НИЯУ МИФИ   
Лабораторная работа №1: Введение в параллельные вычисления. Технология OpenMP  
Касьяненко Данил Б19-505  
Москва, 2021

**Оглавление**

[**Описание используемой среды**](#_6u3k6sbho3ne) **3**

[**Анализ приведенного алгоритма**](#_ftpxs16a6j0c) **4**

[**Графики**](#_y8lqao4452bq) **5**

[**Заключение**](#_dv81q731s6tf) **6**

[**Приложение: код**](#_b15xmqq746jv) **7**

[**Приложение: таблицы**](#_pc0clxumrxcd) **8**

## 

## **Описание используемой среды** Процессор: Name Intel Core i7 8550U

## Codename Kaby Lake-R

## Number of cores 4 Number of threads 8 Оперативная память: 8 Гб, DDR4 Операционная система: Microsoft Windows 10 Professional (x64) Используемая среда разработки: Visual studio 2019, компилятор MSVC

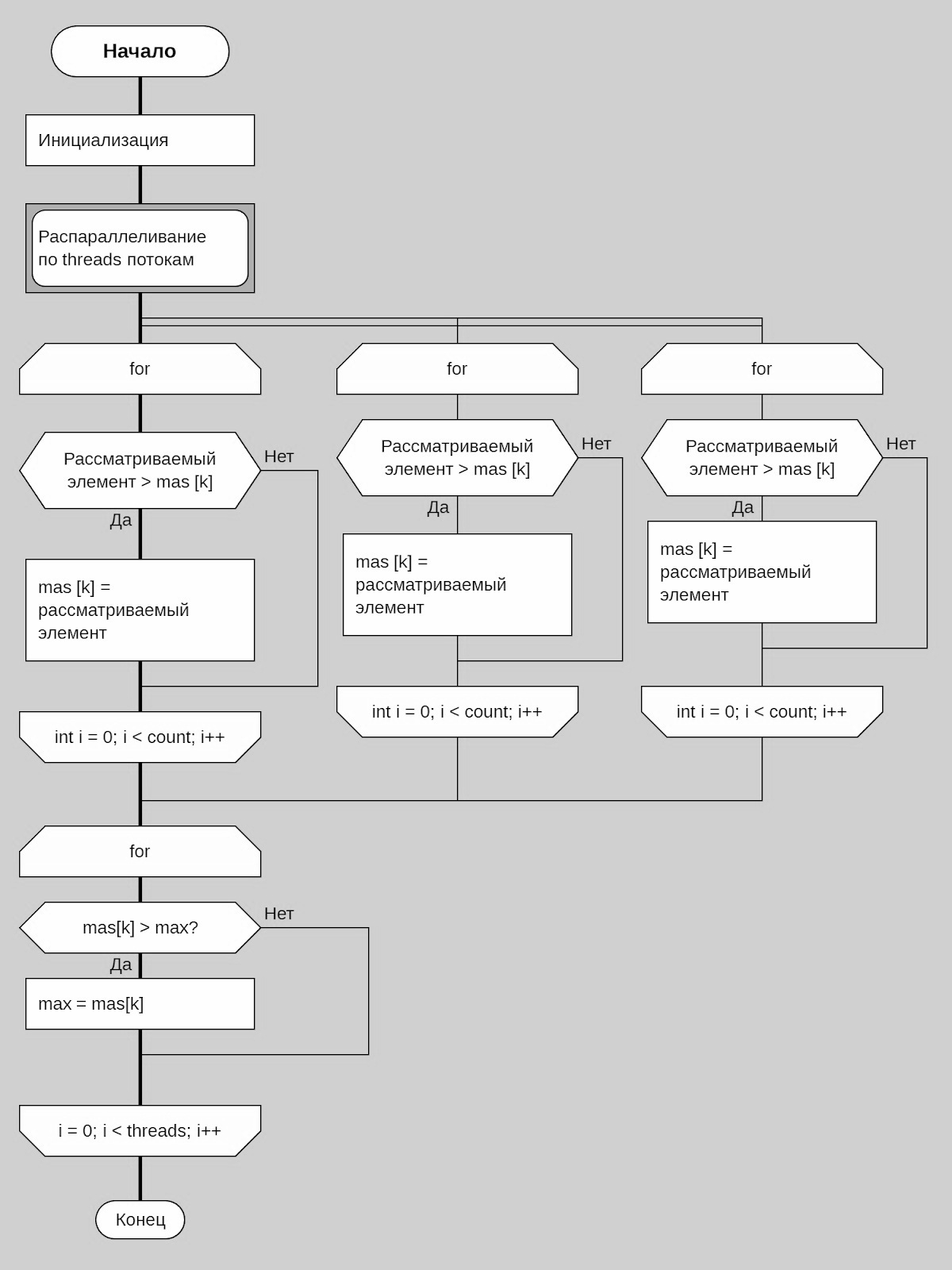
## Версия OpenMP: OpenMP 2,0 (200203), OpenMP LLVM

## 

## 

## **Анализ приведенного алгоритма** Временная сложность последовательного алгоритма T1 = O(N), где N - размер массива. Временная сложность параллельного алгоритма Tp = O(N/p) , где N - размер массива, а p - количество потоков. Формула для ускорения, S = T1/Tp, даёт нам при таком исходе следующую теоретическую оценку: S = T1/(Tp) ~ O(N)/O(N/p) = O(p). Параллельный алгоритм может давать большое ускорение, но использование для этого множества процессов является неэффективной тратой. Оценим эффективность: E = S/p. В нашем случае E ~ O(p)/p. Т.о. эффективность равна O(1). **Описание принципа работы, директивы** Принцип работы данного параллельного алгоритма заключается в том, что существует цикл for, проходящий по всем элементам массива. При этом данный цикл распараллеливается на несколько независимых частей. Делается это с помощью директивы openmp ‘for’, которая делает работу цикла for внутри параллельной области распределённой между потоками, т.е. каждая итерация выполняется в своём потоке, при этом потоков может быть меньше, чем итераций – тогда какой-то итерации придётся подождать. Если бы данной директивы не было, то каждый поток выполнял бы весь for N-ное количество раз (сколько потоков, столько и раз). Директива распространяется только на самый внешний цикл, но у нас он один. Каждая итерация цикла for занимается сравнением i-того элемента массива с текущим максимальным элементом, и если он больше, то максимальным элементом становится i-тый элемент. Для каждого куска цикла for максимальный элемент локален, это можно решить с помощью редукции, но редукция максимумов не работает на версии OpenMP 2.0. Директива ‘for’ обернута в директиву ‘parallel’ (распространяющуюся на блок {}, следующий за ней), которая и занимается распараллеливанием на потоки,параметр num\_threads указывает количество потоков. Параметр shared указывает на общие переменные, default(none) обозначает то, что класс переменных нужно задавать явно, что мы и сделали с помощью параметра shared. Отсутствие директивы ‘parallel’ делает параллельный алгоритм бессмысленным. Параметр reduction(max : max), которого нету, имеет два аргумента: операцию и переменную (точнее, локальные переменные с одинаковым именем). Операция у нас max, т.е. над всеми переменными будет выполняться операция извлечения максимального значения. Редукция максимумом в итоге да максимальный элемент из всего массива. Но так как у нас его нету, у нас просто набор из локальных максимумов, не учитывающий весь массив.

**Блок-схема алгоритма**



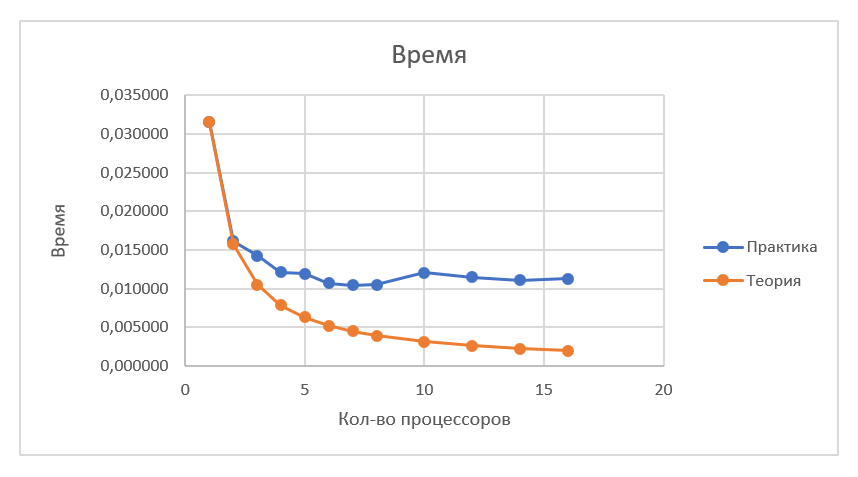
## **Графики**

Был проведен анализ алгоритма (Приложение: код) на 10 различных массивах для различного числа потоков. Количество элементов для каждого эксперимента было постоянным и равнялось n = 10 000 000. Далее данные усредняются.

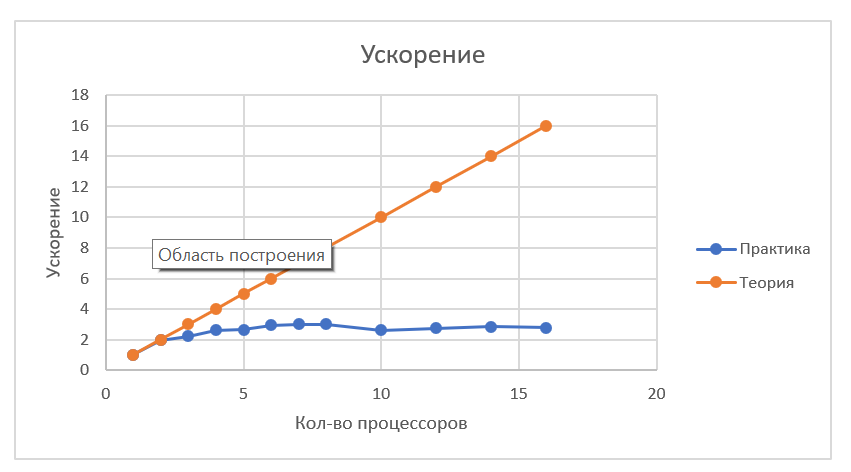
Рассмотрим данный эксперимент в теории.

Сложность: при последовательном алгоритме работает только 1 процессор, он обрабатывает n элементов, соответственно его сложность равняется O(n). При распараллеливании этого алгоритма на р потоков, каждый поток уже обрабатывает n/p элементов, а значит сложность будет О(n/p).

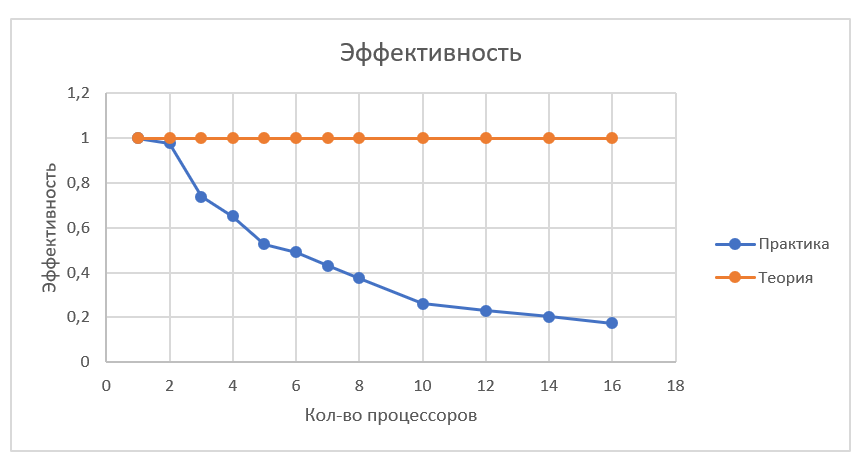
Результат практического эксперимента представлен на графике №1:

График №1

*Ускорение*: Ускорение S считается как отношение времени последовательного алгоритма к времени параллельного. Соответственно в нашем случае ускорение будет равно числу процессоров. Графики ожидаемого и реального ускорения представлены ниже (график №2).

График №2

Эффективность: Эффективность считается как ускорение, деленное на количество процессоров, соответственно в нашем случае эффективность является константой и равно 1.

График №3

Как видно из графика практическое время несколько больше, чем ожидаемое. Это можно объяснить тем, что нам необходимо время на распараллеливание и выдачу задания каждому потоку. Так же присутствует погрешность, значения при различных сидах получились очень разными. Что касается конкретно представленного в приложении №1 алгоритма, то там так же дополнительно время требуется на обработку локальных максимумов, то количество этих максимумов будет равно количеству потоков, которое в свою очередь много меньше числа элементов, поэтому в асимптотической оценке алгоритма мы можем это не учитывать. Что касается ускорения и эффективности, они также меньше теоретических, т.к на прямую зависят от времени параллельного алгоритма.

Анализируя таблицу (Приложение: таблицы) и графики можно сказать, что оптимальным по ускорению являются алгоритмы на 7-8 процессорах. Это объяснимо, ведь ноутбук, на котором проводились опыты, имеет 8 логических процессоров. А дальнейшее распараллеливание на большее число потоков не имеет смысла, т.к время на распараллеливание не окупается.

## **Заключение**

Получены практические результаты, которые показывают, что практические результаты отличаются от того, что должно было получиться в теории при увеличении числа потоков, следовательно необходимы дополнительные исследования для оценки работы алгоритма с большим количеством потоков, так как расхождения больше, чем больше количество параллельных потоков вычисления

## Приложение: код

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#include <malloc.h>

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

{

const int count = 10000000; ///< Number of array elements

const int threads = 2; ///< Number of parallel threads to use

const int random\_seed = 92215; ///< RNG seed

int\* array = 0; ///< The array we need to find the max in

int m = -1; ///< The maximal element

int\* mas = 0;

// time\_t tm = time(NULL);

// srand(tm);

/\* Initialize the RNG \*/

srand(random\_seed);

/\* Determine the OpenMP support \*/

printf("OpenMP: %d;\n======\n", \_OPENMP);

/\* Generate the random array \*/

array = (int\*)malloc(count \* sizeof(int));

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

{

array[i] = rand();

// printf("-- My %d is: %d;\n", i, array[i]);

}

/\* Find The maximal one thread\*/

double start = omp\_get\_wtime();

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

if (array[i] > m) { m = array[i]; }

}

double end = omp\_get\_wtime();

printf("Max for linear algorithm is: %d\n", m);

printf("Lineal time: %f\n", end - start);

// Find the maximal element parallel

int i;

int k = omp\_get\_num\_procs();

printf("You have %d procs;\n", k);

mas = (int\*)malloc(threads \* sizeof(int));

for (int i = 0; i < threads; i++) { mas[i] = -1; }

start = omp\_get\_wtime();

#pragma omp parallel num\_threads(threads) shared(array, count)

{

k = omp\_get\_thread\_num();

#pragma omp for

for (i = 0; i < count; i++)

{

if (array[i] > mas[k]) { mas[k] = array[i]; };

}

}

for (i = 0; i < threads; i++)

{

if (mas[i] > m) { m = mas[i]; };

}

end = omp\_get\_wtime();

printf("======\nMax is: %d;\n", m);

printf("Parallel time: %f\n", end - start);

return(0);

}

## Приложение: таблицы

