Параллельная высокопроизводительная обработка графов*

М.А. Черноскутов[1,2]

[1]Институт математики и механики им. Н.Н. Красовского УрО РАН,
[2]Уральский Федеральный Университет имени первого Президента России Б.Н. Ельцина

В работе описаны методы, позволяющие значительно ускорить работу параллельного поиска в ширину на графе. Основным препятствием для эффективного распараллеливания поиска в ширину на графе являются дисбаланс вычислительной нагрузки внутри отдельных вычислительных узлов, а также значительный объем передаваемых в конце каждой итерации алгоритма. Для устранения этих препятствий в данной статье предлагается два метода. Первый метод позволяет распределить нагрузку между отдельными OpenMP-потоками внутри одного узла. Второй метод позволяет, путем гибридизации обхода графа, добиться снижения общего объема передаваемых данных.

Ключевые слова: параллельные вычисления, обработка графов, балансировка нагрузки

1. Введение

Алгоритмы на графах применяются в различных научных и практических приложениях. Во многих случаях, большие размеры графов предполагают их параллельную обработку на многопроцессорных вычислительных системах [1]. Однако, эффективному распараллеливанию алгоритмов на графах препятствуют такие обстоятельства как интенсивный доступ к памяти и заранее неизвестное (в общем случае) распределение данных по узлам вычислительной системы, что делает подобные алгоритмы типичными представителями задач класса «data intensive» [2].

Интенсивный доступ к памяти является узким местом в практической реализации алгоритмов на графах, т.к. чаще всего, работа таких алгоритмов не сопряжена с большим количеством арифметических вычислений. Этот факт позволяет, с одной стороны, значительно снизить требования к производительности центральных процессоров, но, с другой стороны, повышаются требования к эффективности и скорости работы канала доступа к данным.

Неизвестное распределение данных также значительно осложняет эффективную реализацию параллельных алгоритмов на графах на многоузловых вычислительных системах. В общем случае, заранее неизвестно, на каком вычислительном узле расположены данные о той или иной вершине. Поэтому, возникает потребность в опросе других узлов на предмет наличия на них требуемых данных. Другой вариант — заранее распределять данные между узлами в соответствии с каким-либо правилом.

Объектом исследования данной работы является параллельный поиск в ширину на графах с неравномерным распределением степей вершин и малым диаметром (как правило, не более 10). Такие графы возникают, например, при анализе социальных сетей и приложений физического и математического моделирования [3]. Основной особенностью таких графов является очень малое количество вершин с большими степенями на фоне большого количества вершин со всего лишь несколькими инцидентными ребрами.

*Работа поддержана грантом РФФИ №14-07-00435. При выполнении работ использовался суперкомпьютер «Уран»

2. Параллельный поиск в ширину

Для распараллеливания поиска в ширину на графе используются синхронизированные по уровням алгоритмы. Как видно из названия, данные алгоритмы обрабатывают каждый уровень (или итерацию) алгоритма последовательно и независимо от других уровней. На практике это означает, что, в случае поиска в ширину, обработка уровня $N+1$ начнется только после того, как будет полностью завершена обработка уровня N. При этом вершины и ребра графа как на уровне N, так и на уровне $N+1$ могут обрабатываться параллельно.

На данный момент существует два наиболее распространенных версии синхронизированного по уровням поиска в ширину:

- прямой обход графа («top-down»);
- обратный обход графа [4] («bottom-up»).

Версия поиска в ширину с прямым обходом графа — «стандартный» вариант данного алгоритма. Этот вариант обхода предполагает, что активные на текущей итерации вершины будут помечать своих соседей. Псевдокод алгоритма параллельного поиска в ширину с прямым обходом графа представлен на рис. 1.

```
1  for each u in dist
2        dist[u] := -1
3  dist[s] := 0
4  level := 0
5  do
6        parallel for each vert in V.this_node
7              if dist[vert] = level
8                    for each neighb in vert.neighbors
9                          if neighb in V.this_node
10                               if dist[neighb] = -1
11                                     dist[neighb] := level + 1
12                                     pred[neighb] := vert
13                   else
14                         vert_batch_to_send.push(neighb)
15
16       send(vert_batch_to_send)
17       receive(vert_batch_to_receive)
18
19       parallel for each vert in vert_batch_to_receive
20             if dist[vert] = -1
21                   dist[vert] := level + 1
22                   pred[vert] := vert.pred
23       level++
24 while(!check_end())
```

Рис. 1. Псевдокод параллельного алгоритма поиска в ширину на графе с прямым обходом

В строках 1–4 проходит инициализация массива расстояний до корневой вершины. Далее расположен основной цикл поиска в ширину, который выполняется, пока в графе имеются непомеченные вершины. Сначала, в строках 6–14 происходит разметка тех вершин, которые расположены на данном вычислительном узле. Затем, в строках 16 и 17 выполняется, соответственно, точечная рассылка и прием сообщений, содержащих данные о вершинах, которые должны быть помечены на других узлах. Наконец, в строках 19–22 выполняется разметка вершин, данные о которых были получены путем приема сообщений. Выполнение итерации завершается увеличением номера текущего уровня на единицу в строке 23 и проверкой на наличие непомеченных вершин в строке 24.

Версия поиска в ширину с обратным обходом предполагает, что неактивные вершины будут просматривать свои списки соседей и будут помечаться только в том случае, если среди их соседей имеется активная на текущей итерации вершина. Псевдокод алгоритма параллельного поиска в ширину с обратным обходом графа представлен на рис. 2.

```
1  for each u in dist
2        dist[u] := -1
3  dist[s] := 0
4  level := 0
5  do
6        parallel for each vert in V.this_node
7              if dist[vert] = -1
8                    for each neighb in vert.neighbors
9                          if bitmap_current.neighb = 1
10                               dist[vert] := level + 1
11                               pred[vert] := neighb
12                               bitmap_next.vert := 1
13                               break
14
15       all_gather(bitmap_next)
16       swap(bitmap_current, bitmap_next)
17
18       level++
19 while(!check_end())
```

Рис. 2. Псевдокод параллельного алгоритма поиска в ширину на графе с обратным обходом

Аналогично алгоритму на рис. 1, в строках 1–4 выполняется инициализация поиска в ширину, а в строках 18 и 19 — обновление номера уровня и проверка завершения алгоритма. По иному построена процедура разметки вершин — в случае обратного обхода необходимо знать информацию обо всех активных на данной итерации вершинах. Удобнее всего это сделать с помощью использования битовой маски, длина которой (в битах) равна количеству вершин в графе. Таким образом, синхронизация данных на каждой итерации проходит путем обновления битовой маски с помощью коллективных коммуникаций (строки 15 и 16).

3. Оптимизация производительности параллельного поиска в ширину

Для ускорения параллельного алгоритма поиска в ширину на графе предлагается использование двух методов:

- гибридизация обхода графа;
- распределение вычислительной нагрузки между потоками.

3.1. Гибридизация обхода

Данный метод предполагает сочетание различных разновидностей параллельного алгоритма поиска в ширину для выполнения различных итераций алгоритма. В частности, «top-down» версия алгоритма характеризуется пиками вычислительной нагрузки и нагрузки на подсистему передачи данных на промежуточных итерациях алгоритма. В то же время, начальные и заключительные итерации в прямом обходе графа выполняются значительно быстрее и практически не нагружают сеть передачи данных. Несколько иная ситуация обстоит с «bottom-up» версией алгоритма вследствие использования коллективных операций синхронизации данных. В данном варианте алгоритма поиска в ширину, обмены данными на каждой итерации занимают примерно одинаковое время. Однако, разметка вершин на первых итерациях происходит гораздо дольше, чем на последующих.

Учитывая тот факт, что результат выполнения каждой из итераций алгоритма поиска в ширину, независимо от направления обхода, одинаков, то имеет смысл комбинировать различные варианты обхода графа для достижения максимальной производительности. В данной работе предлагается следующая схема:

- первые две итерации — «top-down»;

- следующие три итерации — «bottom-up»;
- все остальные итерации — «top-down».

3.2. Распределение нагрузки

При обработке графов с неравномерным распределением степеней вершин заранее неизвестно (в общем случае), какие вершины будет обрабатывать тот или иной вычислительный поток. Известным может быть лишь общее количество вершин, которые необходимо обработать в данном потоке. При этом общий объем нагрузки определяется не самим количеством вершин, а количеством инцидентных данным вершинам ребер. Это приводит к дисбалансу нагрузки между потоками, что приводит к большим накладным расходам при распараллеливании синхронизированного по уровням поиска в ширину.

Для устранения дисбаланса нагрузки предлагается перейти от просмотра массива вершин на каждой итерации алгоритма к просмотру массива ребер. Для этого массив ребер необходимо логически разделить на равные части размером max_edges элементов. Для того, чтобы каждый поток мог «знать», к какой вершине принадлежит тот или иной элемент в каждой из равных частей массива ребер, необходимо создать новый массив $part\_column$, в каждой ячейке которого будут храниться номера вершин, инцидентные первому элементу в каждой из частей массива ребер. Параллельная разметка массива $part\_column$ представлена на рис. 3.

```
1  parallel for each i in V.this_node
2        first := V.this_node[i]
3        last := V.this_node[i+1]
4        index := round_up(first/max_edges)
5        current := index*max_edges
6        while(current < last)
7              part_column[index] := i
8              current := current + max_edges
9              index++
```

Рис. 3. Псевдокод параллельного алгоритма разметки массива $part\_column$

Псевдокод новой версии цикла разметки вершин (см. строки 6–14 в алгоритме на рис. 1 и строки 6–13 на рис. 2) с использованием массива $part\_column$ для прямого варианта обхода графа приведен на рис. 4, для обратного варианта обхода графа — на рис. 5.

```
1  // preparation...
2  parallel for each i in part_column
3        first_edge := i*max_edges
4        last_edge := (i+1)*max_edges
5        curr_vert := part_column[i]
6        for each edge in [first_edge;last_edge)
7              if neighbors of curr_vert in [first_edge;last_edge)
8                    if dist[curr_vert] = level
9                          for each k in neighbors of curr_vert
10                               if dist[k] = -1
11                                     dist[k] := level + 1
12                                     pred[k] := curr_vert
13             curr_vert++
14 // data synchronization...
```

Рис. 4. Псевдокод цикла разметки вершин в алгоритме поиска в ширину на графе (прямой обход)

```
1  // preparation...
2  parallel for i in part_column
3        first_edge := i*max_edges
4        last_edge := (i+1)*max_edges
5        curr_vert := part_column[i]
6        for each edge in [first_edge;last_edge)
7              if neighbors of curr_vert in [first_edge;last_edge)
8                    if dist[curr_vert] = -1
9                          for each k in neighbors of curr_vert
10                               if bitmap_current.k = 1
11                                     dist[curr_vert] := level + 1
12                                     pred[curr_vert] := k
13                                     bitmap_next.vert := 1
14                                     break
15             curr_vert++
16 // data synchronization...
```

Рис. 5. Псевдокод цикла разметки вершин в алгоритме поиска в ширину на графе (обратный обход)

4. Тестирование производительности параллельного поиска в ширину

Оба описанных выше метода были встроены в собственную (*custom*) реализацию теста производительности Graph500 [5]. Ядро данного теста представляет собой параллельный поиск в ширину на большом графе, размер которого определяется параметром *scale*, равным логарифму по основанию 2 от количества вершин в графе. При этом средняя степень вершин всегда равна 16. Основным показателем производительности является скорость обхода графа, выраженная в количестве пройденных в секунду ребер (Traversed Edges Per Second — TEPS).

Разработанная *custom*-реализация использует MPI (один процесс на вычислительный узел) для передачи сообщений между узлами и OpenMP (восемь потоков на вычислительный узел) для работы с общей памятью внутри узла.

Тестирование проводилось для графов разного размера на 1, 2, 4 и 8 узлах суперкомпьютера «Уран», расположенного в ИММ УрО РАН. Каждый узел оборудован CPU Intel Xeon X5675 и 46 ГБ ОЗУ. Производительность *custom*-реализации сравнивалась со стандартными реализациями, которые предоставляются оргкомитетом рейтинга Graph500:

- *simple* (представляет собой «top-down» вариант поиска в ширину);
- *replicated* (представляет собой «bottom-up» вариант поиска в ширину).

Результаты тестирования представлены на рис. 6. Как видно, *custom*-реализация значительно превосходит по производительности *simple* и *replicated*-реализации. При этом, в случае с 8-ю узлами видно, что *custom*-реализация сохраняет потенциал дальнейшей масштабируемости.

5. Заключение

Эффективное распараллеливание поиска в ширину на графах с неравномерным распределением степеней затруднено из-за дисбаланса вычислительной нагрузки, а также больших объемов передаваемых данных на малом количестве итераций алгоритма, что создает пик вычислительной нагрузки.

В данной работе предложены методы распределения нагрузки и гибридизации обхода, позволяющие значительно (более чем в три раза) повысить скорость параллельного поиска в ширину на графе по сравнению со своими стандартными «top-down» и «bottom-up» аналогами.

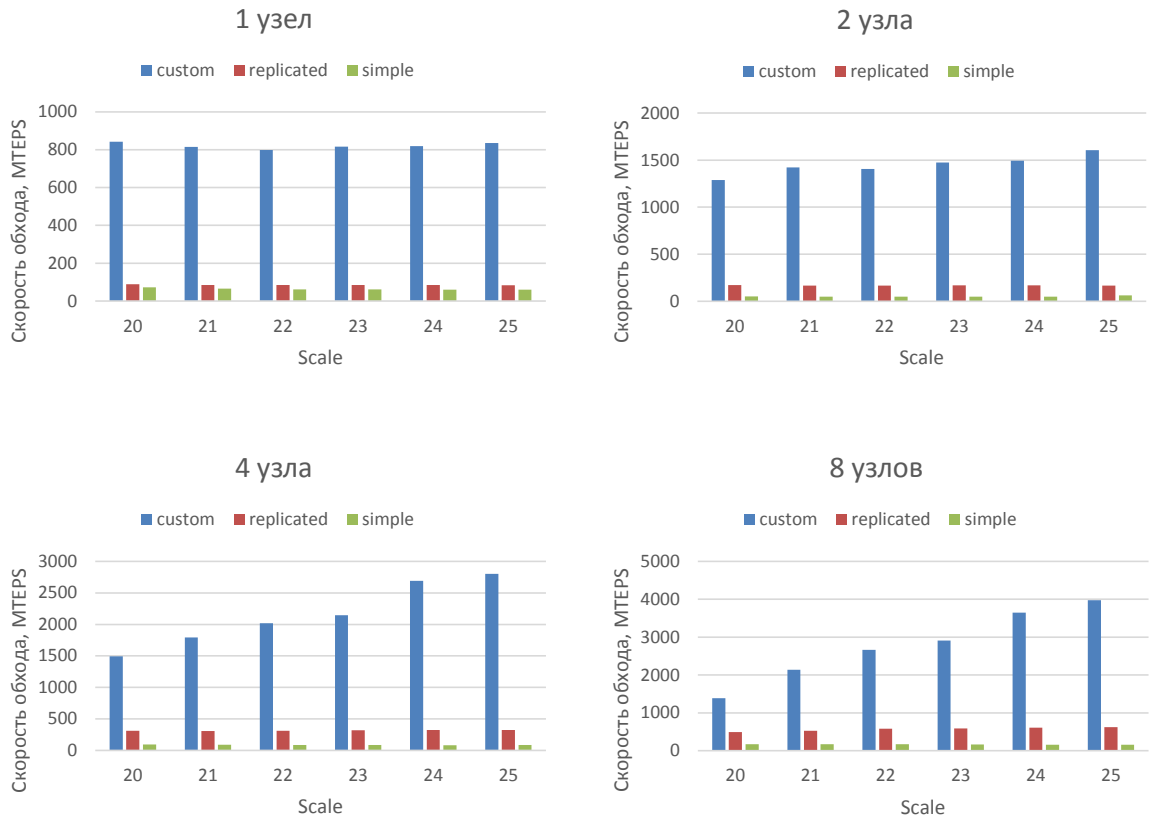

Рис. 6. Результаты тестирования производительности

В качестве направлений дальнейшей работы выбрано изучение масштабируемости представленного алгоритма на графах, имеющих различный размер и различную среднюю степень вершин. Актуальной задачей также является модификация *custom*-реализации для использования ускорителей вычислений с целью повышения эффективности алгоритма.

Литература

1. Mark E.J. Newman. The structure and function of complex networks. SIAM review, 45(2):167–256, 2003.

2. Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. Parallel Processing Letters, 17(01):5–20, 2007.

3. Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. Computing in Science and Engg., 10(2):14–19, March 2008.

4. Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

5. Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. 2010.

Parallel high performance graph processing

M.A. Chernoskutov[1,2]

[1]Institute of Mathematics and Mechanics UB RAS,
[2]Ural Federal University

Paper describes methods devoted to drastically speedup parallel breadth-first search algorithm. Main obstacle on the way to effectively parallelize breadth-first search is workload imbalance within computing nodes as well as significant volume of transferred data in the end of every iteration of the algorithm. Two methods are suggested in this paper to overcome these challenges. First method allows to distribute workloads between OpenMP threads in single node. Second method allows to reduce data transfer volume by using the hybrid graph traversal.

Keywords: parallel computing, graph processing, workload balancing

References

1. Mark E.J. Newman. The structure and function of complex networks. SIAM review, 45(2):167–256, 2003.

2. Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. Parallel Processing Letters, 17(01):5–20, 2007.

3. Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. Computing in Science and Engg., 10(2):14–19, March 2008.

4. Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

5. Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. 2010.