Параллельные и распределенные вычисления. Задание 2 “Метод k-средних”.

Старченко Владимир

Основная часть

Поставленная задача

Реализуйте параллельную версию метода k-средних на C++ с использованием OpenMP, взяв за основу готовый последовательный вариант программы.

Выполните тестирование параллельной программы с помощью предоставленного сервиса. Постарайтесь достигнуть максимальной производительности (ускорения и эффективности) на всех тестовых наборах данных. Обоснуйте выбранную стратегию распараллеливания и выполненные оптимизации.

Обоснование решения

Необходимо параллелить циклы или рекурсивные вызовы. Рекурсии здесь нет, остаются циклы. Для большего увеличения производительности необходимо следить, чтобы разные нити не писали в общие участки памяти.

Описание реализации

Я посмотрел на циклы, которые есть в программе:

- Основной цикл программы. Его паралеллить явно не нужно, так как каждая его следующая итерация зависит от предыдущей.

```
while (!converged) {
  converged = true;
```

- Следующий цикл. Итерации независимы. Довольно большие по размеру: на каждой итерации просматриваются все точки. Итераций много: столько же, сколько точек. Значит, его можно и имеет смысл параллелить.

```
converged = true;
size_t nearest_cluster;

for (size_t i = 0; i < data_size; ++i) {
```

  Самое простое и быстрое изменение - добавить прагму:

```
#pragma omp parallel for private(nearest_cluster) reduction(& : converged)
for (size_t i = 0; i < data_size; ++i) {
```

  Это решение дает результат 0.4.

- Если посмотреть на остальные циклы, на тех данных, на которых тестируется наша программа, не имеет смысла сильно параллелить. Если в цикле порядка K итераций и тело - пара арифметических операций, запуск нескольких потоков может не улучшить, а даже ухучшить производительность. Такой результат я, например, получил в самом начале выполнения дз. Конкретно в том случае паралеллился цикл в функции *FindNearestCentroid*.

Результаты

https://everest.distcomp.org/jobs/59e74e38300000049d143169

Анализ решения

Чтобы получить больше 0.4, было необходимо было изменять боле чем 1 прагму.

Основная проблема в том, что разные потоки изменяют одни и те же данные. Чтобы улучшить результаты - разобьем данные всех потоков на непересекающиеся блоки. Каждый поток будет работать со своими данными, при этом не претендуя на чужие. Данне будут хорошо кешироваться.

Для этого я завел потокам копии переменных *clusters*, *centroids*, *clusters_sizes*. Они по отдельность насчитывают значения от своих блоков, и после завершения работы потоков результаты агрегируются в изначальные переменные.

В итоге, осталась не распаралелленая часть. Та, где данные агрегируются, и та, где нет смысла (при тех тестовых размерностях) параллелить.

Хочется заметить, что выделение памяти для локальных переменных потоков, в соответствии с проведенными замерами, лучше выделять не в потоке и не в цикле, а изначально до цикла while.

Контрольные вопросы

Вопрос 1.

Использованы ли вами все возможности для распараллеливания и оптимизации, которые могли бы дать заметный вклад в производительность? Если нет, то что еще можно было бы сделать? (вес 0.3)

Распараллеливание

Была возможность распараллелить ещё два нижних цикла for в основним цикле while.

Их имело бы смысл трогать (и это дало бы вклад в производителность) при достаточно больших значениях параметра K (количество классов объектов). Однако стоит ещё учитывать, что при изменении последнего цикла изменилось бы последовательность вызова раздомайзера. Тогда результаты могли бы отличаться от запуска к запуску.

Явно не стоит трогать агрегацию результатов изначального распараллеливания. Потоки бы писали в одни и те же ячейки памяти. Плюс к тому, эта фаза значительно меньше по размеру, чем вычисления, проводящиеся до этого.

Оптимизация

Здесь можно сделать очень много.

Оптимизация формата ввода/вывода (использовать raw бинарный формат). Очень много тратится на чтение и преобразование plain/text во входные данные. Если бы использовался mmap, можно было бы напрямую подгружать файл как страничку в память.

Начальные точки. Сейчас используются случайные точки в качестве изначальных центроидов. Есть некоторые оптимизации, помогающие выбирать более удачные точки (например, оптимизация kmeans++ отличается только алгоритмом выбора изначальных точек).

Вопрос 2.

Как зависят ускорение и эффективность от числа потоков? Что происходит при числе потоков большем 12? Почему? (вес 0.3)

Чем польше число потоков тем больше ускорение. Так происходит при увеличении количества потоков от нуля до 12 и от 13 до 24. Рост не линейный, так как при увеличении количества потоков растет и время на дополнительные расходы, например, копирование private переменных и другие. Также есть нераспаралеленные фрагменты алгоритма (например, агрегирование на наждой итерации основного цикла), которые тоже не позволяют S равняться p.

Эффективность падает, так как больше копирований и синхронизаций. Это снижает эффективность.

Всё запускается на 12-ти ядерной машине. Значит, при использовании 13 потоков, начинает использоваться гипертрединг. 2 потока (которые выполняются в одном процессоре) начинают конкурировать и ждать друг друга. В итоге время показаели заметно падают. При этом 24 потока всё таки немного быстрее работают, чем 12. Часть вычислений работают на обоих сопроцессорах одновременно, и общее время уменьшается.

Вопрос 3.

Отличаются ли полученные ускорение и эффективность для различных наборов данных (при одинаковом числе потоков)? Почему? (вес 0.4)

Различаются. Причем очень сильно. В данном алгоритме скорость нахождения центроидов зависит от многих параметров. От рандома. От выбора начальных точек. От данных.

Более того, оптимальность разбиения тоже зависит от начальных точек.

Kmeans ищет локальный минимум, и скорость схождения, очевидно, различается на разных наборах данных. Это происходит из-за того, что на разных данных, минимумы расположены по разному. Траектории, по которым точки центроидов сходятся к конечным положениям тоже разные. Количество итераций внешнего цикла while разное, значит и время сильно отличается. Объем нераспараллеленного кода тоже разный. А нераспараллеленный код влияет на эффективность и ускорение.