МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ   
РОССИЙСКОЙ ФЕДЕРАЦИИ  
  
Федеральное государственное автономное образовательное учреждение   
высшего образования   
«Самарский национальный исследовательский университет   
имени академика С.П. Королёва» (Самарский университет)  
  
Факультет информатики  
Кафедра программных систем  
  
Дисциплина  
**Параллельное программирование  
  
  
  
ОТЧЕТ**по лабораторной работе №1  
 **Параллельный алгоритм матричного умножения**

**– реализация на OpenMP и MPI**

Студент: Гижевская В.Д.

Группа: 6413-020302D  
  
Преподаватель: Оплачко Д.С.  
Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Самара 2021

**Содержание**

[Задание 3](#_Toc87528495)

[Постановка задачи 4](#_Toc87528496)

[Описание работы алгоритмов 5](#_Toc87528497)

[Результаты вычислительных экспериментов 8](#_Toc87528498)

[Выводы 14](#_Toc87528499)

[Список использованных источников 15](#_Toc87528500)

[ПРИЛОЖЕНИЕ А   
Листинг последовательной программы 16](#_Toc87528501)

[ПРИЛОЖЕНИЕ B   
Листинг параллельной программы с помощью OpenMP 17](#_Toc87528502)

[ПРИЛОЖЕНИЕ C   
Листинг параллельной программы с использованием MPI 18](#_Toc87528503)

# **Задание**

1. Необходимо разработать последовательный алгоритм умножения матриц размерностью nxn на языке С.
2. На основе последовательного алгоритма разработать параллельный алгоритм умножения матриц с использованием технологии OpenMP на языке С. Измерить время работы для данных размеров матриц: 10, 50, 100, 200, 400, 800, 1000, 1200, 1500, 2000 с различным количеством потоков p∈{2,4,8}.
3. На основе последовательного алгоритма разработать параллельный алгоритм умножения матриц с использованием библиотеки MPI на языке С. Измерить время работы для данных размеров матриц: 10, 50, 100, 200, 400, 800, 1000, 1200, 1500, 2000 с различным количеством потоков p∈{2,4,8}.

# **Постановка задачи**

Во время выполнения лабораторной работы № 1 необходимо разработать последовательный алгоритм умножения квадратных матриц размерностью nxn на языке С, а также изучить технологию OpenMP и библиотеку MPI и разработать параллельный алгоритм перемножения матриц. Также необходимо измерить время выполнения программ для матриц размерностями n= 10, 50, 100, 200, 400, 800, 1000, 1200, 1500, 2000. Время для работы параллельных алгоритмов измерить при различном количестве процессов p∈{2,4,8}. Необходимо построить графики ускорения по данным из таблиц.

Вычисления будут производиться с использованием процессора Intel Core i5.

# **Описание работы алгоритмов**

1. Последовательный алгоритм

c_{ij}=\sum_{k=0}^{n-1} a_{ik}\cdot b_{kj},\;
0 \le i < m, \; 0 \le j < lУмножение матрицы A размера mxn и матрицы B размера nxl приводит к получению матрицы С размера mxl, каждый элемент которой определяется в соответствии с выражением:

(1)

Как следует из (1), каждый элемент результирующей матрицы С есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

c_{ij}=(a_i,b_j^T), \; a_i = (a_{i0}, a_{i1} \ldots, a_{in-1}), 
\; b_j^T = (b_{0j},b_{1j},\ldots,b_{n-1j})^T . (2)

Этот алгоритм предполагает выполнение mxnxl операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера nxn количество выполненных операций имеет порядок O(n3). Далее предполагается, что все матрицы являются квадратными и имеют размер nxn.

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами. Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы С. При выполнении одной итерации внешнего цикла вычисляется одна строка результирующей матрицы.

Реализация алгоритма представлена в приложении А.

1. Параллельные алгоритм
   1. С использованием OpenMP

Технология OpenMP является одним из наиболее популярных средств программирования для компьютерных систем с общей памятью, базирующихся на традиционных языках программирования. OpenMP состоит из набора директив для компиляторов и библиотек специальных функций.

Одним из важных достоинств технологии OpenMP является реализация идеи «инкрементального программирования», когда разработчик постепенно находит участки в программе, содержащие ресурс параллелизма, с помощью предоставляемых механизмов делает их параллельными и затем переходит к анализу следующих участков.

Таким образом данный подход упрощает процесс адаптации последовательных программ к параллельным ЭВМ. OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле.

Если в параллельной области встретился оператор цикла без дополнительных указаний, то он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву for. Эта директива относится к следующему за ней блоку, содержащему оператор for.

Опции данной директивы:

private(список) – задает список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено.

schedule(type[, chunk]) – опция задаёт, каким образом итерации цикла распределяются между нитями.

Параметрами опции schedule являются следующие:

static – распределение итераций цикла; размер блока – chunk. Первый блок из chunk итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение chunk не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.

dynamic – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает chunk итераций (по умолчанию chunk=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из chunk итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.

guided – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины chunk (по умолчанию chunk=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения chunk.

Реализация алгоритма представлена в приложении Б.

* 1. С использованием MPI

Наиболее распространенной технологией программирования параллельных компьютеров с распределенной памятью является технология MPI. Основным способом взаимодействия параллельных процессов в таких системах является передача сообщений друг другу. Это и отражено в названии технологии – Message Passing Interface.

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Процессы могут выполняться на разных процессорах, но на одном процессоре могут располагаться и несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени). В предельном случае для выполнения параллельной программы может использоваться один процессор – как правило, такой способ применяется для начальной проверки правильности параллельной программы.

Основу MPI составляют операции передачи сообщений.

Реализация алгоритма представлена в приложении B.

# **Результаты вычислительных экспериментов**

В таблице 1 приведены результаты измерений работы алгоритма OpenMP.

График ускорения S параллельных алгоритмов с помощью технологии OpenMP и MPI представлен на рисунке 1.

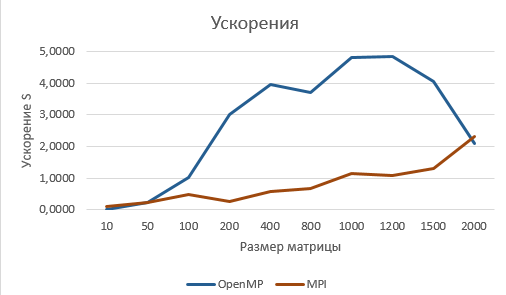


Рисунок 1 – График ускорения работы программы OMP и MPI

Результаты измерений при различных значения параметра schedule() представлены в таблице 2.

Результаты измерений работы алгоритма MPI представлены в таблице 3.

По результатам вычислений видно, что на малых размерах матриц последовательный алгоритм работает быстрее, чем параллельные, так как данные полностью умещаются в кэш-память и доступ к ним осуществляется быстро, поэтому на организацию параллельных вычислений затрачивается больше времени. Но начиная с некоторых размеров матриц, параллельные алгоритмы начинают давать явное преимущество по времени.

Таблица 1 – Сравнение результатов матричного умножения последовательного алгоритма и параллельного с технологией OpenMP

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размерность матриц, n | 10 | 50 | 100 | 200 | 400 | 800 | 1000 | 1200 | 1500 | 2000 |
| Время работы последовательного алгоритма t,c | 0.000015 | 0,000662 | 0,003973 | 0,038088 | 0,435631 | 4,385714 | 11,951927 | 18,948705 | 43,340210 | 131,24581 |
| Время работы алгоритма с помощью OpenMP p=2 t,c | 0,000700 | 0,001185 | 0,003518 | 0,020165 | 0,242062 | 2,438606 | 5,854111 | 9,731681 | 21,652971 | 71,472369 |
| Ускорение S | 0,0214 | 0,5586 | 1,129335 | 1,8888 | 1,7997 | 1,7985 | 2,0416 | 1,9471 | 2,0016 | 1,8363 |
| Время работы алгоритма с помощью OpenMP p=4 t,c | 0,001910 | 0,002134 | 0,003257 | 0,014962 | 0,120617 | 1,440096 | 3,274945 | 5,438775 | 13,030302 | 66,391517 |
| Ускорение S | 0,0079 | 0,3102 | 1,2198 | 2,5456 | 3,6117 | 3,0454 | 3,6495 | 3,4840 | 3,3261 | 1,9768 |

Продолжение таблицы 1

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Время работы алгоритма с помощью OpenMP p=8 t,c | 0,003077 | 0,002785 | 0,003887 | 0,012622 | 0,110508 | 1,188705 | 2,482164 | 3,908729 | 10,667431 | 62,503281 |
| Ускорение S | 0,0049 | 0,2377 | 1,0221 | 3,0176 | 3,9421 | 3,6895 | 4,8151 | 4,8478 | 4,0629 | 2,0998 |

Таблица 2 – Длительность работы параллельных программ с различными значениями параметра schedule() p=8

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| schedule | Размерность матриц, n | 10 | 50 | 100 | 200 | 400 | 800 | 1000 | 1200 | 1500 | 2000 |
| static | Время работы алгоритма t,c | 0,003145 | 0,003673 | 0,00456 | 0,014177 | 0,111539 | 1,021646 | 2,734041 | 4,00074 | 9,7079 | 37,121004 |
| Ускорение S | 0,0048 | 0,1802 | 0,8713 | 2,6866 | 3,9056 | 4,2928 | 4,3715 | 4,7363 | 4,4644 | 3,5356 |
| dynamic | Время работы алгоритма t,c | 0,004119 | 0,01931 | 0,003537 | 0,012348 | 0,083809 | 0,960244 | 2,452417 | 4,293547 | 9,865017 | 36,588624 |
| Ускорение S | 0,0036 | 0,0343 | 1,1233 | 3,0845 | 5,1979 | 4,5673 | 4,8735 | 4,4133 | 4,3933 | 3,5871 |
| guided | Время работы алгоритма t,c | 0,003145 | 0,003673 | 0,00456 | 0,014177 | 0,111539 | 1,021646 | 2,734041 | 4,00074 | 9,7079 | 37,121004 |
| Ускорение S | 0,0048 | 0,1802 | 0,8713 | 2,6866 | 3,9056 | 4,2928 | 4,3715 | 4,7363 | 4,4644 | 3,5356 |

Таблица 3 – Сравнение результатов матричного умножения последовательного алгоритма и параллельного с технологией MPI

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размерность матриц, n | 10 | 50 | 100 | 200 | 400 | 800 | 1000 | 1200 | 1500 | 2000 |
| Время работы последовательного алгоритма t,c | 0.000015 | 0,000662 | 0,003973 | 0,038088 | 0,435631 | 4,385714 | 11,951927 | 18,948705 | 43,340210 | 131,24581 |
| Время работы алгоритма с помощью MPI p=2 t,c | 0,0000152 | 0,000345 | 0,007433 | 0,0160435 | 1,023175 | 5,74573 | 7,74921 | 15,89357 | 29,83429 | 57,74803 |
| Ускорение S | 0,5921 | 8,0145 | 12,1873 | 5,6464 | 0,9908 | 1,0948 | 2,3498 | 1,5899 | 1,8502 | 4,1088 |
| Время работы алгоритма с помощью MPI p=4 t,c | 0,0000135 | 0,0010449 | 0,007559 | 0,077499 | 0,69653 | 6,4739 | 12,43495 | 18,56359 | 35,046439 | 59,46493 |
| Ускорение S | 0,6667 | 2,6462 | 11,9841 | 1,1689 | 1,4555 | 0,9716 | 1,4643 | 1,3612 | 1,5750 | 3,9902 |

Продолжение таблицы 3

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Время работы алгоритма с помощью MPI p=8 t,c | 0,000154 | 0,00285 | 0,008493 | 0,1464 | 0,747489 | 6,63959 | 10,46403 | 17,46438 | 33,63845 | 56,47304 |
| Ускорение S | 0,0584 | 0,9702 | 10,6662 | 0,6188 | 1,3563 | 0,9474 | 1,7402 | 1,4469 | 1,6409 | 4,2016 |

# **Выводы**

Из приведенных выше результатов видно, что алгоритмы на малых размерах матрицы (10-50) параллельные алгоритмы с OpenMP не дают выигрыша по времени исполнения, однако на больших размерах матрицы параллельные версии программ дают значительное ускорение – примерно 2,5 раза. Стоит отметить, что на размерах матрицы от 100 до 400 включительно большее ускорение давали параллельные алгоритмы со значением dynamic параметра schedule, однако, начиная с n = 800, лучшие результаты показывало значение static.

Алгоритмы с MPI начинают давать ускорение только при значениях матрицы более 100 и даже на больших размерах матрицы дают меньшее ускорение, чем алгоритмы с использованием OpenMP. Это связано с тем, что MPI тратит большое количество времени на передачу данных между различными процессами и на подготовку передаваемых сообщений.

# **Список использованных источников**

1. СТО 02068410-004-2018. Общие требования к учебным текстовым документам: методические указания [Электронный ресурс]. URL: https://ssau.ru/docs/sveden/localdocs/STO\_SGAU\_020684.. (дата обращения: 25.09.2021).
2. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления - СПб.: БХВ-Петербург, 2002. - 608 с. (дата обращения: 26.09.2021).

# **ПРИЛОЖЕНИЕ А Листинг последовательной программы**

#include <iostream>

#include <locale.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define N 1500

int main()

{

double\*\* A = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* B = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* C = (double\*\*)malloc(N \* sizeof(double\*));

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

{

A[i] = (double\*)malloc(N \* sizeof(double));

B[i] = (double\*)malloc(N \* sizeof(double));

C[i] = (double\*)malloc(N \* sizeof(double));

}

double t1, t2;

t1 = omp\_get\_wtime();

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

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

{

A[i][j] = 3;

B[i][j] = 1;

}

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

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

{

C[i][j] = 0;

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

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

}

t2 = omp\_get\_wtime();

printf("t = %f\n", t2 - t1);

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

{

free(A[i]);

free(B[i]);

free(C[i]);

}

free(A);

free(B);

free(C);

}}

# **ПРИЛОЖЕНИЕ B Листинг параллельной программы с помощью OpenMP**

#include <iostream>

#include <omp.h>

#include <locale.h>

#include <stdio.h>

#include <stdlib.h>

#define N 50

double\*\* A = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* B = (double\*\*)malloc(N \* sizeof(double\*));

double\*\* C = (double\*\*)malloc(N \* sizeof(double\*));

int i, j, k;

int main()

{

setlocale(LC\_ALL, "Russian");

omp\_set\_num\_threads(8);

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

{

A[i] = (double\*)malloc(N \* sizeof(double));

B[i] = (double\*)malloc(N \* sizeof(double));

C[i] = (double\*)malloc(N \* sizeof(double));

}

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

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

{

A[i][j] = 1;

B[i][j] = 3;

}

double t1, t2;

t1 = omp\_get\_wtime();

#pragma omp parallel for private(i,j,k) shared(A, B, C) schedule(dynamic)

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

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

{

C[i][j] = 0;

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

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

}

t2 = omp\_get\_wtime();

printf("t = %f\n", t2 - t1);

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

{

free(A[i]);

free(B[i]);

free(C[i]);

}

free(A);

free(B);

free(C);

system("pause");

}

# **ПРИЛОЖЕНИЕ C Листинг параллельной программы с использованием MPI**

#include <stdio.h>

#include <cstdlib>

#include "mpi.h"

int main(int\* argc, char\*\* argv)

{

const int size = 2000;

double t1;

double t2;

MPI\_Init(argc, &argv);

int n, r;

MPI\_Status status;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &n);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &r);

int chunk = size / n;

double\* MatrixA = (double\*)malloc(sizeof(double) \* size \* size);

if (r == 0) {

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

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

MatrixA[j + i \* size] = 1.0;

}

double\* MatrixB = (double\*)malloc(sizeof(double) \* size \* size);

if (r == 0) {

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

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

MatrixB[j + i \* size] = 1.0;

}

double\* MatrixPartRes = (double\*)malloc(sizeof(double) \* chunk \* size);

double\* MatrixC = (double\*)malloc(sizeof(double) \* size \* size);

t1 = MPI\_Wtime();

if (r == 0) {

for (int i = 1; i < n; i++) {

MPI\_Send(MatrixA + i \* size \* chunk, size \* chunk, MPI\_DOUBLE, i, 0, MPI\_COMM\_WORLD);

}

}

else

{

MPI\_Recv(MatrixA, size \* chunk, MPI\_DOUBLE, 0, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

}

MPI\_Bcast(MatrixB, size \* size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

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

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

MatrixPartRes[i \* size + j] = 0;

for (int k = 0; k < size; k++)

MatrixPartRes[i \* size + j] += MatrixA[i \* size + k] \* MatrixB[k \* size + j];

}

}

if (r != 0) {

MPI\_Send(MatrixPartRes, size \* chunk, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

if (r == 0) {

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

MatrixC[i] = MatrixPartRes[i];

}

for (int i = 1; i < n; i++) {

MPI\_Recv(MatrixC + i \* size \* chunk, size \* chunk, MPI\_DOUBLE, i,

MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

}

}

t2 = MPI\_Wtime();

if (r == 0) {

printf(" t= %.15f \n", t2 - t1);

}

MPI\_Finalize();

system("pause");

}