Попытка ускорить двоичный поиск. Хорошо известно, что двоичный поиск не является быстрым. Для точечных запросов хеш-таблицы выполняются намного быстрее, идеально обращаясь к O (1). И даже когда нам нужны запросы диапазона. N-арные структуры поиска, такие как B-Trees, намного быстрее, чем двоичный поиск или деревья двоичного поиска. Тем не менее, есть определенный шарм для двоичного поиска. Во-первых, его легко реализовать, и, во-вторых, он не нуждается в дополнительной памяти, но может напрямую работать с отсортированными данными. И иногда данные сортируются в любом случае, поэтому мы получаем бинарный поиск более или менее бесплатно. И для чего-то мы получаем бесплатное O (log n) время поиска не так уж плохо. Вопрос только в том, как мы можем эффективно его реализовать. Конечно, есть текстовый способ реализации бинарного поиска, как объясняется, например, в Википедии. Полученный код выглядит следующим образом:

**while** (lower!=upper) {

**unsigned** middle=lower+((upper-lower)/2);

**unsigned** v=data[middle];

**if** (v==needle) {

**return** middle;

} **else** **if** (v<needle) {

lower=middle+1;

} **else** {

upper=middle;

}

}

**return** notFound;

Не особенно сложно, но насколько это быстро? Мы провели эксперименты, где мы выполнили 1,000,000 случайных поисков для 32-битных целых чисел в наборах данных различного размера и измерили производительность на Broadwell i7-5500U (все эксперименты с использованием -O3):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| set size | 103 | 105 | 107 | 109 |
| classic | 65 ms | 119 ms | 362 ms | 702 ms |

Когда мы внимательно смотрим, мы замечаем, что время выполнения растет немного быстрее, чем предполагает асимптотическая сложность O (log n). Или, другими словами, константы для поиска в наборе данных 4 ГБ кажутся примерно на 4 раз выше, чем при поиске в наборе данных 4 КБ. Это неудивительно, так как 4 Кбайт самого маленького набора легко вписывается в кеш процессора, в то время как 4 ГБ самого большого набора явно превышают размер кеша. Удивительно то, что это всего лишь фактор 4!

На производительность двоичного поиска в значительной степени влияют два эффекта: во-первых, есть недостатки в кэше, так как каждый поиск обращается к элементам O (log n), которые часто не входят в кеш для больших наборов данных. И, во-вторых, есть отраслевые неверные предсказания, поскольку у сравнения v игла есть приблизительно 50%-й шанс того, чтобы быть верным, который приводит к неэффективному выполнению. Для маленьких наборов данных промахи отделения доминируют, в то время как для больших наборов данных отделение отсутствует, несколько скрыты временем ожидания памяти. Поэтому замедление только 4.

Теперь вопрос в том, можем ли мы сделать лучше? Обычно люди пытаются ускорить бинарный поиск, уменьшая промахи филиалов. Существует интересный разговор о бинарном поиске, менее ошибочный, который описывает, как реализовать бинарный поиск без сложной для прогнозирования ветви. Немного ироничный пример кода в этом разговоре неверен, вопреки его названию, но идея верна. Скорректированный код выглядит следующим образом:

**while** (auto\_t half=n/2) {

**auto** middle=lower+half;

lower=((\*middle)<=needle)?middle:lower;

n-=half;

}

**return** ((\*lower)==needle)?lower:notFound;

Код не только достаточно короткий, тело цикла скомпилировано gcc полностью без ассемблерного кода, используя условное перемещение. Результирующая производительность показана ниже:



Это несколько неожиданный результат. Для небольших наборов данных версия без ветвей действительно быстрее, но для самого большого набора данных поиск значительно медленнее. И это после ручной настройки сгенерированного кода, из коробки понадобилось 920 мс для самого большого набора.

Ускорение для небольших размеров данных легко объяснить (отсутствие неверных предсказаний ветвления), но откуда происходит замедление для самого большого набора данных? Мы получаем ключ, компилируя тот же код с clang, который использует ветвь вместо условного перемещения внутри цикла. С этим мы получаем



Который виртуально идентичен производительности реализации текстового блока. По-видимому, условный ход хорош, чтобы избежать неправильных предсказаний отрасли, но не может скрывать латентность памяти, равно как и регулярную реализацию.

Таким образом, хотя мы обнаружили более быструю реализацию для небольших наборов данных, вопрос остается открытым, если мы можем сделать лучше для больших наборов (что может быть общим в базах данных). Мы попытались использовать тройной поиск вместо двоичного поиска, аргумент, что, хотя тройной поиск обращается к большему количеству мест памяти, чем двоичный поиск, задержка памяти может быть скрыта путем параллельного доступа, а число итераций меньше. Существует много способов реализации тройного поиска, все с различными компромиссами. Ниже приведена одна альтернатива, которая вызывает параллельные обращения к памяти:

**while** (**uintptr\_t** step=n/3) {

**auto** t1=lower+step,t2=lower+(step<<1);

**unsigned** cmp=((\*t1)<=needle)+((\*t2)<=needle);

**if** (cmp==2) {

n-=t2-lower; lower=t2;

} **else** **if** (cmp) {

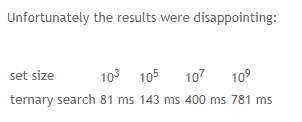
n-=t1-lower; lower=t1;

} **else** {

n=t1-lower;

}

}



Дополнительный доступ к памяти просто не стоит. В худшем случае у нас есть два пропущенных пропущенных кэша на итерацию (даже если они выполняются параллельно), а преимущество расширенного разветвления слишком мало.

Конечно, есть способы ускорить поиск в больших массивах данных. Например, путем хранения данных в оптимизированном к кешированию B-Tree, а не сортированном векторе. Но это заставляет нас отказаться от нашей первоначальной миссии, а именно использовать сортировку, доступную в данных в любом случае. Для малых и средних наборов мы показали улучшения в отношении подхода к текстовым книгам здесь, если у кого-то есть хорошая идея для больших наборов, я был бы рад услышать об этом в комментариях.

Обновление. Как отметил Марсин в комментариях, можно использовать SIMD-версию бинарного поиска для одновременного выполнения 4 запросов. Я использовал код из раздела 5.5.4 тезисов, здесь немного показывается. И действительно, мы получаем хорошую производительность (лучший из 10 запусков):



Обратите внимание, что для многих размеров данных производительность находится посередине между версией текстовой книги и условно-перемещаемой версией, независимо от того, какая из двух на самом деле быстрее. Непонятные. То, что также озадачивает, заключается в том, что дисперсия времени выполнения намного выше, чем для версий, отличных от SIMD. Для самого большого набора данных я обычно видел время работы около 780 мс, с пиками до 950 мс. У меня нет объяснений.

Если, как предложил Марсин в комментарии ниже, мы заменим код ручной сборки

**int** cmpval0 = data\_shifted[result\_vector[i + 0]];

**int** cmpval1 = data\_shifted[result\_vector[i + 1]];

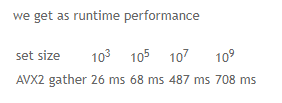
**int** cmpval2 = data\_shifted[result\_vector[i + 2]];

**int** cmpval3 = data\_shifted[result\_vector[i + 3]];

\_\_m128i xm\_cmpvalvec = \_mm\_set\_epi32(cmpval3, cmpval2, cmpval1, cmpval0);

with the AVX2 gather instruction

\_\_m128i xm\_cmpvalvec = \_mm\_i32gather\_epi32(data\_shifted,\_mm\_load\_si128((\_\_m128i\*)(result\_vector)),4);



На первый взгляд производительность AVX2-сборника идентична ручной сборке. И действительно, у меня был точно такой результат на предыдущих экспериментах по сбору. Однако проблема с отклонениями, по-видимому, исчезла при сборе AVX2, среда выполнения теперь относительно стабильна во всех прогонах.