ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ВС

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5
«Многопоточное программирование»
по дисциплине «Архитектура вычислительных систем»

Выполнил: студент гр. ИП-811
Адов Артем Сергеевич
Проверил: ст. преп. Кафедры ВС
Ткачёва Татьяна Алексеевна

Новосибирск 2020

Постановка задачи

1. Для программы умножения двух квадратных матриц DGEMM BLAS разработанной в задании 4 на языке C/C++ реализовать многопоточные вычисления. В потоках необходимо реализовать инициализацию массивов случайными числами типа double и равномерно распределить вычислительную нагрузку. Обеспечить возможность задавать размерность матриц и количество потоков при запуске программы. Многопоточность реализовать несколькими способами.

1)С использованием библиотеки стандарта POSIXThreads.

2)С использованием библиотеки стандарта OpenMP.

3)*Сиспользованием библиотеки IntelTBB.

4)**Сиспользованием библиотеки стандарта MPI.Все матрицы помещаются в общей памятиодного вычислителя.

5)***Сиспользованием технологий многопоточностидля графических сопроцессоров(GPU) -CUDA/OpenCL/OpenGL/OpenACC.

2.Для всех способов организации многопоточности построить график зависимостикоэффициента ускорения многопоточной программы от числа потоков длязаданной размерностиматрицы, например, 5000, 10000 и20000элементов.

3.Определить оптимальное число потоков для вашего оборудования.

4.Подготовить отчет отражающий суть, этапы и результаты проделанной работы

Результат работы

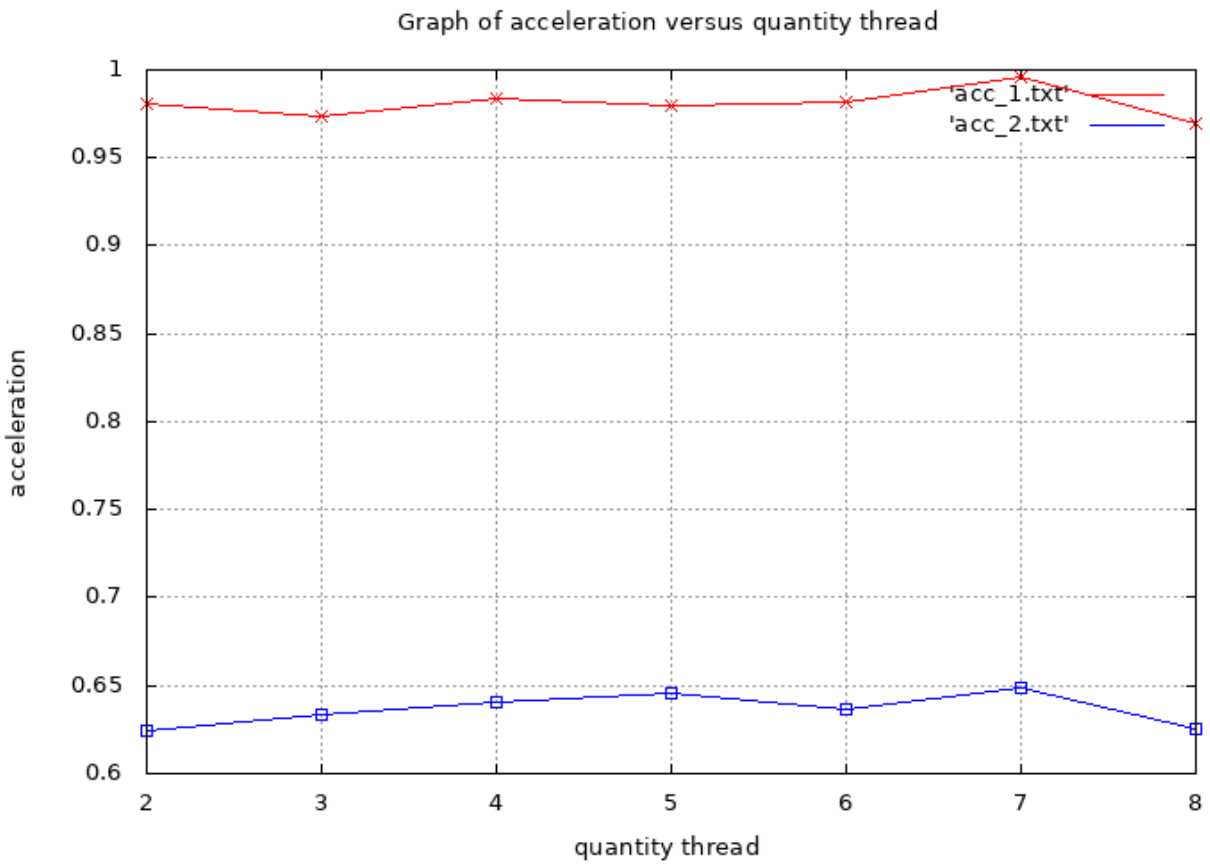

Рис. 1 «График зависимости ускорения от количества потоков для программ с применением ***pthreads(acc_1)*** и ***openmp (acc_2)*** »

Приложение

1.*Последовательная программа*

main.c

```
#include "death_error.h"
#include "death_arg.h"
#include "death_char.h"
#include "death_file.h"
#include "death_matrix.h"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// #define PRINT_DEBUG

int Multiple_Square_Matrix(double* result[], double first[], double second[], int length)
{
	double sum = 0.0;
	*result = (double*)calloc((length*length),sizeof(double));
	if(*result == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	for(short int i = 0; i < length; i++)
	{
```

```
            for(short int j = 0; j < length; j++)
            {
                sum = 0.0;
                for(short int k = 0; k < length; k++)
                {
                    sum = (sum + first[i*length+k] * second[k*length + j]);
                }
                (*result)[i*length + j] = sum;
            }
        }
        return 0;
}

int main(int argc, char* argv[])
{
        struct timespec time_before, time_after;
        double *matrix_first, *matrix_second, *matrix_result;
        double value_time = 0.0;
        int error_flag = 0b0;
        int length_square_matrix = 0;
        srand(time(NULL));
        argc = argc + 1;
        error_flag = (error_flag | Check_Argc(argc));
        error_flag = (error_flag | Convert_Char_To_Int(&length_square_matrix,
argv[INDEX_LENGTH_SQUARE_MATRIX_FROM_ARGV]));
        if(length_square_matrix > MAX_LENGTH_SQUARE_MATRIX)
        {
            return (error_flag | ERROR_SIZE_MATRIX_TOO_LARGE);
        }
```

```
	error_flag = Fill_Rand_Square_Matrix(&matrix_first,
length_square_matrix);
	error_flag = Fill_Rand_Square_Matrix(&matrix_second,
length_square_matrix);
	#ifdef PRINT_DEBUG
	printf("\nmatrix_first:\n");
	Print_Square_Matrix(matrix_first, length_square_matrix);
	printf("\nmatrix_second:\n");
	Print_Square_Matrix(matrix_second, length_square_matrix);
	#endif
	clock_gettime(CLOCK_REALTIME, &time_before);
	error_flag = (error_flag | Multiple_Square_Matrix(&matrix_result,
matrix_first, matrix_second, length_square_matrix));
	clock_gettime(CLOCK_REALTIME, &time_after);
	value_time = ((double)(time_after.tv_sec - time_before.tv_sec) +
(double)(time_after.tv_nsec - time_before.tv_nsec) / 1e9);
	#ifdef PRINT_DEBUG
	printf("\nmatrix_result:\n");
	Print_Square_Matrix(matrix_result, length_square_matrix);
	#endif
	free(matrix_first);
	free(matrix_result);
	free(matrix_second);
	matrix_first = NULL;
	matrix_second = NULL;
	matrix_result = NULL;
	error_flag = (error_flag | Write_In_Table_Time(0, value_time));
	return error_flag;
}
```

2. *Распараллелена при помощи openmp*

main.c

```
#include "death_error.h"
#include "death_arg.h"
#include "death_char.h"
#include "death_file.h"
#include "death_matrix.h"

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <sys/time.h>

// #define PRINT_DEBUG

int Multiple_Square_Matrix_With_Open_MP(double* result[], double first[], double second[], int length, int quantity_thread)
{
	double sum = 0.0;
	int i = 0, j = 0, k = 0;
	*result = (double*)calloc((length*length),sizeof(double));
	#pragma omp parallel for shared(first, second, result) private(j, k) num_threads(quantity_thread) schedule(static)
	for(i = 0; i < length; i++)
	{
		for(j = 0; j < length; j++)
		{
```

```
                sum = 0.0;
                for(k = 0; k < length; k++)
                {
                        sum = (sum + first[i*length+k] * second[k*length+j]);
                }
                (*result)[i*length+j] = sum;
            }
        }
        return 0;
}

int main(int argc, char* argv[])
{
        struct timespec time_before, time_after;
        double *matrix_first, *matrix_second, *matrix_result;
        double value_time = 0.0;
        int error_flag = 0b0;
        int quantity_thread = 0;
        int length_square_matrix = 0;
        srand(time(NULL));
        error_flag = (error_flag | Check_Argc(argc));
        error_flag = (error_flag | Convert_Char_To_Int(&quantity_thread,
argv[INDEX_QUANTITY_THREAD_FROM_ARGV]));
        error_flag = (error_flag | Convert_Char_To_Int(&length_square_matrix,
argv[INDEX_LENGTH_SQUARE_MATRIX_FROM_ARGV]));
        if(length_square_matrix > MAX_LENGTH_SQUARE_MATRIX)
        {
                return (error_flag | ERROR_SIZE_MATRIX_TOO_LARGE);
        }
```

```
        error_flag = Fill_Rand_Square_Matrix(&matrix_first,
length_square_matrix);

        error_flag = Fill_Rand_Square_Matrix(&matrix_second,
length_square_matrix);

        #ifdef PRINT_DEBUG

        printf("\nmatrix_first:\n");

        Print_Square_Matrix(matrix_first, length_square_matrix);

        printf("\nmatrix_second:\n");

        Print_Square_Matrix(matrix_second, length_square_matrix);

        #endif

        clock_gettime(CLOCK_REALTIME, &time_before);

        error_flag = (error_flag |
Multiple_Square_Matrix_With_Open_MP(&matrix_result, matrix_first,
matrix_second, length_square_matrix, quantity_thread));

        clock_gettime(CLOCK_REALTIME, &time_after);

        value_time = ((double)(time_after.tv_sec - time_before.tv_sec) +
(double)(time_after.tv_nsec - time_before.tv_nsec) / 1e9);

        #ifdef PRINT_DEBUG

        printf("\nmatrix_result:\n");

        Print_Square_Matrix(matrix_result, length_square_matrix);

        #endif

        free(matrix_first);

        free(matrix_result);

        free(matrix_second);

        matrix_first = NULL;

        matrix_second = NULL;

        matrix_result = NULL;

        error_flag = (error_flag | Write_In_Table_Time(quantity_thread,
value_time));

        return error_flag;
```

```
}
```

3. *Распараллелена при помощи pthread*

main.c

```
#include "death_error.h"
#include "death_arg.h"
#include "death_char.h"
#include "death_file.h"
#include "death_matrix.h"

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/time.h>

// #define PRINT_DEBUG

typedef struct Data_For_Thread
{
	int start;
	int final;
	int length;
	double* first;
	double* second;
	double* result;
} data_thread;
```

```
void* Function_By_Thread(void* original)
{
	data_thread* data = (data_thread*)original;
	double* a = data->first;
	double* b = data->second;
	double* c = data->result;
	double sum = 0.0;
	int begin = data->start;
	int end = data->final;
	int size = data->length;
	for(short int i = begin; i < end; i++)
	{
		for(short int j = 0; j < size; j++)
		{
			sum = 0.0;
			for(short int k = 0; k < size; k++)
			{
				sum = (sum + a[i*size+k] * b[k*size+j]);
			}
			c[i*size+j] = sum;
		}
	}
	pthread_exit(0);
}

int Multiple_Square_Matrix_With_Thread(double* result[], double first[], double second[], int length, int quantity_thread)
{
```

```
	pthread_t* array_pointer_thread = NULL;
	data_thread* data = NULL;
	int optimal = 0;
	array_pointer_thread =
(pthread_t*)calloc(quantity_thread,sizeof(pthread_t));
	if(array_pointer_thread == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	data = (data_thread*)calloc((quantity_thread+1),sizeof(data_thread));
	if(data == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	*result = (double*)calloc((length*length),sizeof(double));
	if(*result == NULL)
	{
		return ERROR_ALLOCATION_MEMORY;
	}
	optimal = (length / quantity_thread);
	if((length % quantity_thread) != 0)
	{
		for(short int i = 0, temp = 0; i < quantity_thread; i++, temp = (temp +
optimal))
		{
			data[i+1].first = first;
			data[i+1].second = second;
			data[i+1].result = (*result);
			data[i+1].length = length;
```

```
                data[i+1].start = temp;
                data[i+1].final = (temp + optimal);
                if(i == (quantity_thread - 1))
                {
                        data[i+1].final = (data[i+1].final + 1);
                }
                pthread_create(&(array_pointer_thread[i]), NULL,
Function_By_Thread, &(data[i+1]));
        }
    }
    else
    {
        for(short int i = 0, temp = 0; i < quantity_thread; i++, temp = (temp +
optimal))
        {
                data[i+1].first = first;
                data[i+1].second = second;
                data[i+1].result = (*result);
                data[i+1].length = length;
                data[i+1].start = temp;
                data[i+1].final = (temp + optimal);
                pthread_create(&(array_pointer_thread[i]), NULL,
Function_By_Thread, &(data[i+1]));
        }
    }
    for(short int i = 0; i < quantity_thread; i++)
    {
        pthread_join(array_pointer_thread[i], NULL);
    }
    free(array_pointer_thread);
```

```
	array_pointer_thread = NULL;
	for(short int i = 0, li = (i * length); i < quantity_thread; i++, li = (i * length))
	{
		for(short int j = 0; j < length; j++)
		{
			(*result)[li + j] = data[i+1].result[j];
		}
	}
	free(data);
	data = NULL;
	return 0;
}

int main(int argc, char* argv[])
{
	struct timespec time_before, time_after;
	double *matrix_first, *matrix_second, *matrix_result;
	double value_time = 0.0;
	int error_flag = 0b0;
	int quantity_thread = 0;
	int length_square_matrix = 0;
	srand(time(NULL));
	error_flag = (error_flag | Check_Argc(argc));
	error_flag = (error_flag | Convert_Char_To_Int(&quantity_thread,
argv[INDEX_QUANTITY_THREAD_FROM_ARGV]));
	error_flag = (error_flag | Convert_Char_To_Int(&length_square_matrix,
argv[INDEX_LENGTH_SQUARE_MATRIX_FROM_ARGV]));
	if(length_square_matrix > MAX_LENGTH_SQUARE_MATRIX)
	{
```

```
            return (error_flag | ERROR_SIZE_MATRIX_TOO_LARGE);
        }
        error_flag = Fill_Rand_Square_Matrix(&matrix_first,
length_square_matrix);
        error_flag = Fill_Rand_Square_Matrix(&matrix_second,
length_square_matrix);
        #ifdef PRINT_DEBUG
        printf("\nmatrix_first:\n");
        Print_Square_Matrix(matrix_first, length_square_matrix);
        printf("\nmatrix_second:\n");
        Print_Square_Matrix(matrix_second, length_square_matrix);
        #endif
        clock_gettime(CLOCK_REALTIME, &time_before);
        error_flag = (error_flag |
Multiple_Square_Matrix_With_Thread(&matrix_result, matrix_first,
matrix_second, length_square_matrix, quantity_thread));
        clock_gettime(CLOCK_REALTIME, &time_after);
        value_time = ((double)(time_after.tv_sec - time_before.tv_sec) +
(double)(time_after.tv_nsec - time_before.tv_nsec) / 1e9);
        #ifdef PRINT_DEBUG
        printf("\nmatrix_result:\n");
        Print_Square_Matrix(matrix_result, length_square_matrix);
        #endif
        free(matrix_first);
        free(matrix_result);
        free(matrix_second);
        matrix_first = NULL;
        matrix_second = NULL;
        matrix_result = NULL;
```

```
	error_flag = (error_flag | Write_In_Table_Time(quantity_thread,
value_time));
	return error_flag;
}
```

acs_5.sh

```
#!/bin/bash
echo '#Quantity Thread | Time_Execute' > none/table_time.txt;
echo '#Quantity Thread | Time_Execute' > omp/table_time.txt;
echo '#Quantity Thread | Time_Execute' > thread/table_time.txt;
cd none;
# rm table_time.txt
make clean;
make;
cd ../omp;
# rm table_time.txt
make clean;
make;
cd ../thread;
# rm table_time.txt
make clean;
make;
cd ..;
```

```
for((i=2;i<9;i=i+1))
do
echo 'iteration start'
./none/bin/key 2000
echo 'none okey'
./omp/bin/key 2000 $i;
echo 'omp okey'
./thread/bin/key 2000 $i;
echo 'iteration end';
done
gcc -Wall -Werror -o keyacel acceleration.c
./keyacel
gnuplot draw_for_acceleration.plt
xdg-open acceleration.png
```

acceleration.c

```
#include <stdio.h>

int main()
{
    FILE* file_r0 = NULL;
    FILE* file_r1 = NULL;
    FILE* file_r2 = NULL;
    FILE* file_w1 = NULL;
    FILE* file_w2 = NULL;
    double time_0 = 0.0;
    double time_1 = 0.0;
```

```
double time_2 = 0.0;
int size = 0;
file_r0 = fopen("none/table_time.txt","r");
if(file_r0 == NULL)
{
   return -1;
}
file_r1 = fopen("thread/table_time.txt","r");
if(file_r1 == NULL)
{
   return -1;
}
file_r2 = fopen("omp/table_time.txt","r");
if(file_r2 == NULL)
{
   return -1;
}
file_w1 = fopen("acc_1.txt","w");
if(file_w1 == NULL)
{
   return -1;
}
file_w2 = fopen("acc_2.txt","w");
if(file_w2 == NULL)
{
   return -1;
}
while(getc(file_r0) != '\n');
while(getc(file_r1) != '\n');
```

```
    while(getc(file_r2) != '\n');
    do
    {
        fscanf(file_r0,"%d",&size);
        fscanf(file_r1,"%d",&size);
        fscanf(file_r2,"%d",&size);
        getc(file_r0);
        getc(file_r1);
        getc(file_r2);
        fscanf(file_r0,"%lf",&time_0);
        fscanf(file_r1,"%lf",&time_1);
        fscanf(file_r2,"%lf",&time_2);
        getc(file_r0);
        getc(file_r1);
        getc(file_r2);
        fprintf(file_w1,"%d %lf\n", size, (time_0/time_1));
        fprintf(file_w2,"%d %lf\n", size, (time_0/time_2));
    }while(!feof(file_r0));
    fclose(file_w1);
    fclose(file_w2);
    fclose(file_r0);
    fclose(file_r1);
    fclose(file_r2);
    return 0;
}
```