Многопоточный распределенный поиск в ширину с упорядоченными коммуникациями

А. В. Осипов, А. Н. Дарьин (Яндекс*, Москва),
А. А. Наумов (Т-Платформы, Москва)

Описывается параллельная реализация алгоритма поиска в ширину на графе, разработанная в компании Т-Платформы. Ключевой особенностью является оптимизированное внутреннее представление графа, позволяющее упорядочить коммуникации между вычислительными процессами и разделить выполнение на потоки внутри каждого из процессов. Приводится описание оптимизации по направлению и ее многопоточной имплементации. Также приведены результаты исследования производительности разработанной реализации.

Ключевые слова: распределенные вычисления, параллельные вычисления, графы, поиск в ширину.

Построение остовного дерева графа методом поиска в ширину [1] (Breadth-First Search — BFS) используется в качестве бенчмарка (теста производительности) в рейтинге Graph500 [2]. Спецификация бенчмарка, предоставленная Graph500, делит его на следующие шаги:

1) генерация списка ребер;
2) конвертация списка ребер во внутреннее представление (измеряется время, но на рейтинг не влияет);
3) генерация списка из 64 уникальных вершин с минимальной степенью 1 (без учета петель);
4) для каждой вершины из предыдущего списка:
   a) построение остовного дерева алгоритмом BFS с корнем в данной вершине (измеряется время, влияет на рейтинг);

*Исследование выполнено во время работы в компании Т-Платформы

б) валидация построенного остовного дерева;

5) расчет и вывод данных по производительности.

Распределенная версия алгоритма BFS не предполагает высокой вычислительной нагрузки на процессор. Основными факторами, влияющими на производительность, являются задержка при доступе к оперативной памяти, а также запаздывание и пропускная способность коммуникационной сети. Референсная реализация не учитывает этих особенностей и, соответственно, работает весьма медленно. Ограничения референсной реализации вызваны прежде всего выбором внутреннего представления графа — CSR (Compressed Sparse-Row Graph).

CSR предназначен для компактного хранения распределенного графа. Распределение графа по процессам определяется номерами вершин — несколько старших битов номера вершины определяют ранг процесса, который становится владельцем всех ребер с данной вершиной. На данном процессе его локальные вершины хранятся в сокращенном формате (`local_id`) — биты, отвечающие за ранг процесса выкидываются. Все чужие вершины хранятся в формате `global_id`.

На каждом узле CSR представлен двумя структурами: списком указателей `rowstarts` и списком вершин `columns`. Для вершины с `local_id = i` в диапазоне `rowstarts[i]` ...`rowstarts[i+1]` массива `columns` хранятся `global_id` вершин, связанных с `i`.

Референсная реализация BFS на псевдокоде выглядит следующим образом:

Algorithm 1 Референсная реализация BFS

```
1: function ProcessQueue(R, C, Q)
2:     for all v ∈ Q do
3:         for e ← R[v], R[v + 1] do
4:             send v, local(C[e]) to owner(C[e]))
```

где `R` — массив `rowstarts`, `C` — массив `columns`, а `Q` — текущая очередь вершин, посещенных на предыдущем шаге (на старте алгоритма в ней находится корень остовного дерева).

Данный вариант алгоритма имеет несколько проблем: во-первых, малый размер сообщения при посылке (одно ребро), во-вторых, случайный шаблон коммуникации, в-третьих невозможность эффективно разделить выполнение алгоритма на потоки. Размер сообщения можно увеличить, используя промежуточные агрегационные буферы. Но из-за то-

го, что паттерн коммуникации случаен, количество агрегационных буферов должно быть не меньше количества процессов, что значительно увеличит накладные расходы.

Для регуляризации шаблона коммуникации предлагается раделить внутреннее представление на несколько частей — завести по паре массивов `rowstarts`, `columns` на каждую пару процессов. Такой подход позволяет обрабатывать разделы независимо друг от друга. Но для этого необходимо модифицировать формат хранения разделов: использование CSR вызывает неприемлемые накладные расходы на хранение массивов `rowstarts`, суммарный размер которых становится равен количеству вершин в графе. В модифицированном варианте `rowstarts` (`packed_rowstarts`) в каждом элементе хранятся как `local_id` вершины так и смещение внутри соответствующего массива `columns`. А, за счет того, что при таком подходе в `columns` достаточно хранить `local_id` вершины вместо `global_id`, расход памяти в среднем не увеличивается.

Запись на псевдокоде модифицированной реализации выглядит следующим образом:

Algorithm 2 Реализация BFS с «упакованным» CSR

```
1: function ProcessQueue(R, C, Q)
2:     for p ← 0, NP do
3:         for i ← 0, |V[p]| do
4:             if V[p][i] ∈ Q then
5:                 for e ← D[p][i], D[p][i + 1] do
6:                     send V[p][i], C[e] to p
```

где `V` — `local_id` вершин, `D` — `rowstarts`, `C` — `columns`, `Q` — текущая очередь вершин (битовая маска), `NP` — число процессов. Как видно из псевдокода, шаблон коммуникации становится регулярным, что, в свою очередь, позволяет использовать ограниченное количество буферов агрегации при отправке сообщений. Кроме того, разделы обрабатываются независимо друг от друга, что позволяет сделать обработку многопоточной.

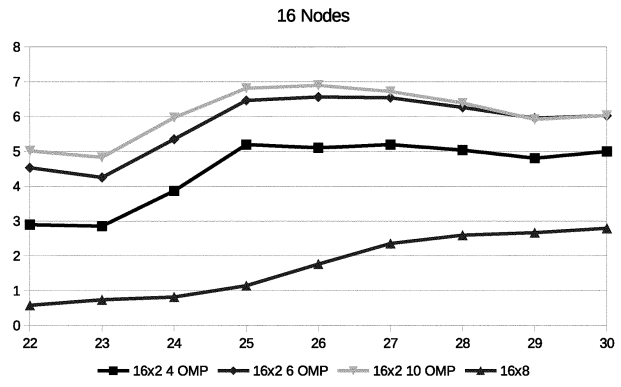
Разбиение на потоки производилось с помощью OpenMP. При разработке многопоточной версии алгоритма мы придерживались следующих принципов:

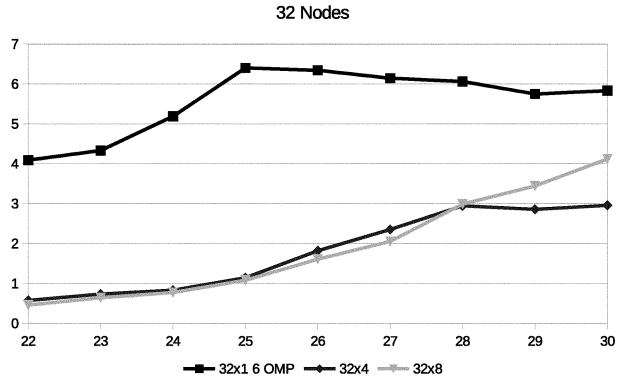
- всеми MPI коммуникациями занимается только главный поток;
- все полученные данные обрабатывает отдельный поток;

- все остальные потоки готовят данные для отправки;
- все коммуникации между потоками производятся через параллельные очереди (для этого использовалась библиотека Intel Thread Building Blocks);

Также значимый вклад в производительность теста вносит оптимизация по направлению [3]. Суть стандартного алгоритма заключается в следующем — выбирается вершина, посещенная на предыдущем шаге и по каждому ребру, связанному с ней проверяется, посещен ли второй конец. Через некоторое количество итераций (3–4) большая часть вершин графа оказывается посещена и большинство таких проверок проходит впустую. Оптимизация по направлению предлагает в определенный момент просто менять направление проверки, то есть выбирать еще непосещенные вершины и делать проверку по их ребрам. Кроме того, необязательно посылать сразу весь список ребер, связанных с выбранной вершиной — часто бывает достаточно проверить только одно ребро. Сортировка массивов `columns` по степеням вершин дополнительно повышает вероятность «угадать» с первой попытки.

Финальная версия алгоритма тестировалась на кластере Ангара-К1 с высокоскоростной коммуникационной сетью Ангара [4]. Сравнение проводилось между чисто MPI и MPI+OpenMP (многопоточной) версиями алгоритма. Результаты представлены на графиках:





Список литературы

[1] Moore E. F. The Shortest Path Through a Maze // Proceedings of the International Symposium on the Theory of Switching. — Harvard University Press, 1959. — P. 285–292.

[2] Graph500 Benchmark Specifications. — [Эл. ресурс]
URL: http://www.graph500.org/specifications

[3] Beamer S., Asanović K., Patterson D. Direction-optimizing breadth first search // Scientific Programming 21. — IOS Press, 2013. — P. 137–148.

[4] Агарков А. А., Исмагилов Т. Ф., Макагон Д. В., Семенов А. С., Симонов А. С. Результаты оценочного тестирования отечественной высокоскоростной коммуникационной сети Ангара // Суперкомпьютерные дни в России: Труды международной конференции (26–27 сентября 2016 г., г. Москва). — М.: Изд-во МГУ, 2016. — С. 626–639.