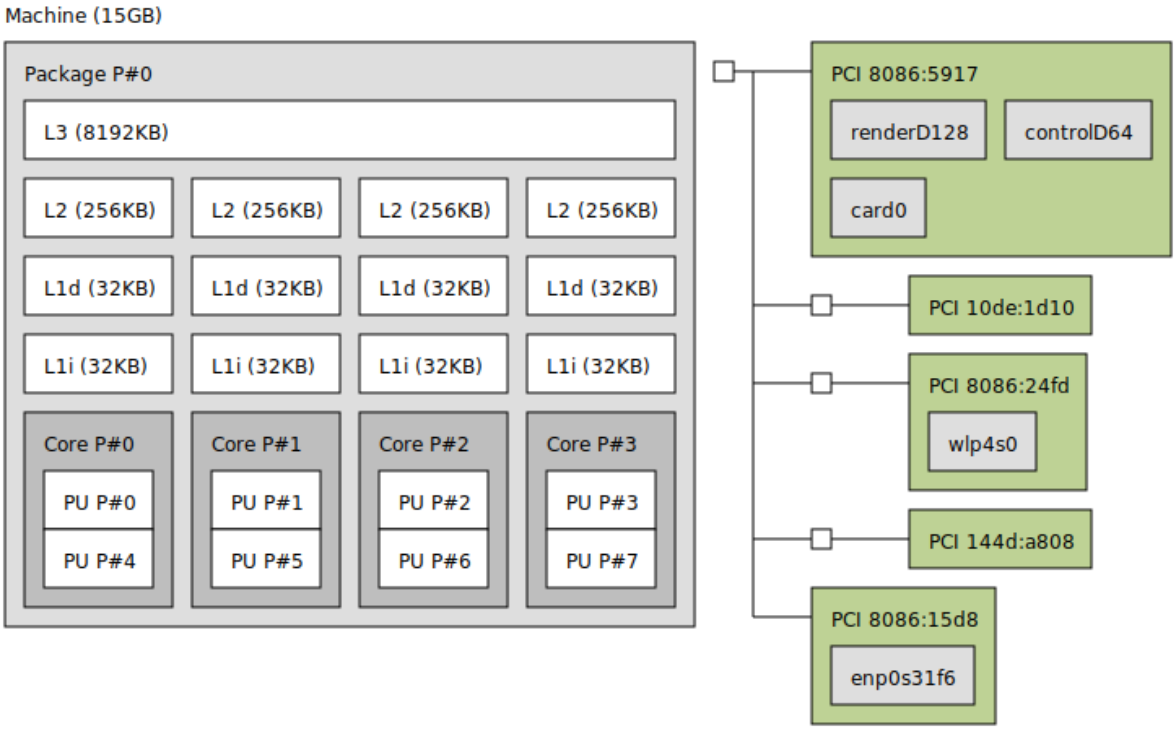
1. Конфигурация системы

- Ubuntu 18.04
- CPU: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
    - 4 cores (8 virtual) 1.8 GHz
    - L1: 32KB + 32KB
    - L2: 256KB
    - L3: 8192KB
    - `lstopo`:



- RAM:
    - 2 slots
    - 1 used:

```
sudo dmidecode -t 17

# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 3.0.0 present.

Handle 0x0004, DMI type 17, 40 bytes
Memory Device
Array Handle: 0x0003
Error Information Handle: Not Provided
Total Width: 64 bits
Data Width: 64 bits
Size: 16384 MB
```

```
Form Factor: SODIMM
Set: None
Locator: ChannelA-DIMM0
Bank Locator: BANK 0
Type: DDR4
Type Detail: Synchronous Unbuffered (Unregistered)
Speed: 2400 MT/s
Manufacturer: Samsung
Serial Number: 40E84A8C
Asset Tag: None
Part Number: M471A2K43CB1-CRC
Rank: 2
Configured Clock Speed: 2400 MT/s
Minimum Voltage: Unknown
Maximum Voltage: Unknown
Configured Voltage: 1.2 V

Handle 0x0005, DMI type 17, 40 bytes
Memory Device
Array Handle: 0x0003
Error Information Handle: Not Provided
Total Width: Unknown
Data Width: Unknown
Size: No Module Installed
Form Factor: Unknown
Set: None
Locator: ChannelB-DIMM0
Bank Locator: BANK 2
Type: Unknown
Type Detail: None
Speed: Unknown
Manufacturer: Not Specified
Serial Number: Not Specified
Asset Tag: Not Specified
Part Number: Not Specified
Rank: Unknown
Configured Clock Speed: Unknown
Minimum Voltage: Unknown
Maximum Voltage: Unknown
Configured Voltage: Unknown
```

- Network card:
  - 00:1f.6 Ethernet controller: Intel Corporation Ethernet Connection (4) I219-V (rev 21)
    - speed: 1Gbit/s
  - 04:00.0 Network controller: Intel Corporation Wireless 8265 / 8275 (rev 78)
- Disk:
  - NVMe SSD Controller SM981/PM981 Samsung Electronics Co Ltd

```
sudo fdisk -l | grep '^Disk /dev/' | egrep -v
'/dev/(loop|mapper|md)'

Disk /dev/nvme0n1: 477 GiB, 512110190592 bytes, 1000215216
sectors
```

- Заявлено up to 3,500 MB/s of sequential read throughput and 250,000 random read IOPS
- Гарантии нет

2. Частота CPU в зависимости от числа тредов

Для тестирования воспользовался семинарским методом по максимальной нагрузке процессора. Итоговая программа поддерживает запуск с заданным числом тредов и итераций. При запуске проводилась привязка к конкретным ядрам (через taskset). В зависимости от числа тредов есть до 8 процессов (по 1 на ядро), они же порождают треды (в равном количестве каждый).

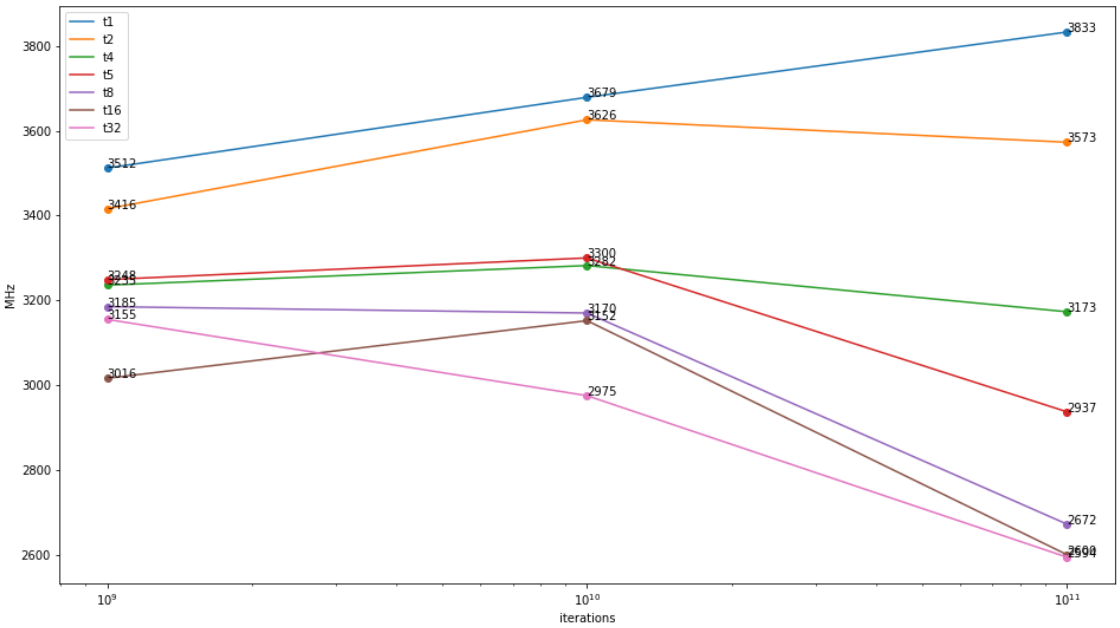
Код

```
long long run_op(long long iterations) {
  auto v = iterations, u = iterations, g = iterations;
  asm (
      "gg%=:\n\t"
      "dec %0\n\t"
      "dec %1\n\t"
      "dec %2\n\t"
      "dec %3\n\t"
      "jnz gg%=\n\t"
      : "+r"(iterations), "+r"(v), "+r"(u), "+r"(g)
      );
  return iterations;
}

int main(int argc, char **argv) {
    if (argc < 3) {
      std::cerr << "Please provide the number of threads to spawn and
number of iterations\n";
    }
    long long threads_count = std::atoll(argv[1]);
    long long iterations = std::atoll(argv[2]);

    std::vector<std::thread> threads;
    threads.reserve(threads_count);

    for (auto i = 0; i < threads_count; ++i) {
      threads.emplace_back([&]() {
        run_op(iterations);
      });
    }
```

```
    for (auto &t: threads) {
      t.join();
    }

    return 0;
}
```

Результаты

В целом видим что на кратковременной нагрузке активно используется турбобуст, быстро испаряющийся с ростом длительности нагрузки:



3. Разрешающая способность времени

Для определения разрешающей способности много раз сделаем замеры подряд и посчитаем diff, усреднив на число итераций:

Код

```
long long run_op_td_ns(long long iterations) {
  long long diffs = 0;
  timeval *vals = new timeval[2 * iterations];
  for (auto i = 0; i != 2 * iterations; i += 2) {
    gettimeofday(&vals[i], NULL);
    gettimeofday(&vals[i + 1], NULL);
  }
```

```
  for (auto i = 0; i != 2 * iterations; i += 2) {
    diffs += (vals[i + 1].tv_sec - vals[i].tv_sec) * 1000000 + (vals[i +
1].tv_usec - vals[i].tv_usec);
  }
  delete []vals;
  return diffs * 1000;
}

void run_op_td_test(long long original_iterations) {
  long long iterations = original_iterations;

  long long diffs = 0;
  long long limit = 100000000;
  while (iterations > 0) {
    auto count = std::min(iterations, limit);
    diffs += run_op_td_ns(count);
    iterations -= count;
  }

  std::cout << "Counting diff: " << std::setprecision(12) <<
std::setiosflags(std::ios::fixed | std::ios::showpoint) << static_cast
<long double>(diffs) / original_iterations << " nanoseconds" << std::endl;
}


long long run_high_res_clock(long long iterations) {
  auto time_points =
std::vector<std::chrono::high_resolution_clock::time_point>(2 *
iterations);
  for (auto i = 0; i != 2 * iterations; i += 2) {
    time_points[i] = std::chrono::high_resolution_clock::now();
    time_points[i + 1] = std::chrono::high_resolution_clock::now();
  }
  std::chrono::nanoseconds diffs = std::chrono::nanoseconds::zero();
  for (auto i = 0; i != 2 * iterations; i += 2) {
    diffs += std::chrono::duration_cast<std::chrono::nanoseconds>
(time_points[i + 1] - time_points[i]);
  }
  return diffs.count();
}

void run_op_high_res_clock_test(long long original_iterations) {
  long long iterations = original_iterations;

  long long diffs = 0;
  long long limit = 100000000;
  while (iterations > 0) {
    auto count = std::min(iterations, limit);
    diffs += run_high_res_clock(count);
    iterations -= count;
  }

  std::cout << "Counting diff: " << std::setprecision(12) <<
std::setiosflags(std::ios::fixed | std::ios::showpoint) << static_cast
```

```
<long double>(diffs) / original_iterations << " nanoseconds" << std::endl;
}

long long run_rdtsc_clock(long long iterations) {
  long long diff = 0;
  for (auto i = 0; i != iterations; ++i) {
    unsigned long long start = __rdtsc();
    unsigned long long end = __rdtsc();
    diff += end - start;
  }

  return diff;
}

void run_op_rdtsc_test(long long original_iterations) {
  auto diffs = run_rdtsc_clock(original_iterations);
  std::cout << "Counting diff: " << std::setprecision(12) <<
std::setiosflags(std::ios::fixed | std::ios::showpoint) << static_cast
<long double>(diffs) / original_iterations << " cycles" << std::endl;
}
```

Результаты

```
# gettimeofday
Counting diff: 26.882650000000 nanoseconds
# high_resolution_clock
Counting diff: 19.621143890000 nanoseconds
# rdtsc
Counting diff: 12.744493970000 cycles -> 3.41 nanoseconds (3724 GHz)
```

4. Измерение числа тактов на операции

Общий подход: чтобы разрешающей способностью времени можно было пренебречь будем делать много итераций между измерениями. Для более удобной генерации этих самих итераций воспользуемся препроцессором:

```
#define ONE(x) x
#define FIVE(x) ONE(x) ONE(x) ONE(x) ONE(x) ONE(x)
#define TEN(x) FIVE(x) FIVE(x)
#define FIFTY(x)    TEN(x) TEN(x) TEN(x) TEN(x) TEN(x)
#define HUNDRED(x)  FIFTY(x) FIFTY(x)
#define FIVE_HUNDRED(x) HUNDRED(x) HUNDRED(x) HUNDRED(x) HUNDRED(x)
HUNDRED(x)
#define THOUSAND(x) FIVE_HUNDRED(x) FIVE_HUNDRED(x)
#define FIVE_THOUSAND(x) THOUSAND(x) THOUSAND(x) THOUSAND(x) THOUSAND(x)
THOUSAND(x)
```

```
#define TEN_THOUSAND(x) FIVE_THOUSAND(x) FIVE_THOUSAND(x)
#define FIFTY_THOUSAND(x)   TEN_THOUSAND(x) TEN_THOUSAND(x)
TEN_THOUSAND(x) TEN_THOUSAND(x) TEN_THOUSAND(x)
#define HUNDRED_THOUSAND(x) FIFTY_THOUSAND(x) FIFTY_THOUSAND(x)
#define MILLION(x)  HUNDRED_THOUSAND(x) HUNDRED_THOUSAND(x)
HUNDRED_THOUSAND(x) HUNDRED_THOUSAND(x) HUNDRED_THOUSAND(x)
HUNDRED_THOUSAND(x) HUNDRED_THOUSAND(x) HUNDRED_THOUSAND(x)
HUNDRED_THOUSAND(x) HUNDRED_THOUSAND(x)
```

Целочисленное сложение:

Пользуясь вышеуказанной идеей посчитаем для сложения:

```
Counting diff: 0.880440509500 cycles
```

То есть 1 цикл на сложение

Код

```
long double plus_test(long long iterations) {
  long double diff = 0;
  iterations /= 10000000;
  for (auto i = 0; i != iterations; ++i) {
    long long copy = iterations;
    long long copy2 = iterations * 2;

    unsigned long long start = __rdtsc();

    asm (
    MILLION("add %0, %1\n\t" "add %1, %0\n\t")
    MILLION("add %0, %1\n\t" "add %1, %0\n\t")
    MILLION("add %0, %1\n\t" "add %1, %0\n\t")
    MILLION("add %0, %1\n\t" "add %1, %0\n\t")
    MILLION("add %0, %1\n\t" "add %1, %0\n\t")
    : "+r"(copy), "+r"(copy2)
    );
    unsigned long long end = __rdtsc();
    diff += static_cast<long double>(end - start) / 10000000;
  }

  return diff / iterations;
}

void run_plus_test(long long original_iterations) {
  auto result = plus_test(original_iterations);
  std::cout << "Counting diff: " << std::setprecision(12) <<
std::setiosflags(std::ios::fixed | std::ios::showpoint) <<  result << "
cycles" << std::endl;
}
```

Целочисленное умножение:

Пользуясь вышеуказанной идеей и добавив зависимостей между данными посчитаем для умножения:

```
Counting diff: 2.006500187170 cycles
```

То есть 2 цикла на умножение

Код

```
long double mul_test(long long iterations) {
  long double diff = 0;
  iterations /= 10000000;
  for (auto i = 0; i != iterations; ++i) {
    long long copy = iterations;
    long long copy2 = iterations * 2;

    unsigned long long start = __rdtsc();

    asm (
    MILLION("imul %0, %1\n\t" "imul %1, %0\n\t")
    MILLION("imul %0, %1\n\t" "imul %1, %0\n\t")
    MILLION("imul %0, %1\n\t" "imul %1, %0\n\t")
    MILLION("imul %0, %1\n\t" "imul %1, %0\n\t")
    MILLION("imul %0, %1\n\t" "imul %1, %0\n\t")
    : "+r"(copy), "+r"(copy2)
    );
    unsigned long long end = __rdtsc();
    diff += static_cast<long double>(end - start) / 10000000;
  }

  return diff / iterations;
}

void run_mul_test(long long original_iterations) {
  auto result = mul_test(original_iterations);
  std::cout << "Counting diff: " << std::setprecision(12) <<
std::setiosflags(std::ios::fixed | std::ios::showpoint) <<  result << "
cycles" << std::endl;
}
```

Целочисленное деление:

Пользуясь вышеуказанной идеей и добавив зависимостей между данными посчитаем для деления:

```
Counting diff: 18.336062261500 cycles
```

То есть 18 циклов на деление

Код

```
long double div_test(long long iterations) {
  long double diff = 0;
  iterations /= 1000000;
  for (auto i = 0; i != iterations; ++i) {
    long long copy = std::numeric_limits<long long>::max();
    long long copy2 = sqrt(std::numeric_limits<long long>::max()) / 8192;
    long long copy3 = copy - 1;
    long long copy4 = copy2 + 1;

    unsigned long long start = __rdtsc();

    asm (
    MILLION(
       "mov $0, %%rdx\n\t""mov %0, %%rax\n\t" "mov %1, %%rbx\n\t""idiv
%%rbx\n\t"
       "mov $0, %%rdx\n\t" "mov %2, %%rbx\n\t""idiv %%rbx\n\t"
       "mov $0, %%rdx\n\t" "mov %3, %%rbx\n\t""idiv %%rbx\n\t"
       "mov $0, %%rdx\n\t" "mov $117, %%rbx\n\t""idiv %%rbx\n\t"
   )

    :: "g"(copy), "g"(copy2), "g"(copy3), "g"(copy4)
    );
    unsigned long long end = __rdtsc();
    diff += static_cast<long double>(end - start) / 1000000 / 4;
  }
   return diff / iterations;
}

void run_div_test(long long original_iterations) {
  auto result = div_test(original_iterations);
  std::cout << "Counting diff: " << std::setprecision(12) <<
std::setiosflags(std::ios::fixed | std::ios::showpoint) <<  result << "
cycles" << std::endl;
}
```

5. Максимальное число арифметических операций за такт

1. Попробуем выполнить операции, не пересекающиеся по данным, и посмотрим, что получится:

```
long long run_op(long long iterations) {
  auto v = iterations, u = iterations, g = iterations;
  asm (
      "gg%=:\n\t"
      "dec %0\n\t"
      "dec %1\n\t"
      "dec %2\n\t"
      "dec %3\n\t"
      "jnz gg%=\n\t"
      : "+r"(iterations), "+r"(v), "+r"(u), "+r"(g)
      );
  return iterations;
}
```

Результат 5 инструкций за цикл:

```
Performance counter stats for 'taskset 1 ./main 1 10000000000':

       2703,239919      task-clock (msec)         #    1,000 CPUs utilized
                10      context-switches          #    0,004 K/sec
                 1      cpu-migrations            #    0,000 K/sec
               183      page-faults               #    0,068 K/sec
    10 025 397 713      cycles                    #    3,709 GHz
    50 007 531 709      instructions              #    4,99  insn per
cycle
    10 001 337 001      branches                  # 3699,759 M/sec
            44 699      branch-misses             #    0,00% of all
branches

       2,704086100 seconds time elapsed
```

2. Попробуем добавить еще одну:

```
long long run_op(long long iterations) {
  auto v = iterations, u = iterations, g = iterations, o = iterations;
  asm (
      "gg%=:\n\t"
      "dec %0\n\t"
      "dec %1\n\t"
      "dec %2\n\t"
      "dec %3\n\t"
      "dec %4\n\t"
      "jnz gg%=\n\t"
      : "+r"(iterations), "+r"(v), "+r"(u), "+r"(g), "+r"(o)
      );
  return iterations;
}
```

Имеем проседание по числу инструкций:

```
 Performance counter stats for 'taskset 1 ./main 1 10000000000':

       5378,305771      task-clock (msec)         #    1,000 CPUs utilized
                14      context-switches          #    0,003 K/sec
                 1      cpu-migrations            #    0,000 K/sec
               188      page-faults               #    0,035 K/sec
    20 101 567 318      cycles                    #    3,738 GHz
    60 010 879 158      instructions              #    2,99  insn per
cycle
    10 001 920 078      branches                  # 1859,679 M/sec
            65 664      branch-misses             #    0,00% of all
branches

       5,378997852 seconds time elapsed
```

3. Запустим теперь два экземпляра на неспаренных cpu:

```
gostkin@gostkin-workstation:~/distrsyshw$ for i in 0 1; do sudo perf stat
taskset `python -c "print hex(1<<${i})"` ./main 1 10000000000 & done
[1] 13398
[2] 13399
gostkin@gostkin-workstation:~/distrsyshw$
 Performance counter stats for 'taskset 0x2 ./main 1 10000000000':

       2782,227584      task-clock (msec)         #    1,000 CPUs utilized
                 9      context-switches          #    0,003 K/sec
                 1      cpu-migrations            #    0,000 K/sec
               185      page-faults               #    0,066 K/sec
    10 034 767 993      cycles                    #    3,607 GHz
    50 007 227 270      instructions              #    4,98  insn per
cycle
    10 001 302 616      branches                  # 3594,710 M/sec
            43 032      branch-misses             #    0,00% of all
branches

       2,783220241 seconds time elapsed

 Performance counter stats for 'taskset 0x1 ./main 1 10000000000':

       2793,019705      task-clock (msec)         #    1,000 CPUs utilized
                 7      context-switches          #    0,003 K/sec
                 1      cpu-migrations            #    0,000 K/sec
               187      page-faults               #    0,067 K/sec
    10 063 978 869      cycles                    #    3,603 GHz
    50 007 611 131      instructions              #    4,97  insn per
cycle
    10 001 351 435      branches                  # 3580,838 M/sec
            46 384      branch-misses             #    0,00% of all
```

```
branches

       2,794038243 seconds time elapsed
```

Результат тот же самый: 5 инструкций за цикл 4. Запустим на спаренных (hyperthreading):

```
gostkin@gostkin-workstation:~/distrsyshw$ for i in 0 4; do sudo perf stat
taskset `python -c "print hex(1<<${i})"` ./main 1 10000000000 & done
[1] 13422
[2] 13423
gostkin@gostkin-workstation:~/distrsyshw$
 Performance counter stats for 'taskset 0x10 ./main 1 10000000000':

       6169,620445      task-clock (msec)         #    1,000 CPUs utilized
                47      context-switches          #    0,008 K/sec
                 1      cpu-migrations            #    0,000 K/sec
               188      page-faults               #    0,030 K/sec
    20 053 377 980      cycles                    #    3,250 GHz
    50 010 966 585      instructions              #    2,49  insn per
cycle
    10 001 962 670      branches                  # 1621,163 M/sec
            58 132      branch-misses             #    0,00% of all
branches

       6,171530146 seconds time elapsed

 Performance counter stats for 'taskset 0x1 ./main 1 10000000000':

       6171,258621      task-clock (msec)         #    1,000 CPUs utilized
                16      context-switches          #    0,003 K/sec
                 1      cpu-migrations            #    0,000 K/sec
               186      page-faults               #    0,030 K/sec
    20 062 014 567      cycles                    #    3,251 GHz
    50 011 637 114      instructions              #    2,49  insn per
cycle
    10 002 043 705      branches                  # 1620,746 M/sec
            60 537      branch-misses             #    0,00% of all
branches

       6,172608473 seconds time elapsed
```

Имеем просадку с 5 инструкций за цикл на 2.5, собственно, что и ожидалось.

6. Производительность кодеков lz4 и zstd

Сгенерируем файл с рандомными данными: `test_rand`. Размер 1250595 килобайт. Сожмем и разожмем lz4:

```
gostkin@gostkin-workstation:~/distrsyshw/test$ sudo perf stat lz4 -9
test_rand test_rand.lz4
Compressed 1280608107 bytes into 1280609346 bytes ==> 100.00%

 Performance counter stats for 'lz4 -9 test_rand test_rand.lz4':

      33583,942226      task-clock (msec)         #    0,999 CPUs utilized
               189      context-switches          #    0,006 K/sec
                 3      cpu-migrations            #    0,000 K/sec
             2 176      page-faults               #    0,065 K/sec
   117 345 162 331      cycles                    #    3,494 GHz
   122 365 825 973      instructions              #    1,04  insn per
cycle
    17 072 879 313      branches                  #  508,364 M/sec
     1 249 547 860      branch-misses             #    7,32% of all
branches

      33,615681804 seconds time elapsed

gostkin@gostkin-workstation:~/distrsyshw/test$ sudo perf stat unlz4
test_rand.lz4 test_rand.unpacked
Successfully decoded 1280608107 bytes

 Performance counter stats for 'unlz4 test_rand.lz4 test_rand.unpacked':

       1260,818243      task-clock (msec)         #    0,999 CPUs utilized
                28      context-switches          #    0,022 K/sec
                 0      cpu-migrations            #    0,000 K/sec
                92      page-faults               #    0,073 K/sec
     3 484 624 099      cycles                    #    2,764 GHz
     4 199 427 448      instructions              #    1,21  insn per
cycle
       556 321 221      branches                  #  441,238 M/sec
         5 303 732      branch-misses             #    0,95% of all
branches

       1,261950012 seconds time elapsed
```

Сожмем и разожмем zstd:

```
gostkin@gostkin-workstation:~/distrsyshw/test$ sudo perf stat zstd
test_rand -o test_rand.zstd
test_rand            :100.00%   (1280608107 => 1280637434 bytes,
test_rand.zstd)

 Performance counter stats for 'zstd test_rand -o test_rand.zstd':

       2027,022855      task-clock (msec)         #    0,995 CPUs utilized
                29      context-switches          #    0,014 K/sec
```

```
                0      cpu-migrations            #    0,000 K/sec
              658      page-faults               #    0,325 K/sec
     5 840 999 322      cycles                    #    2,882 GHz
     8 459 476 059      instructions              #    1,45  insn per
cycle
       748 361 532      branches                  #  369,192 M/sec
        10 028 438      branch-misses             #    1,34% of all
branches

       2,037375895 seconds time elapsed

gostkin@gostkin-workstation:~/distrsyshw/test$ sudo perf stat zstd -d
test_rand.zstd -o test_rand.zstd_unpacked
test_rand.zstd      : 1280608107 bytes

 Performance counter stats for 'zstd -d test_rand.zstd -o
test_rand.zstd_unpacked':

       1110,787489      task-clock (msec)         #    0,999 CPUs utilized
                36      context-switches          #    0,032 K/sec
                 0      cpu-migrations            #    0,000 K/sec
               465      page-faults               #    0,419 K/sec
     3 150 016 259      cycles                    #    2,836 GHz
     3 591 137 850      instructions              #    1,14  insn per
cycle
       507 847 327      branches                  #  457,196 M/sec
         4 995 527      branch-misses             #    0,98% of all
branches

       1,111988203 seconds time elapsed
```

Результат

Скорость сжатия 1 кб lz4: 26.88 микросекунд Скорость разжатия 1 кб lz4: 1 микросекунда

Скорость сжатия 1 кб zstd: 1.63 микросекунды Скорость разжатия 1 кб zstd: 0.89 микросекунд

7. Задача: cache latency

Идея:

- для замера l1 надо прочитать необходимые куски данных из памяти
- для замера l2 нужный нам кусок надо вытеснить из l1 что-нибудь прочитав
- для замера l3 нужный нам кусок нужно вытеснить из l2 что-нибудь опять же прочитав

Эта идея реализуется ниже. Результат:

```
Ram access is 67.7468 cycles
L1 access is 6.24414 cycles
```

```
L2 access is 14.0268 cycles
L3 access is 30.519 cycles
```

Код

```
const int L1_CACHE_SIZE = 32 * 1024 / sizeof(int);
const int L2_CACHE_SIZE = 256 * 1024 / sizeof(int);
const int L3_CACHE_SIZE = 8192 * 1024 / sizeof(int);

int main() {
  int *primary_array = new int[L3_CACHE_SIZE * 64];
  for (int i = 0; i < L1_CACHE_SIZE * 64; ++i) {
    primary_array[i] = 7;
  }

  long double l1_access = 0, l2_access = 0, l3_access = 0, ram_access = 0;

  size_t index = 0;
  size_t seen_elements = 0;

  unsigned long long start = __rdtsc();
  while (index < L1_CACHE_SIZE * 64) {
    auto current = primary_array[index];
    index += current + 57 + 64;
    ++seen_elements;
  }
  unsigned long long end = __rdtsc();

  ram_access = static_cast<long double>(end - start) / static_cast<long
double>(seen_elements);

  std::cout << "Ram access is " << ram_access << " cycles\n";

  for (int i = 0; i < L1_CACHE_SIZE; ++i) {
    primary_array[i] -= 1;
  }

  auto current = 0;
  seen_elements = 0;
  start = __rdtsc();
  for (int mod = 0; mod < 16; ++mod) {
    for (int i = mod; i < L1_CACHE_SIZE; i += 64) {
      current += primary_array[i];
    }
  }
  end = __rdtsc();
  for (int mod = 0; mod < 16; ++mod) {
    for (int i = mod; i < L1_CACHE_SIZE; i += 64) {
      ++seen_elements;
    }
```

```
  }

  l1_access = static_cast<long double>(end - start) / static_cast<long
double>(seen_elements);// / 4.0;

  std::cout << "L1 access is " << l1_access << " cycles with value " <<
current << "\n";

  for (int i = 0; i < L2_CACHE_SIZE - L1_CACHE_SIZE; ++i) {
    primary_array[i] -= 1;
  }
  for (int i = L1_CACHE_SIZE * 60; i < L1_CACHE_SIZE * 61; ++i) {
    primary_array[i] -= 1;
  }


  current = 0;
  seen_elements = 0;
  start = __rdtsc();
  for (int i = 0; i < L2_CACHE_SIZE - L1_CACHE_SIZE; i += 64) {
    current += primary_array[i];
  }
  end = __rdtsc();
  for (int i = 0; i < L2_CACHE_SIZE - L1_CACHE_SIZE; i += 64) {
    ++seen_elements;
  }

  l2_access = static_cast<long double>(end - start) / static_cast<long
double>(seen_elements);// / 4.0;

  std::cout << "L2 access is " << l2_access << " cycles with value " <<
current << "\n";


  for (int i = 0; i < L3_CACHE_SIZE; ++i) {
    primary_array[i] += 1;
  }
  for (int i = L2_CACHE_SIZE * 32; i < L2_CACHE_SIZE * 33; ++i) {
    primary_array[i] -= 1;
  }

  for (int i = L2_CACHE_SIZE * 34; i < L2_CACHE_SIZE * 34 + L1_CACHE_SIZE;
++i) {
    primary_array[i] -= 1;
  }

  current = 0;
  seen_elements = 0;
  start = __rdtsc();
  for (int i = 0; i < L3_CACHE_SIZE - 2 * L1_CACHE_SIZE - 2 *
L2_CACHE_SIZE; i += 64) {
    current += primary_array[i];
  }
  end = __rdtsc();
```

```
  for (int i = 0; i < L3_CACHE_SIZE - 2 * L1_CACHE_SIZE - 2 *
L2_CACHE_SIZE; i += 64) {
    ++seen_elements;
  }

  l3_access = static_cast<long double>(end - start) / static_cast<long
double>(seen_elements);// / 4.0;

  std::cout << "L3 access is " << l3_access << " cycles with value " <<
current << "\n";

  delete[] primary_array;
  return 0;
}
```

9. Задача

Замерив перфоманс, получаем, что unordered set ведет себя лучше:

data structure	fit in l1	fit in l2	fit in l3	fit in ram
unordered_set	52	62	192	514
set	62	172	341	1098

Код

```
int main() {
  std::random_device rd;
  std::mt19937 mt(rd());
  std::uniform_int_distribution<int> dist(-(8192 + 256 + 32) * 1024 * 100,
(8192 + 256 + 32) * 1024 * 100);

  int cache_total = 30; // 30 or 30 + 256 or 30 + 256 + 8192 or (30 + 256
+ 8192) * 50
  std::set<int> set; // or unordered set
  for (int i = 0; i < cache_total * 512 /sizeof(int); ++i) {
    set.insert(i);
    set.insert(-i);
  }

  unsigned long long diff = 0;
  for (int i = 0; i < 100000000; ++i) {
    auto find = dist(mt);

    auto start = __rdtsc();
    bool found = set.find(find) != set.end();
    auto end = __rdtsc();
```

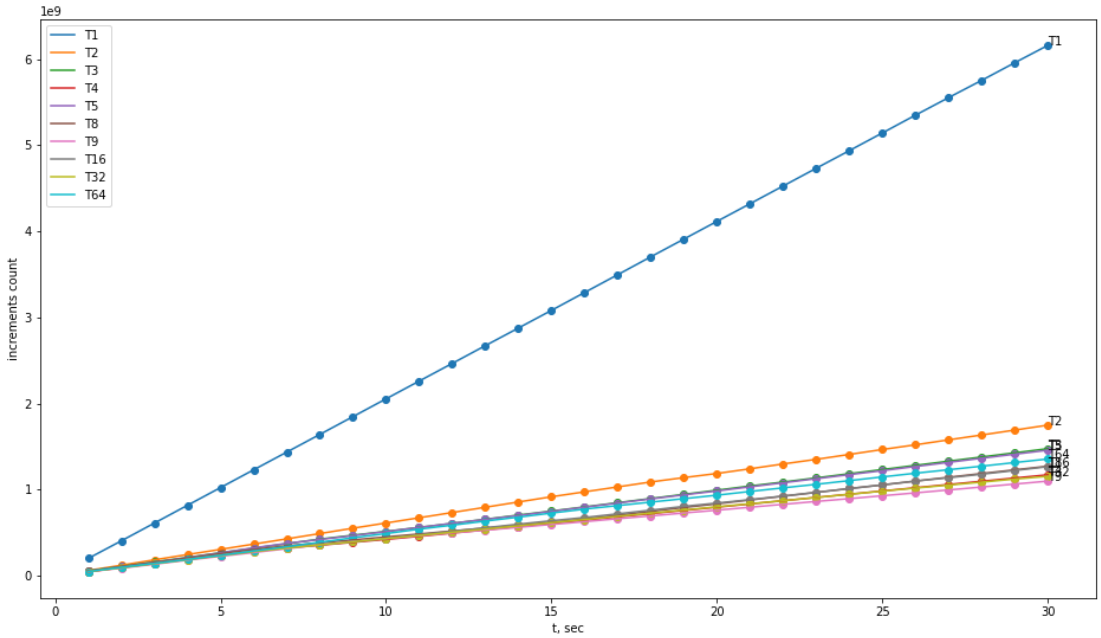
```
      diff += end - start;
      if (i % 10000000 == 0) {
        std::cout << found << std::endl;
      }
    }
    std::cout << static_cast<long double>(diff) / 100000000.0 << std::endl;
    return 0;
}
```

10. Задача

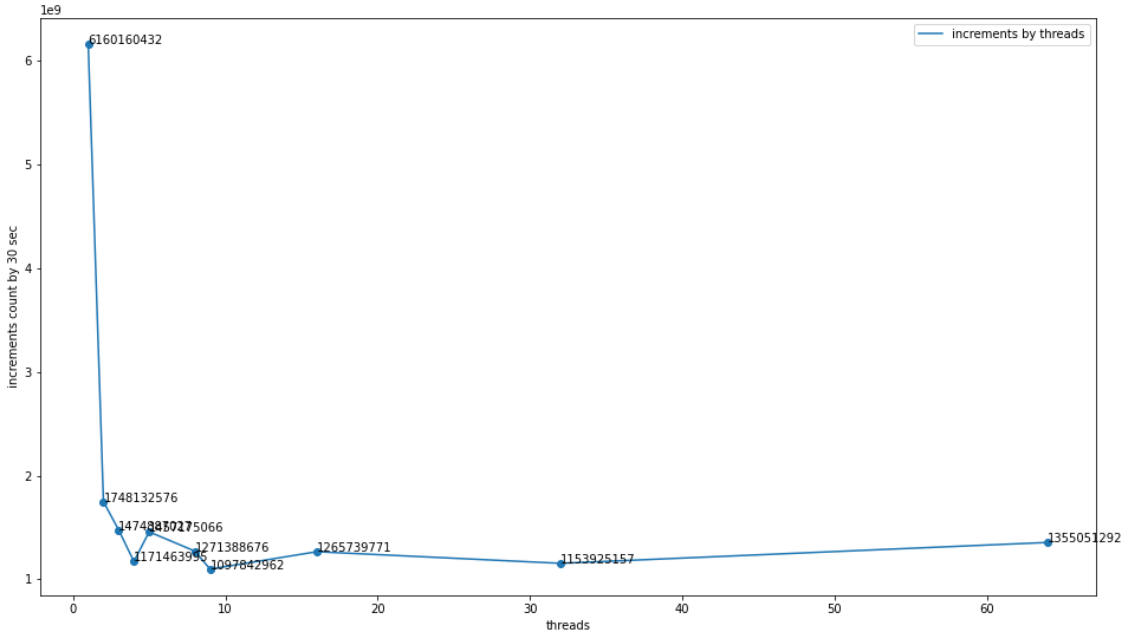
Для начала имплементировал стандартный класс:

```
class Counter1 {
  std::atomic<uint64_t> counter_{0};

public:
  Counter1() : counter_{0} {}

  void Increment(int threadIndex) {
    counter_.fetch_add(1);
  };

  size_t Gather() {
    return static_cast<size_t>(counter_.load());
  };
};
```

Протестировал на разном числе тредов, построил историю для первых 30 секунд:



Насколько я понял в задании спрашивалось также график построить в зависимости от числа тредов, взял за точку среда 30ю секунду работы:

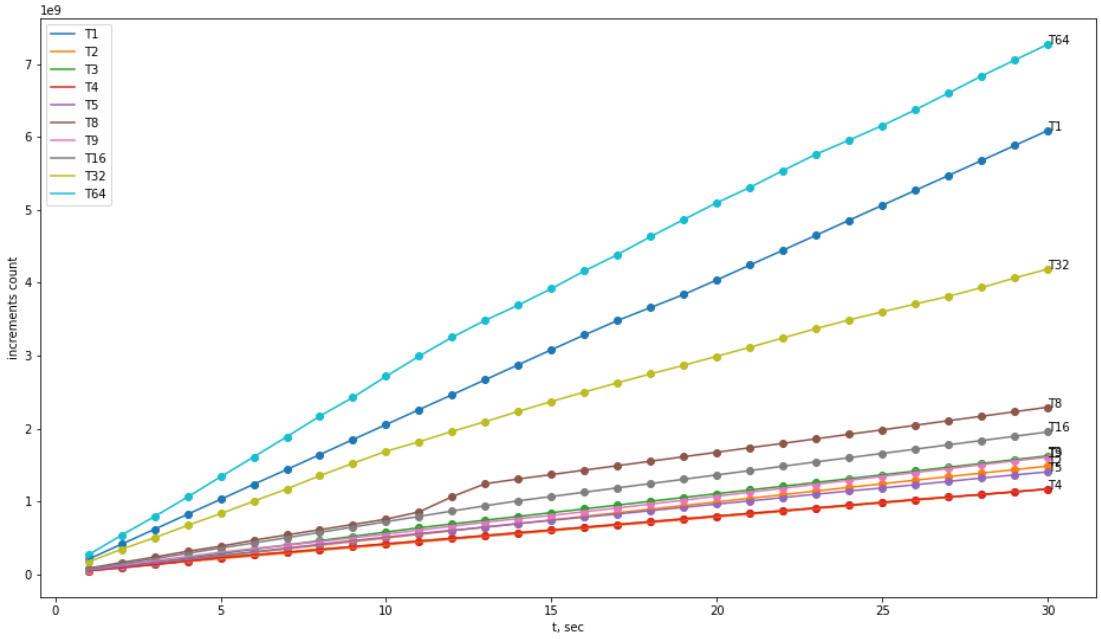


Видим что с ростом числа тредов падает скорость заметно (связываю с конкруренцией за атомик). Идея: давайте сделаем по атомику на тред. Вот что получилось:

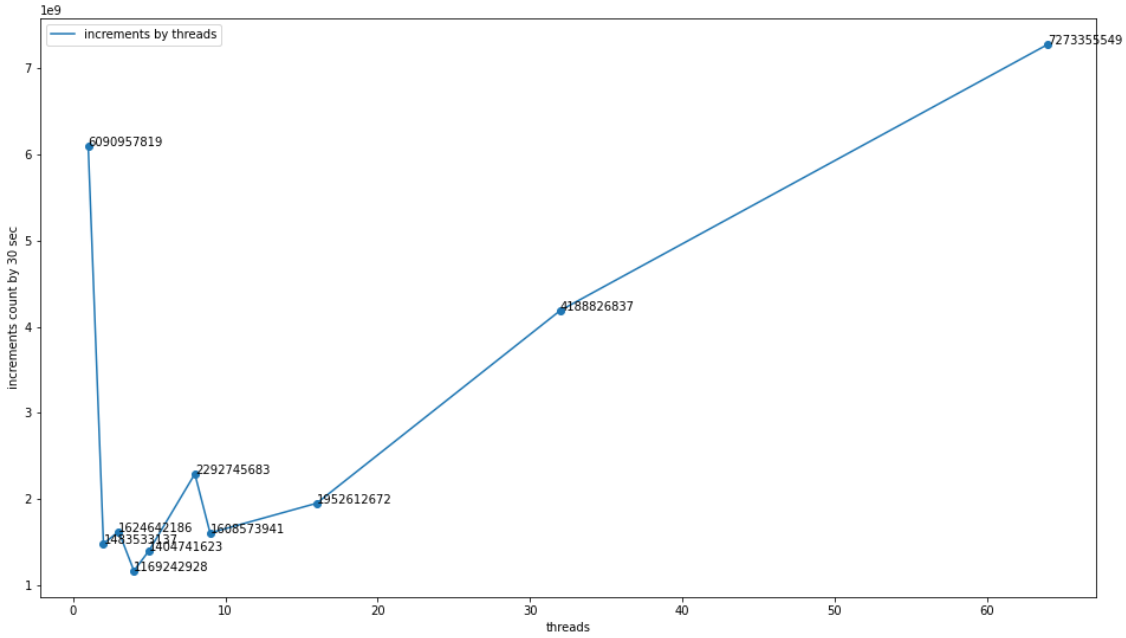
```
class Counter2 {
  std::array<std::atomic<size_t>, 64> values_;
  size_t threads_;
public:
  explicit Counter2(size_t thread_count): threads_{thread_count} {
```

```
    for (size_t i = 0; i != threads_; ++i) {
      values_[i].store(0);
    }
  }

  void Increment(int threadIndex) {
    values_[threadIndex] += 1;
  };

  size_t Gather() {
    size_t result = 0;

    for (size_t i = 0; i != threads_; ++i) {
      result += values_[i].load();
    }

    return result;
  };

};
```

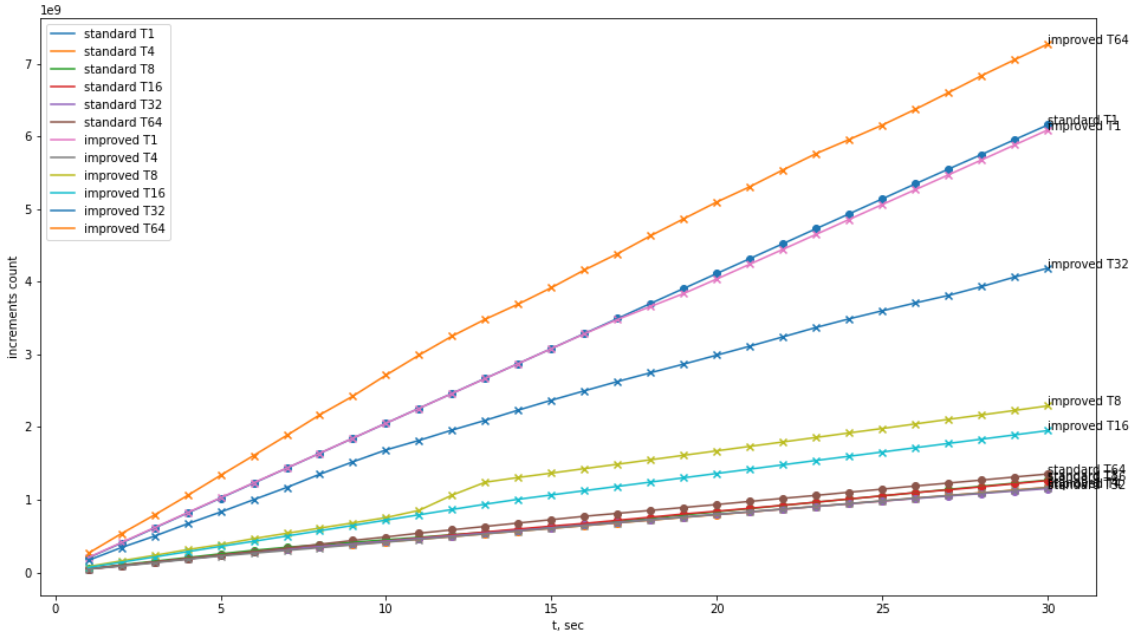
Вот что получилось:



И по 30 секнуде:



Как видно, результат получается гораздо лучше. В частности, этому способствует тот факт, что вызов `Gather` происходит редко, отсюда конкуренция за атомики между тредами возникает сильно реже, чем раньше.



Вспомогательный код:

```
if (argc < 4) {
    std::cerr << "Please provide the number of threads to spawn and number
of iterations\n";
  }
  long long threads_count = std::atoll(argv[1]);
```

```
  long long iterations = std::atoll(argv[2]);
  long long secs = std::atoll(argv[3]);

  std::vector<std::thread> threads;
  threads.reserve(threads_count);

  Counter1 counter{};//{static_cast<size_t>(threads_count)};

  for (auto i = 0; i < threads_count; ++i) {
    threads.emplace_back([i=i, iterations=iterations, &counter]() {
      for (size_t j = 0; j != iterations; ++j) {
        counter.Increment(i);
      }
    });
  }

  auto start = std::chrono::high_resolution_clock::now();
  for (auto i = 0; i < secs; ++i) {
    std::this_thread::sleep_for(std::chrono::seconds (1));
    auto now = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(now -
start);
    std::cout <<'(' << elapsed.count() << ',' << counter.Gather()<< "),"
<< std::endl;
  }
  for (auto &t: threads) {
    t.join();
  }

  assert(counter.Gather() == threads_count * iterations);

  return 0;
```

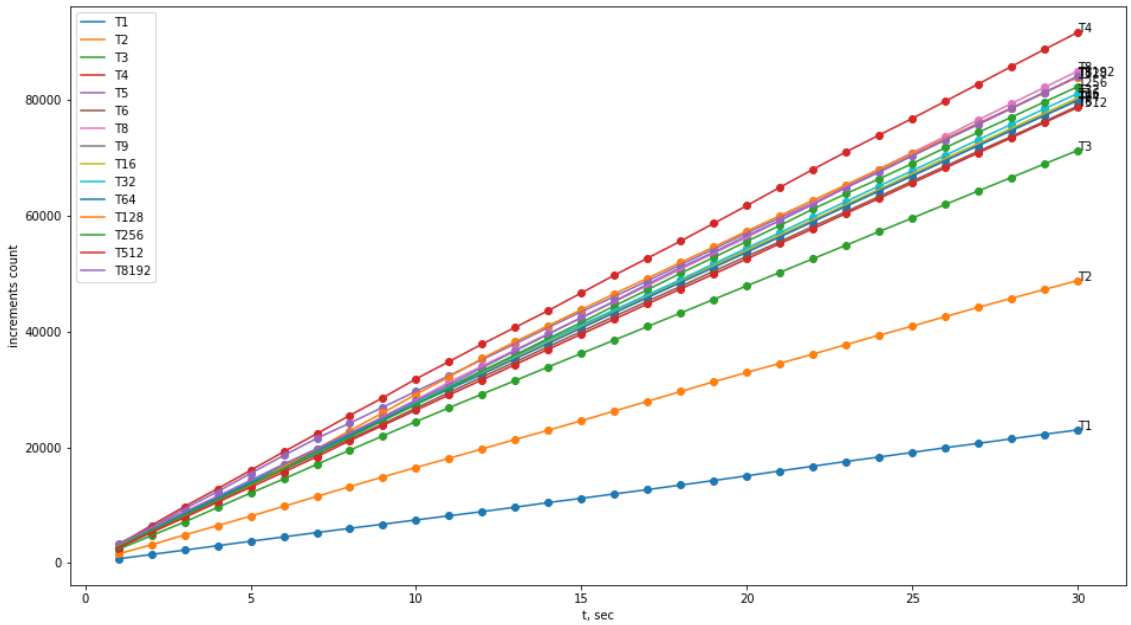
11. Задача

Если правильно понял задачу, критическая секция должна быть общая между всеми тредами. Для нее воспользуемся обычным мьютексом. В остальном можно много чего переиспользовать из предыдущей задачи (Counter2 - оптимизированный, для подсчета результатов, да и способ запуска и остановки тредов):

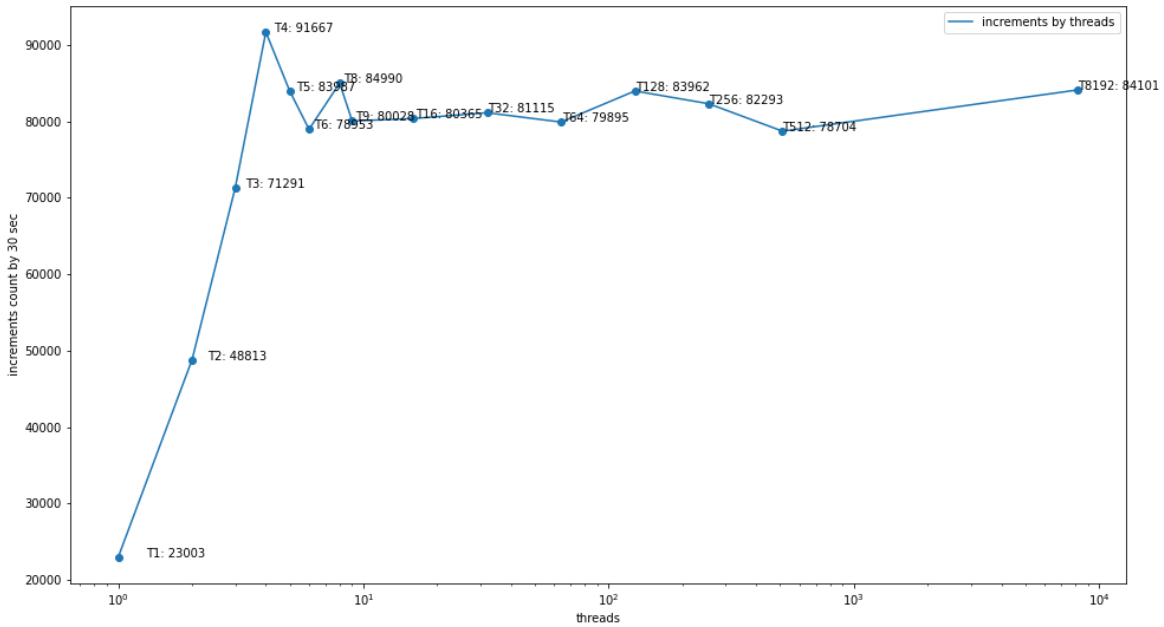
```
int main(int argc, char **argv) {
  if (argc < 4) {
    std::cerr << "Please provide the number of threads to spawn and number
of iterations\n";
  }
  long long threads_count = std::atoll(argv[1]);
  long long iterations = std::atoll(argv[2]);
  long long secs = std::atoll(argv[3]);
```

```
  std::vector<std::thread> threads;
  threads.reserve(threads_count);

  Counter2 counter{static_cast<size_t>(threads_count)};

  std::mutex mutex_;
  for (auto i = 0; i < threads_count; ++i) {
    threads.emplace_back([i=i, iterations=iterations, &counter, &mutex_]()
{
      for (size_t j = 0; j != iterations; ++j) {
        {
          std::unique_lock<std::mutex> lock(mutex_);
          std::this_thread::sleep_for(std::chrono::microseconds(200));
        }
        std::this_thread::sleep_for(std::chrono::microseconds(800));
        counter.Increment(i);
      }
    });
  }

  auto start = std::chrono::high_resolution_clock::now();
  for (auto i = 0; i < secs; ++i) {
    std::this_thread::sleep_for(std::chrono::seconds (1));
    auto now = std::chrono::high_resolution_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::seconds>(now -
start);
    std::cout <<'(' << elapsed.count() << ',' << counter.Gather()<< "),"
<< std::endl;
  }
  for (auto &t: threads) {
    t.join();
  }

  assert(counter.Gather() == threads_count * iterations);

  return 0;
}
```

Посмотрим на графики, что получилось:



И по 30 секнуде:



Как видно, если тредов > 8, все стабилизируется в смысле перфоманса.

12. Производительность диска

Как уже указал в п.1 у меня SSD a-d: последовательное чтение, запись, произвольное чтение, запись, в MB/s:

block size	write	read	randwrite	randread

block size	write	read	randwrite	randread
1k	0.65	757	0.86	44.2
8k	6.7	1524	6.7	86.5
32k	23.5	1671	23.9	174
128k	76.2	1558	77.3	334
256k	121	1568	126	416
1m	188	1370	190	468
4m	364	1477	300	659
16m	337	1600	330	736
32m	355	1550	364	731
128m	387	1519	379	705

Вышеуказанные значения были получены при работе на достаточно крупных объемах данных (~10gb) В моменте скорость записи бывает и до 1000MB/s, но стабильно деградирует до ~380. Если поусреднять, скорости: последовательного чтения - 1671 при block size 32k последовательной записи - 364 при block size 4m произвольного чтения - 736 при block size 16m произвольной записи - 360 при block size 32m

e) скорость произвольной записи: ~37мс (при 16гб данных, 32m block size, скорость 338MB/s). Если данных меньше (2gb), то 23мс, скорость 740MB/s. Если direct = 1, то скорость 1225MB/s на 2gb, скорость 23мс Если sync = 1, то скорость 608MB/s на 2gb, скорость 54мс Если direct и sync = 1, то скорость 994MB/s на 2gb, скорость 31мс

Если direct = 1, то скорость 662MB/s на 16gb, скорость 30мс Если sync = 1, то скорость 390MB/s на 16gb, скорость 85мс Если direct и sync = 1, то скорость 595MB/s на 16gb, скорость 55мс

f) Время доступа при rand read: 43мс (16g данных, 32m блок)