Моделирование Game of life

Выполнил: Колпаков Георгий

Реализация

Реализация модели протокола Game of life выполнена на языке Java и представляет из себя gradle проект и включает в себя 12 классов, из которых 5 используются только для юнит-тестирования.
Интерфейс GameOfLife содержит один метод и ожидает на вход имя файла с количеством ходов, размером поля и изначальной расстановкой игры.
GameOfLifeImpl имплментирует интерфейс GameOfLife, он содержит в себе 3 поля:

объект класса CellInputReader, использующийся для преобразования изначального поля игры из файла во внутреннее представление поля

объект класса GameOfLifeFieldTransformer, использующийся для запуска хода игры

объект класса GameOfLifeWriter, генерирующий выходное поле игры как список строк.

Классом представляющим поле игры является GameOfLifeField.
Он содержит статическое поле масок, использующихся для вычислений, два поля игры(изначальное поле и преобразованное поле), представленных как одинарный массив long'ов, размер(ширина) и границу поля.
Поле игры(массив long'ов) является массивом массивов битов(long используется как массив битов). Элемент массива long'ов под индексом I является набором элементов поля в строке (I % количество элементов в строке) между столбцами (I / 64 * 64) и столбцами (I/64 * 64 + 64).
Следующая таблица наглядно демонстрирует раскладку значений поля gameoffield по массиву field.

0… 63	64… 127	...	64*k… 64*(k+ 1)-1	...
field[0]	field[size]		field[k * size]	
...				
field[size - 1]	field[2 * size - 1]		field[(k + 1) * size - 1]	

Это представление является наиболее эффективным по памяти, что важно по 2 причинам: во-первых, это позволяет сильно сэкономить на необходимой памяти для работы приложения, и запускать его даже для очень больших полей, во-вторых это позволит процессору больше значений хранить в кэше, а значит реже обращаться в оперативную память за новыми участками поля.

Непосредственные вычисления нового поля проводятся в классе GameOfFieldTransformer, в методе transform он разделяет работу над полем между указанным в конструкторе количеством потоков: поле по вертикали разбиается на равные части, кратные количество битов в long, и каждый поток будет вычислять свой кусок поля независимо от других.
Вычисление нового поля проводится 2 способами, первый способ для клеток,

расположенных, либо на границе поле(первая и последняя строки, первый и последний столбцы), либо на границе двух long’ов(поля в столбцах I, для которых I%64 == 0 и I%64 == 63). Этот способ просто вычислияет сумму 8 значений, окружающих вычисляемую клетку, и в соотвествии с правилами игры возвращает новое значение для вычисляемой клетки. Для всех остальных клеток берется long, в котором содержится вычисляемая клетка и берутся long’и выше и ниже её, на каждый из этих long’ов накладывается маска из 3 битов, сдвинутых влево на (61 – позиция вычиляемого бита в его long’е) и у полученных long’ов считается число бит, такая оптимизация(использование bitcount’а для полей, рядом с вычисляемоей клеткой, вместо вычисления клеток по одной) позволяет использовать intrinsic’и jvm, связанные с подсчетом битов в примитивных типов и ускорять процесс.

Вычисления производятся последовательно для подстолбцов шириной 64, чтобы использовать предыдущую строку, хранящуюся в кэше для вычисления следующей и сократить обращения к оперативной памяти.

Ещё одна оптимизация, используемая в GameOfLifeFieldTransformer, это передача новому полю, генерируемому на каждом шаге, массивов полей, используемых в старом поле, только теперь то поле, которое использовалось для хранения вычисленных значений становится полем для хранения значений, над которыми будут производиться вычисления, а, соответственно, поле для хранения значений, вычисленных на предыдущем шаге становится полем для хранения вычисленных значений. Эта оптимизация позволяет не создавать на каждом шаге новый массив ля хранения новых вычисленных значений, что снижает необходимость в сборке мусора, что ускоряет вычисления. Могут возникнуть опасения, что использование старого поля может привести к тому, что мы можем получить некорректные значения, оставшиеся с прошлых вычислений, но в текущей имплементации, мы пересчитываем каждый элемент поля, поэтому все старые значений перезатрутся.

Запуск

Запуск проводился на ноутбуке со следующей конфигурацией:

Процессор Intel Core i7-6700HQ, 4 физических ядра, 8 виртуальных(режим HT), частота 2.6 ггц

ОЗУ: 16гб

ПЗУ: 512 Гб SSD

Операционная система windows 10 x64

JDK: 1.8.0_92

Измерение скорости работы.

Время выполнения для различных сэмплов представлено в следующей таблице:

Сэмпл	Количество шагов	Ширина поля	Время выполнения, с
Input1000.txt	100	1000	0.74
Input10000.txt	1000	10000	378.13

Можем увидеть, что рост времени выполнения близок к линейному с увеличением количества шагов и размера поля.

В следующей таблице представлена зависимость времени выполнения, от количества потоков.

Количество потоков	Время выполнения, с	Эффективность распараллеливания
1	1613,2	100%
2	889,5	90,7%
4	411,79	97,9%
8	378,13	53,3%

Видим, что используемое распараллеливание эффективно, пока используются только железные ядра, начало использование hyper-threading’а(переход от 4 к 8 потокам) сильно снижает эффективность распараллеливания.

На следующем графике представлена зависимость времени выполнения предыдущих 50 шагов от текущего шага в мс для сэмпла input10000.txt. Видм, что после первоначального снижения(после завершения всех инициализаций и проведения компиляторных оптимизаций) время выполнения выходит на плато, что означает, что мы не имеем тяжелых сборок мусора, значит наша оптимизация с переиспользованием полей сработала.

Зависимость времени выполнения предыдущих 50 шагов от шага, мс