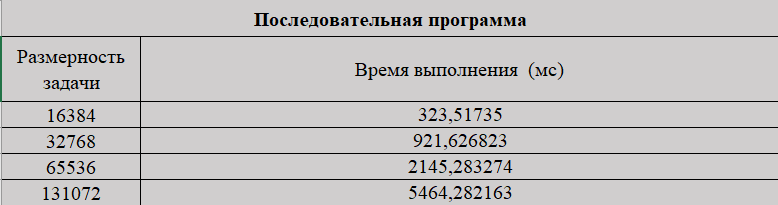
# **Лабораторная работа 3. Жуковский Павел, 12 группа**

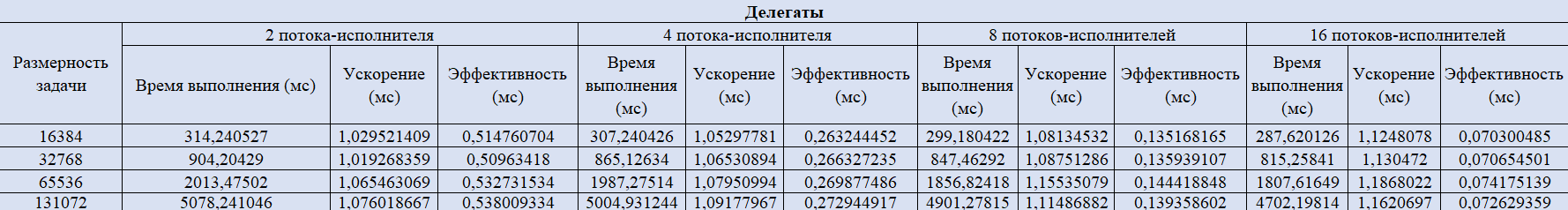
Если какая-то картинка плохо видна, то можно использовать сочетание ctrl + (вращение колёсика мыши) для увеличения.

**Таблицы с расчётами и временными оценками**

Последовательная программа:



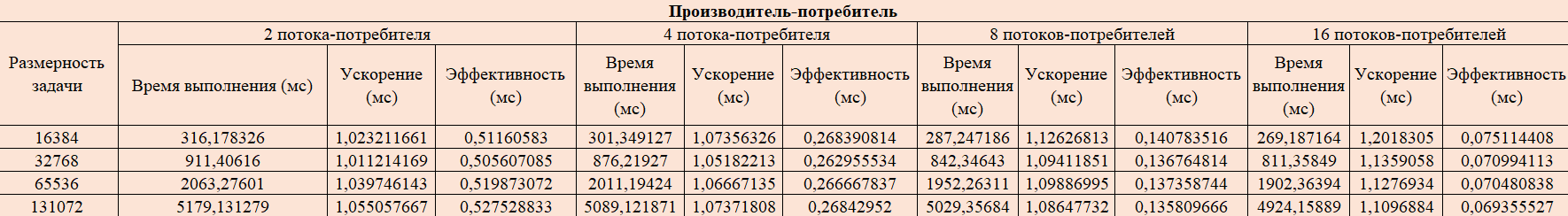
Делегаты:

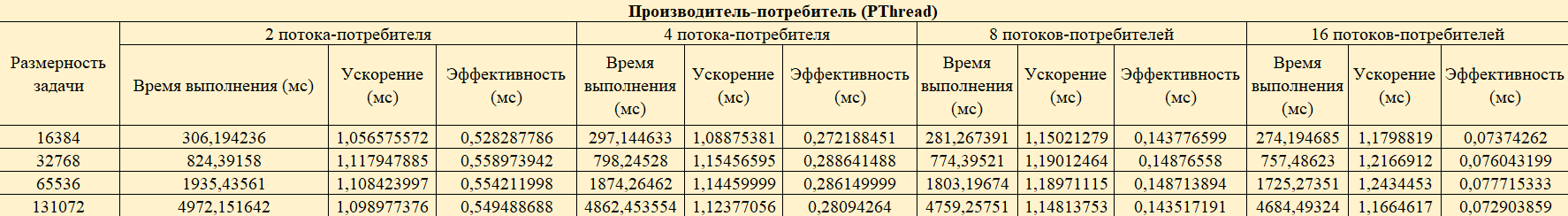


Делегаты (PThread):



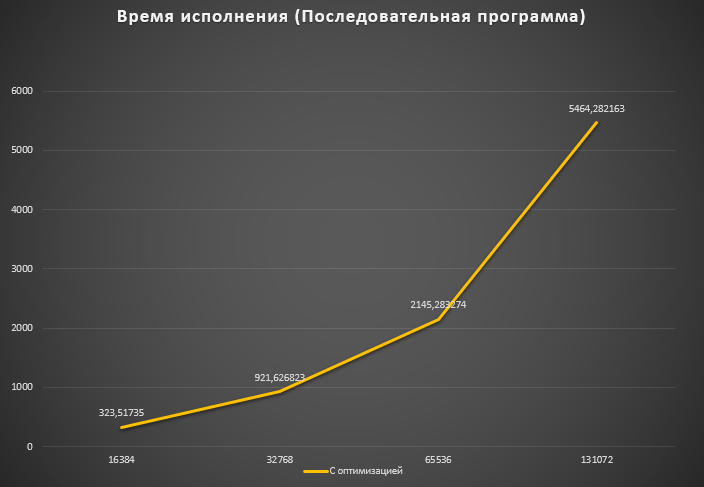
Производитель-потребители:



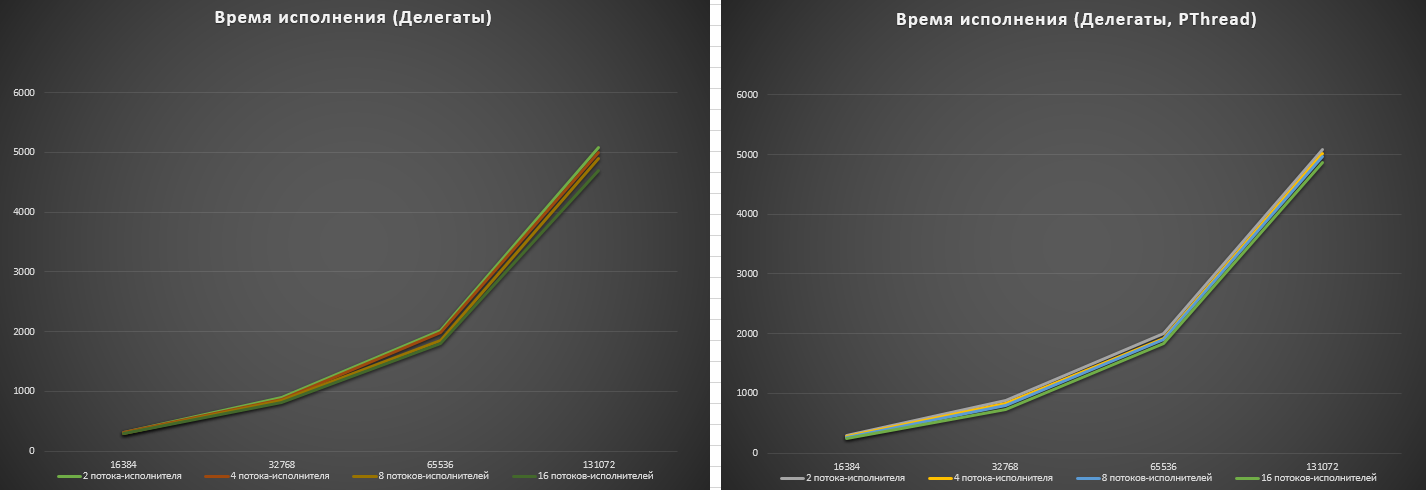


**Графики**

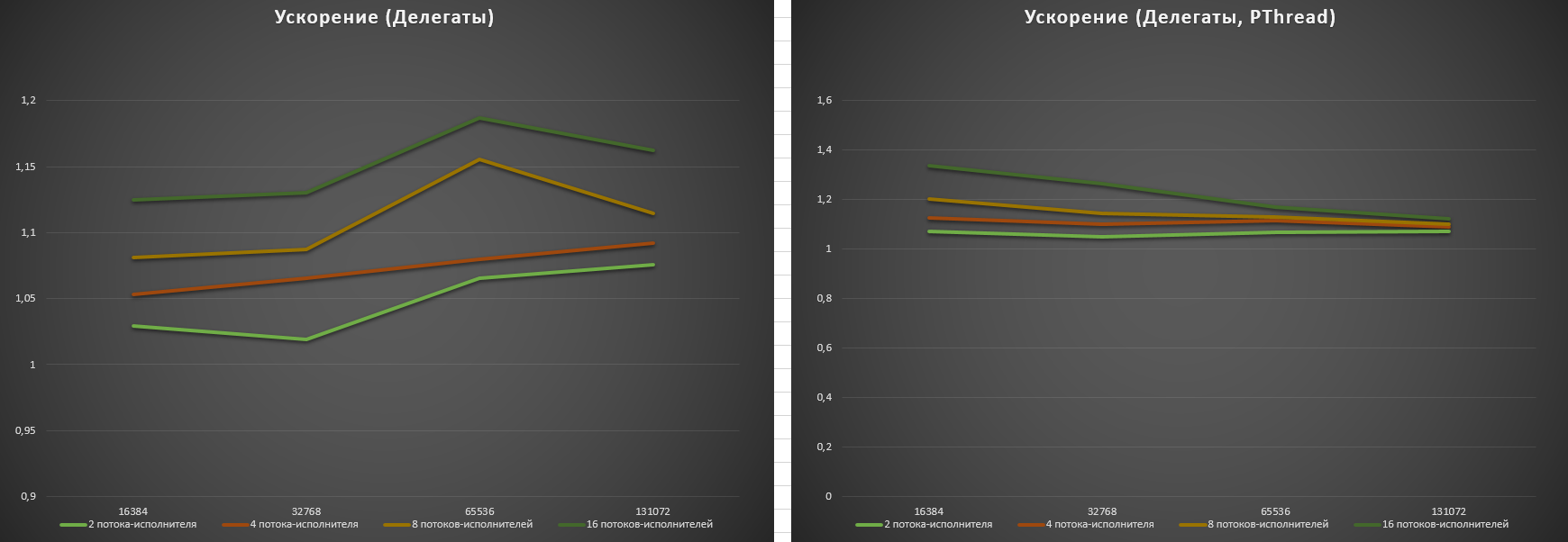
Последовательная программа (время выполнения):



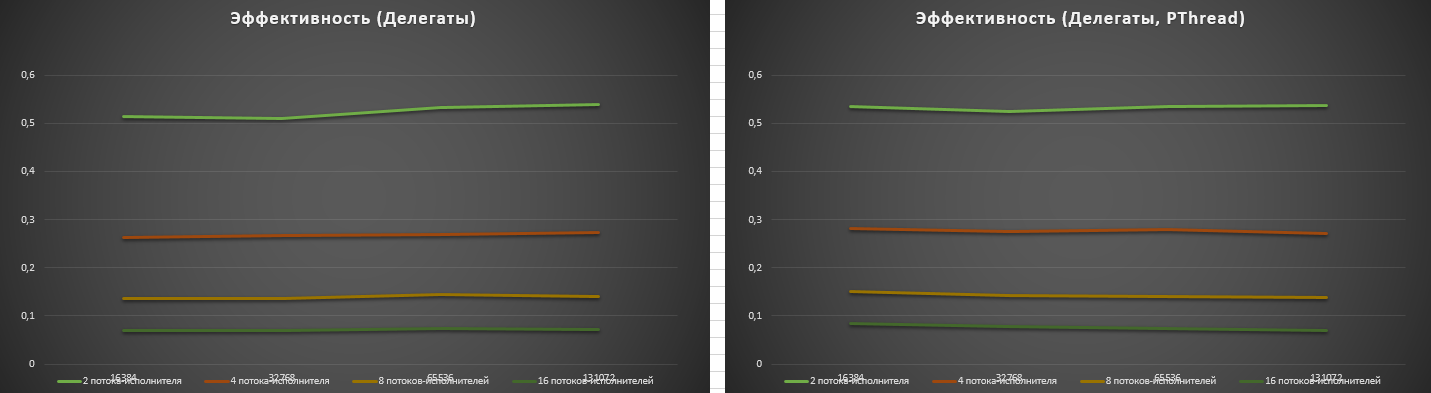
Делегаты (время исполнения):



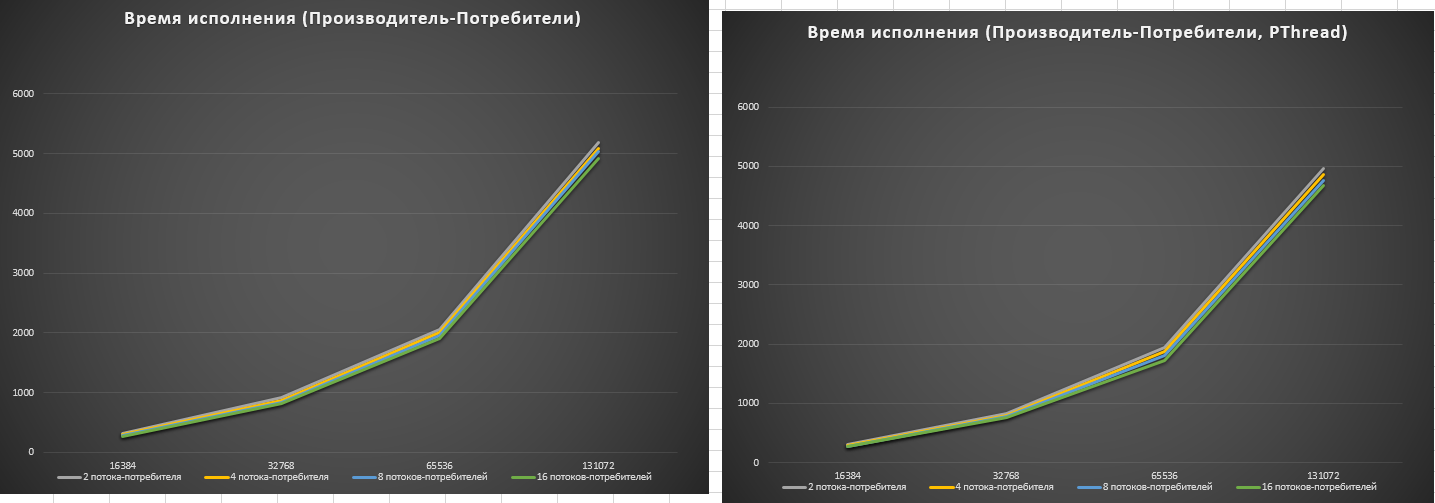
Ускорение делегатов относительно последовательной программы:



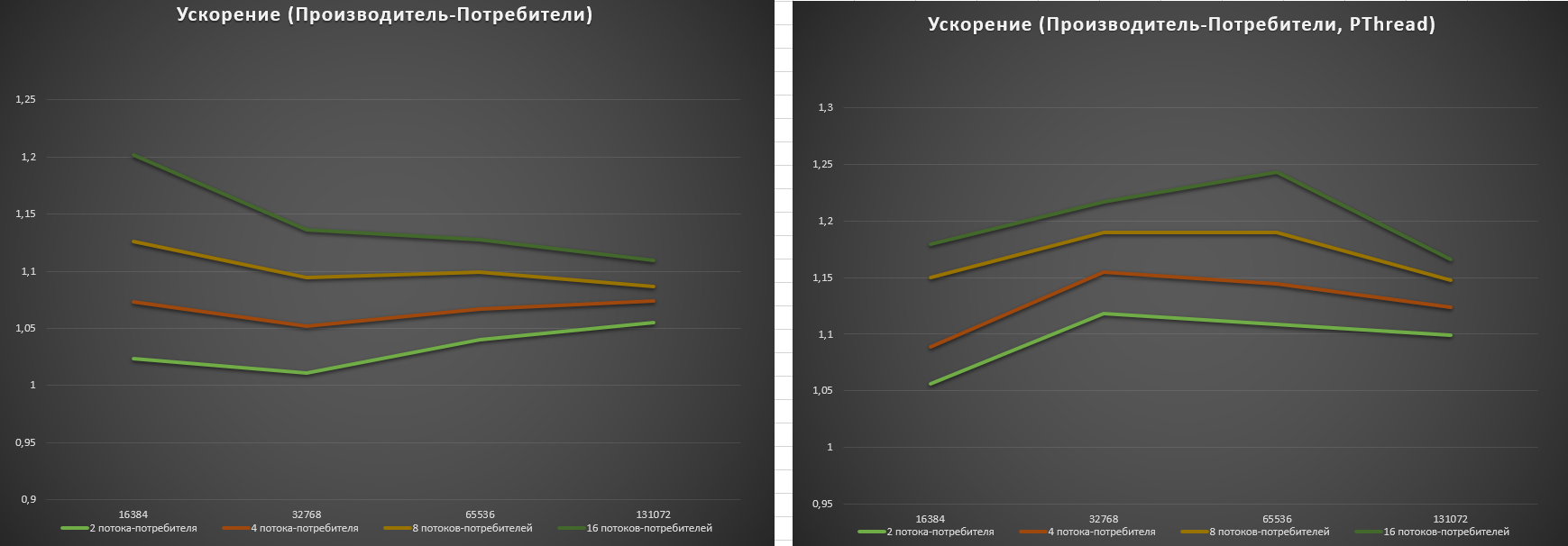
Эффективность делегатов относительно последовательной программы:



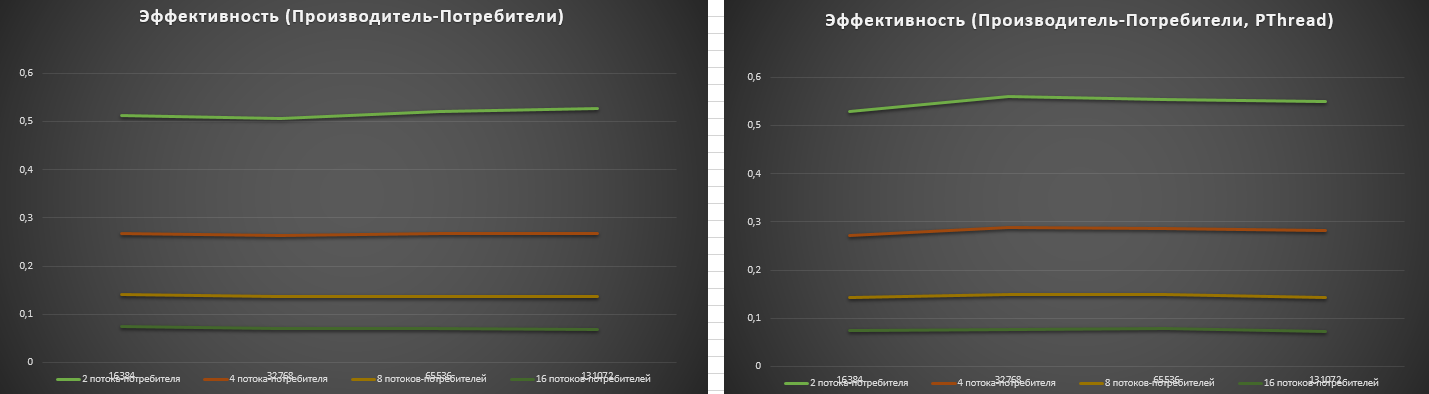
Производитель-потребители (время исполнения):



Ускорение производителя-потребителей относительно последовательной программы:



Эффективность производителя-потребителей относительно последовательной программы:



**Исходный код**

Делегаты:

#include <cstring>

#include <cstdio>

#include <string>

#pragma warning(disable : 4996)

void MergeSortedPartsInCash(FILE\* file, long start, long med, long finish)

{

int\* buffer = new int[finish - start];

fseek(file, start \* sizeof(int), SEEK\_SET);

fread(buffer, sizeof(int), finish - start, file);

fseek(file, start \* sizeof(int), SEEK\_SET);

int\* firstPart = buffer;

int\* secondPart = buffer + (med - start);

int\* firstEnd = secondPart;

int\* secondEnd = buffer + (finish - start);

while (start < finish)

{

if (firstPart >= firstEnd)

{

fwrite(secondPart, sizeof(int), 1, file);

++secondPart;

}

else if (secondPart >= secondEnd)

{

fwrite(firstPart, sizeof(int), 1, file);

++firstPart;

}

else

{

if (\*firstPart < \*secondPart)

{

fwrite(firstPart, sizeof(int), 1, file);

++firstPart;

}

else

{

fwrite(secondPart, sizeof(int), 1, file);

++secondPart;

}

}

start++;

}

delete[] buffer;

}

void MergeSortedPartsInFiles(FILE\* file, long start, long med, long finish)

{

FILE\* first = fopen((std::to\_string(start) + "\_" + std::to\_string(med)).c\_str(), "wb+");

FILE\* second = fopen((std::to\_string(med) + "\_" + std::to\_string(finish)).c\_str(), "wb+");

int firstValue;

int secondValue;

fseek(file, start \* sizeof(int), SEEK\_SET);

for (int i = start; i < finish; ++i)

{

fread(&firstValue, sizeof(int), 1, file);

fwrite(&firstValue, sizeof(int), 1, i < med ? first : second);

}

fseek(file, start \* sizeof(int), SEEK\_SET);

fseek(first, 0, SEEK\_SET);

fseek(second, 0, SEEK\_SET);

long firstLeft = med - start;

long secondLeft = finish - med;

while (start < finish)

{

if (firstLeft == 0)

{

fread(&secondValue, sizeof(int), 1, second);

fwrite(&secondValue, sizeof(int), 1, file);

}

else if (secondLeft == 0)

{

fread(&firstValue, sizeof(int), 1, first);

fwrite(&firstValue, sizeof(int), 1, file);

}

else

{

fread(&firstValue, sizeof(int), 1, first);

fread(&secondValue, sizeof(int), 1, second);

if (firstValue < secondValue)

{

fwrite(&firstValue, sizeof(int), 1, file);

fseek(second, sizeof(int), SEEK\_CUR);

--firstLeft;

}

else

{

fwrite(&secondValue, sizeof(int), 1, file);

fseek(first, sizeof(int), SEEK\_CUR);

--secondLeft;

}

}

start++;

}

fclose(first);

fclose(second);

}

void MergeSortedParts(FILE\* file, long start, long med, long finish, long memoryLimit)

{

if (finish - start < memoryLimit)

MergeSortedPartsInCash(file, start, med, finish);

else

MergeSortedPartsInFiles(file, start, med, finish);

}

#include <iostream>

#include <windows.h>

#include <profileapi.h>

#include "SortMerge.cpp"

#define ENTITY\_LIMIT\_IN\_MEMORY 8192

#pragma warning(disable : 4996)

void MergeSort(FILE\* file, long start, long finish)

{

long len = finish - start;

if (len)

return;

long med = start + len / 2;

MergeSort(file, start, med);

MergeSort(file, med, finish);

MergeSortedParts(file, start, med, finish, ENTITY\_LIMIT\_IN\_MEMORY);

}

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

{

const int startN = 16384;

const int maxN = startN \* 8;

FILE\* defaultFile = fopen("default", "wb");

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

{

int value = rand();

fwrite(&value, sizeof(int), 1, defaultFile);

}

fclose(defaultFile);

for (int N = startN; N <= maxN; N \*= 2)

{

double total = 0.0;

const int runs = argc == 2 ? atoi(argv[1]) : 10;

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

{

system("del main > nul");

system("copy default main > nul");

FILE\* main = fopen("main", "rb+");

fseek(main, 0, SEEK\_SET);

LARGE\_INTEGER liFrequency, liStartTime, liFinishTime;

QueryPerformanceFrequency(&liFrequency);

QueryPerformanceCounter(&liStartTime);

MergeSort(main, 0, N);

QueryPerformanceCounter(&liFinishTime);

fclose(main);

total += 1000. \* (liFinishTime.QuadPart - liStartTime.QuadPart) / liFrequency.QuadPart;

}

printf("N: %d. Average: %lfms.\n", N, total / runs);

}

return 0;

}

Производитель-потребители:

#include <cstdio>

#include <windows.h>

typedef struct

{

int seq;

long start;

long med;

long finish;

HANDLE firstDepfinishency;

HANDLE secondDepfinishency;

HANDLE ready;

} Task;

class TaskQueue

{

public:

TaskQueue(int len);

virtual ~TaskQueue();

void Insert(Task element);

bool Remove(Task\* output, DWORD emptyWaitMs);

private:

HANDLE insertSemaphore\_;

HANDLE removeSemaphore\_;

CRITICAL\_SECTION editingGuard\_;

int outputIndex\_;

int startIndex\_;

int len\_;

Task\* array\_;

};

TaskQueue::TaskQueue(int len)

{

insertSemaphore\_ = CreateSemaphore(NULL, len, len, NULL);

removeSemaphore\_ = CreateSemaphore(NULL, 0, len, NULL);

InitializeCriticalSection(&editingGuard\_);

outputIndex\_ = 0;

startIndex\_ = 0;

len\_ = len;

array\_ = new Task[len\_];

}

TaskQueue::~TaskQueue()

{

CloseHandle(insertSemaphore\_);

CloseHandle(removeSemaphore\_);

DeleteCriticalSection(&editingGuard\_);

delete[] array\_;

}

void TaskQueue::Insert(Task element)

{

WaitForSingleObject(insertSemaphore\_, INFINITE);

EnterCriticalSection(&editingGuard\_);

ReleaseSemaphore(removeSemaphore\_, 1, NULL);

array\_[outputIndex\_] = element;

++outputIndex\_;

if (outputIndex\_ >= len\_)

{

outputIndex\_ -= len\_;

}

LeaveCriticalSection(&editingGuard\_);

}

bool TaskQueue::Remove(Task\* output, DWORD emptyWaitMs)

{

DWORD removeWaitResult = WaitForSingleObject(removeSemaphore\_, emptyWaitMs);

if (removeWaitResult != WAIT\_OBJECT\_0)

{

return false;

}

EnterCriticalSection(&editingGuard\_);

ReleaseSemaphore(insertSemaphore\_, 1, NULL);

\*output = array\_[startIndex\_];

++startIndex\_;

if (startIndex\_ >= len\_)

{

startIndex\_ -= len\_;

}

LeaveCriticalSection(&editingGuard\_);

return true;

}

#include <iostream>

#include <windows.h>

#include <profileapi.h>

#include <vector>

#include <process.h>

#include "SortMerge.cpp"

#include "RequestQueue.cpp"

#define ENTITY\_LIMIT\_IN\_MEMORY 8192

bool allTasksGenerated = false;

int T;

void MergeSortOrder(FILE\* file, long start, long finish)

{

long len = finish - start;

if (len)

return;

long med = start + len / 2;

MergeSortOrder(file, start, med);

MergeSortOrder(file, med, finish);

MergeSortedParts(file, start, med, finish, ENTITY\_LIMIT\_IN\_MEMORY / T);

}

unsigned int \_\_stdcall MergeSortSorterThread(void\* arg)

{

auto\* queue = (TaskQueue\*)arg;

Task task;

FILE\* main = fopen("main", "rb+");

while (true)

{

if (!queue->Remove(&task, 3))

if (allTasksGenerated)

break;

else

continue;

if (task.firstDepfinishency != INVALID\_HANDLE\_VALUE)

WaitForSingleObject(task.firstDepfinishency, INFINITE);

if (task.secondDepfinishency != INVALID\_HANDLE\_VALUE)

WaitForSingleObject(task.secondDepfinishency, INFINITE);

if (task.seq)

MergeSortOrder(main, task.start, task.finish);

else

MergeSortedParts(main, task.start, task.med, task.finish, ENTITY\_LIMIT\_IN\_MEMORY / T);

SetEvent(task.ready);

CloseHandle(task.firstDepfinishency);

CloseHandle(task.secondDepfinishency);

}

fclose(main);

return 0;

}

HANDLE MergeSortTaskGenerator(long start, long finish, TaskQueue\* output, int parallelBarrier)

{

int len = finish - start;

if (len)

return INVALID\_HANDLE\_VALUE;

long med = start + len / 2;

HANDLE readyEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (len > parallelBarrier)

{

HANDLE firstDep = MergeSortTaskGenerator(start, med, output, parallelBarrier);

HANDLE secondDep = MergeSortTaskGenerator(med, finish, output, parallelBarrier);

output->Insert({ 0, start, med, finish, firstDep, secondDep, readyEvent });

}

else

output->Insert({ 1, start, med, finish, INVALID\_HANDLE\_VALUE, INVALID\_HANDLE\_VALUE, readyEvent });

return readyEvent;

}

typedef struct

{

long start;

long finish;

TaskQueue\* output;

int parallelBarrier;

} GeneratorArgs;

unsigned int \_\_stdcall MergeSortTaskGeneratorThread(void\* arg)

{

auto\* arguments = (GeneratorArgs\*)arg;

allTasksGenerated = false;

HANDLE lastEvent = MergeSortTaskGenerator(

arguments->start, arguments->finish, arguments->output, arguments->parallelBarrier);

allTasksGenerated = true;

WaitForSingleObject(lastEvent, INFINITE);

CloseHandle(lastEvent);

return 0;

}

/// @param parallelBarrier Parts that are lower that this barrier will be sorted sequentially.

void MergeSort(long start, long finish, int parallelBarrier)

{

auto\* threads = new HANDLE[T + 1];

GeneratorArgs args;

TaskQueue taskQueue(100);

args.start = start;

args.finish = finish;

args.output = &taskQueue;

args.parallelBarrier = parallelBarrier;

threads[0] = (HANDLE)\_beginthreadex(NULL, 0, MergeSortTaskGeneratorThread, &args, 0, NULL);

for (int i = 1; i < T + 1; ++i)

{

threads[i] = (HANDLE)\_beginthreadex(NULL, 0, MergeSortSorterThread, &taskQueue, 0, NULL);

}

WaitForMultipleObjects(T + 1, threads, TRUE, INFINITE);

delete[] threads;

}

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

{

const int startN = 16384;

const int maxN = startN \* 8;

FILE\* defaultFile = fopen("default", "wb");

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

{

int value = rand();

fwrite(&value, sizeof(int), 1, defaultFile);

}

fclose(defaultFile);

for (T = 2; T <= 16; T \*= 2)

{

printf("T: %d.\n", T);

for (int N = startN; N <= maxN; N \*= 2)

{

double total = 0.0;

const int runs = argc == 2 ? atoi(argv[1]) : 10;

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

{

system("del main > nul");

system("copy default main > nul");

LARGE\_INTEGER liFrequency, liStartTime, liFinishTime;

QueryPerformanceFrequency(&liFrequency);

QueryPerformanceCounter(&liStartTime);

MergeSort(0, N, ENTITY\_LIMIT\_IN\_MEMORY / T);

QueryPerformanceCounter(&liFinishTime);

total += 1000. \* (liFinishTime.QuadPart - liStartTime.QuadPart) / liFrequency.QuadPart;

}

printf(" N: %d. Average: %lfms.\n", N, total / runs);

}

}

return 0;

}