УДК 004.021 DOI: 10.14529/cmse180201

САМАЯ БЫСТРАЯ И ЭНЕРГОЭФФЕКТИВНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА ПОИСКА В ШИРИНУ НА ОДНОУЗЛОВЫХ РАЗЛИЧНЫХ ПАРАЛЛЕЛЬНЫХ АРХИТЕКТУРАХ СОГЛАСНО РЕЙТИНГУ GRAPH500*

© 2018 А.С. Колганов

Институт прикладной математики им. М.В. Келдыша РАН
(125047 Москва, Миусская пл., д. 4),
Московский государственный университет им. М.В. Ломоносова
(119991 Москва, ул. Ленинские горы, д. 1)
E-mail: alexander.k.s@mail.ru

Поступила в редакцию: 08.04.2018

Поиск в ширину является одним из основных алгоритмов обхода графа и базовым для многих алгоритмов анализа графов более высокого уровня. Поиск в ширину на графах является задачей с нерегулярным доступом к памяти и с нерегулярной зависимостью по данным, что сильно усложняет его распараллеливание на все существующие архитектуры. В статье будет рассмотрена реализация алгоритма поиска в ширину (основного теста рейтинга Graph500) для обработки больших графов на различных архитектурах: Intel x86, IBM Power8+, Intel KNL и NVidia GPU. Будет рассмотрены алгоритмы реализации поиска в ширину, такие как top-down обход, bottom-up обход и гибридный обход, содержащий в себе как top-down, так и bottom-up обходы. Будут описаны особенности реализации алгоритма на общей памяти, а также преобразования графа: локальная сортировка вершин графа, глобальная сортировка вершин графа, перенумерация всех вершин графа, сжатое представление вершин графа. Данные преобразования и гибридный алгоритм обхода позволяют достичь рекордных показателей производительности и энергоэффективности на данном алгоритме среди всех одноузловых систем рейтинга Graph500 и GreenGraph500.

Ключевые слова: параллельная обработка графов, BFS, CUDA, Power8, KNL, Graph500.

ОБРАЗЕЦ ЦИТИРОВАНИЯ

Колганов А.С. Самая быстрая и энергоэффективная реализация алгоритма поиска в ширину на одноузловых различных параллельных архитектурах согласно рейтингу Graph500 // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. Т. 7, № 2. С. 5–21. DOI: 10.14529/cmse180201.

Введение

В последнее время все большую роль играют графические ускорители (ГПУ) в неграфических вычислениях. Потребность их использования обусловлена их относительно высокой производительностью и более низкой стоимостью. Как известно, на ГПУ и центральных процессорах (ЦПУ) хорошо решаются задачи на структурных, регулярных сетках, где параллелизм так или иначе легко выделяется. Но есть задачи, которые требуют больших мощностей и используют неструктурированные сетки или данные. Примером таких задач являются: Single Shortest Source Path problem (SSSP) [1] — задача поиска кратчайших путей от заданной вершины до всех остальных во взвешенном графе, задача

*Статья рекомендована к публикации программным комитетом Международной научной конференции «Параллельные вычислительные технологии (ПаВТ) 2018».

Breadth First Search (BFS) [2] — задача поиска в ширину в неориентированном графе, Minimum Spanning Tree (MST) — задача поиска сильно связанных компонент [3, 4] и другие.

Данные задачи являются базовыми в ряде алгоритмов на графах. На данный момент алгоритмы BFS и SSSP используются для ранжирования вычислительных машин в рейтингах Graph500 [5] и GreenGraph500 [6]. Алгоритм BFS (breadth-first search или поиск в ширину) является одним из наиболее важных алгоритмов анализа на графах. Он используется для получения некоторых свойств связей между узлами в заданном графе. В основном BFS используется как звено, например, в таких алгоритмах, как нахождение связных компонент [7], нахождение максимального потока [8], нахождение центральных компонент (betweenness centrality) [9, 10], кластеризация [11] и многие другие.

Алгоритм BFS имеет линейную вычислительную сложность $O(n + m)$, где n — количество вершин и m — количество ребер графа. Данная вычислительная сложность является наиболее оптимальной для последовательной реализации. Но такая оценка вычислительной сложности не применима для параллельной реализации, так как последовательная реализация (например, с помощью алгоритма Дейкстры [12]) имеет зависимости по данным, что препятствует ее распараллеливание. Также производительность алгоритма BFS ограничена производительностью памяти той или иной архитектуры. Поэтому наибольшее значение имеют оптимизации, направленные на улучшение работы с памятью всех уровней.

Данная работа направлена на исследование возможности отображения алгоритма поиска в ширину на больших графах на различные параллельные одноузловые архитектуры, такие как x86, Power8, NVidia, Intel KNL. Целью работы является разработка программы, реализующая алгоритм BFS, которая обладает следующими свойствами: единая программная реализация на сколько это возможно, одинаково эффективное выполнение одной и той же реализации на различных архитектурах, максимальная производительность и энергоэффективность по сравнению с существующими решениями в рейтингах Graph500 и GreenGraph500. На данный момент создать единую программную реализацию с использованием одной технологии, которая будет максимально эффективно работать на всех перечисленных архитектурах, невозможно. Поэтому для архитектуры графических ускорителей NVidia использовалась модель CUDA, а для всех остальных архитектур — модель OpenMP.

Статья организована следующим образом. В разделе 1 дается описание рейтингов Graph500 и GreenGraph500, а также анализ существующих решений. Раздел 2 описывает формат хранения графа и алгоритмы его преобразования. В разделе 3 содержится описание алгоритмов параллельной реализации для ЦПУ и ГПУ. В разделе 4 дается описание полученных результатов.

1. Обзор существующих решений и рейтинг Graph500

1.1. Graph500 и GreenGraph500

Рейтинг Graph500 был создан как альтернатива рейтингу Top500 [13]. Данный рейтинг используется для ранжирования вычислительных машин в приложениях, которые используют нерегулярный доступ к памяти, в отличие от последнего. Для тестируемого приложения в рейтинге Graph500 пропускная способность памяти и коммуникационной сети играют наиболее важную роль. Рейтинг GreenGraph500 является альтернативой рейтинга Green500 и используется в дополнении к Graph500.

В Graph500 используется метрика — количество обработанных ребер графа в секунду (TEPS — traversed edges per second), в то время как в GreenGraph500 используется метрика — количество обработанных ребер графа в секунду на один ватт. Таким образом, первый рейтинг ранжирует вычислительные машины по скорости вычисления, а второй — по энергоэффективности. Данные рейтинги обновляются каждые полгода.

1.2. Существующие решения

Алгоритм поиска в ширину был придуман более 50 лет назад. И до сих пор проводятся исследования для эффективной параллельной реализации на различных устройствах. Данный алгоритм показывает насколько хорошо организована работа с памятью и коммуникационной средой вычислителей. Существует достаточно много работ по распараллеливанию данного алгоритма на x86 системах [14–18] и на ГПУ [19, 20]. Также подробные результаты выполнения реализованных алгоритмов можно увидеть в рейтингах Green500 и GreenGraph500. К сожалению, алгоритмы многих эффективных реализаций не опубликованы в зарубежных источниках.

Описанная реализация алгоритма BFS в данной статье с использованием ускорителя Tesla P100 является лучшей по быстродействию и энергоэффективности среди всех одноузловых системы рейтинга Graph500 на графах с количеством вершин более 2^{25}. Более подробный анализ представлен в разделе 5.

2. Формат хранения графа

Для оценки производительности алгоритма BFS используются неориентированные RMAT графы [21]. RMAT графы хорошо моделируют реальные графы из социальных сетей, Интернета. В данном случае рассматриваются RMAT графы со средней степенью связности вершины 16, а количество вершин является степенью двойки. В таком RMAT графе имеется одна большая связная компонента и некоторое количество небольших связных компонент или висящих вершин. Сильная связность компонент не позволяет каким-либо образом разделить граф на такие подграфы, которые помещались бы в кэш память.

Для построения графа используется генератор, который предоставляется разработчиками рейтинга Graph500. Данный генератор создает неориентированный граф в формате RMAT, причем выходные данные представлены в виде набора ребер графа. Такой формат не очень удобен для эффективной параллельной реализации графовых алгоритмов, так как необходимо иметь агрегированную информацию по каждой вершине, а именно — какие вершины являются соседями для данной. Удобный для этого представления формат называется CSR (Compressed Sparse Rows) [22].

Данный формат получил широкое распространение для хранения разреженных матриц и графов. Для неориентированного графа с N вершинами и M ребрами необходимо два массива: X (массив указателей на смежные вершины) и (массив списка смежных вершин). Массив X размера $N+1$, а массив — $2*M$, так как в неориентированном графе для любой пары вершин необходимо хранить прямую и обратную дуги. В массиве X хранятся начало и конец списка соседей, находящиеся в массиве , то есть весь список соседей вершины J находится в массиве A с индекса $X[J]$ до $X[J+1]$, не включая его. Для иллюстрации на рис. 1 слева показан граф из 4 вершин, записанный с помощью матрицы смежности, а справа — в формате CSR.

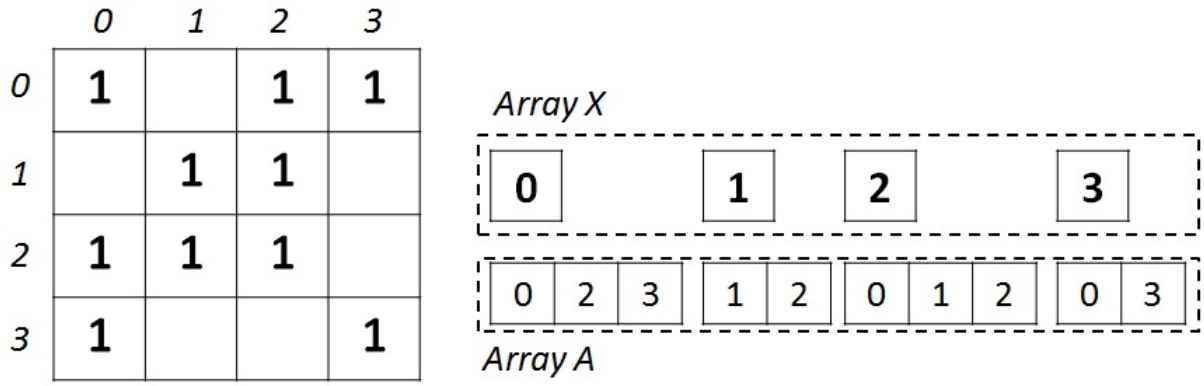

Рис. 1. Представление графа в виде матрицы смежности и в формате CSR

После преобразования графа в CSR-формат, необходимо проделать еще некоторую работу над входным графом для улучшения эффективности работы кэша и памяти вычислительных устройств. После выполнения описанных ниже преобразований, граф остается все в том же CSR-формате, но приобретет некоторые свойства, связанные с выполненными преобразованиями.

Введенные преобразования позволяют построить граф в оптимальном виде для большинства алгоритмов обхода графа в формате CSR. Добавление новой вершины в граф не приведет к выполнению всех преобразований заново, достаточно следовать введенным правилам и добавлять вершину так, чтобы общий порядок вершин не нарушался.

2.1. Локальная сортировка списка вершин

Для каждой вершины выполним сортировку по возрастанию ее списка соседей. В качестве ключа для сортировки будем использовать количество соседей для каждой сортируемой вершины. После выполнения данной сортировки, выполняя обход списка соседей у каждой вершины, мы будем обрабатывать сначала самые тяжелые вершины — вершины, имеющие большое количество соседей. Данную сортировку можно выполнять независимо для каждой вершины и параллельно. После выполнения данной сортировки, номера вершин графа в памяти не изменяются.

2.2. Глобальная сортировка списка вершин

Для списка всех вершин графа выполним сортировку по возрастанию. В качестве ключа будем использовать количество соседей для каждой из вершин. В отличие от локальной сортировки, данная сортировка требует перенумерации полученных вершин, так как меняется позиция вершины в списке. Процедура сортировки имеет сложность $O(N * log(N))$ и выполняется последовательно, а процедура перенумерации вершин может быть выполнена параллельно и по скорости работы сопоставима с временем копирования одного участка памяти в другой.

2.3. Перенумерация всех вершин графа

Занумеруем вершины графа таким образом, чтобы наиболее связные вершины имели наиболее близкие номера. Данная процедура устроена следующим образом. Сначала берется первая вершина из списка для перенумерации. Она получает номер 0. Затем все соседние вершины с рассматриваемой вершиной добавляются в очередь для перенумерации.

Следующая вершина из списка перенумерации получает номер 1 и так далее. В результате данной операции в каждой связной компоненте разница между максимальным и минимальным номером вершины будет наименьшей, что позволит лучшим образом использовать маленький объем кэша вычислительных устройств.

3. Реализация алгоритма

Алгоритм поиска в ширину в неориентированном графе устроен следующим образом. На входе подается начальная заранее неопределенная вершина в графе (корневая вершина для поиска). Алгоритм должен определить, на каком уровне, начиная от корневой вершины, находится каждая из вершин в графе. Под уровнем понимается минимальное количество ребер, которое необходимо преодолеть, чтобы добраться из корневой вершины в отличную от корневой вершину. Также для каждой из вершин, кроме корневой, необходимо определить вершину родителя. Так как у одной вершины может быть несколько родительских вершин, то в качестве ответа принимается любая из них.

Алгоритм поиска в ширину имеет несколько реализаций. Наиболее эффективная реализация — итерационный обход графа с синхронизацией по уровню. Каждый шаг представляет собой итерацию алгоритма, на которой информация с уровня J переносится на уровень $J + 1$. Исходный код последовательного алгоритма представлен по ссылке [2].

Параллельная реализация базируется на гибридном алгоритме, состоящем из top-down (TD) и bottom-up (BU) процедур, который был предложен автором статьи по следующей ссылке [18]. Суть данного алгоритма заключается в следующем. Процедура TD позволяет обойти вершины графа в прямом порядке, то есть, перебирая вершины, мы рассматриваем связи $V1 \Rightarrow V2$ как родитель-потомок. Вторая процедура BU позволяет обойти вершины в обратном порядке, то есть, перебирая вершины, мы рассматриваем связи $V1 \Rightarrow V2$ как потомок-родитель.

Рассмотрим последовательную реализацию гибридного алгоритма TD-BU, исходный код которой показан на рис. 2. Для обработки графа вершин нам необходимо создать дополнительные два массива-очереди, которые будут содержать в себе набор вершин на текущем уровне — Q_{curr}, и набор вершин на следующем уровне — Q_{next}. Чтобы выполнять более быстрые проверки существования вершины в очереди, необходимо ввести массив посещенных вершин. Но так как нам в результате работы алгоритма необходимо получить информацию о том, на каком уровне располагается каждая из вершин, этот массив может быть использован и в качестве индикатора посещенных и размеченных вершин.

Основной цикл работы алгоритма состоит из последовательной обработки каждой вершины, находящейся в очереди Q_{curr}. Если в очереди Q_{curr} больше не остается вершин, то алгоритм останавливается и ответ получен.

В самом начале алгоритма начинает работать процедура TD, так как в очереди содержится всего одна вершина. В процедуре TD мы для каждой вершины V_i из очереди Q_{curr} просматриваем список соседних с этой вершиной V_k и добавляем в очередь Q_{next} тех из них, которые еще не были помечены как посещенные. Также все такие вершины V_k получают номер текущего уровня и родительскую вершину V_i. После завершения просмотра всех вершин из очереди Q_{curr} запускается процедура выбора следующего состояния, которая может либо остаться на процедуре TD для следующей итерации, либо сменить процедуру на BU.

```
void bfs_hybrid(G, N, M, Vstart) {
    Levels = (-1);  Parents = (N + 1); Qcurr+=Vstart; CountQ=lvl=1;
    while (CountQ) {
      CountQ = 0; vis = 0; inLvl = 0;
      if (state == TD)
        for (auto &Vi : Qcurr)
          for (auto &Vk : G.Edges(Vi)) {
            inLvl++;
            if (Levels[Vk] == -1)
              Qnext += Vk; Levels[Vk] = lvl; Parents[Vk] = Vi; vis++;
          }
      else if (state == BU)
        for (auto &Vi : G)
          if (Levels[Vi] == -1)
            for (auto &Vk : G.Edges(Vi)) {
              inLvl++;
              if (Levels[Vk] == lvl - 1)
                Qnext += Vi; Levels[Vi] = lvl;
                Parents[Vi] = Vk; vis++;
                break;
              }
      change_state(Qcurr, Qnext, vis, inLvl, G);
      Qcurr = Qnext; CountQ = Qnext.size();
    }
}
```

Рис. 2. Последовательный гибридный алгоритм BFS

В процедуре BU мы просматриваем вершины не из очереди Q_{curr}, а те вершины, которые еще не были помечены. Данная информация содержится в массиве уровней $Levels$. Если такие вершины V_i еще не были размечены, то мы проходим по всем ее соседям V_k и если эти вершины, которые являются родителями для V_i, находятся на предыдущем уровне, то вершина V_i попадает в очередь Q_{next}. В отличие от процедуры TD, в данной процедуре можно прервать просмотр соседних вершин V_k, так как нам достаточно найти любую родительскую вершину.

Если выполнять поиск только процедурой TD, то на последних итерациях алгоритма список вершин, который необходимо обработать, будет очень большим, а неразмеченных вершин будет достаточно мало. Тем самым процедура будет выполнять лишние действия и лишние обращения в память. Если выполнять поиск только процедурой BU, то на первых итерациях алгоритма будет достаточно много неразмеченных вершин и аналогично процедуре TD, будут выполнены лишние действия и лишние обращения в память.

Получается, что первая процедура эффективна на первых итерациях алгоритма BFS, а вторая — на последних. Ясно, что наибольший эффект будет достигнут, когда мы будем использовать обе процедуры. Для того, чтобы автоматически определять, когда необходимо выполнять переключение с одной процедуры на другую, воспользуемся алгоритмом (процедура change_state), который был предложен авторами той же статьи [18]. Данный алгоритм по информации о количестве обработанных вершин на двух соседних итерациях пытается понять характер поведения обхода. В алгоритме вводится два коэффициента,

которые позволяют настраивать переключение с одной процедуры на другую в зависимости от обрабатываемого графа.

```
state change_state(Qcurr, Qnext, vis, inLvl, G)
{
    new_state = old_state;
    factor = G.M / G.N / 2;
    if(Qcurr.size() < Qnext.size()) { // Growing phase
      if(old_state == TOP_DOWN) {
        if(inLvl < ((G.N - vis) * factor + G.N) / alpha)
          new_state = TOP_DOWN;
        else
          new_state = BOTTOM_UP;
      }
    } else { // Shrinking phase
      if (Qnext.size() < ((G.N - vis)*factor + G.N) / (factor*beta))
        new_state = TOP_DOWN;
      else
        new_state = BOTTOM_UP;
    }
    return new_state;
}
```

Рис. 3. Функция изменения состояния

Процедура смены состояния может переводить не только TD в BU, но и обратно BU в TD. Последняя смена состояния полезна в том случае, когда количество вершин, которые необходимо просмотреть, достаточно мало. Для этого вводится понятие нарастающего фронта и затухающего фронта размеченных и неразмеченных вершин. Следующий исходный код, представленный на рис. 3, выполняет смену состояния в зависимости от полученных характеристик на конкретной итерации обхода графа в зависимости от настроенных коэффициентов *alpha* и *beta*. Данная функция может быть настроена на любой входной граф в зависимости от $factor$ (под $factor$ понимается средняя связность вершины графа).

Описанные выше концепции гибридной реализации алгоритма BFS применялись для параллельной реализации как для ЦПУ подобных систем, так и для ГПУ. Но есть некоторые отличия, которые будут рассмотрены далее.

3.1. Параллельная реализация на ЦПУ на общей памяти

Параллельная реализация для ЦПУ систем (Power 8+, Intel KNL и Intel x86) была выполнена с использованием OpenMP. Для запуска использовался один и тот же код, но для каждой платформы выполнялись свои настройки для директив OpenMP, например, задавались разные режимы балансировки нагрузки между нитями (schedule). Для реализации на ЦПУ с использованием OpenMP можно выполнить еще одно преобразование графа, а именно — сжатие списка вершин соседей.

Сжатие заключается в удалении незначащих нулевых битов каждого элемента из массива , причем данное преобразование делается отдельно для каждого диапазона $[X_i, X_{i+1})$. Происходит уплотнение элементов массива . Такое сжатие позволяет сократить

количество используемой памяти для хранения графа примерно на 30 % для больших графов порядка 2^{30} вершин и 2^{34} ребер. Для более маленьких графов экономия от такого преобразования пропорционально увеличивается в силу того, что уменьшается количество бит, которое занимает максимальный номер вершины в графе.

Такое преобразование графа накладывает некоторые ограничения на обработку вершин. Во-первых, все соседние вершины должны обрабатываться последовательно, так как они представляют собой сжатую, закодированную определенным образом последовательность элементов. Во-вторых, необходимо выполнять дополнительные действия по распаковке сжатых элементов. Данная процедура не является тривиальной и для ЦПУ Power8+ не позволила получить эффекта. Причиной может быть плохая оптимизация компилятора или отличная от Intel работа аппаратуры.

Для того, чтобы выполнять параллельно одну итерацию алгоритма, необходимо создать свои очереди Q_{th_next} для каждого потока OpenMP. А после выполнения всех циклов, выполнить объединение полученных очередей. Также необходимо локализовать все переменные, по которым есть редукционная зависимость. В качестве оптимизации процедура TD выполняется в последовательном режиме, если в очереди Q_{curr} количество вершин меньше заданного порога (например, меньше 300). Для графа разного размера, а также в зависимости от архитектуры, данный порог может иметь разные значения. Параллельные директивы располагались перед циклами $for\ (auto\ \&V_i\ in\ Q_{curr})$ в случае процедуры TD, и $for\ (auto\ \&V_i\ :\ G)$ в случае процедуры BU.

3.2. Параллельная реализация на ГПУ

Параллельная реализация для ГПУ была выполнена с использованием технологии CUDA. Реализация процедуры TD и BU существенно отличаются в случае использования ГПУ, так как существенно отличается алгоритм доступа к данным во время выполнения той или иной процедуры.

Процедура TD была реализована с помощью динамического параллелизма CUDA [23]. Данная возможность позволяет переложить некоторую работу, связанную с балансировкой нагрузки, на аппаратуру ГПУ. Каждая вершина V_i из очереди Q_{curr} может содержать абсолютно разное заранее не известное количество соседей. Из-за этого при прямом отображении всего цикла на набор нитей, возникает сильный дисбаланс нагрузки, так как CUDA позволяет использовать блок нитей фиксированного размера.

Описанная проблема решается путем использования динамического параллелизма. В начале запускаются мастер-нити. Данных нитей будет столько, сколько вершин содержится в очереди Q_{curr}. Затем каждая мастер-нить запускает столько дополнительных нитей, сколько соседей имеется у вершины V_i. Таким образом, количество используемых нитей формируется в динамике во время выполнения программы в зависимости от входных данных.

Данная процедура неудобна для выполнения на графическом процессоре из-за необходимости использовать динамический параллелизм. При большом суммарном количестве нитей, которые необходимо создать из мастер-нитей, возникают большие накладные расходы. Поэтому переключение на процедуру BU выполняется раньше, чем на ЦПУ.

Процедура BU является более благоприятной для выполнения на ГПУ, если проделать некоторые дополнительные преобразования данных. Данная процедура существенно

отличается от процедуры TD тем, что проход выполняется над всеми подряд идущими вершинами графа. Таким образом организованный цикл позволяет выполнить некоторую подготовку данных для хорошего доступа к памяти.

Преобразование заключается в следующем. Известно, что соседние нити одного варпа выполняют инструкции синхронно и параллельно. Для эффективного доступа к памяти требуется, чтобы соседние нити в варпе обращались к соседним ячейкам в памяти. Для примера положим количество нитей в варпе равным 2. Если каждой нити сопоставить одну вершину цикла $for\ (auto\ \&V_i\ :\ G)$, то во время обработки соседей в цикле $for\ (auto\ \&V_k\ :\ G.Edges(V_i))$ каждая нить будет обращаться в свою область памяти, что негативно скажется на производительности, так как соседние нити будут обрабатывать далеко расположенные ячейки в памяти. Для того, чтобы исправить положение, перемешаем элементы массива A таким образом, чтобы доступ к первым двум соседям из V_0 и V_1 осуществлялся наилучшим образом — соседние элементы располагались в соседних ячейках в памяти. Далее, в памяти таким же образом будут лежать вторые, третьи и т.д. элементы.

```
__global__ void bu_align( ... ) {
    idx = blockDim.x * blockIdx.x + threadIdx.x;
    countQNext = 0; inlvl = 0;
    for(i = idx; i < N; i += stride)
      if (levels[i] == 0) {
        start_k = rowsIndices[i];
        end_k = rowsIndices[i + N];
        for(k = start_k; k < start_k + 32 * end_k; k += 32) {
          inlvl++;
          vertex_id_t endk = endV[k];
          if (levels[endk] == lvl - 1) {
            parents[i] = endk;
            levels_out[i] = lvl;
            countQNext++;
            break;
          }
        }
      }
    atomicAdd(red_qnext, countQNext);
    atomicAdd(red_lvl, inlvl);
}
```

Рис. 4. Параллельное ядро для процедуры BU

Данное правило выравнивания применяется для группы нитей варпа (32 нити): выполняется перемешивание соседей — сначала располагаются первые 32 элемента, затем вторые 32 элемента и т.д. Так как граф отсортирован по убыванию количества соседей, то группы вершин, которые располагаются рядом, будут иметь достаточно близкое количество вершин соседей.

Перемешивать таким способом элементы можно не во всем графе, так как во время работы процедуры BU во внутреннем цикле есть досрочный выход. Описанные выше глобальная и локальная сортировки вершин позволяют выходить из данного цикла достаточно рано. Поэтому перемешивается только 40 % всех вершин графа. Данное преобразование требует дополнительной памяти для хранения перемешанного графа, зато

мы получаем заметный прирост в производительности. На рис. 4 представлен исходный код ядра для процедуры BU с использованием перемешанного расположения элементов в памяти.

4. Анализ полученных результатов

Тестированные реализованного алгоритма BFS производилось на четырех различных платформах: Intel Xeon Phi (Xeon KNL 7250) [24], Intel x86 (Xeon E5 2699 v3) [25], IBM Power8+ (Power 8+ s822lc) и GPU NVidia Tesla P100 [26]. Интересующие для сравнения характеристики данных устройств представлены в табл. 1 соответственно порядку их перечисления.

Таблица 1

Технические характеристики устройств

Ядер / Потоков	**Частота, ГГц**	**RAM, GB/s**	**Макс. TDP, Вт**	**Транз., млрд**
68 / 272	1,4	115 / 400	215	8
18 / 36	2,3	68	145	5,69
10 / 80	3,5	205	270	6
56 / 3584	1,4	40 / 700	300	15,3

В последнее время производители все больше задумываются о пропускной способности памяти. Как следствие этому, появляются различные решения проблемы медленного доступа к оперативной памяти. Среди рассматриваемых платформ две имеют двухуровневую структуру оперативной памяти.

Первая из них — Intel KNL, содержит быструю память на кристалле, скорость доступа к которой порядка 400 ГБ/с, и более медленную привычную нам DDR4, скорость доступа к которой не более 115 ГБ/с. Быстрая память имеет достаточно маленький размер — всего 16 ГБ, в то время как обычная память может быть до 384 ГБ. На тестируемом сервере было установлено 96 ГБ памяти такой памяти. Вторая платформа с гибридным решением — Power + NVidia Tesla. Данное решение базируется на новой технологии NVlink [27], которая позволяет иметь доступ к обычной памяти ЦПУ на скорости 40 ГБ/с, в то время как доступ к быстрой памяти осуществляется на скорости 700 ГБ/с. Количество быстрой памяти такое же, как и в Intel KNL — 16 ГБ.

Данные решения схожи с точки зрения организации — имеется быстрая память маленького размера, и медленная память большого размера. Сценарий использования двухуровневой памяти при обработки больших графов очевиден: быстрая память используется для хранения результата и промежуточных массивов, размеры которых достаточно малы по сравнению с входными данным, а исходный граф читается из медленной памяти.

С точки зрения реализации пользователю доступны следующие средства. Для Intel KNL достаточно использовать другие функции выделения памяти — hbm_malloc, вместо привычного malloc. Если программа использовала операторы malloc, то достаточно объявить один define для того, чтобы использовать данную возможность. Для NVidia Tesla необходимо использовать также другие функции выделения памяти — вместо cudaMalloc использовать cudaMallocHost. Данные модификации кода являются достаточными и не требуют каких-либо модификаций в вычислительной части программы.

Эксперименты проводились для графов разного размера, начиная от 2^{25} (4 ГБ) и заканчивая 2^{30} (128 ГБ). Средняя степень связности и тип графа брались из генератора графа для рейтинга Graph500. Данный генератор создает графы Кронекера со средней степенью связности 16 и коэффициентами $A = 0,57$, $B = 0,19$, $C = 0,19$.

Данного вида графы используются всеми участниками таблицы рейтинга, что позволяет корректно сравнивать реализации между собой. Значение производительности считается по метрике TEPS для таблицы Graph500 и TEPS / w для таблицы GreenGraph500. Для вычисления данной характеристики выполняется 64 запуска алгоритма BFS из разных стартовых вершин и берется среднее значение. Для вычисления потребления алгоритма берется текущее потребление системы в момент работы алгоритма с учетом потребления оперативной памяти.

Табл. 2 иллюстрирует полученную производительность в GTEPS на всех тестируемых графах. В таблице указываются два значения — минимальная / максимальная достигнутая производительность на каждом из графов. Также в случае использования Intel KNL были получены результаты выполнения алгоритма при использовании только памяти DDR4. К сожалению, даже при использовании всех алгоритмов сжатия данных, не удалось запустить на предоставленном сервере граф с 2^{30} вершинами на Intel KNL. Но учитывая стабильность работы Intel процессоров и технологичность Intel компиляторов, можно предположить, что производительность не изменится при увеличении размера графа (как это можно видеть для Intel Xeon E5).

Таблица 2

Полученная производительность в GTEPS

Размер графа	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}
Xeon KNL 7250	10,7/30,6	12,9/41	8,4/43,3	4,6/40,2	6,2/42,6	N/A
Xeon KNL 7250 DDR4	6,7/25,2	4,3/27	4,9/28,4	5,7/31,6	10,8/38,8	N/A
Xeon E5 2699 v3	11/16,5	11,8/17,3	12,7/18,5	13,1/18,3	12,1/18,0	12,4/21,1
IBM Power 8+ s822lc	8,8/22,5	9/23,3	7,9/23,4	10,4/23,7	10,1/24,6	7,59/14,8
NVidia Tesla P100	41/282	99/333	34/324	5/274	7,2/61	6,5/52

На рис. 5 отображена средняя производительность протестированных платформ. Можно заметить, что Power 8+ показал не очень хорошую стабильность при переходе с графа размером 64 ГБ на 128 ГБ. Возможно, это связано с тем, что использовался двух сокетный узел из двух аналогичных процессоров, причем у каждого процессора было по 128 ГБ памяти. И при обработке большего графа часть данных размещалось в памяти, не принадлежащей сокету. На графике также не отображена производительность Tesla P100 на более маленьких графах, так как разница между самым быстрым ЦПУ устройством и ГПУ составляет примерно 10 раз. Данное ускорение резко сокращается, когда графы становятся настолько большими, что не помещаются в кэш и доступ к графу осуществляется через NVlink. Но, несмотря на данное ограничение, производительность ГПУ все равно остается больше всех ЦПУ устройств. Такое поведение объясняется тем, что CUDA позволяет лучше контролировать вычисления и доступ к памяти, а также лучшей приспособленностью графических процессоров к параллельным вычислениям.

Табл. 3 иллюстрирует полученную производительность в GTEPS /w на всех тестируемых графах. В таблице указываются среднее энергопотребление при средней

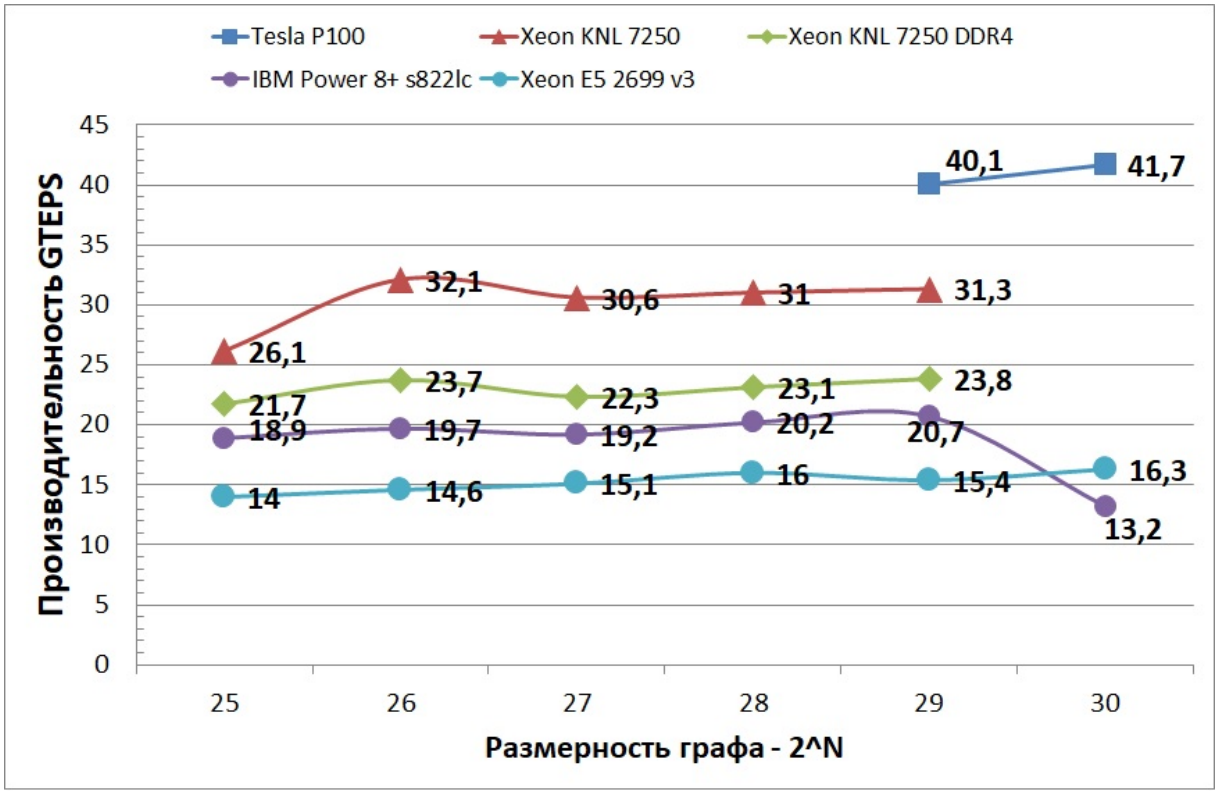

Рис. 5. Сравнение средней производительности

достигнутой производительности на каждом из графов. Резкое падение производительности и энергоэффективности при переходе с графа 2^{28} на граф 2^{29} на NVidia Tesla P100 объясняется тем, что быстрой памяти не хватает, чтобы поместить выровненную часть графа, к которой осуществляется наиболее частый доступ. В случае использования большего количества памяти (например, 32 ГБ) и увеличенного канала связи с ЦПУ NVlink 2.0 можно существенно повысить эффективность обработки графов большого размера.

Таблица 3

Полученная энергоэффективность в MTEPS / w

Размер графа	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}
Xeon KNL 7250	121,4	149,3	142,33	136,56	130,96	N/A
Xeon E5 2699 v3	95,56	98,65	100	101,9	91,12	84,46
IBM Power 8+ s822lc	93,8	97,04	93,2	95,28	92,41	53,23
NVidia Tesla P100	1228,57	1165,71	1235,96	1016,57	195,61	177,45

Заключение

В результате проделанной работы были реализованы два параллельных алгоритма BFS для ЦПУ подобных систем и для ГПУ. Было выполнено исследование производительности и энергоэффективности реализованных алгоритмов на различных платформах, таких как IBM Power8+, Intel x86, Intel Xeon Phi (KNL) и NVidia Tesla P100. Данные платформы имеют различные архитектурные особенности. Несмотря на это, первые три из них очень похожи по строению. Благодаря этому на этих платформах можно запускать OpenMP

приложения без каких-либо существенных изменений. Наоборот, архитектура ГПУ сильно отличается от ЦПУ подобных платформ и использует другую концепцию для реализации вычислительного кода — архитектуру CUDA.

Были рассмотрены графы, которые получаются после генератора для рейтинга Graph500. Была исследована производительность каждой из архитектур на двух классах данных. К первому классу относятся графы, которые помещаются в наиболее быструю память вычислителя. Ко второму классу относятся большие графы, которые нельзя поместить в быструю память целиком. Для демонстрации энергоэффективности использовались метрики GreenGraph500. Минимальный граф, который учитывается в рейтинге GreenGraph500 в классе больших данных, содержит в себе 2^{30} вершин и 2^{34} ребер и занимает в исходном виде 128 ГБ памяти. В классе же малых данных допускается граф любого размера до 2^{30} вершин и 2^{34} ребер, причем в качестве результата принимается наиболее большой граф, который удалось вместить в память.

На данный момент среди всех одноузловых систем в рейтинге Graph500 и GreenGraph500 полученная реализация на NVidia Tesla P100 занимает лидирующие позиции как в классе малых данных (с производительностью 220 GTEPS и эффективностью 1235,96 MTEPS/w), так и в классе больших данных (с производительностью 41,7 GTEPS и эффективностью 177,45 MTEPS/w). Такая высокая энергоэффективность и скорость работы на больших графах была достигнута благодаря новой технологии NVLink, которая связывает ГПУ и ЦПУ между собой и доступна на данный момент только в серверах компании IBM с ЦПУ Power8+.

В дальнейшем планируется исследовать возможность выполнения данного алгоритма на новой архитектуре NVidia Volta с использованием улучшенной технологии NVlink 2.0, а также планируется исследовать параллельную реализацию на нескольких ГПУ.

Литература

1. Cherkassky B.V., Goldberg A.V., Radzik T. Shortest Paths Algorithms: Theory and Experimental Evaluation // Math. Program. 1996. Vol. 73. P. 129–174. DOI: 10.1007/BF02592101.
2. Moore E.F. The Shortest Path through a Maze // Proceedings of the International Symposium on the Theory of Switching (2–5 April 1957). Harvard University Press, 1959. P. 285–292.
3. Chazelle B.A. Minimum Spanning Tree Algorithm with Inverse-ackermann Type Complexity // Journal of the ACM. 2000. Vol. 47, No. 6. P. 1028–1047. DOI: 10.1145/355541.355562.
4. Колганов А.С. Параллельная реализация алгоритма поиска минимальных остовных деревьев с использованием центрального и графического процессоров // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. 2016. Т. 5, № 3. С. 5–19. DOI: 10.14529/cmse160301.
5. Рейтинг Graph500. URL: `http://graph500.org/` (дата обращения 01.12.2017)
6. Рейтинг GreenGraph500. URL: `http://green.graph500.org/` (дата обращения 01.12.2017)
7. Cormen T., Leiserson, C., Rivest R. Introduction to Algorithms. MIT Press, Cambridge. 1990.
8. Edmonds J., Karp R.M. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems // Journal of the ACM. 1972. Vol. 19, No. 2. P. 248–264. DOI: 10.1007/3-540-36478-1_4.

9. Brandes U. A Faster Algorithm for Betweenness Centrality // J. Math. Sociol. 2001. Vol. 25, No. 2. P. 163–177. DOI: 10.1080/0022250X.2001.9990249.

10. Frasca M., Madduri K., Raghavan P. NUMA-Aware Graph Mining Techniques for Performance and Energy Efficiency // Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (Salt Lake City, Utah, USA, November 10–16, 2012). P. 1–11. DOI: 110.1109/SC.2012.81.

11. Girvan M., Newman M.E. Community Structure in Social and Biological Networks // Proc. Natl. Acad. Sci. (USA, June 11, 2002). Vol. 99, No. 12. P. 7821–7826. DOI: 10.1073/pnas.122653799.

12. Dijkstra E.W. A Note on Two Problems in Connexion with Graphs // Numerische Mathematik. 1959. Vol. 1, No. 1. P. 269–271. DOI: 10.1007/BF01386390.

13. Рейтинг Top500. URL: `https://www.top500.org/` (дата обращения 01.12.2017)

14. Bader D.A., Madduri K. Designing Multithreaded Algorithms for Breadth-first Search and St-connectivity on the Cray MTA-2. 2006. P. 523–530. DOI: 10.1109/ICPP.2006.34.

15. Korf R.E., Schultze P. Large-scale Parallel Breadth-first Search // AAAI. 2005. P. 1380–1385.

16. Yoo A., Chow E., Henderson K., McLendon W., Hendrickson B., Catalyurek U. A Scalable Distributed Parallel Breadth-first Search Algorithm on BlueGene/L // Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (Seattle, Washington, USA, November 12–18, 2005). DOI: 10.1109/SC.2005.4.

17. Zhang Y., Hansen E.A. Parallel Breadth-first Heuristic Search on a Shared-memory Architecture // AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications. 2006.

18. Yasui Y., Fujisawa K., Sato Y. Fast and Energy-efficient Breadth-First Search on a Single NUMA System // Lecture Notes in Computer Science. Vol. 8488. P. 365–381. DOI: 10.1007/978-3-319-07518-1_23.

19. Hiragushi T., Takahashi D. Efficient Hybrid Breadth-First Search on GPUs // Lecture Notes in Computer Science. Vol. 8286. P. 40–50. DOI: 10.1007/978-3-319-03889-6_5.

20. Merrill D., Garland M., Grimshaw A. Scalable GPU Graph Traversal // Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New Orleans, Louisiana, USA, February 25–29, 2012). P. 117–128. DOI: 10.1145/2370036.2145832.

21. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A Recursive Model for Graph Mining // Proceedings of the 2004 SIAM International Conference on Data Mining (Florida, USA, April 22–24, 2004). P. 442–446. DOI: 10.1137/1.9781611972740.43.

22. Pissanetzky S. Sparse Matrix Technology. Academic Press. 1984.

23. Динамический параллелизм в CUDA. URL: `https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/` (дата обращения 01.12.2017)

24. Спецификация процессора Intel Xeon Phi 7250. URL: `https://ark.intel.com/ru/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core` (дата обращения 01.12.2017)

25. Спецификация процессора Intel Xeon E5 2699 v3. URL: https://ark.intel.com/ru/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz (дата обращения 01.12.2017)

26. Спецификация процессоров IBM Power8 и NVidia Tesla P100. URL: https://www-03.ibm.com/systems/ru/power/hardware/s822lc-hpc/ (дата обращения 01.12.2017)

27. Технология NVLink. URL: http://www.nvidia.com/object/nvlink.html (дата обращения 01.12.2017)

Колганов Александр Сергеевич, младший научный сотрудник, Институт прикладной математики им. М.В. Келдыша РАН (Москва, Российская Федерация), аспирант, кафедра системного программирования, Московский государственный университет имени М.В. Ломоносова (Москва, Российская Федерация)

DOI: 10.14529/cmse180201

THE FASTEST AND ENERGY-EFFICIENT BREADTH-FIRST SEARCH ALGORITHM ON A SINGLE NODE WITH VARIOUS PARALLEL ARCHITECTURES ACCORDING TO GRAPH500

Keldysh Institute of Applied Mathematics
(Miusskaya sq. 4, Moscow, 125047 Russia),
Lomonosov Moscow State Universisty
(GSP-1, Leninskie Gory 1, Moscow, 119991 Russia)
E-mail: alexander.k.s@mail.ru
Received: 08.04.2018

The breadth-first search algorithm is one of the basic algorithms for graph traversing and is used in many other algorithms of high-level graph analysis. BFS is characterized with intensive irregular memory accesses and a random data dependency, which greatly complicates its parallelization to all existing architectures. The paper considers the implementation of the BFS algorithm (the core benchmark of the Graph500 rating) for processing large graphs on different architectures: Intel x86, IBM Power8+, Intel KNL and NVidia GPU. Algorithms for implementing breadth-first search will be considered, such as top-down traverse, bottom-up traverse, and a hybrid traverse that includes both top-down and bottom-up traverses. The features of the algorithm implementation on shared memory will be shown, as well as graph transformations (local sorting of graph vertices, global sorting of graph vertices, renumbering of all graph vertexes, compressed representation of graph vertices), which allow achieving the best performance and energy-efficiency in this algorithm among all single-node systems of Graph500 and GreenGraph500 ratings.

Keywords: parallel graph processing, BFS, CUDA, Power8, KNL, Graph500.

FOR CITATION

Kolganov A.S. The Fastest and Energy-Efficient Breadth-First Search Algorithm on a Single Node with Various Parallel Architectures According to Graph500. *Bulletin of the South Ural State University. Series: Computational Mathematics and Software Engineering.* 2018. vol. 7, no. 2. pp. 5–21. (in Russian) DOI: 10.14529/cmse180201.

References

1. Cherkassky B.V., Goldberg A.V., Radzik T. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Program.* 1996. vol. 73. pp. 129–174. DOI: 10.1007/BF02592101.
2. Moore E.F. The Shortest Path through a Maze. Proceedings of the International Symposium on the Theory of Switching (April 2–5, 1957). Harvard University Press, 1959. pp. 285–292.
3. Chazelle B.A., Minimum Spanning Tree Algorithm with Inverse-ackermann Type Complexity. *Journal of the ACM.* 2000. vol. 47, no. 6. pp. 1028–1047.
4. Kolganov A.S. Parallel Implementation of Minimum Spanning Tree Algorithm on CPU and GPU. *Vestnik Yuzho-Uralskogo gosudarstvennogo universiteta. Seriya: Vychislitel'naja matematika i informatika* [Bulletin of South Ural State University. Series: Computational Mathematics and Software Engineering]. 2016. vol. 5, no. 3. pp. 5–19. DOI: 10.14529/cmse160301. (in Russian)
5. Graph500. Available at: `http://graph500.org/` (accessed 01.12.2017)
6. GreenGraph500. Available at: `http://green.graph500.org/` (accessed 01.12.2017)
7. Cormen T., Leiserson C., Rivest R. Introduction to Algorithms. MIT Press, Cambridge. 1990.
8. Edmonds J., Karp R.M. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM.* 1972. vol. 19, no. 2. pp. 248–264. DOI: 10.1007/3-540-36478-1_4.
9. Brandes U. A Faster Algorithm for Betweenness Centrality. *J. Math. Sociol.* 2001. vol. 25, no. 2. pp. 163–177. DOI: 10.1080/0022250X.2001.9990249.
10. Frasca M., Madduri K., Raghavan P. NUMA-Aware Graph Mining Techniques for Performance and Energy Efficiency. Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (Salt Lake City, Utah, USA, November 10–16, 2012). pp. 1–11. DOI: 110.1109/SC.2012.81.
11. Girvan M., Newman M.E. Community Structure in Social and Biological Networks. Proc. Natl. Acad. Sci. (USA, June 11, 2002). vol. 99, no. 12. pp. 7821–7826. DOI: 10.1073/pnas.122653799.
12. Dijkstra E.W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik.* 1959. vol. 1, no. 1. pp. 269–271. DOI: 10.1007/BF01386390.
13. Top500. Available at: `https://www.top500.org/` (accessed 01.12.2017)
14. Bader D.A., Madduri K. Designing Multithreaded Algorithms for Breadth-first Search and St-connectivity on the Cray MTA-2. 2006. pp. 523–530. DOI: 10.1109/ICPP.2006.34.
15. Korf R.E., Schultze P. Large-scale Parallel Breadth-first Search. *AAAI.* 2005. pp. 1380–1385.
16. Yoo A., Chow E., Henderson K., McLendon W., Hendrickson B., Catalyurek U. A Scalable Distributed Parallel Breadth-first Search Algorithm on BlueGene/L. Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (Seattle, Washington, USA, November 12–18, 2005). DOI: 10.1109/SC.2005.4.

17. Zhang Y., Hansen E.A. Parallel Breadth-first Heuristic Search on a Shared-memory Architecture. AAAI Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications. 2006.

18. Yasui Y., Fujisawa K., Sato Y. Fast and Energy-efficient Breadth-First Search on a Single NUMA System. *Lecture Notes in Computer Science*. 2014. vol. 8488. pp. 365–381. DOI: 10.1007/978-3-319-07518-1_23.

19. Hiragushi T., Takahashi D. Efficient Hybrid Breadth-First Search on GPUs. *Lecture Notes in Computer Science*. 2013. vol. 8286. pp. 40–50. DOI: 10.1007/978-3-319-03889-6_5.

20. Merrill D., Garland M., Grimshaw A. Scalable GPU Graph Traversal. Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New Orleans, Louisiana, USA, February 25–29, 2012). pp. 117–128. DOI: 10.1145/2370036.2145832.

21. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A Recursive Model for Graph Mining. Proceedings of the 2004 SIAM International Conference on Data Mining (Florida, USA, April 22–24, 2004). pp. 442–446. DOI: 10.1137/1.9781611972740.43.

22. Pissanetzky S. Sparse Matrix Technology. Academic Press. 1984.

23. CUDA Dynamic Parallelism. Available at: `https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/` (accessed 01.12.2017)

24. Intel Xeon Phi 7250. Available at: `https://ark.intel.com/ru/products/94035/Intel-Xeon-Phi-Processor-7250-16GB-1_40-GHz-68-core` (accessed 01.12.2017)

25. Intel Xeon E5 2699 v3. Available at: `https://ark.intel.com/ru/products/81061/Intel-Xeon-Processor-E5-2699-v3-45M-Cache-2_30-GHz` (accessed 01.12.2017)

26. IBM Power8 and NVidia Tesla P100. Available at: `https://www-03.ibm.com/systems/ru/power/hardware/s822lc-hpc/` (accessed 01.12.2017)

27. Nvidia NVLink. Available at: `http://www.nvidia.com/object/nvlink.html` (accessed 01.12.2017)