САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Дисциплина: Архитектура компьютера

Отчет

по лабораторной работе №4

**«OpenMP»**

Выполнил(а): Кузнец Антон Сергеевич

студ. гр. Б10

Санкт-Петербург

2021

**Цель работы:** знакомство со стандартом OpenMP.

**Теоретическая часть**

**OpenMP** – механизм написания параллельных программ для систем с общей памятью. Состоит из набора директив компилятора и библиотечных функций.

Основной поток порождает дочерние потоки по мере необходимости. Модель fork-join. Программирование путем вставки директив компилятора в ключевые места исходного кода программы. Компилятор интерпретирует эти директивы и вставляет в соответствующие места программы библиотечные вызовы для расспараллеливания участков кода.

**shared(var1, var2, ...)** Условие shared указывает на то, что все перечисленные переменные будут разделяться между потоками. Все потоки будут доступаться к одной и той же области памяти.

**private(var1, var2, ...)** Условие private указывает на то, что каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.

**firstprivate(var1, var2, ...)** Это условие аналогично условию private за тем исключением, что указанные переменные инициализируются при входе в параллельный участок кода значением, которое имела переменна до входа в параллельную секцию.

Переменные, которые подпадают под это условие являются приватными для каждого потока, но перед выполнением потока происходит их инициализация значением, которое было получено в предыдущем последовательном коде. Так в следующем примере до входа в параллельный участок кода значение переменной а равнялось 10. Это же значение имеет эта переменна и при входе в параллельный участок кода.

**lastprivate(var1, var2, ...)** Приватные переменные сохраняют свое значение, которое они получили при достижении конца параллельного участка кода. reduction(оператор:var1, var2, ...) Это условие гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.

Это условие действует аналогично условию private за тем исключением, что значение переменной, вычисленное на последней итерации цикла, сохраняется.

**if(выражение)** Это условие говорит о том, что параллельное выполнение необходимо только если выражение истино.

В тех случаях, когда накладные расходы на порождение потоков могут быть больше, чем выигрыш от распараллеливания, то необходимо воспользоваться условием if.

**default(shared|private|none)** Это условие определяет область видимости переменных внутри параллельного участка кода по умолчанию. **schedule(type[,chank])** Этим условием контролируется то, как итерации цикла распределяются между потоками.

Размер блока задает размер каждого пакета на обработку потоком (количество итераций). Тип расписания может принимать следующие значение:

**static** – итерации равномерно распределяются по потокам. Т.е. если в цикле 1000 итераций и 4 потока, то один поток обрабатывает все итерации с 1 по 250, второй – с 251 по 500, третий - с 501 по 750, четвертый с 751 по 1000. Если при этом задан еще и размер блока, то все итерации блоками заданного размера циклически распределяются между потоками. Статическое распределение работы эффективно, когда время выполнения итераций равно, или приблизительно равно. Если это не так, то разумно использовать следующий тип распределения работ.

**dynamic** – работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками. Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую. Стоит отметить, что при этом подходе несколько большие накладные расходы, но можно добиться лучшей балансировки загрузки между потоками.

**guided** – данный тип распределения работы аналогичен предыдущему, за тем исключением, сто размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения. При таком подходе можно достичь хорошей балансировки при меньших накладных расходах.

**runtime** – тип распределения определяется в момент выполнения программы. Это удобно в экспериментальных целях для выбора оптимального значения типа и размера блока. Тип распределения работ зависит от переменной окружения OMP\_SCHEDULE. По умолчанию считается, что установлен статический метод распределения работ.

**reduction** Это условие позволяет производить безопасное глобальное вычисление. Приватная копия каждой перечисленной переменной инициализируется при входе в параллельную секцию в соответствии с указанным оператором (0 для оператора +). При выходе из параллельной секции из частично вычисленных значений вычисляется результирующее и передается в основной поток.

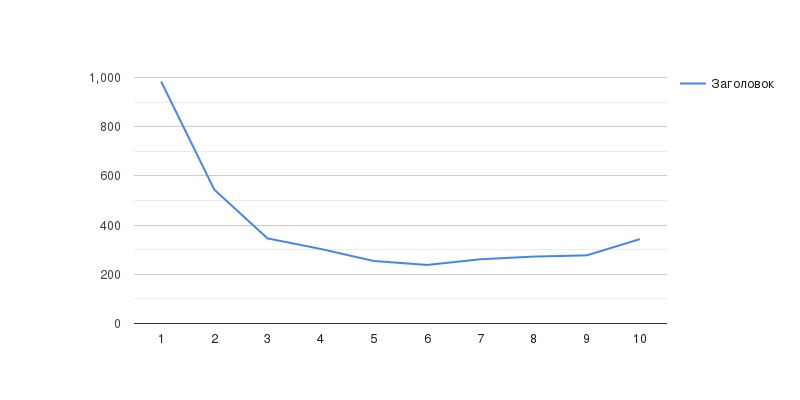
**Практическая часть**

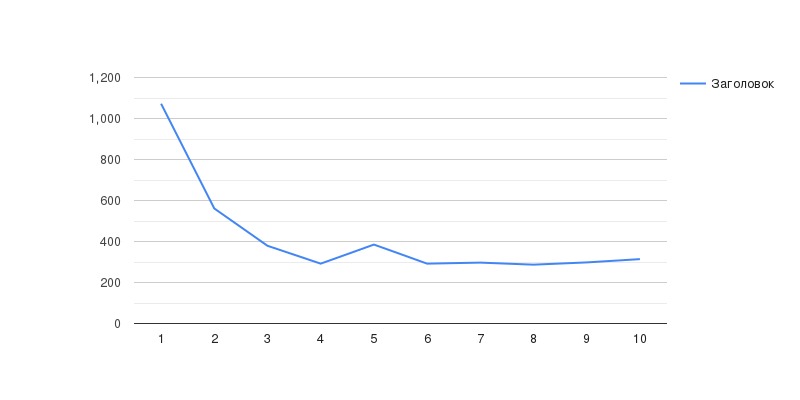
Алгоритм читает содержимое файла. Далее по матрице изображения вычисляет минимальные и максимальные значения каналов r, g, b (в случае P6). Затем исходные значения для каждого пикселя пересчитываются по формулам:

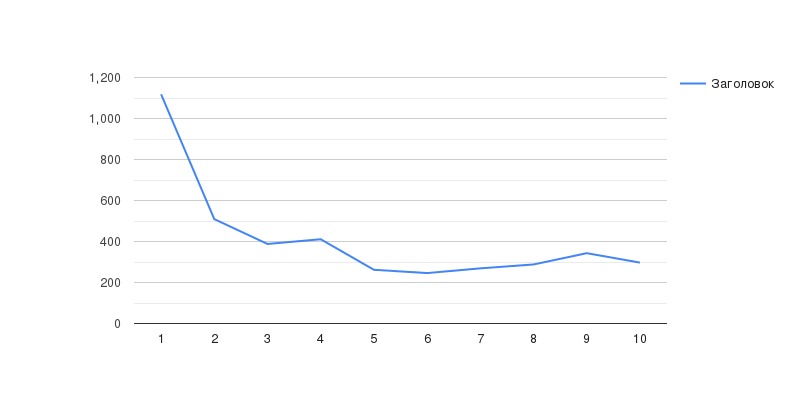
r = (r - minr) \* max\_val / (maxr - minr)  
g = (g - ming) \* max\_val / (maxg - ming)  
b = (b - minb) \* max\_val / (maxb - minb)

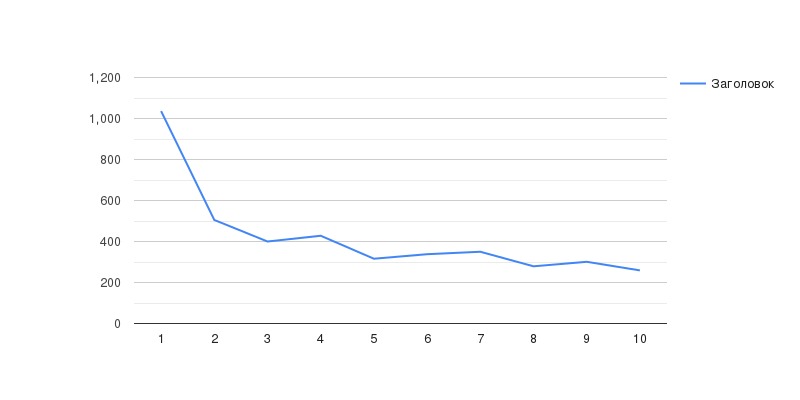
Время работы без omp: 419 мс

Время работы при разном количестве потоков при фиксированном значении chunk\_size:

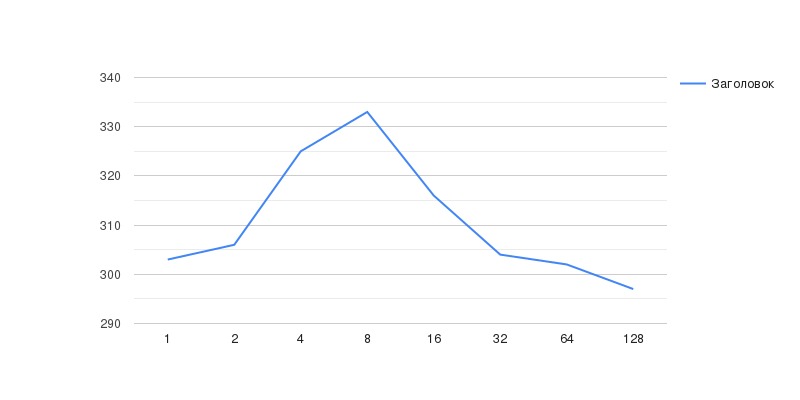
chunk\_size не указан

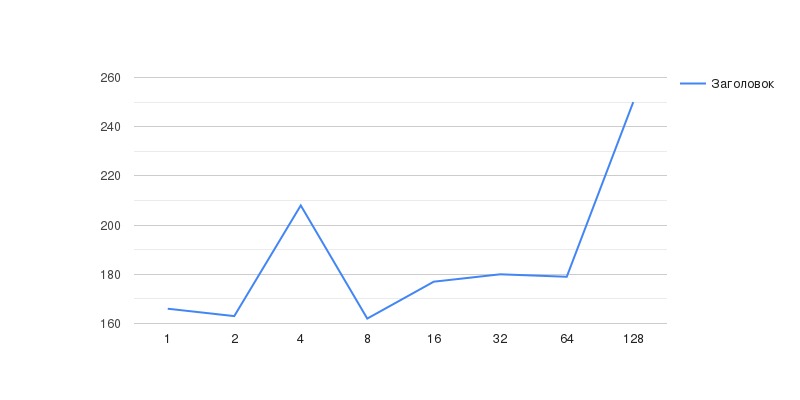
chunk\_size = 4

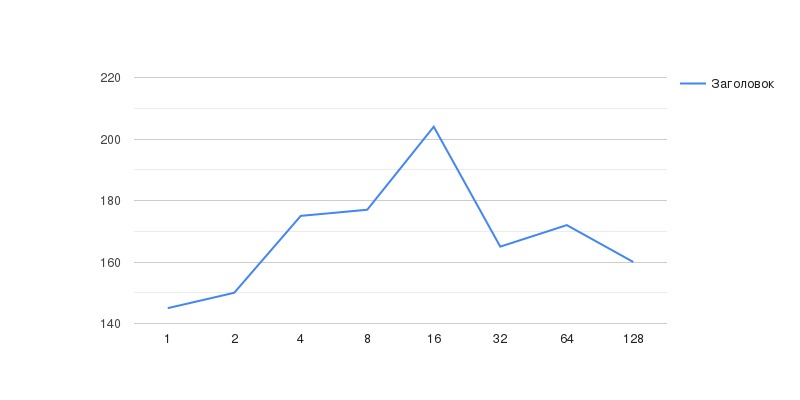
chunk\_size = 32

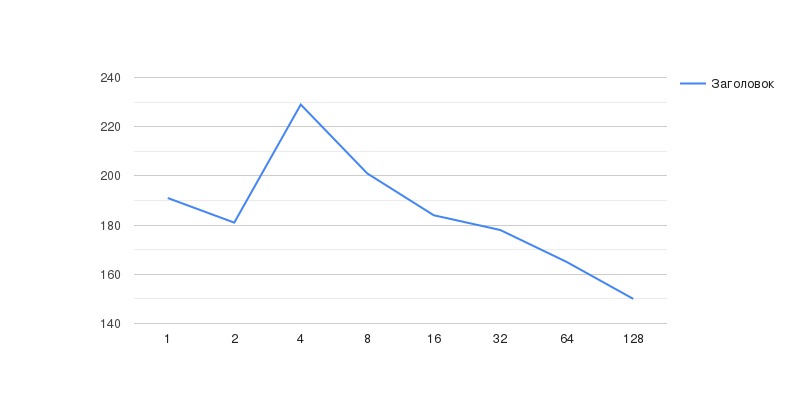
chunk\_size = 64

Время работы при различных значениях chunk\_size при фиксированном количестве потоков:

2 потока

5 потоков

10 потоков

20 потоков

**Листинг**

**main.cpp**

#include <iostream>  
#include <fstream>  
#include <cstring>  
#include <ctime>  
#include <omp.h>  
  
int main(int argc, char \*argv[]) {  
 unsigned int num\_threads = std::stoul(argv[1]);  
 std::ifstream f(argv[2], std::ios::binary|std::ios::ate);  
 int length = f.tellg();  
 char \*buffer = new char[length];  
 f.seekg(0, std::ios::beg);  
 f.read(buffer, length);  
 f.close();  
 char \*bitmap = strchr(strchr(strchr(buffer, '\n') + 1, '\n') + 1, '\n') + 1;  
  
 char \*p = strtok(buffer, "\n");  
 std::string pnm\_type(p);  
 p = strtok(nullptr, " ");  
 unsigned int width = std::stoul(p);  
 p = strtok(nullptr, "\n");  
 unsigned int height = std::stoul(p);  
 p = strtok(nullptr, "\n");  
 unsigned int max\_val = std::stoul(p);  
  
 if(pnm\_type == "P5") {  
 int beg = clock();  
 uint8\_t ming = max\_val, maxg = 0;  
 #pragma omp parallel for reduction(min:ming) reduction(max:maxg) shared(bitmap) schedule(static) num\_threads(num\_threads)  
 for (int i = 0; i < height \* width; ++i) {  
 uint8\_t g;  
 memcpy(&g, bitmap + i, 1);  
 ming = std::min(ming, g);  
 maxg = std::max(maxg, g);  
 }  
 #pragma omp parallel for shared(ming, maxg, bitmap) schedule(static) num\_threads(num\_threads)  
 for (int i = 0; i < height \* width; ++i) {  
 uint8\_t g;  
 memcpy(&g, bitmap + i, 1);  
 g = (g - ming) \* max\_val / (maxg - ming);  
 memcpy(bitmap + i, &g, 1);  
 }  
 int end = clock();  
 std::ofstream o(argv[3], std::ios::binary);  
 o.write("P5\n", 3);  
 o.write(std::to\_string(width).c\_str(), std::to\_string(width).size());  
 o.write(" ", 1);  
 o.write(std::to\_string(height).c\_str(), std::to\_string(height).size());  
 o.write("\n", 1);  
 o.write(std::to\_string(max\_val).c\_str(), std::to\_string(max\_val).size());  
 o.write("\n", 1);  
 o.write(bitmap, width \* height);  
 o.close();  
 std::cout << "Time (" << num\_threads << " thread(s)): " << (double)(end - beg) / **CLOCKS\_PER\_SEC** \* 1000 << " ms\n";  
 }  
 else {  
 int beg = clock();  
 uint8\_t minr = max\_val, ming = max\_val, minb = max\_val, maxr = 0, maxg = 0, maxb = 0;  
 #pragma omp parallel for reduction(min:minr,ming,minb) reduction(max:maxr,maxg,maxb) shared(bitmap) schedule(static) num\_threads(num\_threads)  
 for (int i = 0; i < height \* width; ++i) {  
 uint8\_t r, g, b;  
 memcpy(&r, bitmap + i \* 3, 1);  
 memcpy(&g, bitmap + i \* 3 + 1, 1);  
 memcpy(&b, bitmap + i \* 3 + 2, 1);  
 minr = std::min(minr, r);  
 maxr = std::max(maxr, r);  
 ming = std::min(ming, g);  
 maxg = std::max(maxg, g);  
 minb = std::min(minb, b);  
 maxb = std::max(maxb, b);  
 }  
 #pragma omp parallel for shared(minr, ming, minb, maxr, maxg, maxb, bitmap) schedule(static) num\_threads(num\_threads)  
 for (int i = 0; i < height \* width; ++i) {  
 uint8\_t r, g, b;  
 memcpy(&r, bitmap + i \* 3, 1);  
 memcpy(&g, bitmap + i \* 3 + 1, 1);  
 memcpy(&b, bitmap + i \* 3 + 2, 1);  
 r = (r - minr) \* max\_val / (maxr - minr);  
 g = (g - ming) \* max\_val / (maxg - ming);  
 b = (b - minb) \* max\_val / (maxb - minb);  
 memcpy(bitmap + i \* 3, &r, 1);  
 memcpy(bitmap + i \* 3 + 1, &g, 1);  
 memcpy(bitmap + i \* 3 + 2, &b, 1);  
 }  
 int end = clock();  
 std::ofstream o(argv[3], std::ios::binary);  
 o.write("P6\n", 3);  
 o.write(std::to\_string(width).c\_str(), std::to\_string(width).size());  
 o.write(" ", 1);  
 o.write(std::to\_string(height).c\_str(), std::to\_string(height).size());  
 o.write("\n", 1);  
 o.write(std::to\_string(max\_val).c\_str(), std::to\_string(max\_val).size());  
 o.write("\n", 1);  
 o.write(bitmap, width \* height \* 3);  
 o.close();  
 std::cout << "Time (" << num\_threads << " thread(s)): " << (double)(end - beg) / **CLOCKS\_PER\_SEC** \* 1000 << " ms\n";  
 }  
 return 0;  
}