Fluid-simulation-parallel

Отчёт

Парметры тестов

1. Вычисления производилось до достижения 500 тика (то есть $T = 500$).
2. Все запуски производились с одинаковым значением параметра field[][].
3. Замеры времени осуществлялись при помощи библиотеки chrono.

1. Результаты исходного алгоритма и их анализ

Прогон 1	Прогон 2	Прогон 3	Среднее время работы
187 сек	191 сек	185 сек	187.6 сек

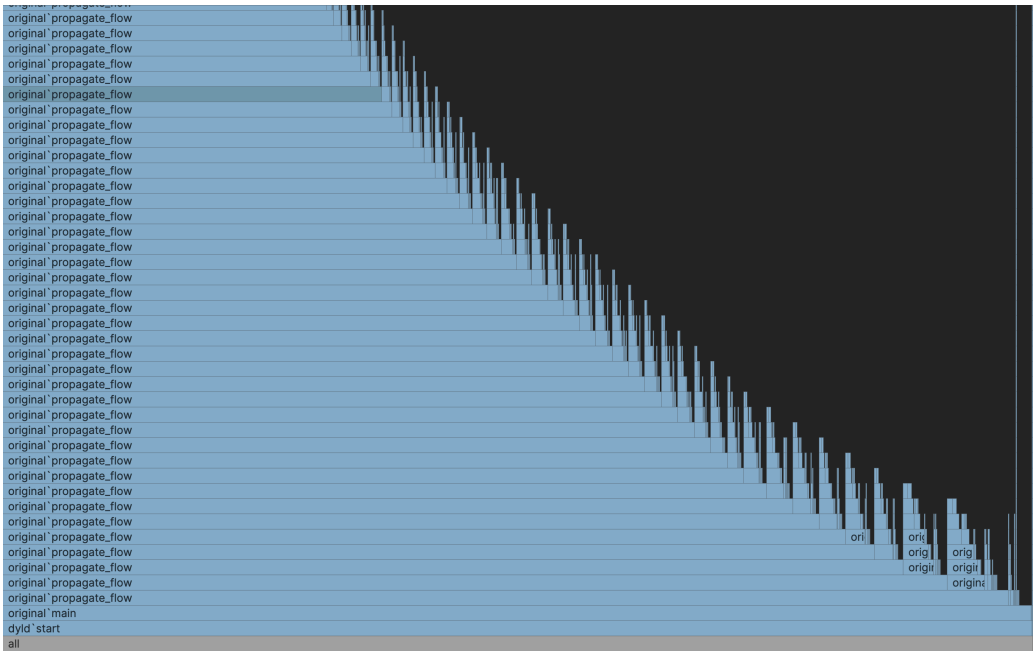

Рис. 1. Flamegraph оригинального алгоритма

Как выглядит Flamegraph

Оси:

- *По горизонтали:* Каждая полоса (бар) отражает совокупное время, проведенное в данной функции и ее вызовах.
- *По вертикали:* Стек вызовов функций.

Из него видно, что в каждом вызове функции `propagate_flow()` заметная часть времени (на графе это иглы) уходит на обработку вызова `VectoreField::get()`. В сумме это 3% от вызова `propagate_flow()`.

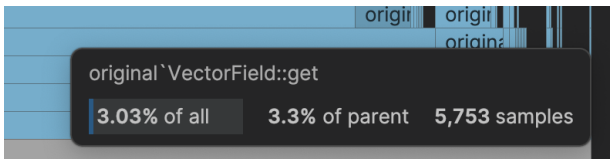

Рис. 2. Затраты на VectoreField::get()

Также из графа видно, что основную часть времени работы алгоритма составляет функция propagate_flow(). И поэтому её надо распараллеливать.

2. Оптимизации без распараллеливания

Был изменён метод VectoreField::get(), изначально он выглядел так:

```cpp
// Изначальный код
Fixed &get(int x, int y, int dx, int dy) {
    size_t i = ranges::find(deltas, pair(dx, dy)) - deltas.begin();
    return v[x][y][i];
}
```

Подбор параметра через `ranges::find()` очень неэффективный, поэтому я переписал эту часть кода через `switch(...) case: `, который работает намного быстрее.

```cpp
// Оптимизированный код
Fixed &get(int x, int y, int dx, int dy) {
     int i = 2*dx + dy;
     switch (i) {
         case 1:
             return v[x][y][3];
         case 2:
             return v[x][y][1];
         case -1:
             return v[x][y][2];
         default:
             return v[x][y][0];
     }
}
```

Результаты замеров:

Прогон 1	Прогон 2	Прогон 3	Среднее время работы
67 сек	74 сек	72 сек	71 сек

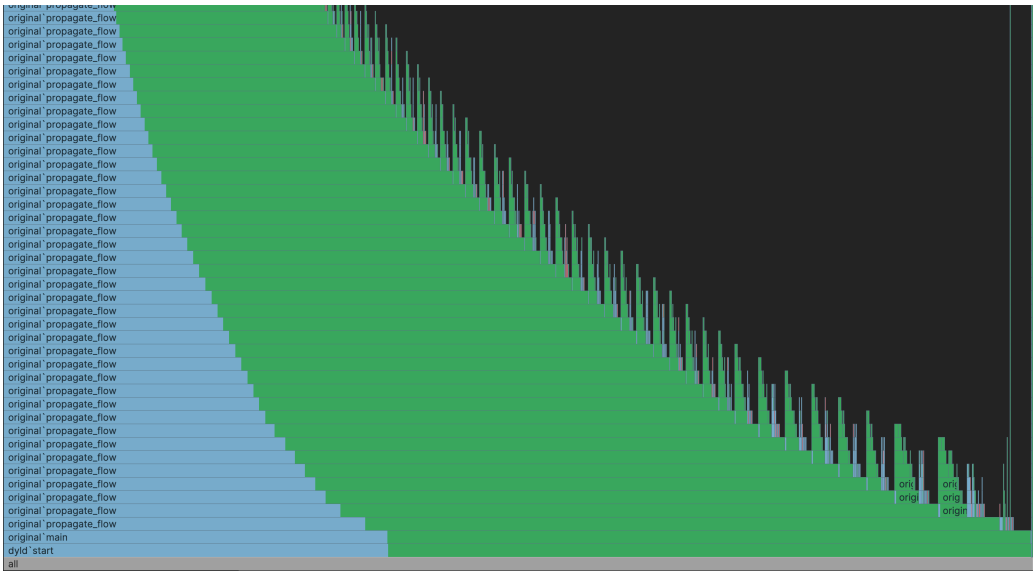

Рис. 3. Flamegraph после оптимизации

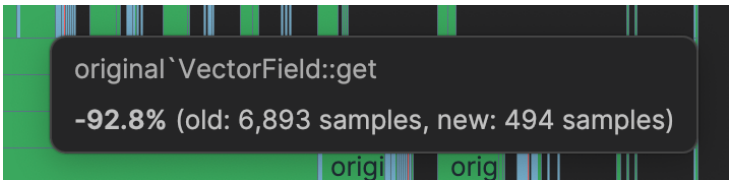

Рис. 4. Изменения затрат на оптимизированный VectoreField::get()

В итоге мы получили ускорение работы программы в $\frac{187.6}{71}$ = **2.6 раза**.

3. Многопоточная оптимизация

Как описывалось в пункте 1, надо распараллелить `propagate_flow()`.

По сути эта функция представляет собой dfs-обход по матрице `field[][]` с целью поиска циклов, и затем прокручивания жидкости по этому циклу. Я, к сожалению, не смог придумать, как его распараллелить. И для меня до сих пор открыт вопрос, возможно ли это в данной ситуации.

Однако в программе были ещё место, которые можно распараллелить, но получив уже меньшую прибавку в скорости. Это `/*Recalculate p with kinetic energy.*/`

Для удобства работы с потоками я написал свой класс `thread_pool`, который получает задания и сам уже назначает их потокам.

Изначально этот код выглядел так:

```cpp
// Recalculate p with kinetic energy
for (size_t x = 0; x < N; ++x) {
    for (size_t y = 0; y < M; ++y) {
        // some logic
    }
}
```

Который я преобразовал в такой:

```cpp

void recalculate_p(size_t x, size_t y, Fixed& total_delta_p) {
    // some logic
}

// внутри main()
thread_pool tpool(thread_pool_size);
...
for (size_t x = 0; x < N; ++x) {
    for (size_t y = 0; y < M; ++y) {
        tpool.add_task(recalculate_p, x, y, std::ref(total_delta_p));
    }
}
tpool.wait_all();
```

Результаты замеров:

В тестах использовался thread_pool на 5 потоков.

Прогон 1	Прогон 2	Прогон 3	Среднее время работы
59 сек	56 сек	62 сек	59 сек

В итоге производительность по сравнению с **однопоточным** вариантом улучшилась в $\frac{71}{59}$ = **1.21 раза**.

Итог

- Алгоритмическое ускорение в **2.6 раза**.
- Многопоточное ускорение ещё в 1.21 раза или от изначального в **3.17 раза**.