Министерство образования и науки Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский энергетический институт»**

**Институт Автоматики и вычислительной техники**

**Кафедра Прикладной математики**

**Разработка и реализация простейших параллельных программ в среде программирования OpenMP. Исследование характеристик эффективности.**

Курсовая работа

студента третьего курса

Жилманова В. Ю.

Научный руководитель:

доцент, к.т.н,

Шамаева О. Ю.

Москва

2016

Оглавление

[1. Постановка задачи. 3](#_Toc468994915)

[2. Стандарт OpenMP 4](#_Toc468994916)

[2.1. Общие сведения об OpenMP 4](#_Toc468994917)

[2.2. Обзор возможностей OpenMP 4](#_Toc468994918)

[2.3. Применение OpenMP в работе 5](#_Toc468994919)

[2.4. Недостатки OpenMP 5](#_Toc468994920)

[3. Анализ баз алгоритмов обработки матриц и их реализация 6](#_Toc468994921)

[3.1. Умножение матриц 6](#_Toc468994922)

[3.1.1. Анализ алгоритма. 6](#_Toc468994923)

[3.1.2. Реализация алгоритма 6](#_Toc468994924)

[3.1.3. Анализ результатов 7](#_Toc468994925)

[3.2. Решение СЛАУ методом Гаусса 8](#_Toc468994926)

[3.2.1. Анализ алгоритма 8](#_Toc468994927)

[3.2.2. Реализация алгоритма 8](#_Toc468994928)

[3.2.3. Анализ результатов 9](#_Toc468994929)

[3.3. Решение СЛАУ методом Якоби 10](#_Toc468994930)

[3.3.1. Анализ алгоритма 10](#_Toc468994931)

[3.3.2. Реализация алгоритма. 10](#_Toc468994932)

[3.3.3. Анализ результатов 11](#_Toc468994933)

[3.3.4 Скриншоты 11](#_Toc468994934)

[3.4. Уравнение теплопроводности 12](#_Toc468994935)

[3.4.1 Постановка задачи 12](#_Toc468994936)

[3.4.2. Описание математической модели 12](#_Toc468994937)

[3.4.3. Реализация алгоритма 13](#_Toc468994938)

[3.4.4. Анализ алгоритма 14](#_Toc468994939)

[3.4.5. Результаты 14](#_Toc468994940)

[4. Заключение 15](#_Toc468994941)

[5. Список используемой литературы 16](#_Toc468994942)

# 1. Постановка задачи.

Изучить стандарт для распараллеливания программ OpenMP.

Разработать параллельные версии основных алгоритмов обработки системам линейных алгебраических уравнений. Теоретически проанализировать разработанные алгоритмы и оценить достигаемое ускорение вычислений за счёт распараллеливания. Сравнить полученные теоретически оценки с реально полученными результатами.

# 2. Стандарт OpenMP

## 2.1. Общие сведения об OpenMP

OpenMP [3] - открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

Первая версия появилась в 1997 году, предназначалась для языка Fortran. Для С/С++ версия разработана в 1998 году. В 2008 году вышла версия OpenMP 3.0.

## 2.2. Обзор возможностей OpenMP

* OpenMP [1] позволяет проводить преобразование уже написанного последовательного кода к параллельному не прибегая к значительным изменениям в программе. Особенно эффективно с помощью OpenMP можно распараллеливать программы, к которых применяются циклы, включающие число итераций значительно превышающее количество используемых ядер.
* OpenMP – гибкий механизм, дающий разработчику большие возможности контроля над программой. Например, можно помечать данные, которые будут общими для всех итераций, а какие будут приватными для каждой итерации. Так *SHARED* применяется к переменным, которые необходимо сделать общими.

*PRIVATE* применяется к переменным, которые необходимо сделать приватными. При входе в параллельную область для каждой нити создается отдельный экземпляр переменной, который не имеет никакой связи с оригинальной переменной вне параллельной области.

* Если процессор, на котором запускается программа с применением OpenMP имеет только одно ядро или по каким-то иным причинам не может использовать много поточность, то программа будет просто работать как последовательная. Это позволяет использовать одну версию программы для нескольких различных систем.
* OpenMP содержит много инструментов синхронизации, которых обычно достаточно для комфортного написания параллельных программ. Например, директива *MASTER ... END MASTER* определяет блок кода, который будет выполнен только master-ом (нулевой нитью). *BARRIER* определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных.
* OpenMP позволяет регулировать число нитей, выполняющих выбранных участок кода. Это число устанавливается с помощью *OMP\_SET\_NUM\_THREADS(n).*

Обычно программа на OpenMP строится из параллельных секций, внедрённых в последовательный код. Именно это и позволяет легко заменять последовательные участки на параллельные при модификации последовательных программ.

## 2.3. Применение OpenMP в работе

С помощью OpenMP мною были распараллелены базовые алгоритмы работы с матрицами и системами линейных математических уравнений. Эти алгоритмы легко поддаются распараллеливанию ввиду наличия несельских вложенных циклов с небольшими общими данными. Это позволяет не затрачивать большие ресурсы на синхронизацию, но замечу, что существует много алгоритмов, которые будет значительно менее удобные для распараллеливая, требующие большой синхронизации и имеющие много общих данных.

## 2.4. Недостатки OpenMP

* OpenMP рассчитан на мультипроцессоры и DSM-системы и не ориентирован на кластеры с распределенной памятью.
* Организация взаимодействия потоков через общие переменные, а не через передачу сообщений, часто приводит к трудно обнаруживаемым ошибкам (race condition – условия гонок), а необходимые для поиска таких ошибок средства отладки – либо отсутствуют вообще, либо мало доступны. [7]

# 3. Анализ баз алгоритмов обработки матриц и их реализация

# 3.1. Умножение матриц

## 3.1.1. Анализ алгоритма.

Умножение матриц [4] – одна из основных операций действия над матрицами. И как следствие алгоритм умножения один из самых известных. Матрица произведения состоит из всех возможных скалярных произведений векторов – строк на векторы – столбцы. Простейшая реализация алгоритма – создание трёх воженных циклов и нахождение произведений i-й строки на j-й столбец. Вычислительная сложность этого алгоритма O(n3).[4]

Данный алгоритм легко поддаётся распараллеливанию, т.к. он не требует синхронизации ни на одном из этапов. Синхронизация не требуется потому, что потоки не изменяет общих данных, а лишь записывают результат вычислений в матрицу произведения.

## 3.1.2. Реализация алгоритма

**double** TimeToMatrixMultiplication(**double** \*\*A, **double** \*\*B, **double** \*\*C)

{

InitMatr(A, B); //Инициализируем матрицы А и B

**double** startTime = omp\_get\_wtime(); //Время начала вычислений

#**pragma omp parallel for //В параллельном цикле вычисляем элементы матрицы С**

**for** (**int** i = 0; i < N; i++)

{

**for** (**int** j = 0; j < N; j++)

{

**for** (**int** k = 0; k < N; k++)

C[i][j] += A[i][k] \* B[k][j];

}

}

**return** omp\_get\_wtime() - startTime; //Возвращаем затраченное время

}

Заметим, что потоки создаются один раз в начале вычисления. Иначе говоря, они создаются за константное время О(1). Это значит, что при большой размерности задачи временные расходы на создание потоков будут минимальны.

## 3.1.3. Анализ результатов

Вычисления выполнены на процессоре AMD phenom 9550 quad core X4.

На графике видно, что чем больше размер матриц, тем больше ускорение от распараллеливания программы. Это и подтверждает размышления прошлого пункта. Так же замечу, что при размерности n >2000, быстродействие программы увеличивается в 4 раза при использовании всех 4х ядер. Большего ускорения на данном процессоре добиться невозможно.

# 3.2. Решение СЛАУ методом Гаусса

## 3.2.1. Анализ алгоритма

Метод Гаусса [4] – один из простейших методов нахождения корней СЛАУ. Он заключается в приведении матрицы коэффициентов к треугольному виду на первом этапе и последующему нахождению корней при обратном проходе. Прямой ход метода Гаусса имеет вычислительную сложность O(n3), обратный ход работает за O(n).

Прямой ход метода Гаусса преобразует исходную матрицу на каждом шаге. В общем случае изменению подвергаются все переменные. Если синхронизировать каждую итерацию по данным, то код будет последовательным на каждом шаге, а за счёт синхронизации накладные расходы будут огромны.

Однако метод Гаусса содержит два вложенных цикла (внутренний вложенный и внешний вложенный) на которых происходит вычитание строк и итерации не изменяют исходные данные для других итераций. Следовательно, их можно выполнять параллельно. Т.к. эти циклы вложены во внешний цикл, то трудоёмкость от создания потоков будет O(n) и O(n2). К тому же на каждой итерации большого цикла число итераций, которые будут распараллеливаться, сокращается и в конце цикл будет состоять из одной итерации, что тоже снизит эффективность распараллеливания.

## 3.2.2. Реализация алгоритма

int Gauss::TrianglNotation() //Распараллеливание на внутренним цикле

{

double mrForStr;

for (int numStr = 1; numStr < sizeMatrix; numStr++)

{

if (abs(Matrix[numStr - 1][numStr - 1]) < esp)

{

if (ReplacingString(numStr)) { return EXIT\_FAILURE; }

}

for (int nIter = numStr; nIter < sizeMatrix; nIter++)

{

if (abs(Matrix[nIter][numStr - 1]) < esp)

{

Matrix[nIter][numStr - 1] = 0;

continue;

}

mrForStr = Matrix[numStr - 1][numStr - 1] / Matrix[nIter][numStr - 1];

RP[nIter] = RP[nIter] \* mrForStr - RP[numStr - 1];

#pragma omp parallel for //Распараллеливание на внутренним цикле

for (int nCol = numStr - 1; nCol < sizeMatrix; nCol++)

{

Matrix[nIter][nCol] = Matrix[nIter][nCol] \* mrForStr

– Matrix[numStr - 1][nCol];

if (abs(Matrix[nIter][nCol]) <= esp) Matrix[nIter][nCol] = 0;

}

}

}

return EXIT\_SUCCESS;

}

int Gauss::TrianglNotation() //Распараллеливание на внешнем цикле

{

double mrForStr;

for (int numStr = 1; numStr < sizeMatrix; numStr++)

{

if (abs(Matrix[numStr - 1][numStr - 1]) < esp)

{

if (ReplacingString(numStr)) { return EXIT\_FAILURE; }

}

#pragma omp parallel for private (multiplierForStr) //Распараллеливание на внешнем цикле

for (int nIter = numStr; nIter < sizeMatrix; nIter++)

{

if (abs(Matrix[nIter][numStr - 1]) < esp)

{

Matrix[nIter][numStr - 1] = 0;

continue;

}

mrForStr = Matrix[numStr - 1][numStr - 1] / Matrix[nIter][numStr - 1];

RP[nIter] = RP[nIter] \* mrForStr - RP[numStr - 1];

for (int nCol = numStr - 1; nCol < sizeMatrix; nCol++)

{

Matrix[nIter][nCol] = Matrix[nIter][nCol] \* mrForStr

– Matrix[numStr - 1][nCol];

if (abs(Matrix[nIter][nCol]) <= esp) Matrix[nIter][nCol] = 0;

}

}

}

return EXIT\_SUCCESS;

}

Вычисления выполнены на процессоре AMD phenom 9550 quad core X4.

## 3.2.3. Анализ результатов

Заметим, что распараллеливание метода Гаусса не приносит эффекта на СЛАУ размерности меньше 100. Что подтверждает предположения, сделанные при анализе алгоритма. Так же заметим, что распараллеливание внешнего цикла ведёт к большему приросту скорости, т.к. затраты на создание потоков на внешнем цикле O(n), тогда как на внутреннем цикле O(n2).

# 3.3. Решение СЛАУ методом Якоби

## 3.3.1. Анализ алгоритма

Метод Якоби [4] – итерационный метод решения СЛАУ. Это значит, что он находит не точный ответ, а ответ с заданной точностью. На каждой итерации его сложность O(n2). Причём итерации вложенного цикла не зависят друг от друга, поэтому можно распараллелить внешний цикл, что даст больший прирост скорости работы. Та же на каждой итерации нужно подставлять полученные значения X в СЛАУ, чтобы узнать погрешность. Эта операция так же требует O(n2) и так же хорошо распараллеливается.

## 3.3.2. Реализация алгоритма.

Непосредственно вычисления корней СЛАУ:

do

{

countIter++;

#pragma omp parallel shared (countIter)

{

#pragma omp for

for (int i = 0; i < sizeMatrix; i++)

{

AnswerI[i] = AnswerIp1[i];

AnswerIp1[i] = 0;

}

#pragma omp barrier

#pragma omp for

for (int i = 0; i < sizeMatrix; i++)

{

for (int j = 0; j < sizeMatrix; j++)

{

if (i != j)

AnswerIp1[i] += AnswerI[j] \* ((-Matrix[i][j]) / Matrix[i][i]);

}

AnswerIp1[i] += RightPath[i] / Matrix[i][i];

}

}

} while (SolutionsError() > esp);

Проверка точности:

double Gauss–Seidel::SolutionsError()

{

double valueI = 0;

double errorI = 0;

double errorMax = 0;

#pragma omp parallel private (errorI, valueI)

{

#pragma omp for

for (int i = 0; i < sizeMatrix; i++)

{

valueI = 0;

for (int j = 0; j < sizeMatrix; j++)

{

valueI += ImputMatrix[i][j] \* AnswerIp1[j];

}

errorI = abs(valueI - InputRigthPath[i]);

#pragma omp critical (error)

{

if (errorI > errorMax)

errorMax = errorI;

}

}

}

return errorMax;

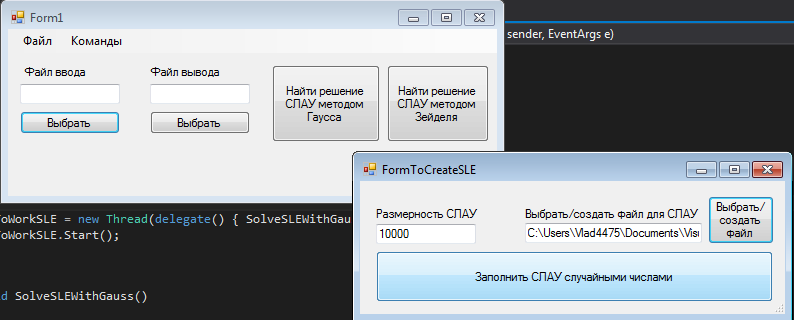
}

В проверке точности введена критическая секция, т.к. в ней сравнивается локальная для каждой нити переменная errorI c глобальной переменной errorMax и так же происходит присвоение значения всё той же глобальной переменной errorMax. Если эта секция не будет критической, то некоторые нити смогут менять значения глобальной переменной одновременно, что может привести к «гонке данных» и ошибочным результатам.

## 3.3.3. Анализ результатов

Метод Якоби хорошо поддаётся распараллеливанию. И чем больше размерность системы, тем лучше она поддаётся распараллеливанию. Разница же между ускорением между плохо и хорошо обусловленными СЛАУ почти отсутствует, хотя плохо обусловленные СЛАУ решаются методом Якоби намного медленнее.

## 3.3.4 Скриншоты



# 3.4. Уравнение теплопроводности

## 3.4.1 Постановка задачи

Пусть имеется пластина заданной формы и известная её температура в начальный момент времени и её теплоёмкость. Так же известна температура отгружающей среды и коэффициент теплопроводности между пластиной и окружающей средой. К каким-то точкам пластины может подводиться/отводиться тепло. Требуется определить, как меняется температура пластины во всех точках в течении времени при этих условиях.

## 3.4.2. Описание математической модели

Исходное ур-е теплопроводности [5] пластины выглядит так:

*, где*

*u – Температура в точке x,y в момент t,*

*x и y – координаты на пластине*

*t – Момент времени*

*ф – функция подачи (отвода тепла) к точке x, y в момент времени t и при температуре u.*

*Граничные условия к этому уравнению будут такими:*

*, где*

*a – коэффициент теплообмена с окружающей средой*

*Tсреды – температура окружающей среды.*

Заменив производные на их численное значение получил дискретное ур-е :

и граничными условиями к нему будут

*u(t=0, x,y) = – начальная температура*

*=*

*=*

*=*

*=*

*где – длина пластины по x*

*– длина пластины по y*

Выражаю и получаю формулу вычисления для t+1 го шага.

Для вычисления t+1 шаге использовать не только t шаг, но и уже вычисленные значение t+1 шага, так что ф-лу можно записать так:

## 3.4.3. Реализация алгоритма

void Plate::EulerIneration(double \*\*lastMatrix, double \*\*nowMatrix)

{

#pragma omp parallel for

for (int i = 1; i < countStepX - 1; i++)

{

for (int j = 1; j < countStepY - 1; j++)

{

nowMatrix[i][j] = LengthSpetT \* ConductivityOfPlait(i, j, lastMatrix[i][j], NumIteration)

\* ((nowMatrix[i - 1][j] - 2 \* lastMatrix[i][j] + lastMatrix[i + 1][j]) / (LenghtStepX \* LenghtStepX)

+ (nowMatrix[i][j - 1] - 2 \* lastMatrix[i][j] + lastMatrix[i][j + 1]) / (LenghtStepY \* LenghtStepY))

+ LengthSpetT \* SupplyHeat(i, j, lastMatrix[i][j], NumIteration)

+ lastMatrix[i][j];

}

}

ShapeBorderCondition(lastMatrix, nowMatrix);

}

void Plate::ShapeBorderCondition(double \*\*lastMatrix, double \*\*nowMatrix)

{

for (int i = 0; i < countStepX; i++)

nowMatrix[i][0] = heatExchangeWithEnvironment \* LengthSpetT \* (ambientTemperature - 2 \* lastMatrix[i][0] + lastMatrix[i][1]) + lastMatrix[i][0];

for (int i = 0; i < countStepX; i++)

nowMatrix[0][i] = heatExchangeWithEnvironment \* LengthSpetT \* (ambientTemperature - 2 \* lastMatrix[0][i] + lastMatrix[1][i]) + lastMatrix[0][i];

for (int i = 0; i < countStepX; i++)

nowMatrix[i][countStepX - 1] = heatExchangeWithEnvironment \* LengthSpetT \* (ambientTemperature - 2 \* lastMatrix[i][countStepX - 1] + lastMatrix[i][countStepX - 2]) + lastMatrix[i][countStepX - 1];

for (int i = 0; i < countStepX; i++)

nowMatrix[countStepY - 1][i] = heatExchangeWithEnvironment \* LengthSpetT \* (ambientTemperature - 2 \* lastMatrix[countStepY - 1][i] + lastMatrix[countStepY - 2][i]) + lastMatrix[countStepY - 1][i];

}

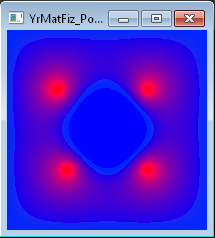
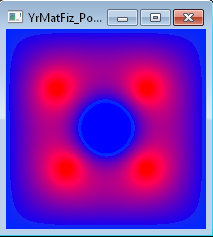
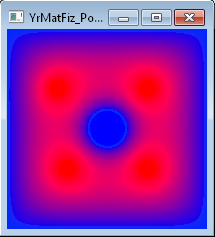
## 3.4.4. Анализ алгоритма

Замечу, что расчёт ведётся покадрово поэтому на каждой итерации обрабатывается n2 ячеек сетки. Обработка происходит в двух циклах, причём один цикл вложен в другой. При этом лучше распараллелить внешний цикл, т.к. это не потребует дополнительных данных и при этом будет тратится меньше времени на создание потоков. Их этого можно сделать вывод, что алгоритм хорошо поддаётся распараллеливанию.

## 3.4.5. Результаты

В ходе эксперимента я брал пластину разных размеров. Эта пластина нагревается в 4ёх точках и охлаждается в одной точке. На экран выводится каждый 1000ый кард. В таблице представлена зависимость времени работы параллельной и последовательной программы от размерности задачи. А так же полученное при этом ускорение от распараллеливания.

|  |  |  |  |
| --- | --- | --- | --- |
| **Размерность** | **Последовательная программа** | **Параллельная программа** | **Ускорение** |
| 50 | 8,6 | 6,6 | 1,303030303 |
| 100 | 18 | 9,5 | 1,894736842 |
| 200 | 58 | 21 | 2,761904762 |
| 300 | 120 | 39 | 3,076923077 |



На скриншотах приведён пример пластины с четырьмя источниками и одном стоком тепла по центру. При этом пластина помещена в холодную среду, поэтому все граничные точки так же являются стоками тепла.

# 4. Заключение

Во время выполнения работы мною была освоен стандарт распараллеливания OpenMP, программный интерфейс OpenGL для создания двумерной графики. Были исследованы простейшие алгоритмы решения систем линейных алгебраических уравнений, а также задача теплопроводности пластины.

Проведено сравнение априорного анализа алгоритмов и данных полученных экспериментально и на основании этого можно сделать следующие выводы:

* нецелесообразно использовать параллельные вычисления при решении задач с малой размерностью.
* от уровня распараллеливания зависит и ускорение от распараллеливания и объём дополнительной памяти, поэтому перед написание параллельной программы нужно исследовать задачу и определить оптимальное решение.

Во всех задачах при увеличении размерности ускорение от распараллеливания стремилось к количеству доступных ядер, что говорит о том, что выбранные алгоритмы хорошо поддаются распараллеливанию. Однако анализ показывает, что при очень больших размерностях ускорение должно уменьшаться из-за увеличение расходов на работу с общей памятью.

# 5. Список используемой литературы

1. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие. – М.: Изд-во МГУ, 2009. – 77с.
2. Баландин М.Ю., Шурина Э.П. Методы решения СЛАУ большой размерности. – Новосибирск: Изд-во НГТУ, 2000. – 70 с.
3. Шамаева О.Ю., Чернецов А.М. Основы параллельного программирования с использованием технологий OpenMP и MPI. Учебное пособие. – М.: Издательский дом МЭИ, 2016. – 84 с..
4. Амосов А.А., Дубинский Ю.А., Копченова Н.В. Вычислительные методы для инженеров. Учеб. пособие. – М.: Высш. шк., 1994. -544 с.
5. Петровский И. Г. Лекции об уравнениях с частными производными. — гл. IV, § 40.
6. Официальный сайт OpenMP Architecture Review Board.
7. В.А. Бахтин, Н.А. Коновалов, Н.В. Поддерюгина, С. Д. Устюгов. Гибридный способ программирования DVM/OpenMP на SMP-кластерах. М.: Изд-во МГУ, 2006 г., стр. 128-130.