

Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–  
Физический Институт)  
Кафедра №42 «Криптология и кибербезопасность»

Лабораторная работа №4:  
«Технология OpenMP. Особенности настройки»

## Описание архитектуры

Architecture: x86\_64  
CPU op-mode(s): 32-bit, 64-bit  
Address sizes: 48 bits physical, 48 bits virtual  
Byte Order: Little Endian  
CPU(s): 12  
On-line CPU(s) list: 0-11  
Vendor ID: AuthenticAMD  
Model name: AMD Ryzen 5 5500U with Radeon Graphics  
CPU family: 23  
Model: 104  
Thread(s) per core: 2  
Core(s) per socket: 6  
Socket(s): 1  
Stepping: 1  
CPU(s) scaling MHz: 52%  
CPU max MHz: 4056.0000  
CPU min MHz: 400.0000  
BogoMIPS: 4192.31

Transient hostname: DESKTOP-J2NEN3H  
Icon name: computer-laptop  
Chassis: laptop ☐  
Machine ID: caf94732efe24b519ce9ca85095c24f4  
Boot ID: 64bd1e3a27ad44128e6b00b59b2c533f  
Operating System: Fedora Linux 38 (Workstation Edition)  
CPE OS Name: cpe:/o:fedoraproject:fedora:38  
OS Support End: Tue 2024-05-14  
OS Support Remaining: 6month 1w 5d  
Kernel: Linux 6.5.6-200.fc38.x86\_64  
Architecture: x86-64  
Hardware Vendor: HUAWEI  
Hardware Model: NBM-WXX9  
Firmware Version: 2.09  
Firmware Date: Wed 2022-03-23

	total	used	free	shared	buff/cache	available
Mem:	7428976	4246056	447036	117204	2735884	2759516
Swap:	7428092	858880	6569212			

# Среда разработки

Компилятор: gcc (GCC) 13.2.1 20230728 (Red Hat 13.2.1-1)

Средство сборки: Makefile

Текстовый редактор: Visual Studio Code

Версия OPENMP: 201511

## Задание 1

При помощи переменной предпроцессора OPENMP определить дату принятия используемого стандарта OpenMP. Вывести на экран версию стандарта и дату принятия;



```
1  #ifdef _OPENMP
2      printf("OPENMP version: %d\n", _OPENMP);
3      printf("Adopted in: %d-%d\n", _OPENMP / 100, _OPENMP % 100);
4  #else
5      printf("OpenMP is not supported.\n");
6  #endif
7
```

Вывод:

OPENMP version: 201511

Adopted in: 2015-11

## Задание 2

Использовать функции `omp_get_num_procs()` и `omp_get_max_threads()` для определения числа доступных процессоров и потоков. Вывести результат;

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of C code:

```
1 printf("Number of available processors: %d\n", omp_get_num_procs());
2 printf("Max number of threads: %d\n", omp_get_max_threads());
```

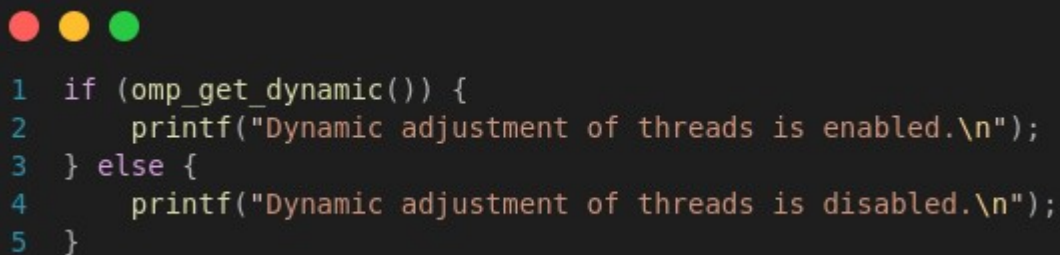
Вывод:

Number of available processors: 12

Max number of threads: 12

## Задание 3

Выяснить и описать назначение опции `dynamic`. Определить её состояние при помощи функции `omp_get_dynamic()`. Вывести результат;

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains five lines of C code:

```
1 if (omp_get_dynamic()) {
2     printf("Dynamic adjustment of threads is enabled.\n");
3 } else {
4     printf("Dynamic adjustment of threads is disabled.\n");
5 }
```

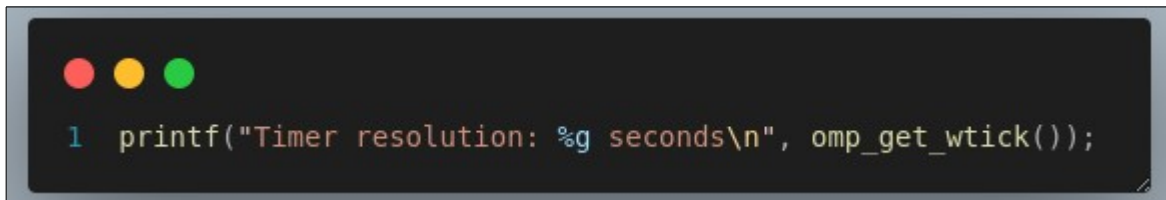
Вывод:

Dynamic adjustment of threads is disabled.

Расписание dynamic характеризуется свойством, в котором поток не ожидает в барьере дольше, чем требуется другому потоку для выполнения окончательной итерации. Это требование означает, что при каждом назначении необходимо назначать итерации по одному потоку по мере их доступности с синхронизацией для каждого назначения. Подходит для циклов for, требующих различных или даже непредсказуемых объемов работы.

## Задание 4

Определить разрешение таймера при помощи `omp_get_wtick()`. Вывести результат с указанием единицы измерения;

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays a single line of code: `1 printf("Timer resolution: %g seconds\n", omp_get_wtick());`.

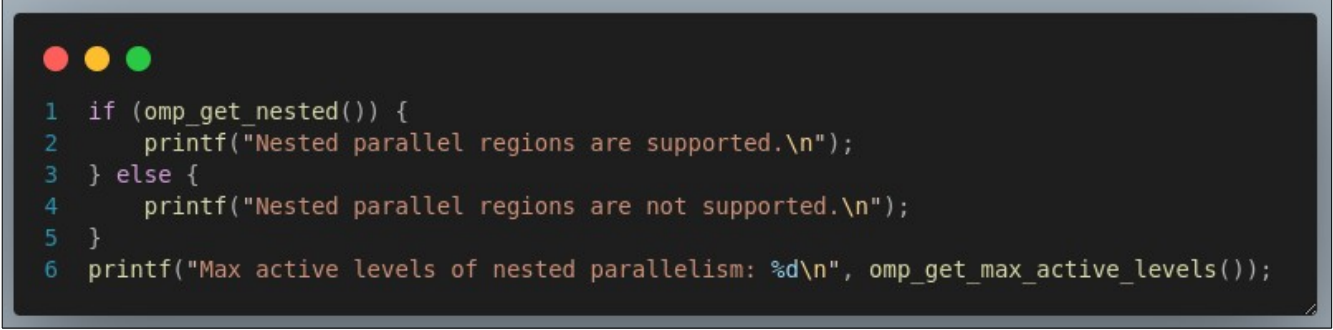
```
1 printf("Timer resolution: %g seconds\n", omp_get_wtick());
```

Вывод:

Timer resolution: 1e-09 seconds

## Задание 5

Уточнить особенности работы со вложенными параллельными областями в OpenMP. Определить текущие настройки среды при помощи функций `omp_get_nested()` и `omp_get_max_active_levels()` и вывести на экран;

A screenshot of a terminal window with a dark background and light-colored text. At the top left, there are three colored circles (red, yellow, green) representing window control buttons. The code is as follows:

```
1 if (omp_get_nested()) {  
2     printf("Nested parallel regions are supported.\n");  
3 } else {  
4     printf("Nested parallel regions are not supported.\n");  
5 }  
6 printf("Max active levels of nested parallelism: %d\n", omp_get_max_active_levels());
```

Вывод:

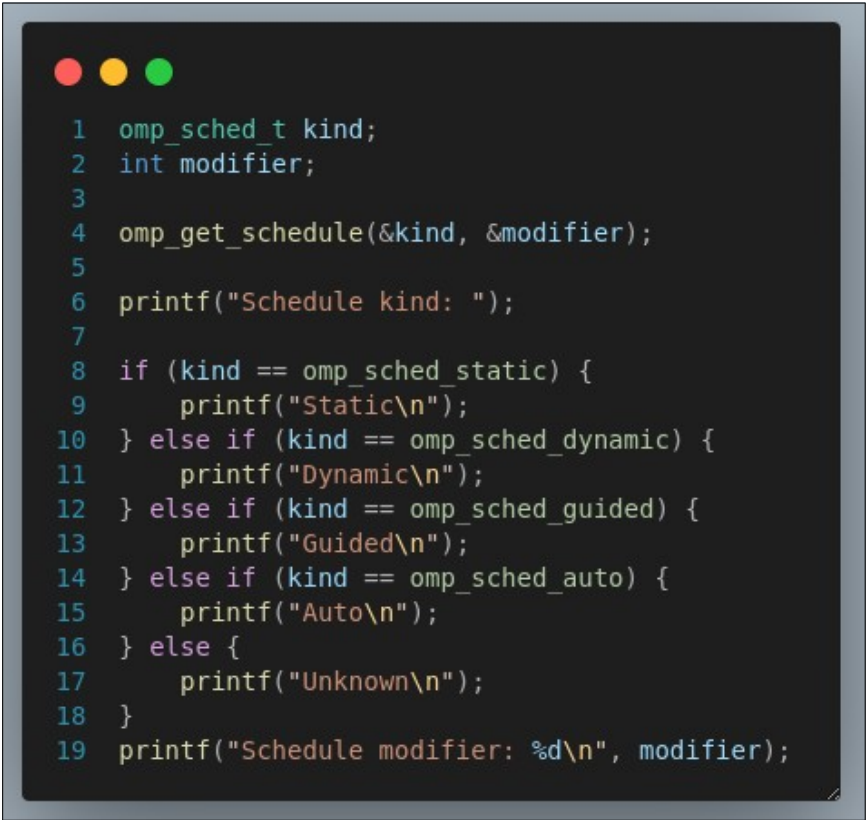
Nested parallel regions are not supported.

Max active levels of nested parallelism: 1

OpenMP поддерживает вложенные параллельные области, позволяя создавать параллельные конструкции внутри других параллельных участков кода. Это означает, что вы можете использовать директивы OpenMP для создания вложенных областей, где каждая внутренняя область может создавать свои собственные потоки.

## Задание 6

Уточнить особенности распределения нагрузки в среде OpenMP. Получить текущие настройки среды с использованием функции `omp_get_schedule()` и вывести их на экран;

A screenshot of a code editor window with a dark background and light-colored text. The code is in C and uses OpenMP. It declares two variables, `omp_sched_t kind;` and `int modifier;`, then calls `omp_get_schedule(&kind, &modifier);` to retrieve the current OpenMP schedule settings. It then uses `printf` to output the schedule kind and modifier. The schedule kind is checked against `omp_sched_static`, `omp_sched_dynamic`, `omp_sched_guided`, and `omp_sched_auto`. The output shown in the next block is "Schedule kind: Dynamic" and "Schedule modifier: 1".

```
1  omp_sched_t kind;  
2  int modifier;  
3  
4  omp_get_schedule(&kind, &modifier);  
5  
6  printf("Schedule kind: ");  
7  
8  if (kind == omp_sched_static) {  
9      printf("Static\n");  
10 } else if (kind == omp_sched_dynamic) {  
11     printf("Dynamic\n");  
12 } else if (kind == omp_sched_guided) {  
13     printf("Guided\n");  
14 } else if (kind == omp_sched_auto) {  
15     printf("Auto\n");  
16 } else {  
17     printf("Unknown\n");  
18 }  
19 printf("Schedule modifier: %d\n", modifier);
```

Вывод:

Schedule kind: Dynamic

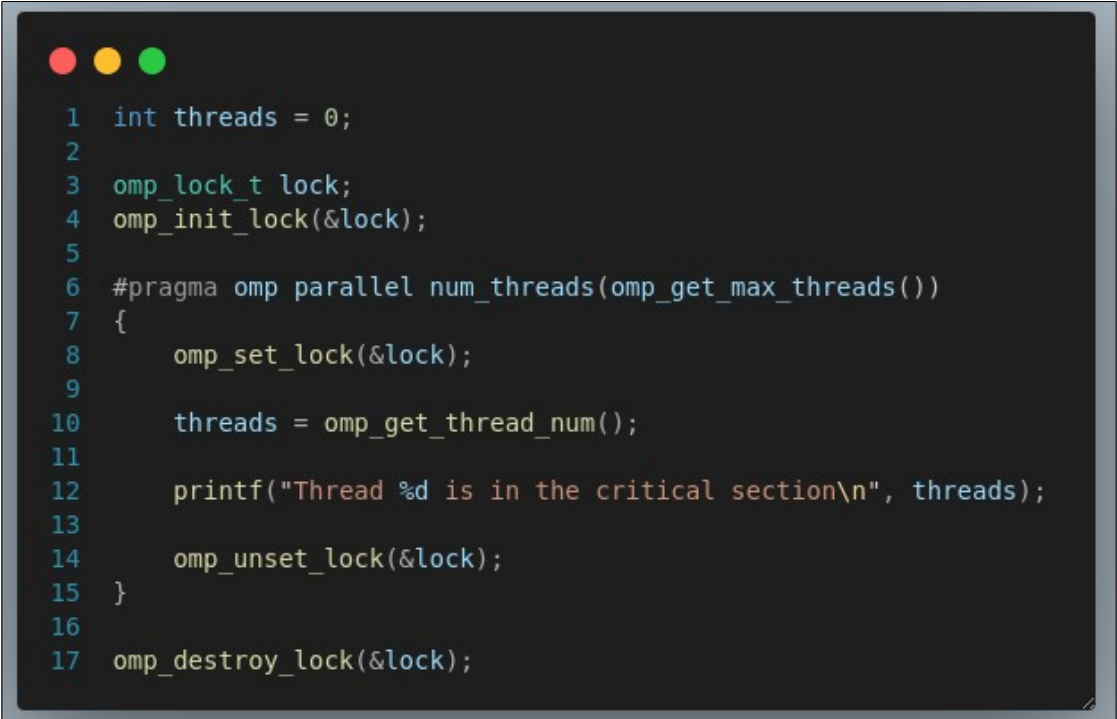
Schedule modifier: 1

Основные виды планировщиков в OPENMP включают:

- 1) static: итерации цикла разделяются заранее между потоками.
- 2) dynamic: итерации цикла назначаются потокам по мере их запроса.
- 3) guided: итерации цикла распределяются динамически, но блоки итераций уменьшаются по мере выполнения

## Задание 7

Разработать пример вычислительного алгоритма, использующего механизм явных блокировок (`omp set lock()`). Обосновать необходимость использования блокировки;



```
1  int threads = 0;
2
3  omp_lock_t lock;
4  omp_init_lock(&lock);
5
6  #pragma omp parallel num_threads(omp_get_max_threads())
7  {
8      omp_set_lock(&lock);
9
10     threads = omp_get_thread_num();
11
12     printf("Thread %d is in the critical section\n", threads);
13
14     omp_unset_lock(&lock);
15 }
16
17 omp_destroy_lock(&lock);
```

В этом примере используется блокировка для защиты переменной “x” и операции вывода, которые являются общими ресурсами. Блокировка гарантирует, что только один поток может войти в критическую секцию в любой момент времени. Критическая секция - это участок исполняемого кода программы, в котором производится доступ к общему ресурсу, который не должен быть одновременно использован более чем одним потоком выполнения.



## Задание 8

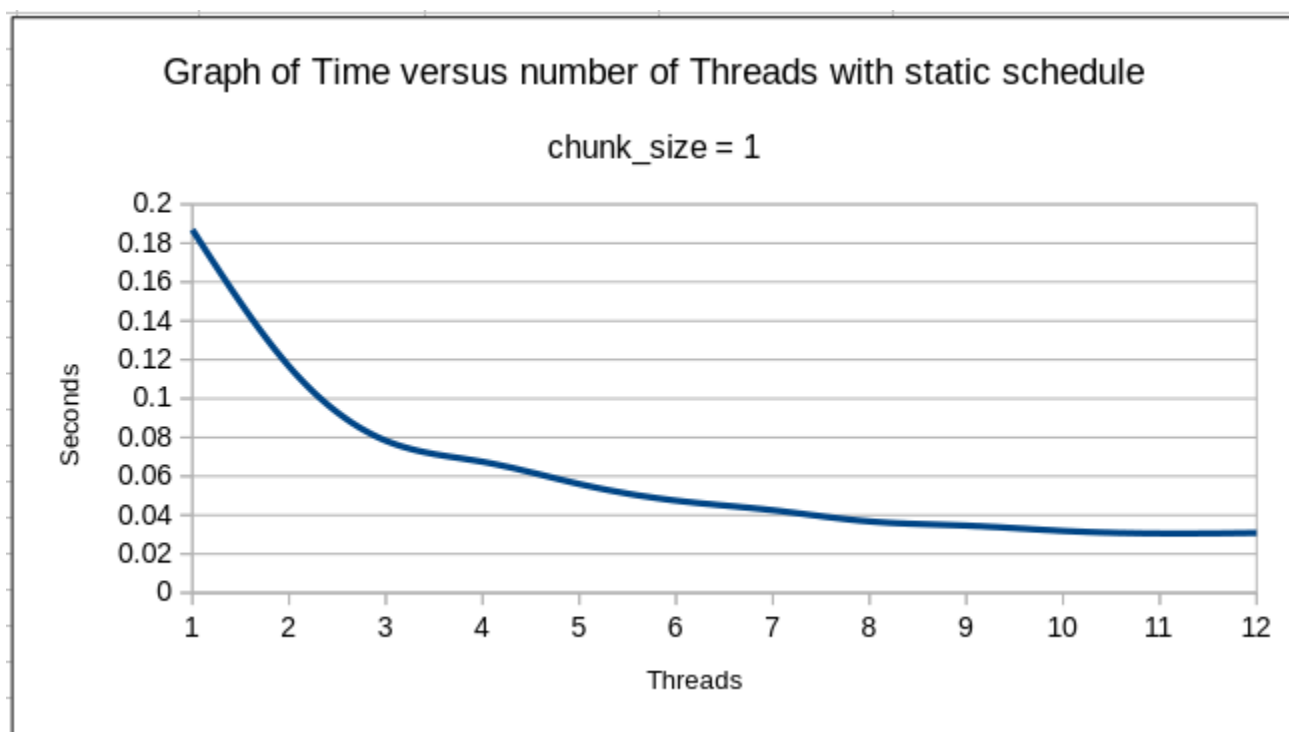
Для одного из алгоритмов, реализованных в предыдущих лабораторных работах, повторить вычислительный эксперимент для разных типов разделения нагрузки и размеров фрагмента (опция `schedule` директивы `parallel`). Сравнить результаты; объяснить наличие/отсутствие разницы;

### Код программы

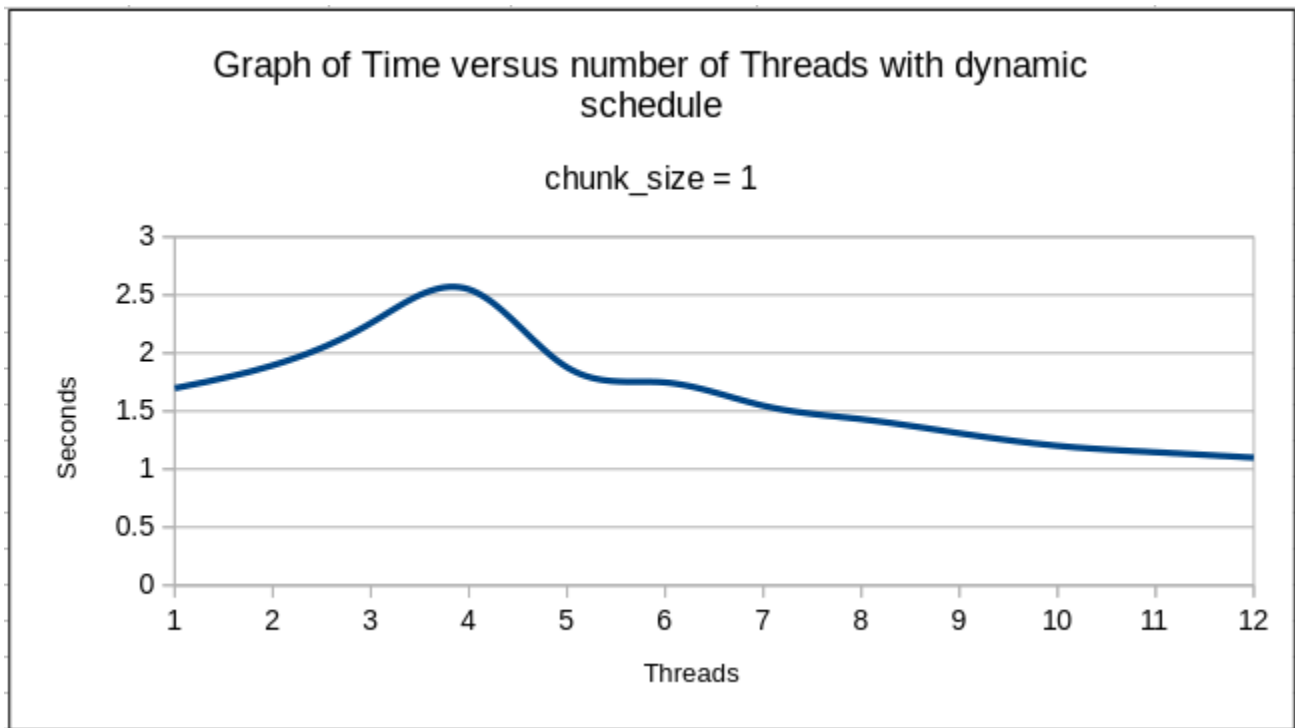
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5
6  int main() {
7      const int count = 100000000;
8      const int random_seed = 920215;
9      const int max_threads = 12;
10     int max = -1;
11
12     srand(random_seed);
13
14     int *array = (int *)malloc(count * sizeof(int));
15     for (int i = 0; i < count; i++) {
16         array[i] = rand();
17     }
18
19     for (int threads = 1; threads <= max_threads; threads++) {
20
21         double start_time = omp_get_wtime();
22
23         #pragma omp parallel num_threads(threads) shared(array, count) reduction(max:max) default(none)
24         {
25             #pragma omp for schedule(static, 1)
26             for (int i = 0; i < count; i++) {
27                 if (array[i] > max) {
28                     max = array[i];
29                 };
30             }
31         }
32
33         printf("Threads: %d, Execution time: %f seconds\n", threads, omp_get_wtime() - start_time);
34     }
35
36     free(array);
37
38     return 0;
39 }
```

В качестве примера мною был выбран алгоритм, реализованный в первой лабораторной работе.

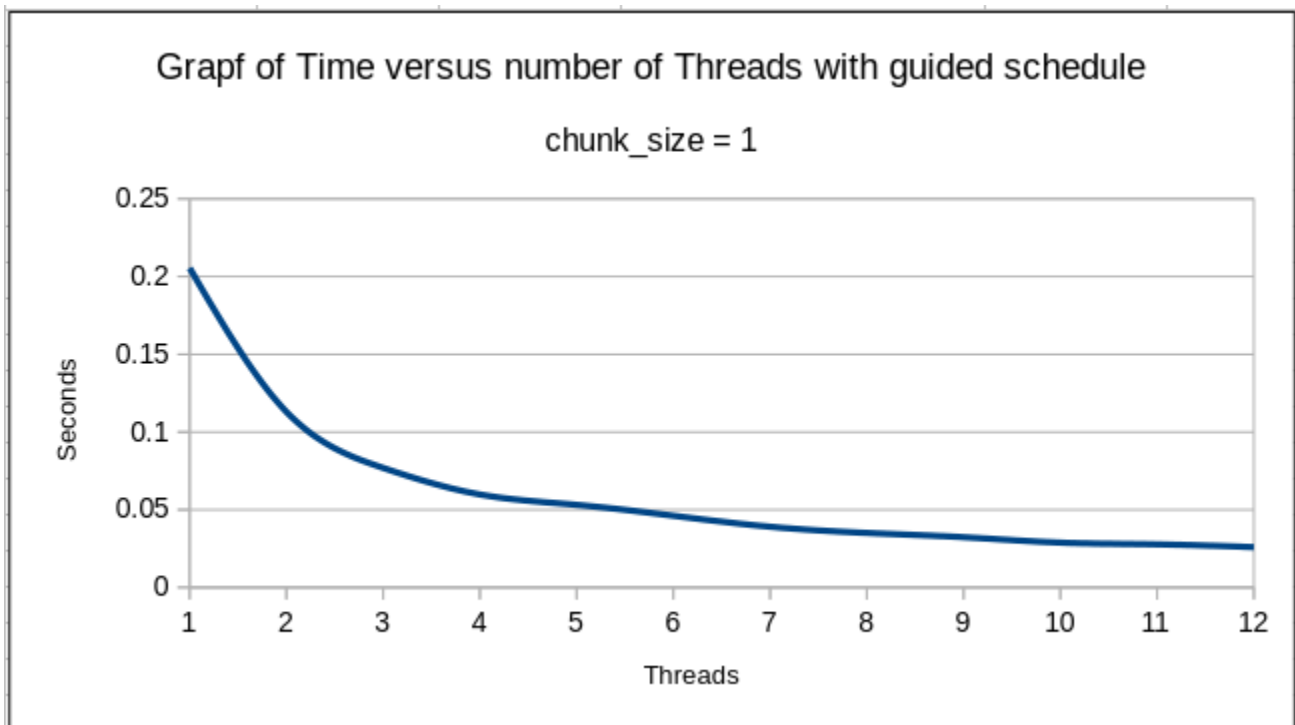
## График №1



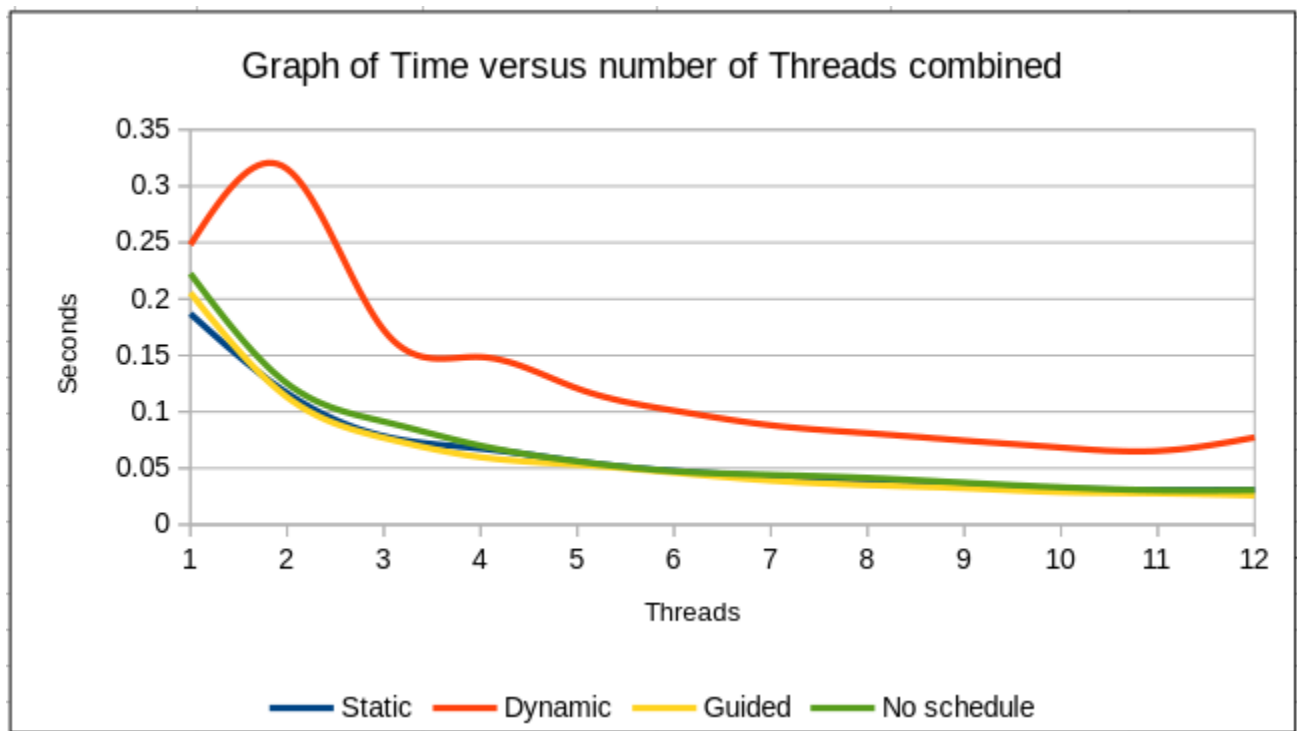
**График №2**



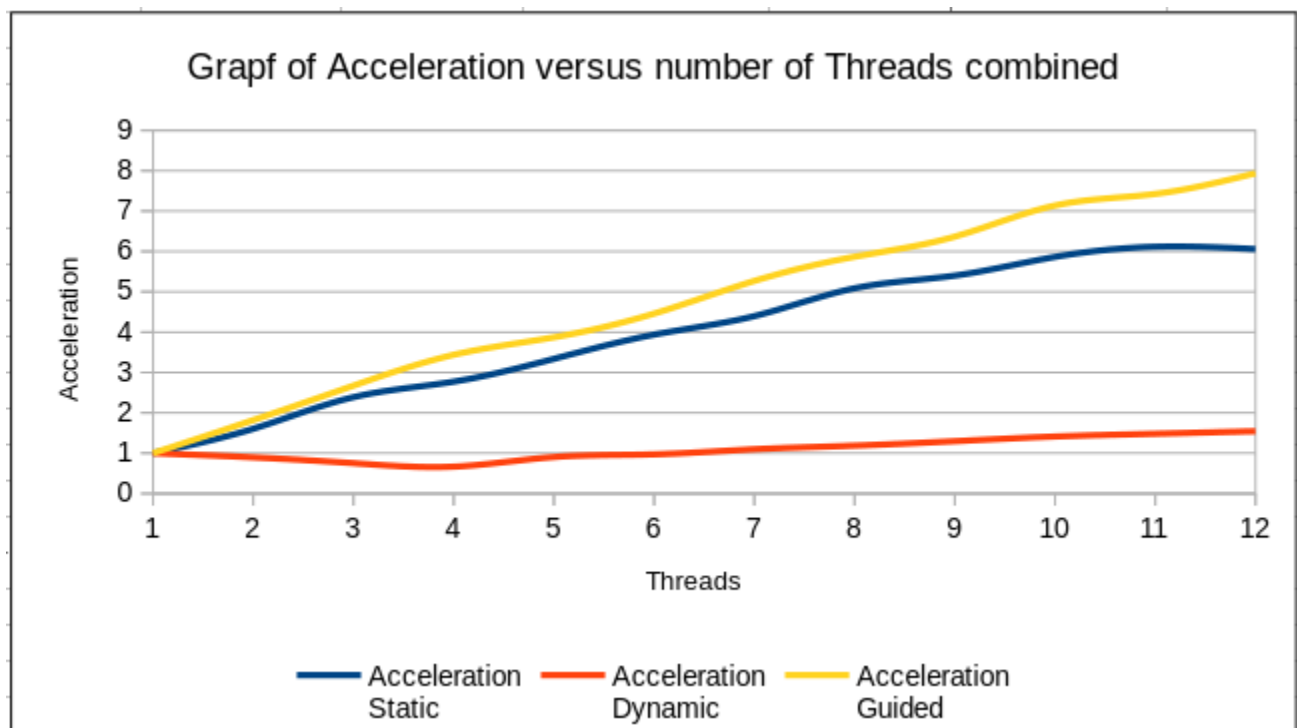
**График №3**



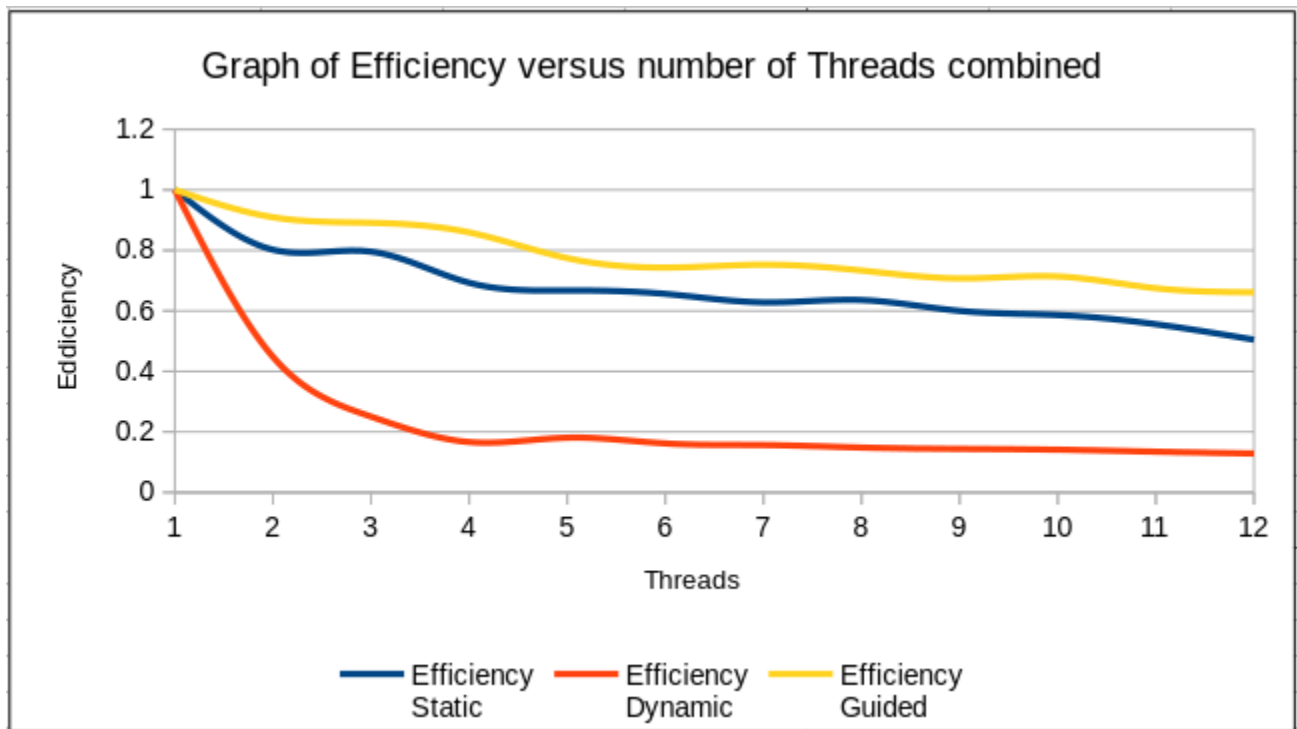
### График №4



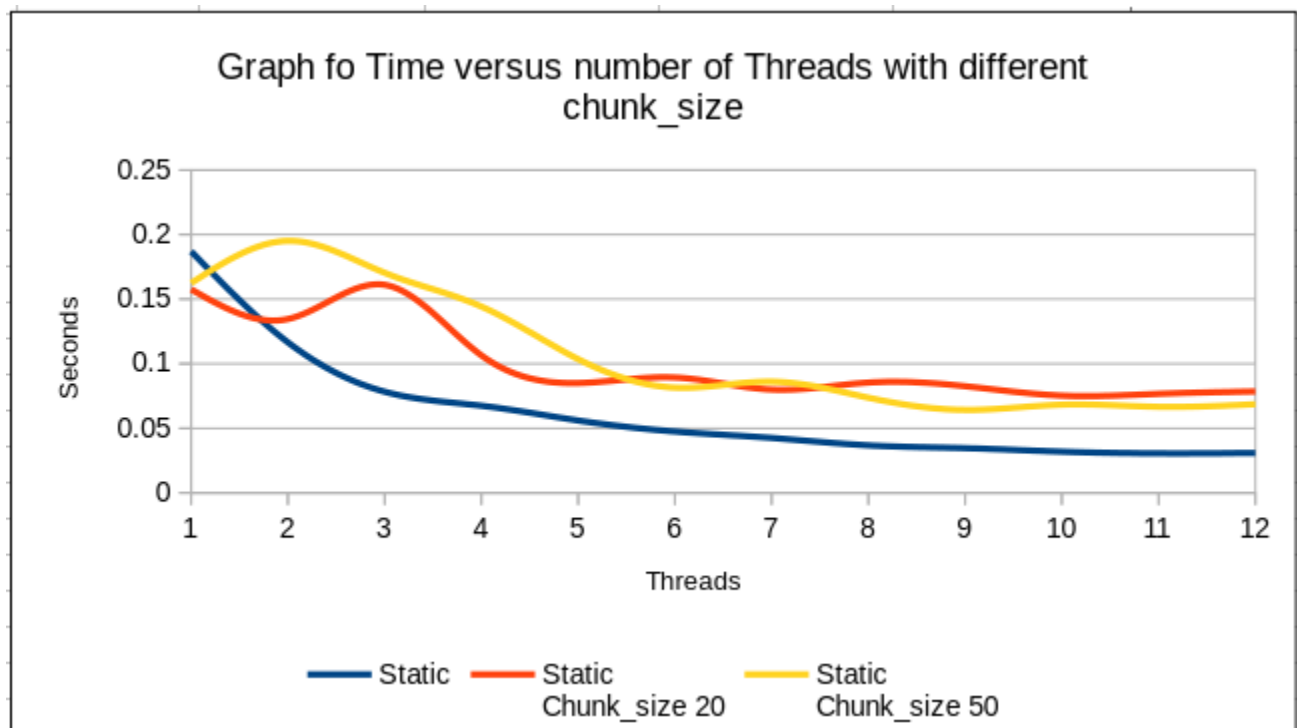
### График №5



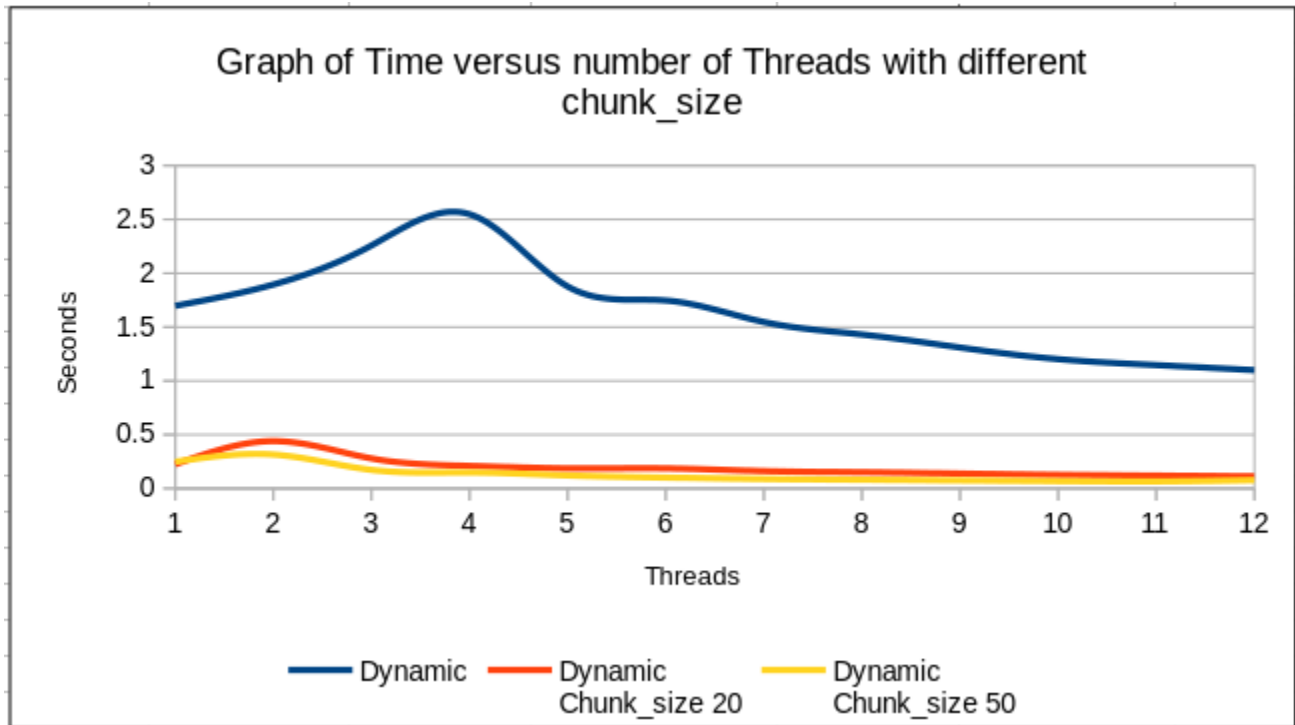
### График №6



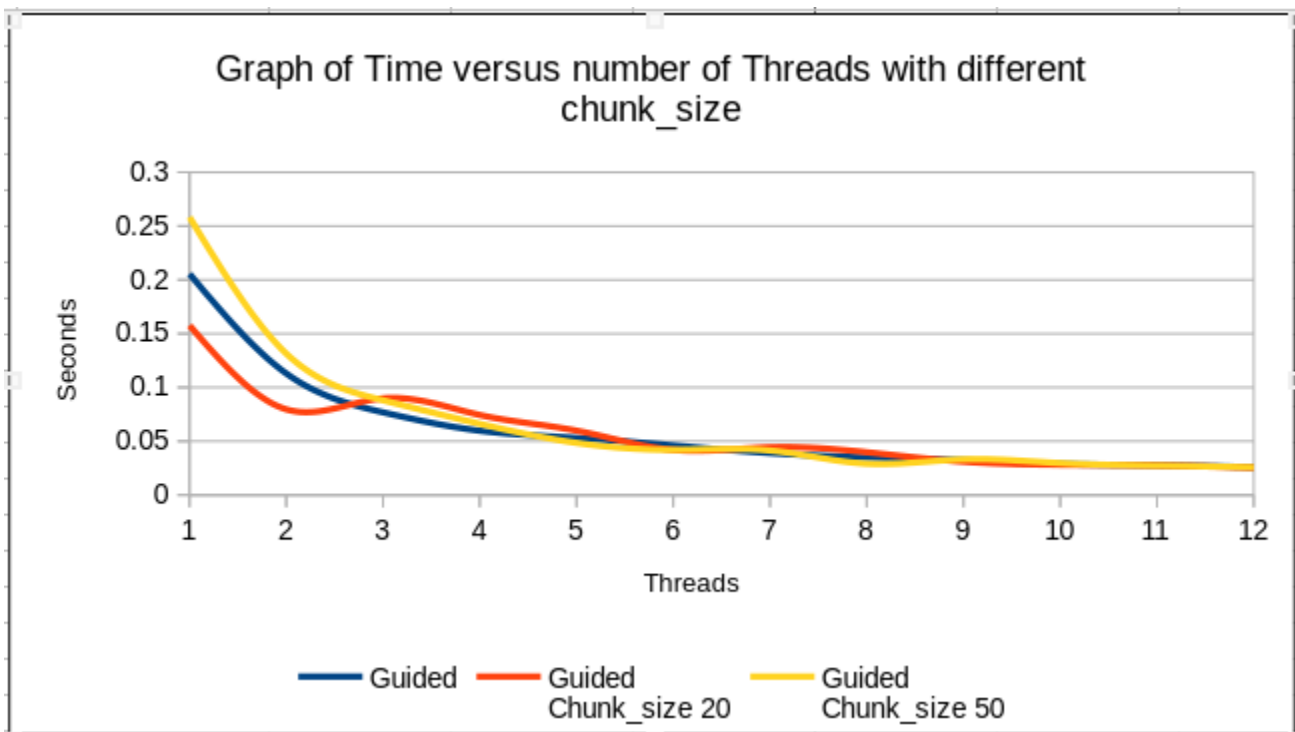
### График №7



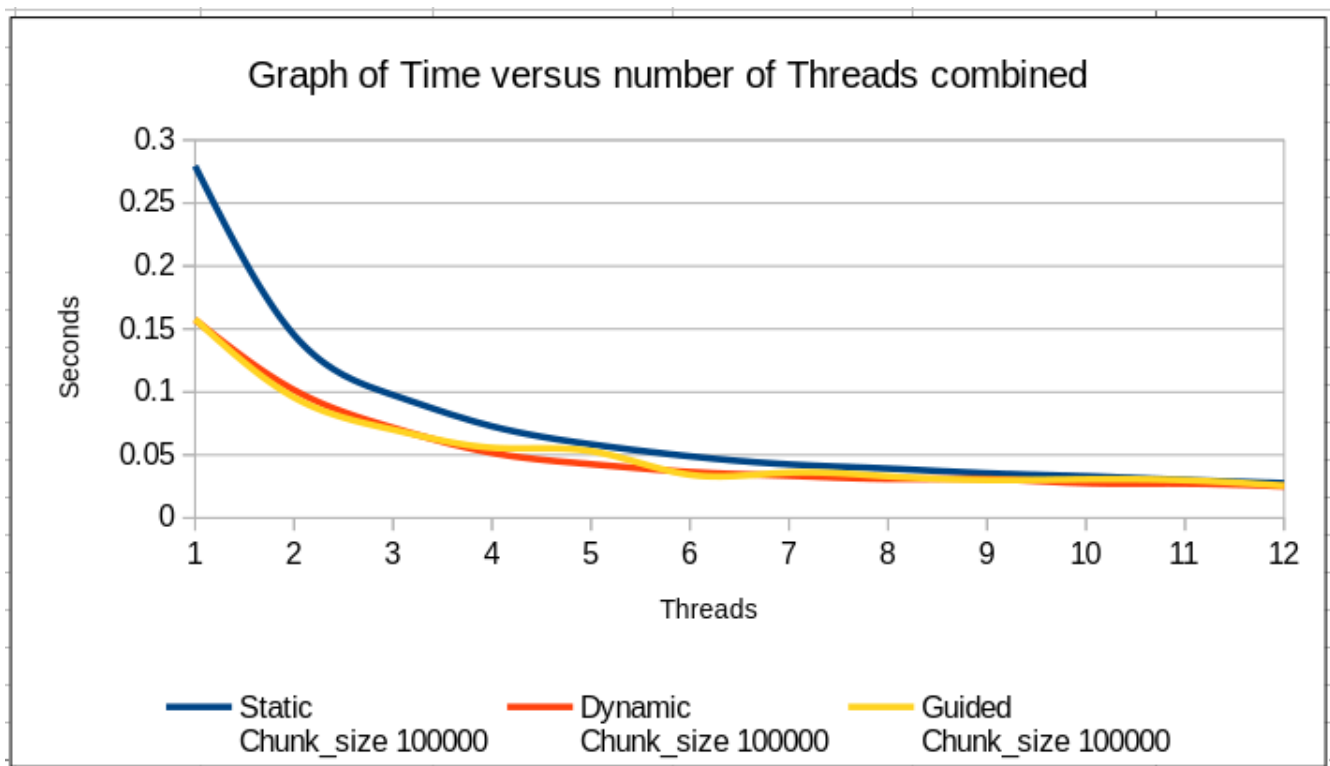
**График №8**



**График №9**



**График №10**



# Таблица с данными

Threads	Static #1	Static #2	Static #3	Static #4	Static #5	Static	Acceleration Static	Efficiency Static	Static Chunk_size 20	Static Chunk_size 50	Static Chunk_size 100000	Static Chunk_size count/12
1	0.161072	0.245908	0.158367	0.158449	0.210833	0.1869258	1	1	0.157591	0.16237	0.279562	0.250921
2	0.130825	0.142803	0.083831	0.080231	0.144949	0.1165278	1.604130516495	0.80206526	0.134566	0.195114	0.145295	0.131399
3	0.087928	0.095451	0.056762	0.054478	0.096891	0.078302	2.387241705193	0.79574724	0.161077	0.170374	0.097846	0.088426
4	0.066058	0.072069	0.065842	0.06068	0.072541	0.067438	2.771817076426	0.69295427	0.106448	0.144255	0.072799	0.066842
5	0.052981	0.057916	0.052689	0.058345	0.058151	0.0560164	3.336983454845	0.66739669	0.085012	0.103537	0.058494	0.066838
6	0.04434	0.048162	0.047364	0.049027	0.048489	0.0474764	3.937236184715	0.65820603	0.089357	0.081451	0.048946	0.044724
7	0.041125	0.042264	0.040883	0.045849	0.042458	0.0425158	4.396619609651	0.62808852	0.079954	0.086095	0.042565	0.050603
8	0.036277	0.037313	0.035816	0.03715	0.03721	0.0367532	5.085973466256	0.63574668	0.085387	0.073671	0.039238	0.052197
9	0.033604	0.037989	0.032926	0.035394	0.033171	0.0346168	5.399857872478	0.59998421	0.082576	0.064212	0.035545	0.050321
10	0.033235	0.033593	0.02987	0.032819	0.02988	0.0318794	5.863529426526	0.58635294	0.075284	0.068282	0.033137	0.052475
11	0.030412	0.036811	0.028586	0.029841	0.02719	0.030568	6.115081130594	0.55591647	0.076677	0.06662	0.03023	0.052178
12	0.027965	0.044007	0.029741	0.027478	0.025076	0.0308534	6.058515431038	0.50487629	0.07847	0.068615	0.027678	0.026968
	Dynamic #1	Dynamic #2	Dynamic #3	Dynamic #4	Dynamic #5	Dynamic	Acceleration Dynamic	Efficiency Dynamic	Dynamic Chunk_size 20	Dynamic Chunk_size 50	Dynamic Chunk_size 100000	Dynamic Chunk_size count/12
	1.747119	1.746966	1.746814	1.498182	1.747009	1.697218	1	1	0.22429	0.248007	0.157376	0.269348
	2.093476	1.974936	1.971538	1.489519	1.944275	1.8947688	0.895738836316	0.44786942	0.440053	0.31523	0.101591	0.144012
	2.353284	2.365151	2.368329	1.817723	2.383489	2.2575952	0.751781364525	0.25059379	0.279011	0.172314	0.07162	0.09675
	2.281663	2.624247	2.785777	2.129926	2.920993	2.5485212	0.66596189194	0.16649047	0.209504	0.148544	0.051663	0.066775
	1.821939	1.819002	1.867985	1.854771	2.021164	1.8769722	0.904231826129	0.18084637	0.186845	0.12064	0.042715	0.048453
	1.741135	1.748187	1.748984	1.747067	1.74903	1.7468806	0.971570695788	0.16192845	0.186574	0.101028	0.036532	0.048553
	1.551288	1.540573	1.555849	1.537403	1.538741	1.5467708	1.097265347911	0.15675219	0.161427	0.088035	0.033487	0.043967
	1.438477	1.433119	1.431853	1.427113	1.421813	1.430475	1.186471626558	0.14830895	0.150651	0.081024	0.030995	0.043983
	1.318818	1.313528	1.303574	1.30868	1.302755	1.309471	1.296109650386	0.14401218	0.138592	0.074461	0.030984	0.045959
	1.204354	1.196643	1.201055	1.205906	1.199498	1.2014912	1.412592951159	0.1412593	0.126757	0.068295	0.027586	0.044476
	1.153161	1.148437	1.138058	1.140949	1.149974	1.1461158	1.480843384237	0.13462213	0.119413	0.06529	0.0272	0.0443
	1.102286	1.095484	1.085235	1.089287	1.124133	1.099285	1.543929008401	0.12866075	0.112768	0.077169	0.025106	0.02496
	Guided #1	Guided #2	Guided #3	Guided #4	Guided #5	Guided	Acceleration Guided	Efficiency Guided	Guided Chunk_size 20	Guided Chunk_size 50	Guided Chunk_size 100000	Guided Chunk_size count/12
	0.157723	0.2733	0.158455	0.158522	0.279395	0.205479	1	1	0.157601	0.258242	0.157813	0.240478
	0.093809	0.145055	0.102005	0.079108	0.144881	0.1129716	1.818855358338	0.90942768	0.079626	0.131457	0.09571	0.103084
	0.071875	0.097324	0.063193	0.061128	0.09081	0.076866	2.673210522207	0.89107017	0.089925	0.088059	0.070094	0.06513
	0.050077	0.072932	0.053477	0.054923	0.067477	0.0597772	3.437414264971	0.85935357	0.074427	0.06618	0.055767	0.065163
	0.053174	0.058557	0.048199	0.052834	0.052645	0.0530818	3.870987796194	0.77419756	0.059861	0.048462	0.053058	0.052212
	0.042756	0.048886	0.050741	0.044227	0.043801	0.0460822	4.4589668028	0.74316113	0.04193	0.041978	0.034276	0.048714
	0.03973	0.033452	0.032897	0.04478	0.04416	0.0390038	5.268178997944	0.752597	0.044425	0.041405	0.036006	0.044074
	0.036787	0.033024	0.036752	0.031705	0.036851	0.0350238	5.866839120826	0.73335489	0.039782	0.02915	0.033072	0.042733
	0.032193	0.028596	0.032782	0.03499	0.032826	0.0322774	6.3660331997	0.70733702	0.03096	0.03319	0.030133	0.031506
	0.029078	0.028194	0.027138	0.029637	0.029892	0.0287878	7.137711113736	0.71377111	0.02818	0.029681	0.030726	0.029179
	0.02733	0.026331	0.027451	0.027084	0.030194	0.027678	7.423910687188	0.67490097	0.027499	0.027372	0.029984	0.029226
	0.025656	0.025011	0.027084	0.026657	0.025046	0.0258908	7.936371220665	0.66136427	0.025465	0.025763	0.025723	0.027852
	No schedule #1	No schedule #2	No schedule #3	No schedule #4	No schedule #5	No schedule						
	0.157484	0.259457	0.278241	0.159148	0.257963	0.2224586						
	0.132815	0.131252	0.146006	0.081393	0.134077	0.1251086						
	0.095098	0.084366	0.098024	0.078525	0.090318	0.0912662						
	0.071266	0.071823	0.073063	0.06606	0.0676	0.0699624						
	0.05709	0.056597	0.058448	0.053468	0.054414	0.0560034						
	0.047744	0.047259	0.048913	0.047363	0.045438	0.0473434						
	0.043573	0.044912	0.044316	0.044796	0.042794	0.0440782						
	0.040516	0.041629	0.043421	0.042216	0.039332	0.0414228						
	0.04013	0.037118	0.035314	0.038162	0.035099	0.0371646						
	0.032974	0.033474	0.032043	0.034747	0.032171	0.0330818						
	0.029958	0.030539	0.030443	0.03038	0.030979	0.0304598						
	0.027829	0.027937	0.028126	0.041046	0.027565	0.0305006						



## Заключение

В ходе данного исследования была разработана параллельная программа для поиска максимального элемента в массиве для разных типов планировщика. Первоначально, на основе параллельной версии алгоритма, были построены параллельные версии для каждого расписания, и было измерено время выполнения программы для всех типов.

При использовании static, время выполнения программы значительно уменьшается (при сравнении с dynamic). Таким образом время выполнения на 12 потоках с размером фрагмента = 1 отличается больше чем в 36 раз. Но, если выставить оптимальный для нашей задачи размер фрагмента, то отличия не будут уже такими большими. 0.03085 для static, и 0.0471 для dynamic. Для данного алгоритма лучше всего себя показало расписание guided. Используя его, программа достигла лучшее время на каждом потоке. В guided размер блока начинается с большого значения и уменьшается по мере выполнения итераций, что позволяет лучше справляться с неравномерной нагрузкой.

В результате работы программы была получена зависимость среднего времени выполнения  $T(n)$  для каждого планировщика. Также была вычислена зависимость ускорения от числа потоков по формуле:  $A(n) = T(n) / T(1)$ . После этого была рассчитана зависимость эффективности от числа потоков по формуле:  $E(n) = A(n)/n$ , Где  $T$  – время (в секундах) выполнения программы,  $n$  – количество потоков,  $A$  – ускорение,  $E$  – эффективность.

При добавлении дополнительных потоков для каждого расписания уровень эффективности снижается с увеличением числа потоков. Эффективность при динамическом расписании достигла наименьшего значения, которое равно ~ 0.1286. Самая лучшая эффективность была достигнута при использовании планировщика типа guided (0.9094). Для static и guided при каждом добавлении нового потока снижение эффективности происходило в интервале  $[0; 0.1]$ , что говорит о монотонном убывании. При добавлении дополнительных потоков ускорение увеличивается. Наибольшее ускорение было достигнуто с помощью guided, и составило 7.9363. С каждым новым потоком оно увеличивалось в интервале  $[0; 1]$ . Static так же показало хорошие результаты ускорения ~ 6.0585.

Дополнительно, я рассмотрел влияние размеров фрагмента на время выполнения алгоритма. Путем сравнения разных типов планирования, мною было установлено, что для static время выполнения незначительно увеличивается с увеличением размера фрагмента, в то время как для dynamic время выполнения сильно уменьшается. Увеличение размера фрагмента в планировщике static приводит к тому, что каждый поток получает больше итераций на выполнение. Это может повлечь более равномерное распределение нагрузки, но также может увеличить неравномерное завершение работы потоков и накладные расходы на управление потоками. В свою очередь, увеличение в dynamic позволяет каждому потоку получать большие порции работы динамически во время выполнения. Это обычно уменьшает накладные расходы на управление потоками и позволяет более эффективно распределить работу между потоками. Так же мною был рассмотрен случай, когда `chunk_size = 100000` и `count/12`. Лучшее всего себя показало расписание dynamic, достигнув минимального времени в  $\sim 0.0249$ . Это самое лучшее время из всех тестов. Такой результат был достигнут из-за того, что каждый поток обрабатывал почти одинаковый размер данных, тем самым снижалось время выполнения работы. Дополнительно я рассмотрел случай, когда `chunk_size = count`. При таком размере фрагментов время выполнения работы на всех потоках почти никак не отличается, из чего мною был сделан вывод, что на один поток распределился весь массив, и остальные потоки остались без действия.

Таким образом, результаты исследования позволяют сделать вывод, что эффективность разных типов планировщика зависит от размеров фрагмента и особенностей данных, с которыми они работают.