МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №2

по дисциплине: **«**Программирование сетевых приложений**»**

на тему: «Организация распределённых вычислений с

использованием сокет *TCP/IP* средствами *LinuxAPI*»

Выполнил: студент гр. ИТП-41

Расшивалов Н. И.

Принял: преподаватель

Гуменников Е. Д.

Гомель 2022

**Цель:** изучить организацию распределенных вычислений с использованием *LinuxAPI*.

**Задание:**

Распределенное вычисление площади поверхности фигуры одним из

предложенных по варианту методов (в таблице варианты 1­20).

Исходными данными для вычисления являются:

– отрезок [A;B];

– точность вычислений;

– количество клиентов решающих задачу.

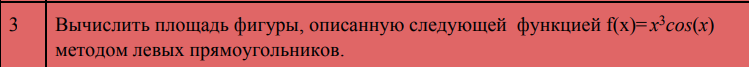


Рисунок 1 – Задание согласно варианту

**Ход работы:**

Результат вычисления с использованием одного сервера представлен на рисунке 1.

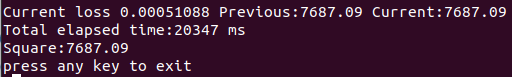


Рисунок 1 – Результат вычисления с использованием одного сервера

Результат вычисления с использованием двух серверов представлен на рисунке 2.

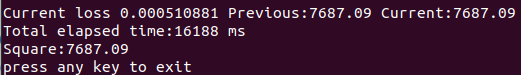


Рисунок 2 – Результат вычисления с использованием двух серверов

Результат вычисления с использованием четырех серверов представлен на рисунке 3.

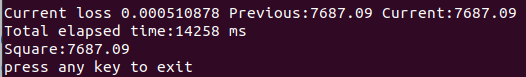


Рисунок 3 – Результат вычисления с использованием четырех серверов

Результат вычисления в онлайн калькуляторе представлен на рисунке 4.

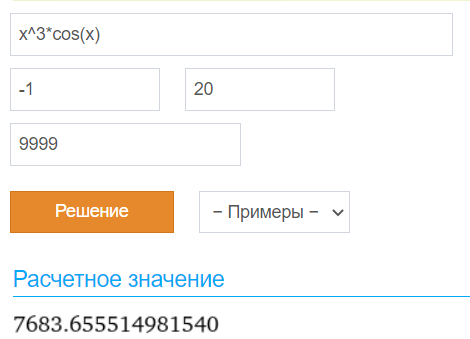


Рисунок 4 – Результат вычисления в онлайн калькуляторе

Исходя из полученных результатов можно сделать вывод что при использовании небольшого промежутка вычисление результата без использования серверов займет меньшее время чем при использовании, однако при повышении размера промежутка решение с использованием серверов производится значительно быстрее.

Листинг программы представлен в приложении А.

**Вывод:** в ходе выполнения лабораторной работы получены навыки программирования сокетов, изучены и протестированы преимущества и недостатки выполнения распределенных вычислений.

**ПРИЛОЖЕНИЕ А**

**Листинг программы**

#define UA\_ARCHITECTURE\_POSIX

#include <iostream>

#include <cstdio>

#include <cstring>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include "rapidjson/document.h"

#include "rapidjson/reader.h"

#include "rapidjson/writer.h"

#include "rapidjson/stringbuffer.h"

#include "rapidjson/memorybuffer.h"

#include <stack>

#include <thread>

#include <boost/thread.hpp>

#include <map>

#include "./common.h"

#include <chrono>

using namespace std;

using namespace rapidjson;

constexpr auto BASE\_PORT = 5001;

constexpr auto SERVERADDR = "127.0.0.1";

constexpr auto HEADER\_SIZE = sizeof(header);

constexpr auto WORKERS\_COUNT = 4;

struct poolItem {

int port;

bool isOpen;

int sock;

bool isEmpty;

};

template <typename T> class ThreadSafeStack {

public:

ThreadSafeStack() {

m\_stack = stack<poolItem>();

}

void push(const T& item) {

boost::mutex::scoped\_lock lock(m\_mutex);

m\_stack.push(item);

}

T pop() {

boost::mutex::scoped\_lock lock(m\_mutex);

if (!m\_stack.empty()) {

T item = m\_stack.top();

m\_stack.pop();

return item;

}

poolItem emptyItem;

emptyItem.isEmpty = true;

return emptyItem;

}

private:

mutable boost::mutex m\_mutex;

std::stack<T> m\_stack;

};

ThreadSafeStack<poolItem> pool;

template <typename T, typename T2> class ThreadSafeMap {

public:

ThreadSafeMap() {

m\_map = {};

}

std::map<T, T2> getAndclear() {

boost::mutex::scoped\_lock lock(m\_mutex);

std::map<T, T2> mapToReturn;

mapToReturn.insert(m\_map.begin(), m\_map.end());

m\_map = {};

return mapToReturn;

}

void push(T id, T2 value) {

boost::mutex::scoped\_lock lock(m\_mutex);

m\_map[id] = value;

}

std::map<T, T2> getMap() {

return m\_map;

}

int size() {

boost::mutex::scoped\_lock lock(m\_mutex);

return m\_map.size();

}

private:

mutable boost::mutex m\_mutex;

std::map<T, T2> m\_map;

};

ThreadSafeMap<int, double> safe\_map;

map<int, double> results\_map;

double getResult(map<int, double> results);

void process(int id, poolItem worker, double start, double end, int n);

int receive(int sock, char\* buffer, int chunk\_size)

{

int offset = 0;

while (chunk\_size > 0)

{

int n = recv(sock, buffer + offset, chunk\_size, 0);

offset += n;

chunk\_size -= n;

}

return offset;

}

double calculateIntegral(double start, double end, double step, int n);

int main() {

for (int i = 0; i < WORKERS\_COUNT; i++) {

poolItem item = poolItem();

item.isOpen = false;

item.port = BASE\_PORT + i;

pool.push(item);

}

double start;

double end;

int n;

double eps;

/\*std::cout << "Input start" << endl;

std::cin >> start;

std::cout << "Input end" << endl;

std::cin >> end;

std::cout << "Input epsilon" << endl;

std::cin >> eps;\*/

start = -1;

end = 20;

n = 1;

eps = 0.001;

double step = (end - start) / WORKERS\_COUNT;

auto start\_time = std::chrono::steady\_clock::now();

double previousResult, currentResult;

currentResult = calculateIntegral(start, end, step, n);

double loss;

do {

previousResult = currentResult;

n = 2 \* n;

currentResult = calculateIntegral(start, end, step, n);

loss = fabs(previousResult - currentResult);

std::cout << "Current loss " << loss << " Previous:" << previousResult << " Current:" << currentResult << endl;

} while (loss > eps);

auto end\_time = std::chrono::steady\_clock::now();

auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time);

std::cout << "Total elapsed time:" << elapsed\_ms.count() << " ms" << endl;

std::cout << "Square:" << fabs(currentResult) << endl;

for (int i = 0; i < WORKERS\_COUNT; i++) {

poolItem item = pool.pop();

if (!item.isEmpty) {

if (item.isOpen) {

header h = header();

h.disconnect = true;

char\* header\_data = new char[HEADER\_SIZE];

h.serialize(header\_data);

send(item.sock, header\_data, HEADER\_SIZE, 0);

close(item.sock);

}

}

}

cout << "press any key to exit" << endl;

int nothing;

cin >> nothing;

return 0;

}

double calculateIntegral(double start, double end, double step, int n) {

auto start\_time = std::chrono::steady\_clock::now();

int id = 0;

double workerStart;

double workerEnd;

for (int i = 0;i < WORKERS\_COUNT;i++) {

bool queued = false;

while (!queued) {

poolItem worker = pool.pop();

if (!worker.isEmpty) {

workerStart = start + step \* i;

workerEnd = start + step \* (i + 1);

std::thread tA(process, id, worker, workerStart, workerEnd, n);

tA.detach();

queued = true;

id = id + 1;

}

}

}

while (safe\_map.size() != WORKERS\_COUNT) {

//cout << "Map size" << results\_map.size() << endl;

}

auto end\_time = std::chrono::steady\_clock::now();

auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time);

//std::cout << "Iteration elapsed time:" << elapsed\_ms.count() << " ms" << endl;

return getResult(safe\_map.getAndclear());

}

void process(int id, poolItem worker, double start, double end, int n)

{

int socket\_desc;

socket\_desc = socket(AF\_INET , SOCK\_STREAM , 0);

//cout << "Connecting to server with port " << worker.port << endl;

if (!worker.isOpen) {

worker.sock = socket(AF\_INET, SOCK\_STREAM, 0);

worker.isOpen = true;

if (worker.sock < 0) {

cout << "Socket error" << endl;

return;

}

sockaddr\_in dest\_addr;

dest\_addr.sin\_family = AF\_INET;

dest\_addr.sin\_port = htons(worker.port);

dest\_addr.sin\_addr.s\_addr = htonl(INADDR\_LOOPBACK);

if (connect(worker.sock, (struct sockaddr\*)&dest\_addr, sizeof(dest\_addr)) < 0)

{

cout << "Connection Failed" << endl;

return;

}

}

StringBuffer s;

Writer<StringBuffer> writer(s);

writer.StartObject();

writer.Key("start");

writer.Double(start);

writer.Key("end");

writer.Double(end);

writer.Key("n");

writer.Int(n);

writer.EndObject();

header h = header();

h.size = s.GetSize();

//cout << s.GetString() << endl;

char\* header\_data = new char[HEADER\_SIZE];

h.serialize(header\_data);

send(worker.sock, header\_data, HEADER\_SIZE, 0);

if (send(worker.sock, s.GetString(), s.GetSize(), 0) == SO\_ERROR) {

cout << "send failed" << endl;

}

char headerBuffer[HEADER\_SIZE] = { 0 };

receive(worker.sock, headerBuffer, HEADER\_SIZE);

header responseHeader;

responseHeader.deserialize(headerBuffer);

char\* buffer = new char[responseHeader.size];

buffer[responseHeader.size] = '\0';

receive(worker.sock, buffer, responseHeader.size);

//cout << "Given response json:" << buffer << endl;

Document d;

d.Parse(buffer);

pool.push(worker);

safe\_map.push(id, d["result"].GetDouble());

cout << "finished" << id << endl;

//cout << "added response with id:" << id << endl;

}

double getResult(map<int, double> results)

{

map <int, double> ::iterator it = results.begin();

double result = 0.0;

for (int i = 0; it != results.end(); it++, i++)

{

result = result + it->second;

}

return result;

}

#include <iostream>

#include <cstdio>

#include <cstring>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include "rapidjson/document.h"

#include "rapidjson/reader.h"

#include "rapidjson/writer.h"

#include "rapidjson/stringbuffer.h"

#include "rapidjson/memorybuffer.h"

#include <stack>

#include <thread>

#include "./common.h"

#include <cmath>

using namespace std;

using namespace rapidjson;

constexpr auto BASE\_PORT = 5001;

constexpr auto SERVERADDR = "127.0.0.1";

constexpr auto HEADER\_SIZE = sizeof(header);

int clientsCount = 0;

int receive(int sock, char\* buffer, int chunk\_size);

int process(int client\_socket);

typedef double(\*pointFunc)(double);

double f(double x) {

return pow(x, 3) \* cos(x);

}

double integral(pointFunc f, double start, double end, int n) {

double x, step;

double sum = 0.0;

double fx;

step = (end - start) / n;

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

x = start + i \* step;

fx = f(x);

sum += fx;

}

return (sum \* step);

}

int main()

{

int serverSocket;

if ((serverSocket = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

cout << "Error socket" << endl;

return 1;

}

int serverNumber = 0;

cout << "Input server number: " << endl;

cin >> serverNumber;

int serverPort = BASE\_PORT + serverNumber;

cout << "Will listen on " << serverPort << " port";

sockaddr\_in serverAddr;

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(serverPort);

serverAddr.sin\_addr.s\_addr = htonl(INADDR\_LOOPBACK);

if (bind(serverSocket, (sockaddr\*)&serverAddr, sizeof(serverAddr)) == SO\_ERROR) {

cout << "Bind function failed with error" << endl;

return -1;

}

if (listen(serverSocket, 0) == SO\_ERROR) {

cout << "Listen function failed with error" << endl;

return -1;

}

cout << "Waiting for connections..." << endl;

int clientSocket;

sockaddr\_in client\_addr;

int client\_addr\_size = sizeof(client\_addr);

while ((clientSocket = accept(serverSocket, NULL, NULL))) {

clientsCount++;

//cout << "Connected client" << endl;

std::thread tA(process, clientSocket);

tA.detach();

}

return 0;

}

int process(int lpParam) {

int client\_socket;

client\_socket = lpParam;

char headerBuffer[HEADER\_SIZE] = { 0 };

char header\_data[HEADER\_SIZE] = { 0 };

header requestHeader;

int bytes\_received = 0;

while (true) {

bytes\_received = receive(client\_socket, headerBuffer, HEADER\_SIZE);

requestHeader.deserialize(headerBuffer);

cout << requestHeader.disconnect << endl;

if (requestHeader.disconnect == true) {

cout << "Client disconnected";

break;

}

if (bytes\_received == 0) {

continue;

}

char\* buffer = new char[requestHeader.size];

buffer[requestHeader.size] = '\0';

receive(client\_socket, buffer, requestHeader.size);

cout << "Given json:" << buffer << endl;

Document d;

d.Parse(buffer);

double start = d["start"].GetDouble();

double end = d["end"].GetDouble();

int n = d["n"].GetInt();

StringBuffer s;

Writer<StringBuffer> writer(s);

writer.StartObject();

writer.Key("result");

writer.Double(integral(f, start, end, n));

writer.EndObject();

header responseHeader = header();

responseHeader.size = s.GetSize();

responseHeader.serialize(header\_data);

send(client\_socket, header\_data, HEADER\_SIZE, 0);

send(client\_socket, s.GetString(), s.GetSize(), 0);

cout << "Sent result:" << s.GetString() << endl;

//delete buffer;

}

clientsCount--;

return 0;

}

int receive(int sock, char\* buffer, int chunk\_size)

{

int offset = 0;

while (chunk\_size > 0)

{

int n = recv(sock, buffer + offset, chunk\_size, 0);

offset += n;

chunk\_size -= n;

}

return offset;

}

struct header {

bool disconnect;

int size;

void serialize(char\* data) {

memcpy(data, this, sizeof(header));

}

void deserialize(char\* data) {

memcpy(this, data, sizeof(header));

}

};