УНИВЕРСИТЕТ ИТМО

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
июль–август 2021 Том 21 № 4 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
July–August 2021 Vol. 21 No 4 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi:10.17586/2226-1494-2021-21-4-473-481
УДК 004.413.5

Оценка применимости методов асинхронного программирования при решении проблемы согласованности данных в микросервисной среде

Константин Владимирович Малюга[1]✉, Иван Андреевич Перл[2], Александр Петрович Слапогузов[3]

[1,2,3] Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация

[1] konstantin.malyuga@gmail.com✉, https://orcid.org/0000-0001-7381-2067
[2] ivan.perl@itmo.ru, https://orcid.org/0000-0002-8903-405X
[3] slapoguzov@gmail.com, https://orcid.org/0000-0003-0699-5478s

Аннотация

Предмет исследования. Рассмотрена проблема организации эффективного взаимодействия микросервисов в отказоустойчивых системах с высокой нагрузкой для обеспечения согласованности данных. В качестве способа организации управления микросервисами выбран шаблон управления Сага (Saga) в оркестрационной форме. Проанализирована целесообразность использования принципов асинхронного программирования при проектировании координатора Саг. **Метод.** Выполнена симуляция процессов работы координатора Саг, которая учитывает специфику синхронных и асинхронных конфигураций управления распределенными транзакциями (Сагами). Представленные синхронные конфигурации включают в себя координатор, управляющий фиксированным набором потоков и координатор, формирующий новый поток для каждой Саги. В группу асинхронных конфигураций входят координатор, управляющий корутинами и координатор, использующий планировщик ядра Linux. **Основные результаты.** Рассмотрены симуляции при различных значениях количества обрабатываемых Саг и доступных координатору процессоров. Показано, что при использовании асинхронных подходов имеет место значительное увеличение скорости выполнения набора Саг (до 9,74 раз) и скорости утилизации процессорного времени (до 88 %). Это подтверждает целесообразность их использования при проектировании координатора Саг. Показано, что различие в эффективности рассмотренных асинхронных подходов незначительно. **Практическая значимость.** Построение оркестратора Саг с применением асинхронных подходов позволит обрабатывать большую нагрузку и эффективнее распределять доступное процессорное время. Результаты исследования могут быть применены при проектировании высоконагруженных распределенных отказоустойчивых вычислительных систем. Оценка, выполненная в работе, будет полезна IT-специалистам и исследователям для решения проблем распределенных вычислений.

Ключевые слова

микросервисы, координатор Саг, оркестрация, асинхронное программирование, программная симуляция, корутины, пул потоков

Ссылка для цитирования: Малюга К.В., Перл И.А., Слапогузов А.П. Оценка применимости методов асинхронного программирования при решении проблемы согласованности данных в микросервисной среде // Научно-технический вестник информационных технологий, механики и оптики. 2021. Т. 21, № 4. С. 473–481. doi: 10.17586/2226-1494-2021-21-4-473-481

Evaluation of the applicability of asynchronous programming methods to the data consistency problem in a microservices environment

Konstantin V. Malyuga[1]✉, Ivan A. Perl[2], Aleksandr P. Slapoguzov[3]

[1,2,3] ITMO University, Saint Petersburg, 197101, Russian Federation

[1] konstantin.malyuga@gmail.com✉, https://orcid.org/0000-0001-7381-2067
[2] ivan.perl@itmo.ru, https://orcid.org/0000-0002-8903-405X
[3] slapoguzov@gmail.com, https://orcid.org/0000-0003-0699-5478

Abstract
The paper considers the problem of an effective microservices interaction and its organization to support data consistency in fault tolerant and high load systems. The "Saga" microservices orchestration template was used for microservices management. The authors assessed the expediency of asynchronous programming principles usage for designing the Saga coordinator. The simulation of processes in the Saga coordinator was conducted; it considers different specifics of asynchronous and synchronous configurations of distributed transactions (Sagas) management. Synchronous configuration group includes a coordinator with a fixed pool of threads and a coordinator that generates a new thread for each new Saga. Asynchronous configuration group consists of a coroutines based coordinator and a coordinator that uses Linux core scheduler. The set of simulations with different numbers of Sagas and available coordinator processors was executed. It was shown that the use of asynchronous approaches significantly reduces the Saga's execution duration up to 9.74 times and improves the processor time utilisation value up to 88 %. The obtained data proves the efficiency of asynchronous programming principles applied to the design of the Saga coordinator. The difference in efficiency between the asynchronous algorithms that were implemented in this paper was insignificant. Asynchronous programming principles used to build the Saga coordinator allow it to handle bigger load and to use processor resources in a more efficient way. The outcome of this research can be applied during the design of fault tolerant and high load systems. The paper might be interesting to IT-specialists and researchers focusing on distributed computing.

Keywords
microservices, Saga coordinator, orchestration, asynchronous programming, programme simulation, coroutines, thread pool

For citation: Malyuga K.V., Perl I.A., Slapoguzov A.P. Evaluation of the applicability of asynchronous programming methods to the data consistency problem in a microservices environment. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2021, vol. 21, no. 4, pp. 473–481 (in Russian). doi: 10.17586/2226-1494-2021-21-4-473-481

Введение

При реализации современных облачных вычислительных систем часто предпочтение отдают микросервисному подходу. В основе данного подхода лежит реализация архитектуры системы в виде набора независимых программ, каждая из которых формирует свою уникальную область логики итогового продукта. В таких системах доступ к данным, например, к базе данных, не предоставляется интересующейся стороне напрямую, а организуется через сервис, ответственный за базу данных. В общем виде база данных не является участником взаимодействия между сервисами, а лишь элементом реализации микросервиса. Это позволяет менять компоненты сервиса без изменения его API [1].

С приходом микросервисной архитектуры появились и новые проблемы. Изолированность данных не позволяет организовать ACID (atomicity, consistency, isolation, durability) транзакции в одной базе данных. Для обхода данной ситуации был разработан алгоритм межсервисной коммуникации 2PC. Алгоритм предполагает создание распределенной транзакции, которая сначала блокирует данные, а затем производит выполнение изменений. Помимо недостатка в ограничении выбора протокола коммуникации, 2PC-протоколы значительно снижают допустимую нагрузку в распределенных системах с большим количеством участников, так как ограничивают доступ к данным во время выполнения транзакции [2]. По этой причине приложения часто разрабатываются с использованием подходов итоговой согласованности данных, вместо строгой согласованности данных.

Одним из вариантов реализации такого подхода является Сага — шаблон взаимодействия микросервисов. Сага представляет собой набор команд сервисам и набор компенсаций. Команды — это запросы на выполнение определенной логики в конкретном сервисе. Компенсация — запрос сервису выполнить действие, отменяющее исполненную ранее команду. Компенсации выполняются в случае, если при исполнении одной из команд произошла ошибка [3].

Саги могут быть определены:

— в хореографической форме, т. е. непосредственно в сервисах-участниках. Такой подход более простой в реализации, однако усложняет модификацию, так как один алгоритм описан сразу в нескольких кодовых базах;
— в оркестрационной форме, когда в системе создается отдельный сервис — координатор, в котором описана процедура выполнения Саги.

Сервис производит отправку команд и компенсаций сервисам-участникам Саги [4]. Наличие координатора усложняет процедуру имплементации Саги, но упрощает расширение компонентов системы, оставляя их независимыми [5].

При отправке команд и компенсаций координатору необходимо получение подтверждений об успешном выполнении направленных запросов для исключения коллизий. В противном случае подконтрольный сервис, развернутый в виде нескольких экземпляров или имеющий параллельную обработку запросов в одном экземпляре, может начать исполнение команды после того, как была начата отработка компенсации для этой команды.

Необходимость получения подтверждений координатором означает, что последний должен расходовать процессорные ресурсы после того, как команда/компенсация была отправлена, ожидая получения подтверждения. При этом использование таких подходов, как пул потоков или создание системного потока для каждой Саги в координаторе для управления Сагами, может оказаться не эффективным ввиду того, что координатор будет вынужден расходовать процессорное время непосредственно на ожидание ответа от сервиса [6–8].

В настоящей работе рассматривается целесообразность использования принципов асинхронного программирования при реализации координатора для решения проблемы расходования процессорного времени

на ожидание ответа от сервиса. Использование такого подхода подразумевает выполнение Саг в виде функций, исполнение которых может быть отложено, например, при ожидании ответа от сервиса.

Как было показано в [6–8], использование корутин (сопрограмм, связанных по принципу кооперативной многозадачности) для выполнения набора независимых задач на различных платформах и языках позволяет уменьшить время их выполнения.

Основная цель работы — оценка эффективности применения подходов асинхронного программирования при реализации координатора Саг с учетом его специфики.

Инструмент оценки эффективности использования подходов асинхронного программирования — программная симуляция процесса работы координатора на уровне процессора, операционной системы и программы выполнения набора Саг.

Моделирование координатора Саг

Минимальным элементом симуляции является *операция* (*System Operation*). Операция может требовать процессорную обработку или представлять собой независимый от обработчика процесс, такой как выполнение отправленной координатором команды на стороне управляемого сервиса.

Набор операций формирует *задачу* (*Task*). Операции в задаче выстроены в четком порядке и отражают действительный процесс, происходящий в координаторе. Например, задача отправки команды сервису может состоять из операций подготовки и оправки запроса, ожидания ответа и обработки ответа.

Набор задач формирует *Сагу* (*Simple Saga*). В процессе выполнения Саги, формирующие ее задачи выполняются одна за одной в порядке их определения.

Операции, участвующие в разработанной симуляции:

— подготовка и отправка запроса для команды управления или компенсации (требует обработки процессором);
— ожидание ответа (не требует обработки процессором);
— обработка ответа (требует обработки процессором).

Задачи по работе с файловой системой могут содержать операции, не требующие процессорной обработки [9]. Общее время, выделяемое на работу с файловой системой, относительно мало по сравнению с временем работы с сетью, как было показано в [10]. Время ожидания в таких задачах занимает незначительную часть. В связи с этим задачи по работе с файловой системой были включены в группу *обработки ответов*, и возможность оптимизации расхода процессорного времени в этой группе исключена из исследования.

Набор Саг может являться частью *корутины* (*Coroutine Saga*). Каждая корутина выполняет только одну Сагу, если эта Сага в данный момент требует процессорной обработки. Если выполнение текущей Саги перестает требовать процессорной обработки, то ее выполнение откладывается, и она добавляется в очередь планировщика корутины [11].

Потоки. В симуляции каждая Сага обрабатывается в *потоке* (*Kernel Thread*). Процессор тратит время на инициализацию потока перед выполнением Саг(и) и на деинициализацию потока после того, как потоковые ресурсы необходимо освободить [12].

Один *процессор* (*Processor*) обрабатывает только один поток в единицу времени, поэтому при назначении процессору большего числа потоков ($N_{th\text{-}in\text{-}p} > 1$), все потоки, кроме первого, добавляются в пул потоков этого процессора. При $N_{th\text{-}in\text{-}p} > 1$ процессор обрабатывает текущий поток только в течение определенного промежутка времени $T_{timeslice}$ [13]. Если время выполнения текущего потока больше $T_{timeslice}$, то процессор переключается на обработку следующего в очереди потока, а текущий добавляет в очередь [14, 15].

Один тип координатора из оцениваемых в симуляции организует управление Сагами с помощью системных вызовов, обязывающих поток «уступить» (метод *sched_yield*) процессорные мощности. Когда вызывается данный метод, ядро системы возвращает обрабатываемый поток в очередь потоков процессора и приступает к обработке следующего в очереди потока [16].

Приоритезация потоков в пуле исключена из симуляции, так как все потоки, обрабатываемые процессором, имеют одинаковую природу, следовательно, равнозначны в приоритете.

В симуляции также не предусмотрена балансировка нагрузки на процессоры. Таким образом, потоки, назначенные процессору в начале симуляции, закреплены за ним до конца ее проведения. Такое упрощение допущено по причине того, что в процессе симуляции новые Саги не добавляются к инициализированным в самом начале. С увеличением количества Саг, инициализированных в симуляции, влияние существования такого упрощения сходит на нет, так как Саги распределяются по процессорам равномерно.

Каждая симуляция выполняется в следующем порядке:

— инициализация Саг: симуляция производит генерацию произвольно составленных Саг и сохраняет их в файл для возможности повторения симуляции;
— часть сгенерированных Саг N_{saga} поступает на вход координатора, инициализированному на определенном количестве доступных ему процессоров N_{proc};
— координатор производит обработку переданных Саг, составляя отчет о выполнении со следующими параметрами:
  1) длительность симуляции T_{sim}, с;
  2) утилизация процессорного времени U_p, %;
  3) дополнительные параметры, не используемые в текущем исследовании, такие как процент траты процессорного времени на ожидание, время выполнения системных операций процессором и другие.

Параметры пп. 2 и 3 измеряются как средние по процессорам в течении симуляции.

— *fixedpool* — координатор, управляющий фиксированным пулом потоков, инициализирует потоки в количестве N_{th}, равном N_{proc} ($N_{th\text{-}in\text{-}p} \leq 1$). При освобождении потока в пуле (завершение обрабатываемой Саги), такой координатор начинает выполнение

следующей доступной Саги в освободившемся потоке;
— *overloaded* — координатор, работающий в перегруженном режиме, создает новый поток для каждой Саги. При достижении $N_{th\text{-}in\text{-}p} \geq 1$ процессор будет переключаться между потоками, если обработка текущего потока занимает больше $T_{timeslice}$;
— *coroutines* — координатор, управляющий корутинами, инициализирует набор корутин на старте, формируя их в количестве $N_{coroutine} = N_{proc}$. Все Саги равномерно распределяются по созданным корутинам;
— *yielding* — координатор, в котором происходит «уступание» процессорного времени. Активный поток перестает быть активным сразу после отправки команды/компенсации. Этот тип координатора обладает свойствами *overloaded* версии: переключение между потоками при $N_{th\text{-}in\text{-}p} \geq 1$ после обработки активного потока дольше $T_{timeslice}$.

Симуляция выполняется путем обработки каждого перечисленного элемента симуляции (операции, процессоры, и др.) в минимальную единицу времени (микросекунда). После обработки каждого элемента текущее значение времени симуляции инкрементируется, и процесс обработки начинается сначала. Так происходит до тех пор, пока все Саги в системе не будут выполнены.

Воздействие на некоторые элементы в системе может быть произведено из разных источников. Так, например, операция ожидания выполняется без участия процессора, если предшествующая ей операция отправки запроса была успешно завершена. Как показано на рис. 1, процессор может продолжать производить работу в потоке, который ожидает ответа от управляемого сервиса.

Здесь система управления симуляцией (Simulation control) формирует тактовый сигнал для процессоров (Tick) и сигнал ожидания (Wait) для операций, не требующих процессорной обработки. Так как сигналы Tick и Wait представляют собой разный тип изменений в один момент времени, важно избежать наложения изменений, которые происходят при их получении, поэтому одномоментные сигналы имеют одинаковый идентификатор времени (*A*).

Результаты

Набор симуляций для четырех типов координаторов выполнен на следующих наборах входных данных:
— N_{proc}: 4, 8, 20, 40, 80;
— N_{saga}: 50, 100, 200, 500, 700, 1000, 1500.

Результаты выполнения симуляций представлены на рис. 2.

В таблице представлены пики ключевых характеристик по результатам симуляций.

Интерпретация результатов исследования

Сравнение всех рассмотренных типов координаторов представлено на рис. 3.

Для детального сравнения типов координаторов каждый рассмотрен в отношении времени выполнения всех Саг и утилизации процессорного времени к этим характеристикам у *coroutines* координатора. *Coroutines* координатор взят за основу, так как он показывает наибольшую эффективность в наибольшем количестве симуляций.

Fixedpool координатор показывает наихудший результат как по временным характеристикам, так и по проценту загрузки процессоров во время симуляции. Как показано на рис. 4, время выполнения 1500 Саг на четырех процессорах может занимать в 40 раз больше времени для *fixedpool* координатора по сравнению с *coroutines* координатором. А показатель утилизации для приведенных входных параметров у *fixedpool* координатора опускается почти до 2 % по отношению к значению *overloaded* координатора.

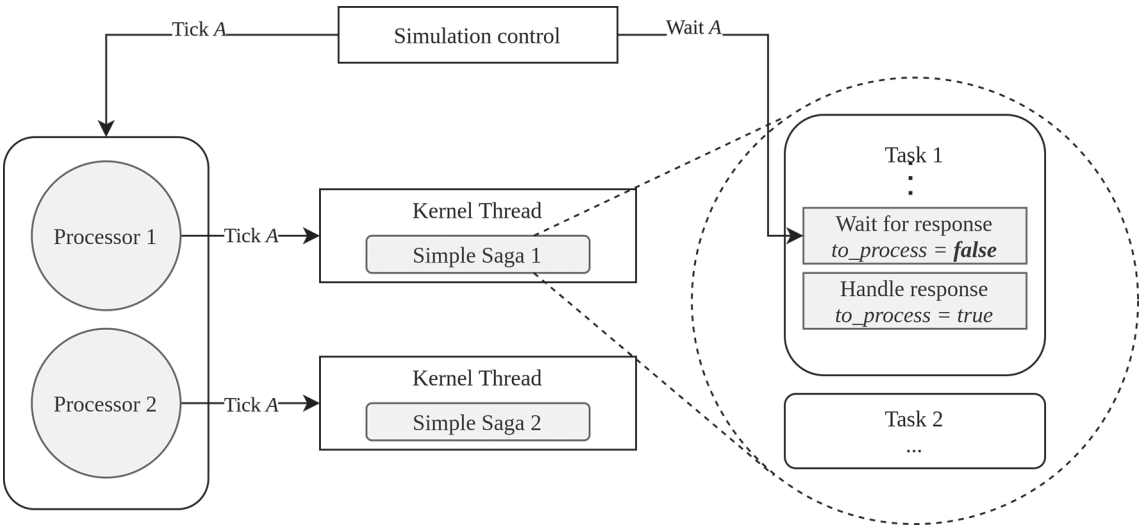

Рис. 1. Упрощенная схема выполнения одного шага симуляции для координатора с фиксированным набором потоков, когда текущая задача одного из потоков не требует процессорной обработки

Fig. 1. A simplified execution flow of a simulation step for a fixed thread pool coordinator when the current task of a thread does not require processing

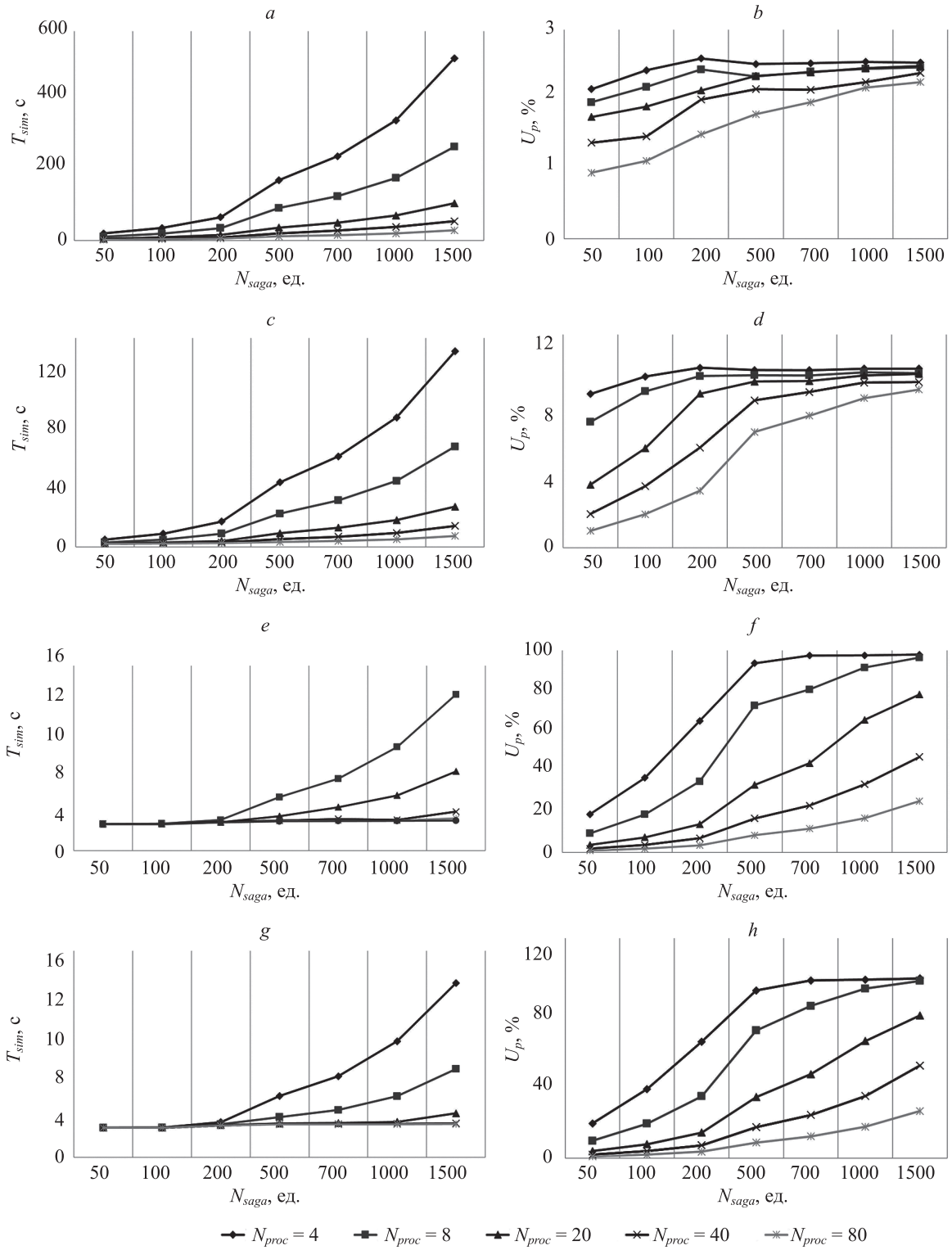

Рис. 2. Время выполнения симуляций T_{sim} и показателя утилизации U_p для координаторов: *fixedpool* (*a*, *b*); *overloaded* (*c*, *d*); *coroutines* (*e*, *f*); *yielding* (*g*, *h*)

Fig. 2. Simulation duration and utilisation value of the coordinators: *fixedpool* (*a*, *b*); *overloaded* (*c*, *d*); *coroutines* (*e*, *f*); *yielding* (*g*, *h*)

Overloaded координатор показывает бóльшую производительность, опережая *fixedpool* координатор по времени выполнения симуляции на всех значениях входных параметров. Максимальным значением утилизации *overloaded* координатора является 10,14 %, что почти в 4 раза превосходит аналогичный показатель *fixedpool* координатора.

Fixedpool и *overloaded* (рис. 5) координаторы имеют относительно низкую утилизацию процессорных ресурсов и высокую длительность выполнения Саг в

Таблица. Максимальные значения T_{sim} и U_p и соответствующие им значения входных параметров для каждого из типов координаторов

Table. T_{sim} и U_p max values and corresponding values of input parameters for each coordinator type

Тип координатора	max(T_{sim}), с	N_{saga}, ед.	N_{proc}, ед.	max(U_p), %	N_{saga}, ед.	N_{proc}, ед.
fixedpool	522,50	1500	4	2,59	200	4
overloaded	131,50	1500	4	10,14	200	4
coroutines	13,51	1500	4	97,91	1500	4
yielding	13,49	1500	4	98,10	1500	4

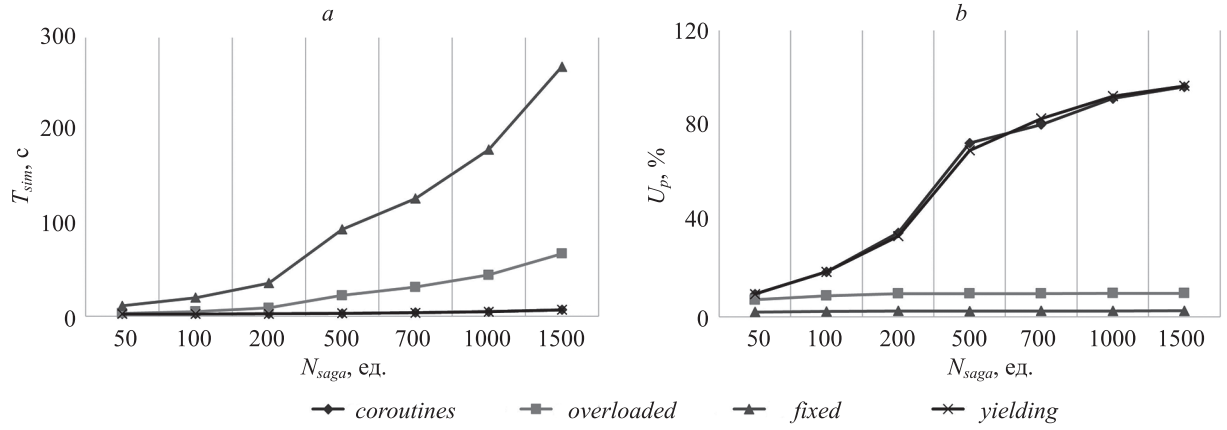

Рис. 3. Время выполнения симуляций (*a*) и показатель утилизации (*b*) всех типов координаторов при N_{proc} = 8

Fig. 3. Simulation duration (*a*) and utilisation value (*b*) for each coordinator type when N_{proc} = 8

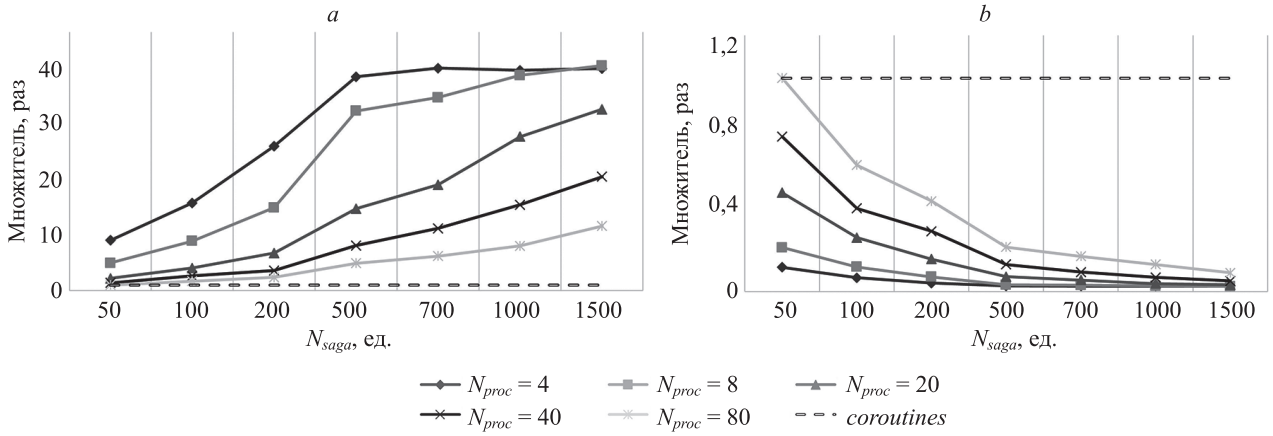

Рис. 4. Отношение длительности симуляции (*a*) и процента утилизации процессорного времени (*b*) *fixedpool* координатора к аналогичным характеристикам *coroutines* координатора

Fig. 4. Simulation duration ratio (*a*) and processor time utilisation ratio (*b*) of the *fixedpool* coordinator to the corresponding characteristics of the *coroutines* coordinator

симуляции. Причиной является расходование процессорных ресурсов на ожидание подтверждений выполнения команд/компенсаций от сервисов.

Overloaded координатор сменяет активный поток процессора при достижении времени работы над процессом равному $T_{timeslice}$, что увеличивает утилизацию процессорного времени на 7,54 % в максимальном значении. Однако такая смена не гарантирует своевременного переключения процессора как с потока, перешедшего в режим ожидания, так и переключения на поток, требующего процессорной обработки.

Таким образом, итоговое значение утилизации остается низким: 10,14 %.

Итоговые показатели *yielding* и *coroutines* координаторов практически совпадают (рис. 6). Максимальное значение выполнения симуляции таких координаторов в 9,74 раза меньше максимального времени выполнения всех Саг в *overloaded* координаторе и в 38,74 раза меньше максимального времени выполнения в *fixedpool* координаторе.

Yielding координатор, по сравнению с *coroutines* координатором, имеет дополнительную трату процес-

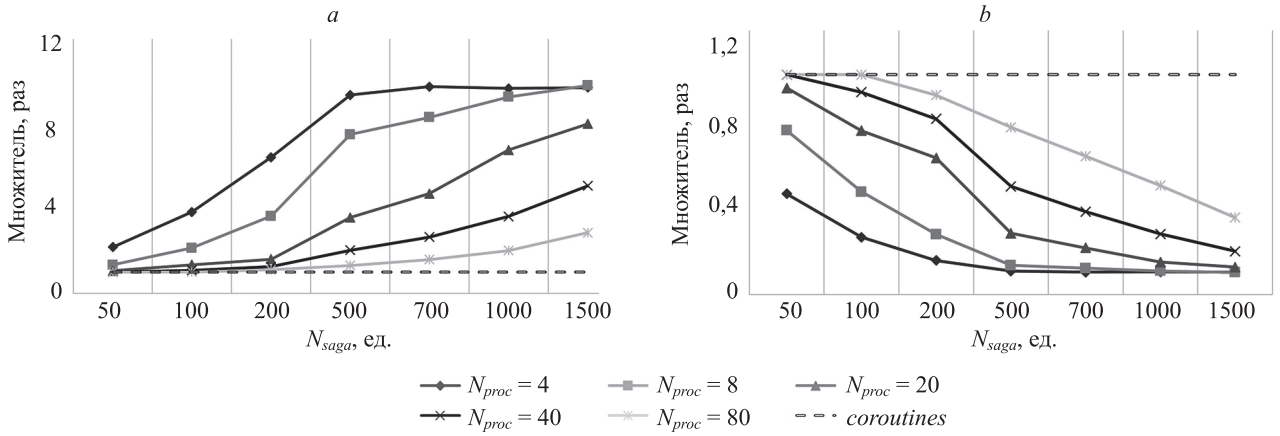

Рис. 5. Отношение длительности симуляции (*a*) и процента утилизации процессорного времени (*b*) *overloaded* координатора к аналогичным характеристикам *coroutines* координатора

Fig. 5. Simulation duration ratio (*a*) and processor time utilisation ratio (*b*) of the *overloaded* coordinator to the corresponding characteristics of the *coroutines* coordinator

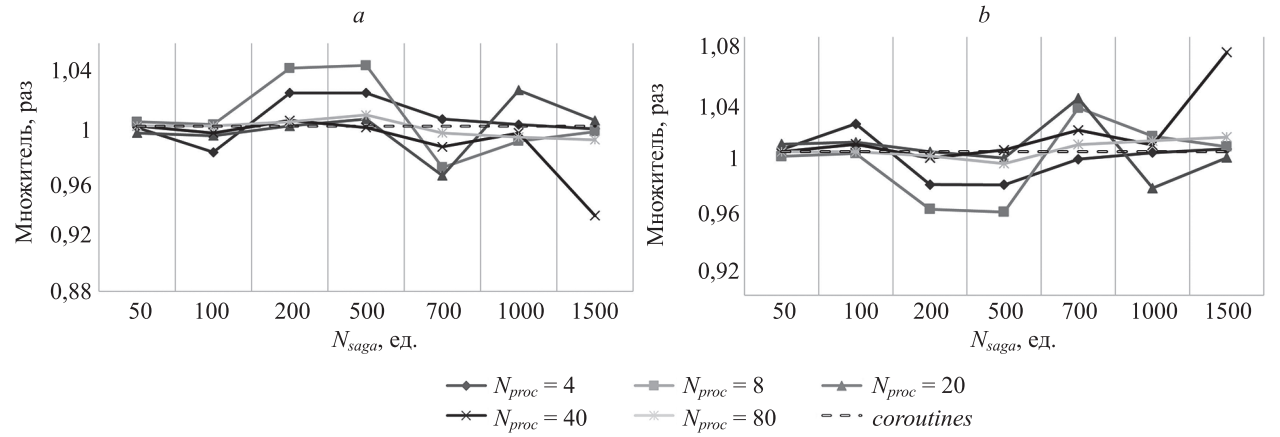

Рис. 6. Отношение длительности симуляции (*a*) и процента утилизации процессорного времени (*b*) *yielding* координатора к аналогичным характеристикам *coroutines* координатора

Fig. 6. Simulation duration ratio (*a*) and processor time utilisation ratio (*b*) of the *yielding* coordinator to the corresponding characteristics of the *coroutines* coordinator

сорного времени, связанную с выделением и освобождением оперативной памяти при переключении между потоками. Такой координатор, как и координатор в перегруженном потоками режиме, переключает активный поток каждые $T_{timeslice}$ работы над одним потоком.

Несмотря на разницу в процессе управления Сагами, обе системы позволяют избежать расхода процессорного времени на ожидание подтверждения команды/компенсации. Итоговые значения для *yielding* и *coroutines* координаторов практически совпадают. Учитывая относительно большой промежуток времени для получения подтверждения, который может превосходить время подготовки запроса и анализа подтверждения в 7–50 раз, возможность переключить «внимание» процессора на другую задачу дает прирост в утилизации процессора на 88 %.

Заключение

В работе представлены результаты симуляции работы координаторов Car, реализующих различные способы организации многопоточного выполнения. Проведенная симуляция с различными значениями количества Car и с различным числом доступных координатору процессоров позволила определить численные значения времени выполнения набора Car и значения утилизации процессорного времени четырех видов координаторов в зависимости от входных параметров, что позволяет сравнить эффективность работы типов координаторов.

Показан значительный прирост к скорости выполнения набора Car (до 9,74 раз) и к значению утилизации процессорного времени (до 88 %) при использовании асинхронного подхода в алгоритме работы координаторов.

Определено, что разница между асинхронными конфигурациями координаторов, построенных на корутинах или использующих планировщик ядра Linux, является несущественной. Показано, что для организации эффективного координатора не так критична конкретная реализация асинхронного подхода, как его общее использование. Важно отметить, что в данном

исследовании не было уделено внимание вопросам затрат памяти, что может являться дополнительным фактором при выборе оптимального асинхронного подхода при решении конкретных задач.

Так как реализованная симуляция показала действительный прирост показателей эффективности при использовании асинхронного подхода, необходимо дальнейшее изучение данного вопроса на основе исследования действительных реализаций рассмотренных систем для получения более точной разницы в показателях эффективности.

Литература

1. Brown K., Woolf B. Implementation patterns for microservices architecture // Proc. 23rd Conference on Pattern Languages of Programs (PLoP'16). 2016. P 1–35.
2. Dürr K., Lichtenthäler R., Wirtz G. An evaluation of saga pattern implementation technologies // CEUR Workshop Proceedings. 2021. V. 2839. P. 74–82.
3. Štefanko M., Chaloupka O., Rossi B. The saga pattern in a reactive microservices environment // Proc. 14th International Conference on Software Technologies (ICSOFT). 2019. P. 483–490. https://doi.org/10.5220/0007918704830490
4. Rudrabhatla C. Comparison of event choreography and orchestration techniques in microservice architecture // International Journal of Advanced Computer Science and Applications. 2018. V. 9. N 8. P. 18–22. https://doi.org/10.14569/IJACSA.2018.090804
5. Richardson C. Microservices Patterns: With examples in Java. NY: Manning Publications, 2018. 520 p.
6. Knoche H. Improving batch performance when migrating to microservices with chunking and coroutines // Softwaretechnik-Trends. 2019. V. 39. N 4. P. 20–22.
7. Stadler L., Würthinger T., Wimmer C. Efficient coroutines for the Java platform // Proc. 8th International Conference on the Principles and Practice of Programming in Java, (PPPJ 2010). Vienna, Austria. 2010. P. 20–28. https://doi.org/10.1145/1852761.1852765
8. Belson B., Holdsworth J., Xiang W., Philippa B. A Survey of asynchronous programming using coroutines in the Internet of Things and embedded systems // ACM Transactions on Embedded Computing Systems. 2019. V. 18. N 3. P. 21. https://doi.org/10.1145/3319618
9. Lee G., Shin S., Song W., Ham T., Lee J., Jeong J. Asynchronous I/O stack: a low-latency kernel I/O stack for ultra-low latency SSDs // Proc. of the 2019 USENIX Annual Technical Conference. 2019. P. 603–616.
10. Malyuga K., Perl O., Slapoguzov A., Perl I. Fault tolerant central saga orchestrator in RESTful architecture // Proc. 26th Conference of Open Innovations Association (FRUCT). Yaroslavl, Russia. 2020. P. 278–283. https://doi.org/10.23919/FRUCT48808.2020.9087389
11. Psaropoulos G., Legler T., May N., Ailamaki A. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins // VLDB Journal. 2019. V. 28. N 4. P. 451–471. https://doi.org/10.1007/s00778-018-0533-6
12. Sung M., Kim S., Park S., Chang N., Shin H. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread // Information Processing Letters. 2002. V. 84. N 4. P. 221–225. https://doi.org/10.1016/S0020-0190(02)00286-7
13. Lozi J.-P., Lepers B., Funston J., Gaud F., Quéma V., Fedorova A. The Linux scheduler: a decade of wasted cores // Proc. 11th European Conference on Computer Systems (EuroSys'16). 2016. P. 2901326. https://doi.org/10.1145/2901318.2901326
14. David F., Carlyle J., Campbell R. Context switch overheads for Linux on ARM platforms // Proc. of the 2007 Workshop on Experimental Computer Science. 2007. P. 3. https://doi.org/10.1145/1281700.1281703
15. Ling Y., Mullen T., Lin X. Analysis of optimal thread pool size // ACM SIGOPS Operating Systems Review. 2000. V. 34. N 2. P. 42–55. https://doi.org/10.1145/346152.346320
16. Nadgowda S., Suneja S., Isci C. Paracloud: Bringing Application Insight into Cloud Operations // Proc. 9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, co-located with USENIX ATC 2017. 2017. P. 1–8.

References

1. Brown K., Woolf B. Implementation patterns for microservices architecture. *Proc. 23rd Conference on Pattern Languages of Programs (PLoP '16)*, 2016, pp. 1–35.
2. Dürr K., Lichtenthäler R., Wirtz G. An evaluation of saga pattern implementation technologies. *CEUR Workshop Proceedings*, 2021, vol. 2839, pp. 74–82.
3. Štefanko M., Chaloupka O., Rossi B. The saga pattern in a reactive microservices environment. *Proc. 14th International Conference on Software Technologies (ICSOFT)*, 2019, pp. 483–490. https://doi.org/10.5220/0007918704830490
4. Rudrabhatla C. Comparison of event choreography and orchestration techniques in microservice architecture. *International Journal of Advanced Computer Science and Applications*, 2018, vol. 9, no. 8, pp. 18–22. https://doi.org/10.14569/IJACSA.2018.090804
5. Richardson C. *Microservices Patterns: With examples in Java*. NY, Manning Publications, 2018, 520 p.
6. Knoche H. Improving batch performance when migrating to microservices with chunking and coroutines. *Softwaretechnik-Trends*, 2019, vol. 39, no. 4, pp. 20–22.
7. Stadler L., Würthinger T., Wimmer C. Efficient coroutines for the Java platform. *Proc. 8th International Conference on the Principles and Practice of Programming in Java, (PPPJ 2010)*, Vienna, Austria, 2010, pp. 20–28. https://doi.org/10.1145/1852761.1852765
8. Belson B., Holdsworth J., Xiang W., Philippa B. A Survey of asynchronous programming using coroutines in the Internet of Things and embedded systems. *ACM Transactions on Embedded Computing Systems*, 2019, vol. 18, no. 3, pp. 21. https://doi.org/10.1145/3319618
9. Lee G., Shin S., Song W., Ham T., Lee J., Jeong J. Asynchronous I/O stack: a low-latency kernel I/O stack for ultra-low latency SSDs. *Proc. of the 2019 USENIX Annual Technical Conference*, 2019, pp. 603–616.
10. Malyuga K., Perl O., Slapoguzov A., Perl I. Fault tolerant central saga orchestrator in RESTful architecture. *Proc. 26th Conference of Open Innovations Association (FRUCT)*, Yaroslavl, Russia, 2020, pp. 278–283. https://doi.org/10.23919/FRUCT48808.2020.9087389
11. Psaropoulos G., Legler T., May N., Ailamaki A. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *VLDB Journal*, 2019, vol. 28, no. 4, pp. 451–471. https://doi.org/10.1007/s00778-018-0533-6
12. Sung M., Kim S., Park S., Chang N., Shin H. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread. *Information Processing Letters*, 2002, vol. 84, no. 4, pp. 221–225. https://doi.org/10.1016/S0020-0190(02)00286-7
13. Lozi J.-P., Lepers B., Funston J., Gaud F., Quéma V., Fedorova A. The Linux scheduler: a decade of wasted cores. *Proc. 11th European Conference on Computer Systems (EuroSys'16)*, 2016, pp. 2901326. https://doi.org/10.1145/2901318.2901326
14. David F., Carlyle J., Campbell R. Context switch overheads for Linux on ARM platforms. *Proc. of the 2007 Workshop on Experimental Computer Science*, 2007, pp. 3. https://doi.org/10.1145/1281700.1281703
15. Ling Y., Mullen T., Lin X. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review*, 2000, vol. 34, no. 2, pp. 42–55. https://doi.org/10.1145/346152.346320
16. Nadgowda S., Suneja S., Isci C. Paracloud: Bringing Application Insight into Cloud Operations. *Proc. 9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2017, co-located with USENIX ATC 2017*, 2017, pp. 1–8.

Авторы

Малюга Константин Владимирович — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, SC 57216858413, https://orcid.org/0000-0001-7381-2067, konstantin.malyuga@gmail.com

Перл Иван Андреевич — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, SC 56830509800, https://orcid.org/0000-0002-8903-405X, ivan.perl@itmo.ru

Слапогузов Александр Петрович — аспирант, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, SC 57216863398, https://orcid.org/0000-0003-0699-5478, slapoguzov@gmail.com

Authors

Konstantin V. Malyuga — Postgraduate, ITMO University, Saint Petersburg, 197101, Russian Federation, SC 57216858413, https://orcid.org/0000-0001-7381-2067, konstantin.malyuga@gmail.com

Ivan A. Perl — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, SC 56830509800, https://orcid.org/0000-0002-8903-405X, ivan.perl@itmo.ru

Aleksandr P. Slapoguzov — Postgraduate, ITMO University, Saint Petersburg, 197101, Russian Federation, SC 57216863398, https://orcid.org/0000-0003-0699-5478, slapoguzov@gmail.com

Статья поступила в редакцию 17.04.2021
Одобрена после рецензирования 25.05.2021
Принята к печати 16.07.2021

Received 17.04.2021
Approved after reviewing 25.05.2021
Accepted 16.07.2021