Hoplang — язык обработки потоков данных мониторинга *

А.В. Адинец[1,2], С.А. Жуматий[1], Д.А. Никитенко[1]

[1]НИВЦ МГУ им. М.В.Ломоносова
[2]Объединённый институт ядерных исследований

При анализе работы больших вычислительных систем необходимо обрабатывать огромный объём данных мониторинга — загрузка процессоров, сетей, памяти, аппаратные счётчики, датчики температуры, и многое другое. Традиционные реляционные СУБД не в состоянии обеспечить обработку такого объёма информации. NoSQL-БД, системы MapReduce и языки запросов к ним масштабируются для обработки больших объёмов данных, но у них есть существенные недостатки. В частности, это непрозрачность взаимодействия с БД, отсутствие поддержки вложенных циклов и запросов, а также невозможность выполнения фильтрации по предикатам в самой БД. Hoplang разработан для анализа потоков данных мониторинга, и стремится избежать перечисленных выше недостатков. Его основные цели — абстракция языка от формы хранения данных, масштабируемость, поддержка распределённых запросов к БД и распределённой обработки данных, лёгкость расширения. Общая концепция языка близка к модели MapReduce, но обеспечивает большую гибкость и прозрачность в обработке потоков данных.

1. Введение

Современные суперкомпьютеры для большинства пользователей и даже администраторов представляют собой “чёрные ящики”. Насколько эффективно работают задачи на таких больших установках не знает практически никто. Для того, чтобы оценить производительность и эффективность только программы требуются специальные средства — профилировщики, трассировщики, средства анализа. Запуск с помощью этих средств часто помогает оптимизировать конкретную программу, но увы, их невозможно применять к каждой запущенной программе, т.к. они вносят очень существенные накладные расходы. А уж об эффективности использования установки в целом даже не идёт речи — таких средств практически нет.

С другой стороны, для общей оценки эффективности работы как отдельной задачи, так и вычислительной установки в целом, вполне достаточно собирать традиционные данные мониторинга, такие как загрузка процессора, интенсивность использования дисков, подкачки, сети, оперативной памяти и т.п. Анализ даже таких данных способен дать качественно новую картину использования суперкомпьютера и даже картину работы отдельной задачи.

Решение разбивается на две проблемы — сбор данных мониторинга и их анализ. Для больших установок объём данных мониторинга становится очень большим и эти проблемы могут помешать нормально работе самих вычислителей. Для сбора данных существует достаточно много средств, многие из них поставляются в открытых исходных кодах и являются расширяемыми. В качестве примеров можно привести Ganglia, Zabbix, Nagios, Clustrx, а также многие другие. В данной работе мы не будем на них останавливаться, заметим лишь, что в наших условиях эффективным выходом может быть распределение данных по физически разным базам данных.

И хотя средств мониторинга существует много, средств для решения второй проблемы — обработка и анализ полученных данных — существует не так уж много. Как уже было упомянуто, данные могут быть распределены на несколько разных физических баз, сам

*Работа выполнена в рамках гос. контракта 07.514.12.4001, совместного проекта Евросоюза и России с кодовым названием HOPSA

состав баз данных тоже может быть разным. Например, данные по задачам хранятся в MySQL, данные по пользователям — в LDAP, а данные мониторинга загрузки — в нереляционной БД, например Cassandra. Схемы хранения данных также могут быть различны и в разных условиях оптимальными могут оказаться разные схемы.

Надо учесть, что и объём данных мониторинга тысяч и десятков тысяч (а близкой перспективе и миллионов) вычислительных узлов — это десятки и сотни гигабайт информации в сутки, по самым оптимистическим прогнозам. С такими потоками данных способны справиться далеко не все реляционные СУБД. Многие NoSQL-базы это могут, но сильно пригрывают в гибкости хранения, да и не всегда все данные можно хранить в одной базе, даже распределённой. Например, задача записи в базу сразу на нескольких серверах, т. к. такой объём просто невозможно собрать и передать через один сервер, разрешима далеко не во всех базах.

Таким образом "традиционное" решение — одна БД и набор SQL- или NoSQL-запросов в данном случае плохо применим.

Эта статья построена следующим образом. В разделе 2 мы описываем нашу текущую инфраструктуру, её недостатки и предъявляемые к создаваемому языку требования. Далее, в разделе 3 описываются основные конструкции языка, а в разделе 4 — возможности его параллельной реализации. Наконец, в разделе 6 приведены итоги и планы дальнейшей работы.

2. Существующие решения и необходимость Hoplang

Для решения задачи обработки больших объёмов данных мониторинга суперкомпьютерных комплексов нами первоначально был использован подход с использованием одной из наиболее распространённых технологий массовой обработки данных — технологии Hadoop [1], свободной реализации концепции MapReduce.

Нами была выбрана высокопроизводительная база данных Cassandra [2], а для формулирования запросов обработки данных — язык PIG [3]. Такая комбинация позволила обрабатывать реальные данные с таких суперкомпьютеров как "Чебышёв" и "Ломоносов". Однако, были выявлены существенные недостатки такого подхода: недостаточная производительность и низкая переносимость.

Недостаточная производительность обусловлена вовсе не низкой скоростью работы какого-то компонента построенного комплекса, а высокими накладными расходами. Процесс компиляции PIG-запроса в реальную java-программу, загрузка и запуск полученной java-программы занимают зачастую больше времени, чем собственно вычисления.

Низкая переносимость связана с особенностями языка PIG. Данный язык требует явного указания источника данных, поддерживает очень ограниченный набор баз данных и ограничен в возможностях. Несмотря на то, что все компоненты построенного комплекса являются opensource и допускают модификацию кода, необходимые изменения потребовали бы кардинальных изменений исходных программных продуктов.

Для решения этой проблемы нами был предложен язык обработки данных Hoplang, ориентированный именно на обработку данных мониторинга. Под словами "язык обработки данных" мы понимаем не только описание синтаксиса, но и концепции обработки данных. Несмотря на то, что язык ориентирован на конкретную задачу, он покрывает достаточно широкий класс задач и может быть использован для многих видов массовых вычислений.

Основные концепции языка: данные представляются в виде именованных или промежуточных потоков, программа на Hoplang задаёт только схему (алгоритм) обработки данных. Это значит, что в программе на Hoplang нельзя задать адрес базы данных или имя таблицы, столбца. Также в программе нельзя указать какие части должны быть распределены по разным вычислительным узлам, какие выполнены совместно, как именно должен быть выполнен запрос к БД, и т.п.

Всё, что касается источников данных, возлагается на “драйвер” базы данных. Это компонент языка, который может быть подключен к интерпретатору, одновременно может быть подключено множество драйверов. Каждый драйвер позволяет осуществлять доступ к базе данных конкретного типа — Cassandra, MySQL, MongoDB [4], CSV-файл и т.п. Драйвер должен осуществлять перевод обращения из программы на Hoplang к источнику данных в запрос на языке базы данных, по возможности, осуществляя его оптимизацию. Кроме того, драйвер должен выполнять преобразования представления данных, хранящихся в БД, ко внутреннему представлению Hoplang.

Например, драйвер может переместить условия выборки из программы на Hoplang в запрос SQL, или “склеить” два вложенных обращения в один сложный запрос к БД. Отображение имён потоков данных на конкретные базы данных и их внутренние схемы задаётся администратором в конфигурационном файле и при работе программы компилятор Hoplang определяет какой драйвер должен обработать какое обращение к потоку данных.

Интерфейс для написания собственных драйверов БД открыт, поэтому добавить поддержку нового типа базы данных не составляет большой сложности. Новый драйвер достаточно расположить в специальном, каталоге и он будет автоматически использован компилятором. На данный момент драйвер может быть написан только на языке ruby [5].

Сам язык построен так, чтобы основные секции обработки данных были независимы или могли обрабатывать данные в режиме конвейера — принимая и отдавая потоки данных. Язык поддерживает агрегатные функции, выполнение которых оптимизируется автоматически. При возможности выполнить их средствами базы данных соответствующий драйвер попытается это сделать. Язык предусматривает и написание собственных агрегаторов, но в этом случае поддержка со стороны базы данных будет ограниченной.

3. Основные конструкции языка

Hoplang является языком со слабой динамической типизацией. На этапе выполнения имеется строгое различие лишь между простыми значениями, кортежами и потоками. Элементами одного потока могут быть либо кортежи, либо простые значения. Из простых типов в настоящее время поддерживается строковый и целочисленный типы, вещественный тип может быть добавлен в будущем, если в нём возникнет необходимость. Типы в программе явно не объявляются. Типы значений, читаемых из БД, определяются динамически на основании схемы БД. В качестве оптимизации в будущем планируется использовать вывод типов при компиляции запроса.

Структура программы Hoplang представляет собой последовательность операторов, как в обычных процедурных языках. Не допускается указание двух операторов в строке, конец оператора определяется концом строки или комментарием. Комментарии в Hoplang начинаются с символа `#` и оканчиваются концом строки. Многострочных комментариев в Hoplang нет.

Переменные Hoplang делятся на *потоковые* и *скалярные*. Потоковые переменные представляют собой именованный поток передачи данных. Запись в такую переменную по сути эквивалентна передаче данных в поток, а чтение — чтению очередного элемента данных из потока. Чтения и запись в потоковые переменные осуществляется только специальными операторами — хопстансами.

Скалярные переменные — или скалярные значения, или кортежи с именованными полями. В такой переменной может храниться, например, загрузка процессора вычислительного узла в виде пользовательского времени, системного времени, времени простоя, времени ввода-выводи и отметки времени. К полям переменной, которая является кортежем, можно обращаться через точку по именам, например `myvar.f1`.

Скалярные переменные могут быть объявлены в любом месте программы оператором `var`, за которым следует имя переменной. Как в большинстве современных языков, пере-

менная имеет область видимости — тот блок, в котором она объявлена.

Hoplang поддерживает основные управляющие конструкции: `if ... else ... end` (рис. 1) и `while ... end` (рис. 2). Конструкции `else` и `elif` опциональны, `elif` может повторяться любое число раз.

```
if cond
...
elif cond
...
else
...
end
```

Рис. 1. Вид оператора `if`

```
while cond
...
end
```

Рис. 2. Вид оператора `while`

Операции над переменными, допускаемые в Hoplang: числовые, строковые, сравнения и логические. Числовые: `+, -, *, /`. Поддерживаются круглые скобки для изменения приоритетов операций. Из строковых операций поддерживается конкатенация — оператор `&`. Операции сравнения: `==, !=, <, >, <=, >=`, возвращают результат логического типа. Логические операции `and, or, xor` поддерживаются для значений логического типа. Для переменных, которые являются кортежами, операции могут выполняться только над их полями, например, как на рис. 3. Присваивания могут осуществляться как для поля переменной, так и для переменной целиком. Во втором случае справа от оператора присваивания должны стоять переменная, либо набор выражений, предварённых именами — рис. 4.

```
a.val = x.counter + y.cpus
```

Рис. 3. Доступ к полям переменных

```
v = name1 => expr1, ... , nameN => exprN
```

Рис. 4. Присваивание полям переменной

Хопстансы — конструкции Hoplang, позволяющие работать с потоками данных. Хопстансы не могут выполняться внутри блоков `if` или `while`. Допускаются вложенные хопстансы. Основной хопстанс Hoplang — хопстанс `each`. Хопстанс читает заданный поток данных, выбирая нужные данные по условию, и выполняет обработку. При окончании потока выполняется специальная “финальная” секция хопстанса.

Общий вид хопстанса `each` представлен на Рис 5. Здесь `stream1` — имя входного потока (потоковая переменная). Это имя должно быть описано в файле конфигурации и может указывать как на “живой поток” данных от системы мониторинга, так и на базу данных. `stream2` — имя выходного потока. Его может использовать любой следующий по тексту программы хопстанс. `v` — переменная хопстанса, в ней будет значение очередной порции данных, полученных из входного потока. Эта переменная объявляется автоматически и

доступна только в теле хопстанса и секции `final`. `cond` — условие выборки из потока. Конструкции `where` и `final` могут отсутствовать.

```
stream2 = each v in stream1 where cond
...
final
...
end
```

Рис. 5. Вид оператора `each`

Важный оператор, который работает внутри хопстанса — оператор `yield`. Этот оператор записывает свои аргументы в выходной поток хопстанса. Общий вид оператора `yield` представлен на рис. 6. Второй вариант возвращает в поток все текущие поля переменной `myvar` под их именами. В случае, если выражение expr является полем переменной, то часть `name =>` может быть опущена и имя поля в выходном потоке будет совпадать с именем поля переменной. Если в качестве аргумента выступает агрегатор, то имя также можно опустить и по умолчанию оно будет совпадать с именем агрегатора.

```
yield name1 => expr1, ..., nameN => exprN
yield myvar.cpu_load
yield myvar
```

Рис. 6. Виды оператора `yield`

Простой пример подсчёта суммы и количества чётных элементов в поле 'v' в потоке на рис. 7. Тело хопстанса пустое, так как все операции тут выполняются агрегаторами `sum` и `count` в секции `final`. Встроенные агрегаторы, допустимые в секции final: `min, max, count, sum, avg, top, bottom`.

```
out = each v in mystream where v.f % 2 == 0
final
  yield sum(v.f), count(v)
end
```

Рис. 7. Пример подсчёта суммы

Ещё один пример, когда поток данных преобразуется в новый поток представлен на Рис. 8. Здесь мы суммируем времена, затраченные на систему, ввод-вывод и прочее в поле потока под именем `other`, а время пользователя и время простоя оставляем как есть. В выходной поток попадают только те элементы потока, где время пользователя больше 10%.

```
out = each v in cpudata where v.user > 10
  yield other => v.sys + v.wio + v.nice,
    idle => v.idle, user => v.user
end
```

Рис. 8. Пример обработки потока

Другой хопстанс — `seq`. Он предназначен для обработки данных из промежуточного потока (обычно полученного от другого хопстанса). В этом хопстансе нельзя обращаться к внешним источникам данных. Вид хопстанса представлен на на рис. 9. Секция `final` опциональна.

```
out = seq v in stream1
...
final
...
end
```

Рис. 9. Вид хопстанса seq

Следующий хопстанс — `group`. Он аналогичен конструкции group by в SQL, то есть производит группировку данных по заданному выражению. В отличие от each, секция `final` вызывается для каждой группы. Общий вид хопстанса представлен на рис. 10.

```
stream1 = group v in stream2 by cond
...
final
...
end
```

Рис. 10. Вид хопстанса `group`

Следующий хопстанс — `sort`. Этот хопстанс производит сортировку потока. В отличии от рассмотренных ранее, он не имеет ни тела, ни секции `final`. Общий вид хопстанса представлен на рис. 11.

```
stream1 = sort v in stream2 by cond
```

Рис. 11. Вид хопстанса `sort`

Ещё один хопстанс — `scan`. Он производит одну операцию над всеми элементами потока. Общий вид представлен на рис. 12. Этот хопстанс, как и `sort`, не имеет тела и секции `final`.

```
stream2 = scan stream1 postfix by OPERATION on field
stream2 = scan stream1 prefix by OPERATION on field
```

Рис. 12. Вид оператора `scan`

Хопстансы `top` и `bottom` — эквивалентны сортировке потока и выборке заданного числа наибольших или наименьших элементов по заданному критерию. Общий вид этих хопстансов представлен на рис. 13. Хопстансы возвращают первые или последние N элементов потока, отсортированных по выражению `expr`.

```
stream2 = top N stream1 by expr
stream2 = bottom N stream1 by expr
```

Рис. 13. Вид операторов `top` и `bottom`

Допускается чтение ровно одного элемента потока. Это должен быть поток, описанный в конфигурации, а не порождённый в программе. Иными словами, допускается чтение одной строки из базы данных. Общий вид такого оператора представлен на рис. 14. Если под условие попадает несколько элементов, будет возвращён только один, какой именно — не специфицируется.

```
myvar = get dbvarname field1, ... fieldN where cond
```

Рис. 14. Вид оператора `get`

4. Возможности параллельной обработки

Выше были описаны синтаксические конструкции языка. Теперь рассмотрим как осуществляется параллельная обработка данных в рамках Hoplang. Первый параллельный принцип, применяющийся в Hoplang — *конвейерность*. Любые два хопстанса могут передавать данные в режиме конвейера через поточную переменную. В простейшей реализации разные хопстансы работают в разных нитях на одном компьютере, в общем случае — в разных процессах на разных компьютерах.

Следующий принцип реализации параллелилизма — *распараллеливание доступа к БД*. Один и тот же источник может храниться в нескольких базах данных. Например, данные мониторинга от десяти тысяч узлов эффективнее распределить по нескольким физическим БД. Или данные о пользователях двух кластеров могут храниться в двух БД.В случае, если данные распределены по разным БД, хопстанс автоматически разделяется на несколько соответствующих хопстансов. По окончании работы всех хопстансов, их результаты при необходимости объединяются.

Распределение вычислений может быть произведено компилятором Hoplang и в случае, если предполагаемый объём вычислений от одной БД слишком велик. Это эвристическая оценка и её работа зависит от реализации компилятора Hoplang. В этом случае, запрос к БД разбивается на несколько, и на каждый полученный запрос создаётся свой экземпляр хопстанса.

Для обеспечения корректности работы и снижения накладных расходов при параллельной работе, в Hoplang введены ограничения на доступ к переменным. Любая переменная доступна на чтение только в своей области видимости, а на запись — только на уровне своего хопстанса. Это значит, что во вложенном хопстансе доступ на запись переменных из объемлющего хопстанса невозможен.

Если в теле хопстанса объявлена переменная, и она используется в секции `final`, то этот хопстанс не может быть распределён на несколько, так как нет возможности обеспечить корректность его работы. Данная проблема будет решена в следующей версии языка, в которой будет добавлена возможность задать пользовательский код для агрегации данных из распараллеленных хопстансов.

5. Текущее состояние

В настоящее время реализована первая версия интерпретатора Hoplang, для реализации использовался язык Ruby. Поддерживаются полноценный синтаксис выражений, драйверы для чтения данных из формата CSV и из БД Cassandra, а также хопстанс `each` и его последовательная версия `seq`. Для драйвера БД Cassandra поддерживается проталкивание предикатов фильтрации в `where` в запрос к БД. Для этого требуется, чтобы предикат был конъюнкцией простых условий, т.е. условий вида <поле> <операция сравнения> <константа>. При этом операция сравнения не может быть `!=`, должно быть хотя бы одно сравнение на равенство, и для всех фильтруемых полей в БД Cassandra должны быть созданы индексы. Нам кажется, что это будет наиболее часто используемый случай проталкивания индексов в запрос к БД.

Для демонстрации возможностей текущей реализации мы провели сравнение между Hoplang и нынешней версии системы запросов к данным мониторинга кластера, использующей связку Pig + Hadoop. Запрос: по пользователю требуется получить среднее число ЦПУ по его задачам, и число самих задач. Код запроса на языке Hoplang представлен на

рис. 15, а код на языке Pig приведён на рис. 16. Сразу бросается в глаза разница в читаемости кода. На Pig более запроса занимает код для чтения данных из БД Cassandra. Реализация БД Cassandra в Pig такова, что данные читаются в нетипизированном виде, и для преобразования его в типизированный вид необходимо выполнить дополнительные манипуляции. Кроме того, при чтении из другой БД в Pig придётся изменить код запроса, в то время как в Hoplang достаточно поменять файл конфигурации. Более того, после реализации в Hoplang полноценного синтаксиса конвейеров в стиле оболочек типа sh и полноценных агрегаторов запрос удастся сократить до одной строчки.

```
ts = each t in tasks where t.user == 'serdyuk'
 yield key => t.key, ncpus => t.ncpus
end
avgs = seq t in out3
 var ncpus, n
 ncpus = ncpus + t.ncpus
 n = n + 1
final
 yield ncpus, n, ncpus / n
end
```

Рис. 15. Код запроса на языке Hoplang для получения среднего числа процессоров за всё время работы кластера по задачам пользователя 'serdyuk'

```
REGISTER hopsa-udfs.jar;

%default cf 'tasks_gra'
%default user 'serdyuk'
%default resdir '/does/not/exist'

tsr = LOAD 'cassandra://hopsa/$cf' USING CassandraStorage() AS (
  k:bytearray, va:{t:(n:bytearray, v:bytearray)}); /* $ */
ts1 = FOREACH tsr {
  u = FILTER va BY n=='user';
  nc = FILTER va BY n=='ncpus';
  GENERATE k AS key, FLATTEN(u.v) AS user, FLATTEN(nc.v) AS ncpus;
};
ts = FOREACH ts1 GENERATE key, user, (long)ncpus AS ncpus;
tsf = FILTER ts BY user == '$user'; /* $ */
gu = GROUP tsf BY user;
timeu = FOREACH gu GENERATE group AS user, AVG(tsf.ncpus) AS
  avgncpus, COUNT(tsf) AS ntasks;
STORE timeu INTO '$resdir' USING PigStorage('\t'); /* $ */
```

Рис. 16. Код запроса на языке Pig для получения среднего числа процессоров за всё время работы кластера по задачам пользователя 'serdyuk'

С использованием Hoplang запрос исполнялся 2.03 секунды, а с использованием Pig — 12.35 сек. Экономия достигается прежде всего за счёт проталкивания запроса в БД. Hoplang-версия использует индексы в БД Cassandra, и поэтому обрабатывает намного меньше данных, в то время как Pig-версия вынуждена сканировать всю БД, после чего извлекать из неё данные по конкретному пользователю. Разумеется, по мере роста объёма собранных

данных, Hoplang-версия будет масштабироваться значительно лучше.

6. Дальнейшее развитие

Hoplang уже реализует множество возможностей для обработки больших объёмов данных. Однако, в ближайших планах у нас стоит его усовершенствование и расширение. Среди основных целей стоит отметить:

- реализация функций, как скалярных, обрабатывающих только одно скалярное значение, так и функций-хопстансов, которые могут выступать самостоятельной ступенью конвейера
- добавление использования пользовательского кода для объединения результатов параллельных хопстансов (редукторы),
- объединение последовательных хопстансов в цепочку, с удалением имён промежуточных “связующих” поточных переменных

Литература

1. Tom White Hadoop: The Definitive Guide O’Reilly, 2009. 501 p.
2. Apache Cassandra DB official cite URL: http://cassandra.apache.org/ (дата обращения: 01.11.2011)
3. Apache PIG Latin official cite URL: http://pig.apache.org/ (дата обращения: 01.11.2011)
4. MondoDB official cite URL: http://www.mongodb.org/ (дата обращения: 01.11.2011)
5. Д. Флэнаган, Ю. Мацумото Язык программирования Ruby Издательство Питер, 2011 496 стр.