САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Белоус Данила Павлович

Номер ИСУ: 334927

студ. гр. М3139

Санкт-Петербург

2021

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: C++. Стандарт OpenMP 2.0.

Теоретическая часть

OpenMP — библиотека для удобного использования многопоточности в программах на C/C++. Многопоточность или multithreading простыми словами это способность процессора выполнять много процессов одновременно. В C/C++ есть встроенные методы для запуска многопоточных программ, но с оболочкой OpenMP это становится ощутимо проще и распараллеливание добавляется буквально в несколько строчек. Основное, для чего используется эта библиотека — это распараллеливание циклов.

При распараллеливании используется fork-join-модель (рис. 1). Главный тред последовательно выполняет команды, пока не дойдёт до региона (блока, секции кода), который нужно распараллелить. На этом моменте главный тред создает команду из нескольких тредов. Далее регион выполняется параллельно — задача делится между тредами. Когда все треды завершили свою работу, они синхронизируются и выключаются, оставляя только подсчитанный результат и главный тред. Далее ситуация повторяется.

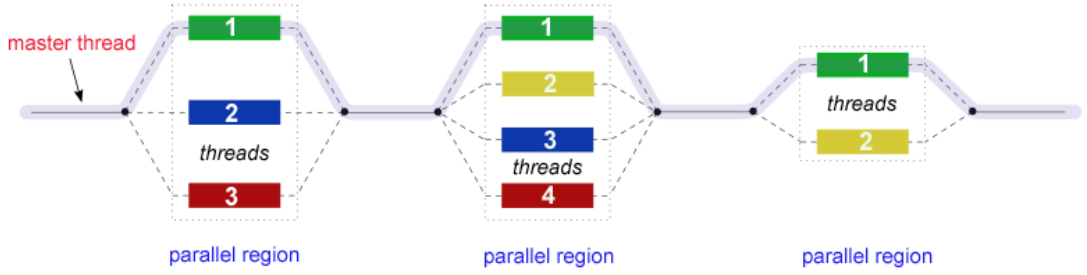

Рисунок 1 — fork-join модель.

Если один тред достиг барьера (конца параллельного региона) раньше остальных тредов, то он, как правило, ждёт (синхронизация), а потом результат со всех тредов собирается в единый. Это поведение может быть изменено с помощью соответствующих команд.

Большая часть общения с OpenMP производится с помощью директив — команд, начинающихся с `#pragma omp`.

После этого префикса обычно следует `parallel` — то, ради чего всё затевается. На этом моменте OpenMP анализирует блок кода, который следует за этой директивой, и решает, на сколько тредов и в каком объеме разделить между ними задачу, которую нужно посчитать. В этом домашнем задании нам требовалось распараллелить только циклы (для этого используется `#pragma omp for`), но ещё бывают команды, которые означают, что следующий блок кода должен выполнить только один тред — `single`, или `atomic` — когда нужно выполнить какую-то простую операцию с общей переменной, и при этом значение должно оставаться всегда корректным. Все эти команды сделаны, чтобы избежать race conditions — ситуаций, когда из-за непредсказуемого порядка выполнения команд разными тредами какие-то данные могут оказаться некорректными. Такое может произойти, потому что мы работаем в shared memory модели — когда каждый тред имеет доступ к одной общей для всех памяти.

Также для предотвращения нежелательного повреждения данных используются модификаторы доступа — `shared`, `private` и ещё несколько, которые здесь рассмотрены не будут. Модификатор `shared` сообщает, что переменная является общей для всех и каждый тред имеет к ней доступ. `private` говорит, что у каждого треда будет локальная копия этой переменной, при этом главный тред (master, тот, с которого начиналось выполнение параллельного блока) будет иметь то значение этой

переменной, которое было до входа в блок, а для остальных тредов значение не сообщается — OpenMP будет сам решать, как их проинициализировать. Модификатор `default` задаёт поведение переменных по умолчанию. `default(private)` — что будут приватными, `default(shared)` — общими, `default(none)` заставляет пользователя указывать тип для каждой локальной переменной отдельно.

Также есть параметр `schedule(type[, chunkSize])`, который отвечает за то, какой объём работы получит каждый из тредов. Типом может быть статический `static` — в этом случае объем работы распределяется заранее и обычно приблизительно равномерно между тредами. Этот параметр лучше применять, когда выполнение задачи внутри цикла занимает всегда приблизительно одно и то же время, когда нет непредсказуемых задержек. Динамический `dynamic` распределяет нагрузку по мере освобождения тредов — когда какой-то из тредов закончил свою работу и простаивает, ему выдают следующую порцию. Размер порции определяется параметром `chunkSize`. Если этот параметр не указывать, то для `dynamic` он обычно равен 1, а для `static` — количеству процессоров. Более подробное и наглядное объяснение работы параметра `schedule` есть здесь.

Также для получения или задания каких-то параметров подключается заголовочный файл `omp.h`, который позволяет получать количество тредов (`omp_get_num_threads()`), номер текущего треда, который выполняет эту команду (`omp_get_thread_num()`) и ещё некоторые команды, которые не использовались при выполнении ДЗ, а потому рассмотрены здесь не будут.

Ссылки:

- Спецификация OpenMP 2.0
- Спецификация OpenMP 5.2
- Полезный листочек с выжимкой из спецификации
- Гайд 1
- Гайд 2
- Гайд 3

Теперь немного поговорим про контрастность. Контрастность — это разность в освещении или цвете, которая делает объект различимым на фоне остальных объектов в этом же поле зрения. На рисунке 2, левая картинка имеет более низкую контрастность, потому что объекты на ней труднее различить.

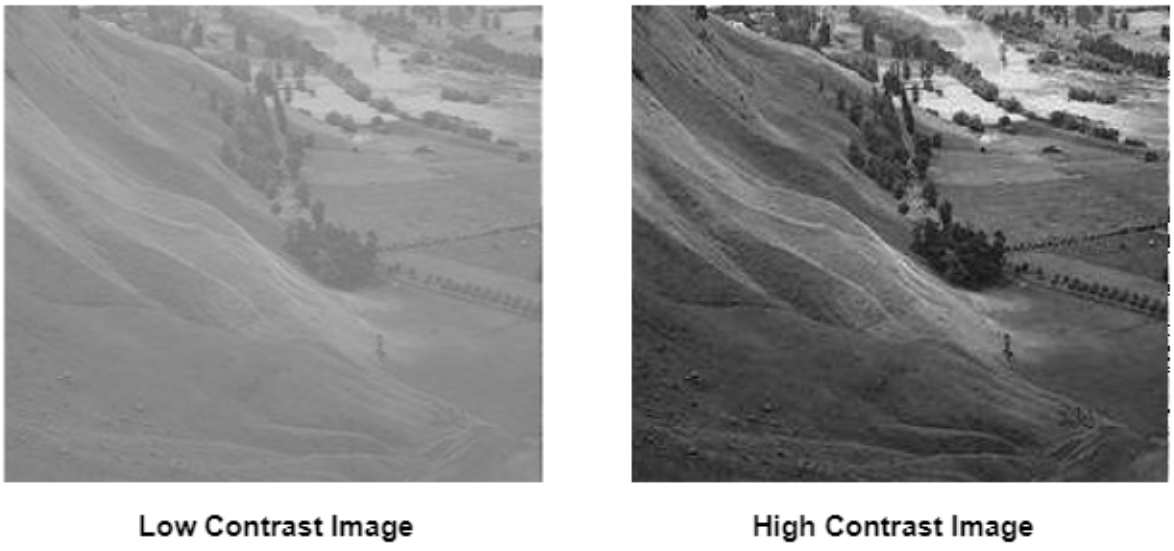

Рисунок 2 — Пример фото с низкой (слева) и высокой (справа) контрастностью.

Примером из жизни может служить туманный день (низкая контрастность) и ясный солнечный день (высокая контрастность).

Более правильным подходом для определения контрастности будет проанализировать гистограммы распределения частот цвета на картинке. Рассмотрим гистограммы для фотографий выше:

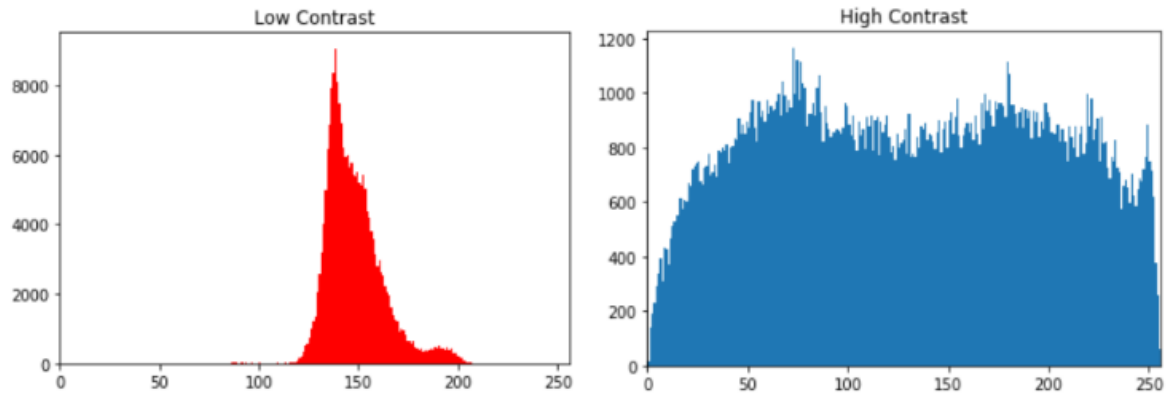

Рисунок 3 — гистограммы распределения цвета.

Из левой гистограммы можно видеть, что значения (цвета) пикселей сконцентрированны в узком диапазоне, и из-за этого их труднее различить. Действительно, взглянем на сопоставление цвета и значения:

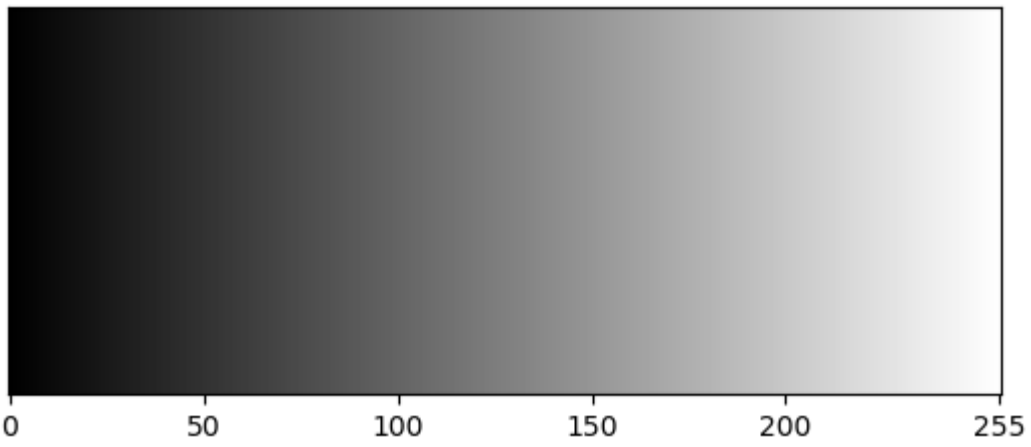

Рисунок 4 — сопоставление цвета и значения оттенков серого.

Очевидно, оттенок 148 от оттенка 150 отличить сложнее, чем 50 от 200. Поэтому чем более равномерно распределены частоты по гистограмме — тем лучше контрастность. На рисунке 3 видно, что для фотографии 2 значения пикселей распределены более равномерно по всему диапазону от 0 до 255, и поэтому предметы на фотографии более различимы.

Как же увеличить контрастность? Для этого мы буквально растянем гистограмму с цветами:

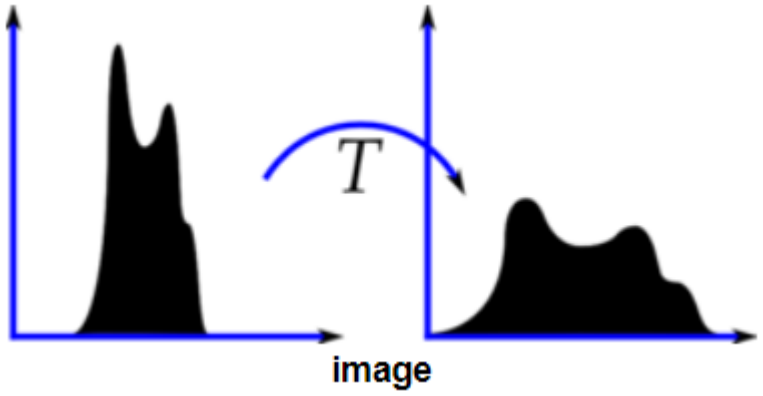

Рисунок 5 — растяжение гистограммы.

При этом из-за растяжения и более равномерного распределения какие-то конкретные значения пикселей могут оказаться незаполненными:

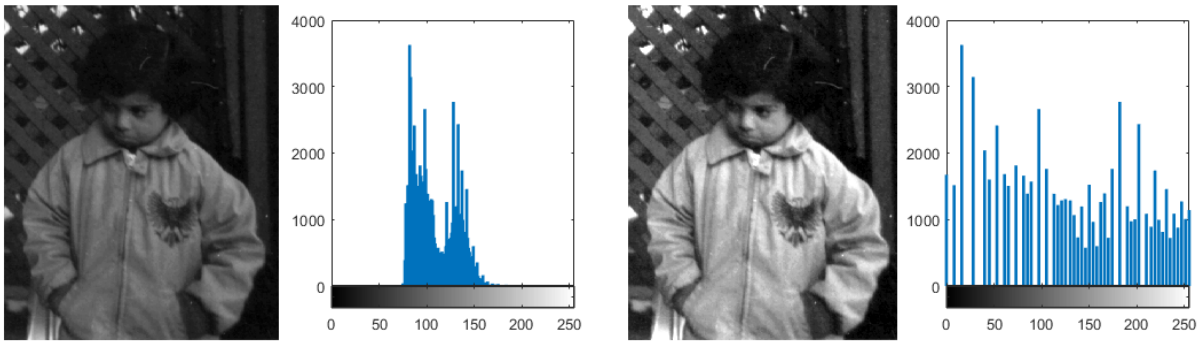

Рисунок 6 — пример отсутствия некоторых цветов в гистограмме.

Как растянуть гистограмму? Возьмём минимальное значение, которое в ней присутствует, и скажем, что это новый 0, а максимальное — новый максимум (255). Функция выглядит так:

$$X_{new} = \frac{X_{input} - X_{min}}{X_{max} - X_{min}} \times 255.$$

Пример применения подобной Min-Max функции к гистограмме показан на рисунке 6.

Но иногда в начале или в конце гистограммы есть небольшой “хвост”, в котором собрано очень маленькое количество пикселей, но которые сильно занижают X_{min} или завышают X_{max}. В этом случае min-max растяжение показывает себя немного хуже (рисунки 7 и 8).

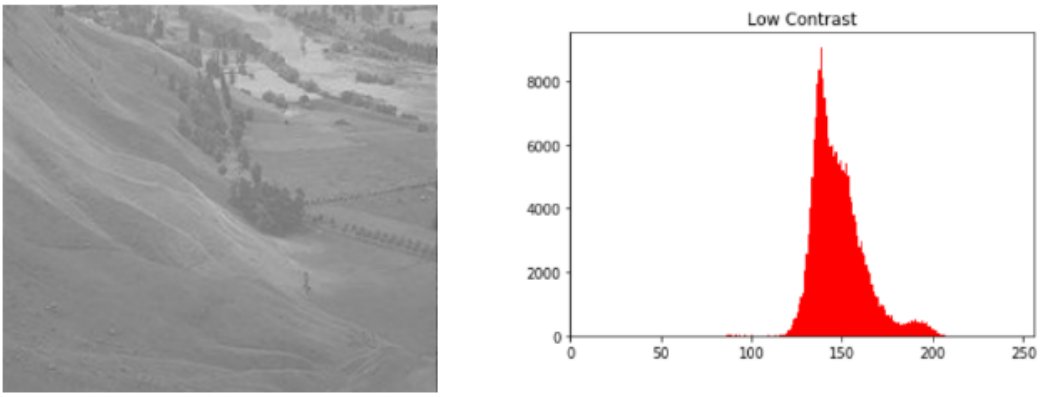

Рисунок 7 — гистограмма изображения с низкой контрастностью.

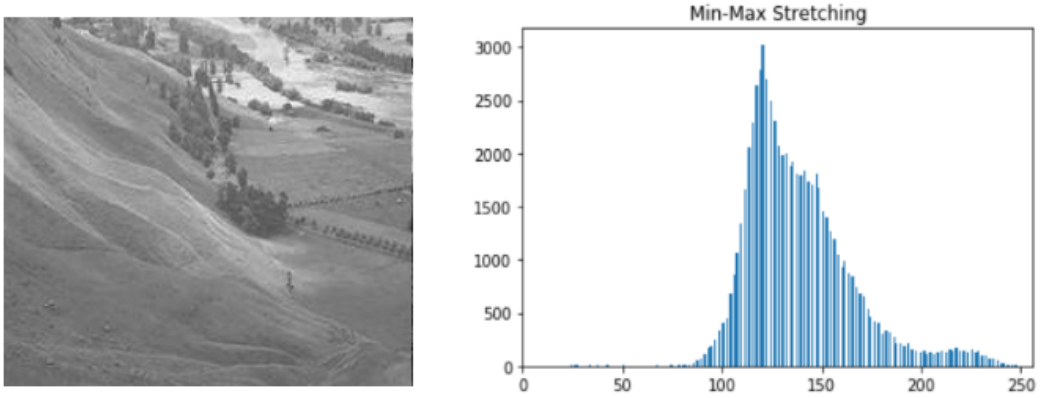

Рисунок 8 — применение Min-Max растяжения к гистограмме.

Чтобы бороться с такими ситуациями, можно “обрезать” такие “хвосты”: игнорировать 2-5% пикселей с самыми низкими и пикселей с самыми высокими значениями. Тогда результат получается лучше:

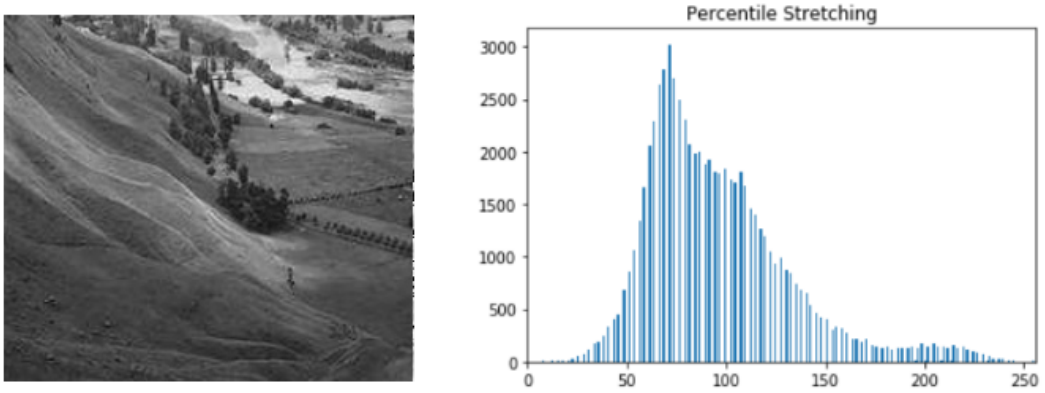

Рисунок 9 — Min-Max растяжение, но игнорируем крайние значения.

С цветными RGB картинками всё просто — как указано в ДЗ, новые крайние значения считаем отдельно по каналам, затем для нового общего минимума берём минимальное из них, а для максимума — максимальное, чтобы при растяжении не потерять информацию в других каналах.

Ссылки:

- О контрастности — 1 и 2
- Об увеличении контрастности — 1 и 2

Практическая часть

Для реализации алгоритма, описанного выше, использовался **C++17**, с компиляторами **GCC 11.2.0** и **Clang 13.0.0**, портированные на Windows с помощью **MSYS2**. Для сборки использовался **CMake** версии **3.21**.

Проект разделён на несколько частей: `main.cpp` считывает картинку, обрабатывает различные ошибки, связанные с параметрами запуска или со вводом-выводом, и использует внутри себя класс `Image`, в котором и происходит основное действие — подсчёт гистограммы, игнорирование “хвостов”, растяжение гистограммы. `Time` — вспомогательный класс, в котором я делаю какие-то метки, чтобы засекать время выполнения различных частей программы.

В классе `Image` функция `EnhanceGlobalContrast` применяет к изображению Min-Max растяжение с учётом процента пикселей с крайними значениями, которые нужно проигнорировать. Все пиксели ниже минимального значения становятся пикселями со значением 0, все пиксели со значениями выше максимального — 255.

Про ограничения: для хранения пикселей используется `uint8_t`, для хранения размера картинки (ширина * высота * количество каналов) — `uint32_t`. Поэтому если будет использоваться больше, чем 8 бит на пиксель, либо на вход подадут картинку размером как 44 8K фотографии (примерно 1,4 ГБ), то программа выдаст ошибку.

Про применение OpenMP. Во имя скорости работы и хорошего распараллеливания пришлось пожертвовать удобством и красивым кодом. Размерность массива сведена и количество циклов сведено к минимуму. Многопоточность используется при применении Min-Max растяжения и при подсчёте гистограммы. Для растяжения `#pragma omp parallel for`

`shared(ignore) schedule (static) default(none)`. Процент игнорирования, понятно, нет смысла копировать каждому треду, а массив с изображением и прочими полями класса и так является общим. Статический тип планировки используется, потому что он лучше всего показал себя в тестах — об этом позже. `default(none)` — чтобы случайно не набагать и явно указывать типы локальных переменных.

При подсчёте, сколько пикселей из “хвоста” нужно убрать, использование OpenMP невозможно, да и не имеет смысла — идти по пикселям нужно подряд до определённого момента, а пикселей всего 256.

Также бывают ситуации, когда новый минимум равен новому максимуму. В этом случае я полагаю, что новое значения пикселя будет 0.

Отдельное внимание стоит уделить измерению времени работы разных частей программы. Если кратко, то существует два основных типа времени: wall time и cpu time. Wall time отражает, сколько времени прошло всего с какого-то момента, а cpu time — сколько времени процессор реально считал нашу задачу, а не простаивал или выполнял что-то другое. В подробности реализации вдаваться не буду.

От нас хотят wall time, поэтому будем приводить в тестах его.

Также не очень понятно, стоит ли во время работы включать время, затрачиваемое на подсчёт гистограммы значений. В итоговую версию кода это время войдёт, потому что просили измерить всё время, кроме ввода и вывода. Однако для корректности выводов относительно работы OpenMP, для логирования и составления графиков учитывалось только время, затрачиваемое на непосредственно растяжение контрастности, игнорируя время на начальную инициализацию гистограммы и поддержание её в корректном состоянии после изменения картинки.

Для ускорения времени работы при компиляции используется флаг `-O3`. При сборке с помощью CMake использовался Release режим. При компиляции из консоли/терминала использовались следующие команды: `g++ -std=c++17 main.cpp Time.cpp Image.cpp -o hw5 -fopenmp -O3 -DNDEBUG` для gcc, и `clang++ -std=c++17 main.cpp Time.cpp Image.cpp -o hw5 -fopenmp=libomp -O3 -DNDEBUG` для Clang. Тестировалась программа на RGB картинке размером 8192x5210 пикселей (примерно 122 МБ).

Далее пойдёт секция графиков. Её удобнее просматривать в google docs, потому что каждый график прикреплён к соответствующей таблице с данными в google sheets, в которых можно потыкать на графики и посмотреть значения на конкретных точках. Архив с логами, на основе которых строились графики, я прикреплю ниже, вместе с исходниками и тестами.

Графики:

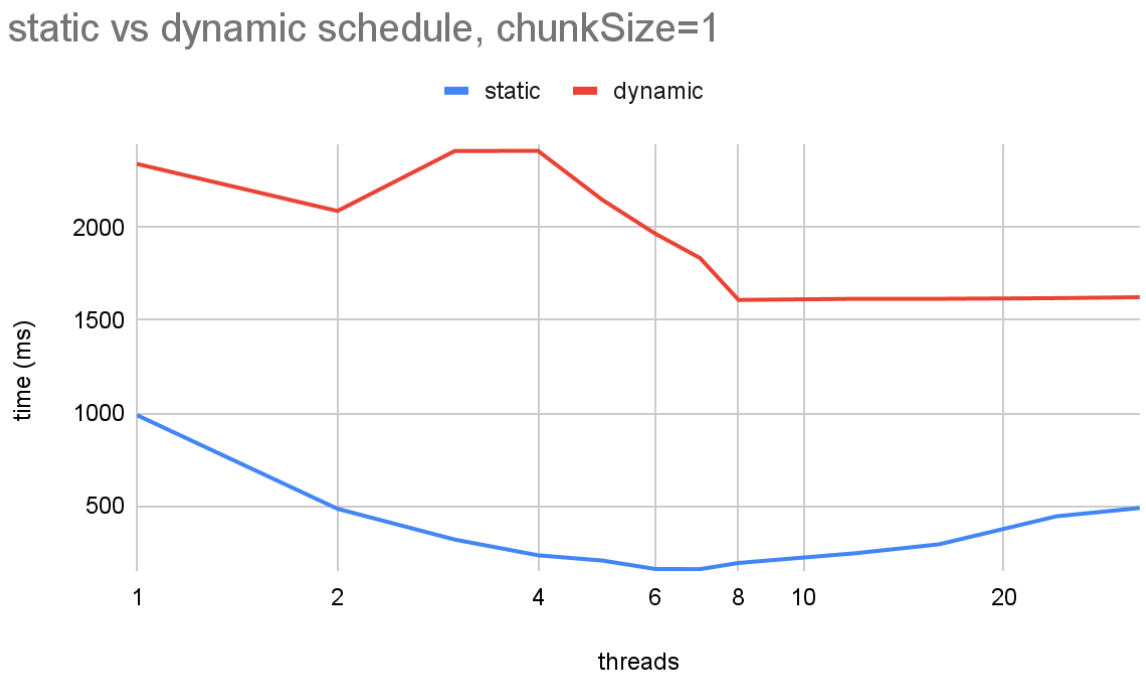

Рисунок 10 — график при параметре chunkSize = 1.

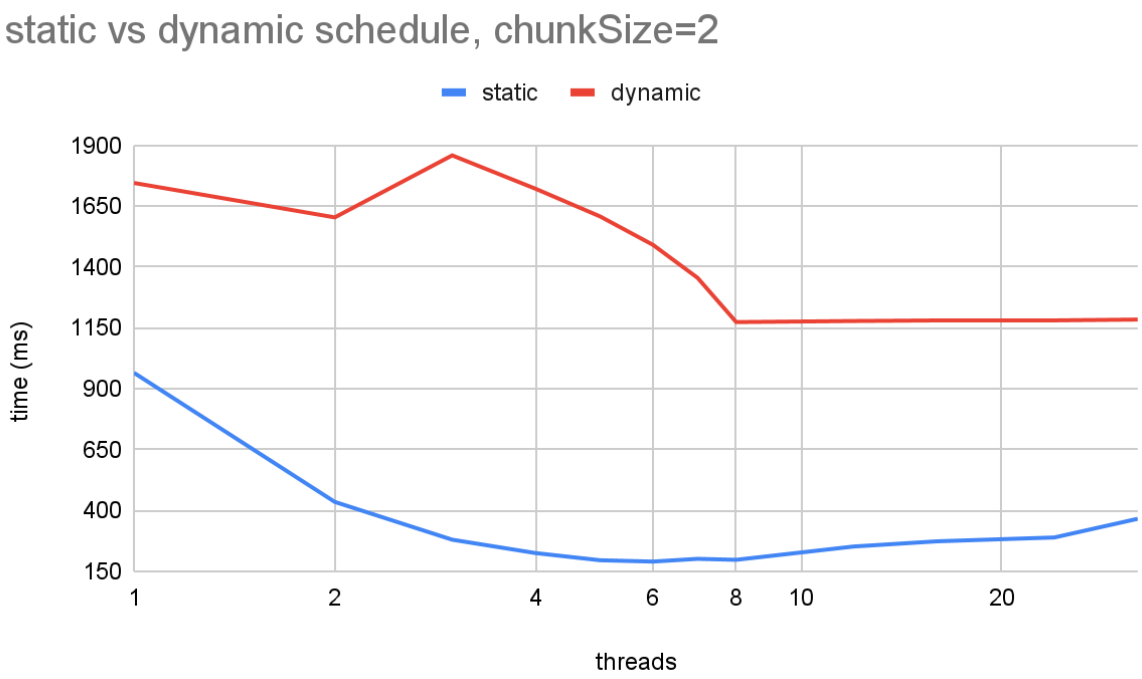

Рисунок 11 — график при параметре chunkSize = 2.

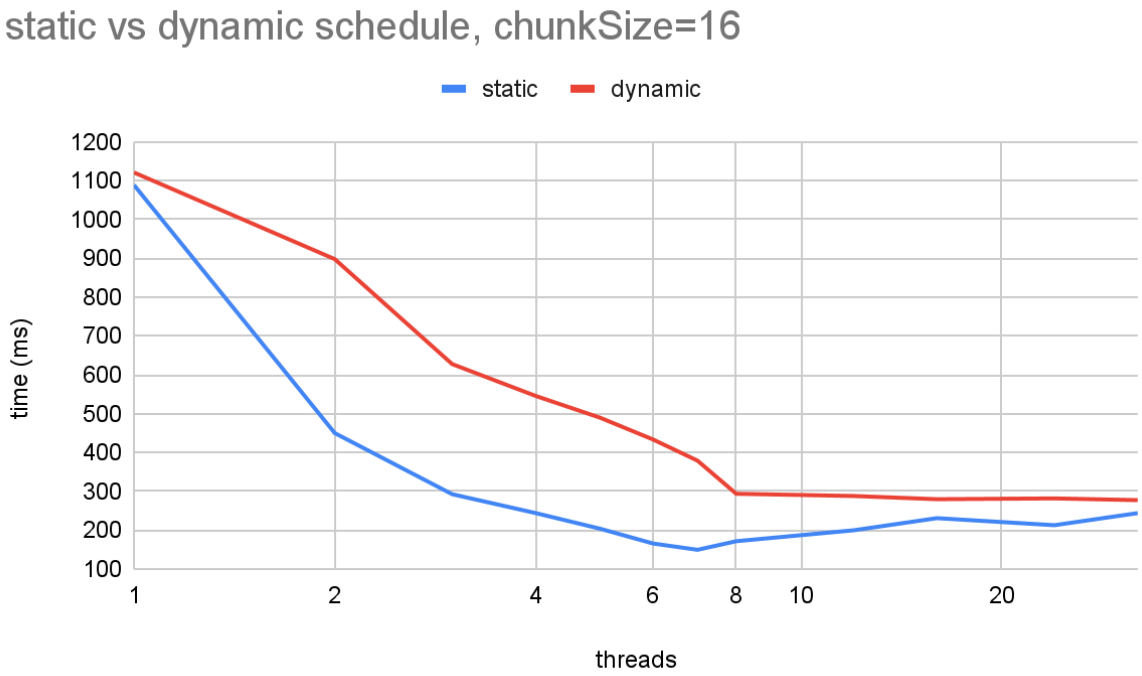

Рисунок 12 — график при параметре chunkSize = 16.

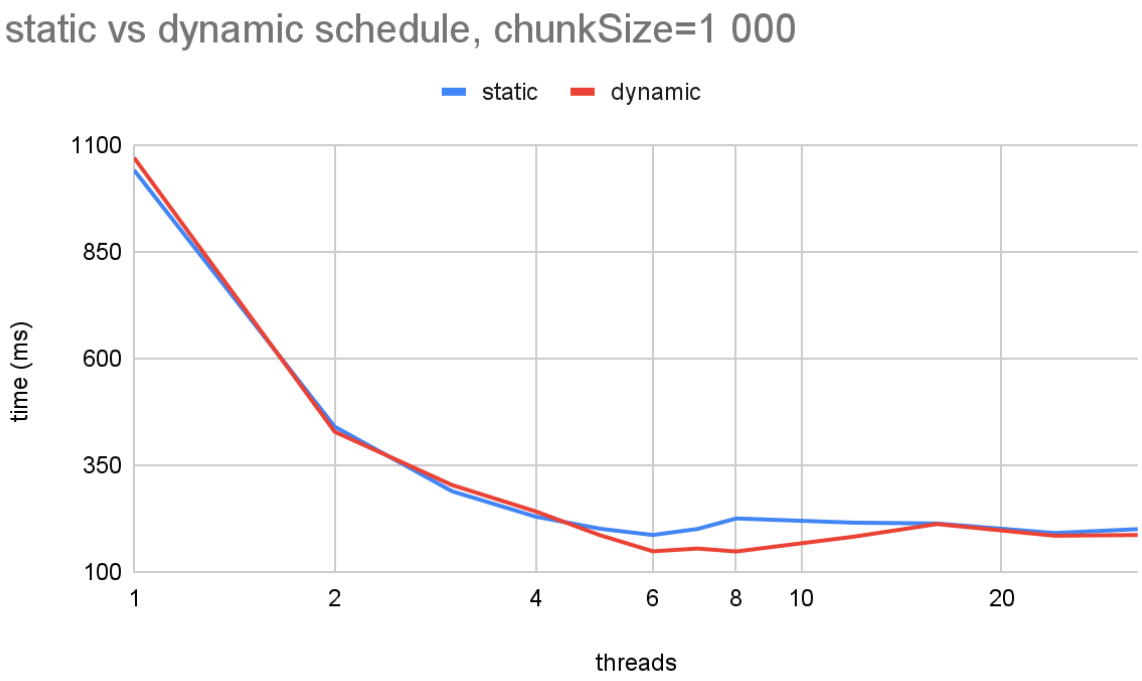

Рисунок 13 — график при параметре chunkSize = 1000.

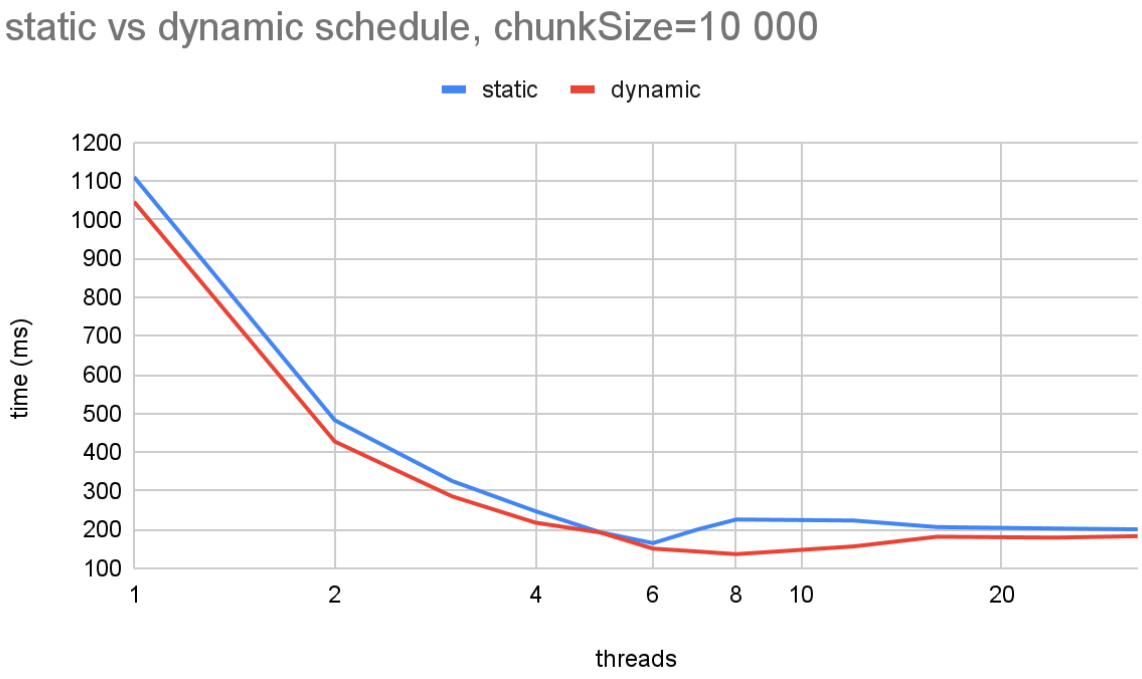

Рисунок 14 — график при параметре chunkSize = 10 000.

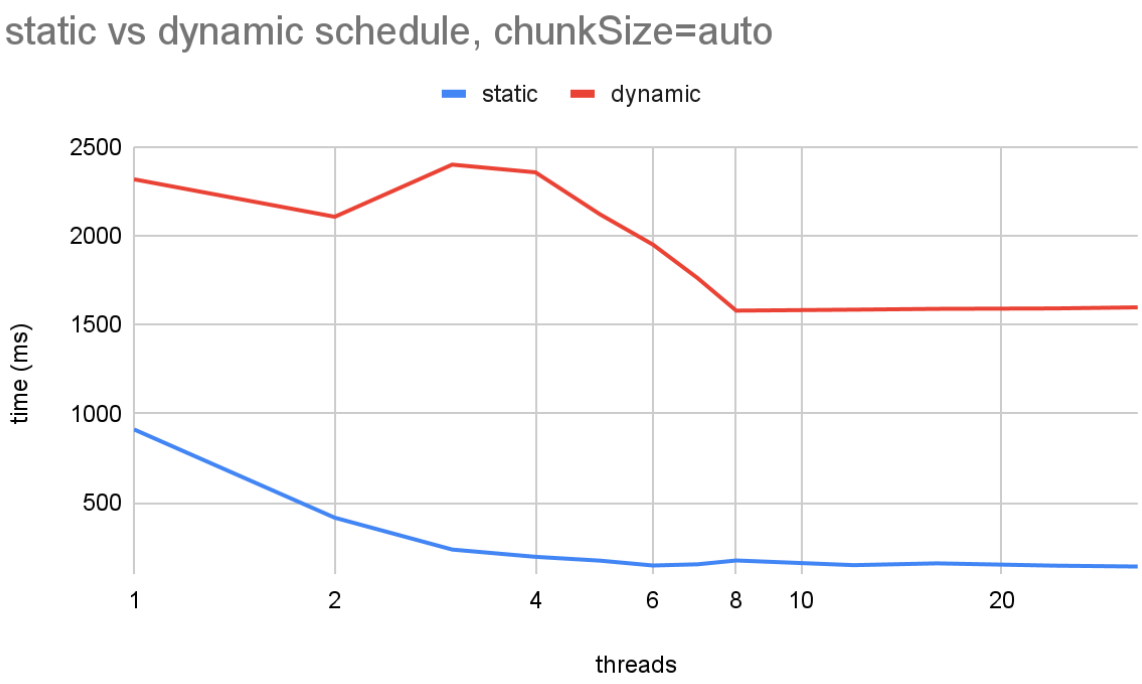

Рисунок 15 — график при параметре chunkSize = auto.

По этим графикам (рис. 10-15) можно сделать вывод, что на маленьких значениях chunkSize, а также при автоподборе, динамическое распределение показывает себя хуже, чем статическое. Однако уже после достаточно небольших (чуть больше 16) значений разница между этими методами становится в пределах погрешности.

При статическом распределении на графиках 10-15, а также при динамическом распределении на графиках 12-14 видно, что при увеличении количества потоков в два раза, время выполнения уменьшаться в два или больше раз. Однако после 8 потоков время перестаёт уменьшаться. Разумно предположить, что это связано с тем, что тестируется программа на процессоре с 8 логическими ядрами, и на 8 потоках получается пиковая производительность.

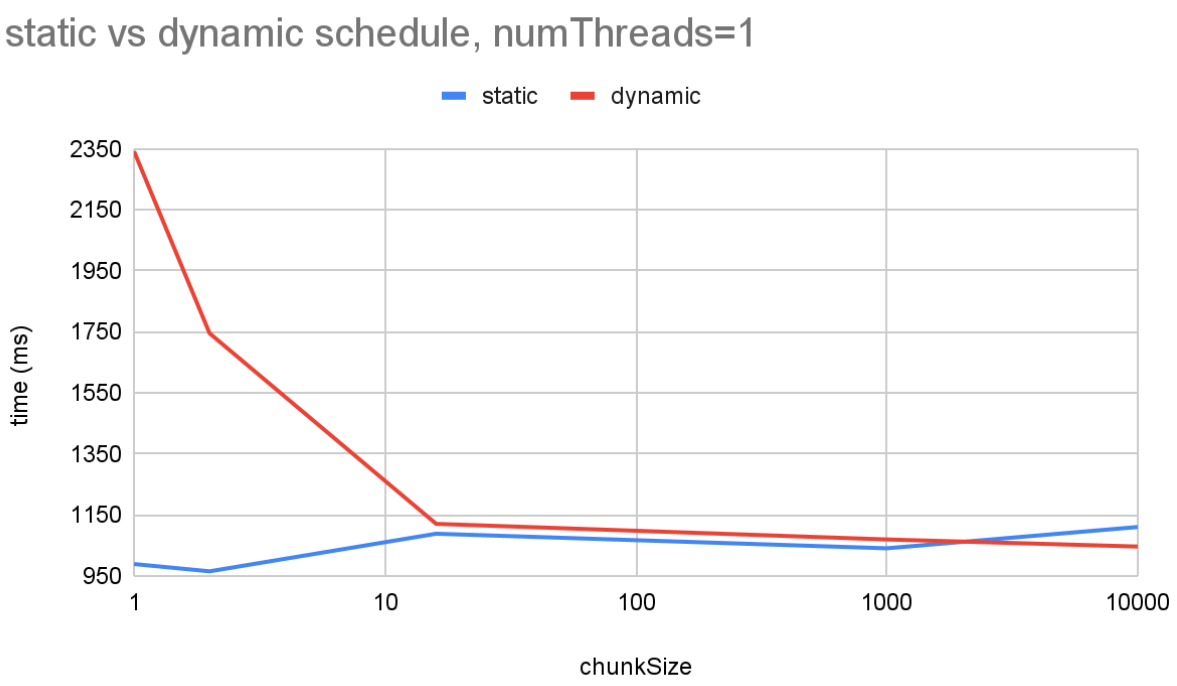

Рисунок 16 — график при числе потоков = 1.

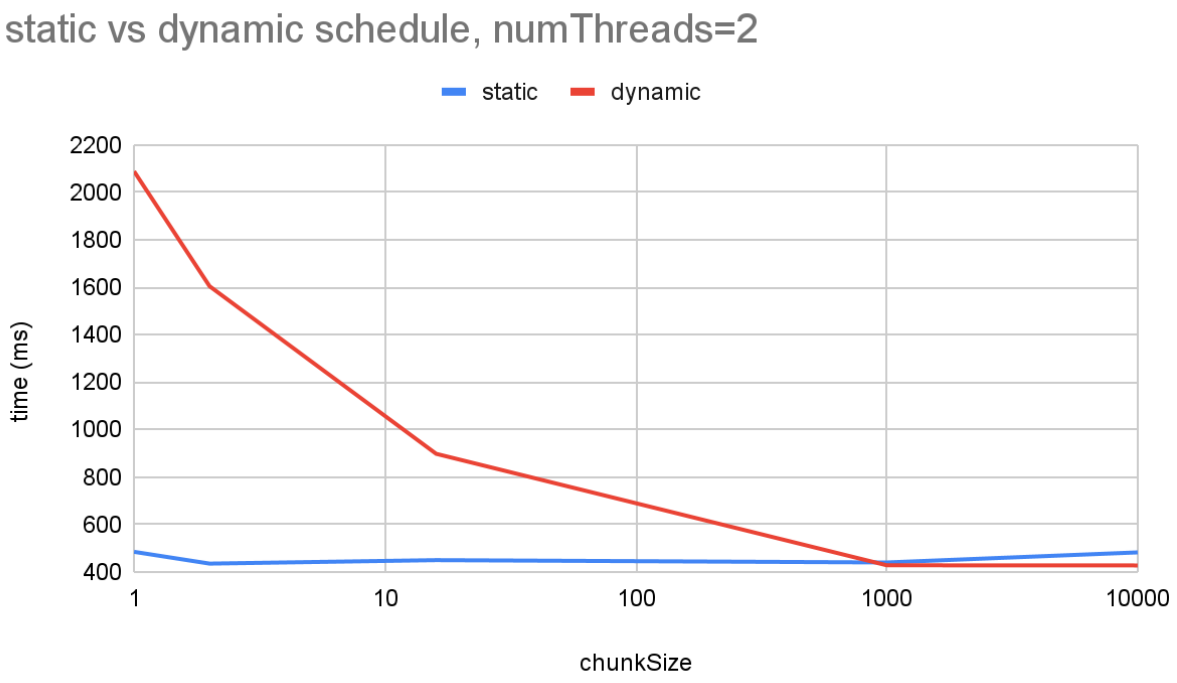

Рисунок 17 — график при числе потоков = 2.

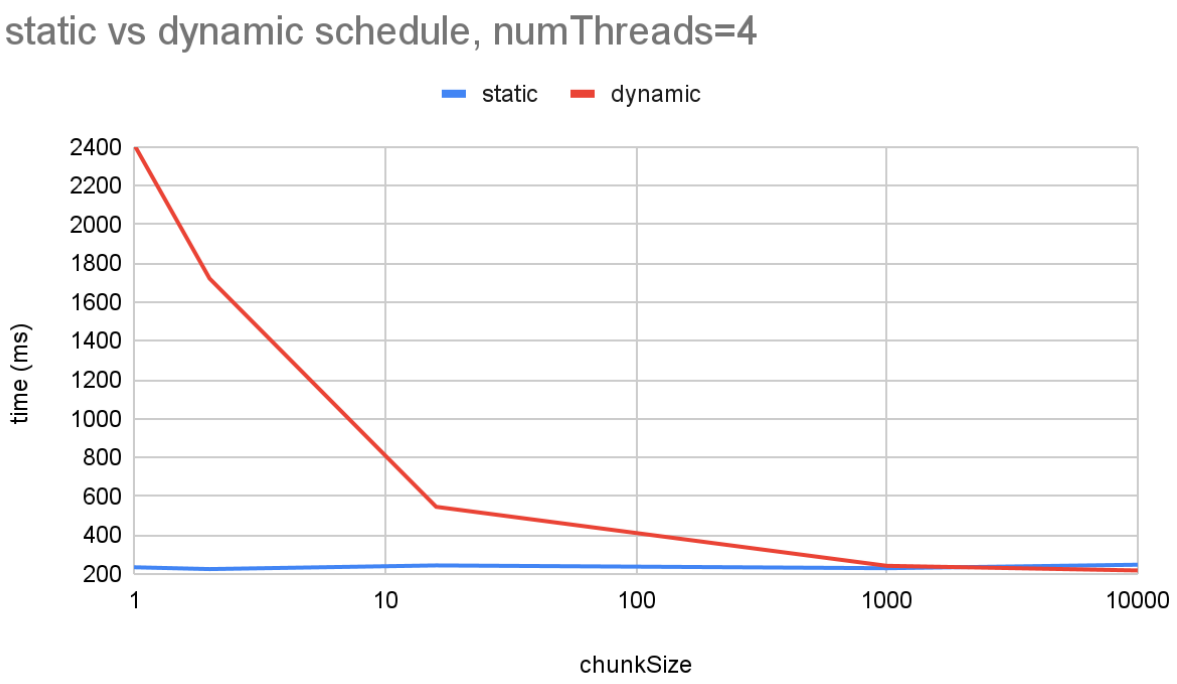

Рисунок 18 — график при числе потоков = 4.

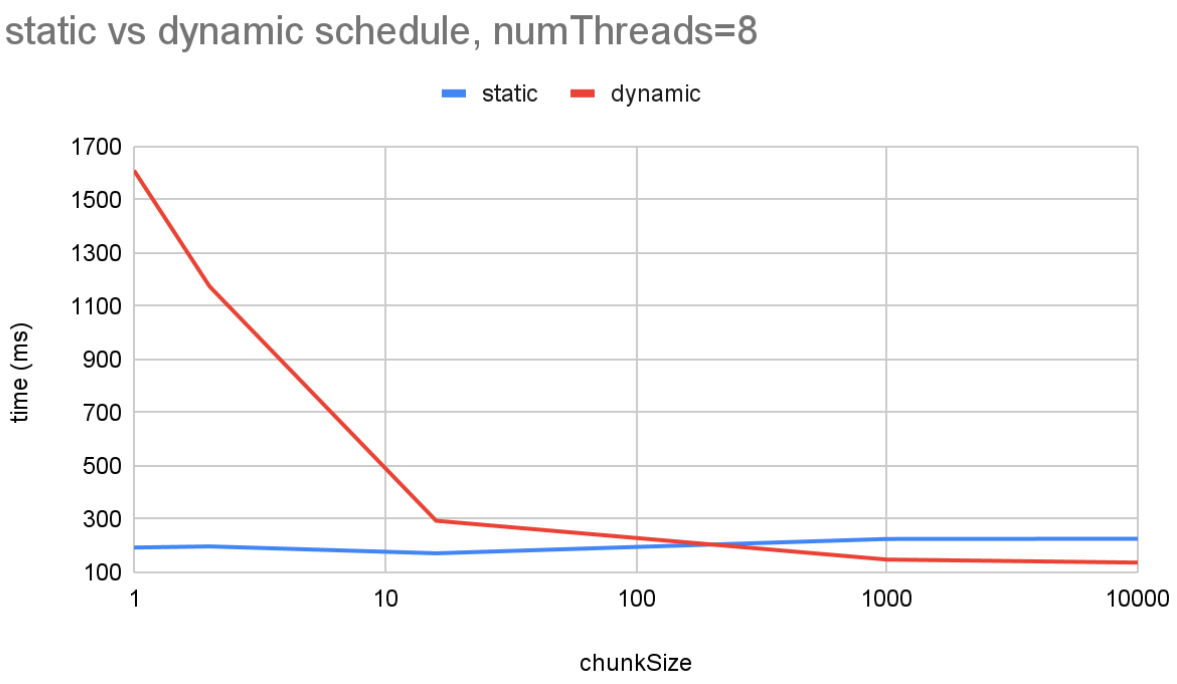

Рисунок 19 — график при числе потоков = 8.

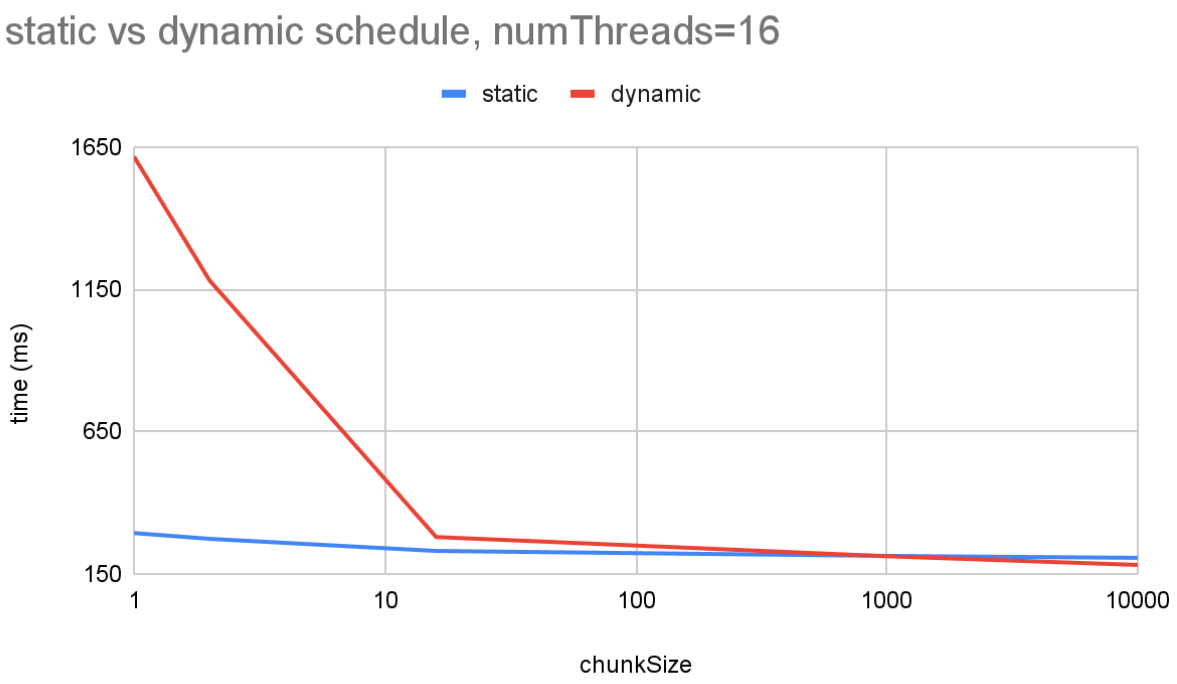

Рисунок 20 — график при числе потоков = 16.

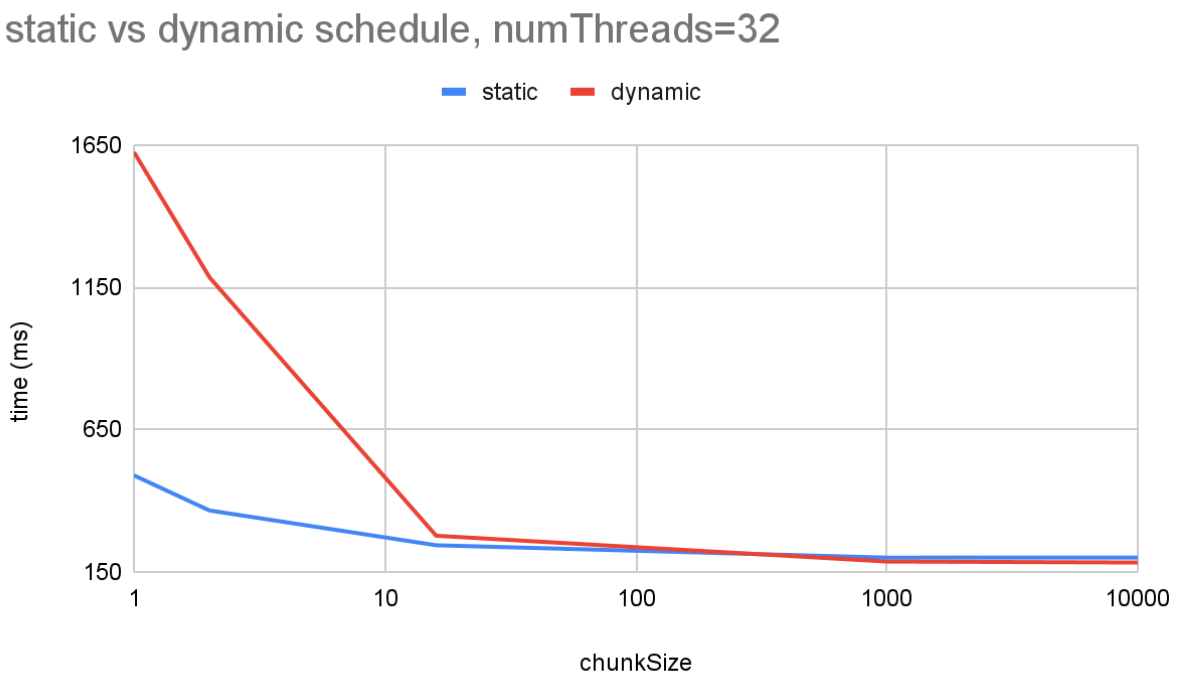

Рисунок 21 — график при числе потоков = 32.

Из графиков 16-21 видно, что при одном и том же количестве тредов статическое планирование показывает почти одинаковые результаты, в то время как результат динамического распределения улучшается с увеличением `chunkSize`. Очевидно, что это связано со спецификой нашей задачи: все итерации внутри цикла занимают примерно одинаковое время, и поэтому, если заранее распределить ресурсы, все треды будут начинать и заканчивать обрабатывать порции почти одновременно, поэтому распределение будет эффективным и не будет зависеть от размера порции. В это же время, при динамическом распределении на маленьких `chunkSize` балансировщику приходится в спешке раздавать задачи освободившимся тредам, причём каждый раз в разном порядке, и само распределение начинает потреблять достаточное количество ресурсов, чтобы соперничать с выигрышем от распараллеливания. С увеличением размера порций нагрузка на планировщик снижается, он уже не мечется, в панике оперируя огромным количеством маленьких кусочков, а начинает более спокойно раздавать задачи, в результате чего его эффективность повышается и становится такой же, как у статического планировщика, а иногда даже и лучше.

При выключенном OpenMP код работает 783 мс, а при включенном с одним потоком — 1036 мс. Из этого можно сделать вывод, что OpenMP всё же замедляет программу, съедая какие-то ресурсы, хоть и не очень много.

Стоит заметить, что ноутбук это не очень стабильная среда для тестирования каких-то алгоритмов. Плохое охлаждение, троттлинг, куча неудаляемого софта от Microsoft на фоне не добавляют тестам точности. Для того, чтобы хоть как-то это компенсировать, в каждой конфигурации было проведено три или более тестов, а затем взят средний результат.

Напомню, что для составления и анализа графиков **не учитывалось** время, затрачиваемое на подсчёт гистограммы распределения цвета. Если хочется, чтобы программа выводила время в том же формате, что и для графиков, то нужно раскомментировать строки 107 и 108 файла `Image.cpp` и закомментировать строки 99 и 100 файла `main.cpp`. В версии, приложенной в архиве и в секции листинга учитывается всё время работы, включая время на подсчёт гистограммы, исключая время на ввод и вывод.

Также стоит отметить, что инструкция выводить тот же самый цвет, если цвет всего один была осознанно проигнорирована: добавление этого частного случая в программу в ряде мест испортило бы код и общую идею, а также я считаю, что чёрный является самым хорошо различимым цветом, а потому — самым контрастным в естественном её понимании. По этой же причине при равенстве минимального и максимального значений, пиксель зануялся (перекрашивался в черный цвет).

Ниже прикреплены ссылки на исходники и тесты. Результаты работы программы на цветных тестах конвертированы в .jpg, потому что без сжатия занимают много места. Тесты прикреплены в исходном качестве.

- Исходники
- Архив с тестами
- Архив с логами (временем работы на тестах)
- Ссылка на таблицу с данными из логов и графиками
- Ссылка на этот отчёт