МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

С.В. ВОСТОКИН

МЕТОД СОЗДАНИЯ ПАРАЛЛЕЛЬНЫХ И РАСПРЕДЕЛЕННЫХ ПРОГРАММ В ПАРАДИГМЕ АКТОРОВ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.04.02 Фундаментальная информатика и информационные технологии

САМАРА
Издательство Самарского университета
2023

УДК 004.423(075)
ББК 3973я7
В780

Рецензенты: д-р техн. наук, проф С.П. Орлов,
д-р физ.-мат. наук Д.Л. Головашкин

Востокин, Сергей Владимирович

В780 Метод создания параллельных и распределенных программ в парадигме акторов: учебное пособие / *С.В. Востокин*. – Самара: Издательство Самарского университета, 2023. – 80 с.

ISBN 978-5-7883-1947-6

Предложен метод визуализации и программирования параллельных и распределенных вычислений на основе алгоритмической интерпретации классической модели акторов. Описано применение метода для программирования процессов с различными типами коммуникационных топологий, представлены примеры программ на языке C++ в системе Templet, приведены практические задания.

Предназначено для обучающихся, изучающих параллельное и распределенное программирование в рамках образовательных программ бакалавриата и магистратуры в области информационных технологий.

Подготовлено на кафедре программных систем.

УДК 004.423(075)
ББК 3973я7

ISBN 978-5-7883-1947-6

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Предлагаемое вашему вниманию учебное пособие «Метод создания параллельных и распределенных программ в парадигме акторов» содержит теоретический материал для проведения практических и лабораторных занятий по курсам, связанным с тематикой параллельных и распределенных вычислений.

Сложные процессы, например, составляющие деятельность крупного предприятия, реализующиеся при обработке больших объемов данных или при проведении ресурсоемких расчетов, изначально имеют параллельную и распределенную природу. Поэтому их не удается описать в виде обычных последовательных алгоритмов. Возникают сложности построения точного и полного описания таких процессов, их понимания, в итоге приводящие к некорректной реализации в виде программных систем.

Строгий математический подход решения проблемы описания параллельных и распределенных процессов основан на логических или алгебраических теориях процессов. Альтернативный, более близкий к практике программирования подход, вариант которого рассматривается в данном пособии, основан на специальном понимании способа организации вычислений – модели вычислений. В качестве такой классической модели вычислений в пособии используется известная модель акторов, предложенная Хьюиттом, Бишопом и Штайгером в 1973 году [1]. Алгоритмическая интерпретация этой изначально функциональной модели вычислений лежит в основе рассматриваемого в данном пособии метода описания параллельных и распределенных процессов. Такой подход является более удобным для освоения программистами. В тоже время он имеет

непосредственную связь со строгим математическим описанием параллельных процессов в виде «машины состояний» [2].

Все примеры процессов, рассматриваемые в учебном пособии, реализованы в виде программ на языке C++ в системе Templet SDK и доступны для экспериментирования непосредственно из браузера. Для самостоятельного изучения примеров потребуется открыть страницу проекта Templet на сервисе GitHub по адресу https://github.com/the-templet-project/templet, найти на странице описания проекта кнопку «launch binder». При нажатии на кнопку происходит запуск среды JupyterLab, где в интерфейсе интерактивных блокнотов можно модифицировать и запускать примеры программ. Файлы примеров, включающие файлы с кодом на языке C++ (с расширением hpp) и файлы интерактивных блокнотов (с расширением ipynb), располагаются в папке /samples/booksmpl/.

На базе рассмотренных примеров в конце глав 2, 3, 4 даются задания для самостоятельного изучения материала пособия на практических и лабораторных занятиях. Предполагается, что задания можно выполнять непосредственно в браузере в среде Templet SDK. Другим вариантом является копирование содержания репозитория проекта на локальный компьютер. В примерах каталога /samples/ рассматривается работа с системой в среде программирования Visual Studio. Стиль изложения алгоритмов и методов программирования жестко не привязан к особенностям системы Templet. Описываемый метод программирования может быть применен для выполнения учебных заданий и решения практических задач с использованием других современных фреймворков и языков программирования, основанных на модели акторов (Akka для Scala, Java, C#; Go, Rust и др.).

Материал учебного пособия скомпонован следующим образом. В первой главе излагается интерпретация модели акторов, основанная на алгоритмическом подходе. Её составной частью является

графическое представление структуры акторных программ, облегчающее понимание их работы. Элементы графической нотации рассматриваются на классическом примере процесса «пинг-понг».

Вторая глава посвящена описанию процессов с коммуникационной топологией «звезда». Важными практическими вариантами таких процессов являются процесс типа «применить ко всем» (map) и процесс «портфель задач» (bag of tasks). Перечисленные типовые процессы рассмотрены на примере задачи о нахождении числа π и задачи об интегрировании функции методом адаптивной квадратуры.

В третьей главе рассматриваются процессы с линейной коммуникационной топологией. На примере соединения акторов в кольцо изложен способ программирования параллельного алгоритма умножения матриц с циркуляцией столбцов. На примере соединения акторов в цепочку показан способ параллельного программирования сеточного метода с линейной декомпозицией области данных путем модификации последовательного итеративного алгоритма Гаусса-Зейделя.

Четвертая глава рассматривает метод распараллеливания ациклических последовательных алгоритмов в задачах обработки паросочетаний элементов информационных массивов. Первый пример главы иллюстрирует реализацию параллельной обработки паросочетаний по методу кругового спортивного турнира. В нем ищется ближайший по «расстоянию» сосед для каждой из «точек» данных, составляющих информационный массив. Второй пример предлагает простую реализацию так называемой сортирующей сети на базе последовательного алгоритма сортировки «пузырьком».

Материал пособия прошел апробацию при преподавании профильных дисциплин бакалавриата и магистратуры Института информатики и кибернетики Самарского национального исследовательского университета имени академика С.П. Королева, частично представлен в работах [3,4], применялся при решении задач обработки данных [5] и организации ресурсоемких вычислений [6].

1. АЛГОРИТМИЧЕСКАЯ ИНТЕРПРЕТАЦИЯ МОДЕЛИ АКТОРОВ

1.1. Классическая модель акторов и её интерпретация

Классическая модель акторов. Принцип построения алгоритмической интерпретации модели акторов. Спецификация модели акторов: цикл обработки сообщений, примитивные операции модели. Анализ спецификации модели акторов.

Классическая модель акторов. Под моделью акторов обычно понимается модель параллельных вычислений, строящаяся на основе понятия актора (англ. actor «актёр», «действующий субъект»), считающегося универсальным примитивом параллельного исполнения. В данной модели актор взаимодействует с другими акторами путём обмена сообщениями. Каждый из акторов в ответ на получаемые сообщения может принимать локальные решения, создавать новые акторы, посылать свои сообщения, устанавливать, как следует реагировать на последующие сообщения. Обработка сообщений в классических интерпретациях модели акторов, основанных на концепциях функционального программирования – это мгновенное действие, совершаемое в соответствии с известной функцией «поведения». Результатом является присвоение актору новой функции «поведения», которая будет использована при обработке следующего сообщения.

Параллелизм модели акторов обусловлен возможностью одновременной отправки нескольких сообщений при обработке полученного актором сообщения и одновременной обработке сообще-

ний в двух и более различных акторах. Важной особенностью классической модели акторов является отсутствие других, явно формулируемых ограничений на поведение акторов и на порядок обмена сообщениями между ними. Например, приведенное выше описание модели лишь неявно предполагает доставку отправленного в актор сообщения, но никак не регламентирует порядок и время доставки сообщений даже между двумя выделенными акторами. Это приводит к неограниченному недетерминизму модели, то есть случайному поведению при многократных запусках одной и той же совокупности связанных акторов (называемых системой акторов) из одинакового начального состояния.

На практике для применения модели акторов с целью описания алгоритмов, осуществляющих параллельные вычисления или обработку данных, требуется конкретизировать модель, введя некоторые допущения и ограничения. Они направлены на адаптацию модели к свойствам реальных компьютеров и языков программирования, параллелизм в которых реализуется в форме параллельного выполнения нескольких последовательностей инструкций (нитей/потоков выполнения).

Принцип алгоритмической интерпретации. Для описания вычислений «в стиле» модели акторов применим следующий прием, используемый в области императивного программирования при распараллеливании последовательных алгоритмов.

Допустим, что в последовательной программе имеется циклический участок и выполняются два условия для кода, содержащегося в теле цикла: (а) нет такой переменной, которой присваиваются значения на разных итерациях цикла; (б) нет такой переменной, которой на одной итерации цикла присваивается значение, а на другой итерации значение переменной используется в вычислениях. Очевидно, что в этом случае код тела цикла можно преобразовать в процедуру, параметризированную значением итератора цикла (в про-

стейшем случае переменной цикла), и запускать на одновременное выполнение экземпляры этой, построенной из тела цикла, процедуры со всеми возможными значениями подстановки фактического значения итератора цикла вместо параметра цикла.

Из этого следует, если в программе имеется цикл, обладающий описанным свойством, достаточно указать путем аннотации в исходном последовательном алгоритме возможность параллельного выполнения этого цикла. В результате применения такой аннотации, мы фактически получим параллельный алгоритм из исходного последовательного алгоритма.

Поступая по аналогии с рассмотренным выше примером, построим такую структуру последовательного кода с распараллеленным циклом, которая бы своим поведением соответствовала (с некоторыми допущениями) поведению акторов в классической акторной модели. Из-за того, что вводимое таким способом определение модели акторов основано на концепции «распараллеливания» исходного описания вычислительного процесса в виде последовательного алгоритма, приведенная далее интерпретация модели акторов названа «алгоритмической интерпретацией» [3].

Спецификация модели акторов. Базовыми сущностями модели являются акторы и сообщения. Для описания операционной семантики модели в первую очередь нам потребуются переменные, описывающие состояния акторов и сообщений в произвольный момент времени (рис. 1.1).

```
18
19 actor's variables  -- active[a]  in {true,false} for any program state
20                      message[a] in set(Messsages) for any program state
21
22 message's variables-- mobile[m]  in {true,false} for any program state
23                      actor[m]   in set(Actors) for any program state
```

Рис. 1.1. Переменные состояния модели и области их значений

Условимся, что обработка поступившего в актор сообщения имеет длительность в дискретном модельном времени. Тогда в некоторый момент наблюдения произвольный актор окажется либо «активным», то есть выполняющим обработку сообщения, либо «пассивным», когда сообщение не обрабатывается. Активность актора *a* обозначается переменной *active*[*a*] булевого типа в строке 19 на рис. 1.1.

Активность актора *a* вызывается сообщением, поступившим в него в момент наблюдения или ранее. Его хранит переменная *message*[*a*] типа «сообщение». Условимся, что в начальный момент времени переменная *message*[*a*] проинициализирована значением некоторого произвольного сообщения (см. строку 20 на рис. 1.1).

Состояние сообщения *m* будем считать «активным» в некоторый момент наблюдения, если сообщение было отправлено, но еще не поступило актору-адресату. Активность сообщения *m* обозначает переменная *mobile*[*m*] булевого типа в строке 22 на рис. 1.1.

Сообщение, являясь активным, «движется» в определенный актор. Соответственно, пассивное сообщение ранее было доставлено в некоторый актор. Связь сообщения *m* с актором в произвольный момент времени хранится в переменной *actor*[*m*] типа «актор» (см. строку 23 на рис. 1.1). Связь сообщения с актором и признак активности в начальном состоянии задаются специальными примитивами (базовыми операциями) модели при инициализации.

Заметим, что в данной спецификации не принципиально, чем являются акторы и сообщения (значениями, адресами, объектами и т.д.). В реализации Templet на языке программирования C++ – это экземпляры объектов, содержащие поля данных, соответствующие рассмотренным переменным.

Опираясь на введенное определение состояния акторов и сообщений, с использованием псевдокода определим общую схему алгоритма вычислительного процесса, показанную на рис. 1.2.

Строка 27 на рис. 1.2 соответствует коду инициализации акторного алгоритма. Этот код может включать операторы или «шаги» трех типов в произвольной последовательности (обозначено знаками { | }, как принято в описании синтаксиса языков программирования на метаязыке EBNF).

Шаг *init*() – это любая операция алгоритма, не изменяющая состояние акторов и сообщений.

```
26 procedure ActorModel
27 |  { init() | send(_,_) | bind(_,_) }
28 |  for( m : mobile[m]=true ) parallel
29 |  |    local a := actor[m]
30 |  |    message[a] := m
31 |  |    mobile[m] := false
32 |  |    active[a] := true
33 |  |    { next(m,a) | send(_,_)  | access(_,a) }
34 |  |    active[a] := false
35 |  end for
36 end ActorModel
```

Рис. 1.2. Алгоритмическая интерпретация модели акторов

Шаг *send*(_,_) – это примитив модели, определяющий в начальном состоянии активное «стартовое» сообщение, и актор, в который это сообщение доставляется (см. рис. 1.3).

```
38 procedure send(m,a) atomic
39 |  actor[m]  := a
40 |  mobile[m] := true
41 end send
```

Рис. 1.3. Алгоритмическая интерпретация примитива send() модели акторов

Для того, чтобы вычисления могли начаться, в модели акторов нужно предусмотреть сообщения, находящиеся на доставке в

начальный момент времени. В описываемой модели принято допущение, что все акторы являются пассивными в начальный момент времени (для любого *a* : *active*[*a*] = *false*).

Шаг *bind*(_,_) также является примитивом модели. Он определяет связь неактивных в начале вычислений сообщений с акторами. Псевдокод операции показан на рис. 1.4.

```
43 procedure bind(m,a) atomic
44 |  actor[m]  := a
45 |  mobile[m] := false
46 end bind
```

Рис. 1.4. Алгоритмическая интерпретация примитива bind() модели акторов

Распараллеливаемый цикл содержится в строках 28-35 рис. 1.2. Пометка *parallel* в голове цикла (строка 28) говорит о том, что шаги внутри независимых итераций цикла могут выполняться одновременно. Тело цикла описывает алгоритм обработки сообщения актором-получателем. Рассмотрим этот алгоритм.

В строке 29 определяется актор-получатель сообщения. Это актор, в который направлялось сообщение *m*, являющееся параметром цикла. Фактическим значением параметра цикла *m* может быть любое сообщение, находящееся на доставке *mobile*[*m*] = *true*, что указано в голове цикла в строке 28.

В строке 30 актор-получатель «связывается» с только что поступившим сообщением, в строке 31 сообщение переводится в пассивное состояние, в строке 32 актор-получатель переводится в активное состояние.

Далее в строке 33 условно показан код обработки сообщения в акторе, специфичный для конкретного алгоритма. Этот код состоит из произвольной последовательности шагов трех типов (также обозначено с использованием операторов { | } метаязыка EBNF).

Шаг *next*(*m*,*a*) обозначает любое действие в алгоритме, не меняющее состояние акторов или сообщений (см. рис. 1.1), которое может изменять только значения переменных, специфичных для конкретного алгоритма. Параметры (*m*,*a*) указывают на контекст выполнения этого шага: актор *a* и сообщение *m*. Иными словами разработчик алгоритма понимает и может проверить в коде, для какой пары сообщение-актор выполняется этот шаг. Шаг *send*(_,_) (см. рис. 1.3) является примитивной операцией модели, посылающей некоторое сообщение в некоторый актор. Шаг *access*(_,*a*) является примитивной операцией модели, проверяющей возможность доступа к некоторому сообщению во время обработки сообщения в акторе *a*. Псевдокод шага показан на рис. 1.5.

```
48 function access(m,a) atomic
49 |  access := ( mobile[m] = false /\ actor[m] = a )
50 end access
```

Рис. 1.5. Алгоритмическая интерпретация примитива access() модели акторов

Цикл обработки сообщения завершается переводом актора-получателя поступившего сообщения в неактивное состояние в строке 34 тела цикла (см. рис. 1.2).

Анализ спецификации модели акторов. Рассмотрим подробнее, при каких условиях возможно параллельное выполнение итераций цикла в строках 28-35 на рис. 1.2 и при каких условиях выполнение специфического для конкретного алгоритма кода в теле цикла (строка 33) будет происходить эквивалентно последовательному алгоритму.

Как показано выше, параллельное выполнение итераций цикла возможно, если нет переменных, которым присваиваются значения на разных итерациях. Такой конфликт по присваиванию может возникнуть в строке 30. Действительно, если два сообщения отправ-

лены в один актор, их обработка должна производиться строго последовательно. Таким образом, параллельно может выполняться любая пара итераций цикла на рис. 1.2, если для фактических параметров этих итераций m_1 и m_2 верно: $actor[m_1] \neq actor[m_2]$.

Заметим, что строгое определение семантики параллельного выполнения требует уточнения того, что в случае возникновения конфликта $actor[m_1] = actor[m_2]$, когда появляются новые сообщения в состоянии $mobile[m] = true$, некоторая готовая выполниться итерация не может откладываться неопределенно долго. Это условие называется справедливостью (fairness) выполнения.

Еще одним ограничением на возможность параллельного выполнения итераций цикла является отсутствие в них переменной, которой в одной итерации присваивается значение, а в другой итерации значение переменной используется в вычислениях. Такого рода конфликт потенциально возможен при параллельном выполнении примитивов модели *send*(*m*,_) и *access*(*m*,_) в разных итерациях цикла с одинаковым значением сообщения *m*.

Для того, чтобы параллельное выполнение таких итераций было возможным и при этом не нарушало желаемое свойство модели, заключающееся в том, чтобы выполнение тела итерации выглядело одинаково при последовательном и параллельном выполнении, вводятся два ограничения.

Во-первых, примитивы модели *send*(_,_) и *access*(_,_) объявляются неделимыми или атомарным, что специально отмечено на рис. 1.3 в строке 38 и на рис. 1.5 в строке 48. Это можно понимать, как мгновенное выполнение двух присваиваний в *send*(_,_) и использование двух параметров конъюнкции в *access*(_,_), относящихся к одному состоянию. Во-вторых, выполнять операцию *send*(*m*,_) можно тогда и только тогда, когда предварительно выполнена проверка *access*(*m*,*a*) = true или истинное значение предиката *access*(*m*,*a*) очевидно из контекста.

Перечисленные ограничения гарантируют, что при выполнении 33 строки цикла на рис. 1.2 в любой параллельной итерации, истинность предиката *access*(*m*,_) может измениться только в результате выполнения примитива *send*(*m*,_) именно в этой итерации. То есть выполнение тела итерации выглядит как последовательное с точки зрения наблюдателя, выполняющего код тела итерации.

Заметим, что выбор фактического параметра цикла при запуске итерации на рис. 1.2 является недетерминированным. Это соответствует случайному порядку доставки сообщений в акторы в классической акторной модели.

Последним важным моментом эквивалентности рассматриваемой интерпретации и классической модели акторов является изоляция состояния акторов. В реальных алгоритмах выполняются не только примитивы модели, но также происходит чтение и модификация специфичных для алгоритма переменных. Поэтому при выполнении шага алгоритма внутри параллельного цикла нужно не только понимать, как происходит изменение значений переменных на этом шаге, но и значения каких переменных на этом шаге гарантированно не изменятся. Иными словами, какие переменные образуют изолированное состояние актора в момент обработки сообщения в нем. Такое понимание в рассматриваемой модели достигается мысленным (или закрепленным синтаксисом конкретной реализации модели) соотнесением каждой переменной некоторому актору или сообщению. Например, контекст изолированного шага *next*(*m*,*a*) будут составлять все переменные, соотнесенные с актором *a*, а также все переменные, соотнесенные с сообщениями *m*, такие, для которых верно условие *access*(*m*,*a*) = *true*. Как показано в примерах глав 2-4, для некоторых алгоритмов возможны и более сложные варианты соотнесения переменных акторам и сообщениям, гарантирующие изоляцию состояния актора.

Таким образом, в основных аспектах рассмотренная модель повторяет поведение классической модели акторов. Однако при этом

модель основана только на концепции распараллеливания последовательного алгоритма, чтобы построить на её основе конкретный вычислительный процесс требуется добавить специфичные для этого процесса шаги алгоритма исключительно с последовательной и детерминированной семантикой выполнения.

Полный псевдокод алгоритмического описания модели акторов, фрагменты которого использованы в этом параграфе, содержится в репозитории проекта Templet по адресу ~/lib/semantics.txt.

1.2. Визуализация акторных алгоритмов и программ

Графическое обозначение состояний модели. Графическое обозначение типов акторов и обработчиков сообщений.

Графическое обозначение состояний модели. Определение операционной семантики модели фактически означает задание некоторого абстрактного автомата (state machine), для которого определено начальное состояние в некотором множестве возможных состояний и функция перехода из одного состояния в другое [2]. Введенное в п. 1.1 определение модели акторов на рис. 1.2 тоже задает такой автомат. Начальное состояние формируется после выполнения шагов строки 27, а переход в следующий шаг определяется циклом в строках 28-35.

Для лучшего понимания особенностей работы автомата, представляющего вычисления в данном варианте модели акторов, визуально изобразим состояния модели. Пара таких визуальных представлений состояний модели будет соответствовать действию функции перехода, а последовательность – поведению модели в дискретном времени.

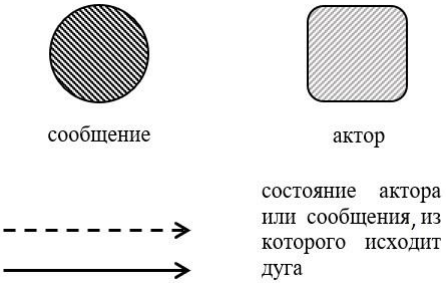

Рис. 1.6. Графические примитивы, иллюстрирующие состояние модели

Введем условные обозначения в виде графических примитивов, показанных на рис. 1.6. Актор обозначим квадратом, а сообщение – кругом. Состояние акторов и сообщений будем обозначать сплошными или пунктирными стрелками, исходящими из акторов и сообщений соответственно. Сплошная стрелка обозначает активное состояние, а пунктирная – неактивное. Направление стрелки показывает связь актора с сообщением либо сообщения с актором.

Обозначение актора в активном состоянии выглядит, как показано на рис. 1.7. Изображение означает состояние, в котором актор *a* обрабатывает сообщение *m*. Справа на рис. 1.7 и следующих рисунках показана математическая запись такого состояния в терминах переменных, введенных в п.1.1.

Рис. 1.7. Активное состояние актора

Обозначение актора в пассивном состоянии выглядит, как показано на рис. 1.8. Изображение означает состояние, в котором актор *a* не обрабатывает сообщения. Сообщение *m* было последним,

которое обрабатывал актор, или это начальное состояние вычислений.

Рис. 1.8. Пассивное состояние актора

Обозначение сообщения в активном состоянии выглядит, как показано на рис. 1.9. Сообщение *m* было отправлено в актор *a* при помощи вызова примитива *send*(*m*,*a*) и в данном состоянии находится «на доставке» в этот актор.

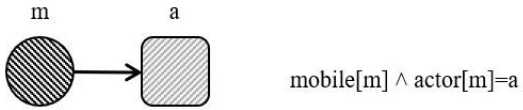

Рис. 1.9. Активное состояние сообщения

Сообщение в пассивном состоянии выглядит, как показано на рис. 1.10. Сообщение *m* было доставлено и обработано в акторе *a* в предшествующем состоянии, или настройка связи сообщения с актром выполнена в начальном состоянии с использованием примитива *bind*(*m*,*a*).

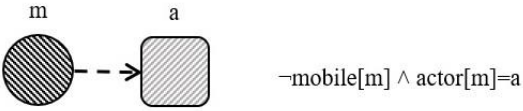

Рис. 1.10. Пассивное состояние сообщения

Графическое обозначение типов акторов и обработчиков сообщений. На основе введенных обозначений состояния модели

для передачи её динамики введем дополнительные обозначения для передачи статических, не меняющихся во времени свойств модели: типов акторов и обработчиков сообщений в однотипных акторах. В нашем определении модели акторов обработчики сообщений – это участки тела цикла обработки сообщений, соответствующие строке 33 на рис. 1.2.

Если в начальном состоянии имеется «стартовое» сообщение, направляющееся в актор, то для акторов такого типа будем использовать обозначение кружка внутри прямоугольника, показанное в левой части рис. 1.11. Одновременно рисунок обозначает, что в теле цикла на рис. 1.2 имеется обработчик «стартового» сообщения.

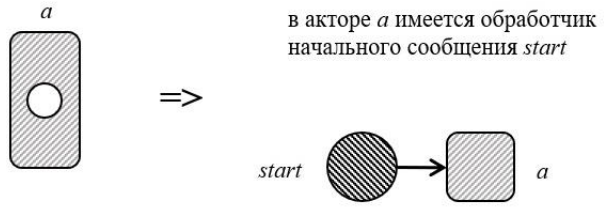

Рис. 1.11. Обработчик начального сообщения в акторе

Если с актором в начальном состоянии связано сообщение, то данное свойство передается путем изображения кружка у границы прямоугольника, представляющего актор. Когда имеется несколько сообщений, связанных с актором в начальном состоянии, то изображается несколько кружков по числу таких сообщений (см. рис.1.12).

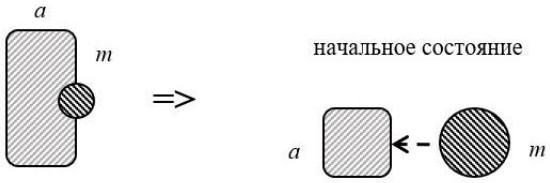

Рис. 1.12. Локальный объект-сообщение в начальном состоянии

Одновременно с фактом наличия связанного с актором сообщения обозначение говорит о том, что имеется обработчик для этого сообщения. Он вызывается в состоянии, показанном в правой части рис. 1.13.

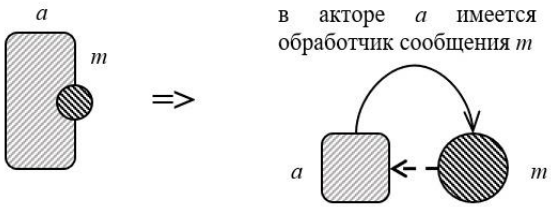

Рис. 1.13. Обработчик сообщения-ответа

Такой обработчик будет часто использоваться в рассматриваемых далее алгоритмах, так как удобно передавать одно и тоже сообщение между парой связанных акторов. В данном случае поступление сообщения в актор будет являться ответом на запрос, который ранее был отправлен в этом сообщении из рассматриваемого актора в другой связанный актор.

Если актор может получать сообщения-запросы, то обработчик этих сообщений обозначается незакрашенным кружком, расположенным на границе актора, как показано на рис. 1.14. Актор может получать сообщения-запросы в разные обработчики в зависимости от типов и назначения в алгоритме поступающих сообщений. В этом случает рисуется несколько кружков.

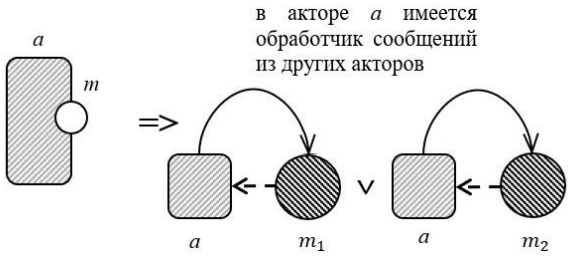

Рис. 1.14. Обработчик сообщений-запросов

Также обозначение рис. 1.14 говорит о том, что в обработчик сообщений-запросов может поступать несколько различных сообщений от разных акторов, как показано в правой части рис. 1.14.

Таким образом можно с использованием акторов передавать взаимодействие между несколькими акторами-клиентами (рис. 1.12 и рис. 1.13), отправляющими свои запросы-сообщения в некоторый «порт» актора-сервера (рис. 1.14). Ниже рассмотрен простейший пример такого клиент-серверного взаимодействия процессов «пинг-понг», поясняющий применение введенных графических обозначений для рассматриваемого варианта модели акторов.

1.3. Метод программирования на примере «пинг-понг»

Структурированный алгоритм процесса. Последовательность состояний, определяемая алгоритмом. Типы акторов, обработчики сообщений. Система акторов процесса «пинг-понг».

Структурированный алгоритм процесса. У процесса «пинг-понг» есть два актора *ping* и *pong*. Актор ping отправляет запрос актору *pong* в сообщении *out*. Актор pong в том же сообщении отправляет ответ актору ping, и на этом процесс завершается. В начале вычислений актор *ping* запускается стартовым сообщением *start*. В терминах введенной модели псевдокод процесса «пинг-понг» можно записать, как показано на рис. 1.15.

Рассмотрим, как построен псевдокод процесса «пинг-понг». За основу его структуры был взят псевдокод модели акторов (рис. 1.2). Строка 27 на рис. 1.2, содержащая абстрактный код инициализации, заменена кодом, запускающим сообщение *start* в актор *ping* (строка 3 на рис. 1.15) и связывающим сообщение *out* с актором *ping* (строка 4 на рис. 1.15). Строка 33 на рис. 1.2, содержащая абстрактный код обработки сообщения в акторе, заменена на рис. 1.15 на код, включающий

```
1  procedure PingPong
2  |-------------init()-----------------
3  |  send(start,ping)
4  |  bind(out,ping)
5  |----------------------------------
6  |  for( m : mobile[m]=true ) parallel
7  |  |   local a := actor[m]
8  |  |   message[a] := m
9  |  |   mobile[m] := false
10 |  |   active[a] := true
11 |  |----------next(m,a)--------------
12 |  |   if m=start /\ a=ping then
13 |  |   |   print "Ping started.."
14 |  |   |   send(out,pong)
15 |  |   else if m=out /\ a=pong then
16 |  |   |   print "Pong received message from Ping.."
17 |  |   |   send(out,ping)
18 |  |   else if m=out /\ a=ping then
19 |  |   |  print "Pong received message from Ping.."
20 |  |   |  stop
21 |  |   end if
22 |  |--------------------------------
23 |  |   active[a] := false
24 |  end for
25 end PingPong
```

Рис. 1.15. Структурированный алгоритм процесса «пинг-понг»

обработку трех событий: в строке 12 – поступление сообщения *start* в актор *ping*; в строке 15 – поступление сообщения-запроса *out* в актор *pong* из актора *ping*; в строке 18 – поступление сообщения-ответа *out* в актор *ping*. Конкретизация модели для других алгоритмов будет выглядеть аналогично: специфичный для алгоритмов код будет помещаться в секции *init* в строке 2 и *next*(*m*,*a*) в строке 11, при этом должны соблюдаться правила вызова примитивных операций модели. В частности, правило вызова операции отправки сообщения разрешает вызов *send*(*m*,_) только при выполнении предусловия *access*(*m*,*a*).

Последовательность состояний алгоритма. Рассмотрим поведение процесса «пинг-понг», описываемое последовательностью его состояний с использованием введенной графической нотации.

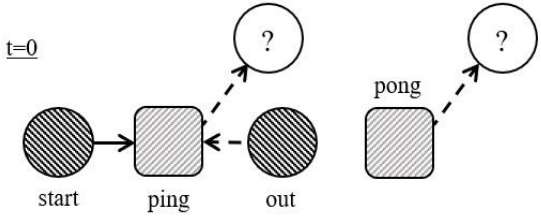

Рис. 1.16. Начальное состояние вычислений в модели «пинг-понг»

Состояние процесса «пинг-понг» в начальный момент времени (t=0) показано на рис. 1.16. Обратите внимание, что модель не предписывает определенных значений переменным *message*[*a*], что передано на рис. 1.16 стрелками, указывающими на произвольные сообщения (со знаками «?» в кружках на рис. 1.16 и др.).

В следующий момент времени (t=1) начинается обработка сообщения *start* в акторе *ping*, как показано на рис. 1.17.

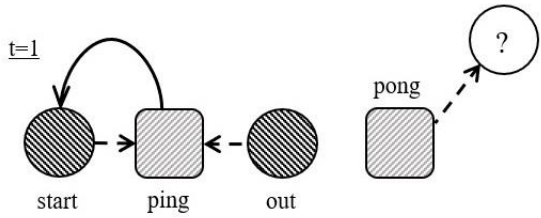

Рис. 1.17. Актор ping обрабатывает начальное сообщение

Результатом обработки сообщения *start* в момент (t=2) является отправка сообщения *out* в актор *pong*, как показано на рис. 1.18. Заметим, что в нашей интерпретации модели можно наблюдать промежуточное состояние, когда актор *ping* еще не закончил обработку

сообщения *start*, а сообщение *out* уже находится на доставке (в том числе уже доставлено или обработано в акторе *pong*). Так проявляется недетерминизм модели акторов. Для простоты не будем рассматривать эти альтернативные последовательности смены состояний процесса.

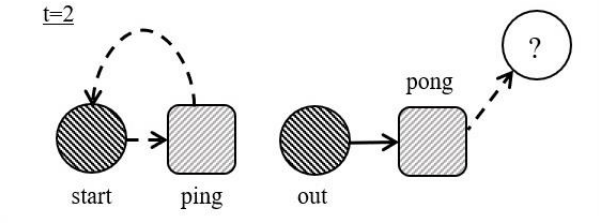

Рис. 1.18. Актор ping отправил сообщение-запрос out в актор pong

На рис. 1.19 показано состояние (t=3), в котором выполняется обработка только что доставленного сообщения *out* в актор *pong*. Также для простоты рассмотрения истории вычислений процесса «пинг-понг» считаем, что обработка сообщения заканчивается одновременно с отправкой *out* в качестве ответного сообщения в актор *ping*.

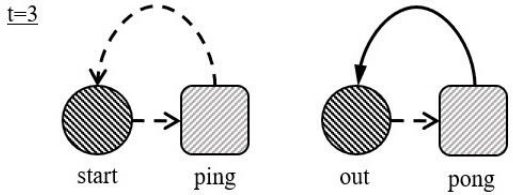

Рис. 1.19. Актор pong обрабатывает сообщение-запрос out

Состояние отправки сообщения-ответа из актора *pong* в актор *ping* в момент времени (t=4) показано на рис. 1.20.

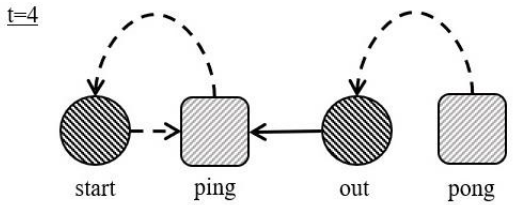

Рис. 1.20. Актор pong отправил сообщение-ответ out актор ping

Обработка доставленного сообщения-ответа выполняется в момент времени (t=5), что показано на рис. 1.21.

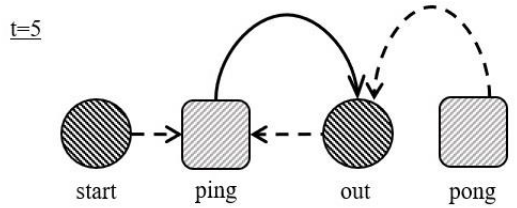

Рис. 1.21. Актор ping обрабатывает сообщение-ответ out от актора pong

Наконец, в момент времени (t=6) в состоянии процесса «пинг-понг» не остается ни одного активного сообщения или актора (см. рис. 1.22). Поэтому состояние больше не может изменяться во времени и является конечным состоянием вычислений.

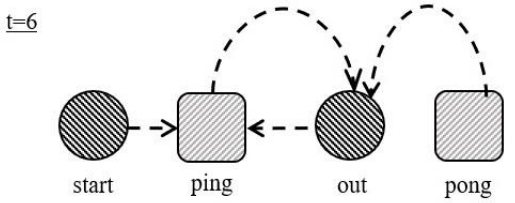

Рис. 1.22. Конечное состояние вычислений в модели «пинг-понг»

Заметим, что окончание вычислений акторного процесса можно задавать двумя способами. Конечное состояние процесса

может обнаружить некоторый актор, входящий в систему акторов процесса. При этом он вызовет специальную операцию *stop* (см. строку 20 на рис. 1.15). Другой вариант завершения показан на рис. 1.22 – это отсутствие объектов модели в активном состоянии. В алгоритме «пинг-понг» и в других алгоритмах в этом пособии условимся задавать окончание вычислений в явном виде при помощи операции *stop*.

Типы акторов, обработчики сообщений. Теперь применим введенную выше графическую нотацию для визуализации кода обработки сообщений в акторах. Это делается путем изображения типов акторов с обозначением обработчиков сообщений-запросов и сообщений-ответов, как показано на рис. 1.23.

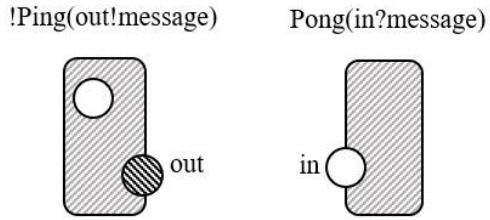

Рис. 1.23. Условные обозначения типов акторов Ping и Pong

Актор *ping* является экземпляром типа *Ping*, точнее – типа !*Ping*(*out*!*message*). Полное имя типа содержит признак (знак «!», записанный перед именем класса акторов) обработки начального сообщения и список всех обработчиков сообщений (знак «!» – для обработчика сообщения-ответа; знак «?» – для обработчика сообщения-запроса) и типов сообщений этих обработчиков. Типы сообщений позволяют контролировать корректность отправки сообщения в актор на стадии проектирования, если это позволяет язык программирования. Актор *pong* является экземпляром типа *Pong*(*in*?*message*).

После задания типа акторов модели указывается код всех обработчиков сообщений для введенных типов акторов. Код обработчиков сообщений акторов типа *Ping* показан на рис. 1.24. Код единственного обработчика сообщения типа *Pong* показан на рис. 1.25.

```
51      void start() {
52  /*$TET$Ping$start*/
53          cout << "Ping started.." << endl;
54          out.send();
55  /*$TET$*/
56      }
57
58      inline void on_out(message&m) {
59  /*$TET$Ping$out*/
60          cout << "Ping received message from Pong.." << endl <<endl;
61          stop();
62  /*$TET$*/
63      }
```

Рис. 1.24. Обработчик начального сообщения start и сообщения-ответа out для акторов типа Ping

```
93      inline void on_in(message&m) {
94  /*$TET$Pong$in*/
95          cout << "Pong received message from Ping.." << endl;
96          m.send();
97  /*$TET$*/
98      }
```

Рис. 1.25. Обработчик сообщений-запросов in акторов типа Pong

Таким образом визуализация типов акторов заменяет и позволяет более наглядно изобразить код обработки сообщений по сравнению с линейной конструкцией из операций if-then-else, фильтрующей контексты вызова обработчиков сообщений в соответствии с типами сообщений и акторов.

Система акторов. Еще один аспект описания акторных алгоритмов, часто требующий визуализации – структура системы акторов, показывающая акторы и их взаимодействие между собой. Большое число распределенных алгоритмов имеет неизменную в течении времени выполнения структуру системы акторов, поэтому для них уместно применение такой нотации. Система акторов процесса «пинг-понг» показана на рис. 1.26.

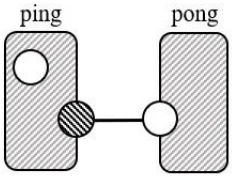

Рис. 1.26. Графическое представление системы акторов для модели «пинг-понг»

На рис. 1.26 показаны акторы *ping* и *pong*, а также связь между ними через сообщение *out,* содержащееся в акторе *ping*. Технически такая связь реализуется за счет того, что сообщение *out* внутри себя содержит поля данных для хранения этой связи. Кроме этого сообщение хранит в своем внутреннем состоянии положение относительно акторов, между которыми оно циркулирует. Благодаря этому свойству при отправке сообщения не нужно уточнять ни его актор-получатель, ни обработчик внутри этого актора (см. код отправки сообщения в строке 54 на рис. 1.24). О типе акторов можно судить по месту расположения и типу «кружков» на пиктограммах акторов.

Рис. 1.26 является визуализацией кода, создающего систему акторов алгоритма «пинг-понг». Этот код структурно относится к секции инициализации модели на рис. 1.2, строка 27.

```
[1]: #pragma cling add_include_path("../../lib/")
     #include "1-pingpong.hpp"

     engine eng;

     Ping ping(eng);
     Pong pong(eng);

     pong.in(ping.out);

     eng.start();

     cout << (eng.stopped() ? "Success!!!" : "Failure...");

     Ping started..
     Pong received message from Ping..
     Ping received message from Pong..

     Success!!!
```

Рис. 1.27. Алгоритмическое представление системы акторов для модели «пинг-понг», результат прогона программы

На рис. 1.26 показано создание объекта *eng*, отвечающего за выполнение системы акторов; создание экземпляров актор *ping* и *pong* соответствующих типов; связывание акторов между собой через сообщение *ping.out* (вызов *pong.in*(*ping.out*) показывает, что сообщение *ping.out* направится в обработчик *on_in*() актора *pong*, а ответное сообщение будет направлено в обработчик *on_out*() актора *ping*).

Далее производится запуск системы акторов, проверка корректности завершения (факта того, что некоторый актор обнаружил достижение конечного состояния алгоритма и просигнализировал об этом вызовом команды *stop*). Результат запуска модели (нижняя часть рис. 1.27) показывает корректность работы спроектированной системы акторов для процесса «пинг-понг».

В первой главе приведена графическая нотация описания акторных алгоритмов и рассмотрена её операционная семантика, основанная на распараллеливании последовательного алгоритма специальной структуры. На типовом примере «пинг-понг» показано применение графической нотации и метод её уточнения путем реализации обработчиков сообщений для комбинаций «тип актора – тип сообщения». Также рассмотрены графическая нотация для описания системы акторов и метод кодирования структуры системы акторов при помощи комплекта для разработки программного обеспечения Templet.

Код всех примеров учебного пособия является интерактивным и доступен для самостоятельных экспериментов через браузер. Полный код рассмотренного в главе примера «пинг-понг» содержится в файле ~/samples/booksmpl/1-pingpong.hpp и файле ~/samples/booksmpl/1-pingpong.ipynb репозитория проекта Templet.

2. ПРОГРАММИРОВАНИЕ ПРОЦЕССОВ С ТОПОЛОГИЕЙ «ЗВЕЗДА»

2.1. Параллельное суммирование числовых рядов

Виды процессов с топологией «звезда». Постановка задачи, метод решения. Шаблон протокола связи акторов, алгоритм протокола.

Виды процессов с топологией «звезда». Процессы с коммуникационной топологией «звезда» – это относительно простая и часто встречающаяся форма организации распределенных вычислений. Её особенностью является выделение управляющего (master) узла, в котором реализуется общий алгоритм управления вычислениями, и группы одинаковых рабочих (worker) узлов, связанных с управляющим узлом по сети. Функция рабочих узлов заключается в выполнении заданий, получаемых от управляющего узла. Задания обрабатываются автономно, для их выполнения рабочим узлам не требуется взаимодействовать с другими узлами в сети. Благодаря описанным ролям вычислительных узлов в организации вычислений, такой типовой процесс или паттерн называют «управляющий-рабочие» (master-worker).

Встречается два сценария применения типового процесса «управляющий-рабочие». В первом сценарии множество задач для вычислений рабочими процессами известно заранее. Во втором сценарии множество задач может пополняться новыми задачами во время вычислений в результате выполнения ранее поступивших задач. Первый сценарий часто называют «применить ко всем» (map), второй – «распределенный портфель задач» (bag of tasks). Второй

сценарий является универсальным, в то время как первый сценарий является более простым для изучения и реализации.

Начнем рассмотрение с простого варианта на хрестоматийном примере параллельного программирования – вычисление числа π методом параллельного подсчета суммы членов числового ряда.

Постановка задачи, метод решения. Требуется выполнить параллельный подсчет суммы числового ряда, задающего число π.

Начинать описание вычислительного процесса рекомендуется с обсуждения метода решения задачи. Для данной задачи зададимся числом акторов-рабочих *n*. Из исходного ряда сформируем *n* рядов по числу акторов-рабочих. Каждый *m*-ый актор-рабочий выполнит суммирование членов с номерами *m*, *m*+*n*, *m*+2*n*, *m*+3*n* и т.д. исходного ряда параллельно с другими акторами. Актор-управляющий сложит суммы частей исходного ряда, полученные от рабочих.

Построение шаблона протокола обмена сообщениями в системе акторов – это следующий шаг решения. Шаблон включает описание типов акторов и пример системы акторов. Шаблон протокола модели «управляющий-рабочие» показан на рис. 2.1.

Условные обозначения в правой части рис. 2.1 показывают, что управляющий актор типа *Master*(*in*?*range*) работает в роли сервера, обрабатывающего сообщения от клиентов-рабочих в архитектуре «клиент-сервер». Управляющий актор содержит обработчик сообщений-запросов *on_in*() для сообщений типа *range*, хранящих информацию о диапазоне суммируемых элементов исходного числового ряда.

Акторы-рабочие типа !*Worker*(*out*!*range*) исполняют роль клиентов в архитектуре «клиент-сервер». Они становятся активными в начале вычислений, обрабатывая начальное сообщение *start*. Каждый актор-рабочий содержит внутреннее сообщение *out* типа *range*, которое он отправляет в качестве запроса в управляющий актор, согласно показанной в левой части рис. 2.1 коммуникационной топо-

логии «звезда». Рис. 2.1 также показывает, что каждый актор типа *Worker* имеет обработчик сообщения-ответа *on_out*(), в который возвращается отправленное в качестве запроса сообщения *out*.

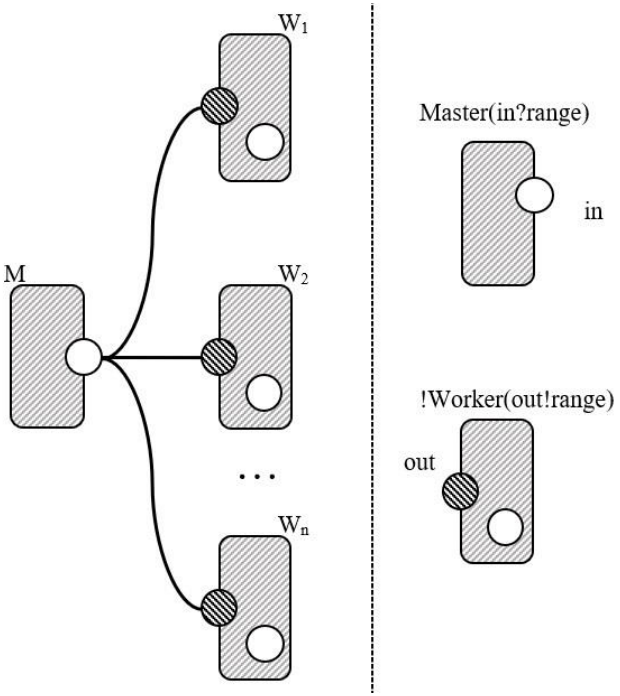

Рис. 2.1. Графическое представление модели «управляющий-рабочие» в виде системы акторов
(справа – условные обозначения типов акторов в системе)

Алгоритмы протокола. Следующий шаг построения спецификации процесса – это описание алгоритмов обработки сообщений в акторах и алгоритма построения системы акторов.

Рассмотрим вначале алгоритм работы управляющего актора типа *Master*. Начальное состояние актора определяется, как показано на рис. 2.2.

```
48      Master() :templet::actor(false)
49      {
50  /*$TET$Master$Master*/
51          current_range = 0;
52          num_of_ready_workers = 0;
53          PI = 0.0;
54  /*$TET$*/
55      }
```

Рис. 2.2. Начальное состояние актора типа Master

Управляющий актор хранит номер текущего поддиапазона суммируемых членов ряда, задающего число π, в переменной current_range. Чтобы определить момент завершения вычислений, он подсчитывает в переменной *num_of_ready_workers* число сообщений от рабочих акторов, в которые были переданы вычисленные частичные суммы поддиапазона членов ряда. В переменной *PI* он накапливает общую сумму членов ряда, добавляя к ней суммы поддиапазонов, полученные от акторов-рабочих.

Алгоритм обработки сообщений от акторов-рабочих показан на рис. 2.3. В обработчике отслеживаются два случая. В строках 65-68 показан случай, когда сообщение от актора-рабочего является запросом номера поддиапазона исходного ряда для суммирования. Этот номер формируется по текущему значению переменной *current_range*, записывается в переменную сообщения *range_id* и отправляется в ответном сообщении в актор-рабочий. Еще раз обратите внимание, что вся информация об адресате хранится в самом сообщении и настраивается, как показано ниже, при формировании системы акторов.

Второй случай – когда в сообщении от актора-рабочего пришло значение суммы поддиапазона числового ряда в переменной сообщения *result*. Обработка этого случая описывается в строках 69-73 на рис. 2.3. Она заключается в сложении суммы поддиапазона числового ряда с текущим значением суммы всего ряда *PI*, увеличении

счетчика числа завершивших свою часть вычислений рабочих процессов *num_of_ready_workers*, и в проверке выполнения условия завершения процесса вычисления числа π. Если условие выполнено, управляющий актор типа *Master* вызывает метод *stop*, останавливающий систему акторов.

```
63      inline void on_in(range&m) {
64  /*$TET$Master$in*/
65          if(m.range_id == NO_RANGE_ID){
66              m.range_id = current_range++;
67              m.send();
68          }
69          else{
70              PI = PI + m.result;
71              num_of_ready_workers++;
72              if(num_of_ready_workers == N) stop();
73          }
74  /*$TET$*/
75      }
```

Рис. 2.3. Обработчик сообщений-запросов in в акторе типа Master

Поведение акторов-рабочих типа *Worker* описывается двумя алгоритмами, показанными на рис. 2.4. В обработчике начального состояния *start* для номера поддиапазона ряда задается признак *NO_RANGE_ID*, по которому управляющий актор понимает назначение сообщения: запрос поддиапазона в начале вычислений или выдача результата. Далее сообщение отправляется в управляющий актор.

Обработчик сообщения *on_out* акторов типа *Worker* на рис. 2.4 содержит математическую часть алгоритма – распараллеливаемый код вычисления суммы членов ряда заданного поддиапазона *range_id*. Результат вычисления записывается в переменную *result* сообщения, отправляемого в строке 124 в управляющий актор.

Алгоритм построения системы акторов, соответствующий левой части рис. 2.1, показан на рис. 2.5. Здесь же ниже приведен результат прогона построенной программы.

```
109     void start() {
110 /*$TET$Worker$start*/
111         out.range_id = NO_RANGE_ID;
112         out.send();
113 /*$TET$*/
114     }
115
116     inline void on_out(range&m) {
117 /*$TET$Worker$out*/
118         double sum = 0.0;
119         for (long i=m.range_id; i < num_steps; i=i+N) {
120             double x = (i+0.5)*step;
121             sum += 4.0/(1.0+x*x);
122         }
123         m.result = sum*step;
124         m.send();
125 /*$TET$*/
126     }
```

Рис. 2.4. Обработчик начального сообщения start и сообщения-ответа out в акторах типа Worker

На рис. 2.5 интерес представляет процесс сборки системы акторов, в результате которого содержащиеся внутри объектов-рабочих *w*[*i*] сообщения *out* получают информацию об акторах, между которыми они передаются. Информацию об акторе-рабочем *w*[*i*] сообщение *w*[*i*].*out* получает по факту объявления типа !*Worker*(*out*!*range*) средствами системы Templet. Информацию об управляющем акторе *m* сообщения получают при вызове *m*.*in*(*w*[*i*].*out*) в теле цикла for на рис. 2.5. В итоге построено полное определение акторного представления алгоритма вычисления числа π. Вывод программы в нижней части рисунка говорит о корректности работы процесса.

Следует обратить внимание на отличие запрограммированного в терминах модели акторов вычислительного процесса от обычной последовательно-параллельной реализации вычислений, например, с использованием технологии OpenMP. В описанной реализации выбор поддиапазонов, вычисление их сумм, а также накопление итоговой суммы, могут происходить асинхронно и в произвольном

```
[1]: #pragma cling add_include_path("../../lib/")

     const int N = 10;
     const long num_steps = 100000;
     const double step = 1.0/(double) num_steps;

     #include "2-1-numseries.hpp"

     engine eng;

     Master m(eng);
     Worker w[N];

     for(int i=0; i<N; i++){
         w[i].engines(eng);
         m.in(w[i].out);
     }

     eng.start();

     cout << "PI = " << m.PI << endl << endl;

     cout << (eng.stopped() ? "Success!!!" : "Failure...");
     PI = 3.14159

     Success!!!
```

Рис. 2.5. Алгоритмическое представление системы акторов для модели «управляющий-рабочие», результат прогона программы

порядке. Например, может выполняться подсчет частичной суммы в одном акторе-рабочем, формирование запроса на выдачу поддиапазона в другом акторе-рабочем, сложение частичной суммы с текущим значением *PI* в управляющем акторе. Такая асинхронность и большое число степеней свободы при выборе продолжения вычислений является важным свойством распределенных алгоритмов. В случае задержки сообщения от одного актора, в распределенных алгоритмах должна быть возможность получить и обработать сообщения от другого актора. При проектировании процессов следует

максимально возможно избегать простоя при ожидании сообщений для высокой эффективности вычислений. Правильное применение модели акторов обеспечивает такую возможность «по построению». Негативной стороной этого свойства модели акторов является необходимость учета недетерминизма при анализе и проверке корректности реализации процесса.

Полный код рассмотренного примера содержится в файле ~/samples/booksmpl/2-1-numseries.hpp, а также в файле ~/samples/booksmpl/2-1-numseries.ipynb репозитория системы Templet.

2.2. Интегрирование методом адаптивной квадратуры

Постановка задачи. Алгоритм адаптивной квадратуры. Метод управления «портфелем задач», алгоритмы обработки сообщений в акторах.

Постановка задачи. Простой сценарий применения коммуникационной топологии «звезда», рассмотренный в п.2.1, пригоден для решения многих практических задач обработки данных, проверки гипотез о свойствах математических или реальных объектов по результатам измерений. Так распределенные вычисления в проектах популярной платформы BOINC (https://boinc.berkeley.edu/) часто выполняются именно с фиксированным набором заранее подготовленных при инициализации вычислений задач. При этом большое число акторов (порядка нескольких тысяч), хотя и может представлять некоторую сложность при реализации вычислений, не является ограничением модели акторов. Проблемой является возможность появление новых задач по мере вычисления задач, сформированных при инициализации расчета. Модификация логики взаимодействия акторов в модели рис. 2.1, учитывающая по-

явление новых задач, позволяет одновременно решить проблему управления большим числом задач, а также проблему балансировки вычислений задач различной длительности. Такая модификация упрощенной модели «применить ко всем» из п. 2.1 на базе топологии «звезда», как уже было отмечено, называется «распределенный портфель задач».

Алгоритм адаптивной квадратуры. Описание «распределенного портфеля задач» выполним на основе известного учебного алгоритма численного интегрирования [7] под названием «алгоритм адаптивной квадратуры». Суть алгоритма состоит в следующем. Пусть перед нами поставлена задача (в виде объекта, записанного в специальную очередь задач) вычислить интеграл некоторой функции *FUNCTION*(*x*) на интервале от *left* до *right*. Для получения значения этого интеграла в данной задаче вычислим аппроксимацию интеграла двумя способами. Первый способ аппроксимации – вычисление площади трапеции с основанием от *left* до *right* и высотой сторон *FUNCTION*(*left*), *FUNCTION*(*right*). Второй способ аппроксимации – вычисление суммы площадей двух трапеций: трапеции с основанием на отрезке [*left, mid*], высотой сторон *FUNCTION*(*left*), *FUNCTION*(*mid*) и трапеции с основанием на отрезке [*mid, right*], высотой сторон *FUNCTION*(*mid*), *FUNCTION*(*right*). Положение точки *mid* удобно взять посередине отрезка [*left, right*].

Если эти два способа вычисления аппроксимации интеграла функции *FUNCTION*(*x*) на отрезке [*left, right*] дают примерно одинаковые значения (различающиеся менее, чем на величину *epsilon*), то за итоговое значение интеграла функции на рассматриваемом отрезке [*left, right*] можно принять значение, полученное вторым способом аппроксимации. В противном случае на базе исходной задачи интегрирования функции на отрезке [*left, right*] нужно сформировать две новые задачи: задачу интегрирования функции на отрезке [*left, mid*] и задачу интегрирования функции на отрезке [*mid, right*].

Новые задачи решаются тем же способом, что и исходная. Аппроксимации интегралов, найденные в отдельных задачах, суммируются в специальной переменной *integral*. Из математических соображений о гладкости интегрируемой функции *FUNCTION*(*x*) следует, что описанный процесс рекурсивного порождения задач завершается для заданного *epsilon*.

Таким образом, в описанном алгоритме выполняется общее свойство программируемого процесса: при обработке некоторой задачи из заданного множества (то есть из «портфеля задач») может быть либо получен частичный результат, либо порождены новые задачи, пополняющие «портфель задач».

Метод управления «портфелем задач». Очевидно, что представленный выше алгоритм легко распараллеливается, так как присутствующее в «портфеле задач» в некоторый момент времени множество задач можно «раздать» на вычисление в акторы-рабочие, как это уже делалось в п. 2.1 при рассмотрении процесса «применить ко всем». Требуется лишь обеспечить непрерывность вычисления формируемых задач в акторах. Вычислив одну задачу, рабочий актор должен немедленно приступить к вычислению новой задачи, если задача имеется в портфеле задач управляющего актора. Для этого не потребуется изменять схему коммуникационного протокола процессов с топологией «звезда» на рис. 2.1 и соответствующий этой схеме код построения системы акторов на рис. 2.5. Требуется лишь переработать алгоритмы обработки сообщений для актора типа *Master* и акторов типа *Worker*.

Начальное состояние управляющего процесса типа *Master* показано на рис. 2.6. Кроме инициализации переменной *integral* в начале работы процесс содержит очередь задач *task_list*, содержащую единственную задачу вычисления интеграла на исходном интервале [*a*,*b*], а также пустой список объектов-сообщений от рабочих процессов *range_list*, в который будут помещаться запросы на

вычисление задачи интегрирования функции на подынтервалах внутри интервала [*a*,*b*]. Заметим, что эти два списка используются в различных алгоритмах, построенных по принципу «портфеля задач». В зависимости от конкретной решаемой задачи будут меняться лишь поля данных в объектах, содержащихся в списках *task_list* и *range_list*.

```
54      Master() :templet::actor(false)
55      {
56  /*$TET$Master$Master*/
57          integral = 0.0;
58  /*$TET$*/
59      }
```

Рис. 2.6. Начальное состояние актора типа Master

Рассмотрим этапы обработки сообщений-запросов в управляющем акторе типа *Master* в обработчике *on_in*. На первом этапе, показанном на рис. 2.7, производится обработка результата вычисления задачи, если он был передан в сообщении. В первом сообщении от акторов-рабочих, когда *m.is_first=true*, результат не передается, поэтому соответствующая секция кода в строках 71-77 пропускается.

```
67  inline void on_in(range&m) {
68  /*$TET$Master$in*/
69    if (m.is_first) m.is_first = false;
70    else {
71  /////// do it when the task is ready //////
72     if(m.area_computed) integral += m.area;
73     else{
74      task_list.push_back(range_task(m.left,m.mid,m.fleft,m.fmid,m.larea));
75      task_list.push_back(range_task(m.mid,m.right,m.fmid, m.fright,m.rarea));
76     }
77  ////////////////////////////////////////
78    }
79    range_list.push_back(&m);
```

Рис. 2.7. Обработчик сообщений-запросов in в акторе типа Master, обработка результата вычисления задачи

Согласно рассмотренному алгоритму адаптивной квадратуры, обработка задачи состоит в проверке признака *m.area_computed* вычисления аппроксимации интеграла на интервале [*m.left, m.right*]. Если аппроксимация вычислена с точностью *EPS*, её значение прибавляется к переменной *integral*. В противном случае, в строках 74 и 75, формируются две новые задачи для интервалов [*m.left, m.mid*] и [*m.mid, m.right*], которые помещаются в переменную *task_list* «портфеля задач». После того, как сообщение от актора-рабочего обработано таким образом, объект этого сообщения *m* помещается в список *range_list* в строке 79.

Далее в коде обработчика сообщений *on_in* следует цикл формирования и отправки ответных сообщений с задачами в акторы-рабочие, показанный на рис. 2.8. Цикл содержит две выделенных секции, которые связаны с особенностями решаемой задачи. Секция в строках 83-85 содержит проверку наличия задач в «портфеле задач». В данном алгоритме это проверка списка *task_list* на пустоту. Секция в строках 89-96 содержит код формирования задачи, то есть заполнение полей данных объектов-сообщений от акторов-рабочих. Эти сообщения поступили ранее (возможно в момент вызова данного обработчика) и были сохранены в строке 79 на рис. 2.7. Перед заполнением сообщение извлекается из списка *range_list* в строках 87-88, а после заполнения отправляется в «свой» актор-рабочий в строке 97 на рис. 2.8.

После кода формирования задач следует проверка условия завершения вычислений. Вычисления завершены, если после попытки формирования задач в цикле while в строках 82-98 в списке сообщений на выдачу задач содержится столько сообщений, сколько имеется акторов-рабочих. Это означает, что в этом состоянии только актор типа *Master* находится в активном состоянии, и не имеется активных сообщений на доставке. В этом случае в явном виде останавливаем систему акторов путем вызова метода *stop* в строке 99 на рис. 2.8.

```
82          while (!range_list.empty() &&
83  //// check if we have something to do /////
84                 !task_list.empty()
85  ////////////////////////////////////////////
86                 ) {
87                 range* r = range_list.front();
88                 range_list.pop_front();
89  ////////// form a new task ////////////////
90                 range_task t = task_list.front();
91                 task_list.pop_front();
92
93                 r->left = t.left; r->right = t.right;
94                 r->fleft = t.fleft; r->fright = t.fright;
95                 r->lrarea = t.lrarea;
96  ////////////////////////////////////////////
97                 r->send();
98          }
99          if (range_list.size() == N)*stop();
100 /*$TET$*/
101     }
```

Рис. 2.8. Обработчик сообщений-запросов in в акторе типа Master, выдача следующих задач, проверка условия останова

Код обработчиков сообщений акторов типа *Worker* показан на рис. 2.9. Обработка начального сообщения *start* включает установку признака *is_first* у сообщения *out* и отправку его в управляющий процесс.

Обработка ответного сообщения от управляющего актора *on_out* содержит код вычисления задачи, специфичный для программируемого алгоритма. В данном примере в строках 145-150 содержится код вычисления двух аппроксимаций интеграла функции *FUNCTION*(*x*) на отрезке [*m.left, m.right*] и формирование признака достаточности полученной аппроксимации *m.area_computed*, который будет использован далее в управляющем процессе при принятии решения о добавлении новой задачи в «портфель задач» (см. строку 72 на рис. 2.7).

```
135     void start() {
136 /*$TET$Worker$start*/
137         out.is_first = true;
138         out.send();
139 /*$TET$*/
140     }
141 
142     inline void on_out(range&m) {
143 /*$TET$Worker$out*/
144         //////// do the task  ////////
145         m.mid = (m.left + m.right)/2;
146         m.fmid = FUNCTION(m.mid);
147         m.larea = (m.fleft + m.fmid)*(m.mid - m.left)/2.0;
148         m.rarea = (m.fmid + m.fright)*(m.right - m.mid)/2.0;
149         m.area = m.larea + m.rarea;
150         m.area_computed = abs(m.area - m.lrarea) < epsilon;
151         ////////////////////////////////
152 
153         std::cout << "range[" << m.left << ";" << m.right << "]";
154         if(m.area_computed) std::cout << " computed: " << m.area;
155         else std::cout << " splited ";
156         std::cout << " by worker # "<< ID << std::endl;
157 
158         out.send();
159 /*$TET$*/
160     }
```

Рис. 2.9. Обработчик начального сообщения start,
обработчик сообщения-ответа out в акторах типа Worker

В построенном таким образом процессе обработка и выдача новых задач происходят параллельно и асинхронно. Задержка при вычислении некоторой задачи в акторе-рабочем не влияет на возможность других акторов-рабочих получать новые задачи на обработку. Количество задач, готовых к выполнению (в списке *task_list*), может многократно превышать количество акторов-рабочих. Для реализации процесса «применить ко всем» требуется лишь заполнить список задач *task_list* в начале вычислений. При этом код секции обработки результата только что завершившейся задачи (строки 71-77 на рис. 2.7) не будет включать добавление новых задач в список *task_list*.

Полный код рассмотренного примера содержится в файле ~/samples/booksmpl/2-2-adaptquad.hpp, а также в файле ~/samples/booksmpl/2-2-adaptquad.ipynb репозитория системы Templet.

2.3. Практические задания

1. Исследование схемы «применить ко всем». На базе описанного в п. 2.1 алгоритма проведите исследование накладных расходов его реализации при параллельном или распределенном выполнении в системе Templet. Определите, какие параметры реализации оказывают влияние на процент накладных расходов. Выполните перенос кода акторного алгоритма п. 2.1 из системы Templet в другую (по желанию) систему программирования, использующую акторы. Проведите аналогичное исследование накладных расходов. Сформулируйте выводы по результатам проведенного исследования.

2. Исследование схемы «портфель задач» без динамического пополнения списка задач. Реализуйте вычисление числа π на основе алгоритма «портфель задач», описанного в п. 2.2. Адаптируйте построенный таким образом алгоритм для параллельного или распределенного выполнения в системе Templet. Также адаптируйте алгоритм из п. 2.1. Выполните сравнительное исследование накладных расходов на распараллеливание у полученных реализаций алгоритмов вычисления числа π. Сформулируйте выводы по результатам исследования.

3. Исследование схемы «портфель задач» с динамическим пополнением списка задач. Методом повышения эффективности вычислений по схеме «портфель задач» является объединение нескольких задач в пакеты (parceling). Выполните переработку кода алгоритма п. 2.2 таким образом, чтобы на вычисление в один рабочий процесс можно было передать пакет задач заданного размера.

Далее адаптируйте полученную модификацию алгоритма и исходный алгоритм п. 2.2 для параллельного или распределенного выполнения в системе Templet. Выполните сравнительное исследование накладных расходов на распараллеливание у полученных реализаций алгоритмов интегрирования методом адаптивной квадратуры. Сформулируйте выводы по результатам исследования.

3. ПРОГРАММИРОВАНИЕ ПРОЦЕССОВ С ЛИНЕЙНОЙ ТОПОЛОГИЕЙ

3.1. Умножение матриц на «кольце процессоров»

Постановка задачи. Метод умножения матриц на «кольце процессоров». Алгоритмы обработки сообщений в акторах. Алгоритм построения системы акторов.

Постановка задачи. Известно много различных алгоритмов умножения матриц, в том числе параллельных. Некоторые алгоритмы позволяют ускорить вычисления за счет снижения числа операций, по сравнению с умножением «по определению». Другие алгоритмы используют особенности архитектуры вычислительных устройств, например, наличие нескольких ядер или процессоров, векторизуемых операций, особой организации памяти вычислительной системы. Рассматриваемый в п. 3.1 алгоритм ориентирован на вычисление произведения матриц на многопроцессорной распределенной системе. Он предусматривает декомпозицию исходных матриц и результата на части и распределение этих частей по различным процессорам, чтобы не дублировать элементы матриц и минимизировать перемещение элементов матриц между процессорами по сети. В литературе алгоритм встречается под названием «ленточный алгоритм умножения матриц» [8].

Базовым методом декомпозиции данных при их обработке в распределенных вычислительных системах является линейная или одномерная декомпозиция. При такой декомпозиции вычислительные узлы (процессоры) образуют линейную коммуникационную

структуру, в которой основные взаимодействия каждого узла по сети происходят между ним и двумя соседними узлами: левым и правым соседом. Узел отвечает за обработку определенной части данных задачи в зависимости от своего порядкового номера в линейной структуре. В п. 3.1 рассматривается применение замкнутой линейной коммуникационной топологии процессоров «кольцо». Далее в п. 3.2 показано использование незамкнутой линейной коммуникационной топологии «цепь».

Метод умножения матриц на «кольце процессоров». Система акторов позволяет наглядно представить работу «кольца процессоров» для умножения матриц. Актор представляет узел вычислительной сети (процессор), а внутреннее состояние актора является частью данных в декомпозиции исходной задачи. На рис. 3.1 показано кольцо из акторов l_1, l_2,…l_N , представляющее структуру связанных по сети процессоров. Кроме них показан вспомогательный актор *stopper*, выполняющий функцию определителя момента завершения вычислений. Также актор *stopper* имитирует сборку результата вычислений.

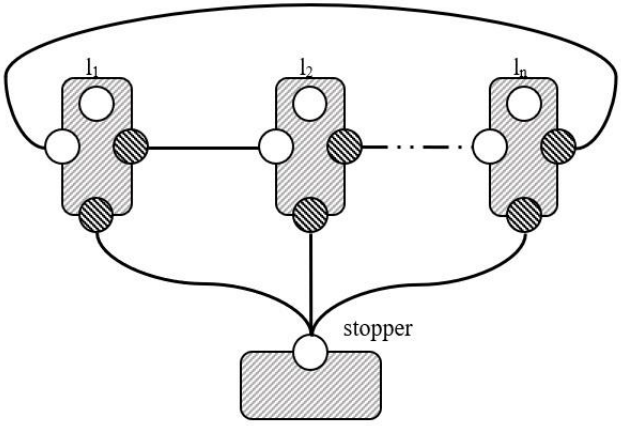

Рис. 3.1. Графическое представление модели «кольцо процессоров» в виде системы акторов

Для простоты условимся, что перемножаемые матрицы имеют размеры $N \times N$. Выполним декомпозицию частей матриц $A \times B = C$, участвующих в вычислениях, следующим образом. Строки матрицы A от первой до N-ной и строки матрицы C от первой до N-ной разместим на акторах $l_1, l_2, \ldots l_N$, показанных на рис. 3.1. Столбцы матрицы B можно распределить произвольным образом между акторами так, чтобы в каждом акторе $l_1, l_2, \ldots l_N$ оказалось по одному столбцу матрицы B.

В каждом из акторов $l_1, l_2, \ldots l_N$ содержится одна строка матрицы A с номером, соответствующим номеру актора в цепи; одна строка матрицы C с номером, соответствующим номеру актора в цепи; один столбец матрицы B с произвольным номером. В таком состоянии каждый актор может выполнить вычисление своего элемента строки C с номером столбца B. Вычисление состоит в получении скалярного произведения строки из A и столбца из B. После вычисления столбец матрицы B можно передать в соседний актор справа. Далее рассматриваемый актор получит другой столбец матрицы B от актора слева и повторит операцию вычисления элемента своей строки матрицы C. Актор может обнаружить окончание вычисления своей строки из C путем учета обработанных столбцов матрицы B. Когда строка из матрицы C вычислена, актор отправляет её в актор *stopper*. Актор *stopper* в свою очередь ведет подсчет полученных от акторов $l_1, l_2, \ldots l_N$ строк матрицы C. Когда все строки вычислены выдается сигнал остановки вычислений *stop*.

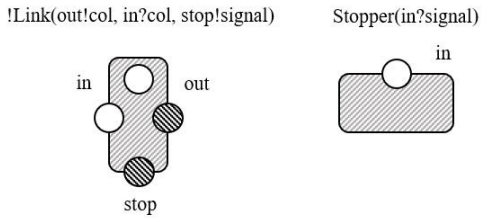

Рис. 3.2. Графическое представление типов акторов в модели «кольцо процессоров»

Алгоритмы обработки сообщений в акторах. Типы акторов модели «кольцо процессоров» показаны на рис. 3.2. Тип *Link* описывает протокол обработки сообщений акторов l_1, l_2,…l_N . Акторы этого типа получают начальное сообщение, в обработчике *start* которого просто запускается обработчик сообщения-запроса *on_in* (рис. 3.3). Также обработчик *on_in* запускается при получении столбца матрицы *B* в процессе циркуляции этих столбцов по цепочке акторов l_1, l_2,…l_N.

```
63      void start() {
64  /*$TET$Link$start*/
65          on_in(out);
66  /*$TET$*/
67      }
68
69      inline void on_in(column&m) {
70  /*$TET$Link$in*/
71          if(num_of_ready_columns!=0 &&  &m == &out) return;
72
73          int j = m.j;
74          for(int k = 0; k < N; k++) C[i][j] += A[i][k] * B[k][j];
75
76          next->in(m); m.send();
77          if(++num_of_ready_columns == N) stop.send();
78  /*$TET$*/
79      }
```

Рис. 3.3. Обработчик начального сообщения start и обработчик сообщения-ответа out акторов типа Link

Рассмотрим части кода обработчика *on_in* в акторах типа *Link*. В строке 71 на рис. 3.3 отслеживается ситуация, когда столбец матрицы *B*, находящийся в акторе в начале вычислений, сделал круг по кольцу на рис. 3.1 и вернулся в свой актор. Номер этого столбца записан в сообщении *out*. В этом случае обработка сообщения *out* прерывается, так как столбец был обработан в начале вычислений (при вызове *on_in* из *start*).

В строках 73-74 на рис. 3.3 содержится код вычисления скалярного произведения строки матрицы *A* и столбца матрицы *B*. Заметим, что при моделировании вычислительного процесса реальное расположение элементов матриц не принципиально. В примере они располагаются в разделяемой всеми акторами памяти. Акторы типа *Link* и сообщения *out* типа *column* содержат лишь номера своих строк или столбцов.

В строке 76 на рис. 3.3 выполняется отправка сообщения, содержащего номер только что обработанного столбца матрицы *B* в соседний справа актор. Этот код имеет следующие особенности. До сих пор использовалась семантика взаимодействия акторов путем отправки сообщения-запроса и возвращения того же сообщения в качестве ответа либо однократной отправки сообщения. В описываемой модели взаимодействия акторов объект-сообщение оказывается поочередно сразу во всех акторах цепочки l_1, l_2,…l_N. Такое поведение получается путем изменения привязки сообщения *m* вызовом *next->in*(*m*), где *next* – это указатель на актор справа от рассматриваемого актора. После изменения привязки сообщение отправляется в актор обычным вызовом *m.send*(). Такой прием можно использовать, если порядок поступления сообщений в акторы не принципиален, как в данном случае. Порядок может нарушаться, так как в силу недетерминизма модели акторов сообщения, двигаясь по цепочке акторов, могут произвольно обгонять друг друга.

В строке 77 на рис. 3.3 определяется момент завершения вычисления строки матрицы *C* в акторе, после чего сообщение *stop* отправляется в актор *stopper*. Актор *stopper*, в свою очередь, определяет момент останова вычислений путем подсчета всех сообщений *stop*, отправленных из акторов цепочки l_1, l_2,…l_N в своём обработчике *in*, как показано на рис. 3.4.

```
125     inline void on_in(message&m) {
126 /*$TET$Stopper$in*/
127         if(++num_of_ready_rows == N) stop();
128 /*$TET$*/
129     }
```

Рис. 3.4. Обработчик сообщений-запросов in в акторе типа Stopper

Алгоритм построения системы акторов модели «кольцо процессоров» показан на рис. 3.5. Вначале создаются объект *stopper* и *N* объектов *l*[ ] типа *Link*. Затем в цикле формируется топология связей между акторами, показанная на рис. 3.1. Дополнительно акторы *l*[ ] подключаются к механизму выполнения акторной модели в объекте *eng*.

Для задания кольцевой связи в специально введенное поле *next* для каждого актора *l*[ ] типа *Link* записывается адрес его соседа по кольцу. Этот адрес затем используется для отправки сообщений со столбцами матрицы *B* (строка 76 на рис. 3.3). Также в цикле назначаются номера строк и столбцов, которые связываются с акторами и сообщениями перед началом вычислений.

```
[1]: /* - init code skipped - */
     engine eng;

     Stopper stopper(eng);
     Link l[N];

     for(int i=0; i<N; i++){
         l[i].engines(eng); stopper.in(l[i].stop);
         l[i].next = &l[(i+1)%N];

         l[i].i = i; l[i].out.j = i;
     }

     eng.start();
     /* - output code skipped - */
```

Рис. 3.5. Алгоритмическое представление системы акторов для модели «кольцо процессоров»

Полный код рассмотренного примера содержится в файле ~/samples/booksmpl/3-1-ringmult.hpp, а также в файле ~/samples/booksmpl/3-1-ringmult.ipynb репозитория системы Templet.

3.2. «Волновой процесс» по алгоритму Гаусса-Зейделя

Постановка задачи. Метод распараллеливания исходного алгоритма. Алгоритмы обработки сообщений в акторах. Алгоритм построения системы акторов. Особенности протекания вычислительного процесса.

Постановка задачи. Помимо ранее рассмотренных примеров акторных моделей процессов из областей численного интегрирования и линейной алгебры, большое прикладное значение имеют технологии параллельного программирования численных методов решения уравнений в частных производных. При помощи уравнений в частных производных описываются разнообразные физические процессы в непрерывных средах. На примере процесса теплопроводности покажем, как модель акторов может использоваться для их распараллеливания.

Алгоритмическим содержанием решения уравнений в частных производных являются итерационные методы решения систем линейных уравнений большой размерности с матрицами специального вида. Компоненты вектора неизвестных из этих уравнений отображаются на точки области пространства (сеточная область) и содержат значения некоторых физических параметров в этих точках. В нашем примере это могут быть значения температуры, содержащиеся в массиве *area*[ ] в алгоритме на рис. 3.6.

Алгоритм на рис. 3.6 содержит два цикла. Внешний цикл в строке 1 перечисляет отсчеты модельного времени, внутренний

цикл в строке 2 перечисляет пространственные отсчеты. Другие явные методы имеют похожую структуру алгоритмов. Явными методы называются потому, что новое значение в точке *area*[*offset*] в строке 3 вычисляется по «явной формуле» как функция (в данном примере это функция вычисления среднего) от значений в соседних точках сеточной области. Итерационный метод, в котором значения переменной на текущем шаге явно зависят от части значений на этом же шаге (в примере это *area*[*offset*-1]) и части значений на предыдущем временном шаге (в примере это *area*[*offset*+1]), обычно называют методом Гаусса-Зейделя.

```
1 for(int wave_num = 1; wave_num <= T; wave_num++)
2     for(int offset = 1; offset < N-1; offset++){
3         area[offset] = 0.5*(area[offset-1]+area[offset+1]);
```

Рис. 3.6. Распараллеливаемый последовательный алгоритм

Метод распараллеливания исходного алгоритма. Распараллеливание алгоритма рис. 3.6 основано на разбиении сеточной области *area*[ ] на части; помещении частей во внутреннее состояние акторов; организации такого обмена сообщениями между акторами, чтобы воспроизвести информационные зависимости между значениями переменных во время выполнения исходного алгоритма; обеспечении возможности параллельного выполнения.

Так как область *area*[ ] является одномерной, система акторов, соответствующая её декомпозиции, будет иметь вид «цепочки процессоров», показанной на рис. 3.7.

Распределение переменных массива *area*[ ] по акторам цепочки производится следующим образом. Акторы *b* и *e* выполняют служебные функции, они начинают и замыкают цепочку. Акторы S_i выполняют операцию из строки 3 исходного алгоритма на рис. 3.6. При этом актор S_1 на рис. 3.6 пересчитывает значение *area*[1], актор

S_i пересчитывает значение *area*[i], последний актор S_n пересчитывает значение *area*[*N*-2] массива *area*[ ], размерность которого согласно алгоритму на рис. 3.6 равна [0..*N*-1].

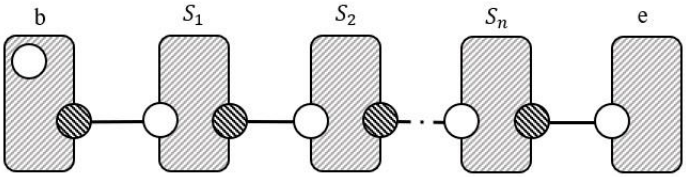

Рис. 3.7. Графическое представление модели «цепочка процессоров» в виде системы акторов

Алгоритмы обработки сообщений в акторах. Далее определим типы акторов в системе рис. 3.7 и обработчики сообщений для них. Типы акторов в модели «цепочка процессоров» показаны на рис. 3.8. Параллельный процесс составляют пять (по числу кружков) последовательных алгоритмов обработки сообщений.

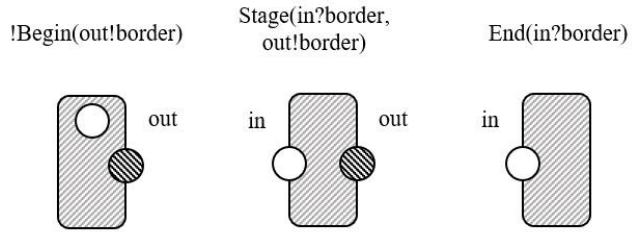

Рис. 3.8. Графическое представление типов акторов в модели «цепочка процессоров»

Актор типа *Begin* начинает обработку (см. рис. 3.9). В обработчике сообщения *start* вызывается обработчик *on_out* для имитации поступления сообщения *out* перед запуском алгоритма. В обработчике *on_out* в строке 64 на рис. 3.9 ведется подсчет числа запущен-

ных итераций по времени в переменной *wave_num*. Если это число не превышает *T*, запускается новая итерация.

```
56      void start() {
57  /*$TET$Begin$start*/
58          on_out(out);
59  /*$TET$*/
60      }
61
62      inline void on_out(border&m) {
63  /*$TET$Begin$out*/
64          if(++wave_num <= T) out.send();
65  /*$TET$*/
66      }
```

Рис. 3.9. Обработчик начального сообщения start
и сообщения-ответа out в акторе типа Begin

Основные вычисления выполняются в акторах типа *Stage*. Для управления ими применен следующий прием на рис. 3.10, рис 3.11. Начало вычислений в акторах *Stage* связано не с событиями поступления сообщений от соседей, а с наступлением состояния, в котором в актор уже поступили сообщения от правого и от левого соседа в цепочке. Обработка событий в акторах типа *Stage* программируется так. Вместо выполнения обработки сообщений в методах *on_in* и *on_out*, только запоминается поступившее сообщение (строка 103 на рис. 3.10). Фактическая обработка передается в метод *calculate* (строка 104 и 110 на рис. 3.10).

Метод *calculate* у акторов типа *Stage* на рис. 3.11 сначала (строка 119) выполняет проверку предусловия выполнения пересчета значения *area*[*offset*]. Если предусловие пересчета выполняется, то вычисляется новое значение *area*[*offset*] (строка 120) и выполняется отправка ответных сообщений в соседние акторы (строка 121). Иначе, если доставлено только одно сообщение от соседа слева *_in* или от соседа справа *out*, обработка завершается. Для проверки

```
101     inline void on_in(border&m) {
102 /*$TET$Stage$in*/
103         _in = &m;
104         calculate();
105 /*$TET$*/
106     }
107
108     inline void on_out(border&m) {
109 /*$TET$Stage$out*/
110         calculate();
111 /*$TET$*/
112     }
```

Рис. 3.10. Обработчики сообщения-запроса in
и сообщения-ответа outв акторах типа Stage

факта доставки сообщений в актор в алгоритме *calculate* используется примитив модели акторов *access*, логика работы которого рассматривалась в п.1.1 на рис. 1.5. Это типичное применение примитива *access* в используемом варианте модели акторов.

```
118     void calculate(){
119         if(access(_in) && access(out)){
120             area[offset] = 0.5*(area[offset-1]+area[offset+1]);
121             _in->send(); out.send();
122         }
123     }
```

Рис. 3.11. Действие calculate, запускаемое в акторах типа Stage

Завершает рассмотрение алгоритмов обработки сообщений метод *on_in* актора типа *End*, показанный на рис. 3.12. Этот обработчик выполняет две функции: фиксирует состояние завершения вычислений и уведомляет об этом систему выполнения вызовом *stop* в строке 154; отправляет ответное сообщение в соседний актор типа *Stage* в строке 155.

```
152     inline void on_in(border&m) {
153 /*$TET$End$in*/
154         if(++wave_num == T) stop();
155         else m.send();
156 /*$TET$*/
157     }
```

Рис. 3.12. Обработчик сообщения-запроса in в акторе типа End

Алгоритм построения системы акторов. Система акторов вычислительного процесса для модели «цепочка процессоров» показана на рис. 3.13. Сборка цепочки акторов начинается с записи сообщения *b.out* во вспомогательную переменную *last_out* и подключения выхода последнего актора типа *Stage* из массива акторов *s*[*N*] ко входу актора *e.in*(). Далее в цикле for устанавливаются все связи акторов *s*[*i*] слева от пиктограммы соответствующих акторов на рис. 3.7. Дополнительно акторы *s*[*i*] привязываются к объекту *eng* механизма выполнения модели и определяется индекс *offset* переменной в массиве *area*[ ], значение которой будет вычислять актор *s*[*i*].

```
[1]: /* - init code skipped - */
     engine eng;

     Begin b(eng);
     Stage s[N];
     End   e(eng);

     area[0] = 100.0;

     border* last_out = &b.out;
     e.in(s[N-1].out);

     for(int i=0; i<N; i++){
         s[i].engines(eng);  s[i].offset = i+1;
         s[i].in(*last_out); last_out = &s[i].out;
     }

     eng.start();
     /* - output code skipped - */
```

Рис. 3.13. Алгоритмическое представление системы акторов для модели «цепочка процессоров»

Особенности протекания вычислительного процесса. В заключении рассмотрим, как выполняется вычислительный процесс в описанной «цепочке процессоров» и какие действия процесса могут потенциально выполняться одновременно.

Процесс по приведенному описанию называют волновым, так как его протекание напоминает распространение волн по цепочке акторов. Каждая волна соответствует выполнению итерации по времени в строке 1 на рис. 3.6 кода исходного алгоритма, об этом говорит и название переменной цикла *wave_num*. Если такая волна сначала проходит от актора *b* до актора *e* и только потом запускается следующая волна, то вычислительный процесс в системе акторов в точности соответствует процессу в исходном последовательном алгоритме рис. 3.6. Однако в акторной версии алгоритма следующая волна запускается сразу же, как только гребень текущей волны отойдет на достаточное расстояние от актора *b*. Поэтому по цепочке акторов может одновременно двигаться несколько волн, параллельно выполняя вычисления.

Рассмотрим по алгоритму обработки сообщений в акторах типа *Stage*, каков минимально допустимый интервал между волнами. Каждый актор типа *Stage* начинает обработку, когда получил сообщения от своего правого и левого соседа. Этот актор является фронтом волны. Так как эти же сообщения используются при проверке предусловия начала вычислений у его акторов-соседей (в строке 119 на рис. 3.11), соседи не могут начать вычисления и изменить свои значения массива *area*[ ]. То есть следующий фронт волны находится минимум через один актор слева и справа от рассматриваемого актора. Максимальное число активных акторов *Stage* не превышает половину от их общего количества.

Заметим, что акторный процесс не содержит никаких дополнительных ограничений, кроме рассмотренных, на порядок следования волн. Это свойство является важным при реализации вычисли-

тельного процесса в распределенных вычислительных системах, где синхронное выполнение параллельных шагов алгоритма может оказаться менее эффективным, чем описанное асинхронное.

Исходя из приведенных выше рассуждений, можно утверждать, что по построению изолированное состояние акторов типа *Stage* составляют переменные *area*[*offset*-1], *area*[*offset*], *area*[*offset*+1] в коде метода *calculate* на рис. 3.11 в строке 120.

Полный код рассмотренного примера содержится в файле ~/samples/booksmpl/3-2-gausswave.hpp, а также в файле ~/samples/booksmpl/3-2-gausswave.ipynb репозитория системы Templet.

3.3. Практические задания

1. Исследование умножения матриц на «кольце процессоров», передача столбцов. Переделайте алгоритм п. 3.1 таким образом, чтобы в каждом сообщении по кольцу могло передаваться несколько столбцов матрицы *B*. Адаптируйте построенный таким образом алгоритм для параллельного или распределенного выполнения в системе Templet. Проведите исследование влияния количества передаваемых в одном сообщении столбцов на эффективность вычислений. Сформулируйте выводы по результатам исследования.

2. Исследование умножения матриц на «кольце процессоров», передача строк. Переделайте алгоритм п. 3.1 таким образом, чтобы в сообщениях, циркулирующих по «кольцу процессоров» передавались строки матрицы *B*. Адаптируйте построенный алгоритм и исходный алгоритм п. 3.1 для параллельного или распределенного выполнения в системе Templet. Проведите сравнительное исследование эффективности построенных программных реализаций. Сформулируйте выводы по результатам исследования.

3. Исследование «волнового процесса», двумерная область. Переделайте алгоритм п. 3.2 таким образом, чтобы расчет поля температуры выполнялся на двумерной сеточной области, с сохранением одномерной декомпозиции области при распараллеливании. Адаптируйте построенный алгоритм для параллельного или распределенного выполнения в системе Templet. Проведите сравнительное исследование влияния геометрических параметров области на эффективность вычислений. Сформулируйте выводы по результатам исследования.

4. Применение «волнового процесса» для сортировки чисел. Переделайте алгоритм работы акторов п. 3.2 так, чтобы они выполняли сортировку чисел. Используйте следующий принцип. В сообщениях по цепочке акторов от *b* до *e* передаются произвольные числа, формируемые актором *b*. Когда актор S_i получает число от своего соседа слева в первый раз, он запоминает это число. Когда число поступает во второй и следующие разы, актор сравнивает только что полученное число с тем, которое он запомнил. Если запомненное число больше, оно передается дальше актору S_{i+1}. Если меньше, то число запоминается в акторе, а ранее запомненное число передается в актор S_{i+1}. Вычисления останавливаются, когда число поступит в актор *e*. Адаптируйте построенный алгоритм для параллельного или распределенного выполнения в системе Templet. Проведите сравнительное исследование эффективности построенной реализации сортировки при параллельном и последовательном выполнении. Сформулируйте выводы по результатам исследования.

5. Применение «волнового процесса» для нахождения простых чисел. Переделайте алгоритм работы акторов п. 3.2 так, чтобы они выполняли поиск простых чисел. Используйте следующий принцип. Актор *b* последовательно генерирует нечетные числа и пе-

редает их по цепочке. Когда актор S_i получает число от своего соседа слева в первый раз, он запоминает это число. Оно будет являться простым. Когда число поступает во второй и следующие разы, актор проверяет делимость только что полученного числа на то, которое он запомнил. Если число не делится нацело, оно передается дальше актору S_{i+1}. Если делится, то отбрасывается как составное. Вычисления останавливаются, когда число поступит в актор *e*. Адаптируйте построенный алгоритм для параллельного или распределенного выполнения в системе Templet. Проведите сравнительное исследование эффективности построенной реализации поиска простых чисел при параллельном и последовательном выполнении. Сформулируйте выводы по результатам исследования.

4. ДИНАМИЧЕСКОЕ РАСПАРАЛЛЕЛИВАНИЕ НА ДАГАХ ПРОЦЕССОВ

4.1. Поиск ближайшего соседа через паросочетания

Постановка задачи. Метод поиска ближайшего соседа перебором паросочетаний. Типы акторов-узлов дага потока работ. Алгоритм синтеза графа потока работ типа «круговой турнир».

Постановка задачи. Еще одной важной, помимо рассмотренных в главах 2 и 3, областью применения распределенных и параллельных вычислений является управление потоком работ (workflow), в частности, управление потоком работ в области научных вычислений (scientific workflow).

Сложные ресурсоемкие вычисления могут быть организованы в виде упорядоченного набора задач – автономных операций с заданными входными и выходными данными. Такое разделение на задачи вызвано тем, что их параллельное выполнение на отдельных узлах дает ускорение вычислений. Разделение на задачи также необходимо, если требуется специально сконфигурированное программное или аппаратное обеспечение на конкретном узле для определенной задачи обработки данных. Помимо передачи данных на специализированный узел может возникнуть необходимость выполнения некоторых этапов вычислений на сайте, где хранятся данные. Такая многозадачная архитектура приложения требует специального подхода к описанию и реализации потоков работ.

Традиционной формой программирования потоков работ является использование дагов (ориентированных ациклических графов)

для описания зависимостей между задачами, составляющими потоки работ. Модель акторов, благодаря своим свойствам, позволяет наглядно и удобно программировать даги потоков работ. Кроме этого, использование модели акторов предпочтительно в случае программирования потоков работ, описываемых сложными автоматически генерируемыми графами. Также модель акторов может быть адаптирована для автоматического распараллеливания потоков работ, заданных в форме ациклических последовательностей запуска задач. Пример программирования потока работ, управляющего попарной обработкой массивов произвольных данных, рассматриваемый далее в п. 4.1, включает оба случая применения модели акторов.

Метод поиска ближайшего соседа перебором паросочетаний. Пусть имеется массив из N точек на прямой с произвольно заданными координатами. Требуется для каждой точки из этого массива определить точку, отстоящую от неё на минимальное расстояние, называемую ближайшим соседом.

Очевидным способом поиска является перебор в N параллельных процессах всех возможных соседей каждой точки путем сопоставления $N(N\text{-}1)$ пар. Если обрабатываются не точки на прямой, а сложные объекты или группы объектов («точки данных»), каждое сопоставление потребует предварительной передачи некоторого объема данных в обрабатывающий узел. Уменьшить количество сопоставлений, следовательно и передач данных, можно, если исключить двойные сопоставления «точек данных» (например, (1,2) и (2,1)). Но в этом случае возникает ограничение на одновременную обработку двух пар «точек данных», содержащих общую точку, (например, (1,2) и (2,3)) из-за того, что нельзя одновременно модифицировать переменную ближайшего соседа для общей точки (2).

Задача попарной параллельной обработки, где каждая возможная пара рассматривается один раз, похожа на задачу организации «кругового турнира» N команд так, чтобы провести игры между

участниками турнира за минимальное число этапов. На каждом этапе играются игры (ведется обработка пары «точек данных») таким образом, чтобы каждая «точка данных» входила только в одну пару этапа. Система акторов, иллюстрирующая обработку массива из четырёх «точек данных», показана на рис. 4.1.

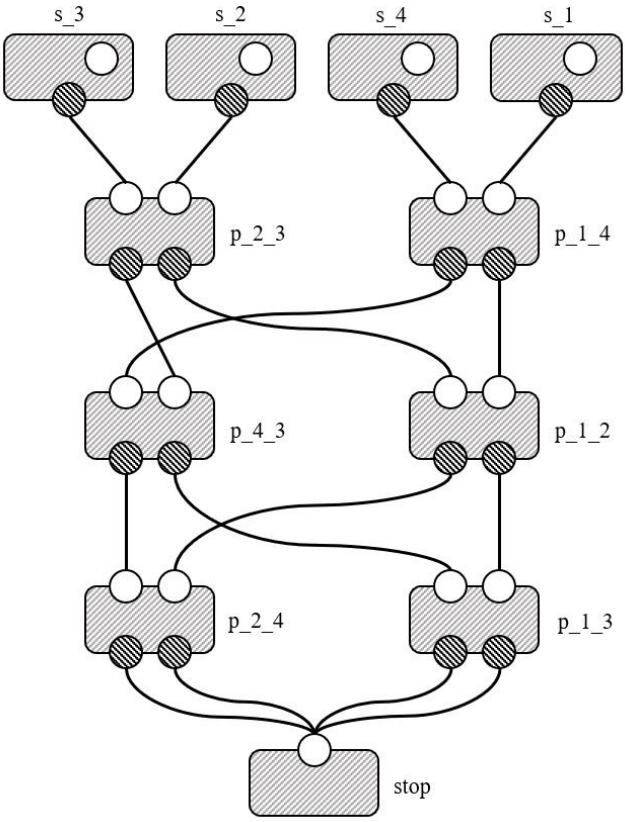

Рис. 4.1. Графическое представление модели «круговой турнир» для 4-х команд в виде системы акторов

Вычисления, как видно из рис. 4.1, инициируются акторами *s_1, s_2, s_3, s_4*. Они отправляют каждый свою «точку данных» в

акторы *p_2_3* и *p_1_4*, параллельно обрабатывающие соответствующие пары на первом этапе. Актор обработки пары запрограммирован так, что вычисления в нём начинаются, как только в каждый из двух обработчиков-запросов актора поступает сообщение.

После завершения обработки пары, входящие в неё «точки данных» формируют новые пары следующих этапов: пары в акторах *p_4_3* и *p_1_2* на втором этапе; пары в акторах *p_2_4* и *p_1_3* на третьем этапе. Остановка вычислений определяется в акторе *stop*, когда в него поступят сообщения со всеми четырьмя «точками данных». Таким образом, графическое представление системы акторов соответствует дагу зависимостей задач при описании потока работ в виде модели «кругового турнира».

Типы акторов-узлов дага потока работ. Рассмотрим подобнее, как программируются акторы, имитирующие зависимости между задачами в даге управления потоком работ. Используемые для описания потока работ типы акторов показаны на рис. 4.2.

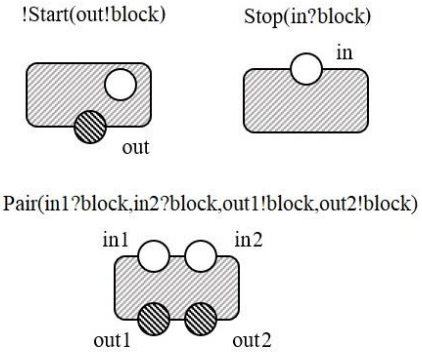

Рис. 4.2. Графическое представление типов акторов
в модели «круговой турнир»

Принцип программирования акторов типа *Start* и *Stop* показан ранее. В акторах типа *Start* просто выполняется отправка сообще-

ния *out* в обработчике стартового сообщения. Для актора типа *Stop* в обработчике сообщения-запроса *on_in* ведется подсчет поступивших сообщений типа *block*. После получения числа сообщений, равного числу обрабатываемых точек данных *N*, выполняется вызов метода *stop*. В акторах типа *Pair*, выполняющих задачу обработки пары «точек данных», использован следующий прием (см. рис. 4.3). В обработчиках сообщений-запросов *on_in1* и *on_in2* фактическая обработка не выполняется, а вызывается метод *current_nearest*. В свою очередь предусловием выполнения этого метода, которое проверяется в строке 153 на рис. 4.3, является поступление двух сообщений *_in1* и *_in2*.

```
152 void current_nearest(){
153   if(access(_in1) && access(_in2)){
154
155     double dist = abs(
156         position[_in1->blockID]-position[_in2->blockID]);
157
158     if(nearest_dist[_in1->blockID] > dist){
159        nearest_dist[_in1->blockID] = dist;
160        nearest_point[_in1->blockID]= _in2->blockID;
161     }
162
163     if(nearest_dist[_in2->blockID] > dist){
164        nearest_dist[_in2->blockID] = dist;
165        nearest_point[_in2->blockID]= _in1->blockID;
166     }
167
168     out1.blockID = _in1->blockID;
169     out2.blockID = _in2->blockID;
170
171     out1.send(); out2.send();
172   }
173 }
```

Рис. 4.3. Действие current_nearest акторов типа Pair

После того, как выполнен расчет дистанции между точками (в строках 155 и 156) и произведен пересчет ближайшего соседа для

первой (в строках 158-161) и второй точки (в строках 163-166), номера точек в полях *blockID* из входных сообщений *_in1* и *_in2* копируются в выходные сообщения *out1* и *out2* (в строках 168-169). После этого сообщения *out1* и *out2* отправляются дальше к акторам следующего этапа (см. рис. 4.1).

Алгоритм синтеза графа потока работ «круговой турнир». Для составления дагов зависимостей задач небольших размеров (см. рис. 4.1) можно воспользоваться ранее рассмотренными методами программирования. Если же даг содержит большое число вершин-задач или имеет регулярную структуру, удобнее воспользоваться следующим методом.

Введем вспомогательный тип *helper* и метод, представляющий обработку пары «точек данных» *pair*. Два параметра метода представляют собой переменные «точки данных» n1 и n2, см. рис 4.4.

```
[1]: /* - code skipped - */
struct helper{
    helper(engine&e):eng(&e){}
    engine*eng;
    block* block_arr[N];

    void start(){
        for(int i=0; i<N; i++){
            Start*s = new Start(*eng);
            s->out.blockID = i;
            block_arr[i] = &(s->out);
        }
    }

    void pair(int n1, int n2){
        Pair* p = new Pair(*eng);
        p->in1(*block_arr[n1]); p->in2(*block_arr[n2]);
        block_arr[n1] = &(p->out1); block_arr[n2] = &(p->out2);
    }

    void stop(){
        Stop* s = new Stop(*eng);
        for(int i=0; i<N; i++) s->in(*block_arr[i]);
    }
};
/* - code skipped - */
```

Рис. 4.4. Вспомогательные процедуры,
используемые для построения дагов обработки паросочетаний

Переменные «точек данных» представлены массивом *block_arr* из указателей на сообщения. Кроме метода *pair* во вспомогательный тип *helper* входит метод *start*, заполняющий массив ссылками на сообщения из акторов типа *Start*, и метод *stop*, подключающий ссылки из *block_arr* ко входу актора типа *Stop*. Дополнительно в методах *start*, *stop* и *pair* выполняется конструирование акторов.

При вызове *helper.pair* в некотором порядке (см. рис. 4.5) происходят следующие действия. Конструируется актор типа *Pair*.

```
[1]: /* - code skipped - */
     help.start();
     cout << endl << "Lucas(Berger) algorithm" << endl;
     for (int r = 1; r <= N + N % 2 - 1; r++) {

         if (N % 2 == 0){ help.pair(r - 1, N - 1);
                 cout << "[" << r - 1 <<","<< N - 1 << "] "; }

         for (int p = 1; p <= (N + N % 2)/ 2 - 1; p++) {
             int t, i, j;

             i = (r + p - 1) % (N + N % 2 - 1);
             j = (N + N % 2 - p + r - 2) % (N + N % 2 - 1);
             if (i > j) { t = i; i = j; j = t; }

             help.pair(i, j); cout << "[" << i <<","<< j << "] ";
         }
         cout << endl;
     }
     cout << endl;
     help.stop();
     /* - code skipped - */
```

Рис. 4.5. Алгоритм синтеза системы акторов в виде дага модели «круговой турнир» для произвольного числа команд N

Ко входам актора присоединяются сообщения типа *block* из массива *block_arr*, имитирующие значения переменных «точек данных» на момент вызова актора типа *Pair*. Ссылки на выходные сообщения актора типа *Pair*, имитирующего вычисленные значения

«точек данных», присваиваются элементам массива *block_arr*. Таким образом в памяти программы автоматически образуется граф зависимостей задач по данным при обработке пар «точек».

От выбора порядка перечисления *helper.pair* зависит распараллеливание вычислений. Так, если, следуя методу составления «кругового турнира» в алгоритме Лукаса-Бергера, сначала перечислить все пары, составляющие первый раунд игр, затем второй раунд и так далее, в итоге будем получать даги зависимостей задач вида рис. 4.1 для заданного числа «точек данных». Формируемый алгоритмом рис. 4.5 порядок перечисления пар для десяти «точек данных» показан на рис. 4.6.

```
Lucas(Berger) algorithm
[0,9] [1,8] [2,7] [3,6] [4,5]
[1,9] [0,2] [3,8] [4,7] [5,6]
[2,9] [1,3] [0,4] [5,8] [6,7]
[3,9] [2,4] [1,5] [0,6] [7,8]
[4,9] [3,5] [2,6] [1,7] [0,8]
[5,9] [4,6] [3,7] [2,8] [0,1]
[6,9] [5,7] [4,8] [0,3] [1,2]
[7,9] [6,8] [0,5] [1,4] [2,3]
[8,9] [0,7] [1,6] [2,5] [3,4]
```

Рис. 4.6. Порядок перечисления вершин дага модели «круговой турнир» по алгоритму Лукаса-Бергера для 10-и команд

После построения описанным способом системы акторов, происходит запуск вычислений в ней. Вычислительный процесс в системе акторов воспроизводит искомый поток работ нахождения ближайшего соседа для каждой «точки данных».

Полный код рассмотренного примера содержится в файле ~/samples/booksmpl/4-1-pairwiseNNS.hpp, а также в файле ~/samples/booksmpl/4-1-pairwiseNNS.ipynb репозитория системы Templet.

4.2. Простая «сортирующая сеть»

Другие применения параллельного перебора паросочетаний. Определение операции обработки пары элементов. Синтез дага потока работ в виде системы акторов.

Другие применения параллельного перебора паросочетаний. Метод синтеза систем акторов, реализующих поток работ на основе паросочетаний, можно адаптировать для решения разнообразных задач. Адаптация заключается в переопределении алгоритма обработки пары «точек данных» и алгоритма синтеза графа потока работ путем перечисления операций попарной обработки в необходимом порядке.

Рассмотрим применение метода для сортировки массива чисел (см. рис. 4.7 на стр. 69). Пусть сортируемый массив содержит четыре элемента – четыре переменные, содержащие числа. Сначала сравним первую переменную со второй и выполним обмен их содержимым, если нарушается порядок следования: в переменной с меньшим номером должно находиться меньшее число. После сравнения-обмена числа в переменных 1 и 2 упорядочены. Возьмем следующую 3-ю переменную и последовательно выполним операцию сравнение-обмен с переменными 1 и 2. В результате в переменных 1, 2 и 3 будут содержаться упорядоченные значения. Наконец, сравним также 4-ю переменную с переменными 1, 2 и 3. В итоге получим массив из 4-х переменных с упорядоченным содержимым. Приведенная процедура для 4-х переменных легко обобщается на массив переменных произвольного размера.

Заметим, что процедура имеет внутренний параллелизм. При выполнении сравнения-обмена третьей переменной со второй переменной одновременно можно начать следующую операцию сравнения-обмена четвертой переменной с первой переменной. Это наглядно видно при рассмотрении дага зависимостей работ на

рис. 4.7, изображающего информационные зависимости между операциями сравнения-обмена в приведенной выше процедуре. Действительно, анализ дага рис. 4.7 показывает, что нет потоков данных как из актора-вершины *p_1_4* в актор-вершину *p_2_3*, так и в обратном направлении. Это означает, что актор *p_1_4* и актор *p_2_3* могут выполнять свои операции сравнения-обмена одновременно.

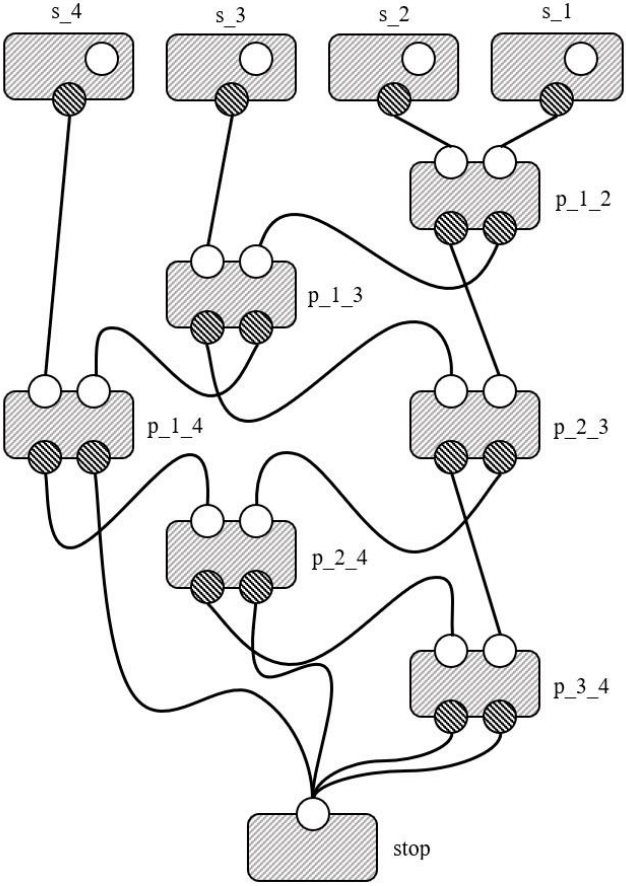

Рис. 4.7. Графическое представление модели «сортирующая сеть» для массива из 4-х элементов в виде системы акторов

Описанная процедура будет иметь практический смысл, если операция сравнения-обмена переупорядочивает не числа, а большие массивы данных, когда в память узла одновременно умещается лишь несколько таких массивов или файлов, представляющих обрабатываемый набор данных.

Определение операции обработки пары элементов. После определения общего принципа организации вычислений по алгоритму приступим к адаптации кода исходного алгоритма п. 4.1. Метод обработки пары элементов *current_nearest* на рис. 4.3 заменим методом *swap*, выполняющим операцию сравнения-обмена пары (см. рис. 4.8).

```
152     void swap(){
153         if(access(_in1) && access(_in2)){
154             if(arr[_in1->blockID] > arr[_in2->blockID]){
155                 int tmp = arr[_in2->blockID];
156                 arr[_in2->blockID] = arr[_in1->blockID];
157                 arr[_in1->blockID] = tmp;
158             }
159             out1.blockID = _in1->blockID;
160             out2.blockID = _in2->blockID;
161 
162             out1.send(); out2.send();
163         }
164     }
```

Рис. 4.8. Действие swap, выполняемое в акторах типа Pair

В методе swap в строках 153 и 159-162 выполняются стандартные действия проверки предусловия запуска операции и передачи номеров обрабатываемых элементов к следующим акторам в потоке работ. Специальный код сравнения-обмена содержится в строках 154-158.

Синтез дага потока работ в виде системы акторов показан на рис. 4.9. Алгоритм построения дага зависимостей задач является обычной процедурой пузырьковой сортировки. Порядок перечисле-

ния пар по этой процедуре для 10-и элементов показан ниже на рис. 4.10.

```
[1]: /* - code skipped - */
     help.start();
     for(int b=0; b < N; b++){
         for(int i=0; i < b; i++){ help.pair(i,b);
             cout << "[" << i+1 <<","<< b+1 << "] ";}
         cout << endl;
     }
     cout << endl;
     help.stop();
     /* - code skipped - */
```

Рис. 4.9. Алгоритм синтеза системы акторов в виде дага модели «сортирующая сеть» для произвольного числа элементов массива

```
[1,2]
[1,3] [2,3]
[1,4] [2,4] [3,4]
[1,5] [2,5] [3,5] [4,5]
[1,6] [2,6] [3,6] [4,6] [5,6]
[1,7] [2,7] [3,7] [4,7] [5,7] [6,7]
[1,8] [2,8] [3,8] [4,8] [5,8] [6,8] [7,8]
[1,9] [2,9] [3,9] [4,9] [5,9] [6,9] [7,9] [8,9]
[1,10] [2,10] [3,10] [4,10] [5,10] [6,10] [7,10] [8,10] [9,10]
```

Рис. 4.10. Порядок перечисления вершин дага модели «сортирующая сеть» для массива из 10-и элементов

Достоинством предложенной технологии программирования является возможность автоматического распараллеливания вычислений при выполнении системы акторов, генерируемой в результате перебора вершин дага. Семейство параллельных алгоритмов сортировки, основанное на операциях сравнения-обмена называется «сортирующие сети». Существуют более эффективные (по сравнению с рассмотренным алгоритмом) «сортирующие сети» как

по количеству операций сравнение-обмен, так и по степени параллелизма. Все они могут быть реализованы с использованием описанного метода программирования путем задания соответствующего порядка обработки пар элементов в информационных массивах.

Полный код рассмотренного примера содержится в файле ~/samples/booksmpl/4-2-sortingnet.hpp, а также в файле ~/samples/booksmpl/4-2-sortingnet.ipynb репозитория системы Templet.

4.3. Практические задания

1. Алгоритм паросочетаний на даге, сравнение с «портфелем задач». Требуется реализовать программную логику алгоритма п. 4.1 с использованием алгоритма «портфель задач». Используйте следующий принцип. За состояние «портфеля» примите множество не рассмотренных пар «точек данных» и множество «точек данных», выданных на рассмотрение в рабочие процессы схемы «портфель-задач» (в составе пар «точек данных»). Множество не рассмотренных пар «точек данных» в начале вычислений генерируется подобно перечислению пар на рис. 4.9, 4.10. Адаптируйте построенный алгоритм и алгоритм п. 4.1 для параллельного или распределенного выполнения в системе Templet. Проведите сравнительное исследование влияния количества «точек данных» на эффективность вычислений в алгоритмах. Сформулируйте выводы по результатам исследования.

2. Сравнение двух сортировочных сетей. Возможны различные порядки применения операций парного обмена данными, приводящие к упорядоченной последовательности. Еще один способ перебора пар в сортирующей сети реализован в примере, содержащимся в файле ~/samples/booksmpl/4-3-sortingnet.ipynb репозитория

проекта Templet. Адаптируйте этот и описанный в п. 4.2 алгоритм для параллельного или распределенного выполнения в системе Templet. Проведите сравнительное исследование влияния количества сортируемых элементов массива на эффективность вычислений. Сформулируйте выводы по результатам исследования.

3. Распараллеливание выражений. Обобщите метод распараллеливания п. 4.1 и п. 4.2 на выражения, составленные из бинарных операций. На практике такими операциями могут быть матричные операции. Для проверки работоспособности алгоритма следует ограничиться скалярными арифметическими операциями (+, -, ×, /). Запрограммируйте типы акторов, соответствующие операциям (+, -, ×, /), и другие вспомогательные типы акторов. Постройте из акторов путем последовательного перечисления операций, как в примерах на рис. 4.5 и рис. 4.9, алгоритмы вычисления произвольных выражений. Адаптируйте их для параллельного или распределенного выполнения в системе Templet. Проведите исследование ускорения алгоритмов, используя фиктивную нагрузку при вычислении выражений. Например, можно приостановить выполнение на заданное время. Сформулируйте выводы по результатам исследования.

ЛИТЕРАТУРА

1. Hewitt C., Bishop P., Steiger R. A universal modular actor formalism for artificial intelligence // Proceedings of the 3rd international joint conference on Artificial intelligence. – 1973. – С. 235-245.

2. Lamport L. Computer science and state machines //Concurrency, Compositionality, and Correctness: Essays in Honor of Willem-Paul de Roever. – 2010. – С. 60–65.

3. Востокин С.В., Бобылева И.В. Применение алгоритмических скелетов для проектирования параллельных алгоритмов акторного типа // Современные информационные технологии и ИТ-образование. – 2020. – Т. 16. – №. 1. – С. 64–71.

4. Востокин С.В., Бобылева И.В. Алгоритмы асинхронных круговых турниров для многозадачных приложений обработки данных // International Journal of Open Information Technologies. – 2020. – Т. 8. – №. 4. – С. 45–53.

5. Vostokin S., Bobyleva I. V. Implementation of frequency analysis of twitter microblogging in a hybrid cloud based on the Binder, Everest platform and the Samara University virtual desktop service //CEUR Workshop Proceedings. – 2020. – Т. 2667. – С. 162–165.

6. Popov S. N., Vostokin S. V., Doroshin A. V. Dynamical systems analysis using many-task interactive cloud computing // Journal of Physics: Conference Series. – IOP Publishing, 2020. – Т. 1694. – №. 1. – С. 012023.

7. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Вильямс. – 2003.

8. Гергель В. П. Теория и практика параллельных вычислений. 2-е изд. – М.: Интуит. – 2016.

Учебное издание

Востокин Сергей Владимирович

МЕТОД СОЗДАНИЯ ПАРАЛЛЕЛЬНЫХ И РАСПРЕДЕЛЕННЫХ ПРОГРАММ В ПАРАДИГМЕ АКТОРОВ

Учебное пособие

Редакционно-издательская обработка
издательства Самарского университета

Подписано в печать 12.09.2023. Формат 60x84 1/16.
Бумага офсетная. Печ. л. 5,0.
Тираж 120 (1-й з-д 1-27 экз.). Заказ № .

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, Самара, Московское шоссе, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.