Московский государственный университет имени
М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Описание языка программирования Frigate

Студент 415 группы
А. Н. Ашабоков

Руководитель практикума
к.ф.-м.н., А.Н. Сальников

Москва, 2019

Содержание

1 Общие сведения

Система Frigate позволяет программисту представить свой алгоритм в виде графа состоящего из множества непересекающихся по вершинам подграфов. Каждый подграф является графом в смысле системы PARUS. Подграфы поочередно назначаются на исполнение. Этим управляет процесс координатор, адрес которого знают все вершины. Координатор содержит условные выражения, соответствующие каждому подграфу. На исполнение назначается тот подграф, чье выражение истинно. Данный механизм позволяет динамически перестраивать граф-программу во время выполнения, назначая на выполнение один из заранее подготовленных подграфов в каждый момент времени. Ребра в графе бывают трех типов: внутренние (на уровне одного подграфа), внешние (на уровне графа соединяют подграфы), управляющие (на уровне графа направлены к процессу координатору). В целях уменьшения дублирования кода при создании большого количества подобных вершин или ребер был введен механизм шаблонов. Суть его в том, что программистом описывается шаблон вершины или ребра, содержащий все внутреннее устройство элемента, а при создании однотипных вершин или ребер, они используют описанный шаблон. Вершины в графе Frigate могут иметь несколько точек входа и выхода ребер, чередующиеся с блоками обработки кода, таким образом достигается более сложная логика в работе кода вершины. Программист описывает программу в синтаксисе языка Frigate, выделяя в программе последовательные участки обработки, соответствующие вершинам графа, и зависимости по данным - ребра. По данному представлению генерируется набор объектов в памяти, соответствующей иерархии классов. Затем из объектов-контейнеров генерируется набор функций с MPI- вставками, который компилируется на распределенной системе и связывается с подсистемой запуска.

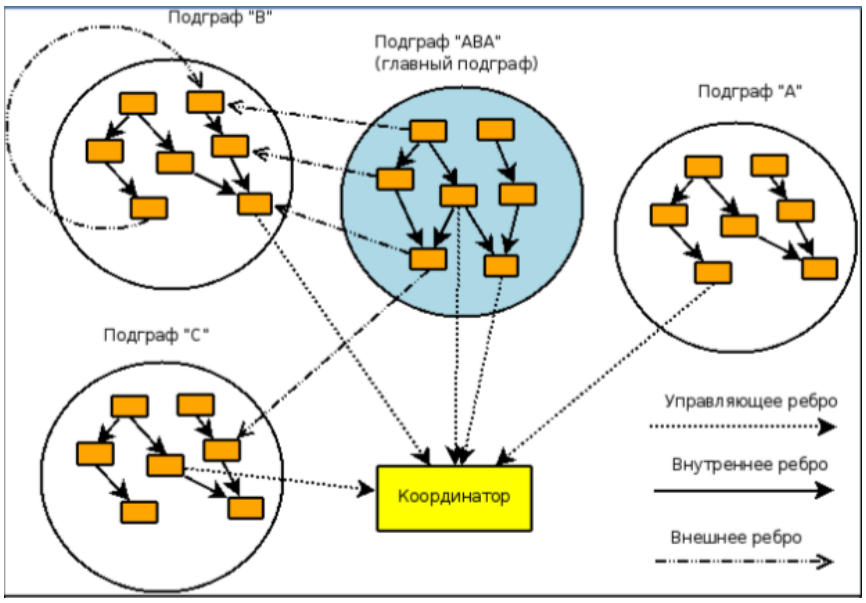

Рис. 1: Общая схема графа программы на языке Frigate.

2 Структура языка

Данный раздел содержит описание структуры языка Frigate. Для наглядности раздел 3 содержит пример кода.
Перед началом разбора структуры языка введем следующие договоренности:

- Поле name внутри любого из объектов содержит имя данного объекта
- Блок вида @...@ содержит некоторый исполняемый код на языке C++. Пример:

```
@
int a = 1;
int b = 2;
int c = a + b;
@
```

Программа на языке Frigate описывается в виде графа, внутри блока <graph> содержится описание всех подграфов <subgraph>, шаблонов вершин <vertex_template>, шаблонов ребер <edge_template> и внешних ребер <external_edge>, соединяющих вершины различных подграфов, а также содержит блоки <header>, <root>, <tail> и <parameters>.

- Блок <header> содержит в себе код на языке C++, который при компиляции MPI программы будет прямиком скопирован в его начало.
- Блок <root> содержит код C++, который выполняется MPI программой перед началом каких-либо действий. Его имеет смысл использовать для проведения подготовительных работ перед началом работы основного вычислительного алгоритма, например для инициализации каких-либо глобальных переменных и выделения памяти.
- Блок <tail> содержит код C++, который выполняется каждой вершиной по завершению основных вычислений. Данный блок можно использовать, например, для очищения выделенной памяти.
- Блок <parameters> содержит перечисление переменных-параметров и списки значений параметров для каждого из них. Параметры в языке нужны для имитиции работы цикла, а именно, для создания большого числа однотипных вершин, ребер и других сущностей, отличающихся только значением параметра. Например, при выполнении следующего кода будут созданы внутренние ребра с именами int_1 и int_2, отличающиеся только значением в поле recv: "sub_1_vert_2" и "sub_1_vert_3" соответственно.

```
...
<parameters>
var i = ("1" "2")
```

```
var j = ("2" "3")
</parameters>
...
<internal_edge>
using i, j;
name = "int_%i%"
template = "edge_template"
send= ("sub_1_vert_1", "ex_1")
recv = ("sub_1_vert_%j%", "ex_1")
</internal_edge>
```

Поле **using** содержит перечисление параметров, использующихся в данной сущности. Это поле является необязательным, т.е. если в описываемом объекте нет необходимости использовать параметры, поле **using** можно не писать. Далее при описании полей ребер, вершин и других объектов, описание поля **using** будем опускать.

- Поле main_subgraph содержит имя подграфа, который далее будет считаться главным.

Рассмотрим теперь структуру подграфа <subgraph>. Подграф содержит описание все своих вершин <vertex>, внутренних ребер <internal_edge> и управляющих ребер <control_edge>. Сразу отметим, что эти блоки могут идти в произвольном порядке.

- Поле condition содержит условие выполнения данного подграфа на языке C++.
- Блок <vertex> содержит описание вершины данного подграфа. Вершины могут быть описаны либо явно внутри подграфа, либо при помощи шаблона вершины (см. далее), путем задания в поле **template** имя используемого шаблона. Очевидно, что использование шаблонов вершины имеет смысл в том случае, когда требуется создать большое количество однотипных вершин, реализующих один и тот же алгоритм. Явное описание вершины совпадает с описанием шаблона вершины и приведено далее по тексту.
- Блок <internal_edge> содержит описание внутренних ребер подграфа. Очевидно, эти ребра могут быть использованы только для связи вершин внутри одного подграфа, для коммуникации с вершинами других подграфов необходимо использовать внешние ребра <external_edge>. Внутреннее ребро содержит следующие поля:
  - Поле **template** содержит название шаблона ребра, используемого данным ребром.
  - Поле send содержит адрес отправителя в формате: (”название вершины отправителя”, ”название exchange блока отправителя”).

  - Поле recv аналогично содержит адрес отправителя: (”название вершины получателя”, ”название exchange блока получателя”).

- Блок <control_edge> содержит описание ребра, соединяющего вершину с координатором. Очевидно, что в <control_edge> нет необходимости описания адреса получателя. т.к. это всегда процесс-координатор. Структура ребра:
  - **template** — название используемого шаблона вершины.
  - send — адрес отправителя, в том же формате, что и для <internal_edge>.

Рассмотрим структуру блока <external_edge>. Данный блок содержит описание внешней вершины, соединяющей вершины различных подграфов. Фактически структура внешнего ребра <external_edge> отличается от структуры внутреннего ребра <internal_edge> только набором параметром, задающих адрес отправителя и получателя. Это связано с тем, что при передаче данныз внутри определенного прдграфа, знание имени подграфа не является необходимым, что, очевидно, неверно для внешних ребер,

- Поле **template** содержит название используемого шаблона ребра.
- Поле send содержит адрес отправителя в формате: (”название подграфа отправителя”, ”название вершины отправителя”, ”название exchange блока отправителя”).
- Поле recv аналогично содержит адрес отправителя: (”название подграфа получателя”, ”название вершины получателя”, ”название exchange блока получателя”).

Рассмотрим структуру блока <edge_template>. Этот блок содержит описание блоков памяти, содержащих данные для отправки по указанному ребру, и блоки памяти, в которые получаемые данные должны быть сохранены.

- Блок <send> содержит в себе перечисление некоторого числа блоков <fragment> — фрагментов памяти, подлежащих отправке по данному ребру. Блок <fragment> имеет структуру:
  - Поле variable — название переменной.
  - Поле type — тип передаваемых данных.
  - Поле left — указатель на левую границу массива с данными в памяти.
  - Поле right — указатель на правую границу массива с данными в памяти.
- Блок recv содержит перечисление блоков <fragment> — фрагментов памяти, в которые произойдет сохранение полученных по данному ребру данных. Структура блока <fragment> описана выше.

Рассмотрим структуру блока <vertex_template>. Структура данного блока совпадает со структурой блока <vertex> в том случае, когда вершина задается не через шаблон вершины, а явно внутри блока <vertex>. Блок <vertex_template> состоит из произвольного количества блоков <code> и <exchange> в произвольном порядке.

- Блок <code> содержит исполняемый код на языке C++. Его можно задать вписав код в специальный блок вида @...@, либо подключив файл с кодом на C++, указав название файла в поле file .
- Блок <exchange> отвечает за обмен данными между вершинами. Управление обменом осуществляется при помощи полей send и recv:
  - Поле send содержит название ребра, по которому будет осуществляться отправка данных.
  - Поле recv содержит название ребра, по которому будет осуществляться получение данных.

  Дополнительно отметим, что блок <exchange> может состоять из произвольного количества полей send и recv, перечисленных в произвольном порядке. Это значит, что один <exchange> блок может осуществлять произвольное число операций передачи и получения данных в произвольном порядке. Также стоит отметить, что возможно чередование блоков <code> и <exchange>. Это значит, что вершина по мере выполнения может отправлять промежуточные данные и получать их от других вершин.

3 Пример кода

```
# test_3
# Answer: OK

<graph>

version = 1.0

<header>
@
some code here
@
</header>

<root>
root = "root"
</root>

<tail>
@
some code here
```

```
@
</tail>

<parameters>
var i = ("1" "2")
var j = ("2" "3")
</parameters>

main_subgraph = "sub_1"

<subgraph>
name = "sub_1"

condition = @ some code here @

<vertex>
name = "sub_1_vert_1"

<code>
@
some code here
@
</code>

<exchange>
name = "ex_1"
send = "int_1"
send = "int_2"
</exchange>
</vertex>

<vertex>
name = "sub_1_vert_2"
<code>
@
smdks
@
</code>

<exchange>
name = "ex_1"
recv = "int_1"
send = "ext_1"
```

```
</exchange>
</vertex>

<vertex>
name = "sub_1_vert_3"

<code>
@
dwd
@
</code>

<exchange>
name = "ex_1"
recv = "int_2"
send = "cont_1"
</exchange>
</vertex>

<control_edge>
name = "cont_1"
template = "edge_template_1"
send = ("sub_1_vert_3", "ex_1")
</control_edge>

<internal_edge>
using i, j;
name = "int_%i%"
template = "edge_template"
send= ("sub_1_vert_1", "ex_1")
recv = ("sub_1_vert_%j%", "ex_1")
</internal_edge>

</subgraph>

<subgraph>
name = "sub_2"
condition = @ some code here @

<vertex>
name = "sub_2_vert_1"

<code>
```

```
@
kd
@
</code>

<exchange>
name = "ex_1"
send = "int_1"
send = "int_2"
</exchange>

</vertex>

<vertex>
name = "sub_2_vert_2"

<code>
@
sfe
@
</code>

<exchange>
name = "ex_1"
recv = "int_1"
recv = "ext_1"
</exchange>
</vertex>

<vertex>
name = "sub_2_vert_3"

<code>
@
jnfkn
@
</code>

<exchange>
name = "ex_1"
recv = "int_2"
</exchange>
</vertex>
```

```
<internal_edge>
using i, j;
name = "int_%i%"
template = "edge_template"
send= ("sub_2_vert_1", "ex_1")
recv = ("sub_2_vert_%j%", "ex_1")
</internal_edge>


</subgraph>

<edge_template>
name="edge_template"
<send>
<fragment>
variable = @ some code here @
type     = @ some code here @
left     = @ some code here @
right    = @ some code here @
</fragment>
</send>

<recv>
<fragment>
variable = @ some code here @
type     = @ some code here @
left     = @ some code here @
right    = @ some code here @
</fragment>
</recv>
</edge_template>

<external_edge>
name = "ext_1"
template = "edge_template"
send = ("sub_1", "sub_1_vert_2", "ex_1")
recv = ("sub_2", "sub_2_vert_2", "ex_1")
</external_edge>
</graph>
```

4 Структура проекта

Проект имеет следующую структуру:

- Parser/
  - Tag.h содержит классы, соответствующте всем лексемам грамматики, напрмер: StringTag — обозначает набор символов вида: "some_string".
  - LexTag.h содержит производные тэги, полученные из базовых лексем, напрмер: main_subgraph — производный тэг, состоящий из лексем MainSubgraphTag, AssignTag и StringTag, ему соответствует фрагмент программы: main_subgraph = "name_of_main_subgraph".
  - Lexer.h содержит класс Lexer с полями text и tagMap, и функцией go. Конструктор класс получает на вход строку, содержащую весь код программы на языке frigate и записывает его в поле text. Далее функция go производит разбор текста на лексемы, описанные в Tag.h.
  - Parser.h содержит класс Parser, который производит рекурсивную свертку из списка лексем типа Tag к единственному тэгу graph. В случае неудачи, вызывается ошибка. Описание правил свертки для каждого тэга описано в соответствующем классе в файле LexTag.h.
  - HexParser.h содержит функцию parse — производит лексический и синтаксический разбор исходного текста, а также проверяет результат на правильность и вызывает исключения, если нужно.
- Exteptions/
  - Exception.h содержит класс исключений Exception — крайне простой класс с двумя полями: название исключения (для классификации ошибок) и сообщение.
- Compiler/
  - ControlEdge.h содержит класс, обозначающий управляющее ребро. Имеет следующие поля:
    - name — название ребра (везде далее при описании других классов будем опускать описание этого поля, т.к. из названия поля очевидно его назначение).
    - send_coord — адрес отправителя.
    - sends — вектор фрагментов для отправки.
    - recvs — вектор фрагментов для чтения.

    Список методов:
    - addSendFragment и addRecvFragment — сеттеры для полей sends и recvs.

    * readTag — метод заполняет поля класс, извлекая информацию из тэгов (этот метод тоже часто встречается в других классах и используется также, так что ниже будем опускать его описание).
    * emplaceParams — в язык была добавлена возможность создания объектов (в том числе и ребер) с использованием параметров в названиях (см. выше описание возможностей языка). Эта функция занимается работой с этими параметрами.
- Coords.h содержит классы (названия классов соответствуют грамматике языка, а назначение — описанию языка выше):
    * Coord — класс, описывающий адрес для работы с ребрами.
    * CodeBlock — класс, содержащий описание кода на языке Си.
    * Fragment — класс, с данными, описывающими фрагмент памяти, с которым будет работать ребро при отправке и получении сообщений.
- Exchange.h содержит единственный класс Exchange, который отвечает за описание блока обмена данными в теле вершины. Названия полей совпадают с названиями соответствующих элементов языка frigate , смысл тоже.
- ExternalEdge.h содержит одноименный класс, описывающий ребра, соединяющие вершины в разных подграфах. Поля и методы аналогично ControlEdge, с тем лишь исключением, что в этом ребре появилось поле, содержащее как адрес отправителя, так и адрес получателя.
- internalEdge.h содержит одноименный класс, описывающий ребра, соединяющие вершины в пределах одного подграфа. Описание полей аналогично ExternalEdge.
- Vertex.h содержит описание класс Vertex, реализующего вершины. Поле template_name содержит название шаблона вершины (если объект вершины создается с помощью шаблона вершины) или пустую строку (если шаблон вершины не используется). Поле body содержит перечисление объектов типа Exchange и CodeBlock, т.е. задает тело вершины. Помимо описаных выше функций readTag и emplaceParam, класс содержит функции rebindSend и rebindRecv. Эти функции подставляют значения параметров в адресные поля Exchange, т.е., к примеру, из "vert_%i%" делает "vert_1".
- Subgraph.h содержит класс, описывающий подграф. Поле condition содержит условие выполнения подграфа. Поля verticies , internal_edges и control_edges содержат соответственно перечисления вершин, внутренних и управляющих ребер подграфа. Помимо стандартного набора сеттеров, имеется функция cyclic , которая отвечает за проверку на цикличность, т.е. проверяет подграф на наличие циклов и вызывыает соответствующее исключение, если цикл есть.
- Graph.h содержит класс Graph, который описывает структуру графа. Назначение полей version, header, root, tail и main_subgraph очевидно из названия, они содержат информацию, хранящуюся в соответствующих блоках

разбираемого кода. Поля subgraphs и external_edges содержат перечисления подграфов и внешних ребер, из которых состоит граф. Функция validate подставляет значения параметров в названия вершин, ребер и т.д., и проверяет получившиеся имена объектов на уникальность. Функция cyclic производит проверку графа на наличие циклов. Функция prettyPrint выводит структуру графа в формате dot.

- StaticHelper.h содержит статичный класс с вспомогательными полями и функциями:
  * location_prefix — хранит путь до рабочей директории.
  * readFile — функция чтения данных из файла.
  * parameters — содержит имена параметров и множества значений.
  * registerParam — добавляет новые параметры в map параметров, параллельно проверяя его на уникальность.
  * replaceAll — функция производит замену всех подстрок в строке.
- crossvector.h содержит описание класса, который может хранить в себе объекты двух различных типов в определенном порядке (самодельный вектор с поддержкой хранения объектов двух различных типов в определенном порядке).