УДК 004.434

ИСПОЛЬЗОВАНИЕ ПРОГРАММНОЙ МОДЕЛИ CHARM++ В КАЧЕСТВЕ ЦЕЛЕВОЙ ПЛАТФОРМЫ ДЛЯ КОМПИЛЯТОРА ПРОБЛЕМНО-ОРИЕНТИРОВАННОГО ЯЗЫКА ДЛЯ ОБРАБОТКИ СТАТИЧЕСКИХ ГРАФОВ

А. С. Фролов[1]

Представлена реализация модуля генерации параллельного программного кода на Charm++ в компиляторе проблемно-ориентированного языка программирования Green-Marl, предназначенного для разработки параллельных алгоритмов анализа статических графов. Приводится описание представления графа в генерируемом коде и способов отображения основных конструкций языка Green-Marl в параллельный код на Charm++. Проведенное оценочное тестирование с использованием типовых графовых задач (поиск кратчайших путей от заданной вершины до остальных вершин графа (SSSP), поиск связных компонент (CC) и вычисление рангов вершин с использованием алгоритма PageRank) показало, что производительность программ на Green-Marl, транслированных в Charm++, находится на одном уровне с реализациями на Charm++, разработанными вручную.

Ключевые слова: проблемно-ориентированные языки программирования, параллельная обработка графов, асинхронные модели вычислений.

1. Введение. Параллельный анализ статических графов с помощью высокопроизводительных вычислительных систем (суперкомпьютеров) представляет собой одну из относительно новых областей приложений, в которой работа с нерегулярными данными в значительной степени преобладает над вычислениями с плавающей запятой. Перечень ключевых проблем, возникающих при обработке больших графов с использованием суперкомпьютеров, сформулирован в [1].

Применение традиционных подходов к распараллеливанию графовых задач для систем с массово-параллельной архитектурой и распределенной памятью, таких как MPI или Shmem в комбинации с OpenMP, затруднено, так как эти модели не отражают специфики графовых задач, и, таким образом, сложность реализации графовых алгоритмов полностью переносится на пользователя (прикладного программиста). В этом контексте интерес представляют потоковые вычислительные модели, позволяющие практически прозрачно отобразить графовые вычисления. Примерами программных потоковых моделей на основе концепции активных сообщений являются Charm++ [2, 3] и Active Pebbles [4, 5]. Перспективные языки Chapel [6–8] и X10 [9, 10], предложенные для суперкомпьютерных систем следующего поколения, тоже поддерживают активные сообщения в своих программных моделях.

Еще большее повышение эффективности (продуктивности) программирования возможно за счет применения высокоуровневых проблемно-ориентированных языков программирования с передачей большей части рутинной работы по организации параллельного выполнения, включая обмен данными, синхронизацию параллельных процессов и др., распараллеливающему компилятору, а обеспечение отказоустойчивости и балансировку вычислений — системе поддержки параллельного выполнения программ (runtime-системе). На настоящий момент существует несколько проблемно-ориентированных языков программирования, предназначенных для параллельной обработки статических графов: Green-Marl [11–13], OptiGraph, Elixir [14], Falcon [15] и др. Достаточно подробный обзор таких специализированных языков приведен в работе [16].

В настоящей статье представлен результат выполнения работ по реализации поддержки Charm++ в компиляторе проблемно-ориентированного языка Green-Marl. Для достижения данной цели в компиляторе Green-Marl автором статьи разработан модуль генерации кода на Charm++, использующий внутреннее представление программы и преобразующий его в код на Charm++. Для оценки качества генерируемого кода использовались базовые графовые задачи: поиск кратчайших путей в графе (SSSP, Single Source Shortest Path), поиск связных компонент (CC, Connected Components) и вычисление рангов вершин в соответствии с алгоритмом PageRank. Исследовались их реализации на Green-Marl, а также на Charm++, выполненные вручную.

[1] Научно-исследовательский центр электронной вычислительной техники (НИЦЭВТ); Варшавское шоссе, 125, 117587, Москва; начальник отдела, e-mail: alexndr.frolov@gmail.com

Во втором разделе статьи приводится краткое описание языка Green-Marl. В третьем разделе описывается программная модель Charm++. В четвертом разделе приводится описание процесса генерации кода на Charm++ компилятором Green-Marl, в пятом разделе — результаты экспериментов и их обсуждение. В заключении сформулированы выводы выполненной работы и планы на дальнейшее развитие компилятора Green-Marl в части генерации кода на Charm++.

2. Green-Marl. Green-Marl [11–13] — проблемно-ориентированный язык, разработанный в лаборатории Pervasive Parallel Laboratory (PPL) в Стенфордском университете, США. Язык Green-Marl предназначен для разработки параллельных алгоритмов анализа статических графов. Компилятор языка поддерживает трансляцию программ на Green-Marl в параллельные модели программирования OpenMP и Pregel [20]. Поддержка трансляции в OpenMP позволяет запускать программы на Green-Marl на вычислительных системах с общей памятью. Программная модель Pregel предназначена для распределенной обработки графов, компилятором поддерживаются следующие реализации Pregel: GPS (Graph Processing System) [17] и Giraph [18]. В настоящий момент Green-Marl поддерживается и развивается в одной из исследовательских лабораторий Oracle в рамках проекта PGX.D [19].

В Green-Marl введены специальные типы для описания графа (тип `Graph`), вершин (`Node`), ребер (`Edge`), а также для описания атрибутов вершин (`N_P`<*тип атрибута*>) и ребер (`N_E`<*тип атрибута*>). Кроме обычных операторов, таких как `While`, `Do-While`, `If`, `If-Else`, определяющих последовательное выполнение программы, в Green-Marl поддерживаются операторы для описания параллельных вычислений. Для описания циклов, итерации которых могут быть выполнены параллельно и независимо друг от друга, в Green-Marl используется специальная конструкция

```
Foreach(iterator : Set)(cond) { ... }
```

Здесь `Set` — итерируемое множество (например, множество вершин графа, множество соседей заданной вершины, коллекции — упорядоченные и неупорядоченные подмножества вершин), `it` — итератор. Дополнительно может быть указано условие (cond), которое задает, какие из итераций должны быть выполнены, таким образом можно фильтровать итерации. Семантика `Foreach` предполагает, что для каждого элемента множества `Set` будут выполнены инструкции, определенные в теле цикла, при этом порядок выполнения итераций цикла не задан. Допускается использование вложенных циклов `Foreach`. Пример вложенного цикла:

```
Foreach(i : G.Nodes) {
  Foreach(j : i.nbrs) {
    Foreach(k : j.nbrs) {
    }
  }
}
```

Кроме того, в Green-Marl поддерживается параллельный цикл `For`, который отличается от `Foreach` тем, что выполнение цикла `For` с точки зрения изменения памяти всегда однозначно определено и существует эквивалентное последовательное выполнение `For`, которое даст такой же результат (т.е. цикл `For` является сериализуемым). С точки зрения консистентности памяти в Green-Marl существуют последовательная и параллельная модели. Последовательная модель соответствует участкам кода (например, `Do-While`), при этом результат выполнения инструкции, модифицирующей какую-либо переменную (или ячейку памяти), будет доступен для следующих инструкций. В параллельной модели консистентности памяти порядок выполнения записей и видимость результата гарантируется только внутри параллельного фрагмента, соответственно общий порядок выполнения всех записей не определен. Однако гарантируется, что на момент завершения параллельного участка кода все записи будут выполнены.

В Green-Marl введена поддержка редукций; для этого в теле цикла `Foreach` используются специальные операторы `+=`, `*=`, `max=`, `min=`, `&&=`, `||=`, обозначающие сложение, умножение, максимум, минимум, логическое И и логическое ИЛИ соответственно. Результат редукции может присваиваться как скалярным переменным, так и атрибутам вершин или ребер.

На рис. 1 представлена реализация на языке Green-Marl алгоритма Беллмана–Форда поиска кратчайших путей в графе от заданной корневой вершины к остальным вершинам графа.

Программа состоит из функции `sssp`, в которой определены три параметра G (тип `Graph`, т. е. непосредственно граф, в котором осуществляется поиск), атрибуты вершин `dist` (тип `N_P`) и атрибуты дуг `len` (тип `E_P`). Параметр root (тип `Node`) задает корневую вершину, от которой будут рассчитываться кратчайшие пути до других вершин графа.

Инициализация (строки 8–12) состоит в том, что для всех вершин, кроме корневой, атрибуту `dist` присваивается значение `+INF`, для корневой вершины значение `dist` устанавливается равным нулю. Кроме того, корневая вершина помечается определенным образом (`updated` равно `true`), что означает, что данная вершина будет обработана при выполнении первой итерации цикла `While` (строки 14–17). В цикле `While` происходит параллельная обработка всех вершин, для которых атрибут `updated` равен `true`: для каждой такой вершины выполняется просмотр исходящих ребер, и если $\mathrm{dist}[v] + \mathrm{weight}(v, u) < \mathrm{dist}[u]$ для ребра (v, u), то атрибуту $\mathrm{dist}[u]$ присваивается новое значение (строки 16–22). Цикл выполняется до тех пор, пока существует хотя бы одна вершина, помеченная как обновленная, для этого выполняется редукция по всем вершинам графа (строка 26). В соответствии с алгоритмом Беллмана–Форда количество итераций ограничено сверху значением $|V| - 1$.

```
1  Procedure sssp(G:Graph, dist:N_P<Int>, len:E_P<Int>,
2                 root: Node)
3  {
4    N_P<Bool> updated;
5    N_P<Bool> updated_nxt;
6    N_P<Int>  dist_nxt;
7
8    Bool fin = False;
9    G.dist = (G == root) ? 0 : +INF;
10   G.updated = (G == root) ? True: False;
11   G.dist_nxt = G.dist;
12   G.updated_nxt = G.updated;
13
14   While(!fin) {
15     fin = True;
16     Foreach(n: G.Nodes)(n.updated) {
17       Foreach(s: n.Nbrs) {
18         Edge e = s.ToEdge();
19         <s.dist_nxt; s.updated_nxt>
20           min= <n.dist + e.len; True, n>;
21       }
22     }
23     G.dist = G.dist_nxt;
24     G.updated = G.updated_nxt;
25     G.updated_nxt = False;
26     fin = ! Exist(n: G.Nodes){n.updated};
27   }
28 }
```

Рис. 1. Реализация SSSP на Green-Marl

Таким образом, Green-Marl является императивным, проблемно-ориентированным языком программирования для реализации параллельных графовых алгоритмов. Использование Green-Marl позволяет значительно упростить разработку параллельных графовых программ как за счет специализации и высокого уровня абстракции языка, так и за счет использования различных оптимизаций компилятором. Однако на данный момент в компиляторе Green-Marl отсутствует поддержка высокопроизводительных вычислительных кластеров, что ограничивает его применение для решения задач, требующих больших вычислительных ресурсов и объемов памяти.

3. Charm++. Язык параллельного программирования Charm++ [2, 3] является расширением языка C++ и основан на объектно-ориентированной модели с управлением асинхронным потоком сообщений. Ниже приводятся основные концепции программной модели Charm++. Базовым объектом в Charm++ является chare (или chare-объект), обладающий, помимо всех свойств объектов в C++ (инкапсулированные данные, множество публичных и приватных методов и др.), интерфейсом из специальных entry-методов, вызовы которых соответствуют передаче данных (т. е. сообщений) между chare-объектами.

Приложение в Charm++ состоит из множества chare-объектов, обменивающихся между собой данными посредством вызовов entry-методов. Кроме того, вызов entry-метода порождает выполнение непосредственно самого метода на том узле, где находится chare-объект, entry-метод которого был вызван, т.е. таким образом реализуется концепция управления потоком данных или активных сообщений.

В процессе выполнения entry-метода могут быть изменены только данные, принадлежащие chare-объекту. Одновременно у chare-объекта может выполняться не более одного entry-метода, что позволяет избежать проблемы обеспечения атомарности изменения данных, принадлежащих chare-объекту, и значительно упрощает программирование.

Кроме простых chare-объектов, в Charm++ имеется возможность создавать массивы chare-объектов. При этом массивы бывают как одномерными, так и многомерными. Использование массивов позволяет создавать большое количество chare-объектов, управлять их распределением по вычислительным узлам и выполнять над ними коллективные операции.

Другим важным элементом программной модели Charm++ является возможность определять момент наступления так называемого “состояния тишины”, т.е. момента в выполнении программы, когда все сообщения были доставлены и обработаны. С помощью данного механизма можно определять момент окончания выполнения асинхронных алгоритмов без каких-либо дополнительных действий со стороны программиста (например, введения счетчиков полученных сообщений и др.).

В Charm++ поддерживается автоматическая динамическая балансировка нагрузки на вычислительные узлы, что реализуется на уровне runtime-системы за счет перемещения chare-объектов между вычислительными узлами. Для прикладного программиста данная возможность совершенно прозрачна, так как программная модель Charm++ не специфицирует расположение chare-объектов в вычислительной системе, и все объекты адресуются в едином адресном пространстве.

Основные концепции модели Charm++ были разработаны почти 20 лет назад в то время, когда массово-параллельные системы (вычислительные кластеры) на основе коммерческих микропроцессоров только набирали популярность. Тем не менее, модель Charm++, вобравшая в себя как принципы dataflow-моделей, так и многопоточных (мультитредовых) моделей вычислений, оказалась достаточно успешной и с момента создания почти не претерпевала изменений.

4. Компилятор Green-Marl. Компилятор Green-Marl выполняет преобразование исходного кода программы на языке Green-Marl в один из следующих возможных вариантов эквивалентного представления программы (source-to-source трансляция): последовательный код на C++, параллельный код для многоядерных систем с общей памятью на C++ с использованием директив OpenMP, параллельный код для распределенных систем на Java с использованием библиотек GPS или Giraph. Системы Giraph [18] и GPS [17] являются реализациями программной модели Pregel [20], в основе которой лежит концепция представления программ анализа графов в вершинно-ориентированном стиле (vertex-centric) и синхронная пошаговая модель вычислений Bulk Synchronous Parallel (BSP) [21].

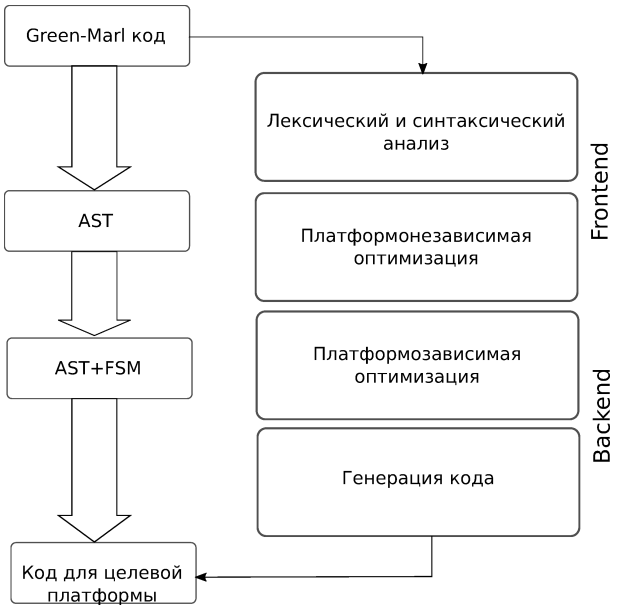


Рис. 2. Основные этапы компилятора Green-Marl

```
3  Int S = 0;
4  Int C = 0;
5  Foreach (n : G.nodes) {
6    If (n.age < K) {
7      S += n.age;
8      C += 1;
9    }
10 }
11 Float val = (C == 0) ? S / (float) C;
```

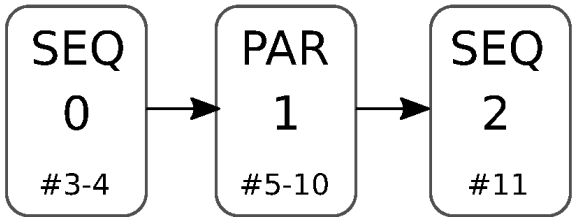


Рис. 3. Фрагмент программы на Green-Marl и соответствующий граф потока управления

Общая схема компилятора Green-Marl приведена на рис. 2. Как видно из рисунка, основные стадии трансляции программ включают в себя лексический и синтаксический анализ кода (в том числе проверку типов, преобразование синтаксических конструкций — раскрытие конструкций “синтаксического сахара” и др.), платформонезависимые оптимизации (слияние циклов, перемещение инвариантных участков кода за пределы циклов и пр.), платформозависимые оптимизации и генерация кода.

Внутреннее представление программы в компиляторе Green-Marl состоит из абстрактного синтаксического дерева (AST-дерева, Abstract Syntax Tree) и графа потока управления программы (рис. 3), узлам которого соответствуют линейные участки кода исходной или трансформированной программы. В компиляторе узлам графа потока управления соответствует структура EBB (Extended Basic Block), содержащая информацию о входящих и исходящих зависимостях, типе участка, наборе инструкций, принадлежащих соответствующему линейному участку, и используемых переменных.

Узлы графа потока управления бывают двух типов: последовательные и параллельные. Последовательным узлам (тип SEQ) соответствуют сегменты кода программы, в которых выполнение инструкций должно быть в строгой последовательности. Параллельным узлам (тип PAR) соответствуют сегменты кода, которые должны быть выполнены для всех вершин графа (соответственно, циклы `Foreach` и `For`). Граф потока управления программы может рассматриваться в качестве конечного автомата, часть состо-

яний которого предполагает выполнение для всех вершин графа.

4.1. Генератор кода Charm++. При разработке генератора кода Charm++ использовались результаты работы по портированию Green-Marl на распределенную модель вычислений Pregel, подробно описанную в [13]. Программная модель Charm++ является более общей, чем Pregel, что позволяет относительно просто реализовать весь функционал Pregel и расширяет возможности по трансляции программ на Green-Marl, не ограничиваясь только конструкциями Pregel. В частности, программная модель Charm++, как и Pregel, идеально подходит для разработки вершинно-ориентированных графовых алгоритмов; с другой стороны, механизм асинхронных активных сообщений, используемый в Charm++, значительно более гибкий, чем пошаговая синхронная схема вычислений, используемая в BSP. Указанная специфика предопределила особенность (и отличие) данной работы от работы [13].

```
Procedure count(G:Graph, age:N_P<Int>, root: Int)
{
  Int S = 0;
  Int C = 0;
  Foreach (n : G.nodes) {
    If (n.age < K) {
      S += n.age;
      C += 1;
    }
  }
  Float val = (C == 0) ? S / (float) C;
}
```

а)

```
module count {
  message __ep_state1_msg;
  readonly CProxy_count_master master_proxy;
  readonly CProxy_count_vertex vertex_proxy;
  chare count_master {
    entry count_master(const CkCallback & cb);
    entry void do_count(int K);
    entry [reductiontarget] void __reduction_S (int S);
    entry [reductiontarget] void __reduction_C (int C);
    entry void __ep_state0();
    entry void __ep_state1();
    entry void __ep_state2();
  }; // count_master
  array [1D] count_vertex {
    entry count_vertex();
    entry void __ep_state1(__ep_state1_msg *msg);
    entry void add_edge(const count_edge &e);
  }; // count_vertex
}; // count
```

б)

```
class count_master: public CBase_count_master {
  public:
    void __reduction_S (int S) { this->S = S; }
    void __reduction_C (int C) { this->C = C; }
    void __ep_state0() {
      S = 0;
      C = 0;
      thisProxy.__ep_state1();
    }
    void __ep_state1() {
      __ep_state1_msg *_msg = new __ep_state1_msg();
      _msg->K = K;
      vertex_proxy.__ep_state1(_msg);
      CkStartQD(CkIndex_count_master::__ep_state2(),
          &thishandle);
    }
    void __ep_state2() {
      val = (C == 0)?((float)S):(0 / ((float)C));
      done_callback.send();
    }
  private:
    CkCallback done_callback;
    int S, C, K;
    float val;
}; // count_master
class count_vertex: public CBase_count_vertex {
  public:
    struct vertex_properties {
      int age;
    };
  public:
    void __ep_state1 (__ep_state1_msg *msg) {
      int K = msg->K;
      delete msg;
      int S, C;
      if (this->props.age < K) {
        S = this->props.age;
        contribute(sizeof(int), &S,
          CkReduction::sum_int,
          CkCallback(CkReductionTarget(
            count_master, __reduction_S), master_proxy));
        C = 1;
        contribute(sizeof(int), &C,
          CkReduction::sum_int,
          CkCallback(CkReductionTarget(
            count_master, __reduction_C), master_proxy));
      }
    }
  private:
    std::list<struct count_edge> edges;
    struct vertex_properties props;
}; // count_vertex
```

в)

Рис. 4. Пример программы на Green-Marl (а) и сгенерированного кода на Charm++ (б, в)

Вместе с тем, для доказательства теоремы существования возможности трансляции программ на Green-Marl в параллельный код на Charm++ достаточно было реализовать заключительный этап компилятора — генерацию кода по готовому внутреннему представлению программы. Далее описывается, какие конкретно конструкции Charm++ использовались при трансляции Green-Marl программ. В качестве примера рассмотрим простую программу, определяющую средний возраст всех зарегистрированных пользователей некоторой социальности сети младше заданного значения `K`. Исходный код программы на Green-Marl и результат работы компилятора представлены на рис. 4.

Представление графа в генерируемом коде. Для представления графа в сгенерированном коде на Charm++ выбран самый простой и естественный способ — вершины графа представляются в виде отдельных chare-объектов, которые описываются классом *name*`_vertex`, где *name* — это имя процедуры в программе Green-Marl. На данный момент поддерживается трансляция Green-Marl программ, содержащих не более одной процедуры.

Для хранения атрибутов вершин создается специальная структура `vertex_properties`, элементы

которой соответствуют атрибутам вершин графа. Ребра графа хранятся в виде списка `edges`, элементами которого являются структуры с идентификаторами вершин и атрибутами ребер. Каждая вершина знает только о собственных исходящих ребрах. Вершины графа представлены в виде одномерного массива chare-объектов (`array` [1D]); так как элементы одномерного массива в Charm++ распределяются по параллельным процессам блочным способом, то вершины графа тоже будут распределены блочным способом.

В примере на рис. 4 вершины представлены классом `count_vertex` (см. рис. 4б, строки 14–18; рис. 4в, строки 26–52). Вершины графа имеют единственный атрибут `age`; таким образом, в `vertex_properties` определено единственное поле `age`. Так как для ребер графа атрибуты не определены, то `edge_properties` не имеет полей.

Построение конечного автомата программы. На основе графа потока управления строится конечный автомат. Для управления конечным автоматом создается chare-объект (*name*`_master`), в котором каждому состоянию конечного автомата соответствует entry-метод (`__ep_state_`i, где i — номер состояния). Выполнение программы начинается с того, что у *name*`_master` вызывается entry-метод `__ep_state_0`, далее выполнение осуществляется в соответствии с диаграммой состояний конечного автомата. При выполнении параллельных состояний, т.е. состояний, соответствующих базовым блокам типа PAR, происходит вызов соответствующего entry-метода у всех chare-объектов класса *name*`_vertex`. Переход к следующему состоянию осуществляется только после окончания всех вычислений, инициированных в текущем состоянии.

Переход между состояниями выполняется двумя способами: с помощью явного вызова `__ep_state_`i, если текущее состояние последовательное, т.е. все команды будут гарантированно завершены к моменту перехода к следующему состоянию, и с помощью механизма обнаружения состояния “тишины” (`CkStartQD`), если текущее состояние является параллельным.

В примере конечный автомат состоит из трех состояний (рис. 3), которым соответствуют три entry-метода класса `count_master`: `__ep_state_0`, `__ep_state_1` и `__ep_state_2`. Поскольку состояние 1 является параллельным, то ему в классе `count_vertex` соответствует entry-метод `__ep_state_1`. Переход от состояния 0 к состоянию 1 осуществляется посредством вызова `__ep_state_1` (строка 8). Переход от состояния 1 к состоянию 2 выполняется с использованием `CkStartQD` (строки 14–15). Так как состояние 3 является терминальным состоянием, то в `__ep_state_2` инициируется обратный вызов `done_callback` для возвращения к внешнему коду (или коду загрузчика Green-Marl-программ).

Глобальные переменные и редукция. В Green-Marl переменные разделяются на два типа: глобальные и локальные (для вершин). Глобальные переменные объявляются в последовательных участках кода и доступны из параллельных участков кода. Глобальные переменные реализуются в виде членов-переменных класса *name*`_master`. В случае, если к глобальной переменной выполняется операция чтения в теле цикла `Foreach`, то значение переменной передается в виде одного из параметров при вызове соответствующего entry-метода у вершин *name*`_vertex` графа. В случае, если в теле `Foreach` выполняется запись в глобальную переменную, то будет вызван соответствующий entry-метод объекта класса *name*`_master`, однако при выполнении нескольких записей из разных вершин в одну глобальную переменную в рамках одного цикла `Foreach` возникают условия гонок и результат будет не определен.

Для реализации редукции компилятором Green-Marl используется механизм `contribute`, реализованный в Charm++. В Charm++ поддерживается выполнение различных операций редукции по массиву chare-объектов. Для этого необходимо сделать вызов функции `contribute`, указать адрес локальной переменной, ее размер в байтах, адрес chare-объекта и entry-метод, который будет вызван у указанного chare-объекта, когда редукция будет завершена. При этом entry-метод должен быть помечен как `reductiontarget` и иметь единственный аргумент, в который будет записан результат выполнения редукции.

В примере на рис. 4 глобальными переменными являются `S`, `C`, `K`. Значение `K` используется внутри цикла `Foreach`, поэтому `__ep_state_2` в `count_vertex` получает значение `K` в качестве одного из полей структуры `__ep_state2_msg`, являющейся параметром. Переменные `S` и `C` используются для записи результата редукций (строки 7 и 8). В сгенерированном коде видно, что в `count_master` присутствуют два entry-метода `__reduction_S` (строка 3) и `__reduction_C` (строка 4) для записи результатов операций редукции в `S` и `C` соответственно.

Коммуникации с соседними вершинами. Очень часто при разработке графовых алгоритмов возникает необходимость анализировать данные соседних вершин. В Green-Marl эта операция легко реализуется посредством вложенных циклов. На рис. 5а приведен пример вложенного цикла `Foreach`, в котором для каждой вершины вычисляется сумма значений атрибутов `boo` соседних вершин и записыва-

ется в поле `foo` текущей вершины.

Программная модель Charm++ явным образом не поддерживает чтение атрибутов удаленных chare-объектов, и, как правило, вместо чтения используется двухстадийная операция, состоящая из двух entry-вызовов (запрос/ответ). Вместе с тем, такой подход влечет дополнительные накладные расходы, связанные с передачей большего числа сообщений. Однако возможно другое решение — преобразование вложенных циклов таким образом, чтобы вместо чтения атрибутов соседних вершин использовались записи (рис. 5б). В компиляторе данная оптимизация называется *edge flipping*. После выполнения этого преобразования компилятор может генерировать код на Charm++ без дополнительных трансформаций. Результирующий код на Charm++ представлен на рис. 5в. Итерации вложенного цикла выполняются посредством вызовов entry-метода `__ep_state_1_recv`.

```
N_P<Int> foo;
N_P<Int> boo;
Foreach(i : G.Nodes) {
  Foreach(j: i.Nbrs)
    i.foo += j.boo;
}
```

а)

```
N_P<Int> foo;
N_P<Int> boo;
Foreach(i : G.Nodes) {
  Foreach(j: i.Nbrs)
    j.foo += i.boo;
}
```

б)

```
class vertex : public CBase_vertex {
 ...
 /*entry*/ void __ep_state_1() {
   int _m0;
   typedef std::list<struct example2_edge>::iterator Iterator;
   for (Iterator _e = edges.begin(); _e != edges.end(); _e++) {
     // Sending messages to each neighbor
     __vertex_ep_bb2_recv_msg *_msg = new __vertex_ep_bb2_recv_msg();
     _m0 = this->props.boo;
     _msg->_m0 = _m0;
     thisProxy[_e->v].__vertex_ep_bb2_recv(_msg);
   }
 }
 /*entry*/ void __ep_state_1_recv(__ep_state_1_recv_msg *msg) {
   int _m0 = msg->m0;
   delete msg;
   this->props.foo = this->props.foo + (_m0);
 }
 ...
};
```

в)

Рис. 5. Исходный вложенный цикл `Foreach` (а), эквивалентный цикл, полученный в результате преобразования, (б) и фрагмент сгенерированного кода на Charm++ (в)

Отметим, что ограничение, связанное с отсутствием возможности чтения значений атрибутов соседних вершин, характерно не только для Charm++, но и для модели Pregel (и ее реализаций Giraph и GPS), поэтому данное преобразование циклов уже было реализовано в компиляторе Green-Marl для соответствующих генераторов кода, что позволило использовать его и для генератора кода на Charm++. Подробно о преобразованиях циклов, реализованных в компиляторе Green-Marl, написано в работе [13].

```
Procedure cc (G:Graph, CC: N_P<Node>) {
  Bool fin = False;
  N_P<Bool> updated;
  N_P<Bool> updated_next;

  Foreach (i: G.Nodes) {
    i.CC = i;
    i.updated = True;
    i.updated_next = False;
  }

  While (!fin) {
    fin = True;
    Foreach (i: G.Nodes)(i.updated) {
      Foreach (j: i.Nbrs)
        If (i.CC < j.CC) {
          j.CC = i.CC;
          j.updated_next = True;
        }
    }
    G.updated = G.updated_next;
    G.updated_next = False;

    fin = ! Exist(n: G.Nodes){n.updated};
  }
}
```

а)

```
Procedure pagerank(G: Graph, e,d: Double, max: Int;
          pg_rank: Node_Prop<Double>)
{
  Double diff;
  Int cnt = 0;
  Double N = G.NumNodes();
  G.pg_rank = 1 / N;
  Do {
    diff = 0.0;
    Foreach (t: G.Nodes) {
      Double val = (1-d) / N + d*
        Sum(w: t.InNbrs) {
          w.pg_rank / w.OutDegree()} ;

      diff += | val - t.pg_rank |;
      t.pg_rank <= val @ t;
    }
    cnt++;
  } While ((diff > e) && (cnt < max));
}
```

б)

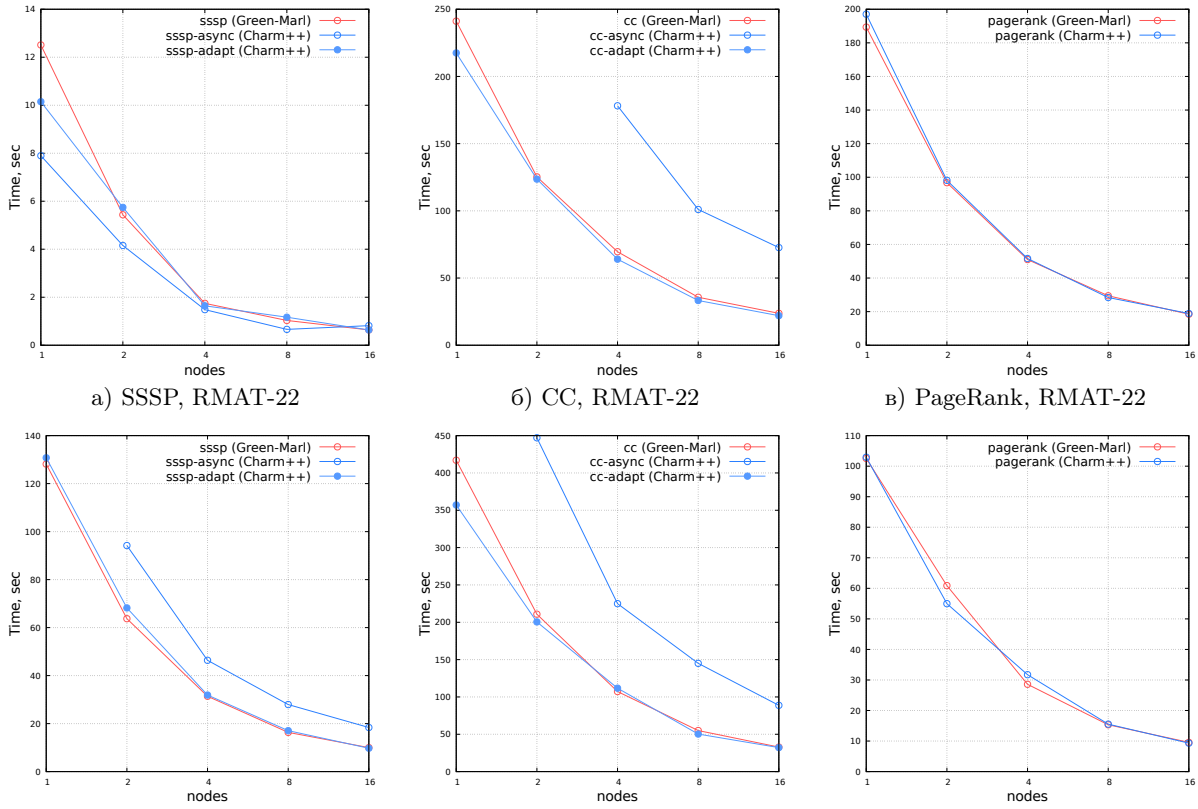
Рис. 6. Реализация CC (а) и PageRank (б) на Green-Marl

5. Оценка эффективности. Набор тестов для оценки эффективности кода, генерируемого компилятором Green-Marl, включает в себя тесты SSSP — поиск кратчайших путей в графе от заданной вершины до остальных вершин графа, CC — поиск связных компонент в графе и PageRank — вычисление

рангов вершин в соответствии с алгоритмом PageRank. Код на Green-Marl для SSSP приведен на рис. 1, для CC и PageRank — на рис. 6.

Для тестирования использовались синтетические графы двух типов: RMAT [22] и Random. RMAT-графы были предложены в качестве синтетического приближения к графам реального мира, характеризующимся степенным законом распределения степени вершин (например, социальные сети и др.). Для генерации RMAT-графов использовался генератор из теста Graph500. Random-графы — это графы с равномерным распределением ребер по вершинам графа.

Для оценки эффективности полученного в результате трансляции кода время выполнения работы программ на Green-Marl сравнивалось со временем выполнения референсных программ, реализованных вручную на Charm++ с использованием вершинно-ориентированного подхода. Запуск тестов проводился на 36-узловом кластере Ангара-К1, установленном в АО "НИЦЭВТ" на базе коммуникационной сети Ангара [23, 24], при этом использовалось до 16 узлов из A-сегмента кластера Ангара-К1 (в узлах A-сегмента используются 6-ядерные процессоры Xeon E5-2660 и 80 ГБ DDR3-памяти). Во всех запусках использовалось 8 процессорных ядер на узел (т.е. ppn равно 8).



а) SSSP, RMAT-22 б) CC, RMAT-22 в) PageRank, RMAT-22

г) SSSP, Random-22 д) CC, Random-22 е) PageRank, Random-22

Рис. 7. Результаты измерения производительности для тестов SSSP, CC и RandomAccess

Результаты исследования производительности полученных реализаций представлены на рис. 7. Во всех случаях использовались графы размером 2^{22}. Для запусков SSSP и PageRank использовались ориентированные графы, для CC — неориентированные.

Для SSSP (так же, как и для CC) использовались следующие две референсные реализации, отличающиеся алгоритмом:

— `sssp-async` (`cc-async`): наивная реализация на Charm++, основанная на полностью асинхронной схеме вычислений: вершины выполняют обновление значений атрибутов (`dist` для SSSP и `CC` для CC) до тех пор, пока возможно хотя бы одно обновление (т.е. пока не будет обнаружено состояние "тишины");

— `sssp-adapt` (`cc-adapt`): адаптированная реализация на Charm++, приближенная к алгоритму, описанному в Green-Marl-реализации.

Основное отличие заключается в том, что в адаптированной реализации используется синхронизация по фронту вершин, т.е. обновляются только вершины, соседние с вершинами, принадлежащими фронту (в реализации на Green-Marl это обеспечивается за счет внешнего цикла `While`). Таким образом, в случае

адаптированной реализации можно говорить о частично-асинхронной схеме вычислений: в рамках одной итерации асинхронно обновляются только соседние вершины, далее выполняются глобальная барьерная синхронизация и переход к следующей итерации.

Как видно из рис. 7, для SSSP и CC производительность реализаций, выполненных на Green-Marl, практически совпадает с производительностью референсных реализаций на Charm++, в которых используется максимально близкий алгоритм. При этом для CC адаптированная версия (`cc-adapt`) показывает более высокую эффективность по сравнению с асинхронной (`cc-async`). Такой результат может показаться неожиданным. Возможны следующие причины: во-первых, вместе с ограничением параллелизма в адаптированной версии ограничивается и количество сообщений, одновременно порождающихся в процессе выполнения, что снижает как накладные расходы на runtime-систему, так и максимальный объем используемой памяти (что тоже может влиять на производительность); во-вторых, в асинхронной реализации CC общее количество сообщений может быть больше из-за эффекта спекулятивного продвижения меток (т.е. продвижения меток, которые не являются минимальными для данной компоненты). В адаптированной версии алгоритма происходит пошаговая синхронизация, что тоже влечет накладные расходы, однако продвижение меток осуществляется равномерно по всем вершинам.

Для теста PageRank различие по времени выполнения версии, полученной с помощью компилятора, и версии, реализованной вручную, оказалась минимальной. В первую очередь, это объясняется тем, что в обеих версиях использовался максимально близкий алгоритм.

Итак, оценочное тестирование показало, что, во-первых, при использовании близких алгоритмов эффективность генерируемого компилятором кода и кода, разработанного вручную, практически одинакова; во-вторых, частично-асинхронная схема организации вычислений, которая используется при трансляции вложенных циклов `Foreach`, в некоторых случаях показывает более высокую эффективность, чем полностью асинхронная схема.

6. Заключение. В настоящей статье представлен генератор кода на Charm++, разработанный для компилятора проблемно-ориентированного языка программирования Green-Marl, предназначенного для анализа статических графов. Таким образом, к существующим системам параллельного выполнения программ (OpenMP, GPS, Giraph), поддерживаемых компилятором Green-Marl в качестве целевых платформ, добавлена асинхронная программная модель Charm++, что расширяет диапазон применения Green-Marl и делает возможным запуск программ на Green-Marl на высокопроизводительных (HPC) вычислительных кластерах.

Исследование производительности генерируемого кода показало, что программы на Charm++, полученные посредством компиляции из Green-Marl, не уступают и в отдельных случаях даже превосходят по производительности программы на Charm++, разработанные вручную. В дальнейшем планируется улучшить качество генерируемого кода за счет использования библиотеки агрегации сообщений Charm++ [25].

Работа выполнена при поддержке гранта РФФИ № 15–07–09368.

СПИСОК ЛИТЕРАТУРЫ

1. *Lumsdaine A., Gregor D., Hendrickson B., Berry J.* Challenges in parallel graph processing // Parallel Processing Letters. 2007. **17**, N 1. 5–20.
2. *Kale L.V., Krishnan S.* CHARM++: a portable concurrent object oriented system based on C++ // ACM SIGPLAN Not. 1993. **28**, N 10. 91–108.
3. *Zheng G., Meneses E., Bhatele A., Kale L.V.* Hierarchical load balancing for Charm++ applications on large supercomputers // Proc. 39th International Conference on Parallel Processing Workshops. Washington, DC: IEEE Press, 2010. 436–444.
4. *Willcock J.J., Hoefler T., Edmonds N.G., Lumsdaine A.* Active pebbles: parallel programming for data-driven applications // Proc. International Conference on Supercomputing. New York: ACM Press, 2011. 235–244.
5. *Willcock J.J., Hoefler T., Edmonds N.G., Lumsdaine A.* Active pebbles: a programming model for highly parallel fine-grained data-driven computations // ACM SIGPLAN Not. 2011. **46**, N 8. 305–306.
6. *Callahan D., Chamberlain B.L., Zima H.P.* The cascade high productivity language // Proc. 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments. Washington, DC: IEEE Press, 2004. 52–60.
7. *Chamberlain B.L, Callahan D., Zima H.P.* Parallel programmability and the chapel language // International Journal of High Performance Computing Applications. 2007. **21**, N 3. 291–312.
8. *Haque R., Richards D.* Optimizing PGAS overhead in a multi-locale chapel implementation of CoMD // Proceedings of the First Workshop on PGAS Applications. Piscataway: IEEE Press, 2016. 25–32.
9. *Charles P., Grothoff C., Saraswat V., Donawa C., Kielstra A., Ebcioglu K., Von Praun C., Sarkar V.* X10: an object-oriented approach to non-uniform cluster computing // ACM Sigplan Notices. 2005. **40**, N 10. 519–538.
10. *Tardieu O., Herta B., Cunningham D., Grove D., Kambadur P., Saraswat V., Shinnar A., Takeuchi M., Vaziri M.,*

Zhang W. X10 and APGAS at petascale // ACM Transactions on Parallel Computing. 2016. **2**, N 4. doi 10.1145/2894746.
11. *Hong S., Chafi H., Sedlar E., Olukotun K.* Green-Marl: a DSL for easy and efficient graph analysis // ACM SIGARCH Computer Architecture News. 2012. **40**, N 1. 349–362.
12. *Hong S., Van Der Lugt J., Welc A., Raman R., Chafi H.* Early experiences in using a domain-specific language for large-scale graph analysis // Proc. First International Workshop on Graph Data Management Experiences and Systems. New York: ACM Press, 2013. doi 10.1145/2484425.2484430.
13. *Hong S., Salihoglu S., Widom J., Olukotun K.* Simplifying scalable graph processing with a domain-specific language // Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. New York: ACM Press, 2014. doi 10.1145/2544137.2544162.
14. *Prountzos D., Manevich R., Pingali K.* Elixir: a system for synthesizing concurrent graph programs // ACM SIGPLAN Notices. 2012. **47**, N 10. 375–394.
15. *Cheramangalath U., Nasre R., Srikant Y.* Falcon: a graph manipulation language for heterogeneous systems // ACM Transactions on Architecture and Code Optimization. 2016. **12**, N 4. doi 10.1145/2842618.
16. *Фролов А.С., Семенов А.С.* Обзор проблемно-ориентированных языков программирования для параллельного анализа статических графов // Вычислительные нанотехнологии. 2017. № 1 (принято в печать).
17. *Salihoglu S., Widom J.* GPS: a graph processing system // Proceedings of the 25th International Conference on Scientific and Statistical Database Management. New York: ACM Press, 2013. doi 10.1145/2484838.2484843.
18. *Schelter S.* Large scale graph processing with apache giraph // Invited talk at GameDuell. Berlin 29th May. 2012.
19. *Hong S., Depner S., Manhardt T., Van Der Lugt J., Verstraaten M., Chafi H.* PGX.D: a fast distributed graph processing engine // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York: ACM Press, 2015. doi 10.1145/2807591.2807620.
20. *Malewicz G., Austern M.H, Bik A.J.C., Dehnert J.C., Horn I., Leiser N., Czajkowski G.* Pregel: a system for large-scale graph processing // Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2010. 135–146.
21. *Cheatham T., Fahmy A., Stefanescu D., Valiant L.* Bulk synchronous parallel computing — a paradigm for transportable software // Tools and Environments for Parallel and Distributed Systems. New York: Springer, 1996. 61–76.
22. *Chakrabarti D., Zhan Y., Faloutsos C.* R-MAT: a recursive model for graph mining // Proc. SIAM Int. Conf. on Data Mining. Philadelphia: SIAM Press, 2004. 442–446.
23. *Жабин И.А., Макагон Д.В., Поляков Д.А., Симонов А.С. Сыромятников Е.Л., Щербак А.Н.* Первое поколение высокоскоростной коммуникационной сети "Ангара" // Наукоемкие технологии. 2014. № 1. 21–27.
24. *Агарков А.А., Исмагилов Т.Ф., Макагон Д.В., Семенов А.С., Симонов А.С.* Результаты оценочного тестирования отечественной высокоскоростной коммуникационной сети Ангара // Тр. Международной конференции "Суперкомпьютерные дни в России". М.: Изд-во Моск. ун-та, 2016. 626–639.
25. *Wesolowski L., Venkataraman R., Gupta A., Yeom J.-S., Bisset K., Sun Y., Jetley P., Quinn T.R., Kale L.V.* TRAM: optimizing fine-grained communication with topological routing and Aggregation of messages // Proceedings of the 43rd International Conference on Parallel Processing. Piscataway: IEEE Press, 2014. 211–220.

Поступила в редакцию
24.01.2017

Application of the CHARM++ Software Model as a Target Platform for a Domain-Specific Language Compiler for the Analysis of Static Graphs

A. S. Frolov [1]

[1] *Scientific Research Center for Electronic Computer Technology; Varshavskoe shosse 125, Moscow, 117587, Russia; Head of Department, e-mail: alexndr.frolov@gmail.com*

Received January 24, 2017

Abstract: The implementation of a code generation mechanism in the domain-specific language (DSL) Green-Marl compiler targeted in the Charm++ framework is presented. Green-Marl is used for the parallel static graph analysis and adopts an imperative shared memory programming model, whereas Charm++ implements a message-driven execution model. The graph representation in the generated Charm++ code and the translation of the basic Green-Marl constructs to Charm++ are described. The evaluation of the typical graph algorithms: Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank shows that the performance of Green-Marl programs translated to Charm++ is the same as for native Charm++ implementations.

Keywords: domain-specific programming languages, parallel graph processing, asynchronous computation models.

References

1. A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in Parallel Graph Processing," Parallel Process. Lett. **17** (1), 5–20 (2007).

2. L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," ACM SIGPLAN Not. **28** (10), 91–108 (1993).

3. G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers," in *Proc. 39th Int. Conf. on Parallel Processing Workshops, San Diego, USA, September 13–16, 2010* (IEEE Press, Washington, DC, 2010), pp. 436–444.

4. J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active Pebbles: Parallel Programming for Data-Driven Applications," in *Proc. Int. Conference on Supercomputing, Tucson, USA, May 31–June 4, 2011* (ACM Press, New York, 2011), pp. 235–244.

5. J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active Pebbles: A Programming Model for Highly Parallel Fine-Grained Data-Driven Computations," ACM SIGPLAN Not. **46** (8), 305–306 (2011).

6. D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade High Productivity Language," in *Proc. 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments, Santa Fe, USA, April 26–26, 2004* (IEEE Press, Washington, DC, 2004), pp. 52–60.

7. B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel Programmability and the Chapel Language," Int. J. High Perform. Comput. Appl. **21** (3), 291–312 (2007).

8. R. Haque and D. Richards, "Optimizing PGAS Overhead in a Multi-Locale Chapel Implementation of CoMD," in *Proc. First Workshop on PGAS Applications, Salt Lake City, USA, November 13–18, 2016* (IEEE Press, Piscataway, 2016), pp. 25–32.

9. P. Charles, C. Grothoff, V. Saraswat, et al., "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing," ACM SIGPLAN Not. **40** (10), 519–538 (2005).

10. O. Tardieu, B. Herta, D. Cunningham, et al., "X10 and APGAS at Petascale," ACM Trans. Parallel Comput. **2** (4) (2016). doi 10.1145/2894746

11. S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for Easy and Efficient Graph Analysis," ACM SIGARCH Comput. Archit. News **40** (1), 349–362 (2012).

12. S. Hong, J. Van Der Lugt, A. Welc, et al., "Early Experiences in Using a Domain-Specific Language for Large-Scale Graph Analysis," in *Proc. First Int. Workshop on Graph Data Management Experiences and Systems, New York, USA, June 23–23, 2013* (ACM Press, New York, 2013). doi 10.1145/2484425.2484430

13. S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying Scalable Graph Processing with a Domain-Specific Language," in *Proc. Annual IEEE/ACM Int. Symp. on Code Generation and Optimization, Orlando, USA, February 15–19, 2014* (ACM Press, New York, 2014). doi 10.1145/2544137.2544162

14. D. Prountzos, R. Manevich, and K. Pingali, "Elixir: A System for Synthesizing Concurrent Graph Programs," ACM SIGPLAN Not. **47** (10), 375–394 (2012).

15. U. Cheramangalath, R. Nasre, and Y. N. Srikant, "Falcon: A Graph Manipulation Language for Heterogeneous Systems," ACM Trans. Archit. Code Optim. **12** (4) (2016). doi 10.1145/2842618

16. A. S. Frolov and A. S. Semenov, "A Survey of Object-Oriented Programming Languages for Parallel Analysis of Static Graphs," Vychisl. Nanotekhnol. No. 1, 2017 (in press).

17. S. Salihoglu and J. Widom, "GPS: A Graph Processing System," in *Proc. 25th Int. Conf. on Scientific and Statistical Database Management, Baltimore, USA, July 29–31, 2013* (ACM Press, New York, 2013). doi 10.1145/2484838.2484843

18. S. Schelter, *Large Scale Graph Processing with Apache Giraph*, invited talk at GameDuell, Berlin, 29th May 2012.

19. S. Hong, S. Depner, T. Manhardt, et al., "PGX.D: A Fast Distributed Graph Processing Engine," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Austin, USA, November 15–20, 2015* (ACM Press, New York, 2015). doi 10.1145/2807591.2807620

20. G. Malewicz, M. H. Austern, A. J. C. Bik, et al., "Pregel: A System for Large-Scale Graph Processing," in *Proc. ACM SIGMOD Int. Conf. on Management of Data, Indianapolis, USA, June 06–10, 2010* (ACM Press, New York, 2010), pp. 135–146.

21. T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, "Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software," in *Tools and Environments for Parallel and Distributed Systems* (Springer, New York, 1996), pp. 61–76.

22. D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proc. SIAM Int. Conf. on Data Mining, Lake Buena Vista, USA, April 22–24, 2004* (SIAM Press, Philadelphia, 2004), pp. 442–446.

23. I. A. Zhabin, D. V. Makagon, D. A. Polyakov, et al., "First Generation of Angara High-Speed Interconnection Network," Naukoemkie Tekhnol., No. 1, 21–27 (2014).

24. A. A. Agarkov, T. F. Ismagilov, D. V. Makagon, et al., "Performance Evaluation of the Angara Interconnect," in *Proc. Int. Conf. on Russian Supercomputing Days, Moscow, Russia, September 26–27, 2016* (Mosk. Gos. Univ., Moscow, 2016), pp. 626–639.

25. L. Wesolowski, R. Venkataraman, A. Gupta, et al., "TRAM: Optimizing Fine-Grained Communication with Topological Routing and Aggregation of Messages," in *Proc. 43rd Int. Conf. on Parallel Processing, Minneapolis, USA, September 9–12, 2014* (IEEE Press, Piscataway, 2014), pp. 211–220.