Лабораторная работа №5	**M3137**	*2023*
OpenMP	**Кадомцев Кирилл Максимович**	

Репозиторий

1. Инструментарий

Язык – C++20, компилятор MinGW 6.3.0

2. Результат работы программы

Запуск производился на процессоре *AMD Ryzen 7 6800H*. Лучший по скорости был получен при использовании 12 потоков. В качестве входных данных использован пример из ТЗ.

```
Time (12 thread(s)): 5233 ms

Process finished with exit code 0
```

3. Описание использованных конструкций OMP

#pragma omp parallel: Данная директива создает параллельный блок кода, внутри которого команды будут выполняться параллельно несколькими потоками.

#pragma omp for schedule: Эта директива используется внутри параллельного блока для распараллеливания цикла. Schedule устанавливает принцип распределения итераций между потоками. ***Подробнее тема планирования будет раскрыта в разделе тестирование.***

#pragma omp atomic: Директива atomic используется внутри цикла для обеспечения гарантии, что только один поток может выполнять эту операцию в заданный момент времени.

omp_get_max_threads(): данная функция возвращает максимально возможное количество потоков.

omp_get_thread_num(): функция, возвращающая номер текущего потока.

omp_set_num_threads(int): функция, устанавливающая число потоков.

4. Описание работы кода

Рассмотрим часть кода, непосредственно использующую директивы и функции *Omp*.

```
double monte_carlo::calc_parallel(int threads_number, int chunk_size) {
    omp_set_num_threads(threads_number);
    int total_hits = 0;
    double semi_rib = rib / 2.0;
    #pragma omp parallel
    {
        unsigned int seed = time(NULL) ^ (time(NULL) -
omp_get_thread_num());

        int local_hits = 0;
        #pragma omp for schedule(dynamic, 10000)
        for (int i = 0; i < n; i++) {
            double cur_point[3];
            for (int t = 0; t < 3; t++) {
                if (generator_number == 0) {
                    cur_point[t] = get_random0(-semi_rib, semi_rib, seed);
                } else if (generator_number == 1) {
                    cur_point[t] = get_random1(-semi_rib, semi_rib, seed);
                } else {
                    cur_point[t] = get_random2(-semi_rib, semi_rib, &seed);
                }
            }
            if (check_hit(cur_point[0], cur_point[1], cur_point[2],
semi_rib)) {
                local_hits++;
            }
        }
        #pragma omp atomic
        total_hits += local_hits;
    }
    double volume_estimate = pow(2 * radius, 3) * double(total_hits) /
double(n);
    return volume_estimate;
}
```

Функция *omp_get_max_threads()* вызывается для того, чтобы определить размеры vector'а, в котором хранятся генераторы псевдослучайных чисел. Сам vector нужен для того, чтобы генерация случайных чисел адекватно «параллелилась», а размер его размер определён таким образом в том случае, когда мы используем количество потоков по умолчанию и не можем точно сказать, сколько потоков установит Omp.

Функция *omp_set_num_threads(threads_number)* устанавливает число потоков. Далее директивой *#pragma omp parallel* создаётся параллельная

область, в рамках которой создаётся столько потоков, сколько было установленно функцией *omp_set_num_threads(threads_number)*, или значение по умолчанию, если данная функция вызвана не была.

Далее в рамках этой области распараллеливается цикл for директивой *#pragma omp for schelude()*. Для того, чтобы значения правильно распараллеливались, каждый поток записывает попадания в свою ячейку vector'а попаданий, а в конце они суммируются.

Стоит также отметить, по какому принципу были выбраны генераторы случайных чисел. Стандарт языка C++ предоставляет три основных генератора – mt19937, minstd_rand и ranlux (остальные способы не являются потокобезопасными в силу того, что используют глобальные параметры). Во втором реализован алгоритм, предоставляющий меньшую точность, но и тратящий меньше времени. Последний, напротив, является наиболее точным, но наименее быстрым. В то время как представляет между собой некоторый компромисс. Однако точность третьего генератора оказалась избыточной для поставленных условий, при этом крайне неэффективной по скорости. Поэтому в качестве третьего генератора был выбран иной – rand_r(), который на данный момент не включён в cpp. Его реализация была вставлена непосредственно в код (ссылка на исходник прилагается в приведённом ниже коде).

```
double monte_carlo::get_random0(double min, double max, unsigned int seed) {
    static thread_local std::mt19937 generator(seed);
    uint32_t rnd = generator();
    return min + (max - min) * (double(rnd) / std::mt19937::max());
}

double monte_carlo::get_random1(double min, double max, unsigned int seed) {
    static thread_local std::minstd_rand generator(seed);
    uint32_t rnd = generator();
    return min + (max - min) * (double(rnd) / std::minstd_rand ::max());
}

double monte_carlo::get_random2(double min, double max, unsigned int *seed)
{
    int randNum = rand_r(seed);
    double normalized = (double)randNum / (double) INT_MAX;
    return min + (max - min) * normalized;
}
```

```
//
https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/rand_r.c;h=50bec5deb
3e8339168cc70d37b616e2b685f80e1;hb=HEAD
int monte_carlo::rand_r(unsigned int *seed) {
    unsigned int next = *seed;
    int result;

    next *= 1103515245;
    next += 12345;
    result = (unsigned int) (next / 65536) % 2048;

    next *= 1103515245;
    next += 12345;
    result <<= 10;
    result ^= (unsigned int) (next / 65536) % 1024;

    next *= 1103515245;
    next += 12345;
    result <<= 10;
    result ^= (unsigned int) (next / 65536) % 1024;

    *seed = next;

    return result;
}
```

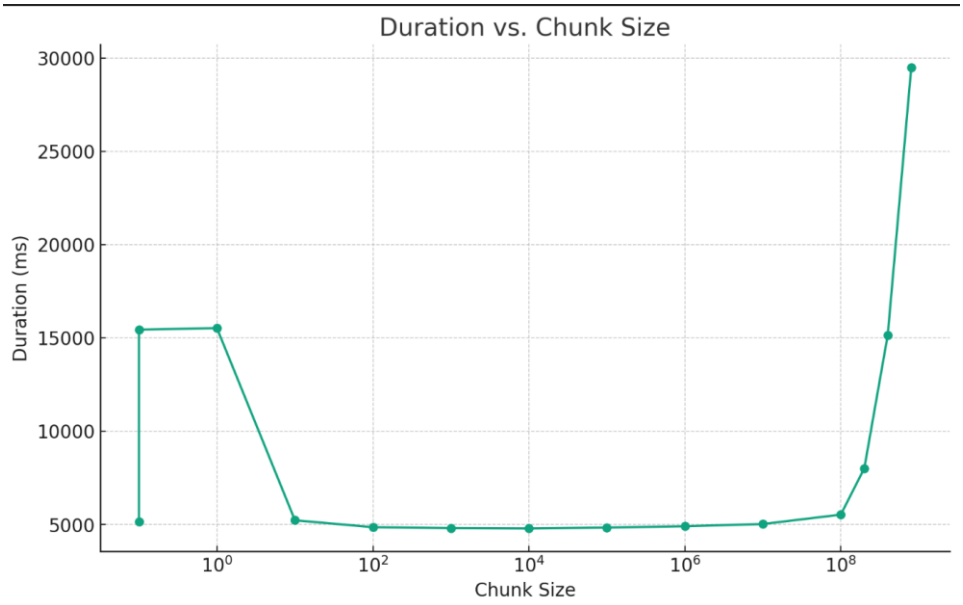
5. Тестирование

Подробнее о планировании. При распараллеливании цикла for, программа должна каким-то образом распределить итерации цикла между потоками. Для этого наборы итераций делятся на блоки – *chunks*, которые распределяются между потоками каким-то образом. При тестировании были использованы следующие виды: *static* и *dynamic*. Первый распределяет все блоки между потоками заранее, в то время как следующий – в процессе выполнения, что, в свою очередь, означает, что потоку будет «передан» очередной блок в том случае, если он обработал свой. Преимущество *dynamic* проявляется в том случае, если итерации цикла выполняются слишком неравномерно по времени.

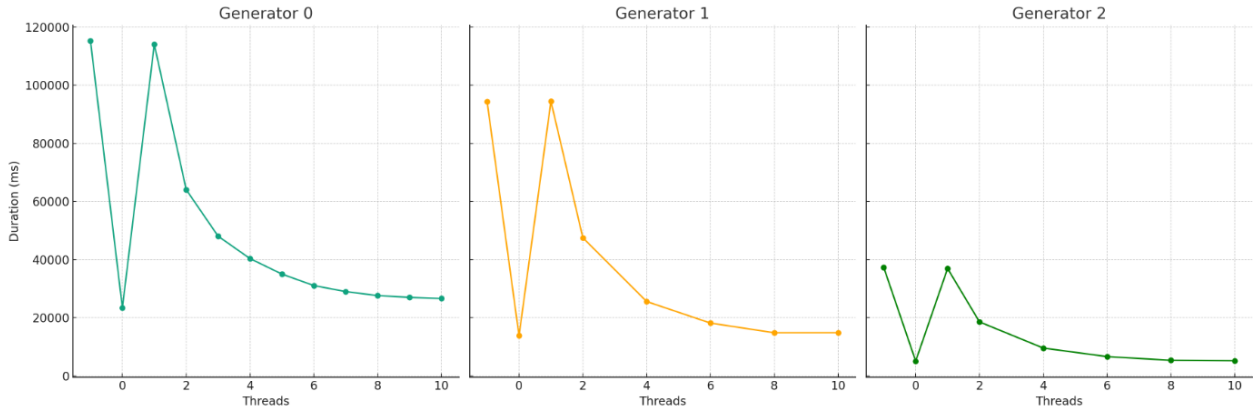
Стоит также отметить важную особенность. Если не указать параметр chunk_size, то static равномерно распределит итерации между потоками, а dynamic будет считать chunk_size равным одному, то есть запрашивать каждую следующую итерацию.

Логично предположить, что поскольку итерации цикла в методе Монте-Карло затрачивают время достаточно равномерно, никакого существенного выигрыша по времени получить от различных параметров планирования не получится. Так это или нет далее рассмотрим непосредственно по результатам тестирования.

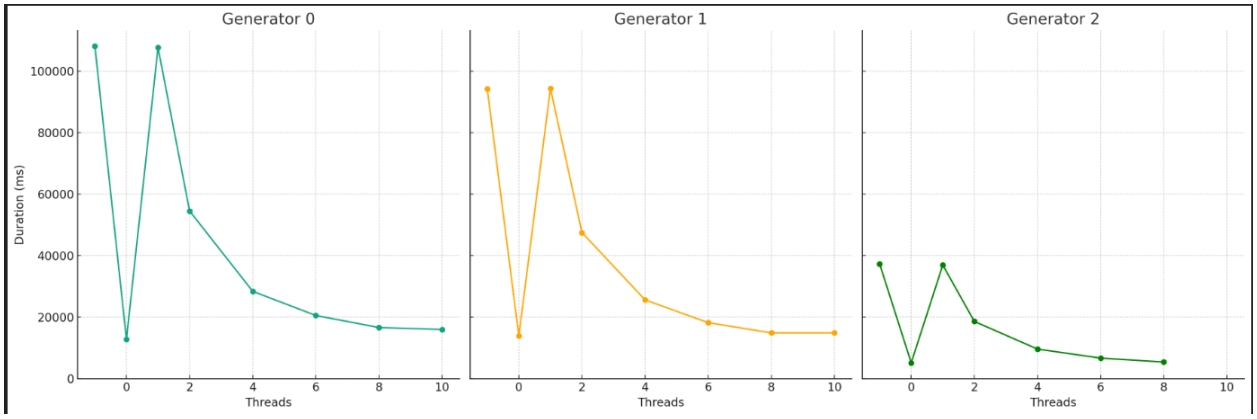
Для каждого из набора параметров было сделано 5-10 запусков, а затем получено среднее значение. Именно эти средние значения далее использованы для построения графиков.



На данном графике представлена зависимость скорости работы от указанного chunk_size (использован dynamic) (в том числе, учтён параметр -1, соответствующий не указанному параметру chunk_size) (тестировалось на 12 потоках). Во-первых очевидно, что при слишком маленьком chunk_size, расходы на распределение итераций становятся слишком велики, из-за чего в силу сбалансированности итераций мы, на самом деле, только теряем время. Далее можно заметить, что начиная с 10^8 время работы резко возрастает. Это связано с тем, что при слишком большом chunk_size потоки используются крайне неэффективно – одни «набирают» слишком много итераций, в то время как остальные проставивают.



На данных графиках приведена зависимость времени работы от количества потоков. Тестировалось при планировании *static* без *chunk_size* Можно отметить, что по мере увеличения количества потоков, прирост скорости становится всё меньше и меньше. Это связано с увеличением расходов на распараллеливание, а также, вероятно, вычислительными возможностями машины тестирования.



Здесь был произведён аналогичный тест, однако при планировании *dynamic* с *chunk_size = 10000*, который ранее в процессе тестирования был установлен как оптимальный. Прирост скорости за счёт динамического распределения итераций всё же заметен, а это значит, что некоторая неравномерность итераций всё-таки присутствует.

Вывод, который следует из сопоставления всех графиков выше – выбор оптимальных параметров планирования достаточно сильно зависит как от самой задачи, так и от входных данных, поскольку при разных входных

значениях один и тот же chunk_size может оказаться слишком большим, или, наоборот, оптимальным.