3. ТЕХНОЛОГИИ ВЫЧИСЛИТЕЛЬНОЙ ОБРАБОТКИ

3.1. ОБЗОР ПРОБЛЕМНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ ДЛЯ ПАРАЛЛЕЛЬНОГО АНАЛИЗА СТАТИЧЕСКИХ ГРАФОВ[1]

Фролов Александр Сергеевич, начальник отдела, АО «НИЦЭВТ», e-mail: frolov@nicevt.ru

Семенов Александр Сергеевич, канд. техн. наук, начальник сектора, АО «НИЦЭВТ», e-mail: semenov@nicevt.ru

Аннотация: В статье содержится обзор подходов к реализации проблемно-ориентированных языков программирования на примере Green-Marl, OptiGraph, Elixir и Falcon, предназначенных для анализа статических графов. Представлены основные особенности рассматриваемых языков, а также их сравнение с точки зрения выразительности параллелизма и применимости для генерации высокоэффективных параллельных программ для суперкомпьютеров и кластерных систем.

Ключевые слова: проблемно-ориентированные языки программирования, параллельная обработка графов, программные модели, суперкопьютеры.

A SURVEY OF DOMAIN-SPECIFIC LANGUAGES FOR PARALLEL STATIC GRAPH ANALYSIS

Frolov Alexander Sergeevich, head of department, JSC NICEVT, e-mail: frolov@nicevt.ru

Semenov Alexander Sergeevich, Ph.D., head of sector, JSC NICEVT, e-mail: semenov@nicevt.ru

Abstract: In the paper a survey of domain-specific languages (DSL) for parallel static graph analysis is presented. The survey includes Green-Marl, OptiGraph, Elixir, and Falcon. A general overview as well as most important aspects of syntax of the selected DSLs are presented in the analysis.

Index terms: domain-specific languages, parallel graph computation, program models, supercomputers.

Введение

Анализ статических графов используется в ряде прикладных областей, таких как анализ веб-графов, анализ социальных сетей, биоинформатика, информационная безопасность и др. Достаточно часто реальные графы могут достигать очень больших размеров, так что обработка таких графов требует огромных вычислительных ресурсов и использования технологий распределенных и/или параллельных вычислений. В контексте наметившейся тенденции на сближение технологий обработки больших данных (Big Data) и параллельных вычислений с применением высокопроизводительных вычислительных комплексов (HPC) [1], параллельная обработка больших графов является той прикладной областью, где слияние этих двух технологических парадигм проявляет себя наиболее заметно.

Как и любая другая инженерно-техническая дисциплина, параллельное программирование требует определенной квалификации, которую можно получить только имея реальный практический опыт. С другой стороны, специалисты по анализу данных, как правило, не имеют такого опыта, поскольку их основная задача – это разработка математических моделей и алгоритмов анализа данных, в частности с применением теории графов.

Таким образом, между конечными пользователями и существующими технологиями параллельной обработки графов существует разрыв, ликвидировать который призваны специализированные фреймворки, такие как Pregel [2], GraphLab [3], GraphX [4] и др., и проблемно-ориентированные языки программирования (DSL).

Отличие подходов в том, что в DSL используются новые синтаксические конструкции, которые могут быть реализованы как «внешний» язык, то есть с привлечением внешнего компилятора DSL, так и с помощью перегрузки операторов базового языка, тогда говорят о встраивании DSL. Второй способ часто применяется при реализации DSL на базе функциональных языков, например Scala.

Графовые фреймворки, такие как Pregel, GraphLab, GraphX, как правило, ограничиваются предоставлением простого и удобного интерфейса (API), но с достаточно мощной поддержкой со стороны runtime-системы.

В данной статье мы рассматриваем несколько существующих DSL для анализа статических графов с применением технологии параллельных (OpenMP, MPI, CUDA и др.) и распределенных (Hadoop [5], Giraph [6], Spark [7] и др.) вычислений. Для анализа были выбраны несколько существующих DSL, в частности Green-Marl [11,12], OptiGraph, Elixir [22] и Falcon [25]. Все перечисленные DSL являются относительно новыми академическими разработками.

Отметим также, что кроме DSL для анализа статических графов существуют проблемно-ориентированные языки для построения алгоритмов обходов графа или, что одно и то же, запросов к графам. Примерами таких языков являются SPARQL [8], Cypher [9], Gremlin [10] и др. Рассмотрение подобных DSL выходит за рамки данной статьи.

[1] Работа выполнена при поддержке РФФИ, грант 15-07-09368.

Далее приводится описание рассматриваемых DSL (Green-Marl, OptiGraph, Elixir и Falcon) и их сравнение в виде сводной таблицы с основными свойствам языков и их компиляторов. Также приводятся рассуждения относительно перспективности применения языков и их компиляторов для разработки инструментальной среды анализа больших статических графов с использованием суперкомпьютерных комплексов.

Green-Marl

Green-Marl [11-15] – проблемно-ориентированный язык разработки параллельных программ анализа графов, разработанный в научно-исследовательской лаборатории PPL в Стенфордском университете. Язык Green-Marl предназначен для разработки параллельных алгоритмов анализа статических графов.

Язык Green-Marl является внешним DSL-языком с собственным компилятором. Компилятор поддерживает трансляцию программ на Green-Marl в следующие параллельные модели программирования: OpenMP – для многоядерных систем с общей памятью, и Pregel – для распределенных систем (в качестве реализации Pregel поддерживаются GPS [17] и Giraph). Компилятор Green-Marl продолжает развиваться в одной из исследовательских лабораторий Oracle в рамках проекта PGX.D [18], но уже как коммерческий проект.

В Green-Marl введены специальные типы для описания графа (тип Graph), вершины (Node), ребра (Edge), а также для описания атрибутов вершин (N_P<тип атрибута>) и ребер (N_E<тип атрибута>). Можно указывать соседей заданных вершин (Nbrs).

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) { … }

Где Set – итерируемое множество (например, множество вершин графа, множество соседей заданной вершины, коллекции – упорядоченные и неупорядоченные подмножества вершин), it – итератор. Дополнительно может быть определено условие (cond), которое задает какие из итераций должны быть выполнены, то есть можно фильтровать итерации.

Семантика Foreach предполагает, что для каждого элемента множества Set будут выполнены операции, определенные в теле цикла, при этом порядок выполнения итераций цикла не задан. Допускается использование вложенных Foreach-циклов, при этом операции fork/join для разных витков цикла выполняются независимо. Пример вложенного цикла:

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

Также в Green-Marl поддерживается 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 (то есть цикл For является сериализуемым).

С точки зрения консистентности памяти в Green-Marl выделяются последовательная и параллельная модели. Последовательная модель соответствует последовательным 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 для каждого процесса в отдельности, соответственно, общий порядок выполнения обращений к памяти для всех процессов 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, уножение, максимум, минимум, логическое И и логическое ИЛИ, соответственно. Результат редукции может присваиваться как скалярным переменным, так и атрибутам вершин или ребер. Простейший пример редукции в Green-Marl:

```
int sum = 0;
Foreach (i : G.Nodes) {
 sum += i.x;
}
```

Кроме того, в Green-Marl поддерживаются встроенные итераторы, реализующие параллельные обходы DFS и BFS, что сильно упрощает разработку алгоритмов, так как данные операции (обход вширь и обход вглубь), часто встречаются в различных алгоритмах. Синтаксически обход вширь выглядит следующим образом:

```
inBFS (iterator : G.Nodes [from|;] root)
  ((filter_expr)) ([navigator_expr]) { statement_block1 }
  (inReverse ((filter2_expr)) { statement_block2 })
```

Для выполнения обхода задается корневая вершина (root), могут быть заданы фильтр для отбрасывания не нужных вершин (filter_expr) и навигатор для задания точки останова обхода (navigator_expr); если навигатор не задан, то обход завершается только после посещения всех достижимых вершин. При выполнении обхода для каждой анализируемой вершины (если значение фильтра для данной вершины истинно) выполняется заданный блок инструкций (statement_block1). Также можно описать действия, которые будут выполнены при обратном обходе (statement_block2). Обход вглубь (DFS) определяется аналогичным образом.

В Green-Marl не поддерживается изменение структуры графа (т. е. добавление и удаление вершин и дуг).

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 вычисляться кратчайшие пути до других вершин графа.

```
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. SSSP на Green-Marl.

Инициализация (строки 8-12) состоит в том, что для всех вершин кроме корневой атрибуту dist присваивается значение +INF, для корневой вершина значение dist устанавливается равным нулю. Также корневая вершина помечается определенным образом (атрибуту updated присваивается значение true), это 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 *dist[v] + weight(v,u) < dist[u]* для ребра *(v,u)*, то атрибуту *dist[u]* присваивается новое значение (строки 16-22). Такая операция называется релаксацией по ребру *(v,u)*. Цикл выполняется до тех пор, пока существует хотя бы одна вершина, помеченная как обновленная, для этого выполняется редукция по всем вершинам графа (строка 16). В соответствии с алгоритмом Беллмана-Форда количество итераций ограничено сверху значением *|V|-1*.

Представление графа (т.е. то, каким образом граф будет храниться, а также как будут храниться атрибуты вершин и ребер) определяется компилятором языка и конкретной платформой, в которую осуществляется трансляция. При трансляции программ в модель Pregel граф представляется стандартным (и единственным) способом в виде вершин, которые обладают атрибутами, состоянием (активное и неактивное) и методом compute, описывающим программу вершины. Более подробную информацию про вершинно-центральную модель программирования Pregel можно найти в [2], а про трансляцию Green-Marl в Pregel – в [14].

OptiGraph

Язык OptiGraph – DSL для анализа статических графов, основанный на спецификации Green-Marl. Язык OptiGraph, также как и Green-Marl, разработан в лаборатории PPL в Стенфордском Университете. Концептуальных нововведений с точки зрения языка в OptiGraph нет.

В отличие от Green-Marl, OptiGraph – встроенный DSL, в качестве базового языка используется функциональный язык Scala. OptiGraph разработан с использованием систем Delite [20] и Forge [21], предназначенных для разработки проблемно-ориентированных языков, также разрабатываемых в Стенфордском Университете. Система разработки высокоэффективных проблемно-ориентированных языков Delite включает в себя генераторы кода для многоядерных систем с общей памятью (OpenMP), графических ускорителей (OpenCL, CUDA). В процессе разработки находятся генераторы кода для вычислительных кластеров, а также для ПЛИС (Verilog).

Elixir

Другим примером проблемно-ориентированного языка, предназначенного для разработки и реализации параллельных алгоритмов анализа статических графов, является Elixir [22]. Язык Elixir разработан в Техасском университете в Остине. В Elixir используются как декларативные, так и императивные конструкции для определения вычислений над графом. В Elixir не поддерживается структурная трансформация графа (т. е. добавление и удаление вершин и дуг).

С точки зрения реализации язык Elixir является внешним DSL, то есть для Elixir разработана своя атрибутная грамматика и транслятор, преобразующий программу на Elixir в параллельный код на C++ с вызовами системы поддержки параллельного выполнения программ Galoios [23]. В данный момент реализована поддержка многоядерных вычислительных систем с общей памятью.

Основной особенностью Elixir является разделение операций над графом (в терминах программной модели Elixir – операторов) и планирования выполнения операций, то есть определения порядка выполнения операторов (в терминах Elixir – планировщиков).

Операторы в Elixir представляются следующим образом:

$$op = [P^{op}, Gd^{op}] \rightarrow [Upd^{op}],$$

где P^{op}– это структурный шаблон (*redex*-патерн), по которому будет проводится поиск подграфов, Gd^{op} – это условное выражение, в котором могут быть использованы переменные, определенные в P^{op}. Upd^{op}– последовательность операций, модифицирующих значения атрибутов вершин, подграфа удовлетворяющего P^{op} и Gd^{op}.

Для задания способа применения операторов к графу используются следующие выражения: foreach op, for i=low..high op, iterate op, где op – это оператор. Выражение foreach op однократно применяет оператор ко всем сопоставленным подграфам, for i=low..high op применяет оператор заданное количество раз, iterate op применяет оператор до тех пор пока имеется хотя бы одно корректное сопоставление.

Для определения порядка выполнения операций выборки подграфов, используются планировщики. В Elixir поддержаны статические и динамические политики планирования:

- metric e (и approx metric e) – определяет строгий (и приблизительный, то есть допускающий нарушения) порядок обработки сопоставленных подграфов в соответствии с заданной метрикой в виде выражения *e* (чем меньше значение метрики, тем выше приоритет сопоставления);

- group V – сопоставления, включающие вершины V, должны быть обработаны вместе, то есть их взаимное расположение в списке очередности сопоставление должно быть как можно более близким друг к другу, данная оптимизация позволяет улучшить пространственно-временную локализацию выборки вершин графа;

- unroll k – сопоставления, образующие цепочки длины *k*, обрабатываются последовательно друг за другом аналогично развертке циклов в императивных языках программирования;

- (op1 or op2) >> fuse – преобразование сопоставленных подграфов по шаблонам op1 и op2 выполняется одновременно, то есть если g – подграф, сопоставленный шаблонам $[P^{op1}, Gd^{op1}]$ и $[P^{op2}, Gd^{op2}]$, то для такого подграфа последовательно будут вы-

полнены преобразования Upd^{op1}и Upd^{op2} (при условии что выполнение Upd^{op1} не нарушит Gd^{op1}).

Пример реализации SSSP на Elixir приведен на рис. 2. Определение графа (строки 1-2) состоит из описания вершин (nodes) и ребер (edges). Каждая вершина включает два атрибута node – идентификатор вершины (Node) и dist – атрибут (тип int), определяющий расстояние данной вершины от корневой вершины. Ребра графа определяются кортежем, состоящем из начальной вершины (src), конечной вершины (dst) и веса ребра (wt).

Идентификатор корневой вершины (source) определяется отдельно (строка 4). Оператор initDist (строки 6-7) задает начальную инициализацию вершин. Шаблону в initDist удовлетворяет любая вершина графа. Для всех вершин кроме корневой значение атрибута dist устанавливается равным +INF (то есть +∞).

Оператор relaxEdge (строки 9-13) задает шаблон, структурная часть которого определена в виде ребра, а условная часть определяется сравнением весов путей до конечной вершины: если сумма весов начальной вершины и ребра меньше веса конечной вершины, то найден новый путь с меньшим весом (условие релаксации), и выполняется обновление атрибута dist в конечной вершине.

Далее в программе определяется, каким образом операторы initDist и relaxEdge будут выполнены. Для initDist используется foreach (строка 15), таким образом initDist будет выполнен однократно для всех сопоставленных подграфов, то есть для всех вершин графа. Для relaxEdge используется конструкция iterate (строка 16), следовательно, оператор relaxEdge будет выполняться для всех сопоставлений, пока такие существуют в графе.

```
1  Graph [nodes(node : Node, dist : int)
2         edges(src : Node,  dst : Node, wt : int) ]
3
4  source : Node
5
6  initDist = [ nodes (node a, dist d) ] =>
7             [ d = if (a == source) 0 else +INF ]
8
9  relaxEdge = [ nodes (node a, dist ad)
10               nodes (node b, dist bd)
11               edges (src a, dst b, wt w)
12               ad + w < bd ] =>
13               [ bd = ad + w ]
14
15 init = foreach initDist
16 sssp = iterate relaxEdge >> sched
17 main = init ; sssp
```

Рис. 2. SSSP на Elixir.

Порядок обработки сопоставленных подграфов определяет, какой из алгоритмов SSSP будет реализован:

- алгоритм Дейкстры:

sched = **metric** ad >> group b

metric ad указывает, что раньше будут обработаны те сопоставления, в которых значение ad (dist для начальной вершины) минимально, >> group b – ребра имеющие общую начальную вершину будут обработаны вместе (что позволит улучшить пространственно-временную локализацию при работе с графом);

- алгоритм Беллмана-Форда:

NUM_NODES : **unsigned int**

sssp = **for** i=1..(NUM_NODES-1) step; step = **foreach** relaxEdges

- алгоритм label-correcting:

sched = **group** b >> **unroll** 2 >> **approx metric** ad

- алгоритм в стиле дельта-степпинг (Δ-stepping):

DELTA : **unsigned int**

sched = **metric** (ad + w)/DELTA

В строке 17 определяется последовательность выполнения программы: сперва выполняется инициализация, далее вычисление расстояний до вершин.

Таким образом, Elixir представляет собой декларативный язык программирования (с элементами императивного программирования, например, циклы for, foreach), основанный на определении операторов над графом и спецификации порядка применения операторов.

Falcon

Еще один пример проблемно-ориентированного языка для обработки статических графов – Falcon [25]. Falcon разрабатывается в Inidian Institute of Science.

Falcon (также как Green-Marl и Elixir) относится к классу внешних DSL, но является расширением C (стандарт C99). В Falcon реализована поддержка как многоядерных систем с общей памятью, так и графических ускорителей. Кроме того, поддерживаются конфигурации из нескольких графических процессоров, а также гибридная схема вычислений, то есть с одновременным использованием и CPU, и GPU.

Falcon является языком со строгой типизацией. В дополнение к стандартным типам C в Falcon поддерживаются следующие типы: Graph – граф, Point – вершина, Edge – ребро, Set – множество вершин (определяется статически), Collection – множество вершин (определяется динамически).

Для определения параллельных участков кода, а также для синхронизации вычислений в Falcon используются специальные операторы:

single ([vertex | collection]) {stmt block1} **else** {stmt block2}

Данная конструкция гарантирует, что для вершины vertex (или коллекции вершин collection) блок инструкций block1 будет выполнены только одним потоком. В случае выполнения конструкции несколькими потоками, только один из них выполнит block1, а все остальные – block2.

foreach (item (advance_expr)

In object.iterator) (condition) { … }

Данная конструкция специфицирует, что блок инструкций в теле цикла может быть выполнен параллельно для заданного множества (значений итератора), где object имеет один из следующих типов: Graph (допустимые итераторы: points, edges, pptynames), Point (nbrs, innbrs, outnbrs), Edge (nbrs, nbr1, nbr2). Выражения advance_expr и condition опциональны и опредляют порядок выполнения итераций (вперед, назад) и условие выполнения итерации.

foreach (item (advance_expr)

In collection) (condition) { … }

Данная конструкция аналогична предыдущей, только итерации осуществляются по элементам коллекций.

parallel { … }

Данная конструкция аналогична определению параллельных секций в OpenMP. Важно отметить, что с ее помощью возможно организовать гибридную схему вычислений, в которой одна часть вычислений (секция) выполняется на ядрах CPU, а другая – на графических ускорителях.

Также в Falcon поддерживаются встроенные операции редукции: ReduxSum и ReduxMul. В будущем планируется добавить возможность задавать пользовательские собственные функции для выполнения редукции.

```
1  int changed = 0;
2
3  void reset (Point p, Graph graph) {
4    t.olddist = t.dist = MAX_VAL;
5    t.uptd = false;
6  }
7  void reset1 (Point p, Graph graph) {
8    t.olddist = t.dist;
9    if (t.dist < t.uptd)
10     t.uptd = true;
11 }
12 void relaxgraph (Point p, Graph graph) {
13   t.uptd = false;
14   foreach (t In p.outnbrs) {
15     MIN(t.dist, p.dist + graph.getWeight(p,t), changed);
16   }
17 }
18
19 void SSSP(Point p, Graph graph) {
20   // initialization
21   foreach(t In graph.points)
22     reset(t, graph);
23   p.dist = 0;
24   p.uptd = true;
25   // distance calculation
26   while (1) {
27     changed = 0;
28     foreach(t In graph.points)(t.uptd)
29       relaxgraph(t,graph);
30     if (changed)
31       break;
32     foreach(t In graph.points)
33       reset1(t, graph);
34   }
35   return;
36 }
```

Рис. 3. Реализация SSSP на Falcon для CPU-платформ.

Пример реализации алгоритма Беллмана-Форда решения задачи SSSP на Falcon для многоядерных систем с общей памятью приведен на рис. 3. Функция SSSP (строки 29-36) на входе имеет граф (graph), в котором будет осуществляться поиск и исходную вершину (p). Инициализация вершин осуществляется с помощью параллельного цикла foreach, в теле которого вызывается функция reset. В функции reset (строки 3-6) выставляются начальные значения атрибутов dist, olddist и uptd. Атрибутам dist и uptd корневой вершины присваиваются значения 0 и true (строки 23-24).

Вычисление расстояний от корневой вершин до других вершин графа выполняется в цикле (строки 26-24) до тех пор, пока найдется хотя бы одна вершина, в которой значение расстояния (т. е. атрибут dist) было обновлено. В соответствии с алгоритмом Беллмана-Форда количество итераций цикла ограничено сверху значением $|V|-1$. В теле цикла параллельно выполняется просмотр вершин, которые были модифицированы на предыдущей итерации (т. е. t.uptd равно true), и для них выполняется relaxgraph. В функции relaxgraph (строки 12-17) осуществляется параллельный анализ исходящих ребер и обновление значения атрибута dist соседних вершин, если *dist[v] + weight(v,u) < dist[u]* для ребра *(v,u)*.

Для разработки программ на графических ускорителях в Falcon достаточно добавить спецификатор <GPU> к типам Graph, Point, и тогда компилятор автоматически разместит граф в памяти графического ускорителя.

Заключение

В статье были рассмотрены четыре проблемно-ориентированных языка программирования, предназначенных для разработки и реализации алгоритмов анализа статических графов. Основные особенности рассмотренных DSL представлены в таблице 1.

Все рассмотренные языки являются академическими разработками, направленными на исследование подходов к созданию проблемно-ориентированных языков для параллельного анализа графов. Концептуально, Green-Marl, OptiGraph и Falcon представляют один подход к построению языка, основанный на параллельных конструкциях циклов foreach и системе специальных типов и итераторов, отражающих специфику графовых алгоритмов. OptiGraph отличается от Green-Marl и Falcon методом реализации, являясь встроенным DSL на базе функционального языка Scala. Кроме того, OptiGraph является представителей семейства генерируемых DSL для разных областей приложений с помощью инструментов Forge и Delite. В Elixir используется совершенно другой подход к построению параллельных программ, а именно декларативный с выделением операторов и спецификации их применения (политик планирования).

Таблица 1

Сравнение DSL для анализа статических графов

	Green-Marl	OptiGraph	Elixir	Falcon
Тип DSL	внешний	Встроенный (в Scala)	внешний	внешний (расширений C)
Тип языка	императивный, процедурный	императивный, процедурный	Декларативный с элементами импер.	императивный, процедурный
поддержка структурных трансформаций графа	-	-	-	+
Поддержка многоядерных систем с общей памятью	+ C++/ OpenMP	+	+ Galoios	+ C++/ OpenMP, Galoios
Поддержка графических ускорителей	-	+	-	+
Поддержка вычислительных кластеров (HPC)	-	-	-	-
Поддержка вычислительных кластеров (BigData)	+	-	-	-
Открытый исходный код	+	+	-	-

Единственным из рассмотренных языков, поддерживающим массово-параллельные вычислительные комплексы является Green-Marl. Однако эта поддержка реализована на базе программных платформ для работы с большими данными (Giraph и GPS используют платформу Hadoop для загрузки и выгрузки графов, разработаны на Java), что изначально является решением, ориентированным на отказоустойчивость и масштабирование приложений. Как показал анализ, поддержки высокопроизводительных кластеров (суперкомпьютеров) в проблемно-ориентированных языках программирования для обработки больших графов на настоящий момент нет. Принимая во внимание тенденцию на расширение спектра прикладных задач, выполняемых на суперкомпьютерах, а именно применение суперкомпьютеров для решения аналитических задач (понимание данных, машинное обучение, искусственный интеллект), становится очевидной потребность в реализации подобных языковых средств с нуля или на базе существующих с ориентацией на вычислительные комплексы класса HPC.

Список литературы:

1. Reed D. A., Dongarra J. Exascale computing and big data // Communications of the ACM. – 2015. – Т. 58. – №. 7. – С. 56-68.
2. Malewicz G. et al. Pregel: a system for large-scale graph processing // Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. – ACM, 2010. – С. 135-146.
3. Low Y. et al. Distributed GraphLab: a framework for machine learning and data mining in the cloud // Proceedings of the VLDB Endowment. – 2012. – Т. 5. – №. 8. – С. 716-727.
4. Gonzalez J. E. et al. Graphx: Graph processing in a distributed dataflow framework // 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). – 2014. – С. 599-613.
5. Shvachko K. et al. The hadoop distributed file system // 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). – IEEE, 2010. – С. 1-10.
6. Avery C. Giraph: Large-scale graph processing infrastructure on hadoop // Proceedings of the Hadoop Summit. Santa Clara. – 2011. – Т. 11.
7. Zaharia M. et al. Spark: cluster computing with working sets // HotCloud. – 2010. – Т. 10. – С. 10-10.
8. Quilitz B., Leser U. Querying distributed RDF data sources with SPARQL // European Semantic Web Conference. – Springer Berlin Heidelberg, 2008. – С. 524-538.
9. Webber J. A programmatic introduction to Neo4j // Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. – ACM, 2012. – С. 217-218.
10. Rodriguez M. A. The Gremlin graph traversal machine and language (invited talk) // Proceedings of the 15th Symposium on Database Programming Languages. – ACM, 2015. – С. 1-10.
11. Green-Marl Specification 0.7.1 https://docs.oracle.com/cd/E56133_01/1.2.0/ Green_Marl_Language_Specification.pdf
12. Hong S. et al. Green-Marl: a DSL for easy and efficient graph analysis // ACM SIGARCH Computer Architecture News. – ACM, 2012. – Т. 40. – №. 1. – С. 349-362.
13. Hong S. et al. Early experiences in using a domain-specific language for large-scale graph analysis // First International Workshop on Graph Data Management Experiences and Systems. – ACM, 2013. – С. 5.
14. Hong S. et al. Simplifying scalable graph processing with a domain-specific language // Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. – ACM, 2014. – С. 208.
15. Sevenich M. et al. Using domain-specific languages for analytic graph databases // Proceedings of the VLDB Endowment. – 2016. – Т. 9. – №. 13. – С. 1257-1268.
16. Malewicz G. et al. Pregel: a system for large-scale graph processing // Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. – ACM, 2010. – С. 135-146.
17. Salihoglu S., Widom J. GPS: a graph processing system // Proceedings of the 25th International Conference on Scientific and Statistical Database Management. – ACM, 2013. – С. 22.
18. Hong S. et al. PGX. D: a fast distributed graph processing engine // Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. – ACM, 2015. – С. 58.
19. Sujeeth A. K. et al. Composition and reuse with compiled domain-specific languages // European Conference on Object-Oriented Programming. – Springer Berlin Heidelberg, 2013. – С. 52-78.
20. Sujeeth A. K. et al. Delite: A compiler architecture for performance-oriented embedded domain-specific languages // ACM Transactions on Embedded Computing Systems (TECS). – 2014. – Т. 13. – №. 4s. – С. 134.
21. Sujeeth A. K. et al. Forge: generating a high performance DSL implementation from a declarative specification // Acm Sigplan Notices. – 2014. – Т. 49. – №. 3. – С. 145-154.
22. Prountzos D., Manevich R., Pingali K. Elixir: A system for synthesizing concurrent graph programs // ACM SIGPLAN Notices. – 2012. – Т. 47. – №. 10. – С. 375-394.
23. Lenharth A. Parallel Programming with the Galois System.
24. Nguyen D., Lenharth A., Pingali K. A lightweight infrastructure for graph analytics // Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. – ACM, 2013. – С. 456-471.
25. Nasre R. et al. Falcon: a graph manipulation language for heterogeneous systems // ACM Transactions on Architecture and Code Optimization (TACO). – 2016. – Т. 12. – №. 4. – С. 54.

ЗАКЛЮЧЕНИЕ НАУЧНО-ТЕХНИЧЕСКОГО СОВЕТА АО «НИЦЭВТ»

В статье представлен сравнительный анализ существующих проблемно-ориентированных языков (DSL) для обработки статических графов, показаны ключевые особенности языков на примере программы поиска кратчайших путей от заданной вершины (SSSP). Все рассмотренные языки ориентированы на использование высокопроизводительных вычислительных систем в качестве аппаратных платформ для выполнения программ на DSL-языках.

Вопрос создания новых программных средств для увеличения продуктивности параллельного программирования за счет повышения уровня абстракции и введения специализированных для предметной области элементов языка является крайне актуальным из-за высокой сложности разработки параллельных приложений средствами общего назначения, используемых в настоящее время.

Приведенный в статье сравнительный анализ DSL-языков позволяет дать объективную оценку состояния научных исследований в области создания проблемно-ориентированных языков для обработки графов и определить направления для дальнейших работ.

Статья рекомендуется для публикации в открытой печати.

Председатель НТС АО «НИЦЭВТ»

Симонов Алексей Сергеевич