celem ćwiczenia jest opracowanie sekwencyjnego programu w języku C do rozwiązywania układu równań liniowych metodą Gaussa-Jordana (dalej oznaczaną GJ) oraz przekształcenie go w wersję równoległą z zastosowaniem API openMP. Wynika mają zostać zaprezentowane następująco.Liczba równań: liczba całkowita (pusty wiersz)

Wiersz 1:

wykaz oddzielonych spacjami elementów 1. wiersza macierzy poszerzonej,

uporządkowanych rzędami

(pusty wiersz)

Wiersz 2:

wykaz oddzielonych spacjami elementów 2. wiersza macierzy poszerzonej,

uporządkowanych rzędami

… … … … …

(pusty wiersz)

Wiersz n:

wykaz oddzielonych spacjami elementów n. wiersza macierzy poszerzonej,

uporządkowanych rzędami

(pusty wiersz)

Czas obliczeń: x.y

1. **Środowisko realizacji ćwiczenia**

Sprzęt:

* Procesor: Intel Core i5-4210U CPU @ 1.70Ghz
* Pamięć RAM: 12,00 GB
* System Linux Ubuntu 14.04
* Środowisko programistyczne Netbeans 8.2

W poniższym zadaniu rejestracja przebiegu sesji terminalowej nie stanowiła koniecznego wymogu.

1. **Przebieg ćwiczenia i uzyskane wyniki**
   1. **Zadanie nr 1**
      1. **Treść polecenia**

Zapoznaj się z algorytmem, wymienionym w tytule. Korzystając z tego algorytmu, ręcznie rozwiąż układ trzech równań z jedną kolumną wyrazów wolnych. Jako współczynniki przyjmij wartości całkowite, w tym ujemne.

Opracuj program sekwencyjny w języku C, realizujący algorytm GJ. Nazwa pliku z programem: GJ.c.

Dane do obliczeń: n(całkowita liczba równań) oraz współczynniki macierzy poszerzonej(n\*(n+1) liczb zmiennoprzecinkowych) mają być wczytywane z pliku.

Przygotuj dwa pliki z danymi testowymi:

dane1 – dla układu 5 równań

dane2 – dla układu 10 równań.

Sprawdź poprawność działania programu na danych z przygotowanych plików. Kod programu, zawartość plików z danymi oraz wyniki sprawdzenia włącz do sprawozdania.

* + 1. **Cel czynności**

Celem powyższego zadania było stworzenie implementacji algorytmu GJ oraz weryfikacja poprawności jego działania. Służyło to stworzeniu podstawy pod główny aspekt polecenia tj. badaniu zrównoleglenia.

* + 1. **Sposób wykonania i rezultaty**
       1. **Generowanie danych**

Dla wygenerowania danych do obliczeń napisany został program w Javie który generował macierze o zadanym rozmiarze oraz zakresie współczynników. Dodatkowo dzięki takiemu rozwiązaniu możliwe było przeprowadzenie badań dla macierzy o większych wymiarach, nawet 200x200 czy 500x500. Dzięki takiemu podejściu osiągnęliśmy również pewność co do posiadania przez zadanie prawidłowego i całkowitego rozwiązania.

package gaussgenerator;

import java.util.\*;

/\*\*

\* @author Kamil Buczel

\*/

public class GaussGenerator {

final static boolean debug = false;

public static void main(String[] args) {

int minimum = 1;

int maximum = 10;

Random randomno = new Random();

boolean pos = randomno.nextBoolean();

int size = 5;

int answer;

int[][] arrayOfVar = new int[size][size];

int[] arrayOfNum = new int[size];

System.out.print(String.valueOf(size) + "\n");

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

arrayOfNum[i] = minimum + (int) (Math.random() \* maximum);

}

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

answer = 0;

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

int randomNum = minimum + (int) (Math.random() \* maximum);

if(pos) randomNum = -randomNum;

arrayOfVar[i][j] = randomNum;

if (debug) {

System.out.print(String.valueOf(arrayOfVar[i][j]) + "\*"

+ arrayOfNum[j] + "\t");

} else {

System.out.print(String.valueOf(arrayOfVar[i][j] + "\t"));

}

answer += arrayOfVar[i][j] \* arrayOfNum[j];

}

System.out.print(answer + "\n");

}

System.out.print("\n\n");

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

System.out.print(String.valueOf(arrayOfNum[i]) + "\t");

}

System.out.print("\n\n");

}

}

* + - 1. **Wersja podstawowa**
         1. **Wczytywanie danych**

Funkcja odpowiadająca za wczytywanie danych do macierzy. Wyszukuje ona liczbę odpowiadająca wymiarowi macierzy, a następnie wczytuje współczynniki oraz wyrazy wolne.

#include<stdio.h>

#include<stdlib.h>

#include<time.h>

//load

double\*\* loadMatrix(const char\* file\_name, double\*\* matrix, int\* rowCount) {

FILE\* file = fopen(file\_name, "r");

int i = 0;

int indexColumn = 0;

int indexRow = 0;

double readdouble = 0;

//scan number of rows/columns which is provided at the beginning

fscanf(file, "%d", &i);

\*rowCount = i;

matrix = (double\*\*) calloc(\*rowCount, sizeof (double\*));

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

matrix[j] = (double\*) calloc((\*rowCount) + 1, sizeof (double));

}

while (!feof(file)) {

fscanf(file, "%lf", &readdouble);

matrix[indexColumn][indexRow] = readdouble;

indexRow++;

if (indexRow > (\*rowCount)) {

indexRow = 0;

indexColumn++;

if (indexColumn > \*rowCount - 1) break;

}

}

fclose(file);

return matrix;

* + - * 1. **Wspomagające funkcje matematyczne**

Wspomagające funkcje matematyczne, czyli dzielenie wiersza macierzy przez wartość środkowego elementu sprowadzonego do jedynki, odejmowanie wierszy macierzy oraz główna pętla matematyczna.

void divide(double\* row, double valueOfMiddleElement, int rowSize) {

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

row[i] = row[i] / valueOfMiddleElement;

}

}

void substract(double\* baseRow, double\* destRow, double multiValue, int rowSize) {

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

destRow[i] = destRow[i] - (baseRow[i] / multiValue);

}

}

void subRows(double\*\* matrix, int rowIndex, int rowCount) {

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

if (i != rowIndex && matrix[i][rowIndex] != 0) {

double value = matrix[rowIndex][rowIndex] / matrix[i][rowIndex];

substract(matrix[rowIndex], matrix[i], value, rowCount + 1);

}

}

}

* + - * 1. **Funkcje wyświetlania**

**Funkcja wyświetlająca służąca do wypisania wyników w oknie konsoli.**

void printFinalMatrix(double\*\* matrix, int rowCount) {

printf("Liczba rownan: %d\n", rowCount);

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

printf("Wiersz %d:\n", i);

for (int j = 0; j < rowCount + 1; j++) {

printf("%.2lf ", matrix[i][j]);

}

printf("\n");

}

}

* + - * 1. **Głównie funkcje obliczające**

Główne funkcje obliczające, czyli implementacja algorytmu GJ oraz zapis rezultatu do macierzy rozwiązań. Funkcja calculate zawiera w krokach kolejno wczytanie macierzy, obliczenie wyników, wyświetlenie macierzy wynikowej i ostatecznie zwolnienie wyników. Wyświetlanie macierzy wynikowej jest zakomentowane ze względu na potencjalne zakłamanie pomiarów czasowych oraz ilość danych przekazanych na konsolę, co pokazane jest w kolejnym podpunkcie.

double\* calculateGJ(double\*\* matrix, int rowCount) {

double\* solutionMatrix = (double\*) calloc(rowCount, sizeof (double));

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

divide(matrix[i], matrix[i][i], rowCount + 1);

subRows(matrix, i, rowCount);

}

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

solutionMatrix[i] = matrix[i][rowCount];

}

return solutionMatrix;

}

void releaseMemory(double\*\* matrix, double\* solution, int rowCount) {

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

free(matrix[i]);

}

free(matrix);

free(solution);

}

void calculate(const char\* inputFileName) {

//creating two dimensional matrix

double\*\* matrix;

double\* solution;

int rowCount = 0;

matrix = loadMatrix(inputFileName, matrix, &rowCount);

solution = calculateGJ(matrix, rowCount);

//printFinalMatrix(matrix, rowCount);

releaseMemory(matrix, solution, rowCount);

}

* + - * 1. **Główny trzon programu**

Główny człon programu zawiera funkcję main oraz calc, które odpowiadają za pomiar oraz obliczenie czasu. Dodatkowo dla zmniejszenia rozrzutu pomiarów wszystkie obliczenia zostały wykonane tysiąckrotnie, a następnie wyliczona została wartość średnia.

void calc(double\* firstTime, double\* secondTime, double\* completeTime) {

//starting the time and choosing accuracy

double timeStart = clock() / (CLOCKS\_PER\_SEC / 1000000);

calculate("dane1");

//getting the time

double firstDataTime = clock() / (CLOCKS\_PER\_SEC / 1000000);

calculate("dane2");

double finishTime = clock() / (CLOCKS\_PER\_SEC / 1000000);

\*firstTime += (firstDataTime - timeStart) / 1000000;

\*secondTime += (finishTime - firstDataTime) / 1000000;

\*completeTime += (finishTime - timeStart) / 1000000;

}

int main() {

int numberOfLoops = 1000;

double firstTime, secondTime, completeTime;

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

calc(&firstTime, &secondTime, &completeTime);

}

printf("\nAVERAGE TIMES:\n"

"First data: %.6lf\n"

"Second data: %.6lf\n"

"Complete time: %.6lf\n",

firstTime / numberOfLoops,

secondTime / numberOfLoops,

completeTime / numberOfLoops);

return (0);

}

* + - 1. **Wersja zrównoleglona**

Wersja zrównoleglona programu z wykorzystaniem openMP. Ważnym jest by pamiętać aby w linii kompilacji uwzględnić dodatkową opcję -fopenmp. Dla uzyskania wyników opartych o zrównoleglenie użyliśmy #pragma omp parallel for num\_threads(thr) gdzie thr oznacza liczbę wątków analizowaną w aktualnej pętli. Samo wybranie funkcji subRows opierało się na analizie pracy programu i oszacowaniu złożoności obliczeń, tak aby wyselekcjonować miejsca gdzie ta złożoność jest największa.

* + - * 1. **Funkcja matematyczna wzbogacona o zrównoleglenie pętli**

void subRows(double\*\* matrix, int rowIndex, int rowCount, int thr) {

#pragma omp parallel for num\_threads(thr)

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

if (i != rowIndex && matrix[i][rowIndex] != 0) {

double value = matrix[rowIndex][rowIndex] / matrix[i][rowIndex];

substract(matrix[rowIndex], matrix[i], value, rowCount + 1);

}

}

}

* + - * 1. **Główna część programu wzbogacona o inkrementację przyznanych wątków**

Główna część programu różni się od wersji podstawowej dodatkową pętlą inkrementującą liczbę analizowanych wątków tak aby wykazać jaki ma ona wpływ na czas wykonania programu. Dolna, zakomentowana część programu odpowiada za wyświetlenie wyników w postaci łatwej do wyeksportowania do programu excel, co z kolei pozwoliło na wygenerowanie wykresu.

int main() {

int numberOfLoops = 100;

int numberOfThreads = 50;

for (int thr = 1; thr <= numberOfThreads; thr++) {

double firstTime = 0,

secondTime = 0,

completeTime = 0;

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

calc(&firstTime, &secondTime, &completeTime, thr);

}

printf("\nAVERAGE TIMES:\n"

"First data: %.6lf\n"

"Second data: %.6lf\n"

"Complete time: %.6lf\n\n"

"Number of threads: %d\n",

firstTime / numberOfLoops,

secondTime / numberOfLoops,

completeTime / numberOfLoops,

thr);

// printf(

// "%.6lf "

// "%.6lf "

// "%.6lf "

// "%d\n",

// firstTime / numberOfLoops,

// secondTime / numberOfLoops,

// completeTime / numberOfLoops,

// thr);

}

return (0);

}

* 1. **Wyniki**

Liczba rownan: 5

Wiersz 0:

1.00 0.00 0.00 0.00 0.00 1.00

Wiersz 1:

0.00 1.00 0.00 0.00 0.00 3.00

Wiersz 2:

0.00 0.00 1.00 0.00 0.00 1.00

Wiersz 3:

0.00 0.00 0.00 1.00 0.00 1.00

Wiersz 4:

1. 0.00 0.00 0.00 1.00 2.00

0.000023

Liczba rownan: 10

Wiersz 0:

1.00 0.00 0.00 0.00 0.00  0.00 0.00 0.00  0.00 0.00 5.00

Wiersz 1:

0.00 1.00 0.00 0.00 0.00 0.00 0.00  0.00  0.00 0.00 2.00

Wiersz 2:

0.00 0.00 1.00  0.00 0.00  0.00 0.00 0.00  0.00 0.00 4.00

Wiersz 3:

0.00 0.00 0.00 1.00 0.00  0.00 0.00 0.00  0.00  0.00 2.00

Wiersz 4:

0.00 0.00 0.00 0.00 1.00  0.00 0.00  0.00 0.00 0.00 1.00

Wiersz 5:

0.00 0.00 0.00 0.00 0.00 1.00 0.00  0.00  0.00 0.00 4.00

Wiersz 6:

0.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 0.00 3.00

Wiersz 7:

0.00 0.00 0.00  0.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00

Wiersz 8:

0.00  0.00  0.00 0.00  0.00  0.00  0.00  0.00 1.00 0.00 5.00

Wiersz 9:

1. 0.00 0.00 0.00 0.00  0.00 0.00 0.00 0.00 1.00 4.00

0.000055

1. **Wnioski z przeprowadzonych prac**
   1. **Podstawowa wersja programu**

AVERAGE TIMES:

First data: 0.000023

Second data: 0.000055

Complete time: 0.000077

Pomiar czasu dla wersji podstawowej przebiegł zgodnie z oczekiwaniami. Obliczenia dla macierzy pięciowymiarowej okazały się około dwukrotnie szybsze niż dla jej 10 wymiarowego odpowiednika. Suma czasu obliczeń oscylowała w okolicach 0.00008. Dzięki przeprowadzeniu badań ponad stukrotnie i uśrednieniu wyników rozwiązaliśmy problem nieregularności pomiarów objawiający się przy pojedynczej iteracji w odchyleniach nawet do 30%.

Jako, iż do generowania wyników stworzony został dodatkowy program w Javie weryfikacja poprawności rozwiązań nie była konieczna.

* 1. **Zrównoleglona wersja programu**

Zrównoleglona wersja programu osiągnęła gorsze wyniki czasowy aniżeli wersja podstawowa. Analiza działania wykazała, iż mogło wynikać to z faktu, iż stworzenie i zarządzanie wątkami wiązało się z większym kosztem czasowym aniżeli potencjalne korzyści jakie płynęły z podziału zadań. Wiązało się to z mało złożonością obliczeniową zadania opierającego się na prostych operacjach matematycznych na liczbach całkowitych.

Postanowiliśmy zatem przeanalizować wpływ utworzonych wątków na czas wykonywania się zadań.

Zależność tą obrazuje wykres oraz tabela zamieszczona poniżej.

Wyk. 1 Zależność czasu wykonania zadania od liczby wątków

|  |  |  |  |
| --- | --- | --- | --- |
| 5 wymiarów | 10 wymiarów | suma | liczba wątków |
| 0,000027 | 0,000059 | 0,000086 | 1 |
| 0,000054 | 0,000113 | 0,000167 | 2 |
| 0,000085 | 0,000205 | 0,000291 | 3 |
| **0,000696** | **0,000399** | **0,001095** | **4** |
| 0,000188 | 0,000366 | 0,000554 | 5 |
| 0,000209 | 0,000418 | 0,000627 | 6 |
| 0,000246 | 0,000485 | 0,000730 | 7 |
| 0,000273 | 0,000541 | 0,000814 | 8 |
| 0,000319 | 0,000634 | 0,000953 | 9 |
| 0,000358 | 0,000715 | 0,001073 | 10 |
| ... | ... | ... | ... |
| 0,000743 | 0,001482 | 0,002226 | 20 |
| ... | ... | ... | ... |
| 0,001130 | 0,002247 | 0,003378 | 30 |
| ... | ... | ... | ... |
| 0,001353 | 0,002694 | 0,004046 | 36 |
| **0,001224** | **0,002414** | **0,003638** | **37** |
| 0,001421 | 0,002827 | 0,004248 | 38 |
| 0,001471 | 0,002923 | 0,004394 | 39 |
| 0,001501 | 0,003016 | 0,004517 | 40 |
| ... | ... | ... | ... |
| 0,001815 | 0,003603 | 0,005418 | 50 |

Tab. 1 Zależność czasu wykonania zadania od liczby wątków

Zgodnie z oczekiwaniami czas wykonywania rośnie liniowo wraz z liczbą utworzonych wątków. Wyjątek stanowi przypadek w którym przyznana zostaje liczba 4 która dla macierzy 5 wymiarowej powoduje nagłe zwiększenie czasu. Fakt, że podobna zależność nie występuje dla macierzy 10 wymiarowej może być powiązany z ilością zrównoleglonych iteracji, tj. 5 iteracji podzielone na 4 zrównoleglone wątki. Teoretycznie proces przebiega w sposób następujący:

- Utworzenie 4 nowych wątków,

- Przeprowadzenie 4 operacji,

- Zamknięcie 4 wątków,

- Otworzenie 4 wątków,

- Wykonanie tylko jednej operacji,

- Zamknięcie 4 wątków.

Sugeruje to stratne zarządzanie zasobami, gdyż w przypadku utworzenia 5 wątków i więcej czas wykonania wraca do oczekiwanej wartości.

Należy jednak zauważyć, iż podobna zależność nie występuje dla 9 wątków i macierzy 10 wymiarowej. Następuje również zastanawiająca poprawa czasu wykonania dla 37 wątków.

* + 1. **Uwagi**

Pomiar czasu zawierał w sobie również proces wczytywania danych oraz zwalniania zasobów. Jako, iż był on wykonywany w każdym z pomiarów nie miał bezpośredniego wpływu na ich prawidłowość. Wyświetlanie wyników zostało jednak dezaktywowane gdyż mocno wpływało na różnicę w pomiarze czasu dla macierzy pięcio i dziesięcio wymiarowych.