Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Голованов Сергей Александрович

Эффективное параллельное моделирование зависимых симуляционных объектов

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Лесин В. М.

Рецензент:
Смаль А. В.

Санкт-Петербург
2017

Оглавление

Введение

Важной составляющей современной авиации является обеспечение безопасности полетов. Помимо профессионализма пилотов и надежности воздушных судов, особую роль в ней занимает такая профессия как авиадиспетчер. Главная функция авиадиспетчера — обеспечение безопасного, упорядоченного и регулярного движения воздушных судов в пределах установленной для него зоны ответственности. Он информирует экипажи об обстановке по маршруту полета и на аэродромах, предоставляет метеорологическую информацию, поддерживает радиосвязь с пилотами в соответствии с правилами и фразеологией радиообмена. Именно авиадиспетчер принимает меры для поддержания безопасности при возникновении сложной ситуации в воздухе и особых случаях полета путем обеспечения безопасных интервалов продольного, вертикального и бокового эшелонирования [14]. Данная работа сопряжена с большой ответственностью, поэтому к профессионализму людей здесь предъявляются достаточно высокие требования. Получить необходимые навыки помогают специализированные промышленные тренажерные системы, направленные на обучение авиадиспетчеров.

Процесс обучения авиадиспетчера на тренажере происходит следующим образом. Используя некоторый сценарий (упражнение), ученику предоставляется картина воздушной обстановки заданной области. Это может осуществляться, например, при помощи имитации авиационного радара. Основываясь на ней, он отдает, при необходимости, голосовые команды. Псевдо-пилот (или несколько псевдо-пилотов) принимает эти команды и управляет направлением движения воздушных судов в сценарии. В качестве псевдо-пилота, вместо человека, может выступать специализированная система, способная распознавать речь и интерпретировать команды. Инструктор же в это время следит за действиями учащегося и способен вносить изменения в упражнение по мере его прохождения. Так авиадиспетчер без серьезных затрат может получить необходимые ему практические навыки управления воздушными судами и рассмотреть сложные ситуации, нечасто встречающиеся в жизни.

Прежде чем воспроизвести сценарий тренажером, его необходимо создать. Обычно это делается заданием плана полета для каждого воздушного судна отдельно. Однако, подход, который использует тренажерная система SimLabs, отличается. При разработке сценария пользователь прежде всего генерирует некоторое его приближение с необходимыми параметрами. Например, задается нагруженность сценария воздушными судами, особенности использования аэропорта. Далее он уже вручную изменяет характеристики упражнения: добавляет или удаляет различные объекты, изменяет планы полетов и т. д. При этом пользователь постоянно перематывает сценарий в различные точки времени для того, чтобы оценить результаты внесенных ранее изменений. Это позволяет достаточно быстро и качественно задавать необходимые для обучения события. Для эффективной интерактивной разработки сценария

такая перемотка должна производится максимально быстро.

Перемещения во времени в сценарии часто требуют производить моделирование системы объектов, задействованных в нем. Это может занимать достаточно много времени. При ускорении вычислений с использованием многопоточности возникают две проблемы, серьезно влияющие на производительность: синхронизация взаимодействия объектов и балансировка нагрузки между потоками. В данной работе рассматриваются возможные варианты устранения этих проблем и предлагается решение, позволяющее эффективно производить полное и частичное моделирование применительно к системе объектов тренажерных продуктов SimLabs.

В главе 1 более подробно рассматриваются особенности используемой в тренажерных продуктах SimLabs модели, а также ситуации, когда происходит ее вычисление. Там же ставится основная цель и задачи, которые необходимо решить для ее достижения. В главе 2 рассматриваются библиотеки, предоставляющие инструментарий для моделирования систем, схожих с рассматриваемой в данной работе. Далее вводится проблема синхронизации объектов, описываются и сравниваются основные подходы ее решения: оптимистичная и консервативная синхронизации. Также в этой главе рассматриваются возможность применения пула потоков и job system, проблема балансировки нагрузки и такие стратегии как work sharing, work stealing, общая очередь. После описывается предложенное решение и обосновываются основные идеи, применяемые в нем. Глава 3 посвящена тестированию решения. Там описывается методология тестирования и анализируется влияние параметров предложенного решения на производительность.

1. Постановка и анализ задачи

1.1. Анализ предметной области

В тренажерных продуктах SimLabs уже имеется реализация объектов моделируемых процессов, а также необходимых для них вычислений, которые осуществляются в одном потоке. Используемая модель имеет следующие особенности:

1. Модель включает в себя совокупность объектов различных типов, например, воздушные суда, зоны опасных метеоявлений, тягачи на земле.

2. Новые объекты могут появляться в системе динамически, т. е. по некоторому условию, которое обычно зависит от текущей конфигурации состояний объектов, или же принудительно со стороны пользователя. Примером нового объекта может служить дополнительное воздушное судно, добавленное пользователем вручную. Также возможно удаление уже существующих объектов, например, когда воздушное судно уходит из моделируемой области пространства.

3. В системе может находиться до тысячи объектов, которые изменяют свои состояния во времени, например, тягачи могут передвигаться, а зоны опасных метеоявлений изменяться в размерах.

4. Объекты могут конфликтовать между собой, согласовывая дальнейшие действия исходя из сложившейся ситуации. Обнаружение конфликтов происходит периодически с неизменным интервалом. После обнаружения конфликта могут следовать действия по его устранению. Так, например, воздушные суда, выяснив, что они достаточно близко сблизились, должны скорректировать свой курс.

Данную модель можно считать дискретно-событийной системой [12]. Время в ней является дискретным, на каждом временном шаге происходит обновление состояний всех объектов, которые определяют новые события - конфликты между объектами и действия по их устранению, т.е. указания для объектов. Моделирование состояния каждого объекта - это отдельный логический процесс.

Вычисление модели происходит в трех различных по своим особенностям режимах работы тренажерной системы:

1. Воспроизведение сценария. В нем вычисляется модель с заданной конфигурацией и результат воспроизводится для пользователя, возможно с ускорением. В данном режиме не ставится жестких требований к скорости моделирования. Однако, важно, чтобы воспроизведение производилось без задержек. При этом пользователь может вручную вносить изменения в модель в реальном времени, например, изменять направление движения воздушных судов.

2. Перемотка вперед. Она используется когда необходимо получить конфигурацию всех объектов в момент времени в будущем. При этом история изменений состояний сохраняется и в дальнейшем используется для просмотра модели в ранние моменты времени. При перемотке очень важна скорость производимых вычислений. Время обновления одного объекта достаточно мало, в среднем единицы микросекунд, но имеет редкие выбросы, которые могут стократно превышать среднее значение. Это вызвано особенностями реализации, например, при вычислении траектории движения некоторых объектов используется кэширование. В однопоточной реализации на перемотку всей модели с тысячей объектов (половина из которых находится на земле, а другая в воздухе) на час вперед может потребоваться до 200 секунд.

3. Сохранение текущей истории. После перехода на некоторое время вперед в сценарии пользователь может вернуться обратно и начать менять модель в ранние моменты времени. Это достаточно частое поведение. Пользователь что-то изменяет, просматривает результат своих действий в сценарии в различные моменты времени, возвращается обратно и продолжает редактирование. Обычно изменения затрагивают часть объектов, состояния которых уже вычислены на какой-то промежуток времени. Поэтому возникает необходимость обновить историю модели с учетом внесенных правок. В этом режиме, как и в перемотке вперед, важна скорость моделирования. В текущей реализации он реализован через полное перевычисление состояний всех объектов.

Однопоточная реализация вычислений в рассматриваемой модели не обеспечивает желаемой скорости получения результатов моделирования и не позволяет в полной мере использовать все преимущества современных многоядерных процессоров. Вполне естественным решением является использование нескольких потоков при моделировании. Однако, применение многопоточности затруднено тем, что объекты в модели взаимодействуют. Это вызывает необходимость использования примитивов синхронизации и неминуемо ведет к увеличению доли последовательно исполняемого кода. При этом может значительно снизиться ускорение S_p при вычислении модели в p потоках. Ускорение определяется следующим образом:

$$S_p = \frac{T_1}{T_p}, \tag{1}$$

где T_p - время вычисления задачи при потоках, T_1 - время выполнения при одном потоке. Теоретическую верхнюю границу ускорения дает закон Амдала. Он описывается следующей математической формулой:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{2}$$

где p - количество процессоров (ядер), α - доля последовательно исполняемого кода ($\alpha \neq 0$).

На практике чем выше эффективность параллелизации $E = \frac{S_p}{p}$, тем лучше. Нет строгой границы, разделяющей низкую и высокую эффективность и позволяющей оценить абсолютное качество решения. Поэтому, для большей конкретики, в данной работе будем считать приемлемой эффективность порядка 80%.

1.2. Цели и задачи

Целью данной работы является разработка технологического решения, которое позволит эффективно параллельно моделировать на однопроцессорной машине систему объектов, схожую с используемой в тренажерных продуктах SimLabs, с учетом их взаимодействия (конфликтов). При этом необходимо предусмотреть возможность работы решения в различных режимах вычислений.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Проанализировать существующие подходы к параллельному моделированию подобных систем
2. Разработать схему параллельного моделирования
3. Реализовать прототип и протестировать его производительность

В дальнейшем, реализованный прототип необходимо будет перенести в тренажерные продукты SimLabs.

2. Моделирование зависимых симуляционных объектов

2.1. Обзор существующих решений

Разработки в области параллельного дискретно-событийного моделирования продолжаются уже более тридцати лет. И на текущий момент существуют библиотеки, реализующие необходимый инструментарий для описания дискретно-событийных систем и их параллельного моделирования, в том числе и для C++, основного языка программирования данной работы. Часть из них ориентирована на распределенные вычисления и имеет достаточно узкую направленность, например, OMNet++ и OverSim. Они предназначены для моделирования интернет сетей. Остальные же, например ROOT-Sim, WARPED, используют оптимистичную стратегию синхронизации взаимодействия логических процессов. Данный тип синхронизации при текущей реализации модели состояний объектов способен вызвать серьезные накладные расходы при моделировании. Отсутствие инструментов со вторым типом синхронизации - консервативным, объясняется жесткой зависимостью производительности моделирования от конкретной моделируемой системы. Оба типа синхронизации далее будут разобраны подробнее.

2.2. Синхронизация логических процессов

Обычно дискретно-событийную систему рассматривают как совокупность некоторого количества логических процессов (ЛП), которые взаимодействуют между собой. Взаимодействие осуществляется путем обмена сообщениями с отметками времени между конкретными ЛП. Вычисления, выполняемые каждым ЛП, представляют собой обработку последовательности событий, где каждое такое событие может изменять переменные состояния и (или) планировать новые события для себя и других ЛП. При этом каждый ЛП должен обрабатывать все свои события, как сгенерированные локально, так и сгенерированные другими ЛП, таким образом, чтобы не допустить ошибок причинно-следственных связей в системе. Другими словами, ЛП не должен допускать ситуаций, когда результат события (1), происходящего раньше некоторого другого события (2), меняется из-за того, что оно (1) было обработано позже события (2). При параллельном исполнении ЛП это одна из основных проблем, называемая проблемой синхронизации.

Механизм синхронизации позволяет решить данную проблему. Он отвечает за правильное взаимодействие между ЛП и обеспечивает корректный результат моделирования системы, без ошибок причинно-следственных связей. При этом синхронизация создает дополнительные накладные расходы. Поэтому эффективность и скорость па-

раллельного моделирования сильно зависит от типа и качества механизма синхронизации.

Выделяют два основных подхода к синхронизации: консервативную синхронизацию и оптимистичную синхронизацию.

2.2.1. Консервативная синхронизация

Консервативная синхронизация не допускает нарушения корректного состояния системы, т. е. ошибок причинно-следственных связей. Главная задача любого консервативного алгоритма - определить, когда безопасно обрабатывать событие. Событие же считается безопасным, когда можно гарантировать, что никакое другое событие, содержащее меньшую отметку времени, позже не будет запланировано для этого ЛП. При этом алгоритм не позволяет ЛП обрабатывать событие, пока оно не будет гарантированно безопасным.

В основе большинства консервативных алгоритмов синхронизации лежит вычисление для каждого ЛП нижней границы времени будущих событий (НГС), которые впоследствии могут быть запланированы для него другими ЛП. Такая граница позволяет определять, какие события безопасны для обработки. Т. к. для ЛП не может быть запланировано событие со временем меньшем чем НГС, то все события этого ЛП до НГС могут быть обработаны в правильном порядке. При этом важным понятием здесь является понятие интервала предвидения (lookahead). Если ЛП находится в момент моделирования в точке времени T, и он может гарантировать, что любое сообщение, которое он отправит в будущем, будет иметь метку времени не меньше $T+L$, независимо от того, какие сообщения он может позже принять, ЛП, как говорят, имеет интервал предвидения L. Таким образом, НГС для ЛП можно определить как наименьший интервал предвидения среди ЛП, связанных с данным, т. е. тех, кто может ему послать сообщение. События же, находящиеся до НГС, могут быть обработаны ЛП. Величина интервала предвидения тесно связана с конкретной моделируемой системой и с увеличением значения положительно сказывается на производительности параллельного моделирования [3, 4, 1].

Нижняя граница будущих событий может быть получена при помощи обмена сообщениями между ЛП, например, алгоритм CMB (Chandy, Misra, Bryant) [8], или же при помощи барьеров, например, централизованный барьер, древовидный барьер и т.д. [5].

2.2.2. Оптимистичная синхронизация

В отличие от консервативных подходов, которые не допускают ошибок причинно-следственных связей, оптимистичные алгоритмы синхронизации ошибки допускают, но способны обнаруживать их и восстанавливать корректное состояние системы. Об-

работка событий осуществляется без блокировок. При этом ошибочные состояния находятся и исправляются во время выполнения. Когда для ЛП планируется событие с отметкой времени меньше одного или нескольких событий, которые он уже обработал, он откатывается назад и обрабатывает эти события в правильном порядке. Откат события включает в себя восстановление состояния ЛП, которое существовало до обработки события (для этого состояния и события предварительно сохраняются), и отправку сообщений, вызванных новым событием.

Откат одного ЛП может затронуть связанные с ним ЛП. Для их информирования применяются специальные сообщения, называемые анти-сообщениями. Анти-сообщение - это дубликат ранее отправленного сообщения. Всякий раз, когда анти-сообщение и его согласующее (положительное) сообщение сохраняются в одной очереди, они удаляются (аннулируются). Чтобы отменить сообщение, ЛП нужно отправить только соответствующее анти-сообщение. Если соответствующее положительное сообщение уже обработано, процесс-получатель откатывается назад, возможно, создавая дополнительные анти-сообщения. Используя эту рекурсивную процедуру, все ошибочные эффекты будут в конечном итоге удалены.

В механизме откатов существуют две проблемы, без решения которых он является неприменимым: некоторые вычисления, например операции ввода-вывода, не могут быть обратимы; вычисления будут постоянно потреблять все больше памяти, потому что история (например, сохраненные состояния) должна сохраняться, даже если отката не происходит. Обе эти проблемы решаются при помощи глобального виртуального времени (ГВВ). ГВВ - это нижняя граница временной отметки любого последующего отката. Заметим, что откаты вызваны сообщениями, прибывающими "из прошлого". Следовательно, наименьшая временная отметка среди необработанных и частично обработанных сообщений и анти-сообщений дает значение для ГВВ. Как только ГВВ будет вычислено, операции ввода-вывода, происходящие в виртуальном времени, меньшем, чем ГВВ, могут быть зафиксированы, а ненужная история очищена. Вычисление ГВВ концептуально похоже на вычисление нижней границы будущих событий в консервативных алгоритмах и сводится к вычислению нижней границы временных отметок будущих сообщений или анти-сообщений, которые впоследствии могут быть получены ЛП.

При отсутствии ограничений на вычисления, некоторые ЛП могут уйти достаточно далеко вперед во времени. Это может негативно отразится на потреблении памяти. Дополнительно, в случае, если для такого ЛП потребуется откат, выполненные вычисления придется отбросить. Поэтому, для ЛП иногда вводят ограничения, определяющие, насколько он может опережать другие ЛП. Такое ограничение достигается путем введения временного окна размера W. ЛП не разрешается продвигаться дальше ГВВ более чем на W. Окно передвигается вперед вместе с увеличением ГВВ.

Все протоколы оптимистичной синхронизации включают в себя описанные выше

понятия и дополнительно имеют различные оптимизации по использованию памяти и (или) выполнению откатов [6].

2.2.3. Выбор подхода синхронизации

Выбор подхода синхронизации во многом зависит от текущей моделируемой системы. Консервативная синхронизация из-за ее низких накладных расходов и незначительного использования памяти обычно может обеспечить хорошую производительность, если из системы можно определить достаточно большой интервал предвидения. С другой стороны, оптимистичная синхронизация может использовать большую степень параллелизма посредством спекулятивного исполнения [15]. Однако для этого требуется сохранение состояния, вычисление ГВВ и откаты, что может привести к большим накладным расходам и серьезно усложнить разработку. Дополнительно стоит отметить возможность лавинного отката в оптимистичной синхронизации. Такое явление обычно значительно замедляет моделирование системы.

В качестве третьего подхода иногда рассматривают смешанную синхронизацию - добавление оптимистичности в консервативную синхронизацию или консерватизма в оптимистичную синхронизацию. При этом свойства полученной синхронизации уже напрямую зависят от основной стратегии и особенностей добавленных модификаций.

В силу того, что текущая модель состояний объектов требует достаточно много времени на восстановление в сравнении со временем одного обновления, было решено использовать консервативный подход к синхронизации с добавлением некоторых спекулятивных вычислений.

2.3. Балансировка нагрузки

Дискретно-событийная система, как уже упоминалось ранее, обычно представляется в виде совокупности логических процессов. При параллельном моделировании, когда логические процессы статически распределяются между потоками, может возникнуть дисбаланс в нагрузке потоков. Часть из них будет простаивать из-за отсутствия работы, в то время как у других потоков будут доступные логические процессы, не участвующие в данный момент в вычислениях. Особенно сильно это будет проявляться, когда время обработки событий будет значительно неоднородным.

Наиболее частый подход при распараллеливании программы - это использование пула потоков и выражение вычислений в виде задач. При этом для планирования задач могут использоваться несколько различных стратегий: общая очередь задач, work sharing, work stealing.

При использовании общей очереди в ней содержатся готовые к выполнению задачи. Очередь же разделяется между всеми потоками, которые в процессе своей работы могут доставать из нее задачи, либо добавлять новые. Такой подход прост и обеспе-

чивает балансировку нагрузки (никакой поток не простаивает, если есть доступная задача), но имеет серьезный недостаток. Из-за того что очередь общая для всех потоков, время, затраченное на доступ к ней, может быть значительным по сравнению с самими вычислениями и увеличивается с ростом числа потоков. Частично данную проблему можно решить работая не с отдельными задачами в очереди, а с блоками. Однако, важно отметить, что увеличивая размер блока, мы уменьшаем качество балансировки. Остальные же стратегии балансировки отказываются от общей очереди задач в пользу раздельной, по одной очереди на каждый поток.

Суть второй стратегия балансировки, work sharing, заключается в том, что каждый поток (или один, специально выделенный) сам осуществляет перераспределение задач из локальных очередей. Так в одной из вариаций этой стратегии [7], это происходит следующим образом. Поток, имея в своей локальной очереди L задач и обращаясь к ней, с вероятностью $\frac{1}{L}$ выполняет действия по балансировке. При этом он случайно выбирает другой поток и, если разница между размерами локальных очередей больше некоторого порога, балансирует очереди путем миграции части задач. В своей работе авторы показывают, что при таких действиях балансировка нагрузки достигается, ожидаемый размер каждой локальной очереди близок к $\frac{N}{p}$, где N - общее число задач в системе, а p - число потоков. К недостаткам данной стратегии можно отнести то, что потоки выполняют действия по балансировке даже тогда, когда она может не требоваться (в локальных очередях всех потоков существуют доступные для выполнения задачи).

Последняя стратегия - work stealing. В ней, когда поток обнаруживает, что его локальная очередь задач пуста, он пытается забрать некоторую задачу у другого потока. В работе [10] теоретически доказано, что данная стратегия обеспечивает производительность близкую к оптимальной для достаточно широкого круга вычислительных задач. На практике она также показывает наилучшие результаты и встречается во многих планировщиках, например в Intel TBB, .NET, Cilk. Производительность work steling'а напрямую зависит от эффективности используемой очереди задач. В качестве такой очереди используют неблокирующую деку, с одного конца которой происходит кража задач другими потоками, а с другого - добавляются и достаются задачи в текущем потоке. Наиболее часто среди реализаций встречается дека на основе циклического массива из работы [9]. Дополнительно стоит отметить, что забирать у другого потока можно не только одну задачу. В работе [2] показано, что в некоторых задачах кража половины задач имеет лучшую производительность.

В дискретно-событийных системах логические процессы обычно взаимодействуют между собой. Представление моделирования в терминах задач вызывает необходимость учета зависимостей между ними. Сделать это эффективно, используя только пул потоков, затруднительно. Учет зависимостей может обеспечить так называемая job system (одной из таких job system является, например, Intel TBB). При исполь-

зовании job system все вычисления, представленные в виде задач, образуют ориентированный граф, ребра которого задают зависимости между задачами. Зависимости устанавливаются в момент описания моделируемой системы в исходном коде. При осуществлении вычислений планировщик job system учитывает их и выполняет те задачи, которые на текущий момент не имеют неразрешенных зависимостей. Часто это происходит при помощи fiber'ов [13].Они позволяют эффективно переключаться между задачами. В процессе работы поток job system, видя неразрешенные зависимости для текущей задачи, откладывает ее и выполняет другую задачу. После разрешения зависимостей отложенная задача может быть возобновлена с прежней инструкции.

В процессе моделирования текущей системы взаимодействующих объектов логические процессы (т.е. объекты) могут динамически создаваться и удаляться. Эффективный учет зависимостей в job system при таких условиях может быть весьма нетривиальным. Поэтому было решено отказаться от представления модели в терминах задач в пользу действий над объектами с динамическим перераспределением их между потоками сходным с work stealing'ом образом.

2.4. Предложенное решение

2.4.1. Синхронизация объектов

При параллельном моделировании каждому объекту ставится в соответствие свое виртуальное время. При очередном обновлении объекта оно увеличивается на заданное значение независимо от остальных объектов. При этом глобальное виртуальное время определяется как минимальное время среди всех объектов на текущий момент.

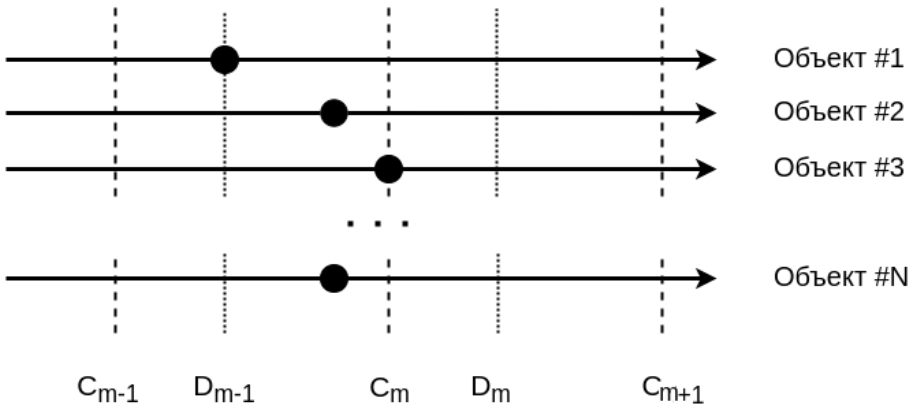

Рис. 1: Типы временных точек (сплошная линия - линия времени объекта, точка на линии - текущее время объекта, штриховая линия - точка обнаружения и разрешения конфликтов C_i, пунктирная линия - точка получения команды на действия по устранению конфликта D_i).

Рассмотрим два типа временных точек [Рис. 1] : C_i - точки обнаружения и разрешения конфликтов, D_i - точки выполнения команды по устранению конфликтов.

Оба типа этих точек являются периодическими. В первых точках мы находим конфликтующие между собой пары объектов и определяем действия, необходимые для их устранения, а во вторых - объекты получают команды на эти действия. Интервал между данными точками - настраиваемый параметр. D_i следуют после C_i с некоторой задержкой, которую можно интерпретировать как время, необходимое на реакцию. При обнаружении конфликта в точках C_i происходит его устранение таким образом, чтобы обеспечить отсутствие конфликтов до точки D_{i+1}. Когда происходит добавление нового объекта, он обязан не конфликтовать ни с каким другим объектом вплоть до ближайшей точки C_i. Таким образом, при вычислении модели гарантируется отсутствие конфликтов между точками C_i и D_{i+1}. Обновления объектов в пределах (C_i, D_{i+1}) могут происходить независимо. Таким образом, граница времени, до которой можно производить обновления параллельно, не нарушая корректное состояние модели, ограничена ближайшей точкой D_i. После достижения последним объектом очередной точки C_i граница может быть передвинута к следующей точке D_{i+1}.

2.4.2. Общая схема вычислений

Общая структура решения представлена на Рис. 2. В решении каждый объект имеет свой дескриптор, который содержит ссылку на него, его текущее виртуальное время и буфер истории с изменениями состояния объекта за последний промежуток времени. Максимальный размер буфера зависит от периода точек C_i и с увеличением его значения возрастает. Имеется мастер, который включает в себя счетчик временной линии, предназначенный для определения глобального виртуального времени, менеджер объектов, позволяющий добавлять или удалять объекты, и несколько работников-потоков, которые непосредственно осуществляют обновления объектов. Каждый работник имеет неблокирующую очередь, которая позволяет получить дескриптор с минимальным в очереди временем, и буфер конфликтов, где расположены идентификаторы конфликтующих пар, обнаруженных данным работником.

Процесс вычислений протекает таким образом. Перед началом моделирования происходит равномерное распределение всех дескрипторов объектов среди работников мастера. Далее каждый работник в цикле выполняет следующие действия [Алг. 1]. Если его очередь (*queue*) не пуста (функция *empty*), он достает из нее дескриптор объекта (*descriptor*), содержащий минимальное в очереди время, и определяет возможность обновления объекта. Для этого его время (*object_ time*) сравнивается с текущей границей, до которой можно произвести обновление. Ее значение возвращает функция *lbts*, которую легко реализовать, используя глобальное виртуальное время, период обнаружения конфликтов и расстояние между точками C_i и D_i. Если обновление возможно, то объект обновляется (функция *update_ object*) пока его время не больше времени следующего в очереди объекта (функция *time_ next_ object*) и

не превышает значение границы. После обновлений дескриптор либо возвращается в очередь (функция *push*), либо, если он имеет максимальное время или удаленный объект (*descriptor_ is_ droped*), убирается из рассмотрения. Если обновить объект нельзя, работник пытается совершить обмен текущего дескриптора на другой. Это происходит в функции *exchange*. В ней работник проходит циклически по всем другим работникам, начиная со своего соседа. При обходе он просматривает объекты с минимальным в каждой очереди временем и обменивается на дескриптор этого объекта, если его можно обновить. В случае успеха, обход прекращается и новый объект помещается в очередь текущего работника. При этом после каждой неудачной попытки проверяется текущий дескриптор, т. к. граница времени к этому моменту уже могла измениться. Если после обхода всех других работников обмен не удался, поток

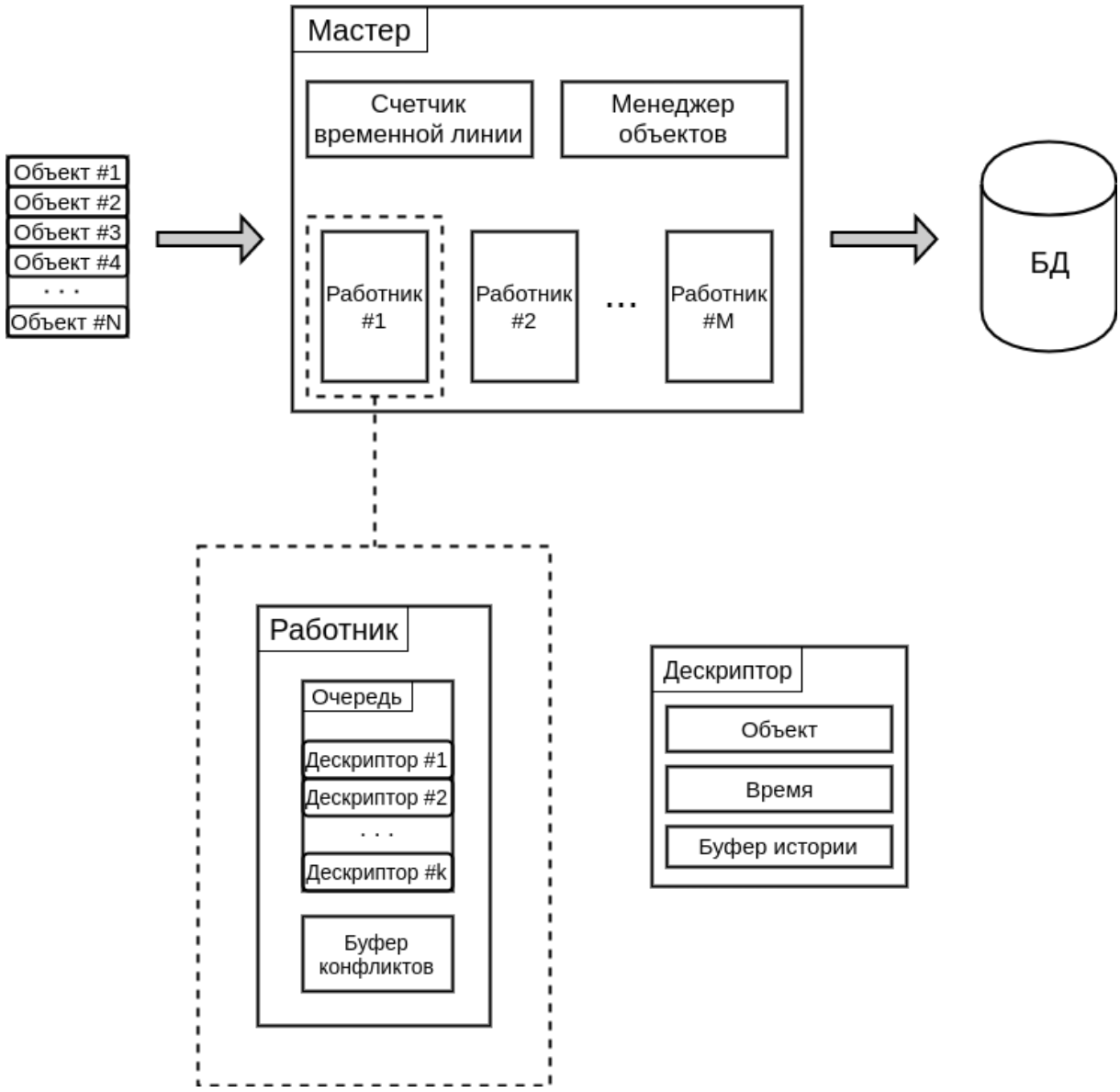

Рис. 2: Общая структура решения.

```
1  while not worker_is_stopped do
2      if not have_work() then
3          wait()
4      end
5      if not queue.empty() then
6          descriptor = queue.pop()
7          if descriptor.object_time < lbts() then
8              descriptor_is_droped = false
9              while descriptor.object_time ≤ time_next_object() and
                descriptor.object_time < lbts() do
10                 descriptor_is_droped = descriptor.update_object()
11                 if descriptor_is_droped then
12                     break
13                 end
14                 global_virtual_time_up()
15             end
16             if not descriptor_is_droped then
17                 queue.push(descriptor)
18             end
19         end
20         else
21             is_success = exchange(descriptor)
22             if not is_success then
23                 yield()
24             end
25         end
26     end
27     else
28         steal()
29     end
30     global_virtual_time_up()
31 end
```

Алгоритм 1: Псевдокод главного цикла работника

работника уступает свой квант времени (функция *yield*). Когда очередь становится пустой, работник забирает половину дескрипторов у другого работника. Это происходит в функции *steal*. Обход работников при этом происходит как при обмене. Когда объектов для обновления не остается (функция *have_work*), поток работника уходит в сон (функция *wait*). Реализовать *have_work* можно, например, через глобальное виртуальное время. Когда оно имеет максимальное значение, работники не имеют больше дескрипторов в очередях. Главный цикл исполняется пока работник не будет остановлен (*worker_is_stopped*).

При обмене, в отличии от краж как в стратегии work stealing, сохраняется баланс по числу объектов у каждого работника, а работа перераспределяется от более за-

груженного работника к менее загруженному равномерно. Важно отметить, почему каждый работник обновляет прежде всего минимальный в своей очереди по времени объект. Делается это по двум причинам:

1. Позволяет снизить количество ненужных спекулятивных вычислений, в случае, когда объект удаляется.

2. Дает лучшее качество балансировки [Рис. 3].

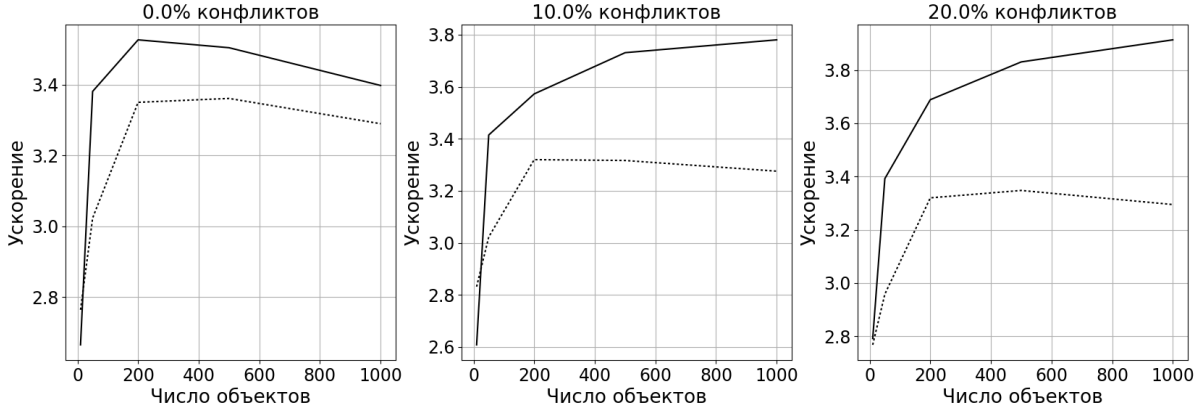

Рис. 3: Сравнение ускорения, получаемого при различных подходах к обновлению объектов, при четырех потоках в (сплошная линия - работником обновляется объект с минимальным в очереди временем (используемый подход), пунктирная линия - обновление каждого объекта выполняется пока его время не достигнет границы, т. е. очередной точки D_i, процент конфликтов вычисляется от максимально возможного числа конфликтующих пар).

В конце каждой итерации главного цикла работника и после очередного обновления объекта вызывается функция *global_ virtual_ time_ up* [Алг. 2]. В ней каждый из работников определяет, можно ли увеличить глобальное виртуальное время (функция *can_ update_ global_ virtual_ time*). Происходит это при помощи счетчика временной линии. Увеличение же глобального времени осуществляется когда нет объектов, у которых время равно текущему глобальному. Если это так, один из работников захватывает (функция *test_ and_ set*) атомарный флаг (*lock*) и инкрементирует глобальное время (функция *inc_ global_ virtual_ time*). Однако, перед этим он производит обновление (функция *update*) менеджера объектов (*object_ manager*) и осуществляет сброс истории изменений объектов за текущее время в базу данных (функция *store_ object_ states*) и, при необходимости, конфликтов (функция *store_ conflicts*). Запись непосредственно в базу данных происходит в специально выделенном потоке. Также в точках C_i (функция *is_ conflict_ point*) этот работник производит обединение буферов конфликтов всех работников (функция *collect_ conflicts*) и разрешение конфликтов (функция *resolution_ conflicts*). Разрешение конфликтов может осуществляться только в одном потоке и способно вызвать простаивание других работников.

```
1  Function global_virtual_time_up()
2      if can_update_global_virtual_time() then
3          if not lock.test_and_set() then
4              if can_update_global_virtual_time() then
5                  object_manager.update()
6                  if is_conflict_point() then
7                      collect_conflicts()
8                      resolution_conflicts()
9                      store_conflicts()
10                 end
11                 store_object_states()
12                 inc_global_virtual_time()
13             end
14             lock.clear()
15         end
16     end
17 end
```

Алгоритм 2: Псевдокод функции global_virtual_time_up

Компенсировать данную проблему призван интервал (C_i, D_i), в пределах которого остальные работники все еще могут обновлять объекты.

2.4.3. Глобальное виртуальное время

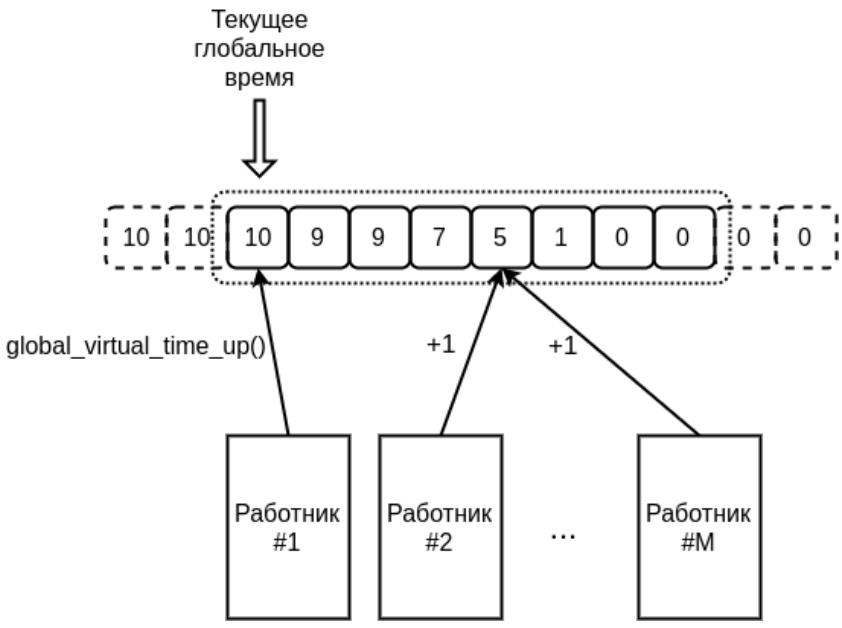

Рис. 4: Вычисление глобального виртуального времени для десяти объектов в модели (пунктирной линией обозначен промежуток времени, который описывается на текущий момент циклическим массивом).

Глобальное виртуальное время в мастере - это атомарная переменная со значением

времени, доступная каждому работнику. Для его вычисления используется простой, но достаточно эффективный алгоритм, основанный на циклическом массиве [Рис. 4]. Каждая ячейка этого массива соответствует определенной точке времени и содержит количество объектов, дошедших до этой точки. Данный массив имеет такой размер, который гарантирует, что текущее время любого объекта не выйдет за временной промежуток, описываемый им. Когда некоторый работник понимает, что ячейка массива, соответствующая текущему глобальному времени, содержит значение, равное общему числу объектов в системе, он зануляет ее, а переменную глобального времени инкрементирует.

Глобальное виртуальное время, точнее его изменение, позволяет определить, когда можно обновить менеджер объектов, произвести разрешение конфликтов, сохранить историю. Все эти операции требуют наличия актуальных состояний всех объектов. Также через глобальное время осуществляется вычисление допустимой границы времени для обновления объектов.

2.4.4. Очередь объектов

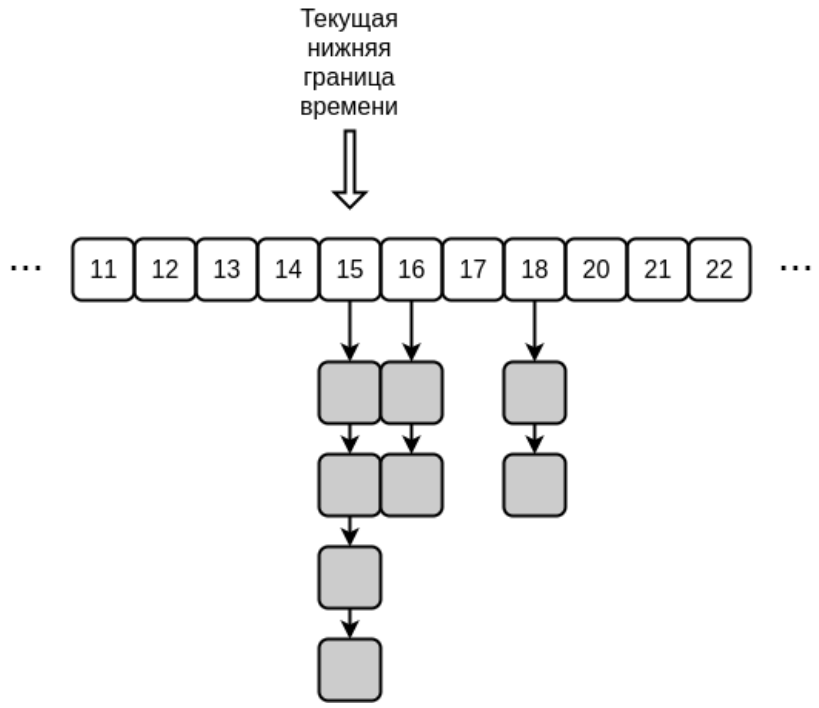

Рис. 5: Очередь объектов на основе массива.

Очередь объектов должна позволять быстро извлекать содержащиеся в ней дескрипторы объектов с минимальным временем. Это можно осуществить при помощи очереди с приоритетами за $O(log(n))$. Однако количество объектов в системе ограничено, как и промежуток возможных значений времени для них (достаточно редко необходимо моделирование системы более чем на час вперед, к тому же допустимо

разбиение на временные интервалы). Поэтому было решено использовать неблокирующую структуру на основе массива, которая позволяет выполнять операции добавления, удаления элементов за $O(1)$ [Рис. 5]. Каждая ячейка этого массива, как и в счетчике временной линии, соответствует точке во времени, и содержит неблокирующий список дескрипторов объектов с соответствующим временем (добавление и удалении элементов осуществляется только в начало). Также у каждого массива есть атомарная переменная, которая содержит значение нижней границы времени (НГВ) объектов в очереди. При добавлении, удалении или обмене дескрипторов она изменяется (инкрементируется) так, чтобы иметь значение времени, которое соответствует объекту в очереди и является минимальным в ней. Инкремент этой переменной может происходить в нескольких потоках, поэтому необходим ряд дополнительных проверок. При использовании одного потока, добавляемые объекты всегда имеют время большее чем НГВ. Однако при нескольких потоках, возможна ситуация, когда при добавлении объекта одним работником, другой работник осуществит инкремент НГВ при обмене. Поэтому после добавления объекта необходимо проверить, что НГВ не больше времени добавленного объекта, и, при необходимости, скорректировать ее. Также после перемещения НГВ вперед, должна быть осуществлена проверка ячейки на его предыдущем значении и, если там будет находится некоторый дескриптор, то старое значение должно быть возвращено обратно. Это нужно из-за возникновения ABA-проблемы [11]. Когда происходит добавление объекта в очередь и граница корректируется, вследствие использования CAS операций, она может быть сразу же передвинута вперед другим потоком. Данные проверки не несут серьезной вычислительной нагрузки, а корректировка границы требуется достаточно редко, поэтому дополнительные операции не сильно уменьшают производительность.

Преобразование массива в циклический затруднена возникновением ABA-проблемы в списке в ячейке массива, когда работник пытается удалить объект из стека, а другой работник успевает сделать это раньше, обновляет объект несколько раз и возвращает его в ту же ячейку из-за цикличности массива. Данную проблему можно решить введением меченных указателей (tagged pointers) [16]. Однако такой подход имеет недостаток. Для его реализации используются свободные в современных процессорах первые шестнадцать бит адреса, которые в будущем могут быть задействованы. Поэтому было решено отказаться от цикличности массива.

2.4.5. Обнаружение конфликтов

Процесс обнаружения конфликтов для модели, используемой в тренажерных продуктах SimLabs, уже реализован и имеет две фазы. На первой фазе просматриваются все пары объектов и при помощи ряда фильтров отсекается часть неконфликтных пар. На второй - потенциально конфликтующие пары подвергаются более тщатель-

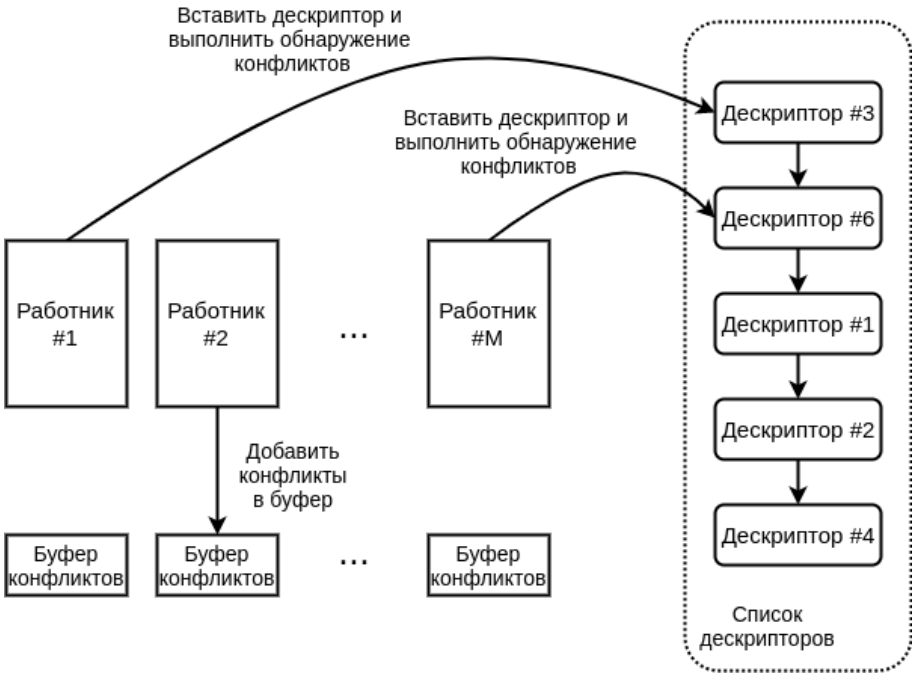

Рис. 6: Процесс обнаружения конфликтов.

ной проверке. В решении данной работы для просмотра всех пар используется общий для всех работников неблокирующий список. По достижении объектом точки времени C_i его дескриптор помещается в список и производится обход, уже имеющихся на текущий момент в нем дескрипторов [Рис. 6]. При этом для добавленного объекта и просматриваемого на текущий момент выполняется первая фаза обнаружения конфликтов и, если отсечения не произошло, - вторая. В случае когда пара объектов конфликтуют, их идентификаторы помещаются в буфер конфликтов данного работника. После содержимое данных буферов объединяется. Таким образом, вычисление конфликтов происходит параллельно.

Также рассматривалась более сложная схема вычисления второй фазы обнаружения конфликтов, когда потенциально конфликтующие пары добавлялись в буфер текущего работника, а уже после производились вычисления с применением work stealing стратегии. Однако, реализация такой схемы оказалась менее производительной, чем выше рассмотренная [Рис. 7].

2.4.6. Динамическое добавление и удаление объектов

После каждого обновления всех объектов следует обновление менеджера объектов, который может добавить новые объекты или же удалить уже существующие. При этом он уведомляет мастера об изменениях в составе объектов. При добавлении нового объекта для него создается дескриптор с соответствующим временем и он добавляется к наименее нагруженному работнику, т. е. к тому, у которого меньше объектов в очереди. Удаление же происходит несколько сложнее, т. к. используют-

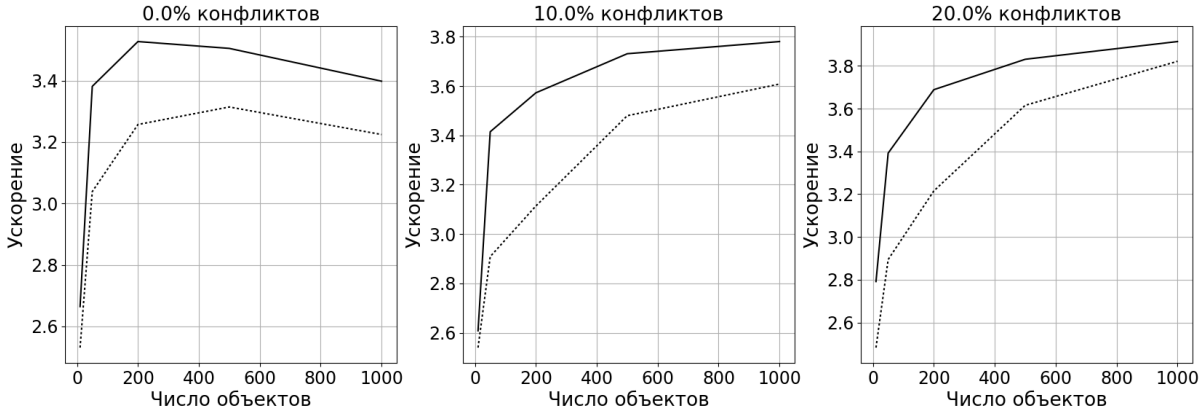

Рис. 7: Сравнение ускорения, получаемого при использовании различных подходов к обработке конфликтов, при четырех потоках (сплошная линия - текущий подход, пунктирная линия - подход с применением work stealing стратегии обработки потенциально конфликтующих пар, процент конфликтов вычисляется от максимально возможного числа конфликтующих пар).

ся спекулятивные вычисления. При удалении дескрипторы объектов помечаются как удаленные, а их идентификаторы запоминаются. Когда некоторый работник понимает, что его текущий дескриптор удален, он редактирует счетчик временной линии в обратном относительно хода времени направлении, отнимая от текущих значений единицу, и, останавливаясь поравнявшись со значением глобального виртуального времени, уменьшает общее число объектов в системе. Далее этот дескриптор убирается из рассмотрения. Физическое удаление такого дескриптора возможно только после прохождения всеми объектами ближайшей точки C_i, т. к. он может находится в списке конфликтов. При этом при обнаружении конфликтов и сбросе истории в базу данных, удаленные объекты фильтруются, для этого ранее были сохранены их идентификаторы.

2.4.7. Режимы работы тренажерной системы

Рассмотрим теперь возможность применения описанного выше решения к различным режимам работы тренажерной системы:

1. В случае воспроизведения сценария, когда нет необходимости иметь сразу все состояния объектов за временной промежуток моделирования, можно производить вычисления наперед интервалами. При этом, если пользователь вносит какие-то изменения в модель, дожидаемся окончания расчета текущего интервала и запускаем частичный перерасчет изменений интервала.

2. При перемотке вперед общую схему можно использовать без изменений.

3. Для сохранения текущей истории необходимо частично перерасчитать модель. Для этого определяются объекты которым на текущий момент требуются изме-

нения, т. е. те, которые имеют ошибочную историю состояний, и далее вычисления производятся также как и в общей схеме. Однако теперь менеджер объектов выполняет другую функцию. Он детектирует, в каких объектах из-за текущих вычислений будут происходить изменения, восстанавливает необходимые состояния и добавляет их к уже имеющимся в вычислениях объектам. Старая история при этом, начиная с некоторого момента, чистится и в ходе расчета дополняется новыми изменениями.

3. Тестирование и анализ результатов

3.1. Методология тестирования

Для тестирования предложенного решения был написан прототип с упрощенной моделью объектов и симуляцией операций обновления объекта, определения конфликта между парой объектов и разрешения всех конфликтов. Время выполнения каждой из этих операций корректировалось при помощи фиктивных инструкций. Количество таких инструкций при каждом вызове - случайная величина, распределенная по нормальному закону, которая определяет продолжительность выполнения операции.

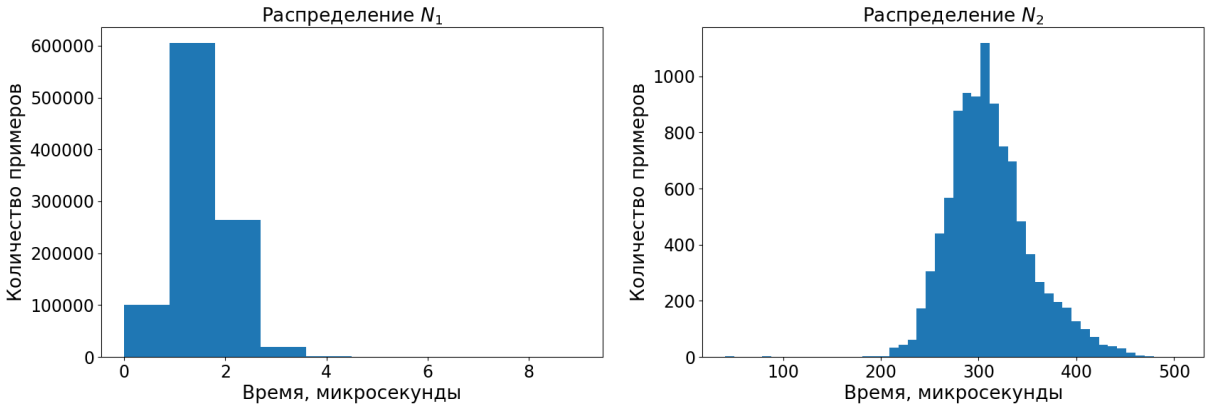

Рис. 8: Гистограммы распределений времени, используемые при тестировании.

При тестировании время операций формировалось при помощи распределений N_1 и N_2 [Рис. 8] следующим образом:

1. Время обновления объекта с вероятностью 0.99 бралось из распределения N_1 и с вероятностью 0.01 - из N_2, т. е. имело выброс.

2. Время определения конфликта между парой объектов бралось полностью из распределения N_1.

3. Время разрешения всех конфликтов предполагалось зависимым от числа объектов и рассчитывалось как случайная величина из N_1, умноженная на процент от числа объектов.

Распределения определялись исходя из оценки времени выполнения операции обновления, которое было получено в результате профилирования. Обновление занимало в среднем 4 микросекунды и имело редкие выбросы, превосходящие среднее значение максимум в 100 раз. Также было замечено, что операция определения конфликта для пары занимает аналогичное время.

При анализе производительности использовались следующие параметры:

1. Число потоков - 4, 8, 12 потоков.

2. Число объектов в системе - 10, 50, 200, 500, 1000 объектов.

3. Период обнаружения и разрешения конфликтов - 20, 35, 50 временных шагов.

4. Интервал между точками обнаружения и разрешения конфликтов C_i и следующими за ними точками выполнения команды по устранению конфликтов D_i - 0, 5, 10 временных шагов.

5. Число конфликтных пар как доля от максимально возможного числа конфликтующих пар - 0, 0.1, 0.2.

6. Доля от числа объектов для операции разрешения конфликтов - 0, 0.1, 0.2.

Границы параметров выбраны из соображений возможности их использования в тренажерной системе.

Тестирование происходило на машине с процессором Intel Xeon E5-2695 v2 (12 ядер, 24 потока) следующим образом. Выполнялось моделирование на 5000 временных шагов 50 раз для каждой конфигурации представленных ранее параметров как через однопоточную реализацию, так и через представленное решение. Далее исследовалось ускорение, вычисляемое через отношение медианы времени моделирования через однопоточную реализацию к медиане времени моделирования через предложенное решение.

3.2. Анализ результатов

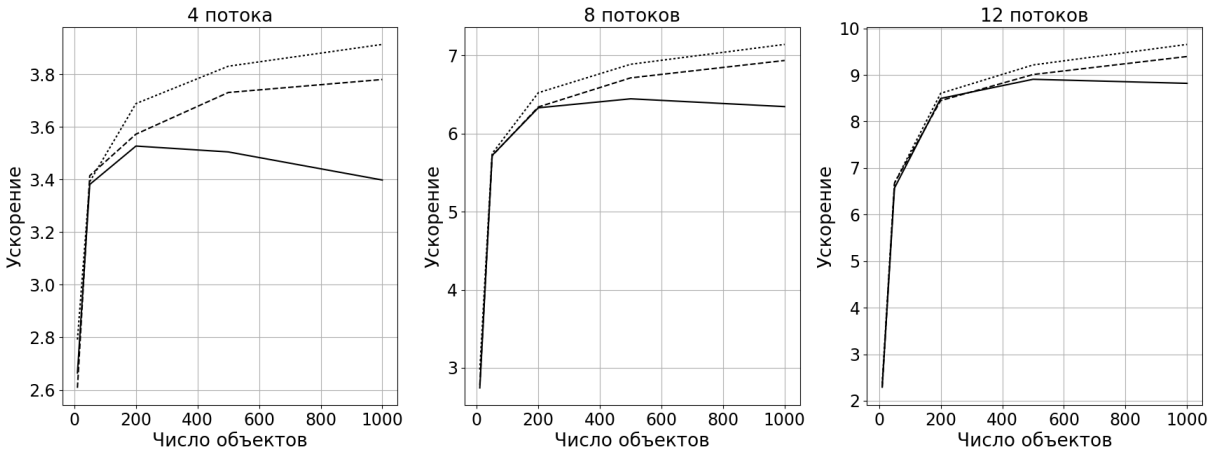

Рис. 9: Графики ускорения при оптимальных параметрах (сплошная линия - 0% конфликтов, штриховая линия - 10% конфликтов, пунктирная линия - 20% конфликтов).

При анализе влияния параметров были выявлены следующие особенности предложенного решения:

1. Увеличение времени выполнения операции по разрешению конфликтов не вызывает сильного снижения производительности и с увеличением числа конфликтов влияние ощущается слабее (наблюдалось снижение ускорения на 0.1 - 0.25).

2. Увеличение интервала между точками времени C_i и D_i положительно влияет на производительность и особенно заметно при небольших интервалах между точками времени C_i (наблюдалось увеличение ускорения на 0.1 - 1). С ростом числа объектов эффективность данного интервала уменьшается.

3. Увеличение числа конфликтов дает прирост к ускорению. Особенно это заметно при большом количестве объектов (наблюдалось увеличение ускорения на 0.1 - 1).

4. Увеличение интервала между точками времени C_i положительно влияет на производительность. Сильнее всего это проявляется когда интервал между точками времени C_i и D_i мал, а также при росте числа потоков и количества объектов (наблюдалось увеличение ускорения на 0.2 - 1).

Графики ускорения при оптимальных параметрах решения представлены на Рис. 9 (интервал между точками C_i - 50 временных шагов, интервал между точками C_i и D_i - 10 временных шагов, доля от числа объектов для операции разрешения конфликтов - 0.2).

4. Заключение

В результате проделанной работы было разработано технологическое решение, которое способно эффективно параллельно моделировать на однопроцессорной машине систему объектов, схожую с используемой в тренажерных продуктах SimLabs. При этом было достигнуто приемлемое ускорение, а также предусмотрена возможность работы решения в различных режимах вычислений. В ходе работы были решены следующие задачи:

1. Проанализированы основные идеи и подходы к параллельному моделированию дискретно-событийных систем, а также способы балансировки нагрузки между потоками
2. Разработана схема параллельного моделирования
3. Реализован прототип и проанализирована его производительность

В настоящий момент проводятся работы по переносу решения в тренажерные продукты SimLabs. Встраивание решения в тренажерную систему осложняется тем, что исходная реализация моделирования не предполагает использование многопоточности. Поэтому ее добавление требует решения множество дополнительных задач.

Список литературы

[1] D. A. Reed A. D. Malony, McCredie B. D. Parallel discrete event simulation using shared memory. — IEEE Transactions on Software Engineering, 1988. — P. 541–553.

[2] D. Hendler N. Shavit. Non-Blocking Steal-Half Work Queues. — 2002.

[3] Fujimoto R. M. Lookahead in parallel discrete event simulation. — Proceedings of the 1988 International Conference on Parallel Processing, 1988. — P. 34–41.

[4] Fujimoto R. M. Performance measurements of distributed simulation strategies. — Transactions of the Society for Computer Simulation, 1989. — P. 89–132.

[5] Fujimoto R. M. Parallel and Distributed Simulation Systems. — JOHN WILEY SONS, INC., 2000. — P. 66–70.

[6] Fujimoto R. M. Parallel and Distributed Simulation Systems. — JOHN WILEY SONS, INC., 2000. — P. 97–136.

[7] L. Rudolph M. Slivkin-Allalouf E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. — 1991.

[8] Liu J. Parallel Discrete-Event Simulation. — 2009. — P. 6–7.

[9] Nhat Minh Lê Antoniu Pop Albert Cohen Francesco Zappa Nardelli. Correct and Efficient Work-Stealing for Weak Memory Models. — 2013.

[10] R. D. Blumofe C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing.

[11] Wikipedia. ABA problem // Википедия, свободная энциклопедия. — 2017. — URL: `https://en.wikipedia.org/wiki/ABA_problem` (online; accessed: 28.05.2017).

[12] Wikipedia. Discrete event simulation // Википедия, свободная энциклопедия. — 2017. — URL: `https://en.wikipedia.org/wiki/Discrete_event_simulation` (online; accessed: 28.05.2017).

[13] Wikipedia. Fiber // Википедия, свободная энциклопедия. — 2017. — URL: `https://en.wikipedia.org/wiki/Fiber_(computer_science)` (online; accessed: 28.05.2017).

[14] Wikipedia. Separation (aeronautics) // Википедия, свободная энциклопедия. — 2017. — URL: `https://en.wikipedia.org/wiki/Separation_(aeronautics)` (online; accessed: 28.05.2017).

[15] Wikipedia. Speculative execution // Википедия, свободная энциклопедия. — 2017. — URL: https://en.wikipedia.org/wiki/Speculative_execution (online; accessed: 28.05.2017).

[16] Wikipedia. Tagged pointer // Википедия, свободная энциклопедия. — 2017. — URL: https://en.wikipedia.org/wiki/Tagged_pointer (online; accessed: 28.05.2017).