МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Лабораторная работа №3 (вариант 2)

по дисциплине: «Высокопроизводительные вычисления».

Выполнил:
студент 3 курса, гр. ИВТВМбд-31
Долгов Н.Н.
Проверил:
д. т. н., профессор кафедры ВТ
Негода В. Н.

г. Ульяновск, 2017

Цель работы:

Изучение методов распараллеливания реализации вычисления определенного интеграла.

Задание по варианту:

Функция: *tg (x)*

Метод численного интегрирования: метод прямоугольников

Исследование многопоточных реализаций выбранного метода численного интегрирования в среде одно-, двух-, трех машинных кластеров.

В данной лабораторной работе использовалась библиотека *mpich* версии *3.2* и дополнительные утилиты для запуска.

Описание кластера.

Для кластера использовались пк со следующими характеристиками:

1. (управляющий):

Название процессора	i5-3210M
Количество ядер	2
Количество потоков	4
Базовая частота	2,5 ГГц
Кэш-память	3 Мб
Макс. частота	3,1 ГГц
ОС	Ubuntu 16.04 64bit

2.

Название процессора	i5-4210U
Количество ядер	2
Количество потоков	4
Базовая частота	1,7 ГГц
Кэш-память	3 Мб
Макс. частота	2,7 ГГц
ОС	Ubuntu 16.04 64bit

3.

Название процессора	fx8320
Количество ядер	8
Количество потоков	8

Базовая частота	3,5 ГГц
Кэш-память	8 Мб
Макс. частота	3,1 ГГц
ОС	Ubuntu 16.04 64bit

Распределение потоков по компьютерам было сделано следующим образом:

пк	Номера потоков
1	1-2
2	3-4
3	5-8

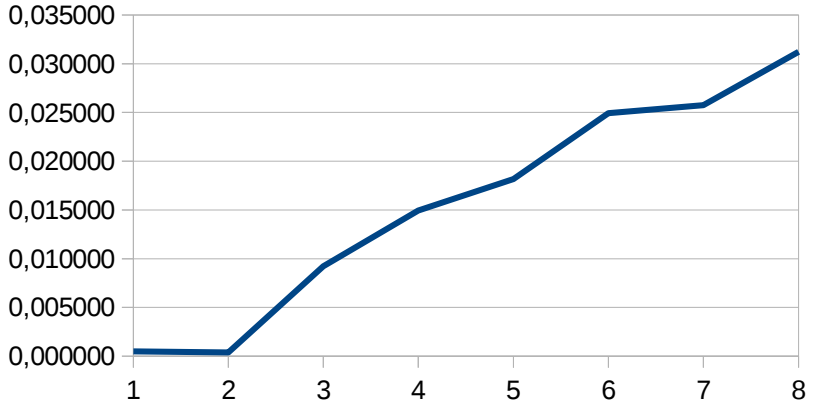
Добавление потоков происходит в порядке, представленном в таблице выше.

Таблица измерений при аргументе от 0 до 1.57

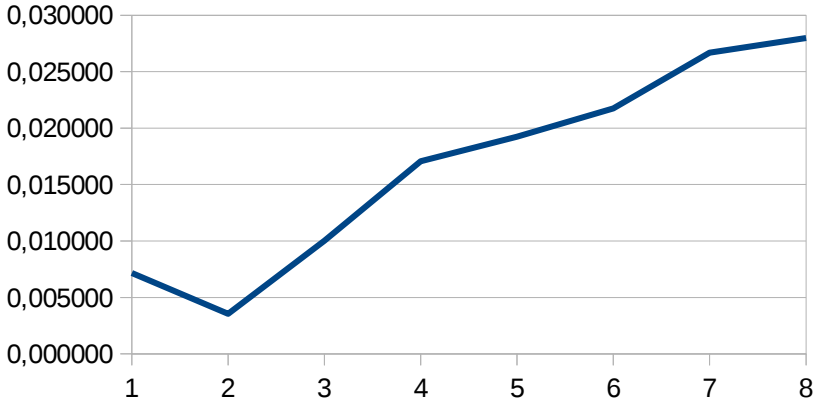
Гранулярность Число потоков	1 000	30 000	300 000	3 000 000	30 000 000	300 000 000
1	0,000500	0,007161	0,056860	0,359668	3,147530	31,270200
2	0,000391	0,003554	0,026708	0,177396	1,926550	17,735800
3	0,009206	0,010030	0,033791	0,155827	1,276960	12,633300
4	0,014947	0,017060	0,027647	0,121922	1,100760	10,770600
5	0,018175	0,019248	0,031833	0,102376	0,840029	8,134890
6	0,024928	0,021750	0,029579	0,091000	0,695006	6,722510
7	0,025761	0,026685	0,033984	0,093133	0,576880	5,459100
8	0,031215	0,027986	0,032639	0,078800	0,479641	4,534340

Графики зависимости времени от числа потоков

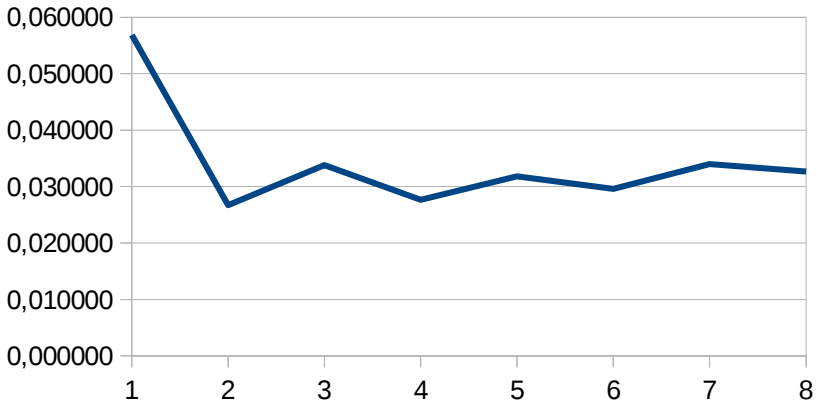
Гранулярность 1000



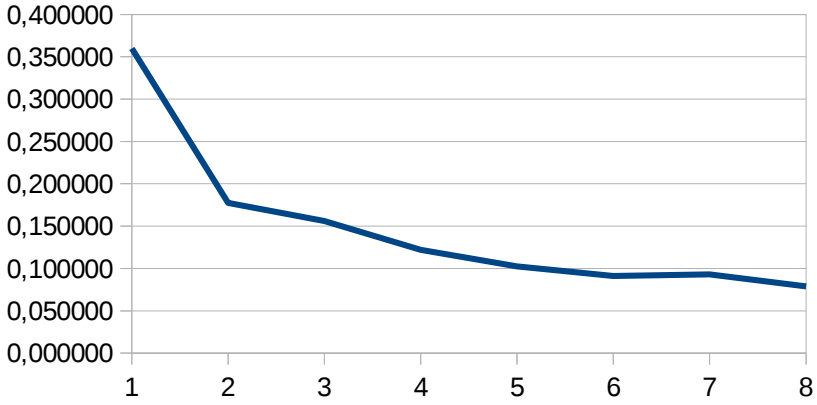
Гранулярность 30 000



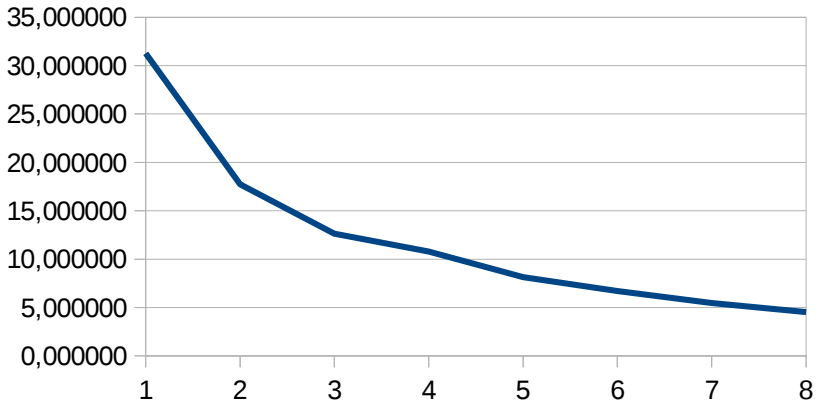
Гранулярность 300 000



Гранулярность 3 000 000



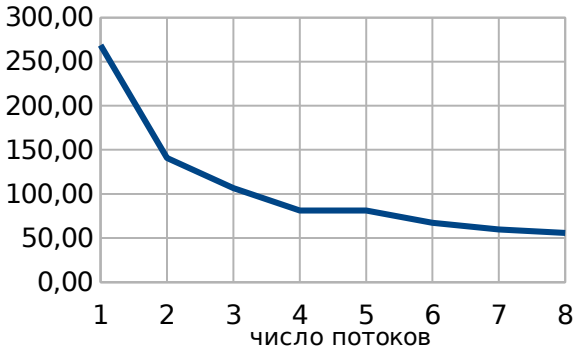
Гранулярность 30 000 000



Выводы

Как видно из графиков, при низких значениях гранулярности наблюдается снижение производительности. Это можно объяснить тем, что затраты на распределение задачи между потоками, рассылка данных по сети превосходят получаемую пользу.

Сравнение с графиками однопроцессорных машин:



При задействовании потоков только одного процессора получаем рост производительности на всем графике уже начиная с гранулярности равной 10 000.

В многомашинном кластере такой вид графика получается только при гранулярности выше 3 000 000.

Исходя из этого можно сказать, что затраты на коммуникацию между пк в кластере, оказывают значительное влияние на производительность при малых объемах вычислений.

Исходный код

```
/*
* launch mpi: mpirun -ppn 1 -n 1 -hostfile mpi_hosts ./vpv_lab3
                mpiexec -f mpi_hosts -n 4 ./vpv_lab3
 * */
#include <iostream>
#include <cmath>
#include <mpi.h>
#include <time.h>
using namespace std;
double integral(int n, double a, double b)
{
    double sum = 0;
    double d_x = (double)fabs(a - b) / n;
    {
        double x = 0;
        bool first_time = true;
        for(int i = 0; i < n; i++)
        {
```

```
            if(first_time){
                x = a + i * d_x;
                first_time = false;
            }
            sum += d_x * tan(x);
            x += d_x;
        }
    }
    return sum;
}
int main(int argc, char *argv[]){
    int n = 1000;
    double a = 0, b = 1.57;
    double res = 0;
    int process_id;
    int ierr;
    int process_num;
    MPI_Status status;
    int master = 0;
    MPI_Init (&argc, &argv);
    ierr = MPI_Comm_rank (MPI_COMM_WORLD, &process_id); //
    /*
Получение количества процессоров
*/
    ierr = MPI_Comm_size (MPI_COMM_WORLD, &process_num);
    double proc_arg[3];
    double delta = fabs(a - b) / (process_num);
    double proc_n = (double) n / (process_num);
    int tag = 1;
    clock_t start = clock();
    if(process_id == master){
        cout << "processors count " << process_num << endl;
        for (int process = 1; process < process_num; process++)
        {
            proc_arg[0] = a + delta * (process);
            proc_arg[1] = proc_arg[0] + delta;
            proc_arg[2] = proc_n;
            cout << "x1=" << proc_arg[0] << " x2=" << proc_arg[1] << endl;
            ierr = MPI_Send (proc_arg, 3, MPI_DOUBLE, process, tag, MPI_COMM_WORLD);
        }
        cout << "recive: x1=" << a << "; x2=" << a+delta << "; process_id: " << process_id << endl;
    } else{
        ierr = MPI_Recv(proc_arg, 3, MPI_DOUBLE, master, tag, MPI_COMM_WORLD, &status);
        cout << "recive: x1=" << proc_arg[0] << "; x2=" << proc_arg[1] << "; process_id: " << process_id << endl;
    }
    ierr = MPI_Barrier (MPI_COMM_WORLD);
    if(process_id != master){
        double res_l = integral(proc_arg[2], proc_arg[0], proc_arg[1]);
        int target = master;
        tag = 2;
        ierr = MPI_Send (&res_l, 1, MPI_DOUBLE, target, tag, MPI_COMM_WORLD);
    } else {
        res = integral(proc_n, a, a+delta); // master process
        for(int i = 0; i < process_num - 1; i++){
            double res_l = 0;
            tag = 2;
            ierr = MPI_Recv (&res_l, 1, MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
            res += res_l;
```

```
        }
        clock_t end = clock();
        float during = ((double)(end - start) / CLOCKS_PER_SEC);
        cout << "res " << res << "; time " << during << endl;
    }
    ierr = MPI_Finalize ();
    return 0;
}
```