Лабораторная Работа №4	M3100	2023
OpenMP	Ерёмин Владимир Ильич	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий и требования к работе: C++20, Clang *(15.0.1)*, CMake *(3.24.202208181-MSVC_2)* и OpenMP 2.0.

Описание: изучить конструкции OpenMP для распараллеливания вычислений. Решить предложенную задачу. Оформить отчет.

Задача: необходимо написать программу, решающую задачу: “Пороговая фильтрация изображения методом Оцу”. Необходимо реализовать алгоритм для трех порогов.

1. Описание конструкций OpenMP для распараллеливания команд

OpenMP использует «fork-join» модель параллельного исполнения. Как сказано в спецификации, она более заточена под оптимизацию работы с большими массивами данных. Причем, предполагается, что написанная программа будет исполняться корректно как при последовательном, так и при параллельном выполнении. Однако никто не запрещает написать программу, которая будет работать верно только при определенных параметрах OpenMP. Также спецификация OpenMP перекладывает всю ответственность за работоспособность кода на программиста. Так реализации API не гарантируют никаких проверок на взаимные блокировки, состояния гонки и другие проблемы, которые могут возникнуть по ходу исполнения программы использующее данное API.

Изначально такая модель, как и при обычном исполнении, имеет единственный поток (master-thread).

Поток – это последовательный исполнитель, имеющий набор локальных переменных и доступ к глобальным (общим между потоками) переменным.

Master-thread стандартно выполняет программу до тех пор, пока не встретит директиву ***parallel*** *(п. 1.2).* Тогда, master-thread создает команду из потоков и становится ее мастером. Следующий за данной директивой блок будет выполнен всей командой параллельно независимо друг от друга до тех пор, пока им не встретится директива «распределяющая работу». В таком случае потоки разделят следующую за ней «работу» между друг другом.

Для контроля команды внутри параллельного региона имеются такие директивы как: ***barrier, critical, single и т. д.*** После каждой директивы распределения работы имеется неявный ***barrier,*** который «выравнивает» ход выполнения блока потоками. Можно проигнорировать его, используя модификатор ***nowait***. ***Barrier*** также неявно стоит в конце параллельного блока, однако master-thread его пропустить не может так, как отвечает за команду в блоке.

Локально, каждый поток в команде сам себе master-thread, который имеет право создать свою команду во *вложенном параллельном регионе.*

1.1. Структура директив в OpenMP

Директивы OpenMP задаются, через директивы языка препроцессора C/C++:

```
1. #pragma omp директива [модификаторы, ...]
```

Это дополнительные указания компилятору (который должен иметь поддержку OpenMP) на то, что требуется сделать в закрепленном к директиве регионе.

Регион – это, следующий за директивой структурированный блок/выражение.

Важно, что любая часть региона не должна попадать на строку с самой директивой (интерпретируется, как часть *pragma*).

Для многих директив регион отмечается блоком «{}».

Однако для некоторых директив (**atomic, section, …**) — это необязательное условие, и такой блок может быть опущен, однако могут и появиться другие условия их использования.

1.2. Omp parallel – база

Основная директива OpenMP:

```
1. #pragma omp parallel [модификаторы, …]
```

Как было описано выше (п. 1): по достижении master-thread'ом – создается команда потоков, для параллельного исполнения региона.

Контроль над исполнением данной директивы производит модификатор ***if(****bool****)***. Так, команда потоков будет создана, если:

- Модификатор не задан.
- Модификатор задан и *bool* принимает истинное значение.

(Буквально, определяет состояние OpenMP в регионе)

Количество потоков в создаваемой команде можно определить через (от менее приоритетного):

- «Зависит от реализации API».
- Переменную среды: **OMP_NUM_THREADS**.
- Через функцию: ***omp_set_num_threads(****int****)***.
- Через модификатор: ***num_threads(****int****)***.

Также, через значение переменной среды **OMP_DYNAMIC** или вызов функции ***omp_set_dynamic(****int****)***, можно разрешить или запретить динамический выбор количества потоков в команде.

(рекомендуется выключить при задании количества потоков вручную)

Как было сказано ранее (п. 1): API разрешает использование вложенных параллельных регионов, однако по умолчанию, это приведет к созданию команды из одного потока, который выполнит вложенный регион последовательно.

1.3. Распределяющие работу директивы

1.3.1. Omp for

Основной метод распараллеливания программы, который предоставляет API OpenMP:

```
1. #pragma omp for [модификаторы, ...]
2. // или комбинированная с parallel версия:
3. #pragma omp parallel for [модификаторы, ...]
```

Прикреплен к следующему за ним циклу ***for***.

В спецификации описаны требования к прикрепленному циклу. Грубо говоря: требуется стандартный цикл по переменной интегрального типа.

Основным модификатором для данной директивы является ***schedule(****type****,*** *chunk-size****)*** – он определяет распределение итераций между потоками команды. Представим 64 итерации на 4 потока - посмотрим на примеры:

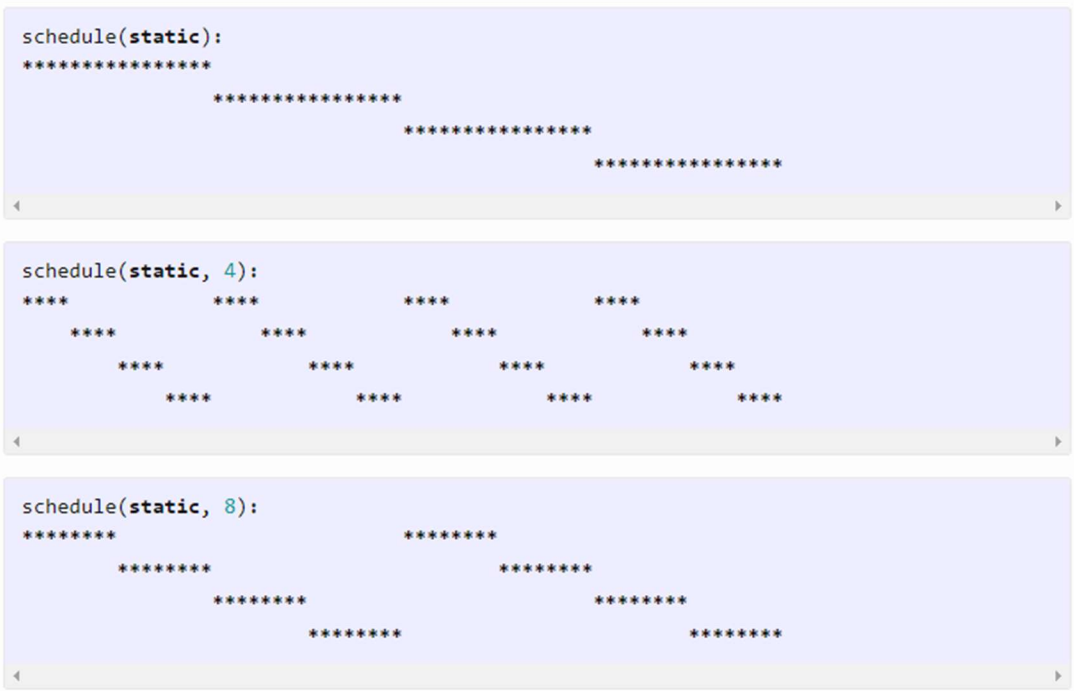

Рисунок 1 – schedule(static, n) пример.

Как видно из примера, работа честно делится на одинаковые части и отдается каждому отдельному потоку. По завершению своей части поток ждет остальных, и процесс повторяется, пока вся работа не будет завершена.

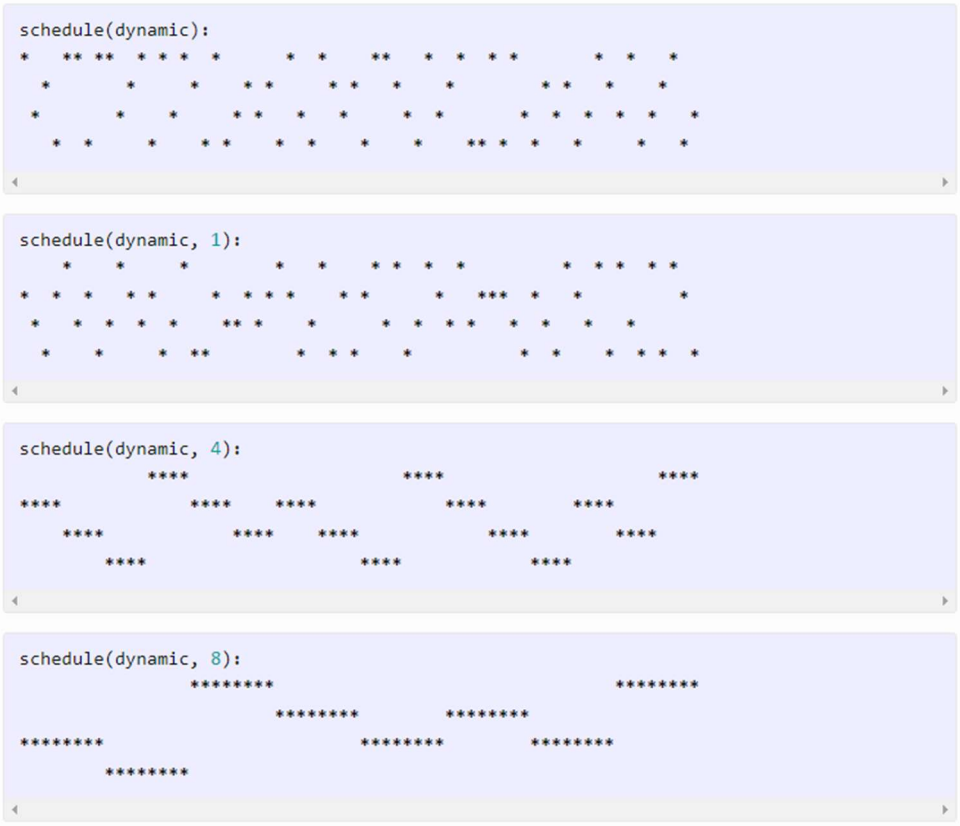

Рисунок 2 – schedule(dynamic, n?) пример.

В случае динамического распределения работа также делится на части, однако по завершению своей части – поток сразу потребует новую часть. Первый график в этом примере показывает, что динамически части могут не только распределятся, но и делится.

1.3.2. Omp single

Директива, которая дает выполнить регион только одному потоку из команды (необязательно master-thread’у), остальные его проигнорируют:

```
1. #pragma omp single [модификаторы, ...]
```

Удобно, если требуется единоразово изменить глобальное значение на основе вычислений в параллельном регионе.

*Директива **omp sections** была пропущена, так как не проявила себя в решении приложенной задачи (п. 2). Затраты на оркестровку потоков оказались больше, чем на требуемые задачи.*

1.4. Мастер и синхронизирующие директивы

1.4.1. Omp critical

Директива не прерывающая параллельность региона, но с ограничением - «один поток за раз»:

```
1. #pragma omp critical (тег)
2. // все critical с одинаковым тегом выполняются последовательно
```

Очень полезная директива при вычислении различных функций на массивах данных. Позволяет избегать состояния гонки при одновременном изменении одного участка памяти, однако за цену последовательного выполнения региона всей командой.

*Существует директива **omp atomic**, конкретнее - ее модификатор **update**, который гарантирует последовательное обновление переменной в прикрепленной к ней выражении.*

*Как оказалось, использование **omp atomic** негативно отразилось на скорости вычисления данных функций (п. 2).*

«Главный козырь» ***critical*** в данном вопросе – это, возможность прикрепить к нему регион-блок и возможность объединять локальные результаты всех потоков в один - финальный.

1.4.2. Omp barrier

Явный ***barrier*** синхронизирует ход выполнения региона командой в месте, где была объявлена директива:

```
1. #pragma omp barrier
```

После потоки из команды продолжают исполнять регион параллельно.

1.5. Остальные глобальные функции

omp_get_max_threads() – возвращает максимальное количество потоков доступное у вычислительной системы.

omp_get_num_threads() – возвращает количество используемых в данном спектре программы потоков. (за параллельными регионами – 1)

omp_get_wtime() – возвращает количество времени прошедшее «с некого момента в прошлом». Начальный момент точно не задан, однако гарантируется, что разница между вызовами функции будет равна количеству секунд прошедшее между вызовами.

1.6. Замечания

По умолчанию все переменные, лежащие во внешнем спектре региона, будут **глобальными –** эквивалент ***default(shared)***.

Локальные переменные для каждого потока были объявлены внутри регионов, поэтому модификатор ***private(****vars, ...****) и сродные с ним*** были опущены и пропущены.

2. Описание работы написанного кода

Выполнение начинается в функции ***main***, где аргументы и их количество сразу передается конструктору структуры ***Config***.

> ***Config*** *– это, структура хранящая общие параметры программы (размер команды, состояние OpenMP, путь к входной картинке и выходной). Занимается их обработкой и валидацией. Также настраивает окружение OpenMP.*

После считывается входная картинка через структуру **PGM**. При считывании используется комбинация операторов ***(>>)*** и функций «чистого» чтения ***(read())***, для параметров и пикселей соответственно.

> ***PGM*** *– это, структура хранящая размеры и массив пикселей соответствующей картинки. При считывании происходит сравнение входных данных с константами* ***(kDepth, kMagic)****. Объявлен тип для гистограммы.*

Для определения состояния ***OpenMP*** во всей программе – я решил использовать глобальную переменную ***omp_global_state***. Её значение явно определяется в основной функции, а используется она везде, поэтому считаю, что ее использование рационально.

После считывания записывается начальное время исполнения ***start***.

Следующая строчка вызывает метод подсчета гистограммы ***PGM::GetHistogram()***.

Гистограмма содержит 255 столбиков, где каждый хранит количество пикселей оттенка соответствующего индекса в картинке.

Данная функция работает параллельно, однако вместо ***omp atomic*** можно разделить работу и локально для каждого потока посчитать столбики, а потом через директиву ***critical*** результаты объединить в один массив, который и будет результатом этой функции.

От значения данной функции вызывается функция ***otsu::GetThresholds()***.

Эта функция в свою очередь вызывает функцию ***otsu::Precalcstats()***, которая делает вычисляет префиксные суммы для внутри классовых значений суммы и вероятности (п. 5.3., п. 5.4.).

Использование ***schedule(dynamic)*** невозможно в данной функции из-за особенностей префиксных сумм.

Затем перебираются все значения порог, чтобы найти максимальную межклассовую дисперсию! Работает аналогично гистограмме, только вместо нахождения локальной суммы – мы находим локальные потокам максимумы, а затем выбираем максимум между ними.

Дисперсия считается в функции ***Variance***, в ней сначала высчитываются внутриклассовые значения ***P, m (ClassStats –*** *это, их пара)*. А затем по формулам (п. 5.3.) – и сама дисперсия.

Параллельно вычислять внутриклассовые значения оказалось не эффективно (п 1.3.2. sections).

После нахождения пороговых значений, изображение преобразуется ***(PGM::Transform())***. Регион распараллеливается просто, так как здесь каждый пиксель обрабатывается изолированно, и состояния гонки не возникает.

Затем мы фиксируем конечное время исполнения алгоритма в ***end*** и выводим найденные значения, количество потоков и разницу между ***end*** и ***begin*** в миллисекундах.

В конце концов выводим готовую картинку аналогично вводу, но наоборот.

3. Результат работы написанной программы

Конфигурация окружения:

1. Процессор:
   AMD Ryzen 5 4600H - 6 ядер 12 потоков @ 3.0 ГГц

2. Компиляция:

```
1. clang++ -Ofast -std=c++20 -fopenmp=libomp -o hard.exe
   hard.cpp
```

3. Максимальная производительность, подключено питание, запуск в ***realtime***.

Результат:

```
1. 78 131 188
2. Time (12 thread(s)): 7.1491 ms
```

Рисунок 3. – Результат работы программы.

4. Эксперименты

Все тесты были выполнены через ***python скрипт***, который осуществлял редактирования кода под нужный тест, компиляцию и запуск.

Все значения времени – среднее арифметическое среди:

- 100+ запусков - ***schedule(****static/dynamic****) – n*** потоков.
- 100+ запусков - ***schedule(****static/dynamic, [0, 50, 100, 200]****) – n*** потоков.
- 500+ запусков - ***OpenMP*** выключен/***OpenMP*** с одним потоком.

4.1. AMD Ryzen 5 4600H - 6 ядер 12 потоков @ 3.0 ГГц

- Конфигурация аналогична (п. 3).
- Windows 11.

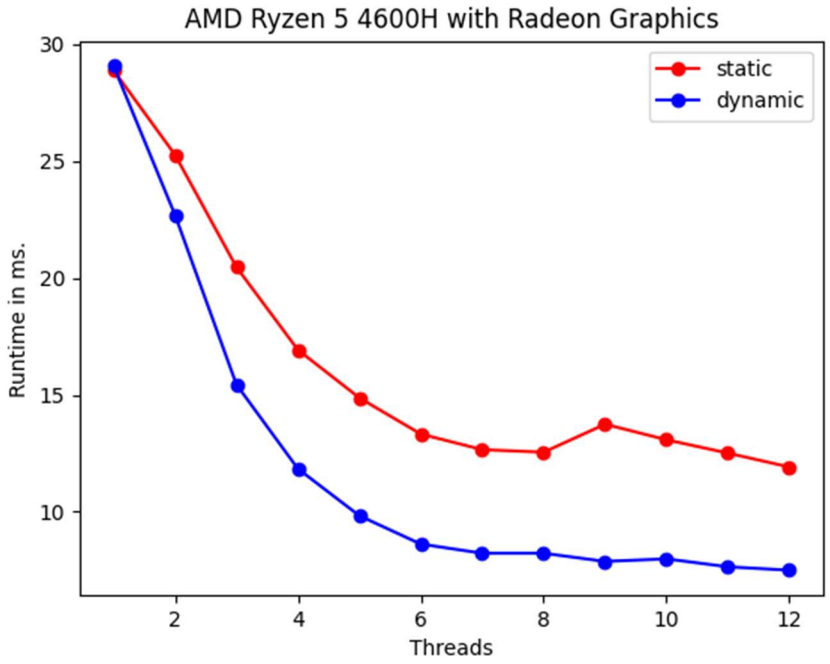

График 1. - ***schedule(****static/dynamic****) – n*** потоков.

С увеличением потоков видна тенденция к уменьшению времени выполнения. Можно заметить небольшое выгибание графика при

переходе на два потока – запомним это. Динамический режим дает большую производительность.

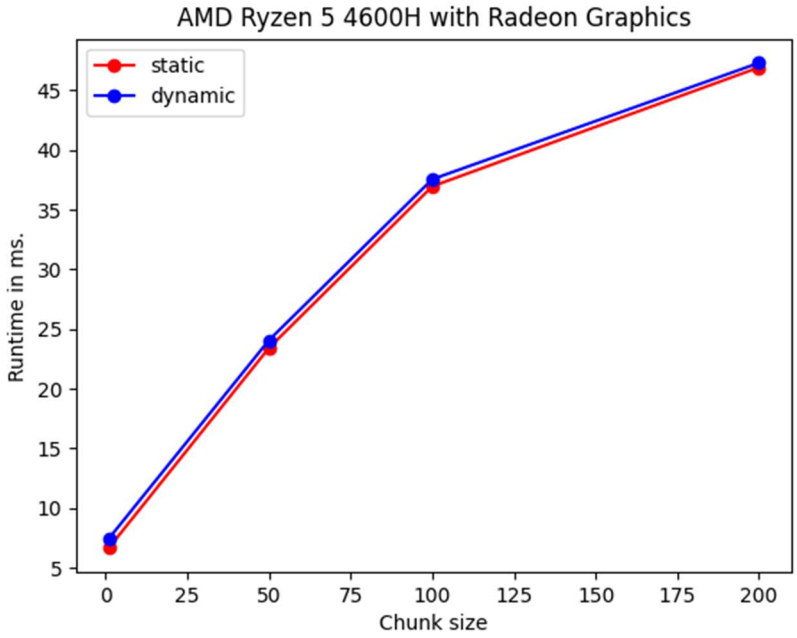

График 2. – ***schedule(****static/dynamic, [0, 50, 100, 200]****) – 12*** потоков.

Большое значение размера частей негативно влияет на время, так как большинство потоков в таком случае остаются не задействованными.

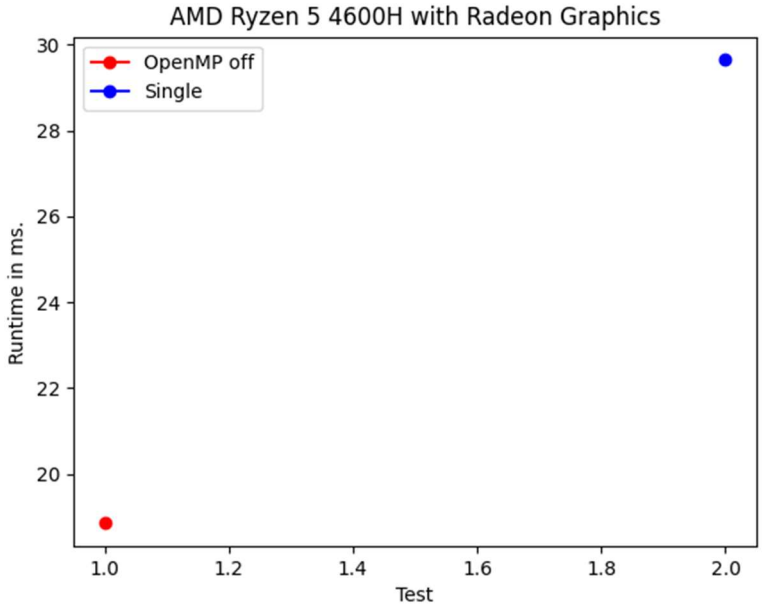

График 3. – ***OpenMP*** выключен/***OpenMP*** с одним потоком.

4.2. Intel Core i5-7500 – 4 ядра 4 потока @ 3.4 ГГц

- Сборка аналогична (п. 3).
- Параллельно шла демонстрация экрана.
- Windows 10.

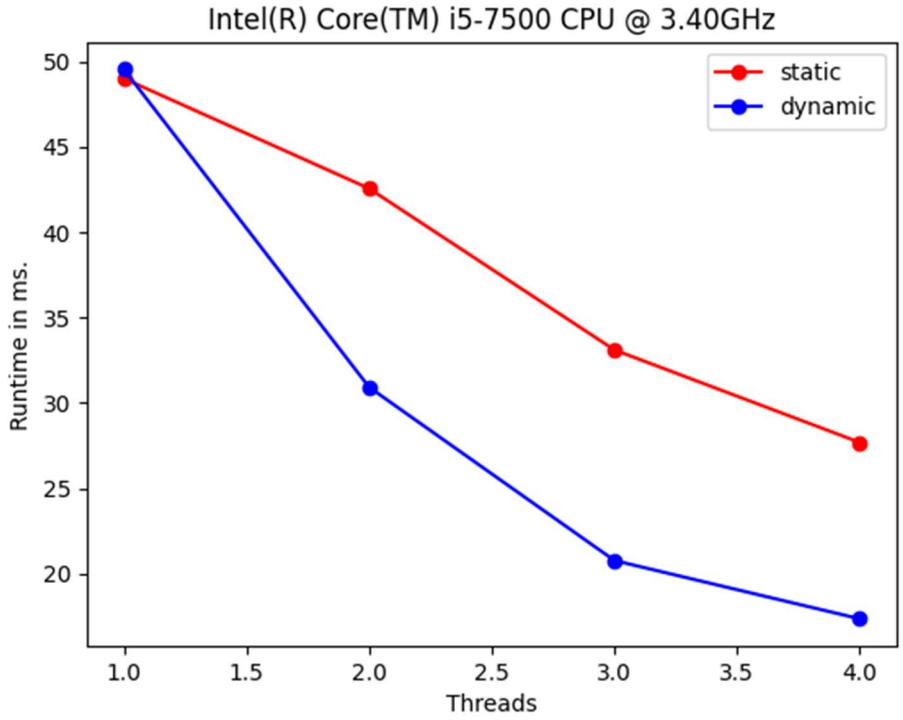

График 4. – ***schedule(****static/dynamic****) – n*** потоков.

Аналогично предыдущему процессору. Однако в этот раз выгибание на двух потоках меньше, возможно это связано с одно-поточными ядрами.

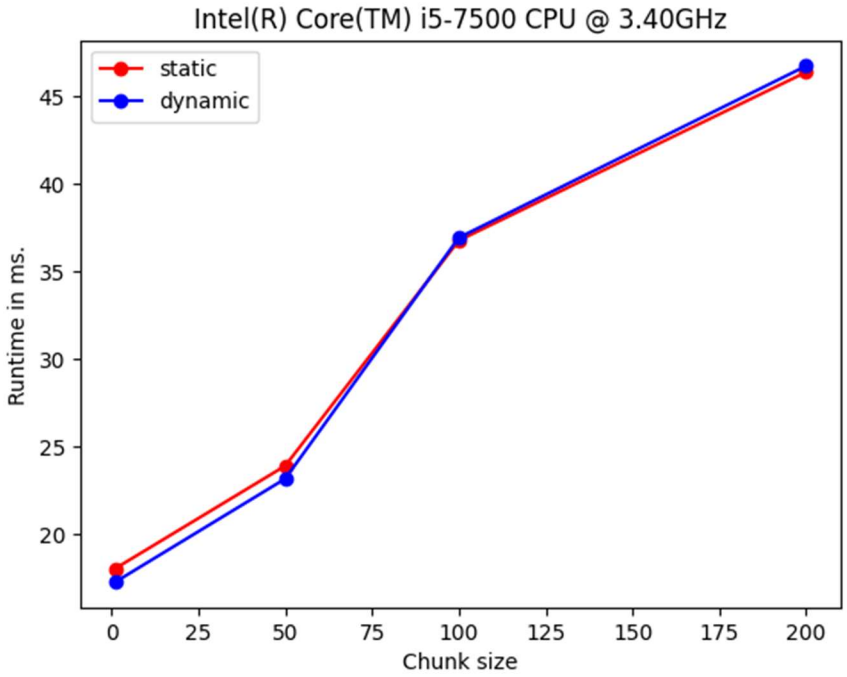

График 5. - ***schedule(****static/dynamic, [0, 50, 100, 200]****) – 4*** потока.

Аналогично предыдущему, однако из-за небольшого количества потоков график вначале растет не с такой большой скоростью.

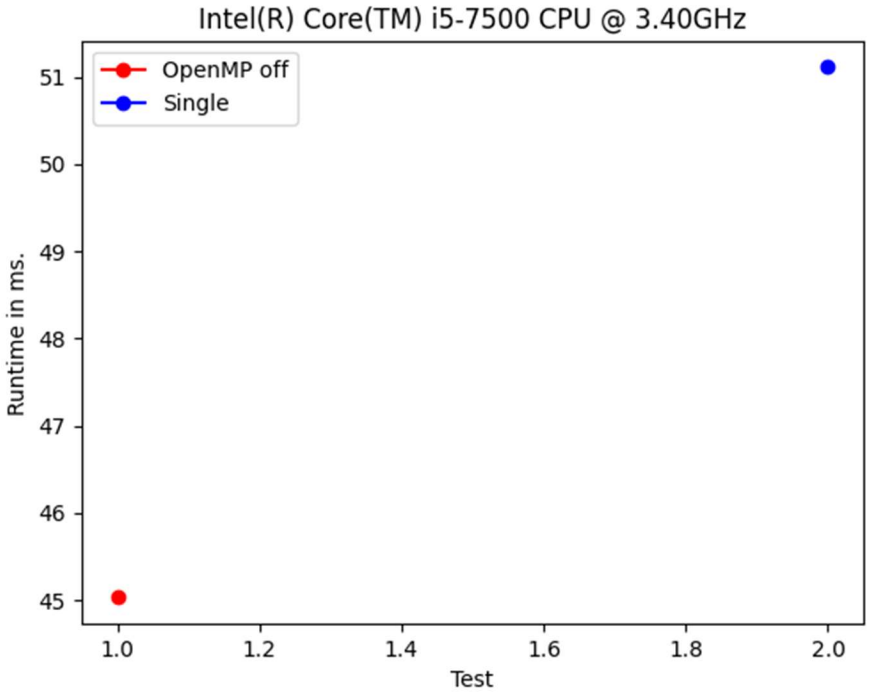

График 6. – ***OpenMP*** выключен/***OpenMP*** с одним потоком.

4.3. AMD Ryzen 7 5800H – 8 ядер 16 потоков @ 3.2 ГГц

- Сборка аналогична (п. 3).
- Параллельно был запущен CLion.
- Windows 11.

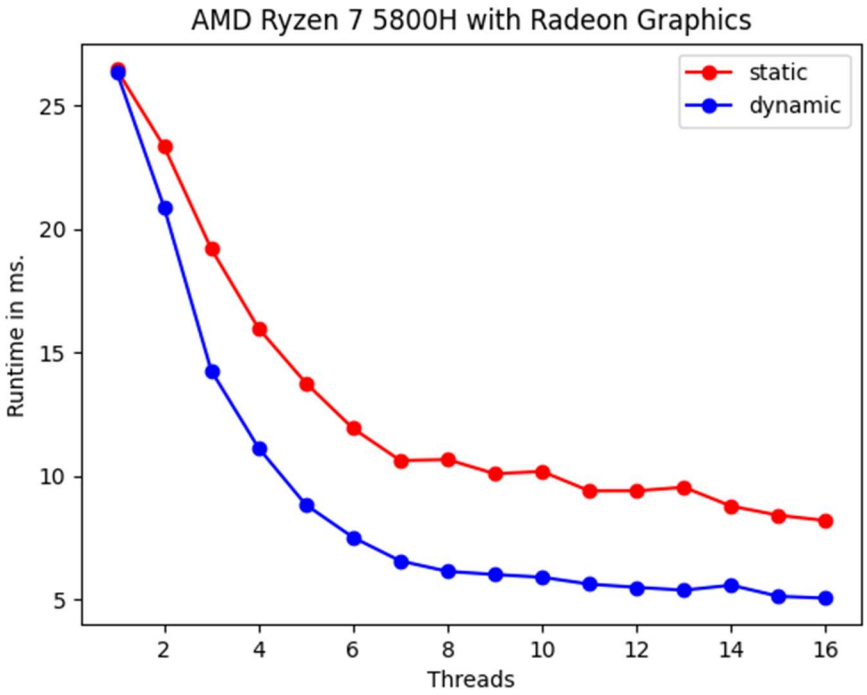

График 7. - ***schedule(****static/dynamic****) – n*** потоков.

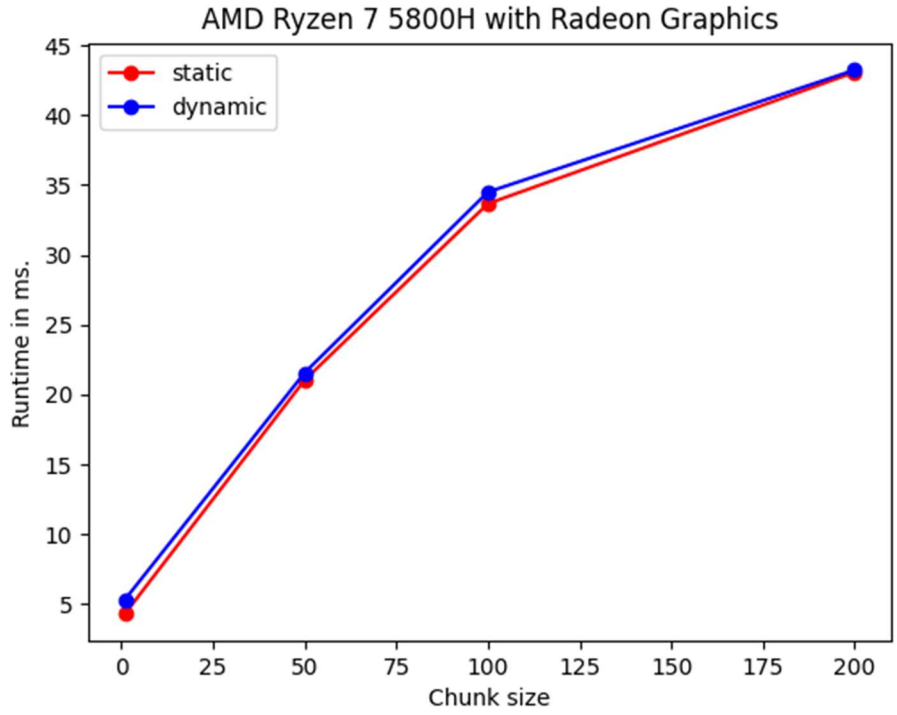

График 8. - ***schedule(****static/dynamic, [0, 50, 100, 200]****) – 16*** потоков.

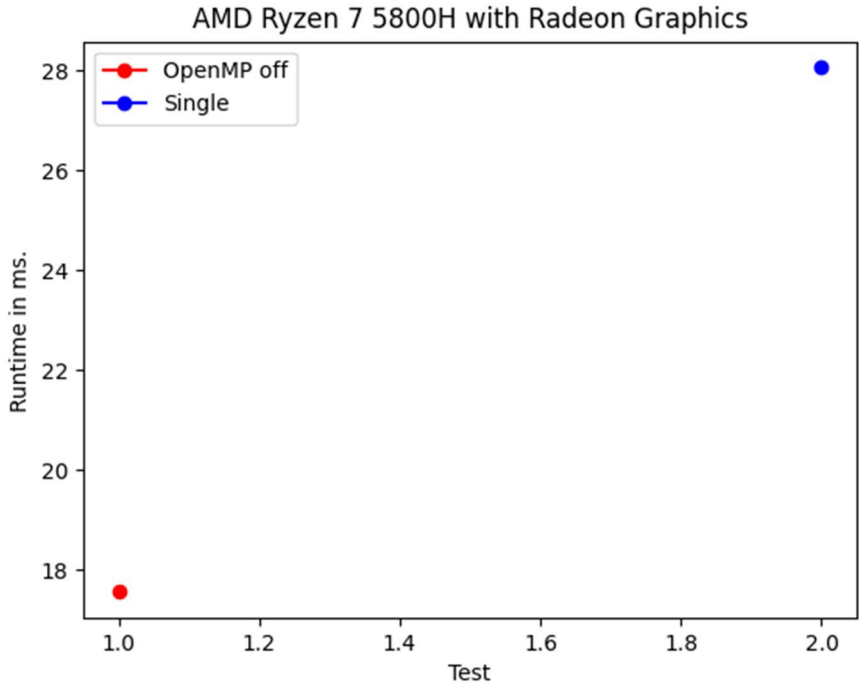

График 9. – ***OpenMP*** выключен/***OpenMP*** с одним потоком.

4.4. Intel Xeon Gold 6338 – 32 ядра 64 потока @ 2.0 ГГц

- Сборка аналогична (п. 3).
- Выполнялось на чистой виртуальной машине.
- Под ее работу было выделено 32 потока.
- Fedora 35.

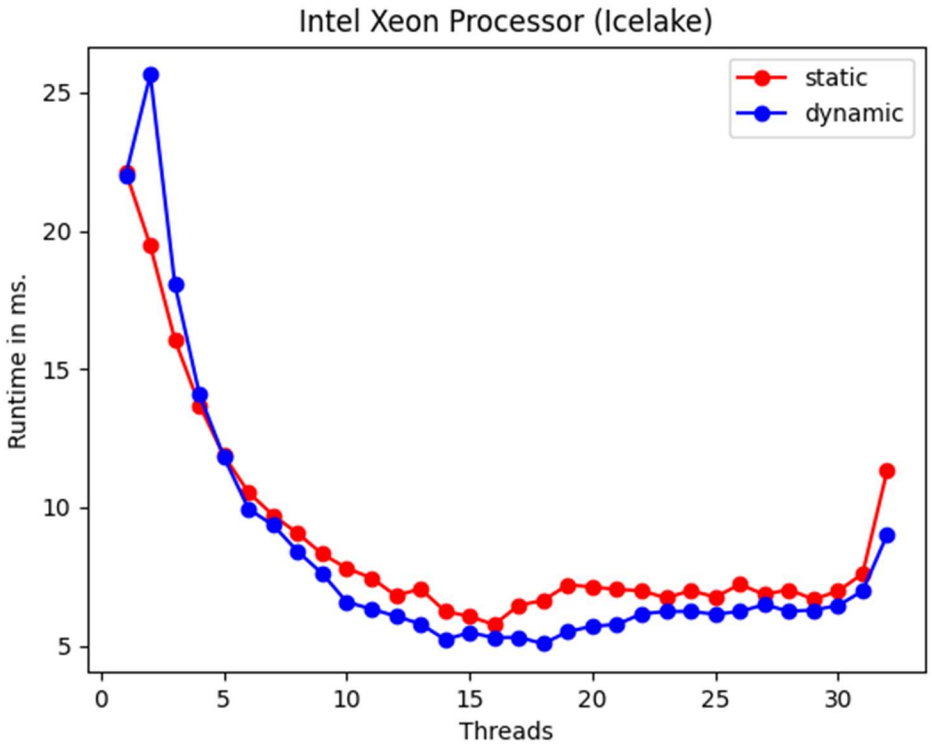

График 10. - ***schedule(****static/dynamic****) – n*** потоков.

Один из самых интересных для рассмотрения график. Сильный выгиб на двух потоках при динамическом распределении. Можно предположить, что динамическая оркестровка потоков тратит больше ресурсов, чем можно получить от нее выгоду.

Также видно, что после 18 потоков график начинает расти, так как эффективность распараллеливания небольших массивов (256) начинает падать – аналогично с графиками 2 и 5.

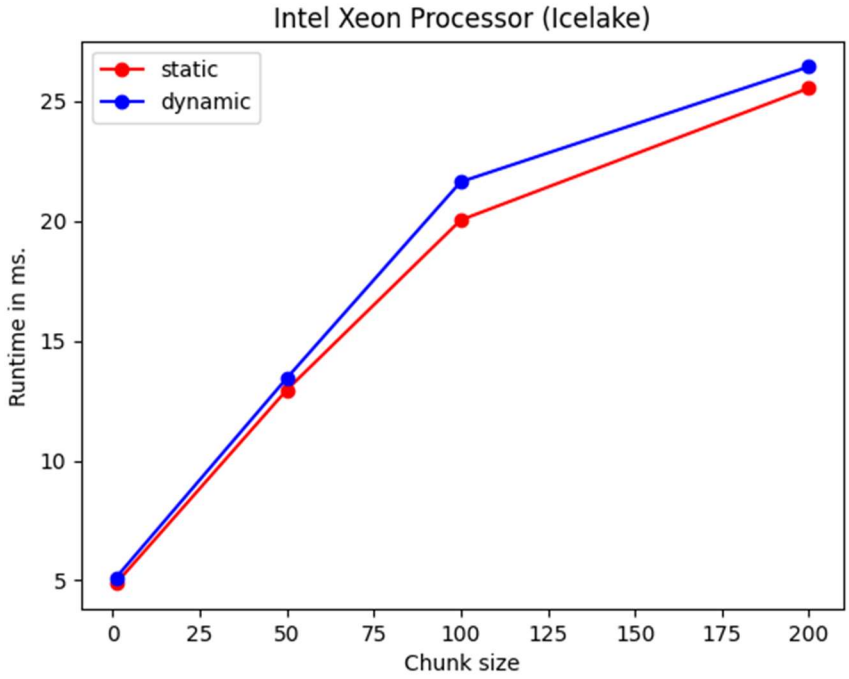

График 11. – ***schedule(****static/dynamic, [0, 50, 100, 200]****) – 18*** потоков.

График очень быстро начинает рост, но после начинает выравниваться. Причем если до этого графики были близки к друг другу, здесь статическая версия работает быстрее. Ресурсов на статическое распределение уходит меньше, а выигрыш возникает в операциях на большом размере данных.

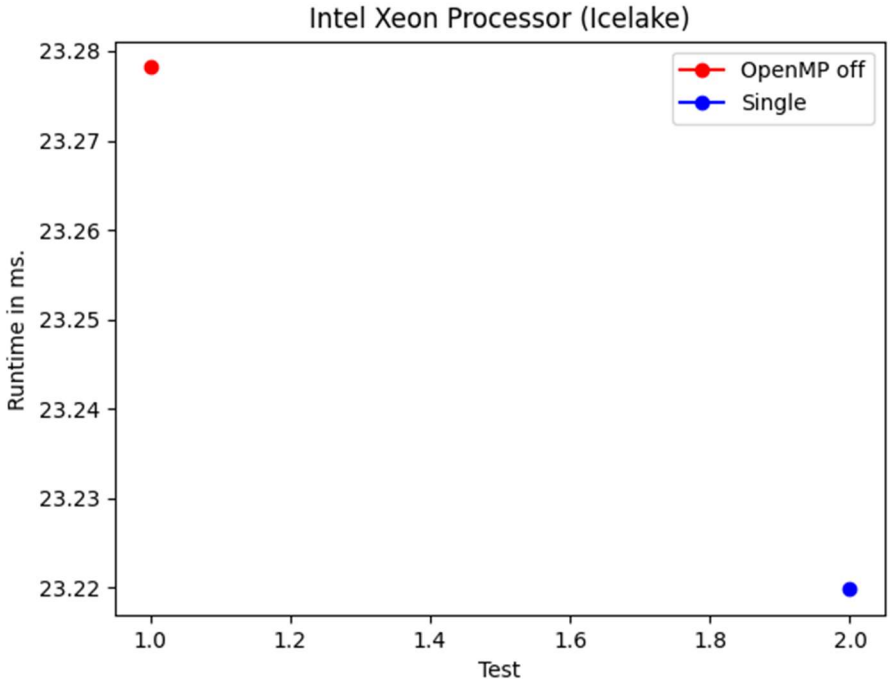

График 12. – ***OpenMP*** выключен/***OpenMP*** с одним потоком.

Это выглядит корректно, так как включённый ***OpenMP*** обязательно навешивает на регионы дополнительную нагрузку. Большая часть которой приходится на синхронизацию потоков. В остальных случаях, я полагаю возникла погрешность в то время, как этот запуск был на чистой ОС.

5. Список источников

1. Спецификация OpenMP - https://www.openmp.org/wp-content/uploads/cspec20.pdf
2. For, schedule() и графики - http://jakascorner.com/blog/2016/06/omp-for-scheduling.html
3. Otsu! Китайцы - https://drive.google.com/file/d/15osD8y6aRGiaAccpddzhTzcvfe6zrSC3/view?usp=sharing
4. Параллельные префиксные суммы - https://stackoverflow.com/questions/35821844/parallelization-of-a-prefix-sum-openmp

6. Листинг кода

/bin/main.cpp

```cpp
#include <omp.h>

#include <array>
#include <cmath>
#include <cstdint>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

namespace fs = std::filesystem;

bool omp_global_state = true;

namespace otsu {
using Thresholds = std::array<uint16_t, 3>;
using ClassStats = std::pair<uint32_t, double>;

inline double sqr(const double x) {
    return x * x;
}

struct PGM {
    static constexpr uint16_t kDepth = 256;
    static constexpr std::string_view kMagic = "P5";
    static constexpr std::array<uint8_t, 4> kTargets = {0, 84, 170, 255};

    using Histogram = std::array<uint32_t, kDepth>;

    uint32_t width{};
    uint32_t height{};
    std::vector<uint8_t> pixels;

    explicit PGM(const fs::path& path) {
        std::ifstream in(path, std::ios::in | std::ios::binary);
        in.exceptions(std::ios::failbit);

        if (std::string line; !(in >> line) || line != kMagic) {
```

```
40.             throw std::invalid_argument("PGM::PGM error: Not a PGM file was
    provided.");
41.         }
42.
43.         in >> width >> height;
44.
45.         pixels.resize(static_cast<size_t>(width) * height, 0);
46.
47.         uint16_t depth;
48.
49.         in >> depth;
50.
51.         in.ignore(1);
52.
53.         if (depth != kDepth - 1) {
54.             throw std::invalid_argument("PGM::PGM error: Not a PGM file was
    provided.");
55.         }
56.
57.         in.read(reinterpret_cast<char*>(pixels.data()),
    static_cast<std::streamsize>(width) * height);
58.     }
59.
60.     [[nodiscard]] Histogram GetHistogram() const {
61.         Histogram result{0};
62.
63. #pragma omp parallel if (omp_global_state)
64.         {
65.             Histogram local{0};
66.
67. #pragma omp for schedule(dynamic)
68.             for (size_t i = 0; i < pixels.size(); ++i) {
69.                 ++local[pixels[i]];
70.             }
71.
72. #pragma omp critical
73.             {
74.                 for (uint16_t i = 0; i < local.size(); ++i) {
75.                     result[i] += local[i];
76.                 }
77.             }
78.         }
79.
80.         return result;
```

```
    }

    void Transform(const Thresholds& thresholds) {
#pragma omp parallel for schedule(dynamic) if (omp_global_state)
        for (size_t i = 0; i < pixels.size(); ++i) {
            if (pixels[i] <= thresholds[0]) {
                pixels[i] = kTargets[0];
            } else if (pixels[i] <= thresholds[1]) {
                pixels[i] = kTargets[1];
            } else if (pixels[i] <= thresholds[2]) {
                pixels[i] = kTargets[2];
            } else {
                pixels[i] = kTargets[3];
            }
        }
    }

    void Print(const fs::path& path) const {
        std::ofstream out(path, std::ios::out | std::ios::binary | std::ios::trunc);
            out.exceptions(std::ios::failbit);

            out << kMagic << '\n';
            out << width << ' ' << height << '\n';
            out << kDepth - 1 << '\n';

            out.write(reinterpret_cast<const char*>(pixels.data()), static_cast<std::streamsize>(width) * height);
        }
    };

    using HistogramStats = std::array<ClassStats, PGM::kDepth>;

    inline ClassStats GetClassStats(const HistogramStats& stats, const uint16_t begin, const uint16_t end) {
        if (begin == 0) {
            return {stats[end].first, stats[end].second / static_cast<double>(stats[end].first)};
        }

        size_t sum = stats[end].first - stats[begin - 1].first;
        double prb = (stats[end].second - stats[begin - 1].second) / static_cast<double>(sum);

```

```
120.          return {sum, prb};
121.      }
122.
123.      double Variance(const HistogramStats& stats, const uint16_t k0, const
   uint16_t k1, const uint16_t k2) {
124.          const ClassStats cs0 = GetClassStats(stats, 0, k0 - 1);
125.
126.          const ClassStats cs1 = GetClassStats(stats, k0, k1 - 1);
127.
128.          const ClassStats cs2 = GetClassStats(stats, k1, k2 - 1);
129.
130.          const ClassStats cs3 = GetClassStats(stats, k2, stats.size() -
   1);
131.
132.          const double mg = (static_cast<double>(cs0.first) * cs0.second +
   static_cast<double>(cs1.first) * cs1.second) +
   (static_cast<double>(cs2.first) * cs2.second +
   static_cast<double>(cs3.first) * cs3.second);
133.
134.          return (static_cast<double>(cs0.first) * sqr(cs0.second - mg) +
   static_cast<double>(cs1.first) * sqr(cs1.second - mg)) +
   (static_cast<double>(cs2.first) * sqr(cs2.second - mg) +
   static_cast<double>(cs3.first) * sqr(cs3.second - mg));
135.      }
136.
137.      HistogramStats PrecalcStats(const PGM::Histogram& histogram) {
138.          HistogramStats result;
139.          std::vector<ClassStats> locals;
140.
141.      #pragma omp parallel if (omp_global_state)
142.          {
143.              const int32_t thread = omp_get_thread_num();
144.              const int32_t threads = omp_get_num_threads();
145.      #pragma omp single
146.              {
147.                  locals.resize(threads + 1);
148.                  locals[0] = {0, 0};
149.              }
150.
151.              uint32_t class_sum = 0;
152.              double class_prb = 0;
153.      #pragma omp for schedule(static) nowait
154.              for (uint16_t i = 0; i < histogram.size(); ++i) {
155.                  class_sum += histogram[i];
```

```
156.                 class_prb += static_cast<double>(i * histogram[i]);
157.
158.                 result[i] = {class_sum, class_prb};
159.             }
160.
161.             locals[thread + 1] = {class_sum, class_prb};
162.
163.     #pragma omp barrier
164.
165.             uint32_t class_sum_offset = 0;
166.             double class_prb_offset = 0;
167.             for (uint16_t i = 0; i < thread + 1; ++i) {
168.                 class_sum_offset += locals[i].first;
169.                 class_prb_offset += locals[i].second;
170.             }
171.
172.     #pragma omp for schedule(static)
173.             for (uint16_t i = 0; i < histogram.size(); ++i) {
174.                 result[i].first += class_sum_offset;
175.                 result[i].second += class_prb_offset;
176.             }
177.         }
178.
179.         return result;
180.     }
181.
182.     Thresholds GetThresholds(const PGM::Histogram& histogram) {
183.         Thresholds result;
184.         double variance = 0;
185.
186.         const HistogramStats stats = PrecalcStats(histogram);
187.
188.     #pragma omp parallel if (omp_global_state)
189.         {
190.             double local_variance = variance;
191.             Thresholds local_result{};
192.
193.     #pragma omp for schedule(dynamic) nowait
194.             for (uint16_t k0 = 1; k0 < histogram.size() - 3; ++k0) {
195.                 for (uint16_t k1 = k0 + 1; k1 < histogram.size() - 2;
    ++k1) {
196.                     for (uint16_t k2 = k1 + 1; k2 < histogram.size() - 1;
    ++k2) {
```

```
197.                         if (const double tmp = Variance(stats, k0, k1,
    k2); std::isless(
198.                                 local_variance, tmp)) {
199.                             local_variance = tmp;
200.                             local_result = {k0, k1, k2};
201.                         }
202.                     }
203.                 }
204.             }
205.
206.     #pragma omp critical
207.             {
208.                 if (std::isless(variance, local_variance)) {
209.                     variance = local_variance;
210.                     result = local_result;
211.                 }
212.             }
213.         }
214.
215.         return result;
216.     }
217.     }  // namespace otsu
218.
219.     struct Config {
220.         constexpr static int32_t kArgAmount = 4;
221.
222.         int32_t threads;
223.         bool open_mp;
224.         fs::path in_path;
225.         fs::path out_path;
226.
227.         Config(const int argc, char** argv) {
228.             if (argc < kArgAmount) {
229.                 throw std::invalid_argument("Config::Parse error: Not
    enough arguments.");
230.             }
231.
232.             if (argc > kArgAmount) {
233.                 std::cout << "Config::Parse warning: Extra arguments will
    be ignored.";
234.             }
235.
236.             SetThreads(std::stoi(argv[1]));
237.
```

```
238.             in_path = argv[2];
239.             out_path = argv[3];
240.         }
241.
242.     private:
243.         void SetThreads(const int32_t value) {
244.             if (value < 0) {
245.                 threads = 1;
246.                 open_mp = false;
247.                 omp_set_dynamic(0);
248.                 omp_set_num_threads(1);
249.             } else if (value == 0) {
250.                 open_mp = true;
251.                 threads = omp_get_max_threads();
252.             } else if (value > 0) {
253.                 open_mp = true;
254.                 threads = std::min(omp_get_max_threads(), value);
255.                 omp_set_dynamic(0);
256.                 omp_set_num_threads(threads);
257.             }
258.         }
259.     };
260.
261.     int main(const int argc, char** argv) {
262.         try {
263.             const Config config(argc, argv);
264.             otsu::PGM image{config.in_path};
265.
266.             omp_global_state = config.open_mp;
267.
268.             const double start = omp_get_wtime();
269.
270.             const otsu::Thresholds thresholds =
    otsu::GetThresholds(image.GetHistogram());
271.
272.             image.Transform(thresholds);
273.
274.             const double end = omp_get_wtime();
275.
276.             std::cout << thresholds[0] << ' ' << thresholds[1] << ' ' <<
    thresholds[2] << '\n';
277.             std::cout << "Time (" << config.threads << " thread(s)): " <<
    (end - start) * 1000 << " ms" << std::endl;
278.
```

```
            image.Print(config.out_path);
        } catch (std::exception& err) {
            std::cout << err.what() << '\n';
        }
        return 0;
    }


```