УДК 004.75

Реализация обработки метаграфов с использованием акторного подхода

The metagraph processing implementation using the actor approach

1. **Семенченко И.И. (Semenchenko I.I.), магистрант кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана, dev.ivansem@gmail.com**
2. **Гапанюк Ю.Е. (Gapanyuk Yu.E.), доцент кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана, gapyu@bmstu.ru**
3. **Ревунков Г.И. (Revunkov G.I.), доцент кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана (Bauman Moscow State Technical University), revunkov@bmstu.ru**
4. **Елисеев Г.Б. (Yeliseev G.B.), студент кафедры «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана, glebdom4@gmail.com**

1. Введение

В настоящее время модели на основе сложных сетей находят все более широкое применение в различных областях технических и естественных наук. Сложные сети рассматриваются в работах И.А. Евина [1], О.П. Кузнецова и Л.Ю. Жиляковой [2], К.В. Анохина [3] и других исследователей. На кафедре «Системы обработки информации и управления» МГТУ им. Н.Э. Баумана в рамках данного направления предложена метаграфовая модель. Данную модель предлагается применять как средство для описания сложных сетей [4], как средство для описания семантики и прагматики информационных систем [5], как средство для описания гибридных интеллектуальных информационных систем [6]. В статье [7] предложен формальный подход к разработке метаграфового исчисления.

В данной статье рассматриваются вопросы обработки метаграфов на основе метаграфового исчисления с использованием акторного подхода.

2. Акторная модель вычислений

В соответствии с [8], модель акторов представляет собой математическую модель параллельных вычислений, которая трактует понятие «актор» как универсальный примитив параллельного вычисления; в ответ на сообщения, которые он получает, актор может принимать локальные решения, создавать новых акторов, посылать сообщения.

Важной особенностью актора является отсутствие разделяемого состояния с другими акторами и остальными сущностями системы, что позволяет добавлять неограниченное количество новых компонентов, не усложняя систему.

Основой коммуникации акторов является система сообщений. Сообщения передаются между акторами по их логическим адресам. Полученное сообщение попадает в “почтовый ящик” актора. Как только актор заканчивает работу над текущей задачей, которая связана с обработкой текущего сообщения, из почтового ящика извлекается следующее сообщение и начинается его обработка. Большинство программных реализаций поддерживают задание сообщениям приоритета, это позволяет влиять на порядок обработки сообщений и обрабатывать важные сообщения в первую очередь.

Использование очереди сообщений решает серьезную проблему, которая обычно существует в любых распределенных и асинхронных системах – проблему блокировки ресурса. Даже если в момент получения сообщения актор занят, отправитель не переходит в блокировку в ожидании получения сообщения актором. Сообщение помещается в очередь обработки, и сразу после этого ресурсы, задействованные отправителем, освобождаются.

Благодаря независимости акторов, акторная система обладает большим потенциалом для распараллеливания. У акторов нет разделяемого состояния, следовательно, нет зависимостей, из-за которых два актора должны находиться на одном физическом узле. Поэтому при использовании акторной модели удается избежать большей части проблем, связанных с ограничением масштабируемости системы.

В силу рассмотренных преимуществ, акторная модель хорошо подходит для реализации метаграфовой системы. Так как (в соответствии с [7]) метаграфовые компоненты могут быть описаны в виде предикатов, наиболее естественное представление метаграфа – создать для каждого предиката соответствующий актор. При таком подходе любое взаимодействие между компонентами метаграфа будет происходить в виде пересылки сообщений.

Сформулируем основные правила трансформации метаграфовой модели данных в акторную модель вычислений:

1. Каждая вершина или метавершина является актором. Такой подход позволяет сделать каждую вершину полностью независимой от окружения. Взаимодействие между вершинами будет происходить посредством сообщений, которые могут передаваться как из вершин с текущей физической машины, так и из вершин, размещенных удаленно.
2. Ребра являются отдельными акторами. В метаграфовой системе ребра предназначены не только для создания связи между вершинами, но и сами по себе являются полноценными объектами, которые имеют собственный набор атрибутов. Выделение ребер в отдельные акторы позволяет свободно размещать их на разных узлах системы.
3. Иерархическая вложенность акторов. Иерархическая вложенность акторов необходима, чтобы иметь инструмент для создания групп вершин, которые гарантированно будут размещены на одной физической машине. Это необходимо, чтобы избежать излишней распределенности тесно связанных компонентов метаграфа по разным физическим узлам.
4. Уникальные адреса акторов. Каждый компонент метаграфа в акторной системе должен обладать уникальным именем, чтобы иметь возможность получать сообщения напрямую от любого отправителя вне зависимости от местоположения.

На рис. 1 показано представление фрагмента метаграфа в метаграфовом виде и в терминах акторной модели.

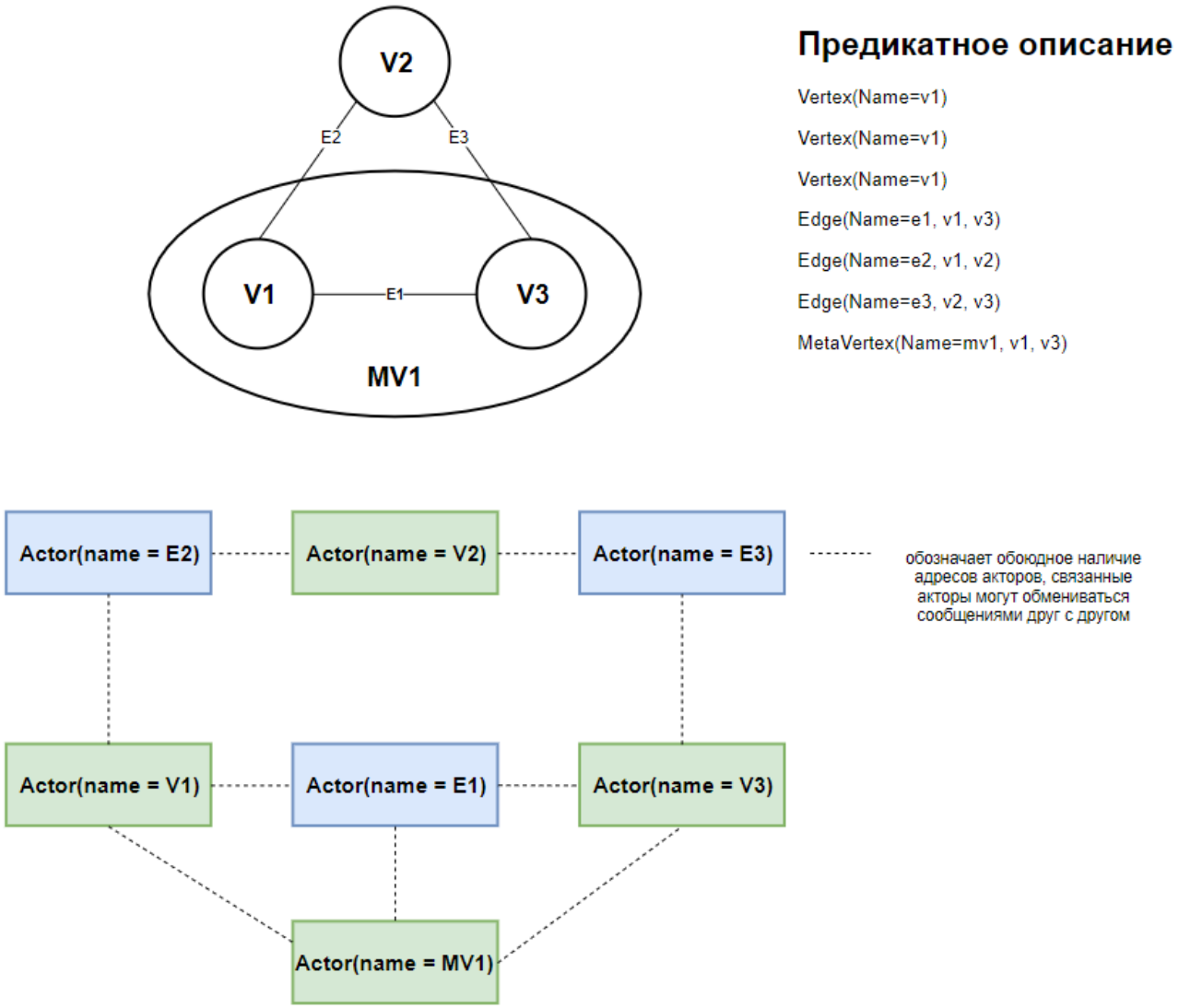

Рис. 1. Представление фрагмента метаграфа в метаграфовом виде (вверху) и в терминах акторной модели (внизу).

Fig. 1. The representation of a fragment of a metagraph in a metagraph form (above) and in terms of an actor model (below).

3. Программная реализация акторной модели метаграфовой системы

Программная реализация выполнена на языке программирования Scala с использованием библиотеки Akka, которая предоставляет широкий набор функций для реализации систем, основанных на акторной модели. Основу библиотеки Akka составляет

набор специализированных классов. За основу взят объектно-ориентированный подход, так как рассмотрение актора в виде объекта позволяет создать гибкую и понятную архитектуру системы.

В соответствии с [8], актор Akka состоит из нескольких взаимодействующих компонентов. ActorRef – это логический адрес актора, позволяющий асинхронно отправлять актору сообщения. Akka не дает получить прямой доступ к актору и поэтому гарантирует, что единственный способ взаимодействия с актором — это асинхронные сообщения. Интерфейс актора полностью закрыт, вызвать метод в акторе напрямую невозможно.

Также необходимо отметить, что отправка сообщения актору и обработка этого сообщения актором — это две отдельные операции, которые с большой вероятностью происходят в разных потоках. Разумеется, Akka обеспечивает необходимую синхронизацию, чтобы гарантировать, что любые изменения состояния будут видимы всем потокам. Такое поведение позволяет отказаться от стандартных примитивов многопоточности и писать последовательный код.

3.1. Реализация базовых элементов метаграфовой модели на основе акторного подхода

Так как Scala поддерживает объектно-ориентированный подход, то наиболее простым и эффективным вариантом представления базовых элементов метаграфовой модели в рамках программной реализации являются классы. В таблице 1 для каждого теоретического элемента метаграфовой модели приводится соответствующий ему класс на языке программирования Scala. Детальное описание элементов метаграфовой модели рассмотрено в [6, 7].

Таблица 1. Соответствие базовых элементов метаграфовой модели и классов реализации с использованием Scala.

Наименование элемента модели	**Описание элемента модели**	**Реализация с использованием Scala**
Вершина	$v_i = \{atr_k\}, v_i \in V,$ где v_i – вершина метаграфа; atr_k – атрибут; V – множество вершин метаграфа.	*class Vertex(* *val attrs: Map[String, Any]* *)*
Метавершина	$mv_i = (\{atr_k\}, MG_j), mv_i \in MV,$ где mv_i – метавершина метаграфа, принадлежащая множеству вершин *MV*; atr_k – атрибут, MG_j – фрагмент метаграфа.	*class MetaVertex(* *private val metaGraph: MetaGraph* *) extends Vertex()*
Ребро	$e_i = (v_s, v_e, \{atr_k\}),$ где e_i – ребро метаграфа; v_S – исходная вершина (метавершина) ребра; v_E – конечная вершина (метавершина) ребра; atr_k – атрибут.	*class Edge(* *val startVertex: Vertex,* *val endVertex: Vertex,* *val attrs: Map[String, Any]* *)*
Фрагмент метаграфа	$MG_i = \{ev_j\}, ev_j \in (V \cup MV \cup E),$ где MG_i – фрагмент метаграфа; ev_j – элемент, принадлежащий объединению множеств вершин *(V)*, метавершин *(MV)* и ребер *(E)* метаграфа.	*class MetaGraph(* *val vertices: Set[Vertex],* *val edges: Set[Edge]* *)*

Класс метавершины наследуется от класса вершины, что позволяет хранить вершины и метавершины в одном множестве vertices в классе MetaGraph. Также наследование помогает избежать дублирования при объявлении одинаковых полей в данных классах и объявлять общие методы. Например, метод-оператор конструирования ребра «++» определен в классе Vertex и при этом используется при создании ребер между метавершинами. Кроме того, классы Vertex и Edge наследуются от интерфейса Component, что позволяет создавать множества, состоящие как из вершин, так и из ребер. Иерархия наследования классов показана на рис. 2.

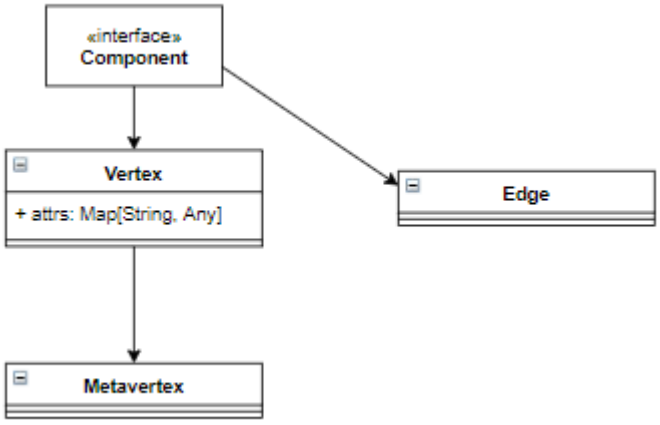

Рис. 2. Иерархия наследования.

Fig. 2. Inheritance hierarchy.

3.2. Реализация основных операторов метаграфового исчисления

В статье [7] предложен формальный подход к разработке метаграфового исчисления. В данном разделе рассмотрим, как реализовать основные операторы метаграфового исчисления с применением акторного подхода.

4.2.1. Оператор конструирования

Оператор конструирования предназначен для создания новых вершин-предикатов на основе существующих. Для синтаксического обозначения оператора конструирования используется символ сложения «+». Данный оператор является n-местным, так как метавершина может включать в себя произвольное количество вершин и ребер.

Поскольку метаграфовая модель изначально включала понятие ребра, то в исчисление включен частный случай оператора конструирования для создания ребер. В этом случае используется синтаксис «++». Единственной особенностью синтаксиса «++» является то, что данный вариант оператора является двухместным (бинарным), то есть допускает не более двух операндов.

Реализация оператора конструирования ребра достаточно проста. Так как данный оператор является бинарным, достаточно у класса Vertex определить функцию «++», которая будет принимать в качестве параметра вторую вершину и возвращать созданное ребро.

```
//функция создания ребра
def ++(endVertex: Vertex): Edge = {
  new Edge(this, endVertex)
}
//пример применения
val v1 = new Vertex("v1")
val v2 = new Vertex("v2")
val e = v1 ++ v2 //создается ребро e
```

В результате выполнения оператора создается ненаправленное ребро. Для создания направленных ребер необходимо после создания добавить ребру специальный атрибут направленности.

Для конструирования метавершин данный подход работать не будет, так как в сложении может участвовать неограниченное число компонентов. И если применить текущий подход при сложении N компонентов, будет создано N-1 метавершин, где первая вершина содержит в себе два первых компонента, а каждая последующая метавершина содержит предыдущую метавершину и очередной компонент. И в результате будет получена метавершина с уровнем вложенности N-2.

При сложении N компонентов должна получиться одна метавершина, в которую вложены все слагаемые. Для достижения такого результата используется следующее решение:

1. Бинарный оператор «+» для компонентов возвращает фрагмент метаграфа, в который входят компоненты, участвующие в сложении.

2. Бинарный оператор «+» для фрагмента метаграфа, принимающий компонент метаграфа в качестве второго аргумента, не создает новый фрагмент метаграфа, а добавляет компонент в одно из множеств vertices или edges.

При таком подходе решается проблема многократной вложенности, однако, в результате получается фрагмент метаграфа, что противоречит теоретической концепции, согласно которой результатом применения данного оператора должна быть метавершина. Пример использования оператора конструирования:

```
//создание вершин и ребер
val v1=new Vertex(); val v2=new Vertex(); val v3=new  Vertex();
val e1 = v1 ++ v2; val e2 = v2 ++ v3;
//ошибка компиляции - result будет иметь тип MetaGraph, невозможно присвоить ссылке класса MetaVertex
val result: MetaVertex = v1 + v2 + v3 + e1 + e2
```

Для решения данной проблемы для класса Metagraph было реализовано неявное преобразование (implicit conversion) в класс MetaVertex:

```
//реализация неявного преобразования
object MetaVertexImplicits {
  implicit def toMetaVertex(metaGraph: MetaGraph): MetaVertex = {
    new MetaVertex("", metaGraph)
  }
}
// импорт преобразования
import components.implicits.MetaVertexImplicits._
//теперь данный фрагмент кода выполняется корректно
val result: MetaVertex = v1 + v2 + v3 + e1 + e2
```

4.2.2. Операторы удаления и транзитивного удаления

Оператор удаления предназначен для удаления вершин-предикатов нижнего уровня из вершин-предикатов верхнего уровня. В этом случае используется синтаксис «-». При транзитивном удалении также транзитивно удаляются все элементы, теряющие логическую целостность в результате удаления. В этом случае используется синтаксис «*-». Сигнатуры методов класса MetaVertex, реализующие данные операторы:

```
//удаление
  override def -(removingVertex: Vertex): MetaVertex
//транзитивное удаление
  def *- (removingVertex: Vertex): MetaVertex
```

Нарушение целостности при удалении происходит лишь в том случае, когда из корневой вершины удаляется вершина-потомок, и при этом от корневой вершины есть альтернативный путь до удаляемой вершины. Таким образом, чтобы не допустить нарушения целостности вершины из метавершины, необходимо проверить наличие удаляемой вершины во всех дочерних метавершинах. Если удаляемая вершина присутствует хотя бы в одной дочерней метавершине, происходит ошибка, и вершина не удаляется:

```
if (metaGraph.vertices.exists(v => v.isInstanceOf[MetaVertex] &&
v.contains(removingVertex))) {
     throw new IllegalAccessException("Нарушение целостности")
}
```

При транзитивном удалении, помимо удаляемой вершины, также удаляются все дочерние метавершины с нарушением целостности. В этом случае нарушения целостности в результирующем метаграфе не возникает, так как удаляются все элементы, которые могут потенциально привести к нарушению целостности.

4.2.3. Оператор замены

Для изменения метаграфовой структуры предназначен оператор замены с синтаксисом «исходный элемент: элемент для поиска -> заменяющий элемент».

Данный оператор не является базовым, так как операция замены состоит из операций удаления и конструирования. Поэтому для реализации данного оператора не требуется создания дополнительных методов.

4. Проведение экспериментов

Для проведения экспериментов необходимо наполнить систему достаточным количеством тестовых данных. При этом необходимо учитывать, что для тестирования необходимо генерировать метаграфы различной структуры. Например, чтобы протестировать операцию создания дочерней вершины, нужно создать разветвленную иерархию из родительских вершин. Для тестирования поиска ребра по имени, необходимо, чтобы метаграф был слабосвязанным, то есть чтобы каждый компонент метаграфа соединяется с небольшим количеством других компонентов. При такой структуре процедура поиска будет выполняться по значительной части метаграфа, что отобразится в метриках производительности системы.

Для наполнения системы данными была разработана вспомогательная библиотека DataGenerator, которая наполняет систему данными, сгенерированными по следующим параметрам:

1. Количество генерируемых компонентов – данный параметр задает количество компонентов, которое будет сгенерировано библиотекой.
2. Вероятность создания корневой вершины – данный параметр отвечает за то, с какой вероятностью будет создана корневая вершина, которая добавляется в корень метаграфовой системы.
3. Вероятность создания ребра – данный параметр отвечает за то, с какой вероятностью будет создано ребро. При этом вершины, которые соединяют ребро, выбираются случайным образом.

4. Вероятность создания вложенной вершины – данный параметр отвечает за то, с какой вероятностью будет создана дочерняя вершина. Родительская вершина, в которую добавляется сгенерированная вершина, задается случайным образом.

Первый параметр задается в виде натурального числа. Остальные параметры (вероятности) задаются в диапазоне от 0 до 1. В зависимости от заданных параметров генерируется именно та структура метаграфа, которая необходима для проведения конкретного эксперимента.

4.1.Аппаратная и программная конфигурация экспериментального стенда

Для проведения экспериментов были использованы две физические машины со следующими характеристиками:

- CPU Intel Core i3, 2 ядра, 1 ГГц;
- 1 Гб. ОЗУ;
- Макс. пропускная способность сети 100 Мбит/с;
- Жесткий диск HDD 10 Гб;
- OS Ubuntu 16.04;
- JVM version "1.8.0_201".

Две физические машины были использованы потому, что полноценная проверка производительности в пределах одной физической машины невозможна, так как время передачи сообщения по сети и загруженность канала связи являются важными параметрами, влияющими на производительность системы.

4.2.Подготовка исходных данных для проведения экспериментов

На основе разработанной библиотеки DataGenerator были сгенерированы два основных набора данных. Параметры генерации наборов данных приведены в таблице 2.

Таблица 2. Параметры генерации наборов данных для проведения экспериментов.

Параметр генерации набора данных	Значение параметра для набора данных №1	Значение параметра для набора данных №2
Количество генерируемых компонентов	10000	10000
Вероятность создания корневой вершины	0,5	0,2
Вероятность создания ребра	0,1	0,6
Вероятность создания вложенной вершины	0,8	0,4

Набор данных №1 предназначен для тестирования метаграфа с большим количеством вложенных вершин. В метаграфе, обладающем такой структурой, обход в основном осуществляется от больших метавершин к вложенным вершинам, так как количество ребер в такой системе незначительно. Данная конфигурация необходима для тестирования скорости взаимодействия вложенных вершин с метавершинами.

Набор данных №2 предназначен для тестирования метаграфа, в котором немного вложенных вершин, однако, присутствует большая связанность посредством ребер.

4.3.Описание экспериментов и полученные результаты

Было проведено две группы экспериментов. Первая группа экспериментов предполагала размещение всего метаграфа на одной физической машине. Вторая группа экспериментов предполагала распределение элементов метаграфа по двум физическим машинам случайным образом.

Результаты экспериментов на примере операции поиска представлены на рис. 3 и 4. На этих рисунках по оси абсцисс отложено количество подходящих для поиска вершин, а по оси ординат время поиска в миллисекундах.

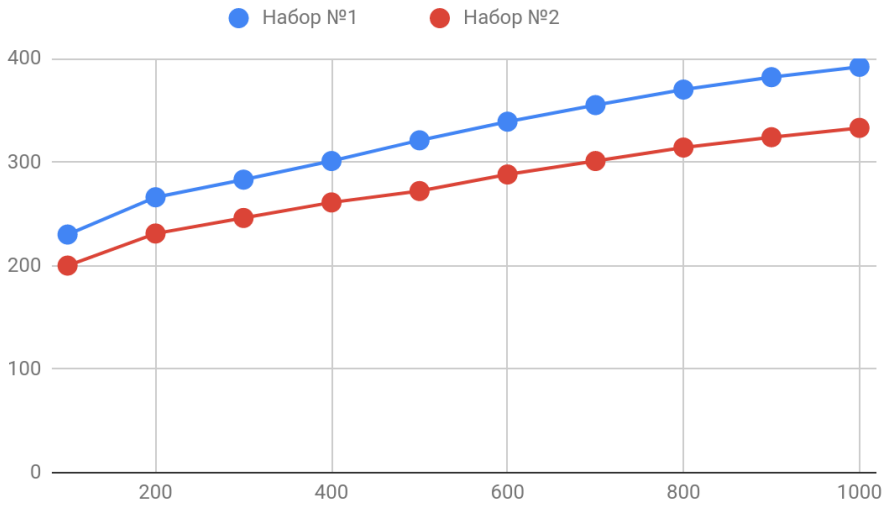

Рис. 3. Зависимость времени выполнения операции поиска вершины от количества подходящих вершин в случае размещения всего метаграфа на одной физической машине.

Fig. 3. The dependence of the time required to perform a vertex search operation on the number of suitable vertices in the case if the entire metagraph is placed on one physical machine.

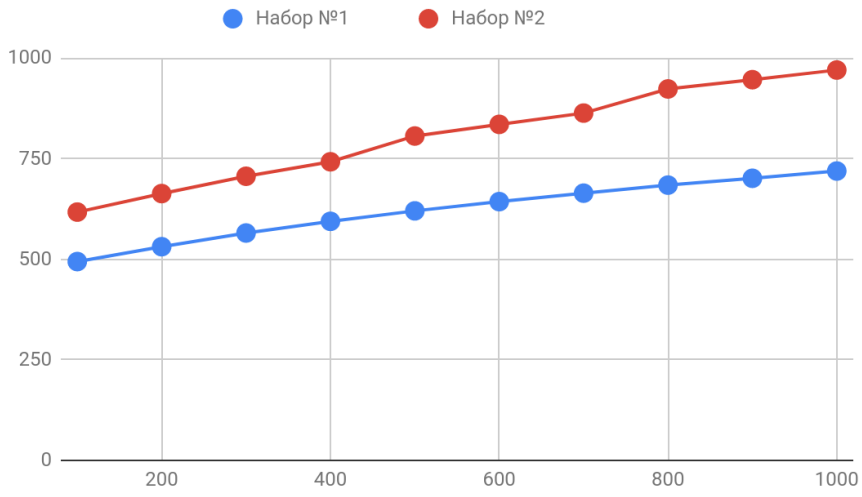

Рис. 4. Зависимость времени выполнения операции поиска вершины от количества подходящих вершин в случае распределения элементов метаграфа на двух физических машинах случайным образом.

Fig. 4. The dependence of the time required to perform a vertex search operation on the number of suitable vertices in the case of distribution of metagraph elements on two physical machines in a random manner.

В случае размещения всего метаграфа на одной физической машине (рис. 3) операция поиска работает эффективнее, когда в наборе данных присутствует меньше вложенных вершин (набор данных №2 обрабатывается за меньшее время, чем набор данных №1).

Но в случае распределения элементов метаграфа на двух физических машинах (рис. 4) результат оказывается обратным, и поиск работает менее эффективно для набора данных, в котором присутствует большая связанность посредством ребер (набор данных №2 обрабатывается за большее время, чем набор данных №1).

Таким образом, если акторы размещены более чем на одном физическом узле, то обход по ребрам становится менее эффективен, чем обход по вершинам.

Также можно отметить, что во всех случаях графики имеют приблизительно линейный характер. То есть время поиска приблизительно линейно зависит от количества подходящих для поиска вершин.

Для второй группы экспериментов была также исследована загруженность сети. Исследование данного показателя является важным, так как если сеть будет полностью занята, то система не сможет получать и отправлять сообщения между акторами, расположенными на различных физических машинах, а уже отправленные данные могут быть потеряны.

Наибольшее количество сообщений передается при создании вложенной вершины, так как необходимо не только отправить сообщение на ее создание в нужный узел, но и осуществить поиск, чтобы найти метавершину, в которую новая вершина должна быть добавлена. Поэтому именно при выполнении данной операции была измерена зависимость загруженности сети от количества акторов в системе. Результаты представлены на рис. 5. В соответствии с условиями эксперимента, все акторы расположены случайным образом на двух физических машинах. На рис. 5 по оси абсцисс отложено суммарное количество акторов в системе (на двух физических машинах), а по оси ординат уровень загруженности сети (мбит/с).

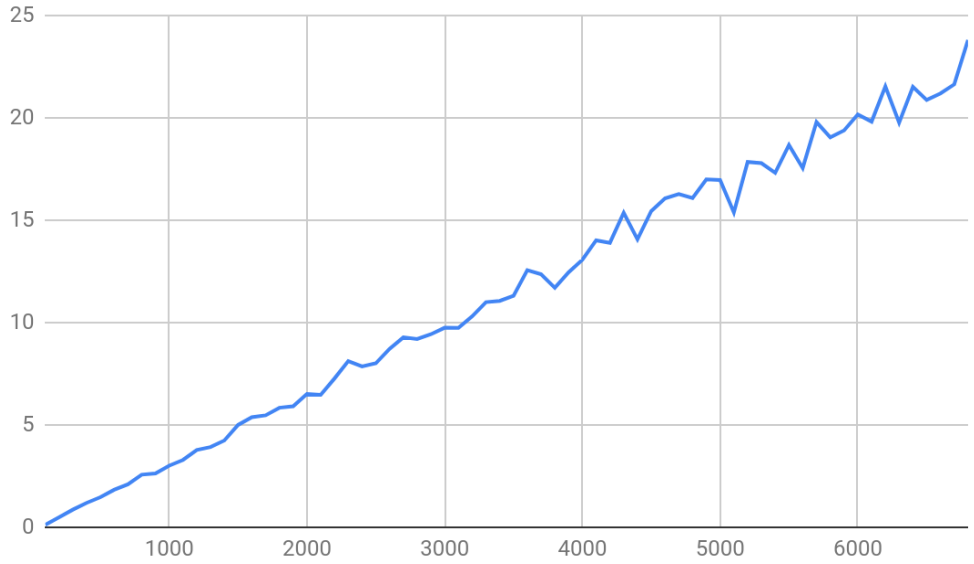

Рис. 5. Зависимость загруженности сети от количества акторов в системе при создании вложенной вершины.

Fig. 5. The dependence of the network load on the number of actors in the system when creating a nested vertex.

Рис. 5 показывает, что даже при выполнении наиболее нагруженной операции, загруженность сети растет приблизительно линейно относительно количества акторов в системе. При этом необходимо отметить, что данный эксперимент проводился только на двух физических машинах. Изучение эффективности работы акторной системы в зависимости от количества используемых сетевых узлов требует дополнительных исследований.

Важной характеристикой является также степень загруженности актора в зависимости от количества входящих сообщений. Поскольку акторы реализованы в виде потоков, то эту характеристику можно измерить через процент времени, в течение которого каждый из потоков занят выполнением операций. В момент, когда все потоки становятся занятыми, система теряет возможность обрабатывать поступающие сообщения, после чего в системе начинают возникать отказы. Для определения процента занятости каждого из потоков

используется JVisualVM. Как и в предыдущем случае, эксперименты проводились для операции создания вложенной вершины.

Результаты представлены на рис. 6. По оси абсцисс отложено количество входящих сообщений, а по оси ординат степень загруженности потока (актора). Значение 1 по оси ординат на графике соответствует 100% загруженности потока.

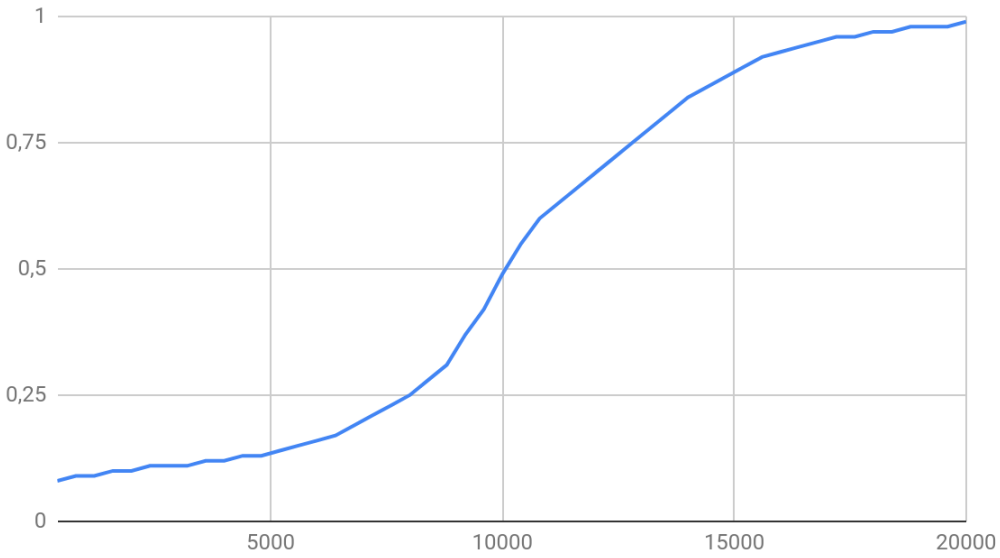

Рис. 6. Зависимость степени загруженности актора от количества входящих сообщений при создании вложенной вершины.

Fig. 6. The dependence of the actor workload on the number of incoming messages when creating a nested vertex.

Рис. 6 показывает, что при достижении интенсивности потока сообщений до 15000 в секунду, потоки (акторы) перестают успевать их обрабатывать. Для увеличения возможностей по обработке сообщений необходимо добавлять дополнительные физические узлы или увеличивать мощность CPU.

При проведении экспериментов также замерялись уровень загрузки CPU и использование оперативной памяти. По результатам измерений данных параметров мы не

приводим детальных графиков, но отмечаем их в выводах по результатам проведения экспериментов.

4.4.Выводы по результатам проведения экспериментов

По результатам проведения экспериментов можно сделать следующие выводы:

- Основным узким местом акторной системы является количество обрабатываемых в единицу времени сообщений. Для увеличения производительности акторной системы необходимо вводить дополнительные физические машины или увеличивать частоту используемых процессоров.
- Система эффективно выполняет операции также в том случае, когда компоненты метаграфа распределены по разным физическим узлам. При этом загрузка канала связи, рассчитанного на 100 мбит/с, в проведенных экспериментах не превышает 25 мбит/с. Таким образом, проведенные эксперименты показывают, что сетевое взаимодействие не является узким местом акторной системы даже при использовании сетевого оборудования со средними характеристиками.
- При распределении акторной системы по нескольким физическим узлам, обход от метавершины к вершине работает быстрее, чем обход по ребрам. Это связано с тем, что ребро может находиться на физическом узле, отличном от физического узла обеих вершин, которые соединяются этим ребром.
- При работе системы загрузка всех ядер CPU доходит до 70%, это говорит о том, что разработанная программная реализация эффективно расходует данный ресурс.
- Акторная система способна использовать весь предоставляемый объем оперативной памяти. Поскольку при запуске Java-машины объем оперативной памяти может быть ограничен сверху, то можно сделать вывод о том, что разработанная система является настраиваемой с точки зрения оперативной памяти, используемой на конкретном узле.

Таким образом, проведенные эксперименты показывают, что акторный подход для обработки метаграфов является работоспособным и в целом достаточно эффективно расходует предоставляемые вычислительные ресурсы. Различные характеристики вычислительной системы (характеристики сети, характеристики узлов сети) могут настраиваться в целях эффективной работы акторной системы.

5. Выводы

Важной особенностью акторного подхода является отсутствие разделяемого состояния между акторами, что позволяет добавлять неограниченное количество новых компонентов, не усложняя систему.

Акторная модель хорошо подходит для реализации метаграфовой системы. Каждый элемент метаграфовой модели может быть представлен в виде актора. Взаимодействие между элемента осуществляется в виде обмена сообщениями.

Как базовые элементы метаграфовой модели, так и основные операторы метаграфового исчисления, могут быть реализованы с использованием акторного подхода. В предлагаемом подходе для реализации используется язык программирования Scala и акторная библиотека Akka.

Результаты экспериментов показывают, что акторный подход для обработки метаграфов является работоспособным и в целом достаточно эффективно расходует предоставляемые вычислительные ресурсы.

Литература

1. Евин И.А. Введение с теорию сложных сетей //Компьютерные исследования и моделирование. 2010, Том 2, №2, с. 121-141.
2. Кузнецов О.П., Жилякова Л.Ю. Сложные сети и когнитивные науки // Нейроинформатика-2015. XVII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: МИФИ. 2015. С. 18.

3. Анохин К.В. Когнитом: гиперсетевая модель мозга // Нейроинформатика-2015. XVII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: НИЯУ МИФИ. 2015. С. 14-15.
4. Черненький В.М., Терехов В.И., Гапанюк Ю.Е. Представление сложных сетей на основе метаграфов // Нейроинформатика-2016. XVIII Всероссийская научно-техническая конференция. Сборник научных трудов. Ч. 1. М.: НИЯУ МИФИ, 2016.
5. Самохвалов Э.Н., Ревунков Г.И., Гапанюк Ю.Е. Использование метаграфов для описания семантики и прагматики информационных систем. Вестник МГТУ им. Н.Э. Баумана. Сер. «Приборостроение». 2015. Выпуск №1. С. 83-99.
6. Черненький В.М., Терехов В.И., Гапанюк Ю.Е. Структура гибридной интеллектуальной информационной системы на основе метаграфов. Нейрокомпьютеры: разработка, применение. 2016. Выпуск №9. С. 3-14.
7. Гапанюк Ю.Е. Подход к разработке метаграфового исчисления. Динамика сложных систем – XXI век. 2018. Выпуск №3. С. 40-46.
8. Vernon V. Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Addison-Wesley Professional, 2015. – 480 p.