Лабораторная работа №4	39	2023
OpenMP	Ляпин Дмитрий Романович	

Цель работы: знакомство с основами многопоточного программирования.

Инструментарий: работа должна быть выполнена на C или C++. В отчёте указать язык и компилятор, на котором вы работали. Стандарт OpenMP 2.0.

Описание

Написать программу, решающую поставленную задачу (см. Вариант). Провести замеры времени на своём компьютере и привести графики времени работы:

1. при различных значениях числа потоков при одинаковом параметре `schedule` (без `chunk_size`);
2. при одинаковом значении числа потоков при различных параметрах `schedule` (с `chunk_size`);
3. с выключенным `openmp` и с включенным с 1 потоком.

Вариант

Hard: Пороговая фильтрация изображения методом Оцу по трём порогам.

Результат:

Со свёрнутыми внешними двумя циклами (ввиду запрета использовать `collapse`), на 8-ми потоках, со `schedule = guided,2048` программа работает на данном примере за 1.714 мс по сравнению с 9.327 мс без `omp`.

Теория

OpenMP

`OpenMP` – библиотека для распараллеливания вычислений для C, C++ и Fortran. Описание необходимых конструкций происходит с помощью директив препроцессора. Чтобы была возможность компиляции без `omp`, все они начинаются с `#pragma omp` и являются фактически комментариями. Фактически программист описывает, что должно быть сделано, а как это будет сделано – дело библиотеки, то есть, openmp – в первую очередь декларативный стандарт.

При входе в параллельный блок (`parallel`, см. ниже) создаётся группа (`team`) потоков, каждый из которых выполняет описанный в секции код. Некоторые части кода можно специфицировать как `single` (их выполнит ровно один из потоков) или `master` (их выполнит главный поток).

В программе использованы следующие директивы:

```
#pragma omp parallel default(none) shared(...)
```

Объявляет параллельный блок. Можно было бы использовать default(shared) по умолчанию, устанавливающий, что все внешние переменные, используемые внутри – shared. Но конструкция default(none) shared(...) заставляет явно перечислить эти переменные, то есть, дополнительно задуматься, правда ли они здесь нужны.

```
#pragma omp for
```

Объявляет цикл, который можно считать независимо. После неё следует обычный цикл for с ограничениями: итерации не должны зависеть друг от друга и это должно быть очевидно компилятору.

```
#pragma omp critical (name)
```

Объявляет секцию, обычно работающую с внешними ресурсами, которую нельзя выполнять двум и более потокам одновременно. Перед входом в эту секцию поток дождётся, пока предыдущий её окончит. Имя секции используется для обозначения нескольких секций, которые нужно синхронизировать друг с другом. Имя секции рекомендовано указывать всегда, так как все неименованные секции считаются

именованными одним и тем же именем, что может привести к излишнему простою.

```
#pragma omp barrier
```

Объявляет барьер синхронизации. Исполнение продолжится только после того, как все потоки дойдут до этого места.

Также можно было бы использовать reduction, но это запрещено ТЗ. Поэтому кодом опишем то, что эта директива на самом деле выполняет: создаётся локальная для потока переменная, в которой аккумулируются значения операции (бинарной, ассоциативной и коммутативной, например, суммы или минимума). В конце выполнения потока, с соответствующими предосторожностями, результат «сливается» в одну глобальную переменную. Пример:

```
        int sum = 0;
#pragma omp parallel default(none) shared(sum)
        {
            int local_sum = 0;
#pragma omp for
            for (int i = 0; i < 100; i++) {
                local_sum += i;
            }
#pragma omp atomic
            sum += local_sum;
        }
```

`Schedule`:

`Schedule` задаёт метод планировки распределения итераций по потокам. Есть три варианта:

- `static` – итерации распределяются равномерно в начале цикла.
- `dynamic` – итерации распределяются динамически. Как только один поток заканчивает свою итерацию, он захватывает следующую.
- `guided` – аналогично dynamic, но итерации распределяются блоками, размер которых зависит от того, сколько итераций осталось.

Можно задать также `chunk_size` – размер блока итераций.

- `static` – итерации распределяются блоками, насколько возможно равномерно.
- `dynamic` – захват итераций происходит блоками
- `guided` – размер блока уменьшается не ниже специфицированного.

По умолчанию `chunk_size = 1`. `Schedule` можно задать как явно в коде, так и runtime, установив, соответственно, `schedule(runtime)`. В среднем runtime-определённый `schedule` дольше, чем соответствующий `schedule`, но заданный явно.

Алгоритм Оцу

Алгоритм Оцу представляет из себя алгоритм разбиения значений на классы, с максимизацией межклассовой дисперсии (минимизации внутренней). В случае одного порога можно найти его быстрее, чем перебором, но нам нужно три порога (четыре класса), да и ТЗ однозначно требует перебора.

У нас есть изображение, которое, по сути, является отображением из $[0,\ w*h)$ в $[1,\ L]$. Поскольку речь о программировании, конечно, у нас $[0,\ L)$, но поправку на это мы (не) внесём позже.

Есть вероятность яркости: $p(l)=\frac{1}{wh}\sum_{x=1}^{w*h}[f(x)=l]=\frac{1}{wh}card\ f^{-1}(\{l\})$

Нам нужно выбрать пороги $f_1 \dots f_{M-1}$, где $M=4-$ количество кластеров. Для удобства дальнейшего описания положим $f_0=0$, $f_M=L$.

Зафиксируем пороги. Определим функции:

$$q_n=\sum_{l=f_{n-1}+1}^{f_n}p(l)$$

$$\mu_n=\sum_{l=f_{n-1}+1}^{f_n}\frac{l\cdot p(l)}{q_n}$$

где $n\in[1,\ M]$

Теперь осталось определить среднее значения яркости (взвешенное по вероятностям):

$$\mu=\sum_{n=1}^{M}\mu_n q_n=\sum_{l=1}^{L}l\cdot p(l)$$

И межкластерную дисперсию:

$$\sigma_B^2 = \sum_{n=1}^{M} q_n (\mu_n - \mu)^2 = \sum_{n=1}^{M} q_n (\mu_n - \mu)^2 = \sum_{n=1}^{M} \left(q_n {\mu_n}^2 - 2 q_n \mu_n \mu + q_n \mu^2 \right) =$$
$$= \sum_{n=1}^{M} q_n {\mu_n}^2 - 2\mu \sum_{n=1}^{M} q_n \mu_n + \mu^2 \sum_{n=1}^{M} q_n = \sum_{n=1}^{M} q_n {\mu_n}^2 - 2\mu \cdot \mu + \mu^2 \sum_{n=1}^{M} \sum_{l=f_{n-1}+1}^{f_n} p(l) =$$
$$= \sum_{n=1}^{M} q_n {\mu_n}^2 - 2\mu^2 + \mu^2 \sum_{l=f_0+1=1}^{f_M=L} p(l) = \sum_{n=1}^{M} q_n {\mu_n}^2 - \mu^2$$

… и понять, что, в целом, нам не очень была нужна μ^2, так как нам нужно максимизировать σ_B^2, и «плюс константа» на аргумент не влияет. Будем максимизировать $g = \sigma_B^2 + \mu^2$. Введём некоторые дополнительные функции:

$s_n = \sum_{l=f_{n-1}+1}^{f_n} l \cdot p(l)$, тогда $g = \sum_{n=1}^{M} q_n {\mu_n}^2 = \sum_{n=1}^{M} q_n \left(\frac{s_n}{q_n} \right)^2 = \sum_{n=1}^{M} \frac{{s_n}^2}{q_n}$.

Чтобы не пересчитывать раз за разом q_n и s_n, введём префиксные суммы соответствующих последовательностей:

$a(f) = \sum_{l=1}^{f} l \cdot p(l)$, $b(f) = \sum_{l=1}^{f} p(l)$, дополнительно определяя $a(0) = b(0) = 0$.
Тогда $s_n = a(f_n) - a(f_{n-1})$, $q_n = b(f_n) - b(f_{n-1})$,

$$g = \sum_{n=1}^{M} \frac{{s_n}^2}{q_n} = \frac{{s_1}^2}{q_1} + \frac{{s_2}^2}{q_2} + \frac{{s_3}^2}{q_3} + \frac{{s_4}^2}{q_4} = \frac{a(f_1)^2}{b(f_1)} + \frac{(a(f_2) - a(f_1))^2}{b(f_2) - b(f_1)} + \frac{(a(f_3) - a(f_2))^2}{b(f_3) - b(f_2)} + \frac{(a(L) - a(f_3))^2}{b(L) - b(f_3)}$$

Формула получилась слишком красивой, чтобы пересчитывать её для сдвига на единицу по шкале яркости. Да и так у нас удобно останется незатронутый ноль в яркостях, который можно будет удобно использовать при обсчёте префиксных сумм in-place. Первый и последний член суммы тоже предпосчитаем, для пущей эффективности.

Практика

Случай без использования omp прост. Считаем вероятности, считаем префиксные суммы, перебираем все возможные пороги, считая для них g, находим оптимальную комбинацию.

Предподсчёт.

С использованием omp можно распараллелить не только собственно подсчёт порогов, но и другие циклы. У меня таких четыре (точнее, пять, но два из них – подсчёт префиксных сумм). Каждый из них я попробовал распараллелить и нет, и сравнил время.

1) Подсчёт вероятностей
2) Подсчёт массива $l \cdot p(l)$
3) Подсчёт префиксных сумм
4) Преобразование картинки в соответствии с порогами

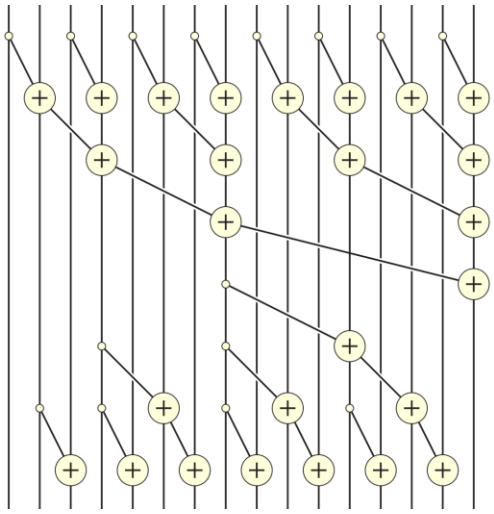
1 – параллельно, 0 – нет.

Время бралось по 10000 запусков.

1	2	3	4	Time, ms
0	0	0	0	2.546
0	0	0	1	2.347
0	0	1	0	4.900
0	0	1	1	3.962
0	1	0	0	2.865
0	1	0	1	2.438
0	1	1	0	4.959
0	1	1	1	3.998
1	0	0	0	1.660
1	0	0	1	0.785
1	0	1	0	3.451
1	0	1	1	2.747
1	1	0	0	1.737
1	1	0	1	0.895
1	1	1	0	3.717
1	1	1	1	2.784

Время, затраченное на пред- и пост-подсчёт

Алгоритмы распараллеливания префиксных сумм, которые мне удалось найти (один из таких ниже), на таких размер данных, по-видимому, крайне неэффективны: распараллеливание даёт время в 2мс, тогда как последовательным проходом – на уровне погрешности (1..3 мкс).



Пример параллельного подсчёта префиксной суммы
(Источник: https://en.wikipedia.org/wiki/Prefix_sum#Parallel_algorithms)

Второй пункт тоже показал небольшую просадку по производительности. Видимо, распараллеливать подсчёт длиной 256 невыгодно из-за накладных расходов.

Рассмотрим теперь оставшиеся два пункта:

1	4	Time, ms
0	0	2.546
0	1	2.347
1	0	1.660
1	1	0.785

Интересно, что выгода 11 по сравнению с 01 больше, чем у 10 по сравнению с 00. И аналогично у 11 относительно 10 больше, чем у 01 относительно с 00. Видимо, это связано с тем, что при работе с omp существенное время занимает начальная инициализация.

Подсчёт порогов.

Происходит перебор возможных значений порогов в трёх циклах. Чтобы эффективно их распараллелить, можно было бы указать `collapse(3)`:

```
#pragma omp for collapse(3)
        for (int try_f1 = 1; try_f1 <= 254; try_f1++) {
            for (int try_f2 = try_f1 + 1; try_f2 <= 255; try_f2++) {
                for (int try_f3 = try_f2 + 1; try_f3 <= 256; try_f3++)
{
```

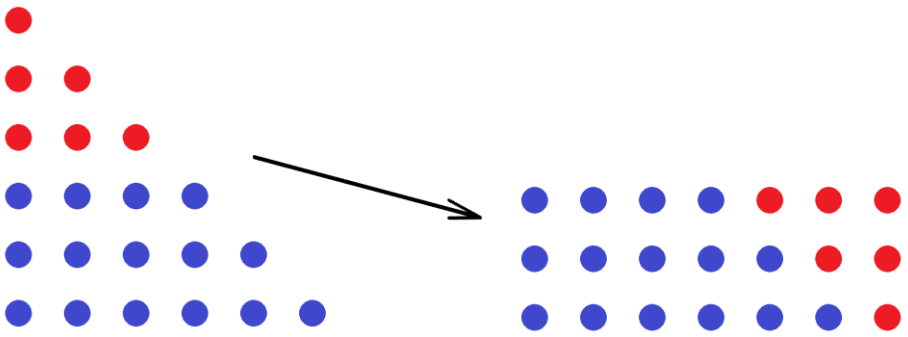
Но оно появится в более позднем стандарте, так что сейчас у нас есть выбор из трёх вариантов:

1) Оставить как было, пометить `#pragma omp for` внешний цикл и упокоиться.
2) Склеить три цикла как квадратные 254x254x254:

```
#pragma omp for
        for (long x = 0; x < 254 * 254 * 254; x++) {
            int try_f1 = x % (254 * 254);
            int try_f2 = (x / 254) % 254 + 1;
            int try_f3 = x % 254 + 2;
            if (try_f2 <= try_f1 || try_f3 <= try_f2) continue;
```

3) С помощью некоторой математики склеить два внешних цикла:

```
#pragma omp for
        for (int x = 0; x < (254 * (255) / 2); x++) {
            int r = x / (255);
            int c = x % (255);
            int try_f1 = c > (r + 127) ? (255 - c) : c + 1;
            int try_f2 = c > (r + 127) ? (128 - r) : (r + 129);
            for (int try_f3 = try_f2 + 1; try_f3 < 256; try_f3++) {
```



(Иллюстрация к методу склеивания)

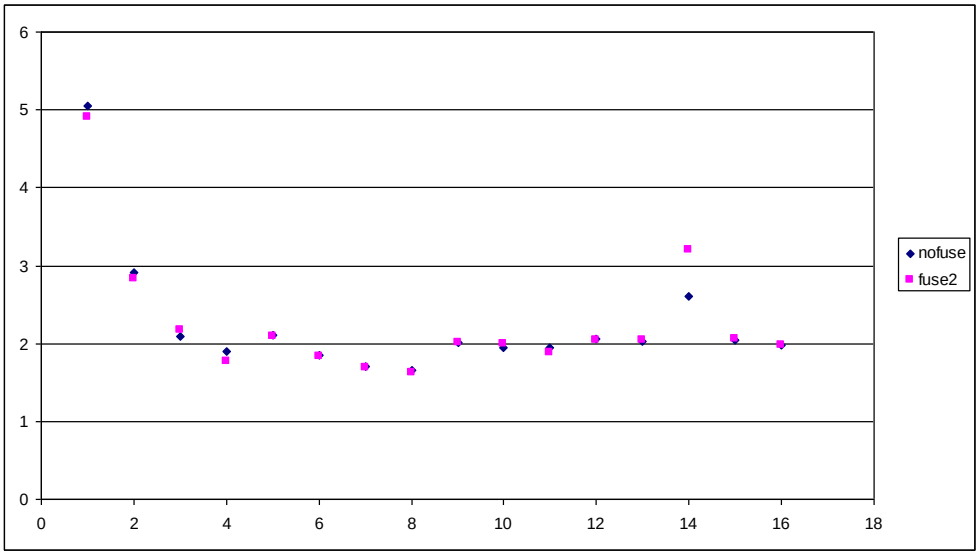
Далее эти методы называю `nofuse`, `fuse3` и `fuse2` соответственно.

Позапускаем программу при постоянном `schedule`, скажем, `static,1`, чтобы выяснить, какое количество потоков оптимально. Время всюду берётся по 1000 запусков, в миллисекундах. Здесь я переключился с `debug` на `release`, поэтому время резко упало примерно в пять раз.

threads	nofuse	fuse2	fuse3
1	5.047	4.904	40.631
2	2.908	2.837	20.682
3	2.084	2.177	15.484
4	1.897	1.774	12.387
5	2.107	2.087	15.673
6	1.845	1.829	13.582
7	1.703	1.687	11.864
8	1.649	1.626	11.269
9	2.011	2.003	13.213
10	1.946	1.992	13.134
11	1.949	1.886	13.063
12	2.053	2.050	13.027
13	2.033	2.047	12.772
14	2.604	3.201	12.673
15	2.042	2.061	11.576
16	1.983	1.981	11.294

Время работы в мс в зависимости
от кол-ва потоков

У меня четырёхядерный процессор с hyperthreading, следовательно, и максимум производительности достигается при восьми потоках. `Fuse3` очевидным образом проигрывает, а вот `nofuse` и `fuse2` ещё поборются.



То же, диаграмма.

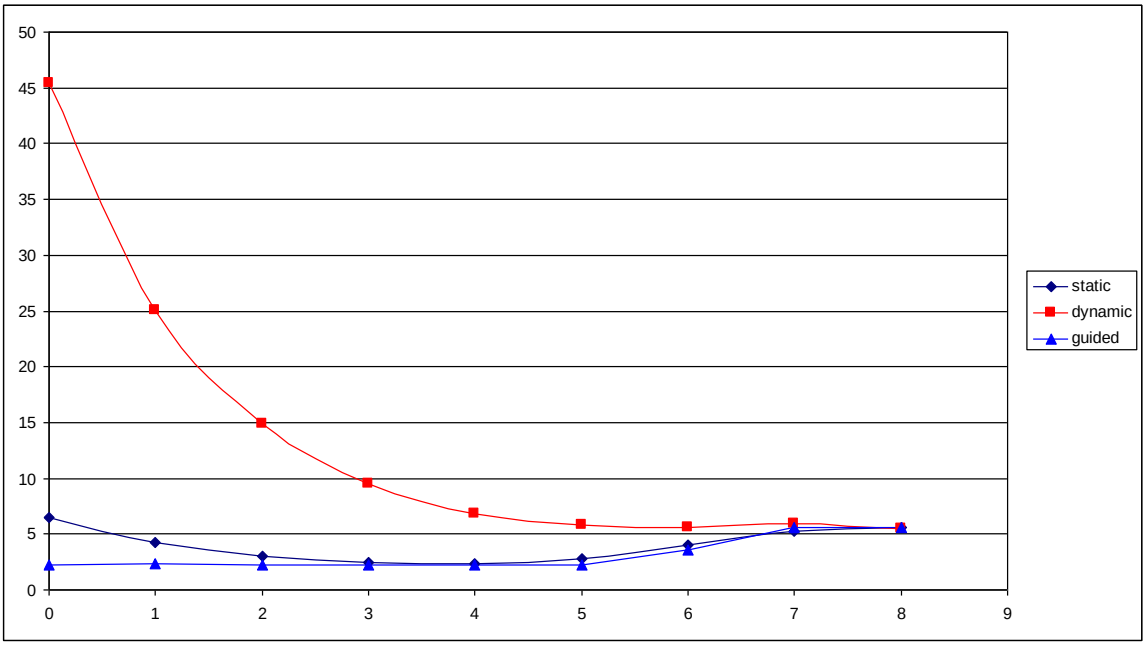
Положим количество потоков равным 8, и попробуем разные `schedule`, с `chunk_size` вплоть до 256*256 для `fuse2` и до 256 для `nofuse` (количество итераций):

LOG2 (chunk_size)	static	dynamic	guided
0	6.485	45.429	2.277
1	4.202	25.093	2.378
2	3.042	14.89	2.257
3	2.497	9.54	2.249
4	2.372	6.838	2.246
5	2.82	5.775	2.25
6	3.978	5.637	3.583
7	5.22	5.975	5.55
8	5.618	5.535	5.547

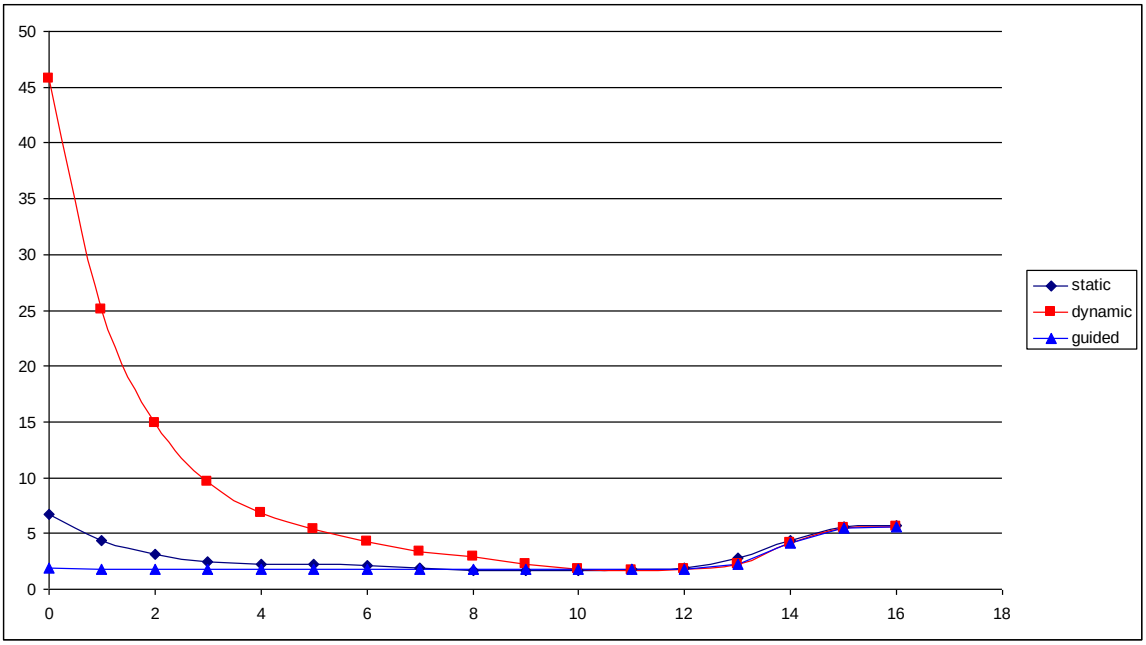
Время работы в зависимости от schedule (nofuse), мс

LOG2 (chunk_size)	static	dynamic	guided
0	6.733	45.714	1.903
1	4.359	25.078	1.83
2	3.088	14.889	1.81
3	2.473	9.565	1.814
4	2.256	6.841	1.813
5	2.238	5.32	1.828
6	2.089	4.229	1.809
7	1.869	3.399	1.796
8	1.671	2.923	1.77
9	1.672	2.265	1.754
10	1.696	1.835	1.742
11	1.764	1.714	1.736
12	1.951	1.807	1.81
13	2.842	2.289	2.282
14	4.335	4.092	4.113
15	5.557	5.534	5.53
16	5.674	5.615	5.61

Время работы в зависимости от schedule (fuse2), мс



Время работы в зависимости от schedule (nofuse), мс

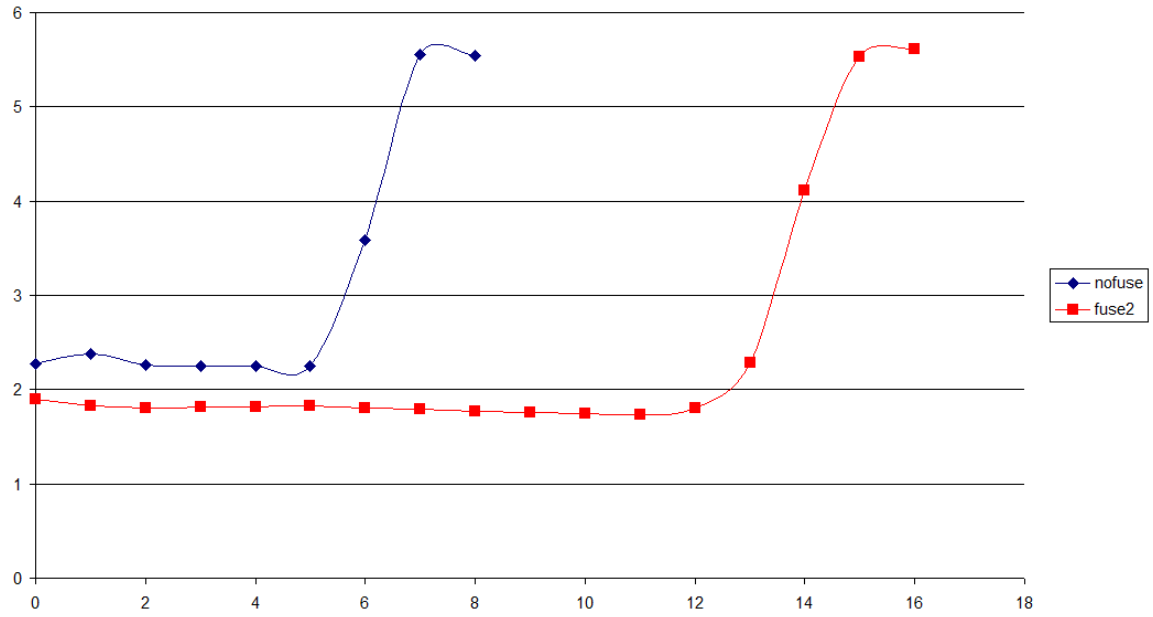


Время работы в зависимости от schedule (fuse2), мс

Видно, что `guided` выигрывает в большинстве случаев. `Static` дал бы хороший результат, если бы удалось адекватно склеить три цикла в один, поскольку все итерации внутреннего занимают примерно одинаковое время. Но у нас идёт распараллеливание не по всем, поэтому итерацией по факту является внутренний `for` или два. `Dynamic`, особенно на маленьких `chunk_size`, тратит слишком много времени на раскидывание итераций по потокам.

«Ступенька» в конце обоих графиков возникает из-за того, что при `chunk_size`, приближающемся к количеству итераций, работают по сути меньше, чем 8 потоков – остальным не достаётся.

Взглянем поближе на `guided`:



Время работы в зависимости от `chunk_size`, мс (`guided`)

В упорной борьбе победил `fuse2`, а из опробованных `chunk_size` наибольшую производительность даёт $2^{11} = 2048$. Можно предположить, что в целом `guided` перестраховывается, и старается к концу раздать поменьше итераций несмотря на то, что их можно продолжать раздавать большими порциями. Оставим в коде эти замечательные значения.

Приложения

1. Результат работы программы.

```
77 130 187
Time (8 thread(s)): 2.00009 ms
```

Входное изображение

Результат работы программы

2. Характеристики компьютера.
(Отчёт CPU-Z)

```
Processors Information
-------------------------------------------------------------------
Socket 1              ID = 0
   Number of cores          4 (max 4)
   Number of threads        8 (max 8)
   Number of CCDs           1
   Manufacturer             AuthenticAMD
   Name                     AMD Ryzen 3 Mobile 5300U
   Codename                 Lucienne
   Specification            AMD Ryzen 3 5300U with Radeon Graphics
   Package                  Socket FP6
   CPUID                    F.8.1
   Extended CPUID           17.68
   Core Stepping
   Technology               7 nm
   Core Speed               2069.1 MHz
   Multiplier x Bus Speed   20.75 x 99.7 MHz
   Base frequency (cores)   99.7 MHz
   Base frequency (mem.)    99.7 MHz
   Instructions sets  MMX  (+),  SSE,  SSE2,  SSE3,  SSSE3,  SSE4.1,
SSE4.2, SSE4A, x86-64, AES, AVX, AVX2, FMA3, SHA
      Microcode Revision    0x8608103
      L1 Data cache         4 x 32 KB (8-way, 64-byte line)
      L1 Instruction cache  4 x 32 KB (8-way, 64-byte line)
      L2 cache              4 x 512 KB (8-way, 64-byte line)
      L3 cache              4 MB (16-way, 64-byte line)
      Clock Speed 0         2069.15 MHz (Core #0)
      Clock Speed 1         1866.15 MHz (Core #1)
      Clock Speed 2         2069.15 MHz (Core #2)
      Clock Speed 3         2069.15 MHz (Core #3)
      Core 0 max ratio (effective) 39.00
      Core 1 max ratio (effective) 39.00
      Core 2 max ratio (effective) 39.00
      Core 3 max ratio (effective) 39.00
Software
-------------------------------------------------------------------
Windows Version            Microsoft   Windows   10   (10.0)   Home
Single Language 64-bit   (Build 19044)
DirectX Version            12.0
```

3. Листинг кода

hard.cpp

```
using namespace std;

#include <vector>
#include <omp.h>
#include <iostream>
#include <string>
#include "linear.h"
#include "parallel.h"

#define incorrect_use runtime_error("Incorrect call: specify threads number, input file and output file");
#define incorrect_thread_number runtime_error("Incorrect thread number: expected positive number, 0 (by default) or -1 (without omp)");

typedef unsigned char uchar;
typedef long long llong;

void process(vector<uchar> &data, int mode, int &f1, int &f2, int &f3) {
    if (mode == -1) {
        process_linear(data, f1, f2, f3);
    } else if (mode >= 0) {
        process_parallel(data, mode, f1, f2, f3, 2);
    } else throw incorrect_thread_number;
}

int main(int argc, char *argv[]) {
    if (argc < 4) throw incorrect_use;
    int mode = stoi(argv[1]);
    char *in_file = argv[2];
    char *out_file = argv[3];

    int w, h;
    vector<uchar> data = read_image(in_file, w, h);

    int f1, f2, f3;
    double begin_time = omp_get_wtime();
    process(data, mode, f1, f2, f3);
    double end_time = omp_get_wtime();
    printf("%u %u %u\n", f1, f2, f3);
    printf("Time (%i thread(s)): %g ms\n",
           mode == -1 ? 1 : mode == 0 ? omp_get_max_threads() : mode,
           (end_time - begin_time) * 1000);

    write_image(out_file, w, h, data);
```

```
    return                                                    0;
}
```

parallel.h

```
#pragma clang diagnostic push
#pragma ide diagnostic ignored "cppcoreguidelines-narrowing-
conversions" "UnusedValue"
#pragma ide diagnostic ignored "UnusedValue"

#ifndef IBM1_PARALLEL_H
#define IBM1_PARALLEL_H

#endif //IBM1_PARALLEL_H


#include <vector>
#include <omp.h>
#include <iostream>
#include <fstream>
#include <string>
#include "inout.h"

typedef unsigned char uchar;
typedef long long llong;

void count_probabilities(vector<uchar> &data, int threads, llong
dest[]) {
    vector<long *> ps(threads);
#pragma omp parallel default(none) shared(ps, data, dest,
threads)
    {

        long ps_local[256]{};
#pragma omp for schedule(guided, 2048)
        for (uchar i: data) ps_local[i]++;
        ps[omp_get_thread_num()] = ps_local;
#pragma omp barrier
#pragma omp for schedule(guided, 2048)
        for (int i = 0; i < 256; i++) {
            llong s = 0;
            for (int t = 0; t < threads; t++) {
                s += ps[t][i];
            }
            dest[i + 1] = s;
        }
    }
}

void process_parallel(vector<uchar> &data, int mode, int &f1, int
&f2, int &f3, int type) {
    if (mode != 0) omp_set_num_threads(mode);
```

```
    int threads = omp_get_max_threads();

    llong p[257]{};
    count_probabilities(data, threads, p);

    llong a[257]{};
    llong b[257]{};
    for (int i = 0; i < 257; i++) a[i] = i * p[i];
    for (int i = 1; i < 257; i++) b[i] = b[i - 1] + p[i];
    for (int i = 1; i < 257; i++) a[i] += a[i - 1];

    double term1[254]{};
    for (int i = 1; i <= 254; i++) {
        term1[i - 1] = (a[i] * a[i] + .0) / b[i];
    }

    double term4[254]{};
    for (int i = 3; i <= 256; i++) {
        llong t = a[256] - a[i];
        term4[i - 3] = (t * t + .0) / (b[256] - b[i]);
    }

    int f1m, f2m, f3m;
    f1m = f2m = f3m = 0;
    double gmax = -1.0;
#pragma omp parallel default(none) shared(a, b, term1, term4,
f1m, f2m, f3m, gmax)
    {
        int f1m_local, f2m_local, f3m_local;
        f1m_local = f2m_local = f3m_local = 0;
        double gmax_local = -1.0;
#pragma omp for schedule(guided, 2048)
        for (int x = 0; x < (254 * (255) / 2); x++) {
            int r = x / (255);
            int c = x % (255);
            int try_f1 = c > (r + 127) ? (255 - c) : c + 1;
            int try_f2 = c > (r + 127) ? (128 - r) : (r + 129);
            for (int try_f3 = try_f2 + 1; try_f3 < 256; try_f3++)
{
                llong t2 = a[try_f2] - a[try_f1];
                llong t3 = a[try_f3] - a[try_f2];
                double g = (
                                term1[try_f1 - 1] +
                                (t2 * t2 + .0) / (b[try_f2] -
b[try_f1])
                            ) + (
                                (t3 * t3 + .0) / (b[try_f3] -
b[try_f2]) +
                                term4[try_f3 - 3]
                            );
                if (g > gmax_local) {
                    gmax_local = g;
```

```
                    f1m_local = try_f1;
                    f2m_local = try_f2;
                    f3m_local = try_f3;
                }
            }
        } // end #pragma omp for

#pragma omp critical (join_max)
        {
            if (gmax_local > gmax) {
                gmax = gmax_local;
                f1m = f1m_local;
                f2m = f2m_local;
                f3m = f3m_local;
            }
        }
    } // end #pragma omp parallel
    f1 = f1m - 1;
    f2 = f2m - 1;
    f3 = f3m - 1;

#pragma omp parallel for default(none) shared(data, f1m, f2m,
f3m) schedule(guided, 2048)
    for (uchar &pix: data) {
        pix = pix < f1m ? 0 : pix < f2m ? 84 : pix < f3m ? 170 :
255;
    }
}

#pragma clang diagnostic pop
```

linear.h

```
#ifndef IBM1_LINEAR_H
#define IBM1_LINEAR_H

#endif //IBM1_LINEAR_H

typedef unsigned char uchar;
typedef long long llong;

#include <vector>
#include <omp.h>
#include <iostream>
#include <fstream>
#include <string>

void process_linear(vector<uchar> &data, int &f1, int &f2, int
&f3) {
    long p[256]{};
```

```
    llong a[257]{};
    llong b[257]{};
    for (uchar i: data) p[i]++;

    for (int i = 0; i < 256; i++) {
        a[i + 1] = a[i] + (i + 1) * p[i];
        b[i + 1] = b[i] + p[i];
    }

    double term4[254]{};
    for (int f3 = 3; f3 <= 256; f3++) {
        llong t = a[256] - a[f3];
        term4[f3 - 3] = (t * t + .0) / (b[256] - b[f3]);
    }

    int f1m, f2m, f3m;
    f1m = f2m = f3m = 0;
    double gmax = -1.0;

    for (int try_f1 = 1; try_f1 <= 254; try_f1++) {
        double term1 = (a[try_f1] * a[try_f1] + .0) / b[try_f1];
        for (int try_f2 = try_f1 + 1; try_f2 <= 255; try_f2++) {
            llong t2 = a[try_f2] - a[try_f1];
            double term2 = (t2 * t2 + .0) / (b[try_f2] -
b[try_f1]);
            for (int try_f3 = try_f2 + 1; try_f3 <= 256;
try_f3++) {
                llong t3 = a[try_f3] - a[try_f2];
                double g = term1 + term2 + (t3 * t3 + .0) /
(b[try_f3] - b[try_f2]) + term4[try_f3 - 3];
                if (g > gmax) {
                    gmax = g;
                    f1m = try_f1;
                    f2m = try_f2;
                    f3m = try_f3;
                }
            }
        }
    }

    f1 = f1m - 1;
    f2 = f2m - 1;
    f3 = f3m - 1;

    for (uchar &pix: data) {
        pix = pix < f1m ? 0 : pix < f2m ? 84 : pix < f3m ? 170 :
255;
    }
}
```

inout.h

```
using namespace std;

#ifndef IBM1_INOUT_H
#define IBM1_INOUT_H

#endif //IBM1_INOUT_H

#include <fstream>
#include <vector>
#include <string>

#define wrong_magic runtime_error("Image file must start with
'P', then encoding type (1-6), then LF.");
#define unsupported_encoding runtime_error("Only P5 encoding is
supported.");
#define width_not_specified runtime_error("Width is not specified
or is zero");
#define height_not_specified runtime_error("Height is not
specified or is zero");
#define depth_not_specified runtime_error("Color depth is not
specified or is zero");
#define unsupported_depth runtime_error("Only 8-bit-depth images
are supported");
#define not_enough_pixels runtime_error("Not enough pixel
bytes");
#define too_many_pixels runtime_error("Too many pixel bytes");
#define cant_open_file runtime_error("Can't open file");

typedef unsigned char uchar;
typedef long long llong;

int read_int(fstream &inp) {
    //Skips one char after number
    char ch;
    int res = 0;
    while ((inp >> noskipws >> ch) && ch >= '0' && ch <= '9') {
        res = res * 10 + ch - '0';
    }
    return res;
}

vector <uchar> read_image(char *file, int &w, int &h) {
    fstream inp(file, fstream::in);
    if (!inp) throw cant_open_file;

    char ch;
    if (!(inp >> noskipws >> ch) || ch != 'P') throw wrong_magic;
    if (!(inp >> noskipws >> ch) || ch < '1' || ch > '6') throw
wrong_magic;
    if (ch != '5') throw unsupported_encoding;
```

```
    if (!(inp >> noskipws >> ch) || ch != '\n') throw
wrong_magic;
    w = read_int(inp);
    if (w == 0) throw width_not_specified;
    h = read_int(inp);
    if (h == 0) throw height_not_specified;
    int d = read_int(inp);
    if (d == 0) throw depth_not_specified;
    if (d != 255) throw unsupported_depth;

    int size = w * h;
    char *arr = new char[size];
    inp.read(arr, size);
    inp.close();
    vector <uchar> data(size);
    for (int i = 0; i < size; i++) data[i] = arr[i];
    return data;
}

void write_image(char* file, int w, int h, vector<uchar> data) {
    fstream out(file, fstream::out);
    if (!out) throw cant_open_file;
    out << "P5\n" << w << ' ' << h << "\n255\n";
    for (uchar i: data) out << (char) i;
    out.close();
}
```