МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра математического моделирования и управления

КИРКОРОВА ЛЮБОВЬ СЕРГЕЕВНА

ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ ЛОКАЛЬНОСТИ АЛГОРИТМОВ ПЕРЕМНОЖЕНИЯ МАТРИЦ, РЕАЛИЗОВАННЫХ НА ДВУХЪЯДЕРНЫХ КОМПЬЮТЕРАХ

Курсовая работа
студентки 3 курса 6 группы

“Допустить к защите“
с предварительной оценкой ____
Руководитель работы
_____

Руководитель
Лиходед Николай Александрович
доктор физико-математических наук, профессор

“___” _____ 2008 г

Минск 2008

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра математического моделирования и управления

"**Утверждаю**"
Заведующий кафедрой
_____И.В. Гайшун
"___" _____ 2007 г.

ЗАДАНИЕ
ПО ПОДГОТОВКЕ КУРСОВОЙ РАБОТЫ

Студенту 3 курса

1. Тема работы ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ ЛОКАЛЬНОСТИ АЛГОРИТМОВ ПЕРЕМНОЖЕНИЯ МАТРИЦ, РЕАЛИЗОВАННЫХ НА ДВУХЪЯДЕРНЫХ КОМПЬЮТЕРАХ

2. **Срок сдачи студентом законченной работы** __ мая 2008 г.

3. **Исходные данные к работе**

1 Способы улучшения локальности многомерных циклов (перестановка циклов, тайлинг) ([1, 2]).
2 Экспериментальные данные, демонстрирующие влияние локализации данных на время выполнения алгоритмов перемножения матриц, реализованных на одноядерном компьютере ([1], рабочие материалы).
3 Технические требования к электронным версиям отчетных документов.

Библиографические описания источников, рекомендуемых студентам к ознакомлению при выполнении работы:

1. Лиходед Н.А. Методы распараллеливания гнезд циклов: Курс лекций. - Мн.: БГУ. 2007. - 100 с. serv314\subFaculty\Каф. Дискр.мат. и алгор\ КУРСЫ ДМА\ 4 курс\ Лиходед\ Лекции \ распараллеливание гнезд циклов
2. Ahmed N., Mateev N., Pingali K. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. Proceedings of the International Conference on Supercomputing. 2000. P. 141--152.

4. **Перечень вопросов, подлежащих разработке или краткое содержание работы**

1 Ознакомиться с теоретическими основами улучшения локальности многомерных циклов ([1, 2]).
2 Изучить экспериментальные данные, демонстрирующие влияние локализации данных на время выполнения на одноядерном компьютере алгоритмов перемножения матриц, ([1], рабочие материалы).
3 Провести численные эксперименты на двухъядерном компьютере.
4 Проанализировать численные эксперименты и исследовать влияние локализации данных на время выполнения на двухъядерном компьютере алгоритмов перемножения матриц.
5 Оформить результаты. Составить компьютерную презентацию к докладу работы на защите.

5. **Перечень графического материала**

1 Логотип БГУ для включения на слайды презентации.
2 Графики и таблицы влияния локализации данных на время выполнения алгоритмов.

6. **Дата выдачи задания** __ октября 2007 г.

7. **Календарный график** работы на весь период (с указанием этапов работы и сроков их выполнения)

1 **октябрь-ноябрь** – изучение основных теоретических вопросов, изучение экспериментальных данных;
2 **ноябрь-декабрь** – изучение технических требований к электронным отчетным документам (doc, ppt) и освоение правил, как их реализовать;
3 **декабрь-апрель** – программирование, практическая реализация задач работы;
4 **апрель-май** – оформление результатов работы (отчета DOC, презентации PPT, подготовка доклада и отладка презентации на защиту);
5 **май –** сдать работу руководителю не менее чем за 7 дней до даты защиты;
6 **май –** выступление с докладом, защита работы на семинаре кафедры.

Руководитель _____ / Н.А. Лиходед / …. октября 2007 г.

Задание принял к исполнению _____ …. октября 2007 г.
(подпись студента)

АННОТАЦИЯ

В данной курсовой работе рассматриваются на примере алгоритма перемножения матриц методы улучшения локальности многомерных алгоритмов: перестановка циклов и тайлинг.

ANNOTATION

In the given course work methods of improvement of localization of multivariate algorithms are considered on an example of algorithm of multiplication of matrixes: rearrangement of cycles and tiling.

АНАТАЦЫЯ

У дадзенай курсавой рабоце даследуюцца на прыкладзе алгарытма перамнажэння матрыц метады паляпшэння лакальнасці шматмерных алгарытмаў: перастаноўка цыклаў і тайлінг.

РЕФЕРАТ

Курсовая работа, 29 с., 8 рис., 2 таблицы, 2 источника, 1 приложение.

ПРОСТРАНСТВЕННАЯ И ВРЕМЕННАЯ ЛОКАЛИЗАЦИЯ ДАННЫХ, ПЕРЕСТАНОВКА ЦИКЛОВ, ВЕКТОР ЛОКАЛЬНОСТИ, ТАЙЛИНГ

Объект исследования –алгоритмы, заданные многомерными циклами; алгоритмы перемножения квадратных матриц.

Цель работы – экспериментально исследовать методы улучшения локализации данных.

Метод исследования – преобразование циклов: перестановка циклов, тайлинг.

Результатами работы являются численные эксперименты на двухъядерном компьютере, визуализация зависимости времени работы алгоритмов от размера матрицы и от размера тайлов, сравнение алгоритмов до и после улучшения локальности.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время, в век бурно развивающихся технологий, большую роль в жизни человека играет решение различных вычислительных задач на компьютерах. И во многих случаях важна скорость решения каких-либо задач вычислительного типа.

Время решения задачи на современном компьютере определяется не столько скоростью выполнения вычислительных операций, сколько скоростью доступа к памяти. Скорость доступа существенно зависит от места в иерархической памяти (диск, оперативная память, кэш второго уровня, кэш первого уровня, регистры), где хранятся данные. Поэтому для быстрой реализации алгоритма, заданного последовательной программой, задача быстрого (до исчезновения из памяти с быстрым доступом) переиспользования данных является одной из важнейших. То есть важной задачей является задача улучшения локальности алгоритмов.

Для параллельных программ эта задача тоже важна, так как и в этом случае она помогает увеличить скорость вычислений.

В данной курсовой работе рассматриваются методы улучшения локальности многомерных циклов: перестановка циклов и тайлинг. Экспериментальные данные получены с помощью программы, написанной на языке C++ на одноядерном и на двухъядерном компьютерах.

Отметим, что перестановку циклов и тайлинг можно выполнить не всегда. Исследование условий, при которых эти преобразования циклов сохраняют корректное выполнение программы, не входило в задачу курсовой работы.

ОСНОВНАЯ ЧАСТЬ

1 Постановка задачи

Необходимо выполнить следующие задания:

1. Ознакомиться с теоретическими основами улучшения локальности многомерных циклов.
2. Получить и изучить экспериментальные данные, демонстрирующие влияние локализации данных на время выполнения алгоритмов перемножения матриц на одноядерном компьютере.
3. Провести численные эксперименты на двухъядерном компьютере.
4. Проанализировать численные эксперименты и исследовать влияние локализации данных на время выполнения алгоритмов перемножения данных на двухъядерном компьютере.
5. Оформить результаты в виде графиков и/или таблиц влияния локализации данных на время выполнения алгоритмов.

2 Улучшение локальности программы. Пространственная и временная локальность данных

Под *улучшением локальности программы* понимается такое ее преобразование, которое позволяет процессору быстро переиспользовать данные (элементы массивов) на большем числе итераций, чем до преобразования. Улучшение локальности программы называют также *локализацией данных*.

Различают задачи улучшения временной локальности и пространственной локальности. *Временная локальность* требует установить такой порядок выполнения операций, при котором данные использовались бы повторно прежде, чем они будут перемещены из памяти с быстрым доступом в память более низкого уровня. Желательно использовать одни и те же данные на всех итерациях самого внутреннего цикла. *Пространственная локальность* предполагает использование данных, расположенных в памяти близко к недавно использованным, а следовательно, расположенных с большой вероятностью в памяти с быстрым доступом.

Дадим более формальное определение локальности данных по времени.

Элементы массива, связанные с выбранным вхождением в некоторый оператор, *локальны по времени*, если индексы этих элементов не зависят от параметра самого внутреннего цикла. Таким образом, при фиксированных значениях параметров каждое данное, локальное по времени, участвует в вычислениях на всех итерациях самого внутреннего цикла.

Следует заметить, что временная локальность элементов массива не зависит от способа размещения (по строкам, по столбцам, развернутый в одну строку) массива в памяти и сохраняется при переразмещении массива.

Теперь дадим более формальное определение пространственной локальности данных.

Элементы массива, связанные с выбранным вхождением в некоторый оператор, *локальны пространственно*, если при увеличении параметра самого внутреннего цикла на единицу получим элемент массива, близко следующий в памяти за элементом массива до увеличения параметра. Определение временной локальности не является строгим, поскольку требует уточнения понятия близкого расположения в памяти. Будем считать, что два элемента расположены в памяти близко, если при перемещении первого элемента в память более высокого уровня перемещается также и второй элемент.

Для пространственной локализации необходимо, чтобы операции алгоритма, которые используют соседние с точки зрения хранения в памяти элементы массива данных, выполнялись бы друг за другом. Если хранение элементов массива осуществляется по строкам (как, например, в языке программирования С), то близко расположенными в памяти элементами массива являются элементы, отличающиеся только последней координатой в индексных выражениях.

3 Вектор локальности

Характеризовать локальность гнезд циклов можно с помощью так называемых *векторов локальности*.

Пусть a – массив размерности большей или равной двум, хранение элементов массива в памяти компьютера осуществляется по строкам. Рассмотрим данные $a(F(J)), J \in V$, используемые на q-ом вхождении массива a в оператор S, где a – некоторый массив с данными, встречающийся в операторе S; F – некоторая функция, которая описывает индексы элементов массива данных a, относящегося к q-ому вхождению элементов этого массива в оператор S; J – итерация цикла; V - область итераций, то есть область изменения параметров гнезда циклов для оператора S.

Теперь определим вектор $\lambda = (\lambda_0, \lambda_1)$, где λ_0 – число внутренних циклов, на итерациях которых используется элемент массива a со всеми фиксированными индексами (то есть используется только один элемент массива); λ_1 – число внутренних циклов, на итерациях которых используются элементы массива a с одним последним переменным индексом (если a – двумерный массив, то используется одна строка). Вектор λ называется вектором локальности данных $a(F(J)), J \in V$. Первая компонента характеризует временную локальность, вторая – пространственную.

Для оптимизации памяти необходимо, чтобы как можно больше векторов локальности λ были ненулевыми и значения координат были как можно больше.

4 Улучшение локальности многомерных алгоритмов с помощью перестановки циклов (на примере алгоритма перемножения матриц)

В данной курсовой работе будем рассматривать алгоритм перемножения квадратных матриц порядка тысяча. Основная часть алгоритма представляет собой тройной цикл по i, j, k с одним оператором S1:

```
int count = 1000;
for (int i = 0; i < count; i++)
        for (int j = 0; j < count; j++)
                for (int k = 0; k < count; k++)
                        S1: c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Перестановкой циклов этого алгоритма можно получать различные модификации данного алгоритма. Всего здесь три цикла, значит существует $3!$, то есть шесть, модификаций алгоритма перемножения квадратных матриц.

Рассмотрим на примере этого алгоритма вектор локальности и исследуем, какая из шести модификаций данного алгоритма является наилучшей.

По постановке задачи и алгоритму имеются три массива a, b и c размерности два. Хранение элементов массива в памяти компьютера осуществляется по строкам, так как для реализации алгоритма на компьютере используется язык программирования C++ именно с таким представлением данных в памяти. В данном случае итерация J представляет собой $J=(i,j,k)$. Область итераций $V=\{(i,j,k)\in \mathrm{Z} \mid 0\le i,j,k<1000\}$. J и V определяются для единственного оператора $S1: c[i][j]=c[i][j]+a[i][k]*[k][j]$.

Теперь определим вектор локальности $\lambda=(\lambda_0,\lambda_1)$ для каждого из массивов a, b и c, то есть определим векторы $\lambda^a=(\lambda_0^a,\lambda_1^a)$, $\lambda^b=(\lambda_0^b,\lambda_1^b)$ и $\lambda^c=(\lambda_0^c,\lambda_1^c)$, где, как уже говорилось в предыдущем пункте, $\lambda_0^i, i=\{a,b,c\}$ – число внутренних циклов, на итерациях которых используется элемент i-го массива со всеми фиксированными; $\lambda_1^i, i=\{a,b,c\}$ – число внутренних циклов, на итерациях которых используются элементы i-го массива с одной последней переменной строкой, так как все массивы в данной задаче – двумерные. Векторы локальности представлены в таблице 1:

	ijk	ikj	jki	jik	kji	kij
λ^c	(1,2)	(0,2)	(0,0)	(1,1)	(0,0)	(0,1)
λ^a	(0,2)	(1,2)	(0,0)	(0,1)	(0,0)	(1,1)
λ^b	(0,0)	(0,1)	(1,1)	(0,0)	(1,2)	(0,2)

Таблица 1 – Векторы локальности для различных модификаций алгоритма перемножения матриц

Как уже говорилось выше, для оптимизации памяти необходимо, чтобы как можно больше векторов локальности λ были ненулевыми и значения координат были бы как можно большими.

Анализируя таблицу 1, несложно увидеть, что наилучшей модификацией (с точки зрения времени реализации) алгоритма является алгоритм с последовательностью циклов ikj, а наихудшими – jki и kji.

Экспериментальные данные были получены на двух компьютерах: одноядерном Intel Pentium 4531, 3000MHz, 1 GB RAM и двухъядерном Intel Core2Duo 6420, 2,13 GHz, 2 GB RAM. В данном случае использовалась матрица размера тысяча на тысячу, программа написана на языке C++. В таблице 2 продемонстрированы полученные данные:

	ijk	ikj	jki	jik	kji	kij
λ^c	(1,2)	(0,2)	(0,0)	(1,1)	(0,0)	(0,1)
λ^a	(0,2)	(1,2)	(0,0)	(0,1)	(0,0)	(1,1)
λ^b	(0,0)	(0,1)	(1,1)	(0,0)	(1,2)	(0,2)
Intel Pentium 4531	25484	12063	32265	21031	44547	12813
Intel Core2Duo 6420	20641	11578	23953	15797	32063	12360

Таблица 2 – Векторы локальности и экспериментальные данные (время вычислений в миллисекундах) для различных модификаций алгоритма перемножения матриц

Из таблицы 2 видно, что векторы локальности в основном верно описывают то, насколько хорошо происходит оптимизация памяти. Исключение: по векторам локальности алгоритм с последовательностью циклов jki должен работать хуже всего, а на практике получилось, что это цикл kji. Для объяснения этого эксперимента требуются более точные исследования эффективности использования иерархической памяти.

Однако мы видим, что локализация данных сильно влияет на время работы программы, потому что самая быстрая и самая медленная модификации алгоритмов могут отличаться раза в два-три и больше для задач большего размера, а это довольно существенная разница, особенно в тех случаях, когда бывает важна скорость работы.

Вышеописанные исследования проводились для квадратной матрицы порядка тысяча. Однако интересным является проверка на практике, насколько полученная закономерность выполняется, если эксперимент проводить на матрицах разной размерности. Проведем исследования для квадратных матриц порядка от ста до тысяча девятисот с шагом сто. Сравним работу двух

модификаций алгоритма – ijk, потому что его чаще всего используют, и самый быстрый по экспериментальным и теоретическим данным ikj. Результаты работы представлены на рисунках 1 – 4:

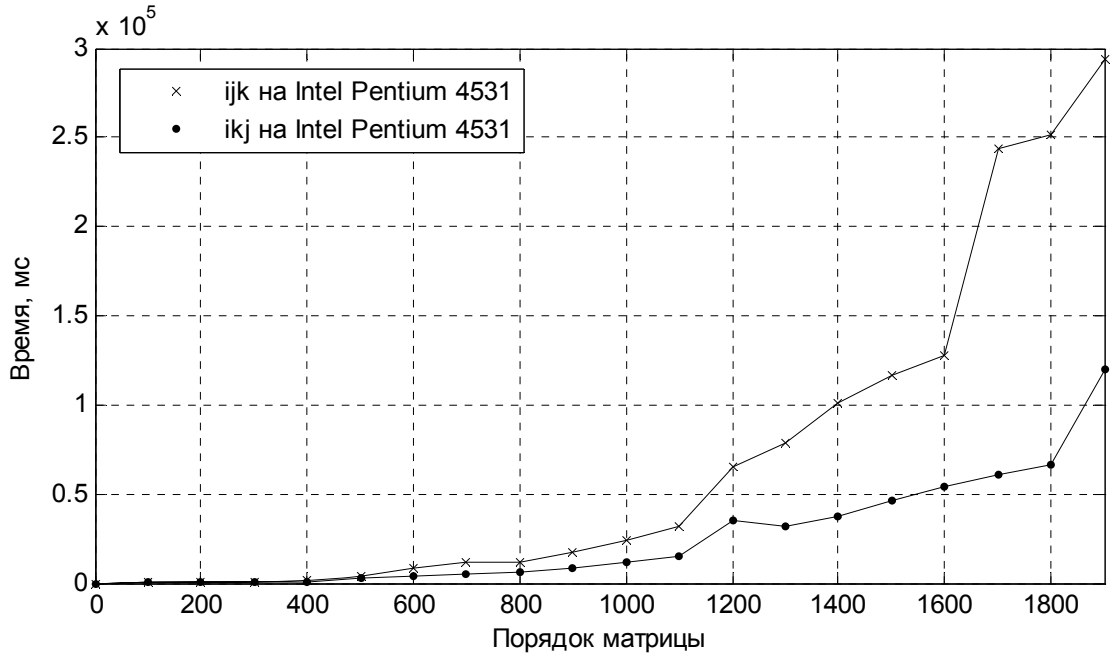

Рисунок 1 – Зависимость времени выполнения программы от размерности матрицы для алгоритма с модификациями ijk и jki на Intel Pentium 4531

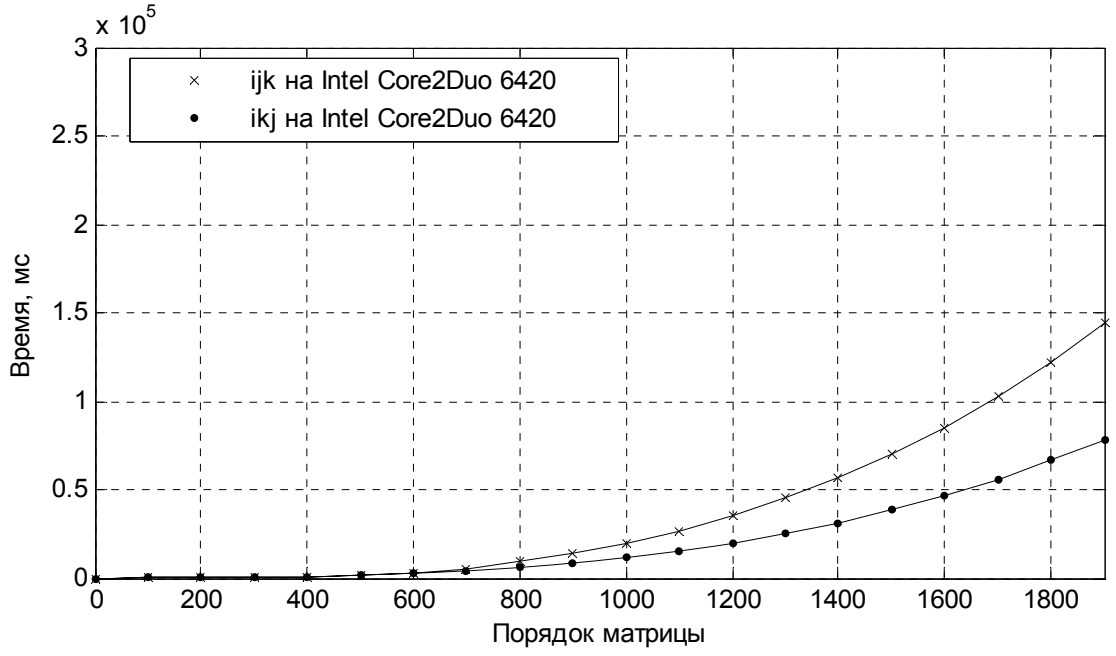

Рисунок 2 – Зависимость времени выполнения программы от размерности матрицы для алгоритма с модификациями ijk и jki на Intel Core2Duo 6420

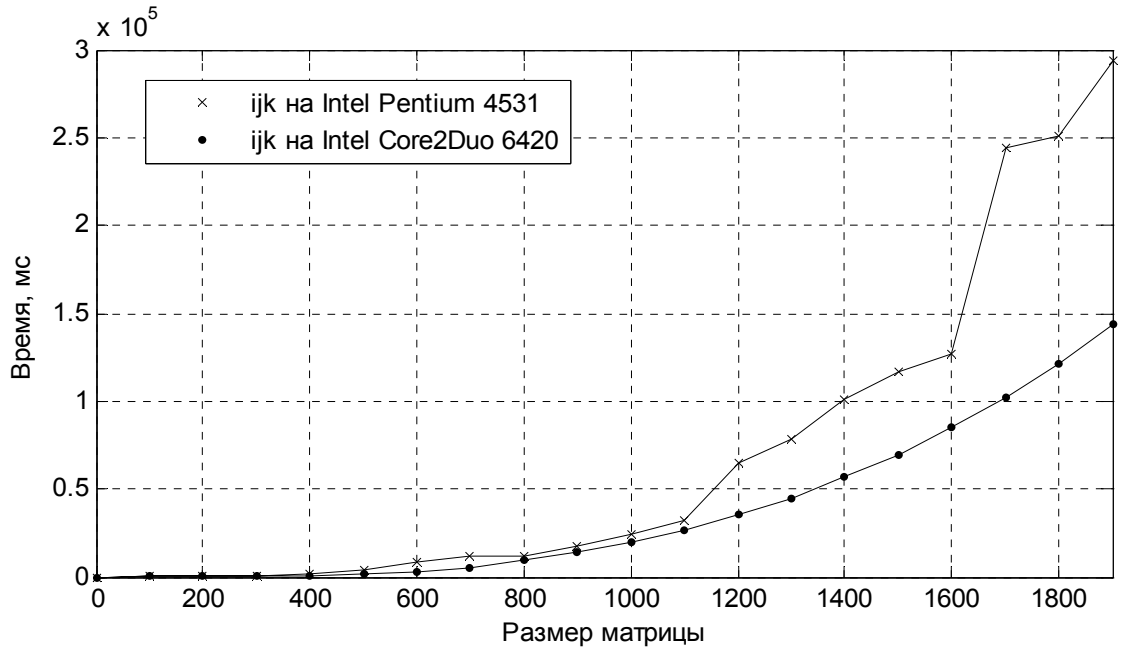

Рисунок 3 – Зависимость времени выполнения программы от размерности матрицы для алгоритма с модификацией ijk на Intel Pentium 4531 и на Intel Core2Duo 6420

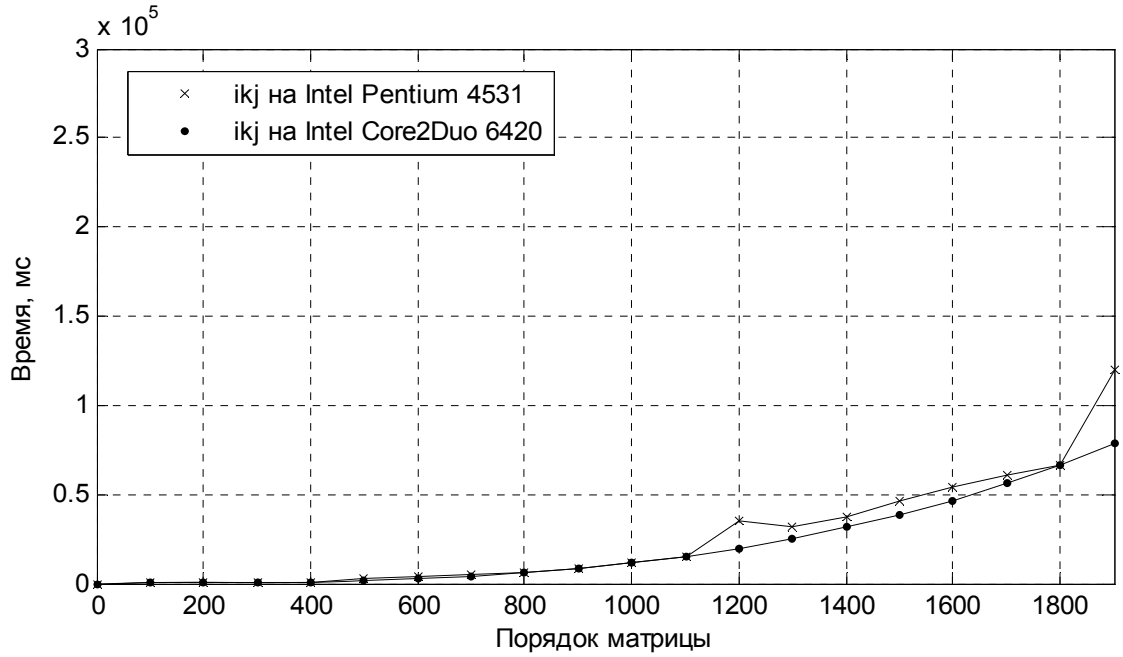

Рисунок 4 – Зависимость времени выполнения программы от размерности матрицы для алгоритма с модификацией jki на Intel Pentium 4531 и на Intel Core2Duo 6420

Таким образом, мы видим, что закономерность выполняется, даже если запускать программу при разных значениях размера матрицы. Однако при увеличении порядка матрицы время выполнения алгоритма на двухъядерном компьютере меняется без скачков значений времени, в отличие от

одноядерного, где видны небольшие скачки. Они проявляются в большей степени при большем порядке матрицы.

Это происходит, по-видимому, вследствие того, что двухъядерному компьютеру не надо переключаться на выполнение сторонних задач, так как время процессора выделяется для выполнения одного потока команд. А для одноядерного компьютера процессорное время распределяется между несколькими потоками команд.

5 Улучшение локальности многомерных алгоритмов с помощью тайлинга

Целью тайлинга является уменьшение накладных расходов на использование иерархической памяти процессора. Под *тайлом* понимается множество операций алгоритма, выполняемых как одна единица вычислений, атомарно. Геометрически тайл можно представить множеством точек многомерной целочисленной решетки. Наиболее частой для использования является форма прямоугольного параллелепипеда.

Исследуем, как влияет применение тайлинга на время выполнения алгоритмов с разной степенью локализованности данных. Рисунки 5, 6, приведенные ниже, демонстрируют зависимость времени выполнения от размера тайлов:

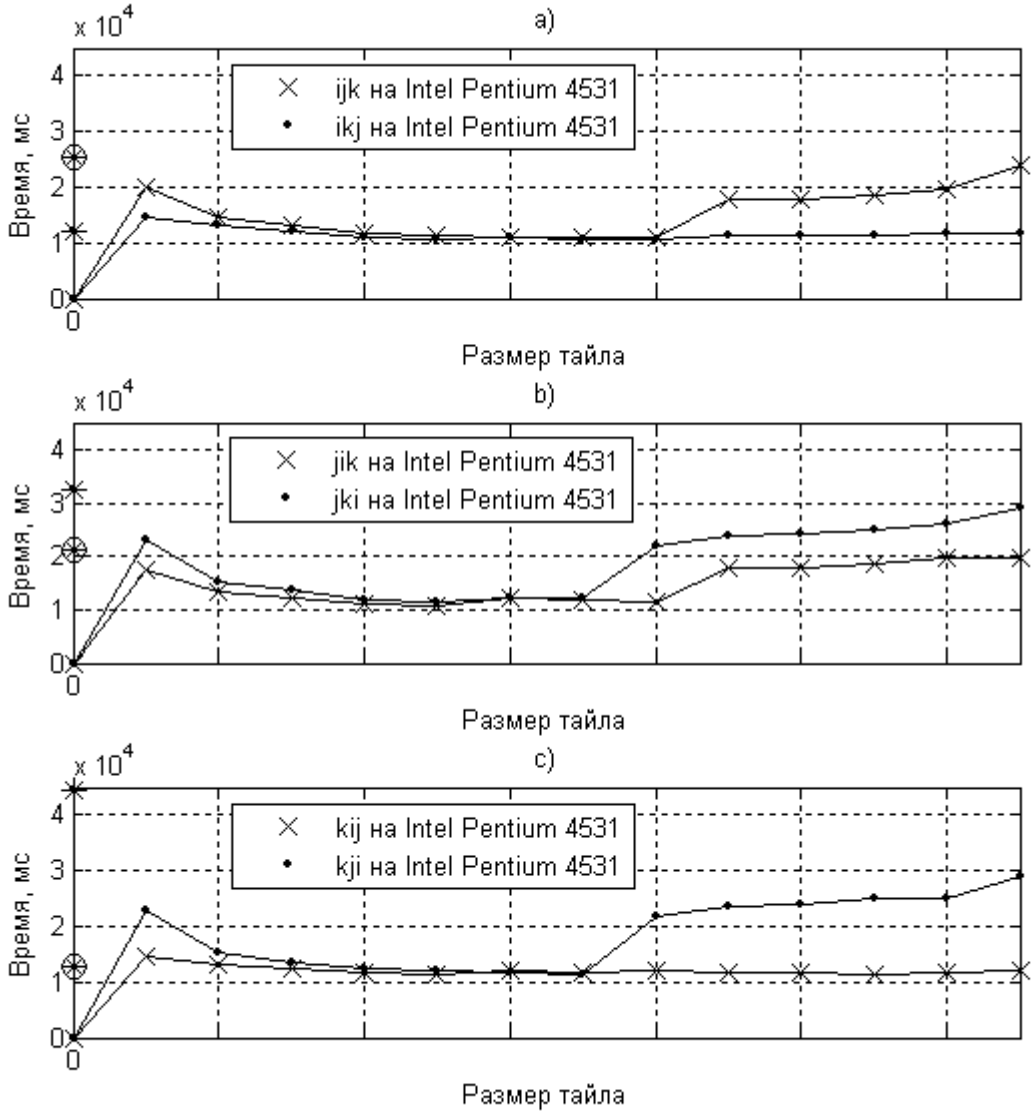

Рисунок 5 – Зависимость времени выполнения программы от размера тайла для алгоритма с различными модификациями на Intel Pentium 4531

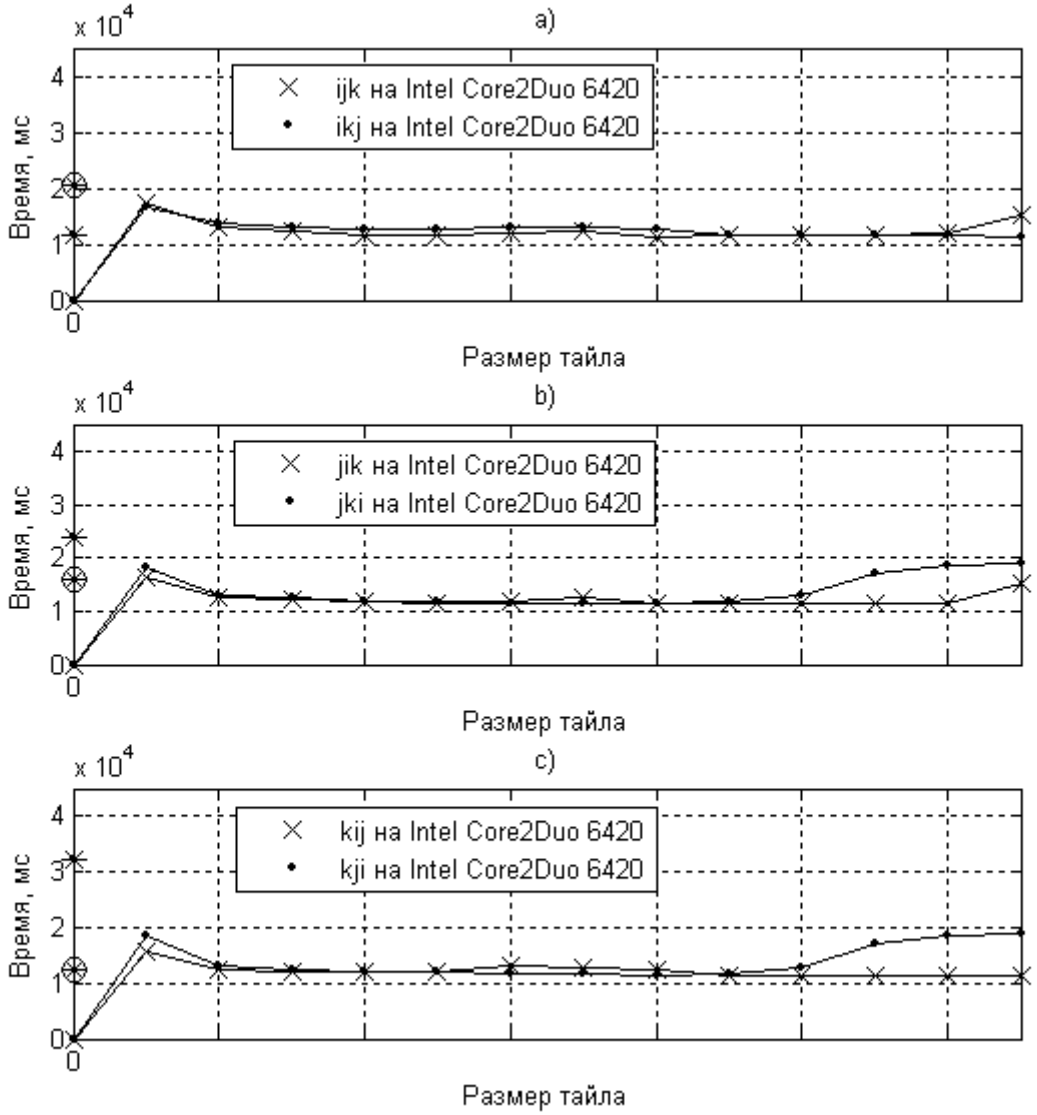

Рисунок 6 – Зависимость времени выполнения программы от размера тайла для алгоритма с различными модификациями на Intel Core2Duo 6420

Звездочками (для графиков, описывающих тайлинг с перестановками циклов *ijk* , *jik* и *kij*) и звездочками в кружочках (*ikj* , *jki* и *kji*) на рисунках 5, 6 обозначено время, которое было получено при перемножении квадратной матрицы порядка тысяча, при различных модификациях алгоритма.

Форма тайла выбрана кубическая, как наиболее простой вариант. Оптимальный размер тайла определим экспериментально, попробовав различные варианты: от двух до пятисот. Матрица является квадратной, порядок ее тысяча. Программа, как и в предыдущем случае, выполнялась на двух компьютерах.

Анализируя на рисунки 5, 6, легко видеть, что оптимальный размер тайла для матрицы порядка тысяча находится приблизительно между десятью и пятьюдесятью. Эта часть графика представлена более крупно на рисунках 7, 8:

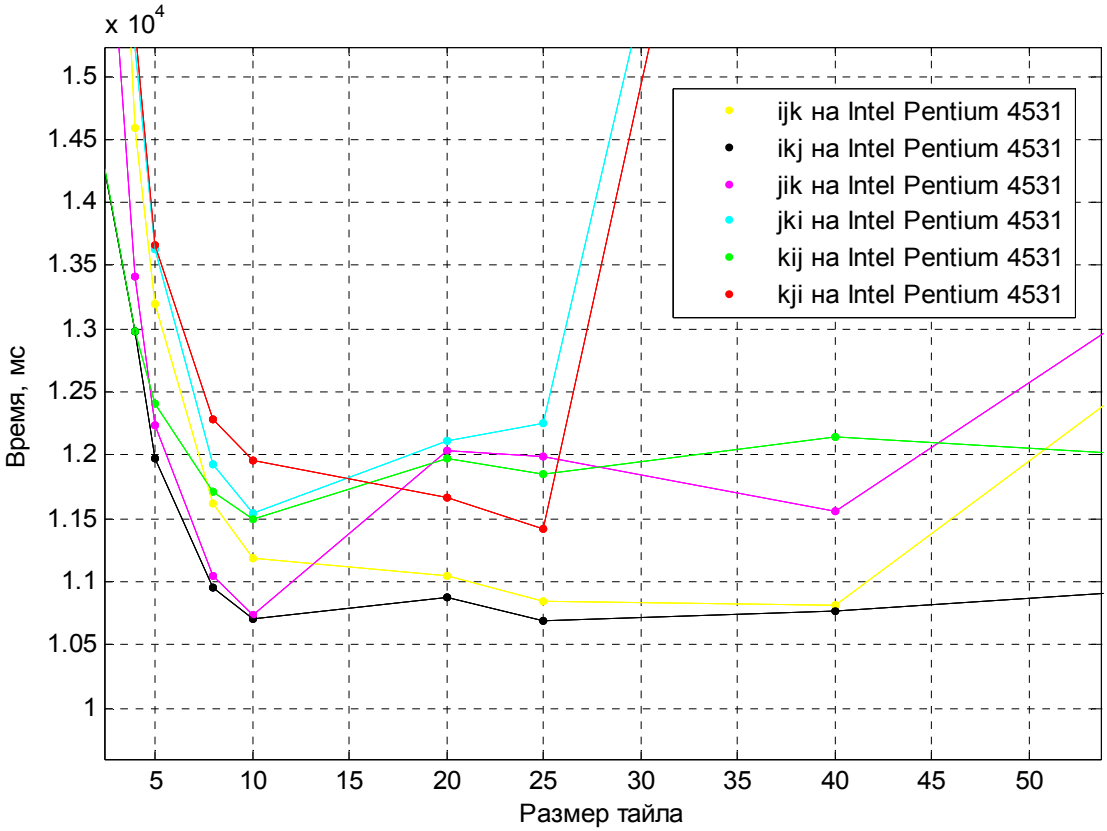

Рисунок 7 – Зависимость времени выполнения программы от размера тайла для алгоритма с различными модификациями на Intel Pentium 4531

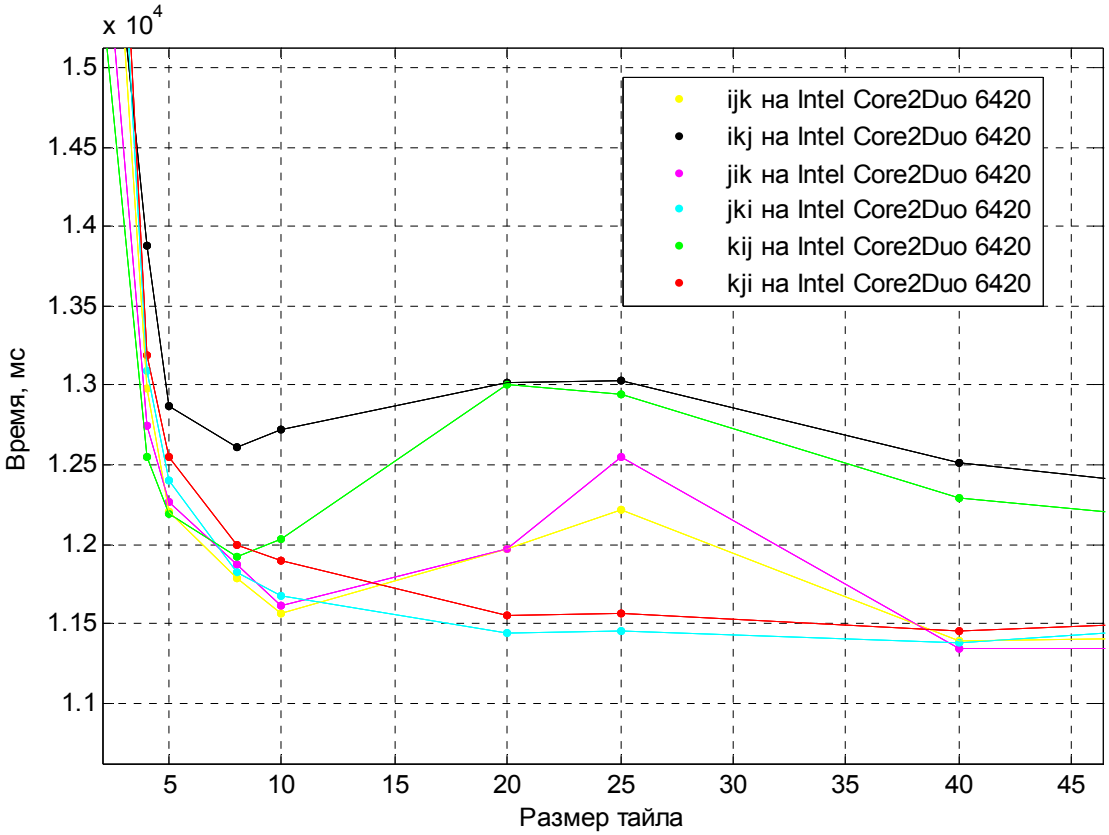

Рисунок 8 – Зависимость времени выполнения программы от размера тайла для алгоритма с различными модификациями на Intel Core2Duo 6420

Анализируя рисунки 5, 6, можно сравнить результаты работы алгоритма перемножения квадратных матриц для различных модификаций алгоритма: с помощью перестановки циклов и с помощью тайлинга. Можно видеть, что использование тайлинга для улучшения локализации алгоритма значительно повышает скорость расчетов. Из рисунков 5, 6 видно, что при некоторых значениях размера тайлов скорость вычислений в алгоритмах с различными модификациями почти совпадает.

Из рисунков 7, 8 видно, что при разной последовательности циклов в алгоритме оптимальный размер тайла не одинаков для всех случаев. В каждом случае он разный. Поэтому для каждой модификации алгоритма при использовании тайлинга необходимо экспериментально подбирать его размер. И тогда время вычислений будет улучшаться.

Также из рисунков 5, 6 можно видеть, что алгоритм перемножения матриц работает быстрее на двухъядерном компьютере. Как и в случае простой перестановки циклов это связано с тем, что двухъядерному компьютеру не надо переключаться на выполнение сторонних задач.

Таким образом, мы видим, что от размера тайла существенно зависит скорость работы программы. При разном размере тайла скорость также различна.

ЗАКЛЮЧЕНИЕ

Таким образом, в данной курсовой работе были рассмотрены задачи, касающиеся улучшения локализации данных. Были рассмотрены теоретические основы, а также получены экспериментальные данные для двух способов улучшения локализации данных: перестановки циклов, а также тайлинг.

При перестановке циклов в основной части алгоритма рассматривалась зависимость времени от размера матрицы. А при тайлинге – зависимость времени от размера тайлов. Экспериментальные данные были получены на двух компьютерах: одноядерном и двухъядерном.

Оказалось, что и перестановка циклов, и тайлинг существенно влияют на скорость работы алгоритма.

При увеличении порядка матрицы время выполнения алгоритма на двухъядерном компьютере меняется без скачков значений времени, в отличие от одноядерного, где видны небольшие скачки. Они проявляются в большей степени при большем порядке матрицы. Это происходит вследствие того, что двухъядерному компьютеру не надо переключаться на выполнение сторонних задач, так как время процессора выделяется для выполнения одного потока команд. А для одноядерного компьютера процессорное время распределяется между несколькими потоками команд.

Использование тайлинга для улучшения локализации алгоритма значительно повышает скорость расчетов. При разной последовательности циклов в алгоритме оптимальный размер тайла не одинаков для всех случаев. В каждом случае он разный. Поэтому для каждой модификации алгоритма при использовании тайлинга необходимо экспериментально подбирать его размер. И тогда время вычислений будет улучшаться.

Таким образом, при использовании перестановки циклов и тайлинга (с оптимальным размером тайла, выбранным с помощью полученных экспериментальных данных), можно значительно улучшить скорость работы алгоритма.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Лиходед Н.А. Методы распараллеливания гнезд циклов: Курс лекций. – Мн.: БГУ. 2007. – 100 с. Ser314\ subFaculty\ Каф. Дискр. мат. и алгор\ КУРСЫ ДМА\ 4 курс\ Лиходед\ Лекции\ Распараллеливание гнезд циклов
2. Ahmed N., Mateev N., Pingali K. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. Proceedings of the International Conference on Supercomputing. 2000. P. 141–152.

ПРИЛОЖЕНИЕ А

Листинг программы, написанной на языке C++

```
#include <iostream.h>
#include <fstream.h>
#include <windows.h>

void matr_multipl_IJK(double **a, double **b, double **c, int count)
{
	for (int i = 0; i < count; i++)
	{
		for (int j = 0; j < count; j++)
		{
			c[i][j] = 0;
			for (int k = 0; k < count; k++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
	}
}

void matr_multipl_IKJ(double **a, double **b, double **c, int count)
{
	for (int i = 0; i < count; i++)
	{
		for (int j = 0; j < count; j++)
			c[i][j] = 0;
		for (int k = 0; k < count; k++)
		{
			for (j = 0; j < count; j++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
	}
}

void matr_multipl_JIK(double **a, double **b, double **c, int count)
{
	for (int j = 0; j < count; j++)
	{
		for (int i = 0; i < count; i++)
		{
			c[i][j] = 0;
			for (int k = 0; k < count; k++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
	}
}

void matr_multipl_JKI(double **a, double **b, double **c, int count)
{
	for (int j = 0; j < count; j++)
	{
		for (int i = 0; i < count; i++)
			c[i][j] = 0;
		for (int k = 0; k < count; k++)
		{
			for (i = 0; i < count; i++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
	}
}
```

```
void matr_multipl_KIJ(double **a, double **b, double **c, int count)
{
      for (int i = 0; i < count; i++)
            for (int j = 0; j < count; j++)
                  c[i][j] = 0;

      for (int k = 0; k < count; k++)
      {
            for (i = 0; i < count; i++)
            {
                  for (int j = 0; j < count; j++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
      }
}

void matr_multipl_KJI(double **a, double **b, double **c, int count)
{
      for (int i = 0; i < count; i++)
            for (int j = 0; j < count; j++)
                  c[i][j] = 0;

      for (int k = 0; k < count; k++)
      {
            for (int j = 0; j < count; j++)
            {
                  for (i = 0; i < count; i++)
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
      }
}

void matr_multipl_IJKBL(double **a, double **b, double **c, int count, int tile)
{
      int tilesize = count/tile;
      for (int i1 = 0; i1 < tilesize; i1++)
      {
            int i1begin = i1*tile;
            int i1end = (i1+1)*tile - 1;

            for (int j1 = 0; j1 < tilesize; j1++)
            {
                  int j1begin = j1*tile;
                  int j1end = (j1+1)*tile - 1;

                  for (int it = i1begin; it <= i1end; it++)
                        for (int jt = j1begin; jt <= j1end; jt++)
                              c[it][jt] = 0;

                  for (int k1 = 0; k1 < tilesize; k1++)
                  {
                        int k1begin = k1*tile;
                        int k1end = (k1+1)*tile - 1;

                        for (int it = i1begin; it <= i1end; it++)
                              for (int jt = j1begin; jt <= j1end; jt++)
                                    for (int kt = k1begin; kt <= k1end; kt++)
                                          c[it][jt] = c[it][jt] +
a[it][kt]*b[kt][jt];
                  }
            }
      }
}
```

```
void matr_multipl_IKJBL(double **a, double **b, double **c, int count, int tile)
{
      int tilesize = count/tile;
      for (int i1 = 0; i1 < tilesize; i1++)
      {
            int i1begin = i1*tile;
            int i1end = (i1+1)*tile - 1;

            for (int it = i1begin; it <= i1end; it++)
                  for (int jt = 0; jt < count; jt++)
                        c[it][jt] = 0;

            for (int k1 = 0; k1 < tilesize; k1++)
            {
                  int k1begin = k1*tile;
                  int k1end = (k1+1)*tile - 1;

                  for (int j1 = 0; j1 < tilesize; j1++)
                  {
                        int j1begin = j1*tile;
                        int j1end = (j1+1)*tile - 1;

                        for (it = i1begin; it <= i1end; it++)
                              for (int kt = k1begin; kt <= k1end; kt++)
                                    for (int jt = j1begin; jt <= j1end; jt++)
                                          c[it][jt] = c[it][jt] +
a[it][kt]*b[kt][jt];
                  }
            }
      }
}

void matr_multipl_JKIBL(double **a, double **b, double **c, int count, int tile)
{
      for (int j = 0; j < count; j++)
            for (int i = 0; i < count; i++)
                  c[i][j] = 0;

      int tilesize = count/tile;
      for (int j1 = 0; j1 < tilesize; j1++)
      {
            int j1begin = j1*tile;
            int j1end = (j1+1)*tile - 1;

            for (int k1 = 0; k1 < tilesize; k1++)
            {
                  int k1begin = k1*tile;
                  int k1end = (k1+1)*tile - 1;

                  for (int i1 = 0; i1 < tilesize; i1++)
                  {
                        int i1begin = i1*tile;
                        int i1end = (i1+1)*tile - 1;

                        for (int jt = j1begin; jt <= j1end; jt++)
                              for (int kt = k1begin; kt <= k1end; kt++)
                                    for (int it = i1begin; it <= i1end; it++)
                                          c[it][jt] = c[it][jt] +
a[it][kt]*b[kt][jt];
                  }
            }
      }
}
```

```
void matr_multipl_JIKBL(double **a, double **b, double **c, int count, int tile)
{
      int tilesize = count/tile;
      for (int j1 = 0; j1 < tilesize; j1++)
      {
            int j1begin = j1*tile;
            int j1end = (j1+1)*tile - 1;

            for (int i1 = 0; i1 < tilesize; i1++)
            {
                  int i1begin = i1*tile;
                  int i1end = (i1+1)*tile - 1;

                  for (int jt = j1begin; jt <= j1end; jt++)
                        for (int it = i1begin; it <= i1end; it++)
                              c[it][jt] = 0;

                  for (int k1 = 0; k1 < tilesize; k1++)
                  {
                        int k1begin = k1*tile;
                        int k1end = (k1+1)*tile - 1;

                        for (int jt = j1begin; jt <= j1end; jt++)
                              for (int it = i1begin; it <= i1end; it++)
                                    for (int kt = k1begin; kt <= k1end; kt++)
                                          c[it][jt] = c[it][jt] +
a[it][kt]*b[kt][jt];
                  }
            }
      }
}

void matr_multipl_KIJBL(double **a, double **b, double **c, int count, int tile)
{
      for (int i = 0; i < count; i++)
            for (int j = 0; j < count; j++)
                  c[i][j] = 0;

      int tilesize = count/tile;
      for (int k1 = 0; k1 < tilesize; k1++)
      {
            int k1begin = k1*tile;
            int k1end = (k1+1)*tile - 1;

            for (int i1 = 0; i1 < tilesize; i1++)
            {
                  int i1begin = i1*tile;
                  int i1end = (i1+1)*tile - 1;

                  for (int j1 = 0; j1 < tilesize; j1++)
                  {
                        int j1begin = j1*tile;
                        int j1end = (j1+1)*tile - 1;

                        for (int kt = k1begin; kt <= k1end; kt++)
                              for (int it = i1begin; it <= i1end; it++)
                                    for (int jt = j1begin; jt <= j1end; jt++)
                                          c[it][jt] = c[it][jt] +
a[it][kt]*b[kt][jt];
                  }
            }
      }
}
```

```
void matr_multipl_KJIBL(double **a, double **b, double **c, int count, int tile)
{
	for (int i = 0; i < count; i++)
		for (int j = 0; j < count; j++)
			c[i][j] = 0;

	int tilesize = count/tile;
	for (int k1 = 0; k1 < tilesize; k1++)
	{
		int k1begin = k1*tile;
		int k1end = (k1+1)*tile - 1;

		for (int j1 = 0; j1 < tilesize; j1++)
		{
			int j1begin = j1*tile;
			int j1end = (j1+1)*tile - 1;

			for (int i1 = 0; i1 < tilesize; i1++)
			{
				int i1begin = i1*tile;
				int i1end = (i1+1)*tile - 1;

				for (int kt = k1begin; kt <= k1end; kt++)
					for (int jt = j1begin; jt <= j1end; jt++)
						for (int it = i1begin; it <= i1end; it++)
							c[it][jt] = c[it][jt] +
a[it][kt]*b[kt][jt];

			}
		}
	}
}

int main ()
{
	char* file_name = "result.txt";
	ofstream out(file_name);
	DWORD start_time, end_time;

	cout << "Enter max count:  ";
	int count = 0;
	int maxcount;
	cin >> maxcount;

	double **a;
	double **b;
	double **c;
	int i, j;

	a = new double*[count];
	for (i = 0; i < count; i++)
		a[i] = new double[count];
	b = new double*[count];
	for (i = 0; i < count; i++)
		b[i] = new double[count];
	c = new double*[count];
	for (i = 0; i < count; i++)
		c[i] = new double[count];

	for (i = 0; i < count; i++)
	{
		for (j = 0; j < count; j++)
		{
```

```
            a[i][j] = 1;
            b[i][j] = 1;
            c[i][j] = 0;
      }
}

// rearrangement of cycles
//
int count = 1000;
for (count = 100; count <= maxcount; count += 100)
{
      a = new double*[count];
      for (i = 0; i < count; i++)
            a[i] = new double[count];
      b = new double*[count];
      for (i = 0; i < count; i++)
            b[i] = new double[count];
      c = new double*[count];
      for (i = 0; i < count; i++)
            c[i] = new double[count];

      for (i = 0; i < count; i++)
      {
            for (j = 0; j < count; j++)
            {
                  a[i][j] = 1;
                  b[i][j] = 1;
                  c[i][j] = 0;
            }
      }

out << "count = " << count << endl;
cout << "count = " << count << endl;
start_time = GetTickCount();
matr_multipl_IJK(a, b, c, count);
end_time = GetTickCount();
out << "Time ijk = " << end_time - start_time << endl;
cout << "Time ijk = " << end_time - start_time << endl;

start_time = GetTickCount();
matr_multipl_IKJ(a, b, c, count);
end_time = GetTickCount();
out << "Time ikj = " << end_time - start_time << endl;
cout << "Time ikj = " << end_time - start_time << endl;

start_time = GetTickCount();
matr_multipl_JIK(a, b, c, count);
end_time = GetTickCount();
out << "Time jik = " << end_time - start_time << endl;
cout << "Time jik = " << end_time - start_time << endl;

start_time = GetTickCount();
matr_multipl_JKI(a, b, c, count);
end_time = GetTickCount();
out << "Time jki = " << end_time - start_time << endl;
cout << "Time jki = " << end_time - start_time << endl;

start_time = GetTickCount();
matr_multipl_KIJ(a, b, c, count);
end_time = GetTickCount();
out << "Time kij = " << end_time - start_time << endl;
cout << "Time kij = " << end_time - start_time << endl;

start_time = GetTickCount();
```

```
  matr_multipl_KJI(a, b, c, count);
  end_time = GetTickCount();
  out << "Time kji = " << end_time - start_time << endl;
  cout << "Time kji = " << end_time - start_time << endl;
  out << endl;
  cout << endl;
  }

  // tiling
//
  out << "Tiling:" << endl;

  int* tiles = new int[14];
  tiles[0] = 2;
  tiles[1] = 4;
  tiles[2] = 5;
  tiles[3] = 8;
  tiles[4] = 10;
  tiles[5] = 20;
  tiles[6] = 25;
  tiles[7] = 40;
  tiles[8] = 100;
  tiles[9] = 125;
  tiles[10] = 200;
  tiles[11] = 250;
  tiles[12] = 500;

  for (i = 0; i < 13; i++)
  {
        int mincount = tiles[i];
        start_time = GetTickCount();
        matr_multipl_KIJBL(a, b, c, count, mincount);
        end_time = GetTickCount();
        out << "count = " << mincount << ".  ";
        out << "Time kij = " << end_time - start_time << endl;
        cout << "count = " << mincount << ".  ";
        cout << "Time kij = " << end_time - start_time << endl;
  }

  for (i = 0; i < 13; i++)
  {
        int mincount = tiles[i];
        start_time = GetTickCount();
        matr_multipl_JIKBL(a, b, c, count, mincount);
        end_time = GetTickCount();
        out << "count = " << mincount << ".  ";
        out << "Time jik = " << end_time - start_time << endl;
        cout << "count = " << mincount << ".  ";
        cout << "Time jik = " << end_time - start_time << endl;
  }

  for (i = 0; i < 13; i++)
  {
        int mincount = tiles[i];
        start_time = GetTickCount();
        matr_multipl_JKIBL(a, b, c, count, mincount);
        end_time = GetTickCount();
        out << "count = " << mincount << ".  ";
        out << "Time jki = " << end_time - start_time << endl;
        cout << "count = " << mincount << ".  ";
        cout << "Time jki = " << end_time - start_time << endl;
  }

  for (i = 0; i < 13; i++)
```

```
	{
		int mincount = tiles[i];
		start_time = GetTickCount();
		matr_multipl_IJKBL(a, b, c, count, mincount);
		end_time = GetTickCount();
		out << "count = " << mincount << ".  ";
		out << "Time ijk = " << end_time - start_time << endl;
		cout << "count = " << mincount << ".  ";
		cout << "Time ijk = " << end_time - start_time << endl;
	}

	for (i = 0; i < 13; i++)
	{
		int mincount = tiles[i];
		start_time = GetTickCount();
		matr_multipl_IKJBL(a, b, c, count, mincount);
		end_time = GetTickCount();
		out << "count = " << mincount << ".  ";
		out << "Time ikj = " << end_time - start_time << endl;
		cout << "count = " << mincount << ".  ";
		cout << "Time ikj = " << end_time - start_time << endl;
	}

	for (i = 0; i < 13; i++)
	{
		int mincount = tiles[i];
		start_time = GetTickCount();
		matr_multipl_KJIBL(a, b, c, count, mincount);
		end_time = GetTickCount();
		out << "count = " << mincount << ".  ";
		out << "Time kji = " << end_time - start_time << endl;
		cout << "count = " << mincount << ".  ";
		cout << "Time kji = " << end_time - start_time << endl;
	}

	for (i = 0; i < count; i++)
	{
		delete[] a[i];
		delete[] b[i];
		delete[] c[i];
	}
	delete[]a;
	delete[]b;
	delete[]c;

	out.close();

	return 0;
}
```