УДК 004.021
DOI: 10.25559/SITITO.16.202001.64-71

Применение алгоритмических скелетов для проектирования параллельных алгоритмов акторного типа

С. В. Востокин[1*], И. В. Бобылева[1,2]

[1] Самарский национальный исследовательский университет им. академика С.П. Королева, г. Самара, Россия, 443086, Россия, г. Самара, ул. Московское шоссе, д. 34

[*] easts@mail.ru

[2] Акционерное общество «Ракетно-космический центр «Прогресс», г. Самара, Россия, 443009, Россия, г. Самара, ул. Земеца, д. 18

Аннотация

В работе предлагается метод проектирования параллельных алгоритмов. Цель метода – обеспечение точности описания алгоритмов без зависимости от языка программирования и архитектуры параллельной вычислительной системы, используя понятную программисту алгоритмическую форму описания. Цель достигнута путем адаптации алгоритмических скелетов Коула для представления структуры и модели акторов Хьюитта для представления семантики параллельных алгоритмов. Идея метода изложена на примере проектирования параллельного алгоритма сортировки. В исходном алгоритме выделяется код, определяющий порядок вызова блоков сравнения и перестановки элементов сортируемой последовательности. Метод распараллеливания для этого кода применим при распараллеливании семейства последовательных алгоритмов с различными блоками сравнения и перестановки. Проектирование заключается в записи алгоритма в такой последовательной форме, для которой известен метод распараллеливания. Данная последовательная форма алгоритма может быть достаточно общей, независимо выполняется сортировка или другие вычисления. Для этого форма последовательного алгоритма приводится в соответствие с некоторой абстрактной моделью параллельных вычислений. В качестве такой модели нами предложена алгоритмическая интерпретация модели акторов. Представлен подробный пример применения метода. На основе алгоритмической интерпретации модели акторов описан параллельный вычислительный процесс алгоритма сортировки. Алгоритмическая интерпретация модели акторов отличает метод от функциональных и объектно-ориентированных интерпретаций. Метод базируется на фундаментальном понятии последовательного алгоритма, что обеспечивает свободу выбора программной реализации, не требует примитивов синхронизации и коммуникации. Для проектирования используется один базовый скелет для всех алгоритмов вместо системы скелетов в методе Коула. Особенности метода обеспечивают его применимость при проектировании алгоритмов многозадачных вычислений и алгоритмов управления потоками работ. Наш опыт применения метода подтверждает эффективность разработанных на его основе программ для широкого класса параллельных архитектур от одиночных мультипроцессорных до гибридных облачных систем.

Ключевые слова: модель параллельных вычислений, алгоритмический скелет, параллельный алгоритм, модель акторов, поток работ, многозадачные вычисления.

Для цитирования: Востокин, С. В. Применение алгоритмических скелетов для проектирования параллельных алгоритмов акторного типа / С. В. Востокин, И. В. Бобылева. – DOI 10.25559/SITITO.16.202001.64-71 // Современные информационные технологии и ИТ-образование. – 2020. – Т. 16, № 1. – С. 64-71.

© Востокин С. В., Бобылева И. В., 2020

Контент доступен под лицензией Creative Commons Attribution 4.0 License.
The content is available under Creative Commons Attribution 4.0 License.

The Use of Algorithmic Skeletons to Design Actor-Based Parallel Algorithms

S. V. Vostokin[a*], I. V. Bobyleva[a,b]

[a] Samara National Research University named after academician S.P. Korolev, Samara, Russia
34 Moskovskoye shosse, Samara 443086, Russia
[*] easts@mail.ru
[b] Joint Stock Company Space Rocket Center Progress, Samara, Russia
18 Zemetsa St., Samara 443009, Russia

Abstract

The paper proposes a method for writing parallel algorithms. Our goal was to make a detailed description of concurrency while carrying no dependencies in programming language or underlying parallel architecture. We also wanted the description to be algorithmic, so simple for the programmer. The goal was achieved by adapting Cole's algorithmic skeletons to represent the structure and Hewitt's actor model to represent the semantics of parallel algorithms. We used a sorting algorithm to show the idea of the method. To get parallel sorting, one should parallelize the code that controls the order of comparison-swap operations. It is obvious, that the parallelization will be the same for algorithms with different comparison-swap operations. The design method consists in writing algorithms in a form with known parallelization. Consequently, the form of an algorithm presents concurrency. For a more general form than in the sorting example, we used an algorithmic interpretation of the actor model. Based on the algorithmic interpretation of the actor model, we described a parallel computational process of the sorting algorithm. The algorithmic interpretation of the actor model distinguishes our method from functional and object-oriented approaches. It also makes code portable, and does not require synchronization and communication primitives. Only one skeleton is used instead of the skeleton system in Cole's method. The features of the method ensure its applicability in the design of many-task and workflow applications. The method confirms its efficiency for a wide class of parallel architectures ranges from single multiprocessor to hybrid cloud systems.

Keywords: parallel computing model, algorithmic skeleton, parallel algorithm, actor model, workflow, many-task computing.

For citation: Vostokin S.V., Bobyleva I.V. The Use of Algorithmic Skeletons to Design Actor-Based Parallel Algorithms. *Sovremennye informacionnye tehnologii i IT-obrazovanie* = Modern Information Technologies and IT-Education. 2020; 16(1):64-71. DOI: https://doi.org/10.25559/SITITO.16.202001.64-71

Введение

Параллельные алгоритмы описывают особый вид управления вычислительным процессом, когда одновременно выполняются несколько элементарных действий. Феномен одновременности и непредсказуемости порядка выполнения действий обуславливает сложность понимания параллельного алгоритма инженером-программистом. Кодирование, которому не предшествуют формализованные процедуры проектирования и анализа, часто приводит к некорректной реализации. Типовые ошибки реализации параллельных алгоритмов, такие как состояния состязания и тупики разного типа, могут не проявиться на стадии тестирования, в итоге попасть в готовый программный продукт. Источником таких ошибок является неполнота описания алгоритма, приводящая к непредсказуемому поведению порождаемого им вычислительного процесса. Поэтому параллельные алгоритмы являются сложными системами, для которых требуется предварительное проектирование.

На стадии проектирования параллельного алгоритма возникают два связанных инженерных вопроса: каким образом точно и безотносительно к реализующему оборудованию описать его поведение; как обеспечить его эффективную реализацию. Традиционно эти вопросы решаются по отдельности и разными средствами. Для точного описания параллельного алгоритма на стадии проектирования используются математические подходы [1]. Они могут основываться на специальных алгебрах [2], логиках [3], автоматных моделях [4]. Математические подходы зачастую непривычны инженеру-программисту, так как они значительно отличаются от современных средств реализации параллельных алгоритмов (промышленных языков C/C++, Java, C#, Python, Fortran 2018 и др.), базирующихся на императивных моделях программирования, то есть на концепции последовательного алгоритма.

С другой стороны, при реализации параллельных алгоритмов инженеры-программисты используют разнообразные модели вычислений с разными методами синхронизации и коммуникации процессов. Системы с разделяемой памятью, включая системы с графическими сопроцессорами, программируются на основе стандарта OpenMP. Для программирования систем с распределенной памятью используется стандарт MPI. Кроме этого, у промышленных языков программирования имеются стандартизированные параллельные библиотеки и расширения. Стандарты де-факто используют специальные модели вычислений, например, модель MapReduce [5] или модели архитектур графических сопроцессоров [6].

В связи с отличием между математическими методами точной спецификации параллельных алгоритмов и современными методами их программной реализации, а также большим разнообразием методов реализации, актуален вопрос об унификации спецификации и реализации на основе специально разработанного метода проектирования. В частности, авторы столкнулись с данной проблемой при разработке многозадачных алгоритмов и алгоритмов потоков работ, возникающих при попарном сопоставлении элементов информационных массивов [7]. Для этого требовался метод проектирования единообразно описывающий разные алгоритмы попарной обработки данных. При этом требовалось учитывать разнообразие используемых аппаратных архитектур, варьирующихся от одиночных многопроцессорных машин до гибридных облачных систем.

В работе предлагается метод проектирования параллельных алгоритмов, который унифицирует спецификацию и реализацию многозадачных алгоритмов и схем потоков работ. Цель метода – обеспечение точности описания алгоритмов указанного типа без внесения зависимостей от языка программирования и архитектуры вычислительной системы, используя понятную инженеру-программисту алгоритмическую форму описания.

Работа состоит из следующих частей. В первой части описана идея метода проектирования параллельных алгоритмов, основанная на концепции алгоритмического скелета Коула [8]. Во второй части алгоритмическая интерпретация модели акторов Хьюитта [9] используется для построения универсального алгоритмического скелета. В третьей части рассмотрено применение данного алгоритмического скелета для описания алгоритма параллельной сортировки. В четвертой части обсуждаются особенности разработанного метода проектирования параллельных алгоритмов. В заключении подводится итог работы и даются рекомендации по областям возможного применения разработанного метода.

Идея метода проектирования параллельных алгоритмов

Известно, что построить алгоритмическое, удобное для понимания и анализа, независящее от реализующей аппаратуры описание параллельного вычислительного процесса, можно на основе последовательного алгоритма. Например, технология OpenMP позволяет применить инкрементное распараллеливание: путем добавления разметки в код можно получить эквивалентную итеративно-параллельную или рекурсивно-параллельную программу без модификации исходного последовательного кода. Получающиеся параллельные программы эффективно исполняются на системах с разделяемой памятью.

С целью адаптации данного метода для описания схем потоков работ, выполняющихся в грид-системах и гибридных облачных системах, мы применяем: (а) технику обобщенного программирования [10] вместо разметки кода; (б) модель акторов [9] вместо моделей итеративного и рекурсивного параллелизма. Данный метод объясняется на примере построения параллельного аналога алгоритма сортировки (Листинг 1) в виде схемы потока работ.

Листинг 1 – Исходный алгоритм последовательной сортировки
для i от 0 до M-1
    выполнить ввод элемента A[i] массива A
для i от 1 до M-1, для j от 0 до j<i
    выполнить перестановку A[j] и A[i], если A[j] >= A[i]

Выполним иерархическую декомпозицию алгоритма Листинга 1. На верхнем уровне иерархии выполняется поэлементное и попарное перечисление элементов некоторой абстрактной последовательности. Затем на нижнем уровне иерархии выполняются действия, специфичные именно для сортировки. Верхний уровень декомпозиции, который представляет собой обобщенный алгоритм, называемый далее алгоритмическим скелетом [8], приведен в Листинге 2.

Листинг 2 – Алгоритмический скелет последовательной сортировки

```
sort(
    prepare(int)
    swap_if_not_ordered(int,int)
)
для i от 0 до M-1
    выполнить prepare(i)
для i от 1 до M-1, для j от 0 до j<i
    выполнить swap_if_not_ordered(j,i)
```

Пусть для алгоритмического скелета (Листинг 2) известен параллельный аналог. Тогда, определяя последовательные процедуры *prepare* и *swap_if_not_ordered*, манипулирующие блоками чисел, получим параллельный алгоритм блочной сортировки. Определив *prepare* и *swap_if_not_ordered* другим способом, получим ещё один параллельный алгоритм и так далее. Таким образом, параллельный алгоритм может быть описан с использованием его алгоритмического скелета, заданного в форме последовательного обобщенного алгоритма.

Алгоритмический скелет акторных алгоритмов

Алгоритмический скелет *sort* из Листинга 2 используется для построения параллельных алгоритмов сортировки. В этом разделе мы исследуем, можно ли по аналогии с рассуждениями из раздела 2 построить алгоритмический скелет произвольного параллельного алгоритма безотносительно к его назначению. В качестве основы для построения алгоритмического скелета будем использовать абстрактную модель параллельного выполнения, для которой построим аналог в виде последовательного обобщенного алгоритма. С учетом требования удобства описания схем потоков работ в качестве основы для нашего алгоритмического скелета подходит модель акторов Хьюитта [9]. Конструируемый алгоритмический скелет *AM* будет иметь структуру, показанную в Листинге 3.

Листинг 3 – Структура алгоритмического скелета акторных алгоритмов

```
AM<actor,message>(
    init(){set(message,actor,boolean)}
    recv(message,actor){ send(message,actor) access(message) }
)
```

В модели используются тип данных *actor* с полями *mes* и *active* и тип данных *message* с полями *act* и *active*. Обозначение типов *actor* и *message* в скобках < > говорит о том, что в алгоритмах, построенных на базе алгоритмического скелета *AM*, эти типы могут быть дополнены необходимыми полями.

По аналогии с Листингом 2 при специализации алгоритмического скелета *AM* потребуется определить две процедуры: *init* для инициализации вычислений и *recv* для обозначения действий, выполняемых при поступлении сообщения в актор. В скобках { } дополнительно указано, что в *init* используется процедура *set*, ранее определенная в алгоритмическом скелете *AM*, а в *recv* – процедуры *send* и *access*.

Семантику модели раскрывают четыре алгоритма. Собственно алгоритмический скелет *AM* показан в Листинге 4.

Листинг 4 – Алгоритмический скелет акторных алгоритмов

```
выполнить init()
пока есть такие m, что m.active = true выполнять
    m.active := false
    m.act.active := true
    выполнить recv(m, m.act)
    m.act.active := false
```

В начале в *init* определяется произвольное число акторов и сообщений. Так как в нашей модели акторы неактивны, пока к ним не поступят сообщения, требуется указать активные сообщения, направляющиеся к определенным акторам. Это делается путем вызова вспомогательной процедуры *set(m, a, true)* из Листинга 5, что означает: сообщение *m* направляется в актор *a*.

Листинг 5 – Вспомогательная процедура *set*

```
вход: mes, act, activity
mes.act := act
mes.active := activity
```

Вызов *set(m, a, false)* означает, что сообщение *m* было доставлено в актор *a* перед началом вычислений. Данная инициализация требуется для управления доступом к состоянию вычислений из актора *a*, как показано далее. Все акторы и сообщения, используемые в процедуре *recv*, должны быть инициализированы. Свойство классической акторной модели динамически создавать акторы и сообщения может быть реализовано менеджером кучи, что предполагает лишь наличие пула акторов и сообщений, инициализируемых аналогично.

В цикле обработки сообщений Листинга 5 сообщение *m*, поступившее в актор *m.act*, становится неактивным, при этом активируется актор *m.act*. Затем выполняется обработка сообщения в акторе *m.act* с использованием *recv(m, m.act)*, которая завершается переходом актора в неактивное состояние.

Параллелизм модели заключается в том, что одновременно можно выполнять любые итерации с *m.active = true*, при том, что для любых выполняющихся одновременно итераций m_1 и m_2, выполняется $m_1.act \neq m_2.act$. Это означает, что если имеются два и более сообщения, поступающих в один и тот же актор, то они обрабатываются последовательно, по одному.

Корректное распараллеливание также предполагает «справедливую» доставку сообщений: если сообщение отправлено (*m.active = true*), то оно будет рано или поздно доставлено (*m.active = false*).

В контексте обработчика сообщения *recv* известен актор, получивший сообщение, и само сообщение. Имеется возможность отправить сообщение в произвольный актор при помощи вызова вспомогательной процедуры *send* из Листинга 6.

Листинг 6 – Вспомогательная процедура *send*

```
вход: mes, act
mes.active := true
mes.act := act
```

Так как все сообщения в нашей версии модели акторов это переменные в глобальной памяти, чтобы исключить состояния состязания требуется определение доступности сообщения в контексте данного вызова *recv*. Для этого служит вспомогательная процедура *access*, показанная в Листинге 7.

Листинг 7 – Вспомогательная процедура *access*

```
вход: mes
вернуть mes.active = false ∧ mes.act = m.act,
где m – это сообщение в текущем вызове recv(m,m.act)
```

Код Листинга 7 означает, что получить доступ можно только к неактивному сообщению, которое перед этим либо было привязано к текущему актору при помощи *set*, либо было получено данным актором в *recv*. Аналогично, вызов *access* используется для проверки доступа к другим переменным, добавляемым при специализации алгоритмического скелета. Например, если переменная *x* по смыслу алгоритма связана с сообщением *m*, то доступ к ней разрешен, если *access(m)=true*. Очевидно, что доступ к переменной *y*, по смыслу алгоритма связанной с актором *a*, разрешен в контексте вызова *recv(–,a)*. Возможно определить и более сложные правила доступа. Заметим, что наше определение правил доступа не требует введения локальной памяти акторов.

Пример определения алгоритма параллельной сортировки

Рассмотрим применение алгоритмического скелета акторных алгоритмов для описания параллельного вычислительного процесса, реализующего алгоритм из Листинга 2. Идея распараллеливания заключается в представлении вычислительного процесса в виде конвейера, по которому последовательно проходят числа от *0* до *M-1*. Впервые попавшее в ступень конвейера число *i* запоминается, а последующие числа *j* проходят дальше. Перед передачей числа *j* дальше по конвейеру выполняется вызов *swap_if_not_ordered(i,j)*. Момент достижения числом *M-1* последней ступени означает остановку вычислений в состоянии, когда массив отсортирован. Дополнительно мы распараллеливаем цикл, содержащий вызовы *prepare* из Листинга 2.

Вначале определяем расширения типов *message* и *actor* для нашего алгоритма сортировки. Тип *message* расширен следующими полями: *j* – для передачи чисел между акторами, *client* – типа *actor*, *server* – типа *actor*. Каждое сообщение будет использоваться для связи между двумя акторами, ссылки на которые хранят поля *client* и *server*.

Акторы типа *prep* используются для предварительной обработки элементов сортируемой последовательности в процедуре *prepare*. Тип *prep* расширяет тип *actor* полем *next* типа *message*. Отправка сообщения *next* является сигналом завершения процедуры *prepare*.

После того, как все акторы типа *prep* завершат выполнение, начинается передача чисел в сортирующий конвейер. Определение момента завершения всех процедур *prepare*, а также выдача чисел в конвейер выполняются единственным актором типа *collector* с дополнительными полями: числовым полем *m* – для подсчета завершившихся процедур *prepare* и выданных чисел; полем *next* – для хранения сообщения, передающего число в сортирующий конвейер.

Ступени конвейера образованы акторами типа *stage* с полями: *i* – для хранения числа, пришедшего в ступень первым; *prev* – для хранения ссылки на сообщение, доставляющее число в ступень; *next* – для хранения сообщения, передающего числа к следующим ступеням.

Тип *stopper* содержит единственный актор, служащий для фиксации момента завершения вычислений. Кроме метки типа он не имеет других особенностей.

Алгоритм инициализации *init* для построения структуры данных, состоящей из объектов описанных типов, показан в Листинге 8. Имена объектов (переменных) записаны в форме *a_<имя_типа>[<номер_экземпляра>]*.

Листинг 8 – Определение инициализации *init* в алгоритме параллельной сортировки

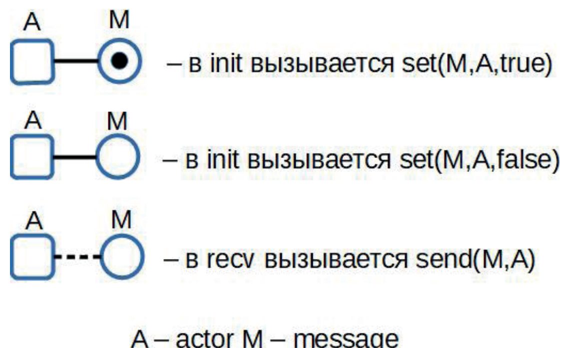
```
для i от 0 до M-1 выполнять
    a_prep[i].next.server := a_collector
    a_prep[i].next.client := 0
    a_prep[i].next.j := i
    выполнить set(a_prep[i].next,a_prep[i],true)
a_collector.m := 0
a_collector.next.server := a_stage[0]
a_collector.next.client := a_collector
выполнить set(a_collector.next,a_collector,false)
для i от 0 до M-3 выполнять
    a_stage[i].i := 0
    a_stage[i].prev := 0
    a_stage[i].next.server := a_stage[i+1]
    a_stage [i].next.client := a_stage[i]
    выполнить set(a_stage[i].next,a_stage[i],false)
a_stage[M-2].i := 0
a_stage[M-2].prev := 0
a_stage[M-2].next.server := a_stopper
a_stage[M-2].next.client := a_stage[M-2]
выполнить set(a_stage[M-2].next,a_stage[M-2],false)
```

Получающуюся при работе *init* информационную структуру (Рисунок 2) можно визуализировать с использованием обозначений Рисунка 1. Эта структура является аналогом диаграммы потоков работ для рассматриваемого параллельного алгоритма. В итоге, на основе описания информационной структуры можно построить точное алгоритмическое определение вычислительного процесса сортировки, определив (в Листинге 9) процедуру *recv* для обработки сообщений в акторах.

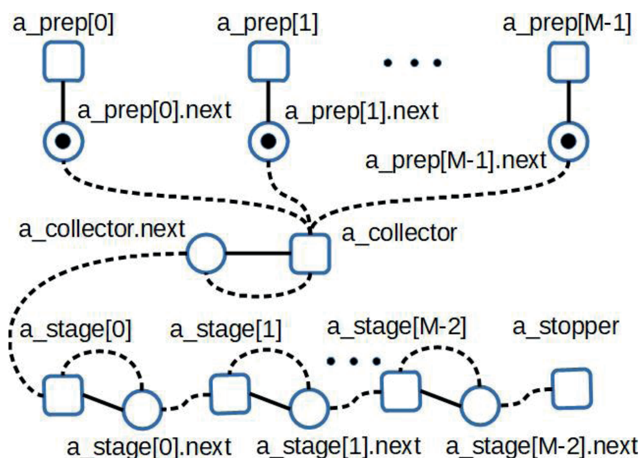
Листинг 9 – Определение обработки сообщений *recv* в алгоритме параллельной сортировки

```
вход: msg, act
если тип act = prep то
    выполнить prepare(msg.j)
    выполнить send(act.next,act.next.server)
если тип act = collector то
    act.m :=  act.m + 1
    если msg ≠ act.next
        если act.m = M то
            act.m := 0
            act.next.j := 0
            выполнить send(act.next,act.next.server)
    иначе msg = act.next
        если act.m < M то
            act.next.j := act.m
            выполнить send(act.next,act.next.server)
если тип act = stage то
    если msg ≠ act.next Ù act.prev = 0 то
        act.prev := msg; act.i := msg.j
        выполнить send(act.prev, act.prev.client)
```

если access(act.prev) Ù access(act.next) то
выполнить swap_if_not_ordered(act.i, act.prev.j)
act.next.j := act.prev.j
выполнить send(act.prev, act.prev.client)
выполнить send(act.next, act.next.server)
если тип act = stopper то
вычисления завершены



Р и с. 1. Условные обозначения, используемые для описания информационной структуры алгоритмов
F i g. 1. Symbols used to describe the information structure of algorithms



Р и с. 2. Информационная структура алгоритма параллельной сортировки
F i g. 2. Information structure of the parallel sorting algorithm

Таким образом, в Листингах 4-8 содержится точная спецификация вычислительного процесса параллельной сортировки. При этом мы не использовали дополнительных примитивов синхронизации и коммуникации процессов, а лишь указали способ распараллеливания цикла обработки сообщений в Листинге 4 и специальным образом сформировали структуру алгоритма.

Обсуждение результатов

Концепция алгоритмических скелетов для автоматизации проектирования параллельных алгоритмов и программ была предложена в работах Мюррея Коула [8]. Положения этой концепции рассмотрены во второй части статьи. Основным отличием нашего подхода от подхода Коула является унификация на базе одного универсального скелета. Альтернативой унификации, позволяющей избежать потенциальных проблем эффективности вычислений по конструируемым программам, по Коулу являются библиотеки алгоритмических скелетов, обзор которых приведен в работе [11].

Наш подход к унификации основан на приеме структурирования последовательного алгоритма, описываемого произвольным псевдокодом. Для передачи такой структуры в реальных программах на объектно-ориентированных языках могут использоваться классы [12], на языках обобщенного программирования [10] – шаблоны, на императивном языке – препроцессоры кода [13]. Подход не зависит от конкретного метода структурирования алгоритма.

В качестве целевой модели организации вычислений в предложенном методе мы рассматриваем модель многозадачных вычислений [14]. Исполнение многозадачной программы основано на динамическом формировании множества независимо вычисляемых задач. В нашем примере (Листинги 2 и 8) такие задачи порождаются последовательными процедурами prepare и swap_if_not_ordered. Для данной целевой модели в экспериментах на многопроцессорных и многоядерных системах с общей памятью, кластерных системах, грид-системах предприятия нами продемонстрирована возможность эффективных вычислений [15,16,17,18,19].

При решении практических проблем возникает ситуация, когда не удается подобрать подходящий алгоритмический скелет из библиотеки. Известным решением является комбинирование алгоритмических скелетов, что требует определения семантики такого комбинирования [20]. Наш подход является универсальным, по крайней мере, для описания многозадачных алгоритмов [14] и алгоритмов управления потоками работ [21].

В области управления потоками работ обычно используют спецификации в форме графов зависимостей работ [22]. Наш подход позволяет моделировать такие графы, а также описывать многократное срабатывание вершин-работ на графах и порождать графы динамически. Это свойство удобно для задания графов потоков работ больших размеров.

За основу для определения семантики параллельного выполнения нами взята модель акторов Карла Хьюитта [9]. Однако для применения в качестве алгоритмического скелета исходная модель акторов модифицирована: введено глобальное состояние, изменена трактовка локальности действий в акторах, сообщения считаются переменными. Работа актора описывается последовательным алгоритмом, поэтому не требует введения специальных примитивов синхронизации. Предложенные модификации следуют главному принципу акторной модели (the Isolated Turn Principle – принцип изолированного шага), однако являются чисто алгоритмическими. На наш взгляд, это более удобно для проектирования и последующей реализации параллельных алгоритмов по сравнению с известными функциональными и объектно-ориентированными подходами, описанными в работе [23].

В работе не исследуется вопрос сходства и различий вычислительных процессов, порождаемых в разных стратегиях выполнения цикла обработки сообщений (последовательной, недетерминированной, параллельной). Предполагается, что это может быть сделано на основе темпоральной логики Лампорта [24] с использованием рассмотренного в работе [25] подхода.

Заключение

Нами предложен метод проектирования параллельных вычислительных процессов, особенностью которого является применение последовательных алгоритмов специальной структуры, называемой алгоритмическим скелетом, с указанием метода распараллеливания. Метод предназначен для унифицированного, не привязанного к конкретной программной платформе, представления многозадачных алгоритмов и алгоритмов управления потоками работ.

Мы считаем, что метод окажется полезным при реализации вычислений в грид-системах и гибридных облачных системах, а также в методических целях для описания распределенных алгоритмов, что планируется в будущих исследованиях.

References

[1] Lamport L. If You're Not Writing a Program, Don't Use a Programming Language. *Bulletin of the EATCS*. 2018; (125). Available at: http://bulletin.eatcs.org/index.php/beatcs/article/view/539/532 (accessed 14.03.2020). (In Eng.)

[2] Gibson-Robinson T., Armstrong P., Boulgakov A., Roscoe A.W. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*. 2016; 18(2):149-167. (In Eng.) DOI: https://doi.org/10.1007/s10009-015-0377-y

[3] Abbassi A., Bandali A., Day, N., Serna J. A Comparison of the Declarative Modelling Languages B, Dash, and TLA+. In: 2018 IEEE 8th International Model-Driven Requirements Engineering Workshop (MoDRE), Banff, AB, 2018; p. 11-20. (In Eng.) DOI: https://doi.org/10.1109/MoDRE.2018.00008.

[4] Ulyantsev V., Buzhinsky I., Shalyto A. Exact finite-state machine identification from scenarios and temporal properties. *International Journal on Software Tools for Technology Transfer*. 2018; 20(1):35-55. (In Eng.) DOI: https://doi.org/10.1007/s10009-016-0442-1

[5] Li R., Hu H., Li H., Wu Y., Yang J. MapReduce Parallel Programming Model: A State-of-the-Art Survey. *International Journal of Parallel Programming*. 2016; 44(4):832-866. (In Eng.) DOI: https://doi.org/10.1007/s10766-015-0395-0

[6] Memeti S., Li L., Pllana S., Kołodziej J., Kessler C. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In: Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC '17). Association for Computing Machinery, New York, NY, USA, 207; p. 1-6. (In Eng.) DOI: https://doi.org/10.1145/3110355.3110356

[7] Vostokin S.V., Bobyleva I.V. Asynchronous round-robin tournament algorithms for many-task data processing applications. *International Journal of Open Information Technologies*. 2020; 8(4):45-53. Available at: https://www.elibrary.ru/item.asp?id=42748925 (accessed 14.03.2020). (In Russ., abstract in Eng.)

[8] Cole M.I. Algorithmic skeletons: structured management of parallel computation. London: Pitman; 1989. (In Eng.)

[9] Hewitt C. Actor Model of Computation: Scalable Robust Information Systems. CoRR. 2010; abs/1008.1459. Available at: http://arxiv.org/abs/1008.1459 (accessed 14.03.2020). (In Eng.)

[10] Stepanov A.A., Rose D.E. From Mathematics to Generic Programming. Addison-Wesley Professional; 2014. (In Eng.)

[11] González-Vélez H., Leyton M. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*. 2010; 40(12):1135-1160. (In Eng.) DOI: https://doi.org/10.1002/spe.1026

[12] Gupta M. Akka Essentials. Packt Publishing Ltd; 2012. (In Eng.)

[13] Grossman M., Shirako J., Sarkar V. OpenMP as a High-Level Specification Language for Parallelism. In: N. Maruyama, B. de Supinski, M. Wahib (ed.) OpenMP: Memory, Devices, and Tasks. IWOMP 2016. Lecture Notes in Computer Science, vol. 9903. Springer, Cham, 2016. p. 141-155. (In Eng.) DOI: https://doi.org/10.1007/978-3-319-45550-1_11

[14] Raicu I., Foster I.T., Zhao Y. Many-task computing for grids and supercomputers. In: 2008 Workshop on Many-Task Computing on Grids and Supercomputers. Austin, TX, 2008; p. 1-11. (In Eng.) DOI: https://doi.org/10.1109/MTAGS.2008.4777912

[15] Vostokin S.V., Sukhoroslov O.V., Bobyleva I.V., Popov S.N. Implementing computations with dynamic task dependencies in the desktop grid environment using Everest and Templet Web. *CEUR Workshop Proceedings*. 2018; 2267:271-275. Available at: http://ceur-ws.org/Vol-2267/271-275-paper-51.pdf (accessed 14.03.2020). (In Eng.)

[16] Vostokin S.V., Bobyleva I.V. Using the bag-of-tasks model with centralized storage for distributed sorting of large data array. *CEUR Workshop Proceedings*. 2019; 2416:199-203. Available at: http://ceur-ws.org/Vol-2416/paper26.pdf (accessed 14.03.2020). (In Eng.)

[17] Vostokin S.V., Kazakova I.V. Implementation of stream processing using the actor formalism for simulation of distributed insertion sort. *Journal of Physics: Conference Series*. 2018; 1096(1):012087. (In Eng.) DOI: https://doi.org/10.1088/1742-6596/1096/1/012087

[18] Popov S.N., Bobyleva I. V., Vostokin S.V. Distributed Block Sort: a Sample Application for Data Processing in Mobile ad HOC Networks. *International Journal of Innovative Technology and Exploring Engineering*. 2019; 8(7S2):565-568. Available at: https://www.ijitee.org/wp-content/uploads/papers/v8i7s2/G10960587S219.pdf (accessed 14.03.2020). (In Eng.)

[19] Vostokin S., Bobyleva I. Building an Algorithmic Skeleton for Block Data Processing on Enterprise Desktop Grids. In: V. Voevodin, S. Sobolev S. (ed.) Supercomputing. RuSCDays 2019. Communications in Computer and Information Science, vol. 1129. Springer, Cham, 2019. p. 678-689. (In Eng.) DOI: https://doi.org/10.1007/978-3-030-36592-9_55

[20] Zandifar M., Abdul Jabbar M., Majidi A., Keyes D., Amato N.M., Rauchwerger L. Composing Algorithmic Skeletons to Express High-Performance Scientific Applications. In: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15). Association for Computing Machinery, New York, NY, USA; 2015. p. 415-424. (In Eng.) DOI: https://doi.org/10.1145/2751205.2751241

[21] Yu J., Buyya R. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Record*. 2005; 34(3):44-49. (In Eng.) DOI: https://doi.org/10.1145/1084805.1084814
[22] Sukhoroslov O. Supporting Efficient Execution of Workflows on Everest Platform. In: V. Voevodin, S. Sobolev (ed.) Supercomputing. RuSCDays 2019. Communications in Computer and Information Science, vol. 1129. Springer, Cham; 2019. p. 713-724. (In Eng.) DOI: https://doi.org/10.1007/978-3-030-36592-9_58
[23] De Koster J., Van Cutsem T., De Meuter W. 43 years of actors: a taxonomy of actor models and their key properties. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016). Association for Computing Machinery, New York, NY, USA; 2016. p. 31-40. (In Eng.) DOI: https://doi.org/10.1145/3001886.3001890
[24] Lamport L. Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co; 2002. (In Eng.)
[25] Vostokin S.V. The Templet parallel computing system: specification, implementation, applications. *Procedia Engineering*. 2017; 201:P. 684-689. (In Eng.) DOI: https://doi.org/10.1016/j.proeng.2017.09.683

Поступила 14.03.2020; принята к публикации 20.03.2020; опубликована онлайн 25.05.2020.
Submitted 14.03.2019; revised 20.03.2020; published online 25.05.2020.

Об авторах:

Востокин Сергей Владимирович, профессор кафедры информационных систем и технологий, Институт информатики, математики и электроники, Самарский национальный исследовательский университет им. академика С.П. Королева (443086, Россия, г. Самара, ул. Московское шоссе, д. 34), доктор технических наук, доцент, http://orcid.org/0000-0001-8106-6893, easts@mail.ru
Бобылева Ирина Владимировна, аспирант кафедры информационных систем и технологий, Институт информатики, математики и электроники, Самарский национальный исследовательский университет им. академика С.П. Королева (443086, Россия, г. Самара, ул. Московское шоссе, д. 34); инженер-программист, Акционерное общество «Ракетно-космический центр «Прогресс» (443009, Россия, г. Самара, ул. Земеца, д. 18), http://orcid.org/0000-0001-5660-3503, ikazakova90@gmail.com

Все авторы прочитали и одобрили окончательный вариант рукописи.

About the authors:

Sergey V. Vostokin, Professor of the Department of Information Systems and Technologies, Institute of IT, Mathematics and Electronics, Samara National Research University named after academician S.P. Korolev (34 Moskovskoye shosse, Samara 443086, Russia), Dr.Sci. (Technology), Associate Professor, http://orcid.org/0000-0001-8106-6893, easts@mail.ru
Irina V. Bobyleva, Postgraduate student of the Department of Information Systems and Technologies, Institute of IT, Mathematics and Electronics, Samara National Research University named after academician S.P. Korolev (34 Moskovskoye shosse, Samara 443086, Russia); Software Developer, Joint Stock Company Space Rocket Center Progress (18 Zemetsa St., Samara 443009, Russia), http://orcid.org/0000-0001-5660-3503, ikazakova90@gmail.com

All authors have read and approved the final manuscript.