Лабораторная работа №5

Оптимизация по скорости

Инструментарий и требования к работе

Допустимые языки	C	C++
Стандарты / версии	C17	C++20
Требования для всех работ	Правила оформления и написания работ	

Цель работы: применение полученных знаний для оптимизации скорости программного обеспечения.

Инструментарий и требования к работе: работа выполняется на C/C++ (C11 и новее / C++20). Используется стандарт **OpenMP 4**. В отчёте указать язык и компилятор, на котором вы работали у себя локально.

Описание работы

Необходимо написать программу, решающую представленную задачу.

В процессе решения необходимо подобрать подходящий генератор псевдослучайных чисел и оптимизировать время выполнения алгоритма, в том числе с использованием многопоточности.

Необходимо написать **3 варианта кода**:

1. однопоточная реализация (без OpenMP);
2. многопоточная реализация с автоматическим распределением работы между потоками;
3. многопоточная реализация, где OpenMP используется только для создания потоков, но не распределения работы между ними.

Помимо написания программы, необходимо провести замеры времени работы на вашем компьютере и на сервере, привести графики времени работы программы (некоторые графики из следующих подпунктов можно объединить в один):

1. при различных значениях числа потоков при одинаковом параметре **`schedule`*** (с одинаковым **`chunk_size`**);
2. при одинаковом значении числа потоков при различных параметрах **`schedule`*** (с **`chunk_size`**);
3. при оптимальной конфигурации **`schedule`** (определённой ранее) от числа потоков;
4. графики для аналогичного распределения работы по потокам, но реализованного вручную (вариант кода №3, описанный выше).

* **`schedule(kind[, chunk_size])`** – kind принимает значение **`static`** или **`dynamic`**, **`chunk_size`** – 1 и несколько других целесообразных значений (5+)

Для минимизации влияния степени загруженности процессора другими процессами, время должно усредняться по нескольким запускам.

Время в программе нужно измерять при помощи **`omp_get_wtime()`** (раздел 3.3). Время следует измерять, конечно же, при сборке с оптимизациями (конфигурация Release).

Про **`schedule`** также можно почитать в спецификации (разделы 2.4.1 и Appendix D).

Задача

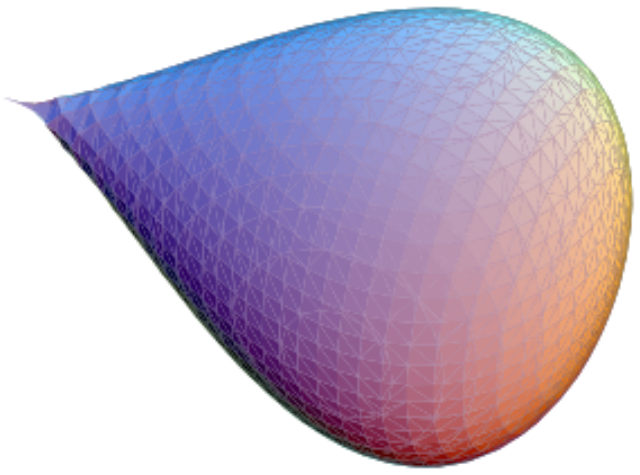
Расчет объёма трёхмерной фигуры методом Монте-Карло (не квази-Монте-Карло). В выходной файл записывается результат расчётов в формате **“`%g\n`”**.

В файле **hit.h** заданы прототипы функций. Функция **hit_test** отвечает, попадает ли точка (входные координаты) в фигуру или нет. Функция **get_axis_range** позволяет получить массив из 6 значений: **x_min, x_max, y_min, y_max, z_min, z_max**, которые определяют область пространства, в которой целиком содержится фигура. Это статический массив, а не динамическое выделение памяти. Изменять файл **hit.h** нельзя.

Для тестирования локально/на GH необходимо реализовать эти функции самостоятельно в файле **hit.cpp / hit.c** на основании фигуры для тестирования. При проверке работы будет использоваться другой файл **hit.cpp / hit.c**.

Файл, содержащий функцию **main** должен называться **main.cpp / main.c**.

Фигура для тестирования: piriform, параметр a=2.



Количество используемых потоков и время работы программы должны выводиться в поток вывода (stdout) в формате C: **“Time (%i thread(s)): %g ms\n”.**

Если openmp не используется, то выводить 0 в качества числа потоков. Если указано использовать число потоков по умолчанию – выводить реально используемое число потоков.

Время работы – без учета времени на парсинг аргументов и вывод результата.

Аргументы программе передаются через командную строку (порядок аргументов может быть произвольным):

Аргумент	**Комментарий**
`--input <name>`	Входной файл, содержащий число точек, используемых в методе Монте-Карло.
`--output <name>`	Выходной файл, в который записан ответ.
`--realization {1|2|3}`	Вариант кода (см. стр. 1)
`[--threads <num_threads>]`	Необязательный параметр, задающий количество потоков для реализаций 2 и 3, натуральное значение. При отсутствии следует использовать значение по умолчанию OpenMP (не хардкодить какое-то значение).
`[--kind {static|dynamic}]`	Используемый тип планирования: **`static`** или **`dynamic`** Необязательный параметр. Если не указан, то использовать оптимальный вариант.
`<--chunk_size [number]>`	Используемый размер блока итераций (натуральное значение). Необязательный параметр. Если не указан, то использовать оптимальный вариант для используемого **`kind`**.
Добавлять свои аргументы не запрещено, но тесты на GH должны проходить и не ломаться.	

Примеры использования (без input и output)

Пример запуска (без явного указания input, output и имени исполняемого файла)	**Комментарий (запускается…)**
`--realization 1`	простая реализация алгоритма без распараллеливания.
`--realization 2` `--threads 8`	вариант кода №2, использующий 8 потоков, с оптимальным *на сервере* значением **`kind`** и **`chunk_size`**

`--realization 2`
`--threads 8`
`--kind static`
	вариант кода №2, использующий 8 потоков, с оптимальным *на сервере* значением **`chunk_size`** для **`kind`** `static`

`--realization 2`
`--threads 8`
`--kind static`
`--chunk_size 100`
	вариант кода №2, использующий 8 потоков, с **`kind`** `static` и **`chunk_size`** `100`

`--realization 3`
`--threads 8`
`--kind static`
`--chunk_size 100`
	вариант кода №3, использующий 8 потоков, с **`kind`** `static` и **`chunk_size`** `100`

`--realization 3`	вариант кода №3, использующий количество потоков по умолчанию для OpenMP и оптимальное *на сервере* распределение работы между потоками

Важно

1. Будет оцениваться как правильность результата, так и скорость работы.
2. Значения по умолчанию для **`--kind`** и **`--chunk_size`** должны быть оптимальные ***для сервера***. Именно по ним будет оцениваться скорость работы. Для варианта кода №2 **`schedule`** должен быть задан явно.

3. Использовать директиву OpenMP `reduction` вместе с директивой OpenMP `for` нельзя (работа не принимается).
4. Выбор генератора псевдослучайных чисел – тоже часть работы. Выбор должен быть обоснован, подходить для поставленной задачи. (И будет сильно влиять на скорость работы, а значит и баллы.)
5. В отчёте должна быть зафиксирована модель CPU, на котором локально проводилось тестирование. “6-Core Intel Core i7” – это не точная модель. Возможный вариант: “AMD Ryzen 7 4700U (8 ядер, 8 потоков)”.
6. Все оси на графиках должны быть подписаны.
7. В разделе, где приводятся результаты тестирования, должны быть указаны тестовые данные (N).
8. Ещё раз: `get_axis_range` возвращает указатель на статический массив, а не на динамически выделенную память…
9. **В процессе написания кода *очень* рекомендуется вспомнить рассмотренные в курсе особенности исполнения программ современным железом (типы данных, кэширование, суперскалярность, …) и применить эти знания для написания быстро работающей реализации.**

Полезное

Компиляция

Для включения OpenMP нужно указать ключ компиляции.

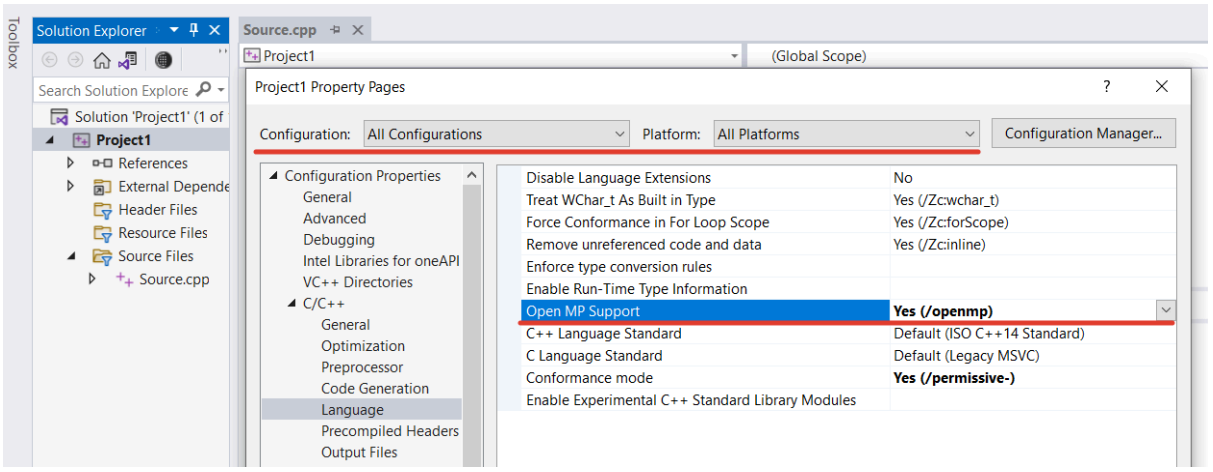
Компиляция через командную строку

Компилятор	**Ключ компиляции**
msvc (компилятор от Microsoft Visual Studio)	`/openmp`
gcc и clang	`-fopenmp`

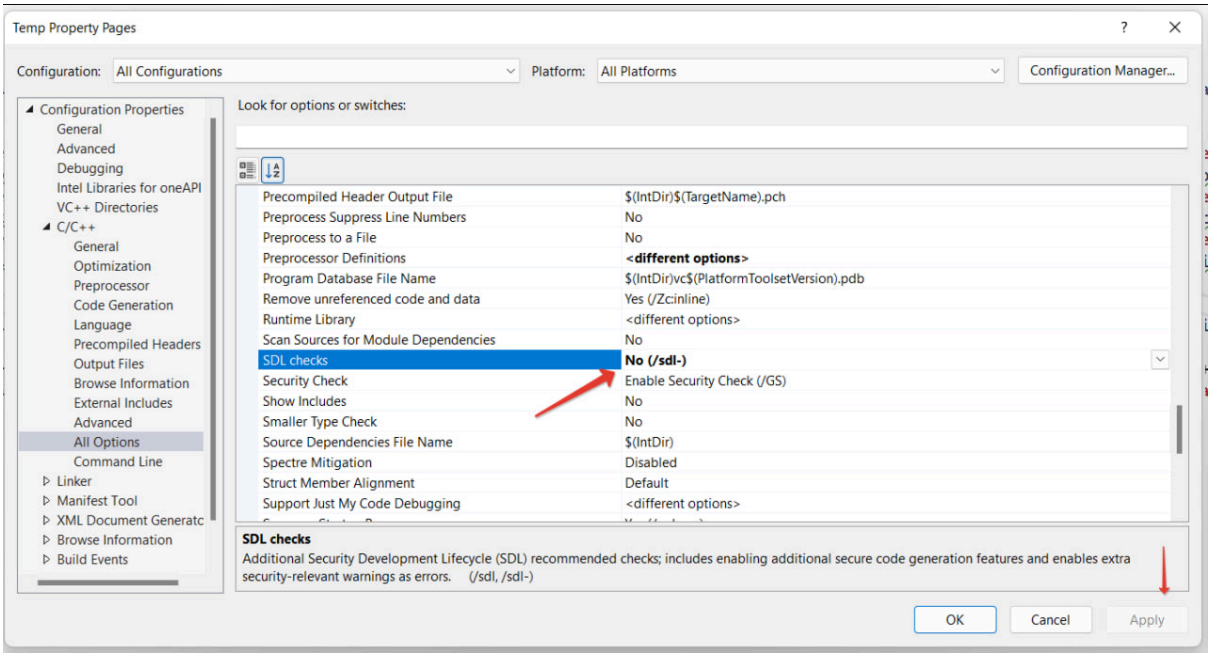
Примечание: clang, который установлен на MacOS по умолчанию, может не содержать OpenMP. Нужно либо установить полный clang, либо поставить gcc. Подробнее можно посмотреть здесь: https://www.programmersought.com/article/93289356924/

Visual Studio

Свойства проекта (ПКМ по проекту в обозревателе проектов) - C/C++ - Language - OpenMP support - Yes.



Дополнительно стоит отключить SDL check, чтобы “немодные” с точки зрения MSVC функции по типу fopen можно было использовать.



CMakeLists

```
OPTION (USE_OpenMP "Use OpenMP" ON)
IF(USE_OpenMP)
    FIND_PACKAGE(OpenMP)
    IF(OPENMP_FOUND)
        SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
        SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
    ENDIF()
ENDIF()
```

Для владельцев Mac на ARM64 также может помочь **`set(CMAKE_OSX_ARCHITECTURES x86_64)`**