*Федеральное государственное бюджетное образовательное учреждение* *высшего профессионального образования*

|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | ***«Московский государственный технический университет  имени Н.Э. Баумана (национальный исследовательский институт)»***  ***(МГТУ им. Н.Э. Баумана)*** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА ИУ7

**Отчёт**

**по лабораторной работе №4**

**Дисциплина: Анализ алгоритмов**

**Тема лабораторной работы: Распараллеливание вычислений**

Студент гр. ИУ7-51Б **\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Громова В.П.** (Подпись, дата) (И.О. Фамилия)

Преподаватель  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Волкова Л.Л.**

(Подпись, дата) (И.О. Фамилия)

Москва, 2019г.

[**Введение**](#_k5dnresp2ym9) **2**

[**Аналитическая часть**](#_zic9qwef1coi) **3**

[Описание алгоритма](#_dhkj7ubicsvx) 3

[Потоки и параллельное программирование](#_wscfucrfmaaa) 4

[**Конструкторская часть**](#_d1jha4kqjiur) **6**

[Схема алгоритма](#_2vh4hivrbi71) 6

[Схема параллельного выполнения алгоритма](#_q6sx59bm0ts2) 8

[**Технологическая часть**](#_j4qi2v2ut5l) **9**

[Средства реализации](#_uzapz83ekot6) 9

[Реализация алгоритма](#_fny2mxnyyspo) 9

[**Экспериментальная часть**](#_u5gcws6x2ktn) **11**

[Примеры работы программы](#_vnbzaepg8dqy) 12

[Постановка эксперимента по замеру времени](#_mmksdbo96pd1) 13

[Сравнительный анализ на материале экспериментальных данных](#_pgwiqdnq7v1p) 14

[**Вывод**](#_uztxswi858ky) **17**

### Введение

Цель данной лабораторной работы: изучить способы распараллеливания вычислений.

Задачи лабораторной работы следующие.

1. Реализовать алгоритм Винограда умножения матриц.
2. Распараллелить вычисления, описать выбранный принцип и точки сбора результатов.
3. Сравнить время выполнения программы при распределении вычислительных задач по потокам (для разного количества рабочих потоков).
4. Описать проделанную работу и обосновать получившиеся результаты.

### Аналитическая часть

В данном разделе будут приведены описание алгоритма и некоторые принципы параллельной реализации.

#### Описание алгоритма

Алгоритм Винограда основан на сокращении количества операций умножения в алгоритме для повышения его эффективности. Результат умножения двух матриц представляет собой скалярное произведение соответствующего строки и столбца. Такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.  
Рассмотрим два вектора V = (v1, v2, v3, v4) и W = (w1, w2, w3, w4). Их скалярное произведение равно: V • W = v1w1 + v2w2 + v3w3 + v4w4.

Это равенство можно переписать в виде: V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) — v1v2 — v3v4 — w1w2 — w3w4.

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем первое: вместо четырех умножений - шесть, а вместо трех сложений - десять, выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй, что позволяет выполнять для каждого элемента лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения. В случае нечетного количества столбцов первой матрицы, необходимо будет провести дополнительные вычисления: к каждой ячейки результирующей матрицы Cij прибавить произведение Ai(n-1) \* B(n-1)j, где n - количество столбцов в первой матрице ( и строк -- во второй).

#### Потоки и параллельное программирование

Процесс — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память). Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого. Поток использует то же самое пространства стека, что и процесс. Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса. У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать. Поток — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса. При этом потоки разных процессов защищены друг от друга. Потоки возникли в операционных системах как средство распараллеливания вычислений. Многопоточность позволяет эффективно использовать все вычислительные ресурсы системы, при этом задачи, реализованные параллельно, выполняются быстрее.

На одном процессоре многопоточность обычно происходит путем временного мультиплексирования: процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

При распараллеливании вычислений существует главный поток, который создает несколько рабочих потоков, передает им аргументы и запускает созданные потоки. Далее главный поток ждет завершения выполнения рабочих потоков. Рабочие потоки могут одновременно читать данные из одной и той же области данных, однако одновременная запись из двух и более параллельно выполняющихся потоков в одну ячейку памяти запрещена. Для обеспечения монопольного доступа на запись к ресурсам памяти используются мьютексы. Задача мьютекса — защита объекта от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищенным мьютексом. Если другому потоку будет нужен доступ к переменной, защищенной мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобожден.

### Конструкторская часть

В данном разделе будут представлены схемы алгоритма и схема распараллеливания.

#### Схема алгоритма

На рисунках 2.1.1 - 2.1.2 представлена схема оптимизированного алгоритма Винограда умножения входной матрицы mtr1 размером q на m на входную матрицу mtr2 размером m на n.

|  |  |
| --- | --- |
|  |  |

*Рис. 2.1.1 - оптимизированный алгоритм Винограда*

|  |
| --- |
|  |
| *Рис. 2.1.2 - оптимизированный алгоритм Винограда (продолжение)* |

#### Схема параллельного выполнения алгоритма

Для реализации многопоточного выполнения, вычисления были разделены на два функциональных блока:

1. создание вспомогательных строки и столбца;
2. вычисление результирующей матрицы.

Каждому потоку были переданы границы ответственности в виде индексов в циклах.

На рисунке 2.2.1 представлена схема распараллеливания для двух потоков. Точки сбора соответствуют тем моментам выполнения программы, в которые оба потока заканчивают свою работу.

|  |
| --- |
|  |
| *Рис. 2.2.1 - схема распараллеливания вычислений* |

### Технологическая часть

В этом разделе будут описаны средства реализации и представлены листинги кода.

#### Средства реализации

Для параллельной реализации алгоритма был выбран язык программирования C++, так как уже были практические навыки работы с ним. Также причиной выбора этого языка программирования стал тот факт, что в стандартной библиотеке реализован класс thread, который представляет нативный поток. Поскольку потоки не использовали разделяемую память, класс mutex не использовался. Для замера времени была использована функция стандартной библиотеки std::chrono::steady\_clock.

#### Реализация алгоритма

При реализации многопоточности было создано три отдельные функции. В листинге 1 представлена функция, отвечающая за создание вспомогательной строки. В листинге 2 -- за создание вспомогательного столбца. В листинге 3 представлена функция, в которой реализовано непосредственно умножение двух матриц.

Листинг 1:

1. void create\_row(Matrix& mtr, vector<int>& row, const int& n\_start, const int& n\_end, const int& m)
2. {
3. for (auto i = n\_start; i < n\_end; i++)
4. for (auto j = 0; j < m; j += 2)
5. row[i] -= (mtr.mtr)[i][j] \* (mtr.mtr)[i][j + 1];
6. }

Листинг 2:

1. void create\_column(Matrix& mtr, vector<int>& column, const int& n\_start, const int& n\_end, const int& m)
2. {
3. for (auto i = n\_start; i < n\_end; i++)
4. for (auto j = 0; j < m; j += 2)
5. column[i] -= (mtr.mtr)[j][i] \* (mtr.mtr)[j + 1][i];
6. }

Листинг 3:

1. void calculate(Matrix& mtr1, Matrix& mtr2, Matrix& res, vector<int>& row, vector<int>& column, const int& n\_start, const int& n\_end)
2. {
3. int d = mtr1.m - 1;
4. int eq = mtr1.m % 2;
5. int buf = 0;
6. for (auto i = n\_start; i < n\_end; i++)
7. {
8. for (auto j = 0; j < mtr2.m; j++)
9. {
10. buf = row[i] + column[j];
11. for (auto k = 0; k < d; k += 2)
12. {
13. buf += ((mtr1.mtr)[i][k] + (mtr2.mtr)[k + 1][j]) \* ((mtr1.mtr)[i][k + 1] + (mtr2.mtr)[k][j]);
14. }
15. if (eq)
16. buf += (mtr1.mtr)[i][d] \* (mtr2.mtr)[d][j];
17. (res.mtr)[i][j] = buf;
18. }
19. }
20. }

В листинге 4 представлена реализация алгоритма для двух потоков выполнения. 4-х и 8-ми поточные реализации выполнены аналогичным образом.

Листинг 4:

1. double multiply\_two(Matrix& mtr1, Matrix& mtr2, Matrix& res)
2. {
3. vector<int> row(mtr1.n);
4. vector<int> column(mtr2.m);
5. int d = mtr1.m - 1;
6. res.zero();
7. int n\_end = mtr1.n;
8. int m\_end = mtr2.m;
9. int n\_end2 = n\_end / 2;
10. auto start = chrono::steady\_clock::now();
11. thread thr\_row(create\_row, ref(mtr1), ref(row), 0, ref(n\_end), ref(d));
12. thread thr\_column(create\_column, ref(mtr2), ref(column), 0, ref(m\_end), ref(d));
13. // первая точка сбора
14. thr\_row.join();
15. thr\_column.join();
16. thread thr\_calculate1(calculate, ref(mtr1), ref(mtr2), ref(res), ref(row), ref(column), 0, ref(n\_end2));
17. thread thr\_calculate2(calculate, ref(mtr1), ref(mtr2), ref(res), ref(row), ref(column), ref(n\_end2), ref(n\_end));
18. // вторая точка сбора
19. thr\_calculate1.join();
20. thr\_calculate2.join();
21. auto end = chrono::steady\_clock::now();
22. double elapsed\_seconds = chrono::duration\_cast<chrono::duration<double>>(end - start).count();
23. return elapsed\_seconds;
24. }

### Экспериментальная часть

В данном разделе будут приведены примеры работы программы, результаты функционального тестирования, а также будут представлены результаты и анализ проведенных экспериментов.

#### Примеры работы программы

На рисунках 4.1.1 - 4.1.3 будут приведены примеры работы программы.

|  |
| --- |
|  |
| *Рис. 4.1.1 - пример работы программы для матриц 1x1 и 1x1* |

|  |
| --- |
|  |
| *Рис. 4.1.2 - пример работы программы для матриц 2x2 и 2x2* |

|  |
| --- |
|  |
| *Рис. 4.1.3 - пример работы программы для матриц 3x3 и 3x3* |

При функциональном тестировании все тесты были пройдены и результаты совпали с ожидаемыми.

#### Постановка эксперимента по замеру времени

При сравнении быстродействия алгоритмов были использованы матрицы размерами в диапазоне от 100x100 до 1000x1000 с шагом 100 (и по строкам и по столбцам) и в диапазоне от 101x101 до 1001x1001 также с шагом 100. Результат одного эксперимента рассчитывался как средний из результатов проведенных испытаний с одинаковыми входными данными. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными. Для проведения эксперимента был использован компьютер с 4 физическими ядрами и 4 логическими процессорами.

#### Сравнительный анализ на материале экспериментальных данных

На рисунках 4.3.1 - 4.3.2 приведены графики зависимости временных затрат работы алгоритмов (в секундах) от размеров матриц.

|  |
| --- |
|  |
| *Рис. 4.3.1 - график зависимости времени работы программы от четных входных размеров матриц для различного количества потоков* |

|  |
| --- |
|  |
| *Рис. 4.3.2 - график зависимости времени работы программы от нечетных входных размеров матриц для различного количества потоков* |

В результате проведенных экспериментов можно сделать следующий вывод: как и предполагалось, многопоточная реализация выполняет вычисления быстрее, чем однопоточная. При разделении вычислений на 2 потока, время выполнения программы сокращается вдвое по сравнению с однопоточной реализацией, а для 4 потоков -- почти вчетверо. Однако вариант реализации с 8 потоками выполняется медленнее, чем аналогичный вариант для 4 потоков. Это связано с тем, что время на маршрутизацию потоков (создание дополнительных потоков, простой потоков в очереди, переключение контекста) превышает время выполнения основных вычислений. Таким образом, самое оптимальное количество потоков - четыре, что соответствует количеству логических процессоров компьютера.

### Вывод

Цель данной лабораторной работы была достигнута. В ходе работы были изучены основные принципы параллельного программирования и изучены инструменты работы с нативными потоками в языке программирования C++, а также был параллельно реализован алгоритм Винограда умножения матриц. Был проведен сравнительный анализ реализаций приведенного алгорима для различного количества потоков выполнения, а также были экспериментально получены зависимости времени выполнения реализаций от размеров входных матриц. В результате: чем больше количество потоков, тем быстрее выполняется программа, однако существует ограничение по максимальному количеству потоков, выражающееся в количестве логических ядер ЭВМ, на которой выполняются вычисления.