ИССЛЕДОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ЗАДАЧИ ПОИСКА ВШИРЬ В ГРАФЕ НА СОПРОЦЕССОРЕ INTEL XEON PHI

Е.А. Головина, А.С. Семенов, А.С. Фролов

1. Введение

Обработка больших графов — относительно новая, интенсивно развивающаяся область приложений для суперкомпьютерных и кластерных систем, характеризуемая работой с большими объемами данных, размещенных в оперативной памяти вычислительных узлов, низкой пространственно-временной локализацией обращений к памяти и, как следствие, неэффективным использованием иерархической кэш-памяти процессоров, высоким количеством промахов в TLB и резким снижением эффективности работы встроенного в процессор контроллера памяти.

Применение специализированных ускорителей и сопроцессоров, таких как Nvidia Kepler, AMD FireStream и Intel Xeon Phi, находит все большее применение в составе крупных высокопроизводительных вычислительных систем, занимающих первые строки в списке Top500. Так, например, первое место в новой редакции списка Top500 (июнь 2013) занял китайский суперкомпьютер, построенный с использованием 48 тысяч ускорителей Intel Xeon Phi с общей пиковой производительностью около 50 PFLOP/s.

Графические ускорители Nvidia и AMD изначально разрабатывались для решения задач, хорошо отображающихся на потоковую архитектуру, например, для задач, работающих с плотнозаполненными матрицами и регулярными потоками данных. Однако даже в случае графических ускорителей нерегулярные приложения могут быть определенным образом оптимизированы для достижения высокой производительности. В отличие от графических ускорителей сопроцессор Intel Xeon Phi имеет архитектуру мультитредового многоядерного процессора и в большей степени универсален, что потенциально позволяет эффективно применить его для решения нерегулярных приложений с интенсивной работой с памятью, таких как графовые приложения, нерегулярные сеточные методы с адаптивными сетками, оптимизационные приложения.

В данной работе представлены результаты исследования производительности ускорителя Intel Xeon Phi на задаче поиска достижимых вершин в графе методом поиска вширь и сравнения с производительностью, полученной на 8-ядерном микропроцессоре Intel Sandy Bridge-EP. Задача BFS лежит в основе рейтингового списка Graph500, включающего как большие суперкомпьютеры, так и многопроцессорные вычислительные узлы.

2. Алгоритмы реализации задачи поиска вширь в графе

Одним из наиболее популярных алгоритмов обработки графов является поиск вширь (breadth-first search, BFS) в графе. Для заданного графа и заданной вершины *r* BFS находит все вершины, достижимые через ребра графа от вершины *r*. Особенностью алгоритма является то, что BFS не должен анализировать вершины, отстоящие от вершины *r* на расстояние *s+1* ребер, до тех пор, пока не проанализирует все вершины, отстоящие от вершины *r* на расстояние *s* ребер. Вершины, отстоящие от исходной вершины *r* на заданное расстояние, называются уровнем.

Во всех рассматриваемых алгоритмах граф задается матрицей смежности, которая хранится в виде разреженной матрицы в формате CRS с плотным хранением строк ненулевых элементов. Распараллеливание осуществлено при помощи технологии OpenMP.

2.1. Подход Queue-based

В первом подходе к реализации поиска вширь в графе [1, 2], который носит название Queue-based и схема которого изображена на рис. 1, каждому уровню соответствует массив вершин *Q*, в результате обхода которых формируется массив Q_{next} вершин следующего уровня. При параллельной работе тредов необходимо, чтобы каждый тред добавлял значение в Q_{next} с использованием атомарной операции __sync_fetch_and_add. Для анализа вершин нужно хранить в массиве *Marked* информацию о том, была ли вершина пройдена ранее.

В рамках подхода Queue-based рассмотрено два алгоритма.

1. Алгоритм simple, см. рис. 1.

2. Чтобы уменьшить число использований атомарной операции __sync_fetch_and_add, в алгоритме block каждый тред получает порцию для заполнения массива Q_{next} размера *size* [2]. Как только эта порция заполнена, тред с помощью атомарной операции получает следующую порцию для заполнения. В результате атомарная операция вызывается не с каждой отмеченной вершиной, а в *size* раз реже.

```
1 Q_counter = 1 // инициализация счетчика вершин Q
2 Q[0] = r // инициализация текущего уровня Q
3 Marked[r] = 1 // отметка начальной вершины r
4 while Q_counter >0 // пока уровень не пуст
5         Q_next_counter = 0 // обнуление счетчика следующего уровня
```

```
6       #pragma omp parallel // параллельная обработка уровня
7       for all vertex in Q  do
8             for all w: (vertex, w) in E do // для всех вершин w, смежных с vertex
9                   if  (Marked[w] != 0) then // если вершина не отмечена
10                        Q_next[__sync_fetch_and_add(Q_next_counter, 1)] = w // добавление w в Q_next
11                        Marked[w] = 1 // отметка вершины w
12                  end if
13            end for
14      end for
15      swap(Q, Q_next)  // переход на следующий уровень: обмен Q и Q_next, Q_counter =  Q_next_counter
16 end while
```

Рис. 1. Схема алгоритма simple, подход Queue-based

2.2. Подход Read-based

Второй подход, назовем его Read-based (ему соответствует алгоритм read) [3], совершенно отличается от первого. Основным рабочим массивом в подходе Read-based является массив *levels*, длина которого равна числу вершин в графе, а в каждой ячейке хранится номер уровня, на котором расположена вершина, или -1, если вершина еще не обрабатывалась. На каждом уровне просматривается весь массив с начала до конца, и анализируются те вершины, которые находятся на текущем уровне. Смежным им вершинам в массиве *levels* приписывается следующий уровень и, тем самым, они помечаются. Схема подхода Read-based изображена на рис. 2.

```
1 numLevel = 0 // инициализация номера уровня
2 levels[r] = numLevel // начальная вершина r будет обрабатываться на первом уровне
3 numLevelVerts = 1 // количество вершин на уровне
4 while numLevelVerts > 0 // есть вершины на текущем уровне
5       numLevelVerts = 0
6       #pragma omp parallel for reduction(+:numLevelVerts) // параллельная обработка массива levels
7       for all vertex in V do // для всех вершин графа
8             if (levels[vertex] != numLevel) then continue // пропуск вершины не на текущем уровне
9             for all w: (vertex, w) in E do // для всех вершин w, смежных с vertex
10                  if (levels[w] != -1) then // если уровень вершины не отмечен
11                        levels[w] = numLevel + 1 //  отметка ее для следующего уровня
12                        numLevelVerts =  numLevelVerts + 1
13                  end if
14            end for
15      end for
16      numLevel =  numLevel + 1
17 end while
```

Рис. 2. Схема алгоритма read, подход Read-based

2.3. Подход Bottom-up

В [4] предложен совершенно другой подход к реализации поиска вширь в графе. Для большого количества типов графов количество вершин на каждом следующем уровне резко возрастает, достигая максимальных значений на нескольких уровнях. Затем количество вершин на уровне падает. Когда вершин на уровне становится много, а значительная часть поиска уже позади, среди соседей вершин текущего уровня очень мало неотмеченных вершин, таким образом, большое количество работы производится впустую. Для таких уровней становится выгодно использовать восходящий подход bottom-up, когда просматриваются соседи всех неотмеченных вершин, и если какой-то сосед неотмеченной вершины лежит на текущем уровне, значит данная вершина является потомком этого соседа в дереве поиска, при этом других соседей данной вершины просматривать уже не нужно. Схема обработки одного уровня при подходе bottom-up приведена на рис. 3.

В реализации используется гибридный подход, в отличие от [4] в данной работе для первых уровней и последних уровней используется подход Read-based. С некоторого уровня обработка производится при помощи подхода bottom-up.

```
1  #pragma omp parallel for reduction(+:numLevelVerts) // параллельная обработка массива levels
2  for all vertex in V do // для всех вершин графа
3         if (!levels[vertex]) then // если вершина vertex не отмечена
4               for all w: (vertex, w) in E  do // для всех вершин w, смежных с vertex
5                     if (levels[w] == numLevel) then // если w принадлежит текущему уровню
6                           levels[vertex] = numLevel + 1 //  отметка vertex для следующего уровня
```

```
7                           numLevelVerts =  numLevelVerts + 1
8                           break // выход из цикла for просмотра вершин w
13                      end if
14                  end for
15              end if
16 end for
```

Рис. 3. Схема обработки одного уровня алгоритмом bottom-up одноименного подхода

3. Исследование производительности

Производительность алгоритмов реализации поиска вширь в графе исследовалась на микропроцессоре Intel Xeon Sandy Bridge-EP и сопроцессоре Intel Xeon Phi, которые условно будем называть Sandy Bridge-EP и MIC, их основные характеристики приведены в табл. 1. Все реализации запускались для неориентированных графов Пуассона, в которых вероятность существования ребра между любыми двумя вершинами постоянна. Характеристикой производительности алгоритмов BFS принято считать количество миллионов пройденных ребер графа в секунду (ME/s). Для получения результатов на MIC все алгоритмы запускались непосредственно на MIC, в режиме native.

Таблица 1. Характеристики используемых процессоров.

	Sandy Bridge-EP	MIC
Название модели	Xeon E5-2660	Xeon Phi 5110P
Частота	2.2 ГГц	1.05 ГГц
Количество сокетов	1	1
Количество ядер	8	60
Количество аппаратных тредов в ядре	2	4
Размер памяти кэшей данных	64 КБ* / 2 МБ* / 20 МБ	32* КБ / 512* КБ
Объем установленной памяти	32 ГБ	8 ГБ
Тип используемой памяти	DDR3	GDDR5
Пропускная способность памяти	51 ГБ/с	352 ГБ/с
Задержка обращения в память	~200 тактов	~300 тактов

* — в расчете на одно ядро.

3.1. Исследование производительности разработанных алгоритмов

Графики производительности различных алгоритмов поиска вширь в графе для задачи с числом вершин N = 134 млн и средней связностью вершины k = 8 в зависимости от числа используемых тредов для Sandy Bridge-EP и MIC приведены на рис. 4 и рис. 5 соответственно. Стоит отметить, что для одного треда для алгоритма simple скорость работы MIC в 20 раз ниже по сравнению с Sandy Bridge-EP. Поэтому достижение высокой производительности на MIC связано в первую очередь с масштабируемостью алгоритмов при использовании большого числа тредов.

Алгоритмы simple и block подхода Queue-based показывают очень низкую масштабируемость, несмотря на то, что в алгоритме block использование атомарной операции сокращено.

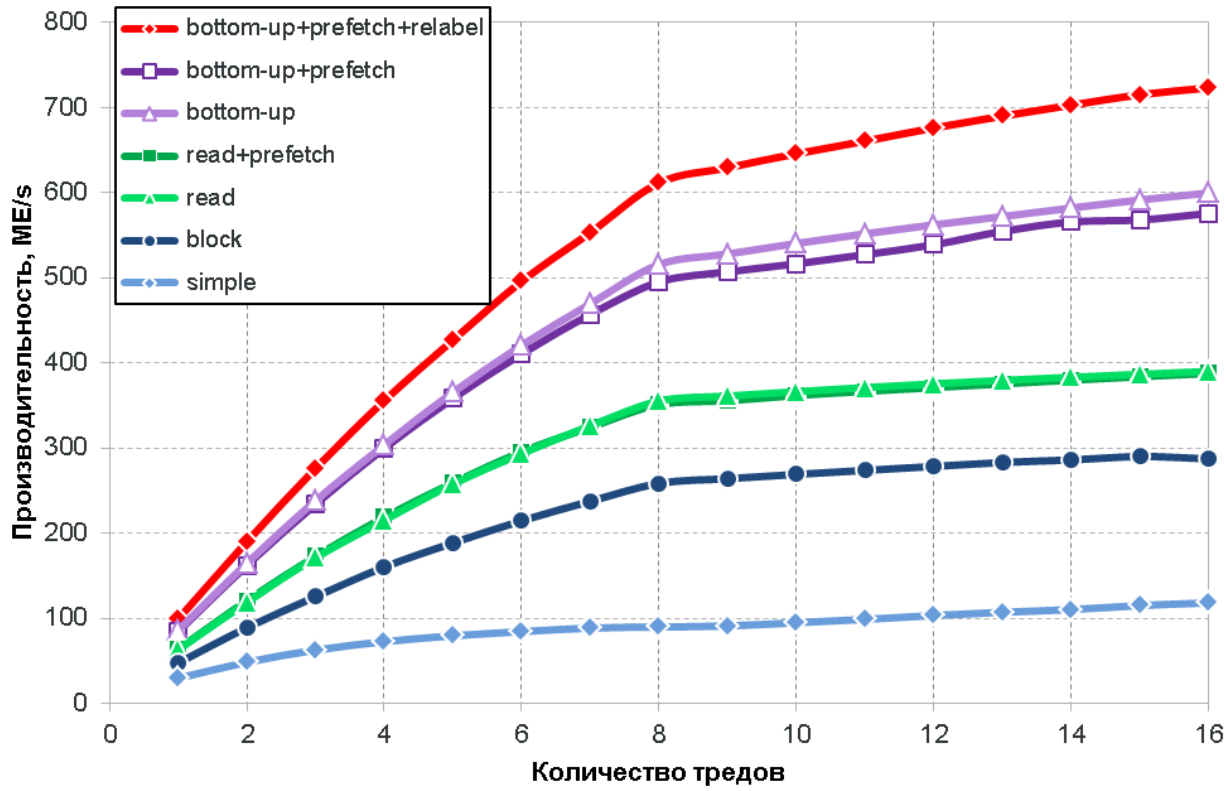

Рис. 4. Производительность различных алгоритмов поиска вширь в графе для задачи с числом вершин N = 134 млн и средней связностью вершины k = 8 в зависимости от числа используемых тредов на микропроцессоре Sandy Bridge-EP

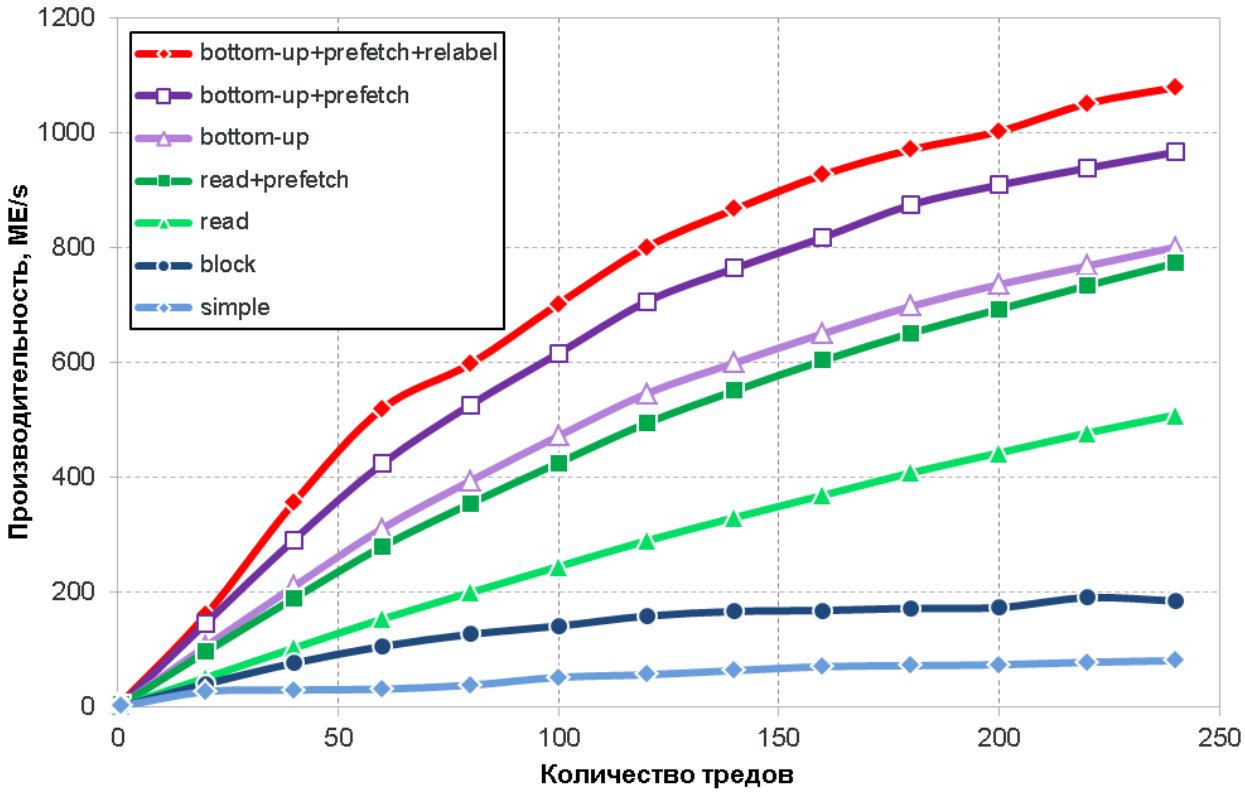

Рис. 5. Производительность различных алгоритмов поиска вширь в графе для задачи с числом вершин N = 134 млн и средней связностью вершины k = 8 в зависимости от числа используемых тредов на сопроцессоре MIC

Подход Read-based дает качественно другой результат, см. рис. 4 и рис. 5, алгоритм read. Преимуществ подхода Read-based несколько. Во-первых, в нем отсутствуют атомарные операции, которые очень сильно

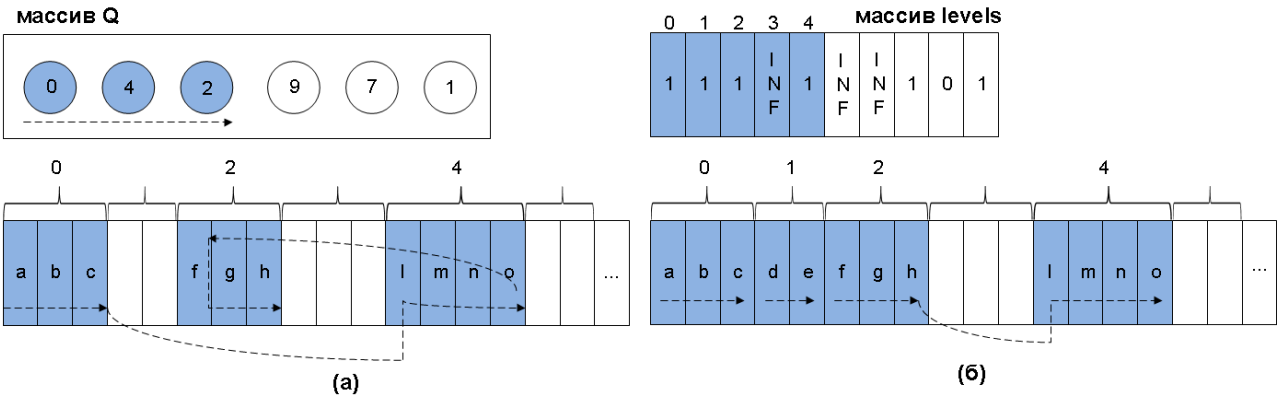

Рис. 6. Шаблоны доступа к памяти в подходах (а) Queue-based и (б) Read-based

ограничивают масштабирование задачи с ростом числа используемых ядер и тредов. Во-вторых, совершенно исчезают массивы Q и Q_{next}, что приводит к экономии объема и пропускной способности памяти. Минимизация объема памяти означает более эффективное использование кэша. И наконец, в-третьих, шаблон доступа к памяти для подхода Read-based более последователен по сравнению с подходом Queue-based, это проиллюстрировано на рис. 6. На рис. 6 (а) изображена очередь Q с вершинами *{0, 4, 2, 9, 7, 1}* и изображен доступ к массиву номеров соседей формата CRS для первых трех вершин очереди Q. На рис. 6 (б) изображен доступ к данным для того же уровня с номером один для подхода Read-based. Доступ к массиву *levels* является полностью последовательным, а к массиву номеров соседей — последовательным с возможными пропусками. Последовательный доступ разрешает использовать большую пропускную способность при последовательном доступе, увеличивает эффективность использования кэша, позволяет эффективно работать аппаратной преднакачке, уменьшает количество промахов в кэш дескрипторов страниц TLB. Также при распараллеливании на большое количество тредов треды лучше разделяют между собой массив номеров соседей, не закачивая его в кэши ядер по несколько раз.

Подход Read-based был изначально предложен для графических процессоров, и именно использование в этом подходе пропускной способности подсистемы памяти натолкнуло авторов статьи на мысль о том, что он может быть с успехом применен для сопроцессора Intel Xeon Phi.

Казалось бы, недостаток подхода Read-based заключается в лишней работе, связанной с просмотром на каждой итерации всего массива *levels*. Но скорость последовательного доступа к массиву очень велика, а также время обработки вершин массива *levels*, которые не принадлежат текущему уровню, очень мало. Поэтому накладные расходы этого алгоритма незначительны.

Тем не менее, случайный доступ остается в подходе Read-based. В строке 10 схемы алгоритма read на рис. 2 обращение к массиву *levels* происходит по вершине *w* из списка соседних вершин, номер которой для графа Пуассона случаен. Сравнение пропускной способности памяти для последовательного (векторизованного) доступа и случайного доступа для Sandy Bridge-EP и MIC приведены в табл. 2. Результаты для векторизованных вариантов программ для сопроцессора MIC взяты из [5]. Все остальные результаты получены при помощи пакета DISBench [6]. Сравнение показывает, насколько дорого обходится случайный шаблон адресов обращений в память. Это связано с тем, что во время работы с потоком случайных адресов при выборке одного слова данных из подгружаемой кэш-строки резко снижается эффективность работы встроенного в процессор контроллера памяти как за счет уменьшения количества полезных данных, передаваемых по шине памяти, так и за счет выполнения дополнительных команд непосредственно в микросхемах памяти, необходимых для работы по не подряд идущим адресам.

Таблица 2. Характеристики пропускной способности чтения в ГБ/с из памяти для Sandy Bridge-EP и MIC при последовательном и случайном шаблонах доступа к памяти.

	Последовательный доступ		Случайный доступ	
	Чтение	Запись	Чтение	Запись
Sandy Bridge-EP	42	19	3.3	2.2
MIC	183	160	3.8	3.4

На рис. 4 и рис. 5 видно, что алгоритм read на Sandy Bridge-EP и MIC показывает очень хорошую масштабируемость. По общей производительности MIC уже обгоняет Sandy Bridge-EP — 509 ME/s против 390 ME/s.

Подход bottom-up обладает всеми преимуществами подхода Read-based по сравнению с подходом Queue-based, позволяя при этом значительно ускорить обработку некоторых уровней, что сказывается на общей производительности, что видно на рис. 4 и рис. 5, алгоритм bottom-up. При этом так же, как и в подходе Read-based, в подходе bottom-up присутствуют случайный доступ к массиву *levels*.

3.2. Оптимизация

Чтобы понять, что же является узким местом реализации на MIC — пропускная способность или задержка случайных обращений, в алгоритм read вводится ручная раскрутка цикла на строке 9 (см. схему алгоритма read на рис. 2) и ручная преднакачка значения *levels[w]* в кэш. Для сопроцессора MIC на 1 треде производительность повысилась в 2.1 раза. Для 240 тредов производительность новой реализации, алгоритма read+prefetch, по сравнению с базовым алгоритмом read выросла на MIC на 52% до 773 ME/s, см. рис. 5. Данное улучшение результатов показывает, что узким местом алгоритма read на MIC была задержка случайных обращений в память. В качестве экспериментов проводилась развертка цикла на строке 9 рис. 2 на 2, 4, 8. Лучший результат дала развертка на 4. В новом алгоритме prefetch ограничением, по-видимому, является уже пропускная способность памяти при случайном доступе.

В то же время на микропроцессоре Sandy Bridge-EP производительность алгоритма read+prefetch совпадает с производительностью алгоритма read, что свидетельствует о том, что для алгоритма read ограничением на Sandy Bridge-EP являлась, по-видимому, пропускная способность памяти.

Для алгоритма bottom-up аналогично введена раскрутка цикла и ручная преднакачка данных в кэш, этот алгоритм на рис. 4 и рис. 5 называется bottom-up+prefetch.

Другим возможным способом повышения производительности является локализация данных даже для случайного обращения к массиву *levels*. Для повышения пространственной и временной локализации обращений к массиву *levels* матрица графа подвергается предобработке. Применяется специальный алгоритм для приведения матрицы графа к ленточной структуре и возможному уменьшению ширины этой ленты. В результате, так как строки матрицы обрабатываются в алгоритмах read и bottom-up последовательно в соответствии с последовательным движением по массиву *levels*, то возрастает количество кэш-попаданий при обращении к массиву *levels[w]*, где *w* — вершина-сосед для данной обрабатываемой вершины. Также списки вершин-соседей сортируются для уменьшения количества TLB-промахов. Алгоритм bottom-up+prefetch с указанной предобработкой называется bottom-up+prefetch+relabel. Производительность последнего алгоритма по сравнению с bottom-up+prefetch для максимального количества тредов на MIC увеличилась на 12%.

На рис. 7 представлена производительность алгоритма bottom-up+prefetch+relabel для микропроцессора Sandy Bridge-EP (16 тредов) и сопроцессора MIC (240 тредов) в зависимости от количества вершин в случайном графе, средняя связность вершины k = 8. Для графов с числом вершин до 16 млн Sandy Bridge-EP значительно превосходит MIC, что связано с эффективным использованием кэш-памяти на Sandy Bridge-EP. Производительность для MIC растет с ростом размера графа и, начиная с 32 млн вершин в графе, превышает производительность на Sandy Bridge-EP, которая на больших графах падает. Максимальный результат для MIC составляет при 134 млн вершин в графе 1080 ME/s, а для Sandy Bridge-EP — 723 ME/s, производительность на MIC превышает производительность на Sandy Bridge-EP на 49%.

Заключение

В статье рассмотрены разные подходы и алгоритмы реализации задачи поиска вширь в графе (breadth-first search, BFS). Наиболее эффективным для микропроцессора Intel Xeon Sandy Bridge-EP и сопроцессора Intel Xeon Phi оказался потоковый подход с использованием пропускной способности памяти при последовательном доступе, с сохранением при этом нерегулярного доступа к памяти. Алгоритм реализации в рамках данного подхода характеризуется отсутствием атомарных операций. Однако для сопроцессора Intel Xeon Phi для получения высокой производительности пришлось осуществлять ручные развертку цикла и преднакачку данных в кэш. В итоге максимальный результат для случайного графа из 134 млн вершин и средней связности вершины 8 для Intel Xeon Phi составляет 1080 ME/s, а для Intel Xeon Sandy Bridge-EP — 723 ME/s, Intel Xeon Phi обгоняет Intel Xeon Sandy Bridge-EP на 49%.

Производительность теста поиска вширь в графе на Intel Xeon Phi на июль 2013 года публикуются впервые. В статье [4] приведены результаты масштабирования поиска вширь в графе с ростом числа тредов на 32-ядерном прототипе Intel Xeon Phi, который носит название Knights Ferry. Однако в этой статье нет результатов абсолютной производительности и сравнения с результатами на традиционных процессорах Intel Xeon.

Реализованный в данной работе алгоритм bottom-up+prefetch использовался для получения на Intel Xeon Phi на графе с 8 млн вершин производительности 1801 ME/s, которая вошла в рейтинговый список Graph500 (июнь 2013) на 92 месте. Других результатов на Intel Xeon Phi в этом списке нет, а в классе одноузловых x86-систем этот результат является третьим среди разных исследовательских групп.

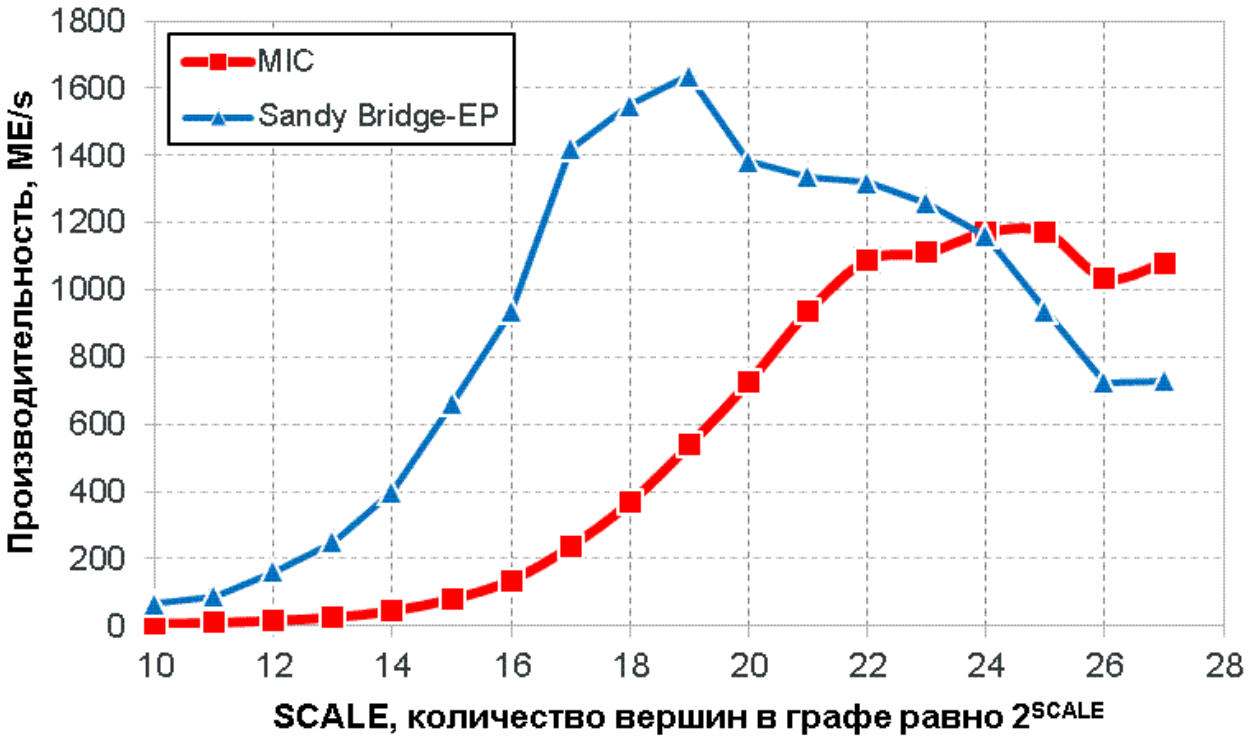

Рис. 7. Производительность алгоритма bottom-up+prefetch+relabel для Sandy Bridge-EP и MIC в зависимости от количества вершин в графе. Средняя связность вершины k = 8

ЛИТЕРАТУРА:

1. V. Agarwal, F. Petrini, D. Pasetto, D. Bader «Scalable Graph Exploration on Multicore Processors» // In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). pp.1 — 11.
2. E. Saule, U. Catalyeurek «An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture» // In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12). pp.1629 — 1639.
3. S. Hong, T. Oguntebi, K. Olukotun «Efficient Parallel Graph Exploration on Multi-Core CPU and GPU» // In Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11). pp.78 — 88.
4. S. Beamer, K. Asanović, D. Patterson «Direction-optimizing breadth-first search» // In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). 2012. 10 pages.
5. E. Saule, K. Kaya, U. Catalyurek «Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi» // arXiv:1302.1078, 5 Feb 2013. URL: http://gdcms.sysu.edu.cn/docs/20130329144208946523.pdf (дата обращения: 28.05.2013).
6. Frolov, M. Gilmendinov «DISBench: Benchmark for Memory Performance Evaluation of Multicore Multiprocessors» // Accepted to 12th International Conference, PaCT 2013, St.Petersburg, Russia, September 30-October 4, 2013.